Shortest Paths and Multicommodity Network Flows

A Thesis
Presented to
The Academic Faculty

by

I-Lin Wang

In Partial Fulfillment
of the Requirements for the Degree
Doctor of Philosophy

School of Industrial and Systems Engineering
Georgia Institute of Technology
April 2003

Shortest Paths and Multicommodity Network Flows

Approved by:

Dr. Ellis L. Johnson, Committee Chair	Dr. Pamela H. Vance
Dr. George L. Nemhauser	Dr. Huei-Chuen Huang
Dr. Joel S. Sokol	

Date Approved

To my Mom, wife, and son

for their unwavering love and support

ACKNOWLEDGEMENTS

First I want to thank Dr. Ellis Johnson for being my advisor and forseeing my potential in my dissertation topics that he suggested in 2000. In the beginning, I was a little reluctant and lacked confidence, since the topics were considered to be difficult to have new breakthrough. My progress turned out to be successful, because of Dr. Johnson's brilliant insight and continuous guidance over the years.

I would also like to thank Dr. Joel Sokol. Dr. Sokol was my former classmate at MIT, and was also on my committee. Although at MIT we didn't have a chance to chat too often, we do have frequent interaction at ISyE. He is very patient and has been most helpful in modifying my writings. He often suggests very clever advice which saves me time and inspires new ideas. I also acknowledge Dr. George Nemhauser, Dr. Pamela Vance, and Dr. Huei-Chuen Huang for having served on my committee. With their advice and supervision, my thesis became more complete and solid.

Special thanks to Dr. Gary Parker. Dr. Parker helped me find financial aid on my first day at GT. Also, he helped me to keep a 1/3 TA job when I took the comprehensive exam. I also thank Dr. Amy Pritchett for kindly supporting me in my first quarter. Without their support and Dr. Johnson's financial aid, I could not have earned my degree so easily. Thanks to Jennifer Harris for her adminstrative help over the years.

I thank Fujitsu Research Lab for giving me a foreign research position after MIT. Special thanks to Chujo san, Miyazaki san, Minoura san and Ogura san. Also I thank Yasuko & Satoru san, and my MIT senpai, Ding-Kuo and Hwa-Ping very much for the good times in Japan over the years.

Before coming to GT, I had an unhappy year at Stanford University. I have to thank Kien-Ming, Chih-Hung, and Li-Hsin for helping me out in that difficult time. Also thanks to Sheng-Pen, Tien-Huei, San-San, Che-Lin and Wei-Pang.

In my four and half years at ISyE, I have been helped by many good friends and classmates. Among them, I especially would like to thank Jean-Philippe. He is usually the first one I consulted with whenever I encountered research problems. I am in debt to Ricardo, Balaji, Dieter, Lisa, Brady, Andreea, Woojin, and Devang for their help. I also appreciated the help from Ivy Mok, Chun How Leung, Yen Soon Lee, and Dr. Loo Hay Lee in Singapore.

I am thankful for the many good people from Taiwan that helped me in Atlanta over the years, particulary Wei-Pin, Po-Long, Jenn-Fong, Chien-Tai & Chen-Chen, Tony & Wendy, James & Tina, Jack & Miawshian, Chih-Heng, Ivy & George, Chia-Chieh, Wen-Chih, Yen-Tai & Vicky, Po-Hsun and Cheng-Huang. I had a very great time with them.

I have to thank my Mom for her endless love and encouragement. Many thanks to my Mother and Father-in-law for taking care of my wife and son these years when I was away. Finally I want to thank my wife, Hsin-Yi, for supporting me to complete my degree and giving me a lovely son. Everytime when I would feel lonely or bored, I would watch our photos/videos, they always cheered me up and made me energetic again. I could not have done it without her unwavering love.

It has been 10 years, and I am always grateful for my Father in Heaven, for raising and educating me, which makes all of this possible.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

SUMMARY

The shortest path problem is a classic and important combinatorial optimization problems. It often appears as a subproblem when solving difficult combinatorial problems like multicommodity network flow (MCNF) problems.

Most shortest path algorithms in the literature are either to compute the 1-ALL single source shortest path (SSSP) tree, or to compute the ALL-ALL all pairs shortest paths (APSP). However, most real world applications require only multiple pairs shortest paths (MPSP), where the shortest paths and distances between only some specific pairs of origin-destination nodes in a network are desired. The traditional single source shortest path (SSSP) and all pairs shortest paths (APSP) algorithms may do unnecessary computations to solve the MPSP problem.

We survey and summarize many shortest path algorithms, and discuss their pros and cons. We also investigate the Least Squares Primal-Dual method, a new LP algorithm that avoids degenerate pivots in each primal-dual iteration, for solving 1-1 and 1-ALL shortest path problems with nonnegative arc lengths, show its equivalence to the classic Dijkstra's algorithm, and compare it with the original primal-dual method.

We propose two new shortest path algorithms to save computational work when solving the MPSP problem. Our MPSP algorithms are especially suitable for applications with fixed network topology but changeable arc lengths. We discuss the theoretical details and complexity analyses. We test several implementations of our new MPSP algorithms extensively and compare them with many state-of-the-art SSSP algorithms for solving many families of artificially generated networks and a real Asia-Pacific flight network.

Our MPSP computational experiments show that there exists no "killer" shortest path algorithm. Our algorithms have better performance for dense networks, but become worse for larger networks. Although they do not have best performance for the artificially generated graphs, they seem to be competitive for the real Aisa-Pacific flight network.

We provide an extensive survey on both the applications and solution methods for MCNF problems in this thesis. Among those methods, we investigate the combination of the primal-dual algorithm with the key path decomposition method. In particular, to efficiently solve the restricted primal problem (RPP) in each primal-dual iteration, we relax the nonnegativity constraints for some set of basic variables, which makes the relaxed RPP smaller and easier to solve since the convexity constraint will be implicitly maintained.

We implement our new primal-dual key path method (KEY), propose new methods to identify max step length, and suggest perturbation methods to avoid degenerate pivots and indefinite cycling problems caused by primal and dual degeneracy. We perform limited computational experiments to compare the running time of the generic primal-dual (PD) method, the Dantzig-Wolfe (DW) decomposition method, and the CPLEX LP solver that solves the node-arc form (NA) of the MCNF problems, with our method KEY. Observations from the computational experiments suggest directions for better DW implementation and possible remedies for improving PD and KEY.

CHAPTER I

INTRODUCTION TO MULTICOMMODITY NETWORK FLOW PROBLEMS

In this chapter, we first describe an air cargo flow problem in the Asia Pacific region. It is this problem that motivates our research. Because the problem can be modeled as a *multicommodity network flow* (MCNF) problem, we then introduce many MCNF related research topics and applications. Finally, we review different MCNF formulations and outline this thesis.

1.1 Asia Pacific air cargo system

Air cargo is defined as anything carried by aircraft other than mail, persons, and personal baggage. When an enterprise has facilities in its supply chain located in several different regions, efficient and reliable shipping via air is a crucial factor in its competitiveness and success. For example, many international electronics companies currently practice just-in-time manufacturing in which components are manufactured in China, Malaysia, Indonesia, Vietnam, or the Philippines, assembled in Taiwan, Japan or Singapore, and shipped to Europe and America.

The Asia Pacific region is expected to have the largest airfreight traffic in the next decade [177] due to the growth of Asia Pacific economies. In particular, China has drawn an increasing amount of investment from all over the world due to its cheap manpower and land. Many enterprises have moved their low-end manufacturing centers to China. Most China-made semifinished products are shipped to facilities in other Asian countries for key part assembly, and then sent out worldwide for sale.

The increasing need for faster air transportation drives the construction of larger and more efficient air facilities in the Asia Pacific region. New airports such as Chek Lap Kok airport (Hong Kong), Kansai airport (Osaka, Japan), Inchon airport (Seoul, Korea),

Table 1: Division of Asia Pacific into sub regions

Region	Countries in the Region
Central Asia	Kazakhstan, Kyrgyzstan, Tajikistan, Turkmenistan, Uzbekistan
South Asia	Afghanistan, Bangladesh, Bhutan, India, Maldives, Nepal, Pakistan, Sri Lanka
Northeast Asia	China, Hong Kong, Japan, Korea (Democratic People's Republic), Korea (Republic of), Macau, Mongolia, Russian Federation (East of Urals), Taiwan
Southeast Asia	Brunei Darussalam, Cambodia, Indonesia, Lao (Peoples' Democratic Republic), Malaysia, Myanmar, Philippines, Singapore, Thailand, Vietnam
South Pacific	America Samoa, Australia, Christmas Island, Cocos (Keeling) Islands, Cook Islands, Fiji, French Polynesia, Guam, Kiribati, Marshall Islands, Micronesia, Nauru, New Caledonia, New Zealand, Niue, Norfolk Island, Northern Mariana Islands, Palau, Papua New Guinea, Pitcairn, Samoa, Solomon Islands, Tokelau, Tonga, Tuvalu, US Minor Outlying Islands, Vanuatu, Wallis and Futuna Islands

Bangkok airport (Thailand), and Pudong airport (Shanghai, Chian) have been constructed. Other airports such as Changi airport (Singapore), Narita airport (Tokyo, Japan), and Chiang-Kai-Shek airport (Taipei, Taiwan) have built new terminals or runways to be able to handle increased air traffic. A more thorough introduction to the air cargo system in the Asia Pacific region can be found in Bazaraa et al. [39].

This thesis is motivated by a study of the Asia Pacific air cargo system. We will describe an Asia Pacific flight network, and give a mathematical model for the air cargo system which corresponds to a multicommodity network flow (MCNF) model.

1.1.1 Asia Pacific flight network

The Asia Pacific region can be divided into five sub-regions as shown in Table 1.

Our research focuses on Northeast and Southeast Asia. Our Asia Pacific flight network (AP-NET) contains two types of nodes: center nodes that represent major cities (or airports) in the Northeast and Southeast Asia regions, and rim nodes that represent major cities in other regions. Arcs represent flight legs between center cities and center/rim cities. Based on data from the Freight Forecast [176], Air Cargo Annual [174], and Asia Pacific Air Transport Forecast [175] published by IATA, we selected 11 center countries (see Table 2) and 20 rim countries (see Table 3). The center countries are the Northeast and Southeast Asian countries that contribute the most intra and inter-regional air cargo traffic. The rim countries are those countries outside the region that have large air cargo traffic with the center countries.

Table 2: Center countries and cities of the AP filght network

center country	center city name
China	Beijing, Dalian, Guangzhou, Kunming, Qingdao, Shanghai, Shenyang, Shenzhen, Tianjin, Xi An, Xiamen
Hong Kong	Hong Kong
Indonesia	Denpasar Bali, Jakarta, Surabaya
Japan	Fukuoka, Hiroshima, Kagoshima, Komatsu, Nagoya, Niigata, Okayama, Okinawa, Osaka, Sapporo, Sendai, Tokyo
Korea	Cheju, Pusan, Seoul
Malaysia	Kota Kinabalu, Kuala Lumpur, Kuching, Langkawi, Penang
Philippines	Cebu, Manila, Subic Bay
Singapore	Singapore
Taiwan	Kaohsiung, Taipei
Thailand	Bangkok, Chiang Mai, Hat Yai, Phuket
Viet Nam	Hanoi, Ho Chi Minh City

Table 3: Rim countries and cities of the AP filght network

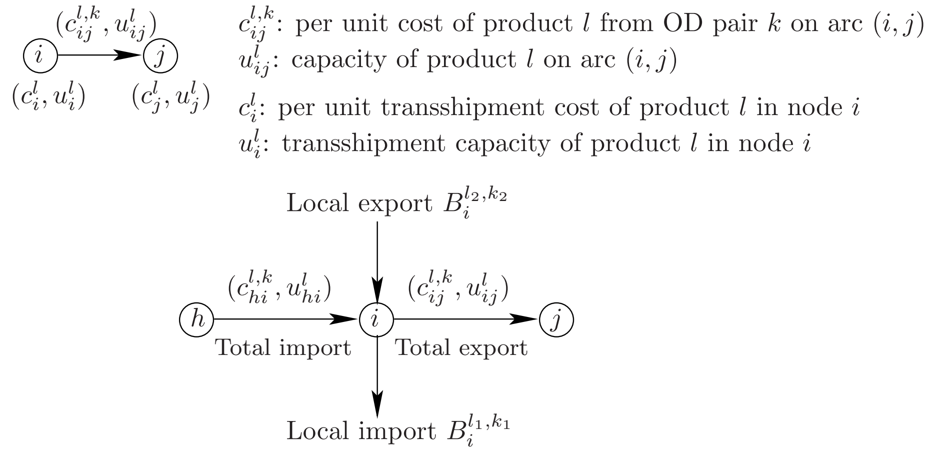
rim country	rim city name
Australia	Adelaide, Avalon, Brisbane, Cairns, Darwin, Melbourne, Perth, Sydney
Bahrain	Bahrain
Canada	Toronto, Vancouver
Denmark	Copenhagen
Finland	Helsinki
France	Paris
Germany	Frankfurt, Munich
Greece	Athens
India	Bangalore, Calcutta, Chennai, Delhi, Hyderabad, Mumbai
Italy	Milan, Rome
Netherlands	Amsterdam
New Zealand	Auckland, Christchurch
Pakistan	Islamabad, Karachi, Lahore
Russian Federation	Khabarovsk, Moscow, Novosibirsk, St Petersburg, Ulyanovsk
Sri Lanka	Colombo
Switzerland	Zurich
Turkey	Istanbul
United Arab Emirates	Abu Dhabi, Dubai, Sharjah
United Kingdom	London, Manchester
USA	Anchorage, Atlanta, Boston, Chicago, Dallas/Fort Worth, Detroit, Fairbanks, Honolulu, Houston, Kona, Las Vegas, Los Angeles, Memphis, Minneapolis/St Paul, New York, Oakland, Portland, San Francisco, Seattle, Washington

For our case study, we use the OAG MAX software package, which provides annual worldwide flight schedules for more than 850 passenger and cargo airlines, to build a monthly flight schedule of all jet non-stop operational flights between 48 center cities and 64 rim cities in September 2000. AP-NET contains 112 nodes and 1038 arcs, where each node represents a chosen city and each arc represents a chosen flight. Among the 1038 arcs, 480 arcs connect center cities to center cities, 277 arcs connect rim cities to center cities, and 281 arcs connect center cities to rim cities. We do not include the flights between rim cities in this model since we are only interested in the intra and inter-regional air cargo flows for the center nodes.

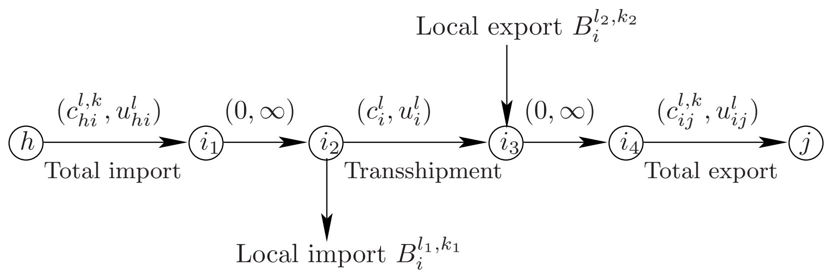
1.1.2 An air cargo mathematical model

The air cargo flow problem AP-NET is, in fact, a multicommodity network flow problem. In particular, given a monthly air cargo demand/supply data for both the center and rim nodes that have to be shipped via the flight legs (arcs) in AP-NET, then each commodity is a specific product for a specific origin-destination (OD) pair. Let $c_{i,j}^{l,k}$ denote the unit flow cost of shipping product l from origin s_k to destination t_k (i.e. OD pair k) on flight leg (i,j), and u_{ij}^{l} denote the capacity of shipping product l on flight leg (i,j). Each airport i has unit transshipment cost c_i^l and transshipment capacity u_i^l for product l (see Figure 1 (a)).

To eliminate the node cost and capacity, we can perform a node splitting procedure as in Figure 1 (b). In particular, suppose node i has to receive $B_i^{l_1,k_1}$ units of import cargo demand l_1 from node s_{k_1}, and send $B_i^{l_2,k_2}$ units of export cargo supply l_2 to node t_{k_2}. We can split node i into four nodes: an import node i_1 for receiving the import cargo, a demand node i_2 for receiving $B_{i_2}^{l_1,k_1}$ units of local import cargo, a supply node i_3 for sending $B_{i_3}^{l_2,k_2}$ units of local export cargo, and an export node i_4 for sending the export cargo. Suppose the original network has $|N|$ nodes and $|A|$ arcs. Then, the new network will have $4\,|N|$ nodes and $|A| + 3\,|N|$ arcs. The transshipment cost and capacity of node i become the cost and capacity of arc (i_2, i_3). Both arc (i_1, i_2) and arc (i_3, i_4) have zero costs and unbounded capacities..



(a) Original network for product l



(b) Transformed network for product l

Figure 1: Network transformation to remove node cost and capacity

In this way, the minimum cost cargo routing problem becomes a minimum cost multicommodity network flow problem. It seeks optimal air cargo routings to minimize the total shipping cost of satisfying the OD demands while not exceeding the airport or flight capacities. Each commodity is defined as a specific product to be shipped from an origin to a destination.

If we have all of the parameters (OD demand data for each node, unit shipping and transshipment costs for each commodity on each flight leg and airport, and the capacity of each flight leg and airport), this MCNF problem can be solved to optimality. The sensitivity analysis can give us insight and suggest ways to improve the current Asia Pacific air cargo system. For example, by perturbing the capacity for a specific flight leg or airport, we can determine whether adding more flights or enlarging an airport cargo terminal is profitable, which flight legs are more important, and whether current practices for certain airline or airport are efficient. Similar analyses can be done to determine whether adding a new origin-destination itinerary is worthwhile, or whether increasing the charge on a certain OD itinerary is profitable.

However, it is difficult to determine those parameters. The flight capacity may be estimated by adding the freight volumes of each aircraft on a certain flight leg. The unit shipping and transshipping cost or revenue is difficult to determine, since it is a function of several variables such as time, distance, product type and OD itinerary (including the flight legs and airports passed). The transshipment capacity for a cargo terminal is also difficult to estimate since it is variable and dynamic. We may first treat it as unbounded and use the optimal transshipment flow to judge the utility of cargo terminals. Correctly estimating these parameters is itself a challenging research topic. This thesis, on the other hand, focuses on how to solve the MCNF problem more efficiently, given all these parameters.

1.2 MCNF problem description

The multicommodity network flow problem is defined over a network where more than one commodity needs to be shipped from specific origin nodes to destination nodes while not violating the capacity constraints associated with the arcs. It extends the *single commodity network flow* (SCNF) problem in a sense that if we disregard the *bundle constraints*, the arc capacity constraints that tie together flows of different commodities passing through the same arc, a MCNF problem can be viewed as several independent SCNF problems.

In general, there are three major MCNF problems in the literature: the *max MCNF* problem, the *max-concurrent flow* problem, and the *min-cost MCNF* problem. The max MCNF problem is to maximize the sum of flows for all commodities between their respective origins and destinations. The max-concurrent flow problem is a special variant of the max MCNF problem which maximizes the fraction (throughput) of the satisfied demands for all commodities. In other words, the max concurrent flow problem maximizes the fraction z for which the min-cost the MCNF problem is feasible if all the demands are multiplied by z. The min-cost MCNF problem is to find the flow assignment satisfying the demands of all commodities with minimum cost without violating the capacity constraints on all arcs.

$|N| = 5$, $|A| = 10$, $|K| = 3$

K^1: $(s^1, t^1) = (1, 4)$, $B^1 = 6$
K^2: $(s^2, t^2) = (5, 3)$, $B^2 = 4$
K^3: $(s^3, t^3) = (3, 1)$, $B^3 = 8$

origins, destinations, and demands
of three commodities

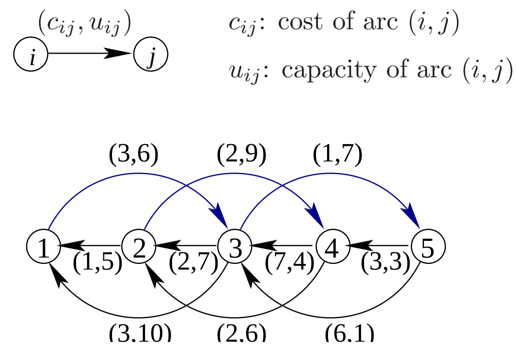


Figure 2: A MCNF example problem

This dissertation will focus on min-cost MCNF problem. Hereafter, when we refer to the MCNF problem, we mean the min-cost MCNF problem.

The existence of the bundle constraints makes MCNF problems much more difficult than SCNF problems. For example, many SCNF algorithms exploit the single commodity flow property in which flows in opposition direction on an arc could be canceled out. In MCNF problems, flows do not cancel if they are different commodities. The max-flow min-cut theorem of Ford and Fulkerson [116] in SCNF problems guarantees that the maximum flow is equal to the minimum cut. Furthermore, with integral arc capacities and node demands, the maximum flow is guaranteed to be integral. None of these properties can be extended to MCNF, except for several special planar graphs [255, 254, 224, 250], or two-commodity flows with even integral demands and capacities [170, 276, 278]. The *total unimodularity* of the constraint matrix in the SCNF LP formulation guarantees integral optimal flows in the cases of integral node supplies/demands and arc capacities. This integrality property also can not be extended to MCNF LP formulations. Solving the general integer MCNF problem is NP-complete [194]. In fact, even solving the two-commodity integral flow problem is NP-complete [107].

Since 1960s, MCNF has motivated many research topics. For example, *column generation* by Ford and Fulkerson [115] was originally designed to solve max MCNF problems, and is still a common technique for solving large-scale LP problems. Seymour [286] proposes the *max-flow min-cut matroid* and several important matroid theorems by studying multicommodity flow. The nonintegrality property of MCNF spurs much research in integral

$$\widetilde{N} = \begin{bmatrix} 1 & -1 & 0 & -1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 1 & 0 & -1 & 0 & -1 & 0 & 0 & 0 \\ -1 & 0 & 0 & 1 & 1 & 1 & 0 & -1 & -1 & 0 \\ 0 & 0 & -1 & 0 & 0 & 0 & 1 & 1 & 0 & -1 \\ 0 & 0 & 0 & 0 & 0 & -1 & 0 & 0 & 1 & 1 \end{bmatrix}$$

Node-arc incidence matrix $\widetilde{N}$

$$\begin{bmatrix} \widetilde{N} & & \\ & \widetilde{N} & \\ & & \widetilde{N} \\ I & I & I \end{bmatrix} X \begin{array}{l} = b_1 = [\ 6\ 0\ 0\ \text{-}6\ 0\ ]^T \\ = b_2 = [\ 0\ 0\ \text{-}4\ 0\ 4\ ]^T \\ = b_3 = [\ \text{-}8\ 0\ 8\ 0\ 0\ ]^T \\ \leq u = [\ 6\ 5\ 9\ 10\ 7\ 7\ 6\ 4\ 1\ 3\ ]^T \end{array}$$

$$X = [\ x^1\ x^2\ x^3\ ]^T\ ,\ x^k \in R_+^{|A|}\ \forall k \in K$$

MCNF constraints

Figure 3: Block angular structure of the example problem in Figure 2

LP related to matroids, algorithmic complexity, and polyhedral combinatorics (see [162] for details). The block-angular constraint structure (see Figure 3) of the MCNF constraints serves as a best practice for decomposition techniques such as *Dantzig-Wolfe decomposition* [88] and *Benders decomposition* [42], and basis partitioning methods such as *generalized upper bounding* [213, 167] and *key variables* [273].

1.3 Applications

MCNF models arise in many real-world applications. Most of them are network routing and network design problems.

1.3.1 Network routing

1.3.1.1 Message routing in Telecommunication:

Consider each requested OD pair to be a commodity. The problem is to find a min-cost flow routing for all demands of requested OD pairs while satisfying the arc capacities. This appears often in telecommunication. For example, message routing for many OD pairs [36, 237], packet routing on virtual circuit data network [221] (all of the packets in a session are transmitted over exactly one path between the OD to minimize the average number of packets in the network), or routing on a ring network [288] (any cycle is of length n), are

MCNF problems.

1.3.1.2 Scheduling and routing in Logistics and Transportation:

In logistics or transportation problems, commodities may be objects such as products, cargo, or even personnel. The commodity scheduling and routing problem is often modeled as a MCNF problem on a time-space network where a commodity may be a tanker [41], aircraft [166], crew [60], rail freight [16, 78, 210], or Less-than Truck-Load (LTL) shipment [109, 38].

Golden [144] gives a MCNF model for port planning that seeks optimal simultaneous routing where commodities are export/import cargo, nodes are foreign ports (as origins), domestic hinterlands (as destinations) and US ports (as transshipment nodes), and arcs are possible routes for cargo traffic. Similar problems appear in grain shipment networks [9, 31].

A disaster relief management problem is formulated as a multicommodity multimodal network flow problem with time windows by Haghani and Oh [164] where commodities (food, medical supplies, machinery, and personnel) from different areas are to be shipped via many modes of transportation (car, ship, helicopter,...,etc.) in the most efficient manner to minimize the loss of life and maximize the efficiency of the rescue operations.

1.3.1.3 Production scheduling and planning:

Jewell [180] solves a warehousing and distribution problem for seasonal products by formulating it as a MCNF model where each time period is a transshipment node, one dummy source and one dummy sink node exist for each product, and arcs connect from source nodes to transshipment nodes, earlier transshipment nodes to later transshipment nodes, and transshipment nodes to sink nodes. Commodities are products to be shipped from sources to sinks with minimum cost.

D'Amours et al. [84] solve a planning and scheduling problem in a Symbiotic Manufacturing Network (SMN) for a multiproduct order. A broker receives an order of products in which different parts of the products may be manufactured by different manufacturing firms, stored by some storage firms, and shipped by a few transportation firms between manufacturing firms, storage firms and customers. The problem is to design a planning and scheduling bidding scheme for the broker to make decisions on when the bids should be assigned and who they should be assigned to, such that the total cost is minimized. They propose a MCNF model where each firm (manufacturing or storage) at each period represents a node, an arc is either a possible transportation link or possible manufacturing (storage) decision, and a commodity represents an order for different product.

Aggarwal et al. [1] use a MCNF model to solve an equipment replacement problem which determines the time and amount of the old equipments to be replaced to minimize the cost.

1.3.1.4 Other routing:

- **VLSI design** [61, 279, 270, 8]: Global routing in VLSI design can be modeled as an origin-destination MCNF problem where nodes represent collections of terminals to be wired, arcs correspond to the channels through which the wires run, and commodities are OD pairs to be routed.

- **Racial balancing of schools** [77]: Given the race distribution of students in each community, and the location, capacity and the ethnic composition requirements of each school, the problem is to seek an assignment of students to schools so that the ethnic composition at each school is satisfied, no student travels more than a specified number of minutes per day, and the total travel time for students to school is minimized. This is exactly a MCNF problem in the sense that students of the same race represent the same commodity.

- **Caching and prefetching problems in disk systems** [6, 7]: Since the speed of computer processors are much faster than memory access, caching and prefetching are common techniques used in modern computer disk systems to improve the performance of their memory systems. In prefetching, memory blocks are loaded from the disk (slow memory) into the cache (fast memory) before actual references to the blocks, so the waiting time is reduced in accessing memory from the disk. Caching tries to keep actively referenced memory blocks in fast memory. At any time, at most one fetch operation is executed. Albers and Witt [7] model this problem as a min-cost

MCNF problem that decides when to initiate a prefetch, and what blocks to fetch and evict.

- **Traffic equilibrium** [215, 110]: The traffic equilibrium law proposed by Wardrop [303] states that at equilibrium, for each origin-destination pair, the travel times on all routes used are equal, and are less than the travel times on all nonused routes. Given the total demand for each OD pair, the equilibrium traffic assignment problem is to predict the traffic flow pattern on each arc for each OD pair that follows Wardrop's equilibrium law. It can be formulated as a nonlinear MCNF problem where the bundle constraints are eliminated but the nonlinear objective function is designed to capture the flow congestion effects.

- **Graph Partitioning** [285]: The graph partitioning problem is to partition a set of nodes of a graph into disjoint subsets of a given maximal size such that the number of arcs with endpoints in different subsets is minimized. It is an NP-hard problem. Based on a lower bounding method for the graph bisection problem, which corresponds to a MCNF problem, Sensen [285] proposes three linear MCNF models to obtain lower bounds for the graph partitioning problem. He then uses branch-and-bound to compute the exact solution.

  Similarly, many NP-hard problems such as min-cut linear arrangement, crossing number, minimum feedback arc set, minimum 2D area layout, and optimal matrix arrangement for nested disection can be approximately solved by MCNF algorithms [217, 204, 206, 219].

1.3.2 Network design

Given a graph G, a set of commodities K to be routed according to known demands, and a set of facilities L that can be installed on each arc, the *capacitated network design* problem is to route flows and install facilities at minimum cost. This is a MCNF problem which involves flow conservation and bundle constraints plus some side constraints related to the installation of the new facilities. The objective function may be nonlinear or general discontinuous step-increasing [124].

Problems such as the design of a network where the maximum load on any edge is minimized [51] or the service quality and survivability constraints are met with minimum cost of additional switches/transport pipes in ATM networks [53] appear often in the telecommunication literature. Bienstock et al. use metric inequalities, aggregated formulations [50] and facet-defining inequalities [52] to solve capacitated survivable network design problems. Gouveia [152] discusses MCNF formulations for a specialized *terminal layout problem* which seeks a minimum spanning tree with hop constraints (a limit on the number of hops (links) between the computer center and any terminal in the network).

Maurras et al. [232, 163] study network design problems with jump constraints (i.e., each path has no more than a fixed number of arcs). Girard and Sansò [133] show that the network designed using a MCNF model significantly improves the robustness of the heuristic solutions at a small cost increase. Gendron et al. [130] write a comprehensive survey paper in multicommodity capacitated network design.

Similar problems also appear in transportation networks, such as locating vehicle depots in a freight transportation system so that the client demands for empty vehicles are satisfied while the depot opening operating costs and other transportation costs are minimized. Crainic et al. [80] have solved this problem by various methods such as branch-and-bound [82] and its parallelization [129, 56], dual-ascent [81], and tabu-search [83]. Crainic also writes a survey paper [79] about service network design in freight transportation. In fact, these problems are facility location problems. Geoffrion and Graves [132] model a distribution system design problem as a fixed charge MCNF problem. More MCNF models for facility location are surveyed by Aikens [5].

1.4 Formulations

Let N denote the set of all nodes in G, A the set of all arcs, and K the set of all commodities. For commodity k with origin s_k and destination t_k, c_{ij}^k represents its per unit flow cost on arc (i,j) and x_{ij}^k the flow on arc (i,j). Let b_i^k be the supply/demand at node i, and B^k be the total demand units of commodity k. Let u_{ij} be the arc capacity on arc (i,j). Without loss of generality, we assume each unit of each commodity consumes one unit of capacity from each arc on which it flows.

1.4.1 Node-Arc form

The node-arc form of MCNF problem is a direct extension of the conventional SCNF formulation. It can be formulated as follows:

$$\min \sum_{k\in K} \sum_{(i,j)\in A} c_{ij}^k x_{ij}^k = Z^*(x) \tag{Node-Arc}$$

$$s.t. \sum_{(i,j)\in A} x_{ij}^k - \sum_{(j,i)\in A} x_{ji}^k = b_i^k \quad \forall i \in N, \forall k \in K \tag{1.1}$$

$$\sum_{k\in K} x_{ij}^k \leq u_{ij} \quad \forall (i,j) \in A \text{ (bundle constraints)} \tag{1.2}$$

$$x_{ij}^k \geq 0 \quad \forall (i,j) \in A, \forall k \in K$$

where $b_i^k = B^k$ if $i = s_k$, $b_i^k = -B^k$ if $i = t_k$, and $b_i^k = 0$ if $i \in N\backslash\{s_k, t_k\}$. This formulation has $|K||A|$ variables and $|N||K| + |A|$ nontrivial constraints.

1.4.2 Arc-Path form

First suggested by Tomlin [297], the arc-path form of the min-cost MCNF problem is based on the fact that any network flow solution can be decomposed into path flows and cycle flows. Under the assumption that no cycle has negative cost, any arc flow vector can be expressed optimally by simple path flows.

For commodity k, let P^k denote the set of all possible simple paths from s_k to t_k, f_p the units of flow on path $p \in P^k$, and PC_p^c the cost of path p using c_{ij}^k as the arc costs. δ_a^p is a binary indicator which equals to 1 if path p passes through arc a, and 0 otherwise. It can be formulated as follows:

$$\min \sum_{k\in K} \sum_{p\in P^k} PC_p^c f_p = Z^*(f) \tag{Arc-Path}$$

$$s.t. \qquad \sum_{p\in P^k} f_p = 1 \quad \forall k \in K \tag{1.3}$$

$$\sum_{k\in K} \sum_{p\in P^k} (B^k \delta_a^p) f_p \leq u_a \quad \forall a \in A \text{ (bundle constraints)} \tag{1.4}$$

$$f_p \geq 0 \quad \forall p \in P^k, \forall k \in K$$

Inequalities (1.4) are the bundle constraints, and (1.3) are the convexity constraints which force the optimal solution for each commodity k to be a convex combination of some simple paths in P^k. Since we only consider the LP MCNF problem in this dissertation, the optimal solution can be fractional. For the binary MCNF problem where each commodity can only be shipped on one path, f_p will be be binary variables.

This formulation has $\sum\limits_{k \in K} |P^k|$ variables and $|K| + |A|$ nontrivial constraints.

1.4.3 Comparison of formulations

When commodities are OD pairs, $|K|$ may be $O(|N|^2)$ in the worst case. In such case, the node-arc form may have $O(|N|^3)$ constraints which make its computations more difficult and memory management less efficient.

The arc-path formulation, on the other hand, has at most $O(|N|^2)$ constraints but exponentially many variables. The problem caused by a huge number of variables can be resolved by column-generation techniques. In particular, when using the revised simplex method to solve the arc-path form, at most $|K| + |A|$ of the $\sum\limits_{k \in K} |P^k|$ variables are positive in any basic solution. In each iteration of the revised simplex method, a shortest path subproblem for each commodity can be solved to find a new path for a simplex pivot-in operation if it corresponds to a nonbasic column with negative reduced cost.

Dantzig-Wolfe (DW) decomposition is a common technique used to solve problems with block-angular structure. The algorithm decomposes the problem into two smaller sets of problems, a *restricted master problem* (RMP) and k subproblems, one for each commodity. It first solves the k subproblems and uses their basic columns to solve the RMP to optimality. Then it uses the dual variables of the RMP to solve the k subproblems, which produce nonbasic columns with negative reduced cost to be added to the RMP. The procedures are repeated until no more profitable columns are generated for the RMP. More details about DW will be discussed in Section 2.3.2.

Jones et al. [187] investigate the impact of formulation on Dantzig-Wolfe decomposition for the MCNF problem. Three formulations: origin-destination specific (ODS), destination specific (DS), and product specific (PS) are compared in Table 4 where shortest path, shortest path tree, and min-cost network flow problems are the subproblems for ODS, DS, and PS respectively.

Table 4: Comparison of three DW formulations [187]

subproblem*	$ODS \prec DS \prec PS$
RMP size	$ODS > DS > PS$
RMP sparsity**	$ODS \succ DS \succ PS$
number of master iterations	$ODS < DS < PS$
number of commodities	$ODS \gg DS > PS$
convergence rate	$ODS > DS > PS$
*: $A \succ B$ means A is harder than B ; **: $A \succ B$ means A is sparser than B	

Although the ODS form has a larger RMP and more subproblems (due to more commodities), it has a sparser RMP, easier subproblems, and better extreme points produced which contribute to faster convergence than other forms.

Knowing this fact, our solution methods to the min-cost ODMCNF problem will be path-based techniques.

1.5 Contributions and thesis outline

We consider the following to be the major contributions of this thesis:

- We survey and summarize MCNF problems and algorithms that have appeared in the last two decades.
- We survey and summarize shortest path algorithms that have appeared in the last five decades.
- We observe the connection between a new shortest path algorithm and the well-known Dijkstra's algorithm.
- We propose three new multiple pairs shortest path algorithms, discuss their theoretical properties, and analyze their computational performance.
- We give two new MCNF algorithms, and compare their computational performance with two standard MCNF algorithms.

This thesis has six chapters. Chapter 1 discusses MCNF applications and their formulations. Chapter 2 surveys MCNF solution techniques from the literature. Since our approaches require us to solve many *multiple pairs shortest paths* (MPSP) problems, we will propose new efficient MPSP algorithms. Chapter 3 first reviews the shortest path solution techniques that have appeared in the last five decades, and then introduces a new shortest path method which uses the *nonnegative least squares* (NNLS) technique in the *primal-dual* (PD) algorithmic framework, and discusses its relation to the well-known Dijkstra's algorithm. Chapter 4 discusses the theoretical details of our new MPSP algorithms. Chapter 5 presents the computational performance of our new MPSP algorithms. Finally, Chapter 6 illustrates our primal-dual MCNF algorithms, analyzes their computational performance, and suggests new techniques to improve the algorithms. Chapter 7 concludes this thesis and suggests future research.

CHAPTER II

SOLUTION METHODS FOR MULTICOMMODITY NETWORK FLOW PROBLEMS

In this chapter, we try to summarize most of the solution techniques that have appeared in the literature, especially those which appeared after 1980.

Due to the special block-angular structure of MCNF problems, many special solution methods have been suggested in the literature. *Basis partitioining methods* partition the basis in a way such that computing the inverse of the basis is faster. *Resource-directive methods* seek optimal capacity allocations for commodities. *Price-directive methods* try to find the optimal penalty prices (dual variables) for violations of the bundle constraints. *Primal-dual methods* raise the dual objective while maintaining complementary slackness conditions and will be discussed in more detail in Chapter 6. In the last two decades, many new methods such as *approximation methods*, *interior-point methods* and their parallelization have been proposed. Although the convex or integral MCNF problems are not the main interests of this dissertation, we still summarize recent progress in these topics.

2.1 Basis partitioning methods

The underlying idea of basis partitioining methods is to partition the basis so that the special network structure can be maintained and exploited as much as possible to make the inversion of the basis more efficient. Hartman and Lasdon [167] propose a Generalized Upper Bounding (GUB) technique which is a specialized simplex method whose only nongraphic operations involve the inversion of a working basis with dimension equal to the number of currently saturated arcs. Other basis partitioning methods based on GUB techniques have also been suggested by McCallum [238] and Maier [228].

Grigoriadis and White [156, 155] observe that, in practice, the number of active bundle constraints in the optimal solution are considerably smaller than the number of capacitated arcs. Therefore they partition the constraints into two groups: current and secondary constraints. Using the dual simplex method with Rosen's partition technique [273], they iteratively solve sequences of LPs that use only updated current constraints. These LPs do not need to be solved to optimality. The algorithm only requires a basis change when dual feasibility is maintained and the objective value decreases.

Kennington [200] implements the primal partitioning methods of Saigal [277] and Hartman and Lasdon [167]. His implementation of basis partitioning methods are computationally inferior to resource-directive subgradient methods.

EMNET, developed by McBride [233] and based on the factorization methods (variants of basis partition methods) of Graves and McBride [154], is designed to quickly solve LP problems with network structure and network problems with side constraints. Combined with other techniques such as resource-directive heuristics and price-directive column generation, they conclude that basis partitioning methods are computationally efficient. In particular, McBride and Mammer [236] use a capacity allocation heuristic to produce a hot-start basis for EMNET which shortens the computational time. McBride [235] uses a reource-directive decomposition heuristic to control the size of the working inverse and dramatically reduce the solution time for MCNF problems with many side constraints. Mamer and McBride [229, 237] also apply column generation techniques of DW decomposition to solve message routing problems and find a dramatic reduction in the number of pivots needed. The combination of EMNET and column generation shows a larger reduction in computation time than direct application of EMNET. A similar specialized simplex algorithm has also been proposed by Detlefsen and Wallace [92]. Their paper addresses more details of the network interpretation of basis inversion. Independent of the number of commodities, their working basis has dimesion at most equal to the number of arcs in the network, which seems to be suitable for telecommunication problems that often have a large number of commodities.

Castro and Nabona [70] implement a MCNF code named PPRN to solve min-cost MCNF problems which have a linear or nonlinear objective function and additional linear side constraints. PPRN is based on a primal partitioning method using Murtagh and Saunders'

[246] strategy of dividing the variables into three sets: basic, nonbasic and superbasic.

Farvolden et al. [109] partition the basis of the master problem using the arc-path form in DW decomposition. Their techniques isolate a very sparse and near-triangular working basis of much smaller dimension, and identify nine possible types of simplex pivots that can be done by additive and network operations to greatly improve the efficiency. Such techniques are later used to solve a MCNF problem with jump constraints (i.e., commodities can not be shipped along more than a fixed number of arcs) [232] and further refined by Hadjiat et al. [163] so that the dimension of the working basis is at most equal to the number of arcs in the network (almost half the size of the working matrix of [109]).

2.2 Resource-directive methods

Suppose on each arc (i,j) we have assigned capacity r_{ij}^k for each commodity k so that $\sum\limits_{k\in K} r_{ij}^k \leq u_{ij}$ is satisfied. Then the original problem is equivalent to the following *resource allocation problem* (RAP) that has a simple constraint structure but complex objective function:

$$\begin{aligned} \min \sum_{k\in K} z^k(r^k) &= z(r) && \text{(RAP)} \\ s.t. \quad \sum_{k\in K} r_{ij}^k &\leq u_{ij} \quad \forall (i,j)\in A \\ 0 \leq r_{ij}^k &\leq u_{ij} \quad \forall (i,j)\in A,\ \forall k\in K \end{aligned}$$

For each commodity k, $z^k(r^k)$ can be obtained by solving the following single commodity min-cost network flow subproblem.

$$\begin{aligned} \min \sum_{(i,j)\in A} c_{ij}^k x_{ij}^k &= z^k(r^k) \\ s.t. \sum_{(i,j)\in A} x_{ij}^k - \sum_{(j,i)\in A} x_{ji}^k &= b_i^k \quad \forall i\in N \\ 0 \leq x_{ij}^k \leq r_{ij}^k & \quad \forall (i,j)\in A \end{aligned}$$

It can be shown that the objective function $z(r)$ is piecewise linear on the feasible set of the capacity allocation vector r. There are several methods in the literature to solve the RAP such as *tangential approximation* [131, 132, 201], *feasible directions* [201], and the *subgradient method* [131, 161, 153, 168, 199]. Shetty and Muthukrishnan [289] give a parallel projection algorithm which parallelizes the procedure of projecting new resource allocations in the resource-directive methods.

2.3 Price-directive methods

Price-directive methods are based on the idea that by associating the bundle constraints with "correct" penalty functions (dual prices, or Lagrange multipliers), a hard MCNF problem can be decomposed into k easy SCNF problems.

2.3.1 Lagrange relaxation (LR)

Lagrange relaxation dualizes the bundle constraints using a Lagrangian multiplier $\pi \geq 0$ so that the remaining problem can be decomposed into k smaller min-cost network flow problems. In particular,

$$
\begin{aligned}
\min \sum_{k \in K} c^k x^k &+ \sum_{a \in A} \pi_a \Big(\sum_{k \in K} x_a^k - u_a\Big) = L(\pi) \\
s.t. \qquad \widetilde{N} x^k &= b^k, \; \forall k \in K \\
x &\geq 0
\end{aligned}
$$

where $\widetilde{N}$ is the node-arc incidence matrix. The Lagrangian dual problem seeks an optimal π^* for $L^* = \max\limits_{\pi \geq 0} L(\pi)$. This is a *nondifferentiable optimization problem* (NDO) having the format $\max\{\varphi(y) : y \in Y\}$ where φ is a concave nondifferentiable function and Y is a convex nonempty subset of $R_+^{|A|}$.

Subgradient methods are common techniques for determining the Lagrange multipliers. They are easy to implement but have slow convergence rates. They usually do not have a good stopping criterion, and in practice usually terminate when a predefined number of iterations or nonimproving steps is reached. Papers regarding subgradient methods have been listed in Section 2.2. Chapter 16 and 17 of [3] also present illustrative examples.

Bundle methods are designed to solve general NDO problems and thus are suitable for solving the Lagrangian dual problems. In particular, let $g(y')$ denote a supergradient of φ

at y', that is, $\varphi(y) \leq \varphi(y') + g(y - y')$ for all y in Y. Define the *bundle* to be a set β that contains all $\varphi(y_i)$ and $g(y_i)$ generated in previous iterations. The generic bundle method starts with an initial y' and β and searches for a tentative ascent direction d. It tentatively updates $y'' = y' + \theta d$. If the improvement $\varphi(y'') - \varphi(y')$ is large enough, the solution y' will move to y''. Otherwise, it adds $\varphi(y'')$ and $g(y'')$ to the bundle β so that next iteration may give a better tentative ascent direction.

In general, bundle methods converge faster and are more robust than subgradient methods. Frangioni's Ph.D. dissertation [118] is a rich resource for the bundle method and its application to solving MCNF problems. Gendron et al. [130] use the bundle method to solve a multicommodity capacitated fixed charge network design problem. Cappanera and Frangioni [59] give its parallelization together with discussion on other parallel price-directive approaches.

Both the subgradient methods and bundle methods are dual based techniques, in which usually extra effort has to be made to obtain a primal feasible solution. Next we will introduce methods that improve the primal feasible solution and compute the Lagrangian multipliers by solving a LP in each iteration.

2.3.2 Dantzig-Wolfe decomposition (DW)

As previously mentioned in Section 1.4.3, here we illustrate more details about the DW procedures using the arc-path form. The primal path formulation is as follows:

$$\min \sum_{k \in K} \sum_{p \in P^k} PC_p^c f_p = Z_P^*(f, s) \tag{P\_PATH}$$

$$s.t. \quad \sum_{p \in P^k} f_p = 1 \quad \forall k \in K \tag{2.1}$$

$$\sum_{k \in K} \sum_{p \in P^k} (B^k \delta_a^p) f_p \leq u_a \quad \forall a \in A \tag{2.2}$$

$$f_p \geq 0 \quad \forall p \in P^k, \forall k \in K$$

whose dual is

$$\max \sum_{k \in K} \sigma_k + \sum_{a \in A} u_a(-\pi_a) = Z_D^*(\pi, \sigma) \tag{D\_PATH}$$

$$s.t. \;\; \sigma_k + \sum_{a \in A} B^k \delta_a^p(-\pi_a) \leq PC_p^c \;\; \forall p \in P^k, \forall k \in K \tag{2.3}$$

$$\pi_a \geq 0 \;\; \forall a \in A$$

$$\sigma_k : \text{free} \;\; \forall k \in K$$

where σ_k are the dual variables for the convexity constraint (2.1) and $-\pi_a$ are the dual variables for the bundle constraints (2.2).

The complementary slackness (CS) conditions are:

$$(-\pi_a)(\sum_{k \in K} \sum_{p \in P^K} (B^k \delta_a^p) f_p - u_a) = 0 \;\; \forall a \in A \tag{CS.1}$$

$$\sigma_k(\sum_{p \in P^K} f_p - 1) = 0 \;\; \forall k \in K \tag{CS.2}$$

$$f_p(PC_p^{c+\pi} - \sigma_k) = 0 \;\; \forall p \in P^k, \;\; \forall k \in K \tag{CS.3}$$

where $PC_p^{c+\pi} := \sum_{a \in A} B^k \delta_a^p (c_a + \pi_a)$ and c_a is the original cost of arc a. The reduced cost of path p is $PC_p^c - \sum_{a \in A} B^k \delta_a^p(-\pi_a) - \sigma_k = \sum_{a \in A} B^k \delta_a^p (c_a + \pi_a) - \sigma_k = PC_p^{c+\pi} - \sigma_k$.

Suppose a feasible but nonoptimal primal solution (f, s) is known. We construct the RMP which contains the columns associated with positive f and s, solve the RMP to optimality, and obtain its optimal dual solution $(\sigma^*, -\pi^*)$. This optimal dual solution can be used to identify profitable columns (i.e. columns with reduced cost $PC_p^{c+\pi^*} - \sigma_k^* < 0$) and add them to construct a new RMP. The procedure for generating new columns is equivalent to solving k subproblems. Each subproblem corresponds to a shortest path problem from s^k to t^k for a single commodity k. In particular, using $(c_{ij} + \pi_{ij}^*)$ as the new cost for each arc (i, j), if the length of the shortest path p^{k*} for commodity k, $PC_{p^{k*}}^{c+\pi^*} = \sum_{(i,j) \in A} B^k \delta_{ij}^{p^{k*}} (c_{ij} + \pi_{ij}^*)$, is shorter than σ_k^*, then we add the column corresponding to path p^{k*} to the RMP. The algorithm iteratively solves the RMP and generates new columns until no more columns have negative reduced cost (which guarantees optimality). During the procedure, primal feasibility and complementary slackness are maintained.

The DW algorithm stated above can also be viewed as a procedure to identify the optimal dual variables $(\sigma^*, -\pi^*)$ by solving sequences of smaller LP problems (RMP). Compared to LR, DW spends more time in solving the RMP, but obtains better dual variables. In general, DW requires fewer iterations than LR to achieve optimality. Another advantage of DW is that it always produces primal feasible solutions which can be used as upper bounds in minimization, while its dual objective can be used as a lower bound. LR, on the other hand, usually only guarantees an optimal dual solution and may require extra effort to obtain an optimal primal solution.

Unlike the conventional dual-based subgradient methods, the *volume algorithm* of Barahona and Anbil [24] is an extension of the subgradient methods. It produces approximate primal solutions in DW procedures. The method is similar to the subgradient method and the bundle method introduced in Section 2.3.1, and has the following steps:

- **Step 0**: Given a feasible dual solution $-\overline{\pi}$ to the bundle constraint (2.2), we solve $|K|$ subproblems. Each subproblem has the following form:

$$\begin{aligned} \min \sum_{(i,j)\in A} (c_{ij}^k + \overline{\pi}_{ij}) x_{ij}^k &= \overline{Z}^k \\ s.t. \sum_{(i,j)\in A} x_{ij}^k - \sum_{(j,i)\in A} x_{ji}^k &= b_i^k \quad \forall i \in N \\ x_{ij}^k &\geq 0 \quad \forall (i,j) \in A \end{aligned} \tag{Vol(k)}$$

  Suppose (Vol(k)) has optimal solution $\overline{x}^k$ and optimal objective value $\overline{Z}^k$.
  Set $t = 1$, $\overline{x} = [\overline{x}^1 \ldots \overline{x}^{|K|}]^T$ and $\overline{Z} = \sum_{k=1}^{|K|} \overline{Z}^k - \sum_{(i,j)\in A} u_{ij}\overline{\pi}_{ij}$.

- **Step 1**: Compute $v_{ij}^t = u_{ij} - \sum_{k\in K} \overline{x}_{ij}^k$, and $\pi_{ij}^t = \max\{\overline{\pi}_{ij} + \theta v_{ij}^t, 0\}$ for each arc (i,j) where the step length $\theta = \overline{f}\frac{UB-\overline{Z}}{\|v^t\|^2}$, $\overline{f}$ is a parameter between 0 and 2, and UB is the upper bound on the objective for P_PATH which can be computed by any primal feasible solution.

  For each commodity k, we solve (Vol(k)) using the new π_{ij}^t instead of $\overline{\pi}_{ij}$ in the objective function, and obtain its optimal solution x^{kt} and optimal objective value

Z^{kt}

Set $x^t = [x^{1t} \dots x^{|K|t}]^T$ and $Z^t = \sum\limits_{k=1}^{|K|} Z^{kt} - \sum\limits_{(i,j)\in A} u_{ij}\pi_{ij}^t$.

Update $\overline{x} = \eta x^t + (1-\eta)\overline{x}$ where η is a parameter between 0 and 1.

- **Step 2**: if $Z^t > \overline{Z}$ then update $\overline{\pi}_{ij} = \pi_{ij}^t$ for each arc (i,j) and $\overline{Z} = Z^t$.

  Let $t = t+1$, and go to Step 1.

The algorithm terminates when either $\left\|v^t\right\|$ becomes too small or $\left|\sum\limits_{(i,j)\in A} u_{ij}\overline{\pi}_{ij}\right|$ is smaller than some threshold value.

Suppose $Z^t \leq \overline{Z}$ for all iterations $t_c \leq t \leq t_d$. In these iterations, the volume algorithm uses the new $\overline{x}_{ij}^k$ but the old $\overline{\pi}_{ij}$ and $\overline{Z}$ to compute the new subgradients v_{ij}^t. In such case, $\overline{x} = (1-\eta)^{t_d-t_c}x^{t_c} + (1-\eta)^{t_d-t_c-1}\eta x^{t_c+1} + \dots + (1-\eta)\eta x^{t_d-t_c-1} + \eta x^{t_d-t_c}$. That is, $\overline{x}$ is a convex combination of $\{x^{t_c}, \dots, x^{t_d}\}$ which is an approximation to the optimal arc flow solution.

The basic idea is, given $(\overline{Z}, -\overline{\pi})$ where $-\overline{\pi}$ is a feasible dual variable to the bundle constraint and $\overline{Z}$ is computed using $-\overline{\pi}$, the volume algorithm moves in a neighborhood of $(\overline{Z}, -\overline{\pi})$ to estimate the volumes below the faces that are active in the neighborhood; thus better primal solutions are estimated at each iteration. On the other hand, using the conventional subgradient methods, the subgradient v is usually determined by only one active face, and no primal solution can be computed.

The volume algorithm has been tested with success in solving some hard combinatorial problems such as crew scheduling problems [25], large scale facility location problems [26, 27], and Steiner tree problems [23]. In [23], the problem is formulated as a non-simultaneous MCNF problem. It is believed that the volume algorithm is advantageous for solving set partitioning-type problems including the MCNF problems. It will be interesting to learn how this new algorithm performs in solving general MCNF problems. To the best of our knowledge, no such work has been done yet.

2.3.3 Key variable decomposition

Based on a primal partitioning method of Rosen [273], Barnhart et al. [36] give a column generation algorithm which is especially suitable for problems with many commodities (OD pairs). In particular, they select a candidate path called a *key path*, $key(k)$, for each commodity k. After performing some column operations to eliminate $key(k)$ for each k, they obtain the following CYCLE form:

$$\min \sum_{k\in K}\sum_{p\in P^k} CC_p^c f_p + \sum_{a\in A} M\alpha_a = Z_C^*(f,\alpha) \tag{CYCLE}$$

$$s.t. \quad \sum_{p\in P^k} f_p = 1 \quad \forall k \in K \tag{2.4}$$

$$\sum_{k\in K}\sum_{p\in P^k} B^k(\delta_a^p - \delta_a^{key(k)})f_p - \alpha_a \le u_a - \sum_{k\in K}(B^k\delta_a^{key(k)}) \ \forall a \in A \tag{2.5}$$

$$f_p \ge 0 \quad \forall p \in P^k, \forall k \in K$$

$$\alpha_a \ \ge 0 \quad \forall a \in A$$

where α_a is an artificial variable for each arc a with large cost M, and $CC_p^c = PC_p^c - PC_{key(k)}^c$ represents the cost of several disjoint cycles obtained by the symmetric difference of path p and $key(k)$. This CYCLE formulation is essentially the same as the P_PATH formulation in Dantzig-Wolfe decomposition. The artificial variables with large costs here are used to get an easy initial primal feasible solution for the algorithm.

When the number of commodities (OD pairs) is huge, both CYCLE and P_PATH will have many constraints, which makes the RMP more difficult. To overcome this computational burden, Barnhart et al. exploit Rosen's key variable concept which relaxes the nonnegativity constraints for each key path (i.e., it allows $f_{key(k)}$ to be negative). The RMP can be solved by iteratively solving these easier problems RELAX(i), each of which contains only (1) all of the bundle constraints and (2) the nonnegativity constraints for all variables except the key variables.

$$\min \sum_{k\in K}\sum_{p\in P^k} CC_p^{c,i} f_p^i + \sum_{a\in A} M\alpha_a^i = Z_{CR(i)}^*(f,\alpha) \tag{RELAX(i)}$$

$$s.t. \sum_{k\in K}\sum_{p\in P^k} B^k(\delta_a^p - \delta_a^{key(k,i)})f_p^i - \alpha_a^i \le u_a - \sum_{k\in K}(B^k\delta_a^{key(k,i)}) \ \forall a \in A \tag{2.6}$$

$$f_p^i \ge 0 \quad \forall p \in P^k \setminus key(k,i), \forall k \in K \ ; \ f_{key(k,i)} : \text{free} \quad \forall k \in K$$

$$\alpha_a^i \ \ge 0 \quad \forall a \in A$$

After solving RELAX(i), the algorithm will check the sign of key variables by calculating $f^{i*}_{key(k,i)} = 1 - \sum\limits_{p \in P^k \setminus key(k,i)} f^{i*}_p$. For those key variables with negative signs, the algorithm will perform a *key variable change* operation which replaces the current key variable with a new one. Among all the positive path variables, the one with the largest value is usually chosen since intuitively that path is more likely to have positive flow in the optimal solution. This algorithm maintains dual feasibility and complementary slackness while trying to achieve primal feasibility (which will be attained when all key variables become nonnegative).

Like the subproblems of DW, the subproblems of RELAX(i) are shortest path problems for each commodity k. Suppose in some iteration we solve the RMP of RELAX(i) and obtain the optimal dual solution associated with the bundle constraint (2.6), denoted by $-\pi^*$. A shortest path p^{k*} is generated if its reduced cost $PC^{c+\pi^*,i}_{p^{k*}} - PC^{c+\pi^*,i}_{key(k,i)} < 0$. Barnhart et al. [34, 35] further use a *simple cycle heuristic* to generate more good columns to shorten the overall computational time. In particular, for commodity k, the symmetric difference of shortest path p^{k*} and key path $key(k,i)$ forms several simple cycles. Intuitively, these simple cycles have better potential to contain positive flows since they contribute to both the shortest path p^{k*} and the key path $key(k,i)$. Each simple cycle corresponds to a *simple column* in the RELAX(i) formulation. Suppose p^{k*} and $key(k,i)$ form $\widetilde{n}$ simple cycles. Instead of adding only one column, they add all these $\widetilde{n}$ simple columns at once. Such heuristics do speed up the LP solution procedures.

We will exploit this key variable concept in Section 6.2.

2.3.4 Other price-directive methods

The primal column generation techniques correspond to the dual cutting-plane generation schemes which increasingly refine the polyhedral approximation of the epigraph of the Lagrangian $L(\pi)$. Goffin et al. [137] propose a specialized *analytic center cutting plane method* (ACCPM) [136, 18] to solve the nonlinear min-cost MCNF problems.

Using a piecewise continuous, smooth, convex and nonseparable *linear-quadratic penalty* (LQP) function to penalize the overflow of bundle constraints for each arc, Pinar and Zenios [266] design a parallel decomposition algorithm which parallelizes the procedures of solving the nonlinear master problem and independent subproblems in the framework of price-directive methods.

Schultz and Meyer [283] give a price-directive interior-point method using a barrier function (see Section 2.6). Many approximation algorithms are also based on price-directive techniques and will be discussed in Section 2.5. The Section 1.3 of Grigoriadis and Khachiyan [160] cites other old price-directive methods that we omit in this section.

2.4 Primal-dual methods (PD)

The *primal-dual method* is a dual-ascent LP solution method which starts with a feasible dual solution, and then iteratively constructs a primal feasibility subproblem called the *restricted primal problem* (RPP) based on the complementary slackness (CS) conditions. It uses the optimal dual solution of RPP to improve the current dual solution if primal infeasibility still exists. The algorithm terminates when all primal infeasibility disappears.

Jewell [182, 181] proposes a PD algorithm to solve the min-cost MCNF problem. Given feasible dual variables, he uses the CS conditions to construct a restricted network which can be viewed as $|K|$ layers of single-commodity networks plus some connecting arcs between layers representing the bundle constraints. The RPP becomes a max MCNF problem. He uses a technique similar to the resource-directive methods which first identifies augmenting paths inside each layer until the layer is saturated, and then solves a capacity reallocation problem between layers. His approach requires enumeration of all paths and cycles, which is not efficient.

Barnhart and Sheffi [32, 38] present a network-based primal-dual heuristic (PDN) which solves a large-scale MCNF problem that is impractical to solve using other exact solution methods. In each primal-dual iteration of PDN, the RPP is solved by a network-based *flow adjustment algorithm* (FAA) instead of using conventional LP methods. When the FAA experiences insufficient progress or cycling, the simplex method will be invoked to solve the RP.

Given a feasible dual solution, PDN iteratively identifies an admissible set S^k which contains arcs with zero reduced cost in the node-arc form for each commodity k, searches for a flow assignment that satisfies the CS conditions using the FAA, and improves the current dual solution if the RPP is solved with positive primal infeasibility. PDN terminates when either the RPP cannot be solved by the FAA or optimality is achieved. In the first case, if the simplex method is invoked but fails due to the huge size of RPP, Barnhart uses a primal solution heuristic [33] that generates feasible primal solutions from the known dual feasible solution.

Polak [268] gives a Floyd-Warshall-like algorithm to determine the step length required to improve the dual solution in the PD procedures of solving MCNF problems. More details about PD algorithms will be discussed in Chapter 6.

2.5 Approximation methods

Given a desired accuracy $\epsilon > 0$, an *ϵ-optimal* solution is a solution within $(1+\epsilon)$ (for minimization problems, or $(1-\epsilon)$ for maximization problems) of the optimal one. A family of algorithms is called a *fully polynomial time approximation scheme* (FPTAS) when it computes an *ϵ-optimal* solution in time polynomial in the size of problem input and ϵ^{-1}.

Shahrokhi and Matula [287] present the first combinatorial FPTAS for a special case of the max-concurrent flow problem that has arbitrary demands and uniform capacities. The idea is to start with a flow satisfying the demands but not the bundle constraints, and then iteratively reroute flow to approach optimality. They also use an exponential function of the arc flow as the "length" of the arc to represent its congestion, so that their algorithm will iteratively reroute flow from longer paths (more congestion) to shorter paths (less congestion). Using different ways for measuring the congestion on arcs and a faster randomized method for choosing flow paths, Klein et al. [205] further improve the theoretical running time.

Leighton et al. [218] use the same randomized technique of [205] but a different rerouting scheme to solve a max-concurrent flow problem that has nonuniform integer capacities. Instead of only rerouting a single path of flow as in [287, 205], they solve a min-cost flow problem which takes more time but reroutes more flows and makes greater progress at each iteration. Goldberg [138] and Grigoriadis and Khachiyan [157] independently use different randomized techniques to reduce the randomized algorithmic running time of [218] from a factor of ϵ^{-3} to a factor of ϵ^{-2}. Radzik [269] achieves the same complexity using a deterministic approximation algorithm.

Most of the approximation algorithms mentioned above are designed for max MCNF or max-concurrent flow problems, which can be viewed as packing LPs. In particular, let $\hat{P}^k$ denote the polytope of all feasible flows for commodity k. Then, the max MCNF problem can be viewed as packing feasible paths from the polytope $\bigcup\limits_k \hat{P}^k$ subject to the bundle constraints. Similarly, the min-cost MCNF problem can be viewed as packing paths subject to bundle and budget constraints. It can be solved by iterations of a separation problem that determines the optimal budget using bisection search [267].

Recently, Grigoriadis and Khachiyan[157, 158], Plotkin et al. [267], Garg and Könemann [128, 208], and Young [309] have investigated approximation methods for solving packing and covering problems, especially on applications in the MCNF fields. These algorithms are based on Lagrangian relaxation and can be viewed as randomized potential reduction methods [157, 158]. For example, given a current flow f_1, [267] computes an optimal flow f_1^* (or [157] computes an approximately optimal flow) that uses a "penalty" potential function as the arc length to represent the violation of bundle constraints, and updates the flow $f_2 = (1-\sigma)f_1 + \sigma f_1^*$ using some predefined constant σ so that the total "potential" in the next iteration will be reduced. The length for each arc is usually an exponential function [157, 158, 267, 309] (or a logarithmic function [159]) measuring the congestion (i.e., violation of bundle constraints). While [267, 157] improve the potential by a fixed amount and reroute flows at each iteration, [309] gives an "oblivious" rounding algorithm which builds the flow from scratch without any flow rerouting operations. The running time of [267] and [309] depend on the the maximum possible "overflow" of the bundle constraints for points in the polytope $\bigcup\limits_k \hat{P}^k$, denoted as the *width* of the problem.

Karger and Plotkin [192] combine the approximate packing techniques of [267] and the flow rerouting order of [269] to give a better deterministic approximation algorithm to solve min-cost MCNF problems. They develop new techniques for solving the "packing with budget" problems. Such techniques can be used to solve problems with multiple cost measures on the arc flows. Its direct implementation is slow, but a fast practical implementation is proposed by Oldham et al. [256, 139] using techniques from [159, 192, 267] to fine-tune many algorithmic parameters.

Garg and Könemann [128, 208] propose a simple combinatorial approximation algorithm similar to Young's [309], which has the most improvement at each iteration using shortest path computations. Fleischer [111] further improves the algorithms of [128] and gives approximation algorithms that have the best running time for solving the max MCNF, max-concurrent flow and min-cost MCNF problems.

Grigoriadis and Khachiyan [158] use a 2-phase exponential-function reduction method to solve general block-angular optimization problems. Both phases are resource-sharing problems. The first phase is to obtain a feasible solution interior to the polytope $\bigcup_k \hat{P}^k$. The second phase uses a modified exponential potential function to optimize on both bundle and flow conservation constraints.

Awerbuch and Leighton [19, 20] give deterministic approximation algorithms which use local-control techniques similar to the preflow-push algorithm of Goldberg and Tarjan [141, 143]. Unlike previous approximation algorithms that usually relax the bundle constraints and then determine shortest paths or augmenting paths to push flow towards sinks, their algorithms relax the flow conservation constraints and use an "edge-balancing" technique which tries to send a commodity across an arc (u, v) if node u has more congestion than node v. Based on these edge-balancing methods [19, 20] (denoted as *first-order algorithms*), Muthukrishnan and Suel [248] propose a *second-order distributed flow algorithm* which decides the amount of flow to be sent along an arc by a combination of the flow sent on the same arc in previous iteration and the flow determined using the first-order algorithms. Their experiments show a significant improvement in running time compared to the first-order algorithms. Kamath et al. [190, 262] generalize the algorithm of [20] to solve min-cost MCNF problems, but their algorithm is inferior to [192].

Schneur and Orlin [280, 281] present a penalty-based approximation algorithm that solves the min-cost MCNF problems as a sequence of a finite number of scaling phases. In each phase, $|K|$ SCNF problems are solved, one for each commodity. Besides the original linear flow cost $c_{ij}^k x_{ij}^k$, they assign a quadratic penalty term ρe_{ij}^2 for each arc (i,j) where $e_{ij} = \max\{\sum_{k \in K} x_{ij}^k - u_{ij}, 0\}$ represents the overflow (or excess) on arc (i,j). Thus each SCNF problem becomes a nonlinear min-cost network flow problem which is solved by a negative cycle canceling algorithm. In particular, at each scaling phase their algorithm iteratively sends exactly δ units of flow along negative δ-cycles which are defined as a cycle that allows flow of δ units. In the end of a scaling phase, the flow x is (δ,ρ)-*optimal* which means there exists no negative cycle that allows flow of δ units. By using new δ (usually $\frac{\delta}{2}$) and ρ (usually $R\rho$ where $1 < R < 2$) in the next scaling phase, the algorithm can be shown to terminate in finite number of scaling phases.

Bienstock [48] solves network design problems and other MCNF test problems (from [139]) by an approximate algorithm based on [158, 267] with new lower bounding techniques. Instead of using specialized codes such as shortest path or min-cost network flow programs, he uses CPLEX 6.5 to solve the subproblems. In his recent comprehensive paper [49], he reviews many exponential potential function based approximation algorithms and discusses detailed issues regarding empirical speedups for algorithms based on this framework.

The following summarizes the current fastest FPTAS :

- max MCNF problems: [111]
- max-concurrent flow problems: [111] (when graph is sparse, or $|K|$ is large), [218] and [269]
- min-cost MCNF problems: [111] (when $|K| > \frac{|A|}{|N|}$), [159]

2.6 Interior-point methods

Kapoor and Vaidya [191] speed up Karmarkar's algorithm [193] for solving MCNF problems. Using special computing and inversion techniques for certain matrices, Vaidya's path-following interior-point algorithm [300] achieves one of the current fastest running times for obtaining exact MCNF solutions. Murrary [245] also gives a specialized MCNF algorithm based on the algorithm of Monteiro and Adler [242].

Putting the flow imbalance as a quadratic term (equal to 0 if and only if flow conservation is maintained) in the objective function, Kamath and Palmon [189, 262] present an interior-point based FPTAS that solves a quadratic programming (QP) min-cost MCNF problem subject to only the bundle and nonnegativity constraints. With a sufficiently small ϵ, their algorithm can compute an exact solution using the rounding algorithm of Ye [307].

While these algorithms introduced above have good theoretical time bounds, they are considered to be practically unattractive. We now introduce some practical interior-point MCNF algorithms.

Schultz and Meyer [283] develop a decomposition method based on solving logarithmic barrier functions. Their algorithm involves three phases: relaxed phase, feasibility phase, and refine phase. After obtaining a solution of the relaxed problem (i.e., relaxing the bundle constraints), the feasibility phase tries to obtain a feasible interior point which will be refined to optimality in the refine phase. Both the feasibility and refine phases find an approximate solution of the *barrier problem* using a generalization of the *Frank-Wolfe* method [120] which does a multi-dimensional search rather than a line search and is more easily parallelized. Instead of using a complex coordinator [283] to determine step lengths in the search directions, Meyer and Zakeri [239, 310] further improve this procedure using multiple simple coordinators that take more advantage of parallelism.

General interior-point methods for LP usually need to solve a symmetric system of linear equations. In many cases, the Cholesky factorization still creates a dense submatrix even if the best fill-in minimization heuristics are applied. To speed up the factorization by taking advantage of the MCNF structure, basis partitioning techniques are used by Choi and Goldfarb [76] and Castro [67] in their primal-dual interior-point algorithms. To efficiently solve the dense linear system, Choi and Goldfarb [76] suggests parallel and vector processing while Castro [67] applies a *preconditioned conjugate gradient* (PCG) method. Parallel implementation on primal-dual interior-point methods using PCG techniques can be found in Castro and Frangioni [69] and Yamakawa et al. [305].

A specialized *dual affine-scaling* method (DAS) for solving an undirected min-cost MCNF problem has been implemented by Chardaire and Lisser [72, 73]. Starting with a feasible dual interior point, the algorithm iteratively searches for an improving direction by solving a symmetric linear system (similar to [76, 67]) that approaches the optimal dual solution. Using the same partitioning techniques of [76, 67], they solve the linear system without any special techniques. The same authors also implement a specialized analytic center cutting plane method (ACCPM) [136], a cutting plane method previously introduced in Section 2.3.4 which can be viewed as an interior-point method since it computes the central point of a polytope in each iteration.

2.7 Convex programming methods

Although this dissertation concentrates on solving the linear MCNF problems, here we introduce some nonlinear convex programming techniques in the literature ([90] in Section 2.7.2 and [188] in Section 2.7.3 are for LPs). Most of them are related to *augmented Lagrangian* methods which find a saddle point of a smooth min-max function obtained by augmenting the Lagrangian with quadratic terms of both primal and dual variables.

2.7.1 Proximal point methods

Based on their previous work [178], Ibaraki and Fukushima [179] apply a primal-dual proximal point method to solve convex MCNF problems. The method identifies an approximate saddle point of the augmented Lagrangian at each iteration and guarantees that these points converge to the unique Kuhn-Tucker point of the problem. Similar methods are also developed by Chifflet et al. [75] and Mahey et al. [227].

2.7.2 Alternating direction methods (ADI)

De Leone et al. [90] propose three variants of *alternating direction methods* (ADI) [209] which may be viewed as block Gauss-Seidel variants of augmented Lagrangian approaches that take advantage of block-angular structure and parallelization. These methods take alternating steps in both the primal and the dual space to achieve optimality. In particular, ADI is designed to solve the following problem:

$$\begin{aligned} \min \quad & G_1(x) \;+\; G_2(z) \\ s.t. \quad & Cx + b \;=\; Dz \end{aligned} \tag{ADI}$$

For example, let x be the flow vector, z be the proximal vector that reflects the consumption of the shared capacity, and p be the associated dual variables. The *resource proximization method* (RP_ADI) uses the following settings: $G_1(x) := \sum_{k\in K}\sum_{(i,j)\in A} c_{ij}^k x_{ij}^k$; $G_2(z) := 0$ if $\sum_{k\in K} z_{ij}^k \leq u_{ij}\ \forall (i,j) \in A$, or $G_2(z) := \infty$, otherwise; $C := I_{|A||K|}$; $D := I_{|A||K|}$; $b := 0$ and Λ is a diagonal penalty matrix whose k^{th} diagonal entry is $\Lambda_k \in (0, \infty)$. Define the augmented Lagrangian

$$L_\Lambda^k = \min_{\substack{\widetilde{N}x^k = b^k \\ 0 \leq x_{ij}^k \leq u_{ij}}} \sum_{(i,j)\in A} ((c_{ij}^k + p_{ij}^k)x_{ij}^k + \frac{1}{2}\Lambda_k (x_{ij}^k - z_{ij}^k)^2)$$

where $\widetilde{N}$ is the node-arc incidence matrix. Starting with a suitable dual variable $p_0 \geq 0$ and an initial capacity allocation z_0 satisfying $\sum_{k\in K} z_0^k = u$, RP_ADI iteratively computes the minimizer x^{k*} of L_Λ^k for each commodity k, updates multiplier p, adjusts the capacity allocation r, and updates the penalty matrix Λ of L_Λ^k until the termination criterion is met. Although RP_ADI is guaranteed to achieve the optimal objective value, primal feasibility may not be satisfied.

They also propose two other methods, the *activity and resource proximization method* (ARP_ADI) and the *activity proximation method* (AP_ADI). Their computational results show that AP_ADI and RP_ADI run significantly faster than ARP_ADI, and all of them are faster than the general-purpose LP solver MINOS 5.4 [247]. A similar method is also developed by Eckstein and Fukushima [99].

2.7.3 Methods of games

Kallio and Ruszczyński [188] view the solution procedure to a LP as a nonzero-sum game with two players. In particular, the primal players minimize the augmented Lagrangian function for the primal problem and the dual players maximize the augmented Lagrangian function for the dual problem. Each player optimizes his own objective by assuming the other's choice is fixed. They propose a parallel method where processors carry out under-relaxed Jacobi steps for the players. Based on the same idea of [188], Ouorou [257] solves *monotropic programming problems* [271] which minimize a separable convex objective function subject to linear constraints. He solves convex min-cost MCNF problems using a block-wise Gauss-Seidel method to find an equilibrium of the game in a primal-dual algorithmic framework. These techniques are similar to the alternating direction method discussed previously.

2.7.4 Other convex and nonlinear programming methods

Nagamochi et al. [249] use the relaxation method of Bertsekas [43, 46], a specialized PD algorithm, to solve strictly convex MCNF problems.

Lin and Lin [222] solve quadratic MCNF problems by a projected Jacobi method. They introduce a new *dual projected pseudo-quasi-Newton* (DPPQN) method to solve the quadratic subproblems induced in the projected Jacobi method. The DPPQN method computes a fixed-dimension sparse approximate Hessian matrix, which overcomes the difficult computation of an indefinite-dimension dense Hessian matrix that arises in the conventional Lagrangian Newton method.

Ouorou and Mahey [258] use a minimum mean cycle cancelling based algorithm [142] to solve a MCNF problem with a nonlinear separable convex cost function. More techniques about solving nonlinear convex MCNF problems are introduced in Chapter 8 and Chapter 9 of [45]. For survey on algorithms for nonlinear convex MCNF problems, see Ouorou et al. [259].

2.8 Methods for integral MCNF problems

All the algorithms introduced before this section are fractional solution methods. Adding the integrality constraints makes the MCNF problems much harder.

Much research has been done regarding the characterization of integral MCNF problems. For example, Evans et al. [106] have given a necessary and sufficient condition for unimodularity in some classes of MCNF problems. Evans makes the transformation from some MCNF problems to their equivalent uncapacitated SCNF problems [100, 101, 103], and proposes a specialized simplex algorithm [102] and heuristics [104, 105] for certain MCNF problems. Truemper and Soun [298, 299, 292] further investigate the topic and obtain additional results about unimodularity and total unimodularity for general MCNF problems. Other research about the characteristics of quater-integral, half-integral and integral solutions of MCNF problems can be found in [224, 282, 197, 198].

Aggarwal et al. [1] propose a resource-directive heuristic which first determines an initial feasible integral capacity allocation for each commodity on each arc, and then performs parametric analysis by varying the right-hand-side and using the corresponding dual variables to solve knapsack-type problems which determine better allocation for the next iteration. Their methods produce good integral feasible solutions but may require many LP computations for the parametric analysis, and thus may not be suitable for large problems.

The general integral MCNF algorithms are usually based on LP-based branch-and-bound schemes such as *branch-and-cut* [50, 57] or *branch-and-price* [34, 35, 13]. In branch-and-cut, cuts (if possible, facet-inducing ones) are dynamically generated throughout the branch-and-bound search tree to cut off fractional solutions. On the other hand, branch-and-price [302, 37] generates variables that are used in conjunction with column generation to strengthen the LP relaxation and resolve the symmetry effect due to formulations with few variables. Barnhart et al. [34, 35] apply *branch-and-price-and-cut* which generates both variables and cuts during the branch-and-bound procedures to solve binary MCNF problems in which integral flow must be shipped along one path for each commodity (OD pair). Alvelos and Carvalho [13] propose another branch-and-price algorithm to solve the general integral MCNF problems which allows integral flow to be shipped along several paths for each commodity (OD pair).

2.9 Heuristics for feasible LP solutions

Occasionally, there are problems that need fast and good feasible primal or dual solutions. Barnhart proposes a dual-ascent heuristic [33] to quickly generate a good feasible dual solution based on the CS conditions (see Section 2.3.2) Suppose σ^* is the optimal dual variable for (2.1) and $-\pi^*$ is the optimal dual variable for the bundle constraint (2.2). Then, (CS.3) implies that the optimal flow should be assigned along shortest paths using $c_a + \pi^*_a$ as the new arc length for arc a, and (CS.1) implies that $\pi^*_a > 0$ when arc a is saturated and $\pi^*_a = 0$ when a is under-capacitated. Thus Barnhart's heuristic increases π_a

when a shortest path demand exceeds its capacity u_a for arc a, and decreases π_a ($\pi_a > 0$) when a shortest path demand is less than u_a.

The same paper also gives a primal solution generator based on the observation from (CS.1) that in the optimal solution an arc with a positive dual price should be saturated. So, the heuristic will try to send flow along the shortest path that contains arcs with positive dual prices first, and then along the arcs with zero dual prices. In general, this primal solution generator does not guarantee a feasible primal solution, but it works well in all of their tests.

2.10 Previous computational experiments

Comprehensive survey papers on solution methods and computational results for MCNF problems were done more than two decades ago by Assad [15, 17], Ali et al. [10], and Kennington [201]. All of these surveys suggest that the price-directive methods (DW) outperform the resource-directive methods, which are in turn superior to the basis-partitioning methods. A brief summary of the computational experiments done before 1990 can also be found in Farvolden et al. [109].

There are some popular MCNF test problems used by many researches for evaluating efficiency of their algorithms. For example, the Mnetgen family (mnet) created by Ali and Kennington [11], and the PDS family (PDS) from Carolan et al. [62] which models a patient distribution system where decisions have to be made in evacuating patients away from a place of military conflict, are commonly tested. The size of the PDS family depends on the planning horizon since it is a time-space network.

Chardaire and Lisser [73] implement four algorithms to solving undirected min-cost MCNF problems up to size $|N| = 119$, $|A| = 302$, $|K| = 7021$: (1) a simplex method using the basis-partitioning technique from [202], (2) a dual affine scaling method (DAS), (3) DW whose master problem is solved by a basis partitioning technique, and (4) an analytic center cutting plane method (ACCPM) [136]. They compare these four methods with CPLEX 4.0. The results show that DW is the most efficient implementation, DAS is the slowest in general, the basis partitioning method performs better than the DAS for smaller cases, and CPLEX 4.0 performs well for smaller cases but worse for larger cases. Similar undirected message routing problems up to $|N| = 100$, $|A| = 699$, $|K| = 1244$ are solved by McBride and Mamer [237] who compare (1) EMNET (see Section 2.1), (2) EMNET using shortest path pricing, and (3) CPLEX 6.5. Their results show that EMNET using shortest path pricing improves the generic EMNET, which in turn is much faster than CPLEX 6.5. A similar conclusion is also drawn in another paper by the same authors [229] which tests more MCNF problems such as PDS, mnet, dmx and JLF (which are all available at the website maintained by Antonio Frangioni[1]).

McBride [234] summarizes the computational performance of several MCNF algorithms (on different platforms), including (1) decomposition methods [266, 311],[283], (2) interior point methods [62],[225],[231], (3) a primal simplex method with advanced basis [236], and (4) a simplex method using basis-partitioning technique (EMNET) [233], on specific test problem sets such as KEN and PDS from the NETLIB library. He concludes that the basis-partitioning approach is efficient, especially in postprocessing which is often required in practice. However, the interior-point approaches can take more advantage of multiple processors operating in parallel.

Castro [67] shows that his specialized interior-point method, IPM, is faster than CPLEX 4.0, which in turn is much faster than PPRN [70]. Castro [66] discusses the empirical running times of some recent algorithms that have appeared in the last decade, compared with CPLEX 6.5. In basis-partition codes, EMNET [235] performs similarly to CPLEX 6.5 but better than PPRN [70] for PDS test problems. In price-directive codes, bundle methods [119] perform similarly to CPLEX 6.5.

Solving the QP test problems from the mnet and PDS families with size up to 500,000 variables and 180,000 constraints, Castro [68] concludes that IPM [67, 68] performs up to 10 times faster than the CPLEX 6.5 barrier solver. Both IPM and the CPLEX 6.5 barrier solver are much faster than PPRN [70], which in turn is much faster than ACCPM [137].

Despite their excellent theoretical running times, only few approximation algorithms have been implemented for practical use. [158], [139, 256], and [48] have reported their

[1] http://www.di.unipi.it/di/groups/optimize/Data/MMCF.html

computational experiments. MCMCF, the approximation code in [139, 256], is shown to be a few magnitudes faster than CPLEX 4.0.9 (dual simplex method) and PPRN [70] on problems generated by RMFGEN, MULTIGRID, and TRIPARTITE generators (see [139, 256]). These three problem generators are also used in [48].

Schneur and Orlin [281] compare their scaling algorithm (SAM.M) with other codes DW, PDN, and PPLP (DW and PDN are from [32], PPLP is from [109]). Their experiments show that PPLP and SAM.M are faster than DW and PDN on problems up to $|N| = 85$, $|A| = 204$, $|K| = 18$.

Farvolden et al. [109] compare their basis-partitioining method, PPLP, with MINOS and OB1 [226] on large-scale LTL networks and show that PPLP outperforms the others by a large factor.

In [32, 38], Barnhart compares her algorithm PDN to conventional DW decomposition and two primal-dual (PD) algorithms. It is shown that PDN is more advantageous for large-scale MCNF problems. In particular, her heuristics can produce good solutions for some freight assignment problems which the other LP-based methods (DW, PD) can not solve. Barnhart [33] also develops heuristics which show promising performance in obtaining a fast and good solution compared to OB1 and OSL [97].

Barnhart et al. [36] experiment with their PATH and CYCLE-RELAX algorithms (they use the formulation (P_PATH) and (RELAX(i)) as introduced in Section 2.3.3)) on message routing problems which contain many OD commodities. The CYCLE-RELAX algorithm is much faster than OB1 and OSL on solving problems up to $|N| = 324$, $|A| = 1019$, $|K| = 150$. When compared with their PATH algorithm, which is the conventional DW decomposition using column generation, the CYCLE-RELAX algorithm also displays faster running times on problems up to $|N| = 500$, $|A| = 1300$, $|K| = 5850$. In their later papers [34, 35], they solve binary MCNF problems up to $|N| = 50$, $|A| = 130$, $|K| = 585$. Their simple cycle heuristic that generates all the simple cycles from the symmetric difference between the shortest path and key path has shown to be very effective. In particular, the time to solve the LPs and the number of iterations of generating columns have been reduced by an average of 40% and 45% respectively, but the total number of columns generated is not increased. This means the simple cycle heuristic helps to generate good columns in fewer iterations, which shortens the overall computational time. Therefore, we will also incorporate this simple cycle heuristic in our proposed algorithm in Chapter 6.

2.11 Summary

We have introduced and summarized most of the methods in the literature for solving the MCNF problems. Different solution techniques may be advantageous for different MCNF problems depending on the problem characteristics. This dissertation focuses on solving the class of MCNF problems in which many OD commodities have to be routed. Therefore, the algorithms by Barnhart et al. [36, 34, 35] seem to be most suitable for our purpose. In Chapter 6, we will propose new primal-dual column generation algorithms which follow the PD algorithmic framework. We will also exploit the ideas from the CYCLE-RELAX algorithm.

In our algorithms (see Chapter 6), subproblems which seek shortest paths between multiple pairs must be repeatedly solved. Therefore, efficient multiple pairs shortest paths (MPSP) algorithms will speed up the overall computational efficiency of our MCNF algorithms. We introduce several shortest path algorithms in Chapter 3, give our own new MPSP algorithms in Chapter 4, and describe their computational performance in Chapter 5. We will introduce our MCNF algorithms and their computational performance in Chapter 6.

CHAPTER III

SHORTEST PATH ALGORITHMS: 1-ALL, ALL-ALL, AND SOME-SOME

In this thesis, we focus on solving ODMCNF, a class of min-cost MCNF problems where the commodities represent origin-destination (OD) pairs.

As introduced in Section 2.3.2, when we use Dantzig-Wolfe decomposition and column generation to solve ODMCNF, we generate new columns by solving sequences of shortest path problems for specific commodities (OD pairs) using $c+\pi^*$ as the new arc lengths where c is the original arc cost vector and $-\pi^*$ is the optimal dual solution vector associated with the bundle constraint after we solve the RMP. Since the RMP changes at each iteration of the DW procedures, π^* will also change. Therefore, the shortest paths between specific OD pairs have to be repeatedly computed with different arc costs on the same network. Efficient shortest path algorithms will speed up the overall computational time needed to solve the ODMCNF.

In this chapter, we first survey shortest path algorithms that have appeared in the literature, and then discuss the advantages and disadvantages of these shortest path algorithms in solving multiple pairs shortest path problems. Finally, we introduce a new shortest path algorithm based on a new LP solution technique that follows the primal-dual algorithmic framework to solve sequences of nonnegative least squares (NNLS) problems, and show its connection to the well-known Dijkstra's algorithm.

3.1 Overview on shortest path algorithms

During the last four decades, many good shortest path algorithms have been developed. We can group shortest path algorithms into 3 classes:

- those that employ combinatorial or network traversal techniques such as *label-setting methods*, *label-correcting methods* and their hybrids

- those that employ Linear Programming (LP) based techniques like primal network simplex methods and dual ascent methods
- those that use algebraic or matrix techniques such as Floyd-Warshall [113, 304] and Carré's [64, 65] algorithms.

The first two groups of shortest path algorithms are mainly designed to solve the *Single Source (or Sink) Shortest Path* (SSSP) problem, which is the problem of computing the shortest path tree (SPT) for a specific source (or sink) node. Algebraic shortest path algorithms, on the other hand, are more suitable for solving the *All Pairs Shortest Paths* (APSP) problem, which is the problem of computing shortest paths for all the node pairs.

3.2 Notation and definition

For a *digraph* $G := (N, A)$ with $n = |N|$ nodes and $m = |A|$ arcs, a *measure matrix* C is the $n \times n$ matrix in which element c_{ij} denotes the length of arc (i, j) with *tail* j and *head* i. $c_{ij} := \infty$ if $(i, j) \notin A$. A *walk* is a sequence of r nodes $(n_1, n_2.., n_r)$ composed of $(r-1)$ arcs, (n_{k-1}, n_k), where $2 \leq k \leq r$ and $r \geq 2$. A *path* is a walk without repeated nodes so that all n_k are different. A *cycle* is a walk where all n_k are different except that the starting and ending nodes are the same; that is, $n_1 = n_r$. The length of a path (cycle) is the sum of lengths of its arcs. When we refer to a shortest path tree with root t, we mean a tree rooted at a sink node t where all the tree arcs point towards to t.

The *distance matrix* X is the $n \times n$ array with element x_{ij} as the length of the shortest path from i to j. Let $[succ_{ij}]$ denote the $n \times n$ *successor matrix*. That is, $succ_{ij}$ represents the node that immediately follows i in the shortest path from i to j. We could construct the shortest path from i to j by tracing the successor matrix. In particular, the shortest path from i to j is $i \to k_1 \to k_2 \to \ldots \to k_r \to j$, where $k_1 = succ_{ij}$, $k_2 = succ_{k_1 j}, \ldots$, $k_r = succ_{k_{r-1} j}$, and $j = succ_{k_r j}$. If node i has a successor j, we say node i is the *predecessor* of node j. Let x^*_{ij} and $succ^*_{ij}$ denote the shortest distance and successor from i to j in G.

We say that node i is *higher* (*lower*) than node j if the index $i > j$ ($i < j$). A node i is said to be the *highest* (*lowest*) node in a node set $LIST$ if $i \geq k$ ($i \leq k$) $\forall$ $k \in LIST$ (see Figure 4 in Section 4.1).

For convenience, if there are multiple arcs between a node pair (i, j), we choose the shortest one to represent arc (i, j) so that c_{ij} in the measure matrix is unique without ambiguity. If there is no path from i to j, then $x_{ij} = \infty$.

Given two arcs (s, k) and (k, t), a *triple comparison* $s \to k \to t$ compares $c_{sk} + c_{kt}$ with c_{st}. A *fill-in* happens when there is no arc (s, t) (i.e., $c_{st} = \infty$) but the triple comparison $s \to k \to t$ makes $c_{sk} + c_{kt} < \infty$.

Path algebra is an ordered semiring $(S, \oplus, \otimes, e, \emptyset, \preceq)$, with $S = \mathbb{R} \cup \{\infty\}$ and two binary operations defined as in Table 5.

Table 5: Path algebra operators

generalized addition $(\oplus)$	$a \oplus b$	$=$	$\min\{a, b\}$	$\forall a, b \in S$
generalized multiplication$(\otimes)$	$a \otimes b$	$=$	$a + b$	
unit element e has value 0				
null element $\emptyset$ has value ∞				
the ordering $\preceq$ is the usual ordering ($\leq$) for real numbers.				

Path algebra obeys the commutative, associative and distributive axioms. (See [108, 65, 21, 63, 274, 275] for details.) We can also define generalized addition $(\oplus)$ and generalized multiplication $(\otimes)$ of matrices with elements in S as follows: Suppose $A, B, C, D \in S^{n \times n}$, where $A = [a_{ij}]$, $B = [b_{ij}]$, then $A \oplus B = C$, and $A \otimes B = D$ satisfy $c_{ij} = a_{ij} \oplus b_{ij}$ and $d_{ij} = \sum_{k=1}^{n} (a_{ik} \otimes b_{kj})$, where the symbol $\sum$ denotes a generalized addition.

Many shortest path algorithms in the literature can be viewed as methods for solving the linear systems defined on the path algebra, and will be reviewed in Section 3.4.

3.3 *Single source shortest path (SSSP) algorithms*

The Single Source (or Sink) Shortest Path (SSSP) problem determines the shortest path tree (SPT) for a specific source (or sink) node. SSSP algorithms in the literature can be roughly grouped into two groups: (1) Combinatorial algorithms which build the SPT based on combinatorial or graphical properties. (2) LP-based algorithms which view the SPT problem as a LP problem and solve it by specialized LP techniques.

In this section we also introduce a new algorithm of Thorup and give a summary of previous computational experiments for many SSSP algorithms.

3.3.1 Combinatorial algorithms

The basic idea of the combinatorial SSSP algorithms is that, creating a node bucket (named LIST, usually initiated by the root node), the algorithms select a node from LIST, scan the node's outgoing arcs, update the distance labels for the endnodes of these outgoing arcs, put the updated nodes into LIST, and then choose another node from LIST. The procedures repeat until LIST becomes empty.

The differences between combinatorial shortest path algorithms are in the ways of maintaining node candidates to be scanned from LIST. Many sophisticated data structures have been used to improve the time bounds.

Label-setting methods, first proposed by Dijkstra [95] and Dantzig [85], choose the node with the smallest distance label. Methods that use special data structures to quickly choose the min-distance-label node have been developed. These data structures include the binary heap of Johnson [186], Fibonacci heap of Fredman and Tarjan [121], Dial's bucket [93], and radix heap of Ahuja et al. [4]. Many other complicated bucket-based algorithms have been suggested in the long computational survey paper by Cherkassky et al. [74].

On the other hand, label-correcting methods, first proposed by Ford [114], Moore [243], Bellman [40] and Ford and Fulkerson [117], have more flexibility in choosing a node from LIST. These data structures include the queue of Bellman [40], dequeue of Pape [264] and Levit [220], two-queue of Pallottino [260], dynamic bread-first-search of Goldfarb et al. [146], small-label-first (SLF) or large-label-last (LLL) of Bertsekas [44], and topological ordering of Goldberg and Radzik [140]. Yen [308] also gives another special node scanning procedure which eliminates approximately half of the node label updating operations of Bellman's algorithm.

Some efficient label-correcting algorithms [264, 220, 260, 140] are based on the following heuristic: suppose a node i in LIST has been scanned in previous iterations, and some of its successors, say, nodes $i_l, \ldots, i_k$, are also in LIST. Then it is better to choose node i to scan before choosing nodes $i_l, \ldots, i_k$. The rationale is, if node i_l, a successor of node i, is chosen before node i to scan, then later when node i is updated and scanned, node i_l will have to be reupdated. Thus choosing node i rather than choosing any of its successors tends to save label updating operations. However, such heuristics may not guarantee a polynomial time algorithm. For example, the dequeue implementation of [264, 220] is very efficient in practice but has a pseudopolynomial theoretical running time. The other algorithms [260, 140] guarantee polynomial time bounds, but perform worse in practice.

The threshold algorithm of Glover et al. [134] combines techniques from both label-setting and label-correcting algorithms. In particular, the algorithm maintains two queues: NEXT and NOW. Nodes are put into NOW from NEXT if their distance label are less than the threshold, and the algorithm scans nodes in NOW and puts new nodes to NEXT. After NOW is empty, a new threshold is set and the algorithm iterates until finally NEXT becomes empty. The performance of this algorithm is sensitive to the way of adjusting threshold value; thus a fine-tuning procedure may be required to achieve better efficiency.

Although label-setting methods have more favorable theoretical time bounds than label-correcting methods, their overhead in sorting the node candidates may worsen their empirical performance, especially for sparse networks. The current best time bound of combinatorial algorithms to solve SSSP is $O(\min\{m + n \log n,\ m \log\log C,\ m + n\sqrt{\log C})$ obtained by [121],[184] and [4] for cases with nonnegative arc lengths, and $O(mn)$ by most label-correcting algorithms for cases with general arc lengths and no negative cycles (see Chapter 4 and Chapter 5 of [3]).

3.3.2 LP-based algorithms

Suppose we are solving an ALL-1 shortest path problem which determines a SPT rooted at sink node t. This ALL-1 SSSP can be formulated as the following LP:

$$\min \sum_{(i,j)\in A} c_{ij}x_{ij} = Z^{P*}_{ALL-1}(x) \tag{SSSP}$$

$$s.t. \sum_{(i,j)\in A} x_{ij} - \sum_{(j,i)\in A} x_{ji} = b_i = \begin{cases} 1 & \text{, if } i \neq t \\ -(n-1) & \text{, if } i = t \end{cases} \quad \forall i \in N \tag{3.1}$$

$$x_{ij} \geq 0 \quad \forall (i,j) \in A$$

In particular, the problem can be viewed as if every node other than t (a total of $n-1$ nodes) sends one unit of flow to satisfy the demand $(n-1)$ of the root node t. The constraint coefficient matrix of (3.1) is the node-arc incidence matrix $\widetilde{N}$ introduced in Section 1.2. It is totally unimodual and guarantees that the LP solution of SSSP will be integral due to the integral right hand side. Moreover, because there exists a redundant constraint in SSSP, we can remove the last row of (3.1) to get a new coefficient matrix $\overline{N}$ and obtain the following new LP:

$$\min cx = Z^{P*}_{ALL-1}(x) \tag{ALL-1-Primal}$$
$$s.t.\ \ \overline{N}x = 1\ \forall i \in N \setminus t \tag{3.2}$$
$$x \geq 0$$

whose dual is

$$\max \sum_{i \in N \setminus t} \pi_i = Z^{D*}_{ALL-1}(\pi) \tag{ALL-1-Dual}$$
$$s.t.\ \ \pi_i - \pi_j \leq c_{ij}\ \forall (i,j) \in A, i,j \neq t \tag{3.3}$$
$$\pi_i \leq c_{it}\ \forall (i,t) \in A \tag{3.4}$$
$$-\pi_j \leq c_{tj}\ \forall (t,j) \in A \tag{3.5}$$

This LP thus can be solved by a specialized simplex method, usually called the network simplex method.

In network simplex method, a tree corresponds to a basis, and each dual variable π_i corresponds to a distance label associated with node i (thus $\pi_t = 0$). Let the reduced cost associated with an arc (i,j) be $c^{\pi}_{ij} = c_{ij} - \pi_i + \pi_j$. Then starting from the root r, we can set π such that every tree arc has zero reduced cost.

Primal network simplex starts with a spanning tree $T(t)$ rooted at t which can easily be constructed by any tree-search algorithm, and then identifies some non-tree arc (u,v) with $c^{\pi}_{uv} < 0$. Adding (pivoting in) the non-tree arc (u,v) to the tree $T(t)$ will create a cycle with negative reduced cost. The algorithm sends flow along the cycle, identifies (pivots out) a tree arc (u,w) to become a non-tree arc [145], and updates the duals associated with the subtree rooted at node u so that the reduced cost for all tree arcs remains 0. The algorithm iterates these procedures until no more non-tree arc has negative reduced cost, which means optimality has been achieved. Notice that in each iteration, the objective is strictly improved, which means each iteration is a nondegenerate pivot.

In fact, the label-setting and label-correcting methods in Section 3.3.1 can be viewed as variants of primal network simplex methods [94]. In particular, if we add artificial arcs which have a very large cost M from all the other nodes to the root t, the primal network simplex method which chooses the most negative reduced cost non-tree arc will correspond to the Dijkstra's algorithm. The label-correcting methods may be viewed as "quasi-simplex" algorithms since they identify a profitable non-tree arc, but only update the dual for one node instead of all the nodes in its rooted subtree. Dial et al. [94] suggests that the superiority of the "quasi-simplex" methods to the "full-simplex" methods is caused by the extra overhead involved in maintaining and updating the data structures in the "full-simplex" method. Indeed, the label-correcting methods have more pivots but each pivot is cheap to accomplish, while the primal network simplex methods have fewer pivots but each pivot involves more operations.

Goldfarb et al. [145] propose an $O(n^3)$ primal network simplex algorithm. The current fastest network simplex based SSSP algorithm is by Goldfarb and Jin [148] which has time bound $O(mn)$ and is as fast as the best label-correcting algorithms.

Several dual ascent algorithms [261] such as auction algorithms [47, 45, 71] and relaxation algorithms [45] have been proposed. The Dijkstra's algorithm can also be viewed as a primal-dual algorithm (See Section 5.4 and Section 6.4 of [263]). We will discuss in more detail the connection between three algorithms, Dijkstra's algorithm, primal-dual algorithm, and a new method called least squares primal-dual method (LSPD), in Section 3.6 later on.

In practice, the LP-based algorithms tend to perform worse than the combinatorial algorithms for solving SSSP [94, 58, 212].

3.3.3 New SSSP algorithm

Recently, Thorup [295, 296] proposed a deterministic linear time and space algorithm to solve the undirected SSSP for graphs with either integral or floating point nonnegative arc lengths on a random access machine (RAM). Based on Thorup's algorithm, Pettie and Ramachandran [265] use the minimum spanning tree structure to create a new algorithm on a pointer machine for the undirected SSSP with nonnegative floating point arc lengths. Their algorithm will have a better time bound of $O(m + n \log \log n)$ as long as the maximal ratio of any two arc lengths is small enough. These new algorithms, although shown to have good worst-case complexity, may be practically inefficient [14].

3.3.4 Computational experiments on SSSP algorithms

Extensive computational survey papers on SSSP algorithms have been written comparing the efficiency of the label-correcting, label-setting and hybrid methods on both artificially generated networks [260, 134, 135, 173, 96, 241, 74] and real road networks [312].

According to Cherkassky et al. [74], no single best algorithm exists for all classes of shortest path problems. They observed a double bucket implementation of Dijkstra's algorithm named DIKB and a label-correcting method named GOR1 which uses a *topological-scan* idea to be robust for most of their test cases. Zhan and Noon [312], using the same set of codes as Cherkassky et al., concluded that PAPE and TWO_Q, two variants of label-correcting methods, perform best on real road networks. They also recommend DIKA, a Dijkstra approximate buckets implementation, for cases with smaller arc lengths, and DIKB, which is more favorable for cases with larger arc lengths.

3.4 All pairs shortest path (APSP) algorithms

The All Pairs Shortest Path (APSP) problem determines the shortest paths between every pair of nodes. Obviously, it can be solved by n SSSP algorithms, one for each node as the root. Also, it can be solved by the LP reoptimization techniques which we will discuss in more detail in Section 3.5.2. In this section, we focus on algebraic algorithms based on the path algebra introduced in Section 3.2.

The optimality conditions of APSP are

$$x_{ij} = \begin{cases} \min\limits_{k \neq i,j} (c_{ik} + x_{kj}) & \text{for } i \neq j \\ 0 & \text{for } i = j \end{cases}$$

That is, suppose we know x_{kj}, the shortest distance from all other nodes k to j. Then, the shortest distance from i to j, x_{ij}, can be computed by choosing the shortest path among all possible paths from i to j via any node k. In fact, this is the well-known *Bellman's equation*. Using path algebra, Bellman's equation for solving APSP can be rewritten as

$$X = CX \oplus I_n \tag{3.6}$$

where I_n denotes an $n \times n$ identity matrix with 0 as the diagonal entries and ∞ as the off-diagonal entries.

It can be shown (see [65]) that (3.6) has a unique solution X^* if G has no cycles of zero length. An *extremal solution* X^* for (3.6) can be obtained recursively as

$$\begin{aligned} X^* &= I_n \oplus CX \\ &= I_n \oplus C(I_n \oplus CX) = I_n \oplus C \oplus C^2(I_n \oplus CX) \\ &= I_n \oplus C \oplus C^2 \oplus .... \oplus C^{n-1} \oplus C^n \oplus ... \\ &= I_n \oplus (I_n \oplus C) \oplus (I_n \oplus C^2) \oplus .... \oplus (I_n \oplus C^{n-1}) \oplus (I_n \oplus C^n) \oplus ... \\ &= I_n \oplus (I_n \oplus C) \oplus (I_n \oplus C^2) \oplus .... \oplus (I_n \oplus C^{n-1}) \\ &= I_n \oplus C \oplus C^2 \oplus .... \oplus C^{n-1} \\ &= (I_n \oplus C)^{n-1} \end{aligned} \tag{3.7}$$

In particular, the (i, j) entry of C^k, represents the shortest distance from i to j using paths containing at most k arcs. That is, the shortest distance from node i to node j can be obtained by the length of a shortest directed path from i to j using at most $n - 1$ arcs assuming no cycles with negative length [291].

Since we are only interested in the shortest path (not cycle) lengths, the generalized addition $(I_n \oplus C^k)$ will zero all the diagonal entries of C^k while retaining all the off-diagonal ones. The properties $(I_n \oplus C^k) = (I_n \oplus C^{n-1})$ for $k \geq n$ and $A \oplus A = A$ are used in the above derivation. They hold because we assume there are no negative cycles, and any path in G contains at most $n-1$ arcs.

Now we will review APSP algorithms that solve Bellman's equations (3.6), and then summarize APSP methods that involve matrix multiplications.

3.4.1 Methods for solving Bellman's equations

The APSP problem can be interpreted as determining the shortest distance matrix X that satisfies Bellman's equations (3.6). Techniques analogous to the direct or iterative methods of solving systems of linear equations thus could be used to solve the APSP problem. In particular, direct methods such as the *Gauss-Jordan* and *Gaussian elimination* correspond to the well-known *Floyd-Warshall* [113, 304] and *Carré's* [64, 65] algorithms, respectively. Iterative methods like the *Jacobi* and *Gauss-Seidel* methods actually correspond to the SSSP algorithms by Bellman [40] and Ford [117], respectively (see [65] for proofs of their equivalence). The relaxation method of Bertsekas [45] can also be interpreted as a Gauss-Seidel technique (see [261]).

Since the same problem can also be viewed as inverting the matrix $(I_n - C)$, the *escalator method* [244] for inverting a matrix corresponds to an inductive APSP algorithm proposed by Dantzig [87]. Finally, the *decomposition algorithm* proposed by Mill [240] (also, Hu [172]) decomposes a huge graph into parts, solves APSP for each part separately, and then reunites the parts. This resembles the *nested dissection method* (see Chapter 8 in [98]), a partitioning or tearing technique to determine a good elimination ordering for maintaining sparsity, when solving a huge system of linear equations. All of these methods (except the iterative methods) have $O(n^3)$ time bounds and are believed to be efficient for dense graphs.

Here, we review the algorithm proposed by Carré [64], which corresponds to Gaussian elimination. This algorithm is seldom referred to in the literature. We are more interested in this algorithm because it inspires us to develop our new MPSP algorithms in Chapter 4.

3.4.1.1 Carré's algorithm

The Gaussian elimination-like APSP algorithm proposed by Carré contains three phases: one LU decomposition procedure and n successive forward/backward operations. Suppose we first solve an ALL-1 shortest path tree rooted at a sink node t, and then repeat the same procedures for different sink nodes to solve the APSP problem. The reason for such unconventional notation (solving ALL-1 instead of 1-ALL shortest path problems) is that we usually solve for a column vector (corresponding to an ALL-1 distance vector) when we solve a system of linear equations. Here the t^{th}column in the distance matrix represents an ALL-1 shortest path tree rooted at a sink node t.

Algorithm 1 Carré

```
begin
    Initialize: ∀ s, x_ss := 0 and succ_ss := s;
                ∀(s,t), if (s,t) ∈ A then
                            x_st := c_st; succ_st := t;
                        else x_st := ∞ ; succ_st := 0;
    LU;
    for each sink node t do
        Forward(t);
        Backward(t);
end
```

There are three procedures: LU, $Forward(t)$, and $Backward(t)$.

Procedure LU

```
begin
    for k = 1 to n − 2 do
        for s = k + 1 to n do
            for t = k + 1 to n do
                if s ≠ t and x_st > x_sk + x_kt then
                    x_st := x_sk + x_kt; succ_st := succ_sk;
end
```

Procedure LU **:** The LU procedure is analogous to the LU decomposition in Gaussian elimination. That is, using node k, whose index is smaller than both s and t, as an intermediate node, if the current path from s to t is longer than the joint paths from s to k and k to t, we update x_{st} and its associated successor index. In particular, a shorter path from s to t is obtained by joining the two paths from s to k and from k to t.

In terms of algebraic operations, let l_{st} and u_{st} denote the (s,t) entry in the lower and upper triangle of X, respectively. The LU procedure is simply the variant of LU decomposition in Gaussian elimination. In particular, first we initialize them as $l_{st} := c_{st}$ $\forall$

$s > t$, $u_{st} := c_{st}\ \forall\ s < t$. Then the procedures compute

$$\left.\begin{array}{l} l_{st} := l_{st} \oplus \sum\limits_{k=1}^{t-1} (l_{sk} \otimes u_{kt}) \\ u_{ts} := u_{ts} \oplus \sum\limits_{k=1}^{t-1} (l_{tk} \otimes u_{ks}) \end{array}\right. \text{ for } t = 2, \ldots, (n-1) \text{ and } s = (t+1), \ldots, n$$

The complexity of LU is $O(n^3)$. For a complete graph with n nodes, this procedure will perform exactly $\sum\limits_{k=1}^{n-2} \sum\limits_{s=k+1}^{n} \sum\limits_{\substack{t=k+1 \\ j \neq i}}^{n} (1) = \frac{n(n-1)(n-2)}{3}$ triple comparisons.

Procedure Forward(t)
begin
  for $s = t + 2$ to n ***do***
    for $k = t + 1$ to $s - 1$ ***do***
      if $x_{st} > x_{sk} + x_{kt}$ ***then***
        $x_{st} := x_{sk} + x_{kt}$; $succ_{st} := succ_{sk}$;
end

Procedure *Forward*(t) **:** For any sink node t, *Forward*(t) is a procedure to obtain shortest paths from higher nodes to node t on G'_L, the induced subgraph representing the lower triangular part of X obtained after LU.

In terms of algebraic operations, this procedure is the same as solving $L \otimes y = b$, where the right hand side b is the t^{th} column of the identity matrix I_n. First we initialize $y_s := b_s$ $\forall\ s$. Then

$$y_s := b_s \oplus \sum_{k=t}^{s-1} (l_{sk} \otimes y_k), \text{ for } s = (t+1), \ldots, n$$

The complexity of *Forward*(t) is $O(n^2)$. For a complete graph with n nodes, this procedure will perform exactly $\sum\limits_{t=1}^{n-2} \sum\limits_{s=t+2}^{n} \sum\limits_{k=t+1}^{s-1} (1) = \frac{n(n-1)(n-2)}{6}$ triple comparisons in solving APSP.

Procedure Backward(t)
begin
  for $s = n - 1$ down to 1 ***do***
    for $k = s + 1$ to n ***do***
      if $x_{st} > x_{sk} + x_{kt}$ ***then***
        $x_{st} := x_{sk} + x_{kt}$; $succ_{st} := succ_{sk}$;
end

Procedure $Backward(t)$ **:** For any sink node t, $Backward(t)$ is a procedure that computes shortest paths from all nodes to node t on G'_U, the induced subgraph representing the upper triangular part of X obtained after LU.

In terms of algebraic operations, this procedure is the same as solving $U \otimes x = y$, where the right hand side y is the solution to the previous operation, $L \otimes y = b$. First we initialize $x_s := y_s \ \forall \ s$. Then

$$x_{n-s} := y_{n-s} \oplus \sum_{k=n-s+1}^{n} (u_{n-s,k} \otimes x_k), \text{ for } s = 1, \ldots, (n-1)$$

The complexity of $Backward(t)$ is $O(n^2)$. For a complete graph with n nodes, this procedure will perform exactly $\sum\limits_{t=1}^{n} \sum\limits_{\substack{s=1 \\ s\neq t}}^{n-1} \sum\limits_{\substack{k=s+1 \\ k\neq t}}^{n} (1) = \frac{n(n-1)(n-2)}{2}$ triple comparisons in solving APSP.

More about Carré's algorithm : Table 6 is an example that illustrates how Carré's algorithm performs the triple comparisons ($s \to k \to t$) to solve an APSP problem for a 4-node complete graph.

Table 6: Triple comparisons of Carré's algorithm on a 4-node complete graph

LU: $s \to k \to t$, $k < s,t$	
$k = 1$	$k = 2$
2→1→3	
2→1→4	
3→1→2	
3→1→4	3→2→4
4→1→2	
4→1→3	4→2→3

$Forward(t)$: $s \to k \to t$, $s > k > t$			
$t = 1$	$t = 2$	$t = 3$	$t = 4$
3→2→1	4→3→2		
4→3→1			
4→2→1			
$Backward(t)$: $s \to k \to t$, $k > s$			
$t = 1$	$t = 2$	$t = 3$	$t = 4$
3→4→1	3→4→2	2→4→3	2→3→4
2→4→1	1→3→2	1→2→3	1→2→4
2→3→1	1→4→2	1→4→3	1→3→4

Carré gave an algebraic proof of the correctness and convergence of his algorithm. He also gave a graphical interpretation of his algorithm in [64].

Carré's algorithm has several good properties compared with other algebraic APSP algorithms. First, it performs the least number of triple comparisons, $n(n-1)(n-2)$, for complete graphs as shown by Nakamori [251]. The Floyd-Warshall algorithm performs the same number of triple comparisons but uses a different ordering. Second, it decomposes the APSP into n shortest path trees, one for each sink node. Therefore, it can save many operations depending on the number of sink nodes in the MPSP problem. The Floyd-Warshall algorithm, on the other hand, has to do triple comparisons over the entire distance matrix.

3.4.1.2 Direct methods vs. iterative methods

Sparsity and stability are the two major concerns in solving linear systems. It is a well-known fact in numerical algebra that iterative methods usually converge faster but are more numerically unstable (i.e. affected by rounding errors) than direct methods. However, the operators in path algebra only involve comparison and addition; thus the numerical instability is not a concern in solving Bellman's equations in path algebra. This may explain why iterative methods such as label-correcting methods, instead of direct methods such as Floyd-Warshall or Carré's algorithms, are still the most popular methods used to solve either SSSP or APSP problems.

In this thesis, we investigate ways of improving the direct methods to exploit sparsity and make them competitive with the popular label-correcting methods.

3.4.2 Methods of matrix multiplication

As derived in equation (3.7), Shimbel [291] suggests a naive algorithm using $\log(n)$ matrix squarings of $(I_n \oplus C)$ to solve the APSP problem. To avoid many distance matrix squarings, some $O(n^3)$ distance matrix multiplication methods such as the *revised matrix* [171, 306] and *cascade* [108, 211, 306] algorithms perform only two or three successive distance matrix squarings. However, Farbey et al. [108] show that these methods are still inferior to the Floyd-Warshall algorithm which only needs a single distance matrix squaring procedure.

Aho et al. (see [2], pp.202-206) show that computing $(I_n \oplus C)^{n-1}$ is as hard as a single distance matrix squaring, which takes $O(n^3)$ time. Fredman [122] proposes an $O(n^{2.5})$ algorithm to compute a single distance matrix squaring, but it requires a program of exponential size. Its practical implementation, improved by Takaoka [293], still takes $O(n^3((\log\log n)/\log n)^{\frac{1}{2}})$ which is just slightly better. Recently, much work has been done in using block decomposition and fast matrix multiplication techniques to solve the APSP problem. These new methods, although they have better subcubic time bounds, usually require the arc lengths to be either integers of small absolute value [12, 294, 313] or can only be applied to unweighted, undirected graphs [284, 126, 125]. All of these matrix multiplication algorithms seem to be more suitable for dense graphs since they do not exploit sparsity. Their practical efficiency remains to be evaluated.

3.4.3 Computational experiments on APSP algorithms

To date, no thorough computational analysis on solving the APSP problem has been reported. It is generally believed, however, that the algebraic algorithms require more computational storage and are more suitable for dense graphs, but are unattractive for graphs with large size or sparse structure (as is the case in most real world applications).

None of those matrix-multiplication algorithms has been implemented, despite their better complexity. The Floyd-Warshall algorithm is the most common algebraic APSP algorithm, but it can not take advantage of sparse graphs. On the other hand, Gaussian elimination is shown to be more advantageous than Gauss-Jordan elimination in dealing with sparse matrix [22]. This suggests that a sparse implementation of Carré's algorithm, rather than a sparse Floyd-Warshall algorithm, may be practically attractive.

Goto et al. [150] implement Carré's algorithm [65] sparsely using code generation techniques. The problem they faced is similar to ours. In particular, for a graph with fixed topology, shortest paths between all pairs must be repeatedly computed with different numerical values of arc lengths. To take advantage of the fixed sparse structure, they used a preprocessing procedure which first identifies a good node pivoting order so that the fill-ins in the LU decomposition phase are decreased. They then run Carré's algorithm once to record all the nontrivial triple comparisons in the LU decomposition, forward elimination and backward substitution phases. Based on the triple comparisons recorded in the preprocessing procedure, they generate an ad hoc shortest path code specifically for the original problem. Excluding the time spent in the preprocessing and code-compiling phases, this ad hoc code seems to perform very well, up to several times faster than other SSSP algorithms they tested, on the randomly generated grid graphs.

In fact, the attractive performance of Goto's implementation may be misleading. First, the largest graph they tested, a 100-node 180-arc grid graph, is not as large as many real-world problems. For larger graphs, the code generated may be too large to be stored or compiled. Second, their experiments were conducted in 1976, since which time many good SSSP algorithms and efficient implementations have been proposed. However, their sparse implementations intrigued us and convinced us to further investigate ways of improving algebraic shortest path algorithms for solving shortest paths between multiple node pairs.

3.5 Multiple pairs shortest path algorithms

The ODMCNF usually contains multiple OD pairs. Thus we will focus on solving the *Multiple Pairs Shortest Path* (MPSP) problem, which is to compute the shortest paths for q specific OD pairs (s_i, t_i), $i = 1 \dots q$. Let $|s|$ ($|t|$) denote the number of distinct source (sink) nodes.

Obviously the MPSP problem can be solved by simply applying an SSSP algorithm $\hat{q}$ times, where $\hat{q} = \min\{|s|, |t|\}$ (we call such methods repeated SSSP algorithms), or by applying an APSP algorithm once and extracting the desired OD entries. Both of these methods can involve more computation than necessary. To cite an extreme example, suppose that we want to obtain shortest paths for n OD pairs, (s_i, t_i), $i = 1 \dots n$, which correspond to a matching. That is, each node appears exactly once in the source and sink node set but not the same time. For this specific example, we must apply an SSSP algorithm exactly n times, which is as hard as solving an APSP problem. Such "overkill" operations may be avoided if we use label-setting methods, since it suffices to terminate once all the destination nodes are permanently labeled. To do this, extra storage and book-keeping procedures are required. On the other hand, using the Floyd-Warshall algorithm, we still need to run until the last iteration to get all n OD entries. Either way we waste some time finding the shortest paths of many unwanted OD pairs.

In this section, we first review related methods appearing in the literature, and then briefly introduce our methods.

3.5.1 Repeated SSSP algorithms

Due to the simplicity and efficiency of the state-of-the-art SSSP algorithms, the MPSP problem is usually solved by repeatedly applying an SSSP algorithm $\hat{q}$ times, once for each source (or sink) node.

Theoretically, label-setting algorithms might be preferred due to their better time bounds. For cases with general arc lengths, we can (see [252, 183, 127]) first use a label-correcting algorithm to obtain a shortest path tree rooted at same node r, and then for each arc (i, j) we transform the original arc length c_{ij} to $c^{\pi}_{ij} = c_{ij} + d_i - d_j$, where d_i denotes the shortest distance from r to i. The nonnegative transformed arc length c^{π}_{ij} corresponds to the *reduced cost* in the LP. We then are able to repeatedly use the label-setting algorithm for the remaining $(\hat{q} - 1)$ SSSP problems.

Although this method has better theoretical time bounds, as argued in section 3.3.1, there is no absolute empirical superiority between label-setting and label-correcting methods. Therefore, whether it pays to do the extra arc length transformation, or simply apply a label-correcting method $\hat{q}$ times, is still quite debatable.

3.5.2 Reoptimization algorithms

The concept of reoptimization can be useful in two aspects: (1) when a MPSP problem must be solved with different arc lengths, and (2) when a shortest path tree must be calculated based on a previous shortest path tree.

After we have obtained a shortest path tree, suppose there are k arcs whose lengths have been changed. For cases involving only a decreasing-length-modification of some arcs, Goto and Sangiovanni-Vincentelli [151] proposed an $O(kn^2)$ algebraic algorithm that resembles the Householder formula for inverting modified matrices. Fujishige [123] gave another algorithm based on Dijkstra's algorithm which requires less initial storage than Goto's, but can not deal with cases that have negative arc lengths. For more general cases in which some arc lengths might increase, only LP algorithms such as primal network simplex methods have been proposed to attack such problems.

LP reoptimization techniques may also be advantageous for solving the MPSP problem.

For example, suppose we have obtained a shortest path tree rooted at r, and now try to obtain the next shortest path tree rooted at s. Dual feasibility will be maintained when switching roots, since the optimality conditions guarantee that the reduced cost, c_{ij}^{π}, remains nonnegative for each arc (i, j). We may thus use the current shortest path tree as an advanced basis to start with, and apply any dual method (e.g. the dual simplex method by Florian et al.[112], or the dual ascent method by Nguyen et al. [253]) to solve the newly rooted SSSP problem. The same problem may be solved by adding an artificial arc with long length from s to r while deleting the previous tree arc pointing toward s. Then, a primal network simplex method can be applied since primal feasibility is maintained.

These reoptimization methods exploit the advanced initial basis obtained from the previous iteration. Also, if the new root s is close to the old root r, many of the previous shortest path tree arcs might remain in the newly rooted shortest path tree.

Florian et al. [112] reported up to a 50% running time reduction in several test problems when compared with Dial's and Pape's algorithms (implemented in [94]). More recently, Burton [58] reported that Florian's algorithm runs faster than Johnson's algorithm (a repeated SSSP algorithm, see [183]) *only* when s is reachable by at most two arcs from r. In all other cases Johnson's algorithm outperforms Florian's.

More computational analysis of the MPSP problem is still required to draw a conclusion on the empirical dominance between these reoptimization algorithms and the repeated state-of-the-art SSSP algorithms.

3.5.3 Proposed new MPSP algorithms

It is easy to see that repeated SSSP algorithms are more efficient for MPSP problems with few sources (i.e. $\hat{q} \ll n$). For cases with many sources, are the repeated SSSP algorithms still superior in general? Can we modify an APSP algorithm so that it runs faster on MPSP problems, is competitive in solving MPSP problems on sparse graphs, and takes advantage of special properties of MPSP problems such as fixed topology or requested OD pairs? Chapter 4 and Chapter 5 try to answer these questions.

We propose two new algebraic shortest path algorithms based on Carré's algorithm

[64, 65] in Chapter 4. Like other algebraic APSP algorithms, our algorithms can deal with negative arc lengths. Even better, our algorithms can save half of the storage and running time for undirected or acyclic graphs, and avoid unnecessary operations that other algebraic APSP algorithms must do when solving MPSP problems.

To verify their empirical performance, in Chapter 5, we compare our codes with the state-of-the-art SSSP codes written by Cherkassky et al. [74] on artificial networks from four random network generators.

3.6 On least squares primal-dual shortest path algorithms

The *least squares primal-dual* (LSPD) algorithm [28] is a primal-dual algorithm for solving LPs. Instead of minimizing the sum of the absolute infeasibility to the constraints when solving the restricted primal problem (RPP) as does the conventional primal-dual algorithm, LSPD tries to minimize the sum of the squares of the infeasibility.

In particular, Leichner et al. [216] propose a LP phase I algorithm that strictly improves the infeasibility in each iteration in order to solve the following feasibility problem:

$$Ex = b, \; x \geq 0.$$

In stead of using the conventional simplex phase I method that solves

$$\begin{aligned} &\min |b - Ex| \\ &s.t. \;\; x \geq 0, \end{aligned}$$

they seek solutions to a special case of the *bounded least-squares problem* (BLS) called the *non-negative least-squares problem* (NNLS) formulated as follows:

$$\begin{aligned} &\min \|b - Ex\|^2 \\ &s.t. \;\; x \geq 0. \end{aligned} \tag{NNLS}$$

Similar algorithms have been proposed by Dantzig [86], Björck [54, 55], Van de Panne and Whinston [301], Lawson and Hanson [214], and Goldfarb and Idanani [147]. Recently, Gopalakrishnan et al. [149, 28, 30, 29] give variants of LSPD algorithms to solve several classes of LP and network problems. Since the algorithm is impervious to degeneracy, it is considered to be efficient in solving highly degenerate problems such as assignment problems. Their computational experiments show that LSPD terminates in much fewer iterations than CPLEX network simplex and Hungarian method in solving the assignment problems.

The bottleneck of LSPD lies in solving NNLS efficiently, which in turn depends on solving a *least-squares problem* (LS) efficiently. When LSPD is applied to solve SCNF problems [149, 30], special properties of the node-arc incidence matrix can be exploited to derive a special combinatorial implementation which solves LS very efficiently. In this section we will give a specialized LSPD algorithm to solve the ALL-1 and 1-1 shortest paths on networks with nonnegative arc lengths, and discuss its connection to the original primal-dual algorithm and the well-known Dijkstra's algorithm.

3.6.1 LSPD algorithm for the ALL-1 shortest path problem

Given an initial feasible dual solution π (we can use $\pi = 0$ because $c_{ij} \geq 0$) for the dual problem ALL-1-Dual in Section 3.3.2, first we identify those arcs $(i,j) \in A$ satisfying $\pi_i - \pi_j = c_{ij}$, which we call *admissible arcs*. Let $\hat{A}$ be the set that contains all the admissible arcs. Let $\hat{G} = (N, \hat{A})$ denote the *admissible graph*, which contains all the nodes in N but only arcs in $\hat{A}$. By defining $c^{\pi}_{ij} = c_{ij} - \pi_i + \pi_j$ as the *reduced cost* for arc (i,j), the admissible arc set $\hat{A}$ contains all the arcs $(i,j) \in A$ such that $c^{\pi}_{ij} = 0$. We call a node i an *admissible node* if there exists a path from i to t in $\hat{G}$. Assuming the sink t is initially admissible, the admissible node set $\hat{N}$ is the connected component of $\hat{G}$ that contains t.

We can solve the restricted primal problem (RPP), which seeks the flow assignment x^* on admissible graph $\hat{G}$ that minimizes the sum of squares of the node imbalance (or slackness vector) $\delta = b - \hat{E}x_{\hat{A}}$:

$$\min \sum_{i \in N \setminus t} \delta_i^2 = \sum_{i \in N \setminus t} (b_i - \hat{E}_{i\cdot} x_{\hat{A}})^2 \tag{3.8}$$

$$s.t. \;\; x_a \geq 0 \;\forall\; a \in \hat{A},$$

where $\hat{E}$ is the column subset of the node-arc incidence matrix (with the row corresponding to node t removed) that corresponds to the admissible arcs $\hat{A}$. All the non-admissible arcs have zero flows.

Problem (3.8) is a NNLS problem and can be solved by the algorithm of Leichner et al. [216]. The optimal imbalance δ^* can be used as a dual improving direction to improve π (see Gopalakrishnan et al. [149, 28] for the proof) in the LSPD algorithm. Here we give a special implementation (Algorithm 2) of Gopalakrishnan et al.'s algorithm, *LSPD-ALL-1*, for solving the ALL-1 shortest path problem. It contains a procedure *NNLS-ALL-1* to solve the RPP (3.8).

Algorithm 2 LSPD-ALL-1

```
begin
    Initialize: ∀ node i, π_i := 0 ; δ*_t := 0; add node t to N̂;
                Identify admissible arc set Â and admissible node set N̂;
    while |N̂| < n do
        δ* = NNLS-ALL-1(Ĝ, N̂);
        θ = min_{(i,j)∈A, δ*_i > δ*_j} { (c_ij − π_i + π_j) / (δ*_i − δ*_j) }; π = π + θδ*;
        Identify admissible arc set Â and admissible node set N̂;
end
```

$$\theta = \min_{(i,j)\in A,\delta_i^*>\delta_j^*}\left\{\frac{c_{ij}-\pi_i+\pi_j}{\delta_i^*-\delta_j^*}\right\};\ \pi=\pi+\theta\delta^*;$$

Procedure LSPD-ALL-1$(\hat{\mathbf{G}}, \hat{\mathbf{N}})$

```
begin
    for i = 1 to n do
        if node i ∈ N̂ then
            δ*_i = 0;
        else
            δ*_i = 1;
    return δ*;
end
```

Applying the algorithm *LSPD-ALL-1*, we obtain the following observations:

1. $\delta_i^* = 0, \forall i \in \hat{N}$ and $\delta_i^* = 1, \forall i \in N \setminus \hat{N}$.

2. Let $\hat{N}^k$ denote the admissible nodes set obtained in the beginning of iteration k, then $\hat{N}^k \subseteq \hat{N}^{k+1}$ and $\left|\hat{N}^{k+1}\right| \geq \left|\hat{N}^k\right| + 1$

3. In at most $n-1$ major iterations, the algorithm *LSPD-ALL-1* terminates with $\hat{N} = N$.

Now we show that algorithm *LSPD-ALL-1* will correctly compute an ALL-1 shortest tree.

Theorem 3.1. *The δ^* computed by the procedure* NNLS-ALL-1 *solves problem (3.8).*

Proof. Because no non-admissible node has a path of admissible arcs to t, no non-admissible node can ship any of its imbalance (initialized as $\delta_i^* = 1\ \forall i \in N \setminus t$) to t via admissible arcs, and thus its optimal imbalance remains 1. On the other hand, each admissible node can always ship its imbalance to t via uncapacitated admissible arcs so that its optimal imbalance becomes zero. Therefore the δ^* computed by procedure *NNLS-ALL-1* corresponds to the optimal imbalance δ_i^* for the RPP (3.8). □

Lemma 3.1. *Algorithm* LSPD-ALL-1 *solves the ALL-1 shortest path problem.*

Proof. Algorithm *LSPD-ALL-1* is a specialized LSPD algorithm for ALL-1 shortest path problem. By Theorem 3.1, the δ^* solves quadratic RPP (3.8). δ^* is a dual ascent direction [149, 28]. The algorithm *LSPD-ALL-1* iteratively computes the step length θ to update dual variables π, identifies admissible arcs (i.e., columns), and solves the quadratic RPP (3.8) until $\sum_{i \in N \setminus t} \delta_i^{*2}$ vanishes, which means the primal feasibility is attained. Since the dual feasibility and complementary slackness conditions are maintained during the whole procedure, *LSPD-ALL-1* solves the ALL-1 shortest path problem. □

Now we compare algorithm *LSPD-ALL-1* with the original primal-dual algorithm for solving the ALL-1 shortest path problem.

3.6.2 LSPD vs. original PD algorithm for the ALL-1 shortest path problem

The only difference between algorithm LSPD and the original PD algorithm is that they solve different RPP. The original PD algorithm solves the following RPP:

$$\min \sum_{i \in N \setminus t} \delta_i \qquad \text{(RPP-ALL-1)}$$

$$s.t.\quad \hat{E}_{i\cdot} x_{\hat{A}} + \delta_i = 1,\ i \in N \setminus t$$

$$x_{\hat{A}}, s \geq 0$$

whose dual is

$$\max \sum_{i \in N \setminus t} \rho_i \qquad \text{(DRPP-ALL-1)}$$

$$s.t. \quad \rho_i \leq \rho_j, \, \forall (i,j) \in \hat{A}, i,j \neq t$$

$$\rho_i \leq 0, \, \forall (i,t) \in \hat{A}$$

$$\rho_j \geq 0, \, \forall (t,j) \in \hat{A}$$

$$\rho_i \leq 1, \, \forall i \in N \setminus t$$

The optimal dual solution ρ^* of DRPP-ALL-1 will be used as a dual-ascent direction in the PD process. It is easy to observe that $\rho_i^* = 1$ for each node i that can not reach t along admissible arcs in $\hat{A}$ (i.e. i is non-admissible). Also, if node i is admissible, that is, there exists a path from i to t with intermediate nodes $\{i_1, i_2, i_3, ..., i_k\}$, then $\rho_{i_1}^* = \rho_{i_2}^* = \rho_{i_3}^* = ... = \rho_{i_k}^* = 0$. In other words, the original PD algorithm will have $\rho^* = 0$ for all the admissible node, and $\rho^* = 1$ for all the non-admissible nodes. Thus the improving direction ρ^* obtained by the original PD algorithm is identical to the one obtained by LSPD introduced in previous section.

Therefore we can say that algorithm LSPD and the original PD algorithm are identical to each other in solving the ALL-1 shortest path problem since they produce the same improving direction and step length and construct the same restricted network $\hat{G}$.

Next we will compare these two algorithms with the famous Dijkstra's algorithm.

3.6.3 LSPD vs. Dijkstra's algorithm for the ALL-1 shortest path problem

First we review the Dijkstra's algorithm. For our convenience, we construct a new graph $G'' = (N, A'')$ by reversing all the arc direction of A so that the original ALL-1 shortest path problem on G to sink t becomes a 1-ALL shortest problem from source t with nonnegative arc length on G''. Initialize a node set V as empty and its complement $\overline{V}$ as the whole node set N. The distance label for each node i, denoted as $d(i)$, represents the distance from t to i in G''. Define $pred(j) = i$ if node i is the predecessor of node j.

We say a node is *permanently labeled* if it is put into V. A node is *labeled* if its distance label is finite. A node is *temporarily labeled* if it is labeled but not permanently labeled.

Algorithm 3 Dijkstra($\mathbf{G}''$)

begin

Initialize: $\forall$ node $i \in N \setminus t$, $d(i) := \infty$, $pred(i) = -1$;
$d(t) := 0$, $pred(t) := 0$; $V := \emptyset, \overline{V} := N$;

while $|V| < n$ ***do***

let $i \in \overline{V}$ be a node such that $d(i) = \min\{d(j) : j \in \overline{V}\}$

$V := V \cup \{i\}; \overline{V} := \overline{V} \setminus \{i\}$

for each $(i, j) \in A$ ***do***

if $d(j) > d(i) + c_{ij}$ **then**

$d(j) := d(i) + c_{ij}$; $pred(j) := i$;

end

Dijkstra's algorithm starts by labeling t, and then iteratively labels temporary nodes with arcs from permanently labeled nodes. This is identical to the *LSPD-ALL-1* which grows admissible nodes only from admissible nodes. In fact, in every major iteration, the set of admissible nodes in *LSPD-ALL-1* is the same as the set of permanently labeled nodes in Dijkstra. To show this, we only need to show that both algorithms will choose the same nodes in every major iteration.

Theorem 3.2. *Both algorithm Dijkstra and* LSPD-ALL-1 *choose the same node to become permanently labeled (in Dijkstra) or admissible (in LSPD-ALL-1) in each major iteration.*

Proof. We already know that both algorithms start at the same node t. In *LSPD-ALL-1*, we will identify an admissible node j_1 by identifying the admissible arc (j_1, t) such that $(j_1, t) = \arg \min\limits_{(j,t) \in A, s_j^* > s_t^*} \{\frac{c_{jt} - \pi_i + \pi_j}{s_j^* - s_t^*}\} = \arg \min\limits_{(j,t) \in A} \{c_{jt}\}$. The second equality holds because $s_j^* = 1$, $s_t^* = 0$, and $\pi = 0$ in the first iteration. This is the same as Dijkstra's algorithm in the first iteration.

Assume both algorithms have the same set of admissible (or permanently labeled) nodes V^k in the beginning of the k^{th} major iteration. Algorithm *LSPD-ALL-1* will choose an admissible arc (i_k, j_r) such that $(i_k, j_r) = \arg \min\limits_{(i,j) \in A, \delta_i^* > s_j^*} \{\frac{c_{ij} - \pi_i + \pi_j}{\delta_i^* - s_j^*}\} = \arg \min\limits_{(i,j) \in A, j \in V^k, i \notin V^k} \{c_{ij} + \pi_j\}$. Again, the second equality holds because $s_j^* = 0$ for each $j \in V^k$, and $\delta_i^* = 1$, $\pi_i = 0$ for each $i \notin V^k$. If node j is admissible in the k^{th} iteration, let $j \to j_p \to ... \to j_2 \to j_1 \to t$ denote the path from j to t. Then we can calculate $\pi_j = c_{jj_p} + ... + c_{j_2 j_1} + c_{j_1 t}$ since $\pi_t = 0$ and all the arcs along this path are admissible thus having zero reduced cost. So, $(i_k, j_r) =$

$\arg \min_{(i,j)\in A, j\in V^k, i\notin V^k} \{c_{ij} + \pi_j\} = \arg \min_{(i,j)\in A, j\in V^k, i\notin V^k} \{ \sum_{(p,q)\in path\{i\to j\to j_p\to \ldots \to j_1 \to t\}} c_{pq}\}$. Therefore, in the beginning of the $(k+1)^{st}$ major iteration, node i_k becomes admissible with $\pi_{i_k} = \sum_{(p,q)\in path\{i_k\to j_r\to j_{r-1}\to \ldots \to j_1 \to t\}} c_{pq}$.

Dijkstra's algorithm in the k^{th} iteration will choose the node reachable from V^k with the minimum distance label. That is, choose node i_k reachable from a permanent labeled node j_r such that $d(i_k) = \min_{(j,i)\in A, j\in V^k} d(i) = \min_{(j,i)\in A, j\in V^k} \{d(j) + c_{ji}\}$. Let $t \to j_1 \to j_2 \to \ldots \to j_p \to j$ denote the path from t to a permanently labeled node j. Since j is permanently labeled, $d(j) = \sum_{(p,q)\in path\{j\to j_p\to \ldots \to j_2\to j_1 \to t\}} c_{pq}$. Therefore, node i_k will become permanently labeled in the $(k+1)^{st}$ major iteration with distance label $d(i_k) = \sum_{(p,q)\in path\{t\to j_1\to j_2\to \ldots \to j_r \to i_k\}} c_{pq}$.

Therefore, these two algorithms perform the same operation to get the same shortest distance label for the newly permanently labeled (or admissible) node. $\square$

From these discussion, we conclude that when solving the ALL-1 shortest path problem with nonnegative arc length, all the three algorithms, Dijkstra, *LSPD-ALL-1*, and the original PD algorithm, will perform the same operations in each iteration. In fact, this result is due to the nondegeneracy of the problem structure. Remember that the basis corresponds to a spanning tree in the network problem. In this ALL-1 shortest path problem, each node other than t has supply 1 to send to t. In each iteration of these algorithms, the primal infeasibility will strictly decrease, hence each pivot is always nondegenerate.

LSPD is an algorithm designed to take advantage of doing nondegenerate pivots in each iteration. Therefore, in this special case, it just performs as efficiently as the other two algorithms. Next we will see that because the 1-1 shortest path problem does not have the nondegenerate property, thus *LSPD-1-1* does do a better job than the original PD algorithm in some sense.

3.6.4 LP formulation for the 1-1 shortest path problem

Unlike the ALL-1 shortest path problem which searches for a shortest path tree (SPT), the 1-1 shortest path problem only needs part of the SPT, namely, the shortest path between certain two nodes, s and t. It can be viewed as sending a unit flow from s to t with the minimal cost via uncapacitated arcs. Its linear programming formulation is similar to the

ALL-1 formulation in Section 3.3.2 except now the node imbalance vector b only has two nonzero entries: $+1$ for s, -1 for t, and 0 for all the other nodes.

Since the node-arc incidence matrix has one redundant row, we remove the row corresponding to t, to get the following primal and dual formulations:

$$\min cx = Z^{P*}_{1-1}(x) \tag{1-1-Primal}$$

$$s.t.\ \ \overline{N}x = \begin{cases} 1 & \text{, if } i = s \\ 0 & \text{, if } i \in N \setminus \{s,t\} \end{cases} \quad , i \in N \setminus t \tag{3.9}$$

$$x \geq 0,$$

whose dual is

$$\max \pi_s = Z^{D*}_{1-1}(\pi) \tag{1-1-Dual}$$

$$s.t.\ \ \pi_i - \pi_j \leq c_{ij}\ \forall (i,j) \in A, i,j \neq t \tag{3.10}$$

$$\pi_i \leq c_{it}\ \forall (i,t) \in A \tag{3.11}$$

$$-\pi_j \leq c_{tj}\ \forall (t,j) \in A \tag{3.12}$$

Here the right-hand-side of (3.9) only has one nonzero entry ($+1$ for node s). This makes the dual objective $Z^{D*}_{1-1}(\pi)$ differ from that of ALL-1, $Z^{D*}_{ALL-1}(\pi)$, in which $Z^{D*}_{1-1}(\pi)$ maximizes only π_s while $Z^{D*}_{ALL-1}(\pi)$ maximize the sum $\sum\limits_{i \in N \setminus t} \pi_i$. Therefore, we give a new procedure *NNLS-1-1* to solve the nonnegative least-squares problem in our 1-1 shortest path algorithm, *LSPD-1-1*. We will also illustrate the difference between solving the ALL-1 and 1-1 shortest path problem when the original PD algorithm is applied. Finally we will explain the connections between the Dijkstra, *LSPD-1-1*, and original PD algorithms when they are used to solve the 1-1 shortest path problem.

3.6.5 LSPD algorithm for the 1-1 shortest path problem

Before we give the new *LSPD-1-1* shortest path algorithm, the *admissible node* set $\hat{N}$ has to be redefined as follows: a node i is *admissible* if it is reachable from s only via admissible arcs. All the other definitions such as admissible arcs $\hat{A}$ and the admissible graph $\hat{G}$ remain the same as in Section 3.6.1. With a procedure *NNLS-1-1* that solves the RPP (3.8), the algorithm *LSPD-1-1* is shown below as Algorithm 4.

Algorithm 4 LSPD-1-1

begin

Initialize: $\forall$ node i, $\pi_i := 0$; $\delta_s^* := 0$; add node s to $\hat{N}$;

Identify admissible arc set $\hat{A}$ and admissible node set $\hat{N}$;

while node $t \notin \hat{N}$ ***do***

$\delta^* = NNLS\text{-}1\text{-}1(\hat{G}, \hat{N})$;

$\theta = \min_{(i,j)\in A, \delta_i^* > \delta_j^*} \{\frac{c_{ij} - \pi_i + \pi_j}{\delta_i^* - \delta_j^*}\}$; $\pi = \pi + \theta\delta^*$;

Identify admissible arc set $\hat{A}$ and admissible node set $\hat{N}$;

end

Procedure LSPD-1-1$(\hat{\mathbf{G}}, \hat{\mathbf{N}})$

begin

for $i = 1$ to n ***do***

if node $i \in \hat{N}$ ***then***

$\delta_i^* = \frac{1}{|\hat{N}|}$;

else

$\delta_i^* = 0$;

return δ^*;

end

Applying algorithm *LSPD-1-1*, we obtain the following observations:

1. $\delta_i^* = \frac{1}{|\hat{N}|}$, $\forall i \in \hat{N}$ and $\delta_i^* = 0$, $\forall i \in N \setminus \hat{N}$.

2. Let $\hat{N}^k$ denote the admissible nodes set obtained in the beginning of iteration k. Then $\hat{N}^k \subseteq \hat{N}^{k+1}$ and $\left|\hat{N}^{k+1}\right| \geq \left|\hat{N}^k\right| + 1$.

3. In at most $n-1$ major iterations, the algorithm *LSPD-1-1* terminates when node t becomes admissible. Then, s can send its unit imbalance to t via some path composed only by admissible arcs so that the total imbalance over all nodes becomes 0.

Now we show that algorithm *LSPD-1-1* will correctly the shortest path from s to t.

Theorem 3.3. *The δ^* computed by the procedure* NNLS-1-1 *solves problem (3.8).*

Proof. The RPP (3.8) is a quadratic programing problem. If we relax the nonnegativity constraints, it is a least-squares problem which can be solved by solving the normal equation $\hat{E}^T\hat{E}x^* = \hat{E}^T b$. In other words, $\hat{E}^T\delta^* = \hat{E}^T(b - \hat{E}x^*) = 0$. Note that each row of $\hat{E}^T$ contains only two nonzero entries (i.e., $+1$ and -1) which represent an admissible arc. In other words, $\hat{E}^T\delta^* = 0$ implies $\delta^*_i = s^*_j$ for each admissible arc (i,j) which implies all admissible nodes have the same optimal imbalance δ^*. Since the total system imbalance is 1 (from the source s), the optimal least-squares solution δ^*_i for the RPP (3.8) will be $\frac{1}{|\hat{N}|}$ for each admissible node i. Using the optimal imbalance δ^*, it is easy to compute the unique optimal arc flow x^* and verify that $x^* \geq 0$ by traversing nodes on the component that contains the source node s (For more details in application of LSPD on network problems, see [149, 30].). Thus the optimal imbalance δ^* by the procedure *NNLS-1-1* solves (3.8) $\square$

Lemma 3.2. *Algorithm* LSPD-1-1 *solves the 1-1 shortest path problem from s to t.*

Proof. Algorithm *LSPD-1-1* is a specialized LSPD algorithm for 1-1 shortest path problem. By Theorem 3.3, the δ^* solves quadratic RPP (3.8). δ^* is a dual ascent direction [149, 28]. The algorithm *LSPD-1-1* iteratively computes the step length θ to update dual variables π, identifies admissible arcs (i.e., columns), and solves the quadratic RPP (3.8) until $\sum\limits_{i \in N \setminus t} \delta^{*2}_i$ vanishes, which means the primal feasibility is attained. Since the dual feasibility and complementary slackness conditions are maintained during the whole procedure, *LSPD-1-1* solves the 1-1 shortest path problem from s to t. $\square$

Intuitively, we can view this algorithm as the following: starting from the source s, *LSPD-1-1* tries to reach t by growing the set of admissible nodes. The algorithm keeps propagating the unit imbalance along all the admissible arcs so that the unit imbalance will be equally distributed to each admissible node before t becomes admissible. Once t becomes admissible, all the imbalance flows to t so that the optimal system imbalance δ^* becomes 0. Then the algorithm is finished.

To further speed up algorithm *LSPD-1-1*, we observe that for each admissible node k, $s^*_k = \frac{1}{|\hat{N}|}$ and $\theta s^*_k = \min\limits_{(i,j)\in A, \delta^*_i > s^*_j} \{\frac{c_{ij} - \pi_i + \pi_j}{\delta^*_i - s^*_j}\} \cdot \frac{1}{|\hat{N}|} = \min\limits_{(i,j)\in A, \delta^*_i = \frac{1}{|\hat{N}|}, s^*_j = 0} \{\frac{c_{ij} - \pi_i + \pi_j}{\frac{1}{|\hat{N}|}}\} \cdot \frac{1}{|\hat{N}|} = \min\limits_{(i,j)\in A, \delta^*_i > s^*_j} \{c_{ij} - \pi_i + \pi_j\} \cdot 1$. Thus we can speed up the algorithm *LSPD-1-1* using $\hat{\theta} = \min\limits_{(i,j)\in A, \delta^*_i > s^*_j} \{c_{ij} - \pi_i + \pi_j\}$ and $\hat{s}_k = 1$ instead of the original θ and s^*_k. This modification will not affect the selection of new admissible arc and node; thus the algorithm achieves the same objective using simpler computations.

For our convenience, we will use this modified version of the *LSPD-1-1* algorithm in later sections.

3.6.6 LSPD vs. original PD algorithm for the 1-1 shortest path problem

When the original PD algorithm solves the 1-1 shortest path problem, the primal RPP formulation is as follows:

$$\min \sum_{i \in N \setminus t} \delta_i \tag{RPP-1-1}$$

$$s.t. \quad \hat{E}_{i \cdot} x_{\hat{A}} + \delta_i = \begin{cases} 1 & \text{, if } i = s \\ 0 & \text{, if } i \in N \setminus \{s,t\} \end{cases} \quad , i \in N \setminus t$$

$$x_{\hat{A}}, s \geq 0$$

whose dual is

$$\max \rho_s \tag{DRPP-1-1}$$

$$s.t. \quad \rho_i \leq \rho_j, \; \forall (i,j) \in \hat{A}, i,j \neq t$$

$$\rho_i \leq 0, \; \forall (i,t) \in \hat{A}$$

$$\rho_j \geq 0, \; \forall (t,j) \in \hat{A}$$

$$\rho_i \leq 1, \; \forall i \in N \setminus t$$

Unlike when solving the ALL-1 shortest path problem, the original PD algorithm will have degenerate pivots when solving RPP-1-1, which is a major difference from the LSPD algorithm since the LSPD algorithm guarantees nondegenerate pivots at every iteration.

If s and t are not adjacent and all the arc costs are strictly positive, we start the algorithm with $\pi = 0$ which makes $\hat{A}$ empty in the first iteration. Then the optimal solution for DRPP-1-1 in the first iteration will be $\rho_s^* = 1$, $\rho_i^* \leq 1 \; \forall i \in N \setminus \{s,t\}$. That is, we are free to choose any ρ_i^* as long as it does not exceed 1. This property of multiple optimal dual solutions is due to the degeneracy of RPP-1-1. When we have multiple choices to improve the dual solution, there is no guarantee of improving the objective of RPP-1-1 at any iteration. In fact, we may end up cycling or take a long time to move out the degenerate primal solution.

If we are very lucky, by choosing the "right" dual improving direction, we may even solve this problem much faster.

To eliminate the uncertainty caused by primal degeneracy when solving RPP-1-1, we have to choose the dual improving direction in a smart way. One way is to choose $\rho_i^* = 0$ for non-admissible nodes. Then, by the first constraint in DRPP-1-1, admissible nodes will be forced to have $\rho_i^* = 1$. This is because we want to maximize ρ_s, and the best we can do is $\rho_s^* = 1$. By doing so, we force all the nodes reachable from s (i.e., admissible nodes) to have $\rho_i^* = 1$. Then the original PD algorithm chooses the same admissible arcs and nodes as *LSPD-1-1*. Next, we will show that this specific PD algorithm performs the same operations as Dijkstra's algorithm in each iteration.

3.6.7 LSPD vs. Dijkstra's algorithm for the 1-1 shortest path problem

The Dijkstra's algorithm for the 1-1 shortest path problem is the same as the ALL-1 case in Section 3.6.3, except that it terminates as soon as the sink t is permanently labeled. In this section, we show that algorithm *LSPD-1-1* performs the same operations as Dijkstra's algorithm does.

Algorithm *LSPD-1-1* starts at source node s, and then identifies admissible arcs to grow the set of admissible nodes. This is the same as Dijkstra's algorithm. If both algorithms choose the same node in each iteration, the admissible node set $\hat{N}$ in the *LSPD-1-1* algorithm will be equivalent to the permanently labeled node set V in Dijkstra's algorithm.

The following proposition explains that both algorithms choose the same nodes in every major iteration.

Theorem 3.4. *Both Dijkstra and* LSPD-1-1 *choose the same node to become permanently labeled (in Dijkstra) or admissible (in* LSPD-1-1*) in each major iteration.*

Proof. We already know that both algorithms start at s. In *LSPD-1-1*, we will identify an admissible node i_1 by identifying the admissible arc (s, i_1) such that $(s, i_1) = \arg \min\limits_{(s,i)\in A, s_s^* > \delta_i^*} \{\frac{c_{si} - \pi_s + \pi_i}{s_s^* - \delta_i^*}\} = \arg \min\limits_{(s,i)\in A} \{c_{si}\}$.The second equality holds because $s_s^* = 1$, $\delta_i^* = 0$, and $\pi = 0$ in the first iteration. This is the same as Dijkstra's algorithm in the first iteration.

Assume both algorithms have the same set of admissible (or permanently labeled) nodes V^k in the beginning of the k^{th} major iteration. Algorithm *LSPD-1-1* will choose an arc (i_r, j_k) such that $(i_r, j_k) = \arg \min_{(i,j)\in A, \delta_i^* > s_j^*} \{\frac{c_{ij}-\pi_i+\pi_j}{\delta_i^*-s_j^*}\} = \arg \min_{(i,j)\in A, i\in V^k, j\notin V^k} \{c_{ij} - \pi_i\}$. Again, the second equality holds because $\delta_i^* = 1$ for each $i \in V^k$, and $s_j^* = 0$, $\pi_j = 0$ for each $j \notin V^k$ If node i is admissible in the k^{th} iteration, let $s \to i_1 \to i_2 \to ... \to i_p \to i$ denote the path from s to i. Then we can calculate $\pi_s = c_{si_1} + c_{i_1i_2} + ... + c_{i_pi} + \pi_i$ since all the arcs along this path are admissible and thus have zero reduced cost. That is, $-\pi_i = \sum_{(p,q)\in path\{s\to i_1\to i_2 ...\to i\}} c_{pq} - \pi_s$ for each admissible node i. So, (i_r, j_k) $= \arg \min_{(i,j)\in A, i\in V^k, j\notin V^k} \{c_{ij} - \pi_i\} = \arg \min_{(i,j)\in A, i\in V^k, j\notin V^k} \{\sum_{(p,q)\in path\{s\to i_1\to i_2 ...\to i\to j\}} c_{pq} - \pi_s\} =$ $\arg \min_{(i,j)\in A, i\in V^k, j\notin V^k} \{\sum_{(p,q)\in path\{s\to i_1\to i_2 ...\to i\to j\}} c_{pq}\}$. The last equality holds since π_s is fixed when we compare all the arcs $(i,j) \in A, i \in V^k, j \notin V^k$.

Therefore, in the beginning of the $(k+1)^{st}$ major iteration, node j_k becomes admissible with $\pi_{j_k} = 0$, and $\pi_s = \sum_{(p,q)\in path\{s\to i_1\to i_2 ...\to i_r\to j_k\}} c_{pq}$.

The criterion to choose the new admissible arc (i_r, j_k) is the same as Dijkstra's algorithm which we will explain next.

Dijkstra's algorithm in the k^{th} iteration will choose a node reachable from V^k with minimum distance label. That is, it chooses a node j_k reachable from some permanent node i_r such that $d(j_k) = \min_{(i,j)\in A, i\in V^k, j\notin V^k} d(j) = \min_{(i,j)\in A, i\in V^k, j\notin V^k} \{d(i) + c_{ij}\}$. Let $s \to i_1 \to i_2 \to ... \to i_p \to i$ denote the path from s to a permanently labeled node i. Since i is permanently labeled, $d(i) = \sum_{(p,q)\in path\{s\to i_1\to i_2 ...\to i_p\to i\}} c_{pq}$. Therefore, node j_k will become permanently labeled because $d(j_k) = \min_{(i,j)\in A, i\in V^k, j\notin V^k} \{d(i) + c_{ij}\} = \min_{(i,j)\in A, i\in V^k, j\notin V^k}$ $\{\sum_{(p,q)\in path\{s\to i_1\to i_2 ...\to i\to j\}} c_{pq}\}$. That is, node j_k will become permanently labeled in the $(k+1)^{st}$ major iteration and will have distance label $d(j_k) = \sum_{(p,q)\in path\{s\to i_1\to i_2 ...\to i_r\to j_k\}} c_{pq}$.

It is easy to see that these two algorithms perform the same operation to identify the same newly permanently labeled (or admissible) node. $\square$

From this proposition, we observe that in the *LSPD-1-1* algorithm, π_i represents the shortest distance between an admissible node i and the most recent admissible node in V^k, while in Dijkstra's algorithm $d(i)$ represents the shortest distance between s and i. In other words, $d(i) = \pi_s - \pi_i$ for any admissible node i. Therefore, when t becomes admissible, $d(t) = \pi_s$.

Here we use a physical example to illustrate these two algorithms. Consider each node as a ball, and each arc as a string connecting two balls. Here we assume all the arc length is positive.

For the *LSPD-1-1* algorithm, we can think as follows: in the beginning we put all the balls on the floor, and then we pick up ball s and raise it. We keep raising s until the string (s, i_1) becomes tight; then if we raise ball s further more, ball i_1 will be raised. We keep increasing the height of s until finally ball t is raised. At this moment, the height of s (i.e. π_s) is the shortest path between s and t, and the shortest path consists of all arcs on the path to t whose strings are tight.

For Dijkstra's algorithm, we use a plate which has one hole for these n balls to fall through. In the beginning we put the plate on the floor, then put ball s in the hole and all the other balls on the plate. We will put ball s on the floor so that when we raise the plate, s will stay on the floor. Then we begin to raise the plate until the string (s, i_1) is so tight that ball i_1 begins to pass through the hole. We keep raising the plate until finally ball t is about to fall through the hole. At that time, the height of ball t (i.e. $d(t)$) is the shortest path length between s and t, and the shortest path consists of all arcs on the path to t whose strings are tight.

From these discussion, we conclude that when solving the 1-1 shortest path problem with nonnegative arc lengths, Dijkstra and *LSPD-1-1* algorithm are, in fact, identical to each other. The original P-D algorithm will face the problem of primal degeneracy when solving the RPP-1-1. However, if we choose the improving dual direction intelligently (i.e., $\rho^* = 0$ for all non-admissible nodes and $\rho^* = 1$ for all admissible nodes), the original PD algorithm will perform same operations as the *LSPD-1-1* algorithm.

3.6.8 Summary

In summary, the LSPD algorithm is identical to Dijkstra's algorithm for solving both ALL-1 and 1-1 shortest path problems. The original PD algorithm, on the other hand, is identical to the Dijkstra's algorithm for solving the ALL-1 shortest path problem, but needs to choose a specific dual improving direction (there are multiple ones) so that it will be identical to the Dijkstra's algorithm.

For the general arc cost cases, both LSPD and the original PD algorithms can be applied, but Dijkstra's algorithm is not applicable. The theoretical and practical performances of LSPD and the original PD algorithms for solving shortest path problems with general arc costs remain to be investigated.

CHAPTER IV

NEW MULTIPLE PAIRS SHORTEST PATH ALGORITHMS

We have reviewed most of the shortest path algorithms in the literature in Chapter 3. Our purpose is to design new algorithms that can efficiently solve the MPSP problem. In this Chapter, we propose two new MPSP algorithms that exploit ideas from Carré's APSP algorithm (see Section 3.4.1.1).

Section 4.1 introduces some definitions and basic concepts. Section 4.2 presents our first new MPSP algorithm ($DLU1$) and proves its correctness. Section 4.3 gives our second algorithm ($DLU2$) and describes why it is superior to the first one. Section 4.4 summarize our work.

4.1 Preliminaries

Based on the definition and notation introduced in Section 3.2, we define the *up-inward arc adjacency list* denoted $ui(i)$ of a node i to be an array that contains all arcs which point upwards into node i, and *down-inward arc adjacency list* denoted $di(i)$ to be an array of all arcs pointing downwards into node i. Similarly, we define the *up-outward arc adjacency list* denoted $uo(i)$ of a node i to be an array of all arcs pointing upwards out of node i, and *down-outward arc adjacency list* denoted $do(i)$ to be an array of all arcs pointing downwards out of node i.

For convenience, if there are multiple arcs between a node pair (i, j), we choose the shortest one to represent arc (i, j) so that c_{ij} in the measure matrix is unique without ambiguity. If there is no path from i to j, then $x_{ij} = \infty$.

Define an induced subgraph denoted $H(S)$ on the node set S which contains only arcs (i, j) of G with both ends i and j in S. Let $[s, t]$ denote the set of nodes $\{s, (s+1), \ldots, (t-1), t\}$. Figure 4 illustrates examples of $H([1, s] \cup t)$ and $H([s, t])$ which will be used to explain our algorithms later on.

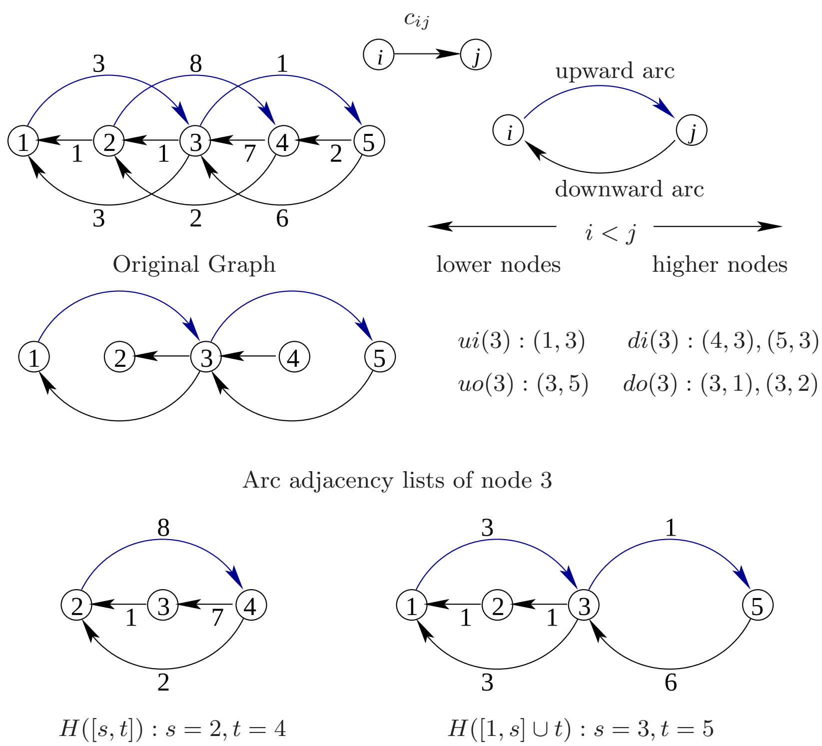


Figure 4: Illustration of arc adjacency lists, and subgraphs $H([2, 4])$, $H([1, 3] \cup 5)$

Carré's algebraic APSP algorithm [64, 65] uses Gaussian elimination to solve $X = CX \oplus I_n$. After a LU decomposition procedure, Carré's algorithm performs n applications of forward elimination and backward substitution procedures. Each forward/backward operation in turn gives an optimal column solution of X which corresponds to an ALL-1 shortest distance vector. This decomposability of Carré's algorithm makes it more attractive than the Floyd-Warshall algorithm for MPSP problems.

Inspired by Carré's algorithm, we propose two algorithms $DLU1$ and $DLU2$ that further reduce computations required for MPSP problems. We use the name DLU for our algorithms since they contain procedures similar to the LU decomposition in Carré's algorithm and are more suitable for dense graphs. Not only can our algorithms decompose a MPSP problem as Carré's algorithm does, they can also compute the requested OD shortest distances without the need of shortest path trees required by other APSP algorithms.

Therefore our algorithms save computational work over other APSP algorithms and are advantageous for problems where only distances (not paths) are requested. For problems that require tracing of shortest path for a particular OD pair (s,t), $DLU1$ solves the shortest path tree rooted at t as Carré's algorithm does, while $DLU2$ can trace the path without the need of computing that shortest path tree.

For sparse graphs, node ordering plays an important role in the efficiency of our algorithms. A bad node ordering will incur more *fill-in arcs* which resemble the fill-ins created in Gaussian elimination. Computing an ordering that minimizes the fill-ins is NP-complete [272]. Nevertheless, many fill-in reducing techniques such as Markowitz's rule [230], minimum degree method, and nested dissection method (see Chapter 8 in [98]) used in solving systems of linear equations can be exploited here as well. Since our algorithms do more computations on higher nodes than lower nodes, optimal distances can be obtained for higher nodes earlier than lower nodes. Thus reordering the requested OD pairs to have higher indices may also shorten the computational time, although such an ordering might incur more fill-in arcs. In general, it is difficult to obtain an optimal node ordering that minimizes the computations required. More details about the impact of node ordering will be discussed in Section 5.2.1. Here, we use a predefined node ordering to start with our algorithms.

The underlying ideas of our algorithms are as follows: Suppose the shortest path in G from s to t contains more than one intermediate node and let r be the highest intermediate node in that shortest path. There are only three cases: (1) $r < \min\{s,t\}$ (2) $\min\{s,t\} < r < \max\{s,t\}$ and (3) $r > \max\{s,t\}$. The first two cases correspond to a shortest path in $H([1, \max\{s,t\}])$, and the last case corresponds to a shortest path in $H([1,r])$ where $r > \max\{s,t\}$. Including the case where the shortest path is a single arc, our algorithms systematically calculate shortest paths for these cases and obtain the shortest path in G from s to t. When solving the APSP problem on a complete graph, our algorithms have the same number of operations as the Floyd-Warshall algorithm does in the worst case. For general MPSP problems, our algorithms beat the Floyd-Warshall algorithm.

4.2 *Algorithm* $DLU1$

Our first shortest path algorithm $DLU1$ reads a set of q OD pairs $Q := \{(s_i, t_i) : i = 1, \ldots, q\}$. We set i_0 to be the index of the lowest origin node in Q, j_0 to be the index of the lowest destination node in Q, and k_0 to be $\min_i\{\max\{s_i, t_i\}\}$. $DLU1$ then computes x^*_{st} for all node pairs (s,t) satisfying $s \geq k_0$, $t \geq j_0$ or $s \geq i_0$, $t \geq k_0$ which covers all the OD pairs in Q. However, the solution does not contain sufficient information to trace their shortest paths, unless i_0 and k_0 are set to be 1 and j_0 respectively in which case $DLU1$ gives shortest path trees rooted at sink node t for each $t = j_0, \ldots, n$.

Algorithm 5 DLU1($\mathbf{Q := \{(s_i, t_i) : i = 1, \ldots, q\}}$)

begin
  Initialize: $\forall$ s, $x_{ss} := 0$ and $succ_{ss} := s$
    $\forall (s,t)$, ***if*** $(s,t) \in A$ ***then***
      $x_{st} := c_{st}$; $succ_{st} := t$
      if $s < t$ ***then*** add arc (s,t) to $uo(s)$ and $ui(t)$
      if $s > t$ ***then*** add arc (s,t) to $do(s)$ and $di(t)$
    else $x_{st} := \infty$; $succ_{st} := 0$
    set $i_0 := \min_i s_i$; $j_0 := \min_i t_i$; $k_0 := \min_i\{\max\{s_i, t_i\}\}$
    if shortest paths need to be traced
    then reset $i_0 := 1$; $k_0 := j_0$
  G_LU;
  Acyclic_L(j_0);
  Acyclic_U(i_0);
  Reverse_LU(i_0, j_0, k_0);
end

In the k^{th} iteration of LU decomposition in Gaussian elimination, we use diagonal entry (k,k) to eliminate entry (k,t) for each $t > k$. This will update the $(n-k)\times(n-k)$ submatrix and create fill-ins. Similarly, Figure 5(a) illustrates the operations of procedure *G_LU* on a 5-node graph. It sequentially uses node 1, 2, and 3 as intermediate nodes to update the remaining 4×4, 3×3, and 2×2 submatrix of $[x_{ij}]$ and $[succ_{ij}]$. *G_LU* computes shortest paths for any node pair (s,t) in $H([1, \min\{s,t\}] \cup \max\{s,t\})$ (as defined in Section 4.1). Note that $x^*_{n,n-1}$ and $x^*_{n-1,n}$ will have been obtained after *G_LU*.

The second procedure, *Acyclic_L*(j_0), computes shortest paths for all node pairs (s,t) such that $s > t \geq j_0$ in $H([1,s])$. Figure 5(b) illustrates the operations of *Acylic_L*(2) which updates the column of each entry (s,t) that satisfies $s > t \geq 2$ in the lower triangular part

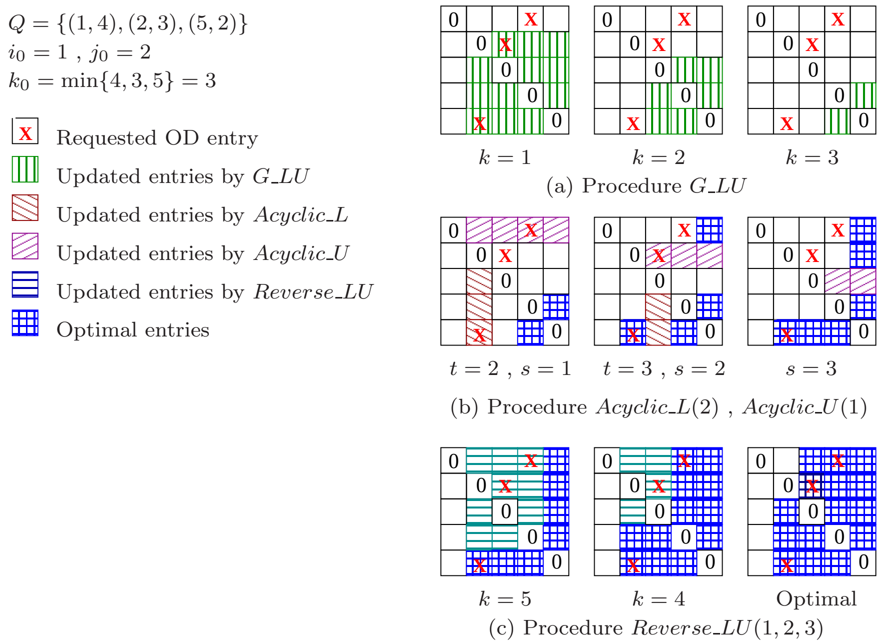


Figure 5: Solving a 3 pairs shortest path problem on a 5-node graph by Algorithm $DLU1(Q)$

of $[x_{ij}]$ and $[succ_{ij}]$ since node 2 is the lowest destination node in Q. Note that x^*_{nj} for all $j \geq j_0$ will have been obtained after $Acyclic\_L(j_0)$.

Similar to the previous procedure, $Acyclic\_U(i_0)$ computes shortest paths for all node pairs (s, t) such that $i_0 \leq s < t$ in $H([1, t])$. Figure 5(b) also illustrates the operations of $Acylic\_U(1)$ which updates the row of each entry (s, t) that satisfies $t > s \geq 1$ in the upper triangular part of $[x_{ij}]$ and $[succ_{ij}]$ since node 1 is the lowest origin node in Q. Note that x^*_{in} for all $i \geq i_0$ will have been obtained after $Acyclic\_U(i_0)$.

By now, for any OD pair (s, t) such that $s \geq i_0$ and $t \geq j_0$ the algorithm will have determined its shortest distance in $H(1, \max\{s, t\})$. The final step, similar to LU decomposition but in a reverse fashion, $Reverse\_LU(i_0, j_0, k_0)$ computes length of the shortest paths in $H([1, r])$ that must pass through node r for each $r = n, \ldots, (\max\{s, t\} + 1)$ from each origin $s \geq k_0$ to each destination $t \geq j_0$ or from each origin $s \geq i_0$ to each destination $t \geq k_0$. The algorithm then compares the x_{st} obtained by the last step with the one obtained previously, and chooses the smaller of the two which corresponds to x^*_{st} in G. Figure 5(c) illustrates the operations of $Reverse\_LU(1, 2, 3)$ which updates each entry (s, t) of $[x_{ij}]$ and $[succ_{ij}]$ that satisfies $1 \leq s < k$, $2 \leq t < k$. In this case, it runs for $k = 5$ and 4 until entry $(2, 3)$ is updated. Note that x^*_{st} for all $s \geq i_0$, $t \geq k_0$ or $s \geq k_0$, $t \geq j_0$ will have been obtained after $Reverse\_LU(i_0, j_0, k_0)$ and thus shortest distances for all the requested OD pairs in Q will have been computed.

To trace shortest paths for all the requested OD pairs, we have to set $i_0 = 1$ and $k_0 = j_0$ in the beginning of the algorithm. If $i_0 > 1$, $Acylic\_U(i_0)$ and $Reverse\_LU(i_0, j_0, k_0)$ will not update $succ_{st}$ for all $s < i_0$. This makes tracing shortest paths for some OD pairs (s, t) difficult if those paths contain intermediate nodes with index lower than i_0. Similarly, if $k_0 > j_0$, $Reverse\_LU(i_0, j_0, k_0)$ will not update $succ_{st}$ for all $t < k_0$. For example, in the last step of Figure 5(c), if node 1 lies in the shortest path from node 5 to node 2, then we may not be able to trace this shortest path since $succ_{12}$ has not been updated in $Reverse\_LU(1, 2, 3)$. Therefore even if Algorithm $DLU1$ gives the shortest distance for the requested OD pairs earlier, tracing these shortest paths requires more computations.

It is easily observed that we can solve any APSP problem by setting $i_0 := 1$, $j_0 := 1$ when applying $DLU1$. For solving general MPSP problems where only shortest distances are requested, $DLU1$ can save more computations without retrieving the shortest path trees, thus makes it more efficient than other algebraic APSP algorithms. More details will be discussed in following sections.

4.2.1 Procedure G_LU

For any node pair (s, t), Procedure G_LU computes shortest path from s to t in $H($ $[1, \min\{s, t\}] \cup \max\{s, t\}$ $)$. The updated x_{st} and $succ_{ij}$ will then be used to determine the shortest distance in $H([1, \max\{s, t\}])$ from s to t by the next two procedures.

Figure 5(a) illustrates the operations of G_LU. It sequentially uses node $k = 1, \ldots, (n-2)$ as intermediate node to update each entry (s, t) of $[x_{ij}]$ and $[succ_{ij}]$ that satisfies $k < s \leq n$ and $k < t \leq n$ as long as $x_{sk} < \infty$, $x_{kt} < \infty$ and $x_{st} > x_{sk} + x_{kt}$.

Thus this procedure is like the LU decomposition in Gaussian elimination. Graphically speaking, it can be viewed as a process of constructing the *augmented graph* G' obtained by either adding fill-in arcs or changing some arc lengths on the original graph when better paths are identified using intermediate nodes whose index is smaller than both end nodes of

Procedure G_LU
begin
    for $k = 1$ to $n-2$ ***do***
        for each arc $(s,k) \in di(k)$ ***do***
            for each arc $(k,t) \in uo(k)$ ***do***
                if $x_{st} > x_{sk} + x_{kt}$
                    if $s = t$ and $x_{sk} + x_{kt} < 0$ ***then***
                        Found a negative cycle; ***STOP***
                    if $x_{st} = \infty$ ***then***
                        if $s > t$ ***then***
                            add a new arc (s,t) to $do(s)$ and $di(t)$
                        if $s < t$ ***then***
                            add a new arc (s,t) to $uo(s)$ and $ui(t)$
                    $x_{st} := x_{sk} + x_{kt}$; $succ_{st} := succ_{sk}$
end

the path. For example, in Figure 6 we add fill-in arc $(2,3)$ because $2 \to 1 \to 3$ is a shorter path than the direct arc from node 2 to node 3 (infinity in this case). We also add arcs $(3,4)$ and $(4,5)$ and modify the length of original arc $(4,3)$.

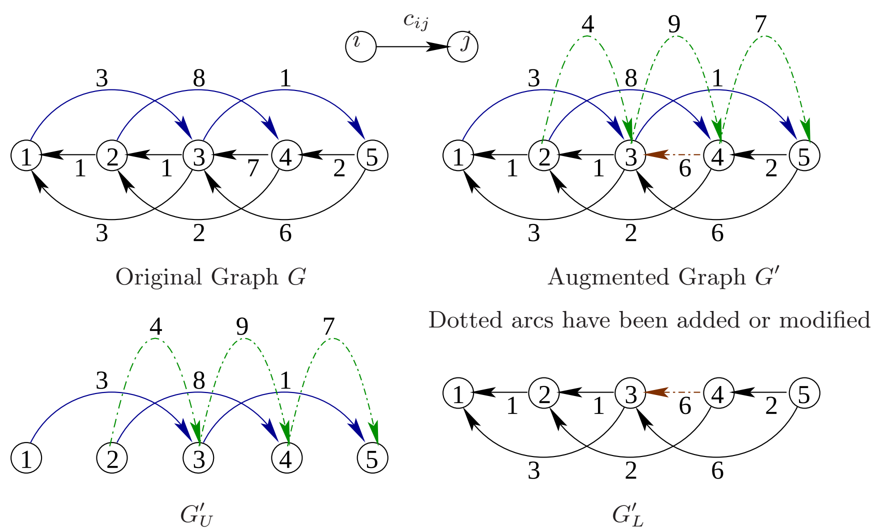


Figure 6: Augmented graph after procedure *G_LU*

If we align the nodes by ascending order of their index from the left to the right, we can easily identify the subgraph G'_L (G'_U) which contains all the downward (upward) arcs of G'. (see Figure 6)

We can initialize the arc adjacency lists $ui(i)$, $di(i)$, $uo(i)$ and $do(i)$ (see Section 4.1) when we read the graph data. If $x_{st} = \infty$ and $x_{sk} + x_{kt} < \infty$ where $k < \min\{s,t\}$, we add a fill-in arc (s,t) to G. The adjacency lists $di(k)$, $do(k)$, $ui(k)$, and $uo(k)$ for node $k > 1$ are updated during the procedure whenever a new arc is added. Given two arcs (s,k) and (k,t), a *triple comparison* $s \to k \to t$ compares $c_{sk} + c_{kt}$ with c_{st}. If there exists no arc (s,t), we add a fill-in arc (s,t) and assign $c_{sk} + c_{kt}$ to be its length; Otherwise, we update its length to be $c_{sk} + c_{kt}$.

G_LU is a sparse implementation of the triple comparisons $s \to k \to t$ for any (s,t) and for each $k = 1, \ldots, (\min\{s,t\} - 1)$. In particular, a shortest path for any node pair (s,t) in $H([1, \min\{s,t\}] \cup \max\{s,t\})$ will be computed and stored as an arc (s,t) in G'. Thus $x_{n,n-1} = x^*_{n,n-1}$ and $x_{n-1,n} = x^*_{n-1,n}$ since $H([1, n-1] \cup n) = G$. *G_LU* can also detect negative cycles.

The complexity of this procedure depends on the topology and node ordering. The number of triple comparisons is bounded by $\sum_{k=1}^{n-2}(|di(k)| \cdot |uo(k)|)$, or $O(n^3)$ in the worst case. It is $\frac{n(n-1)(n-2)}{3}$ on a complete graph. In practice a good node ordering may reduce the number of comparisons for non-complete graphs. Determining a good node ordering is the same as determining a good permutation of columns when solving systems of equations so that the number of fill-ins required is reduced in the LU decomposition. More implementation details of our algorithms regarding sparsity techniques will be covered in Chapter 5.

4.2.2 Procedure $Acyclic\_L(j_0)$

After obtaining the shortest paths in $H([1,t])$ from each node $s > t$ to each node t in previous procedure, $Acyclic\_L(j_0)$ computes their shortest paths in $H([1,s])$ from each node $s > t$ to each node $t \geq j_0$. The updated x_{st} and $succ_{ij}$ will then be used to compare with the shortest distances in $H([1,r])$ for each $r = (s+1), \ldots, n$ from each node $s > t$ to each node $t \geq j_0$ by the last procedure.

This procedure does sequences of shortest path tree computations in G'_L, the acyclic subgraph of augmented graph G' that contains all of its downward arcs (see Section 4.2.1). Its subprocedure, $Get\_D\_L(t)$, resembles the forward elimination in Gaussian elimination. Each application of subprocedure $Get\_D\_L(t)$ gives the shortest distance in G'_L from each

```
Procedure Acyclic_L(j0)
begin
      Initialize: ∀ node k, dô(k) := do(k)
      for t = j0 to n − 2 do
            Get_D_L(t);
end
```

```
Subprocedure Get_D_L(t)
begin
      put node t in LIST
      while LIST is not empty do
            remove the lowest node k in LIST
            for each arc (s, k) ∈ di(k) do
                  if s ∉ LIST, put s into LIST
                  if x_st > x_sk + x_kt then
                        if x_st = ∞ then
                              add a new arc (s, t) to dô(s)
                     x_st := x_sk + x_kt ; succ_st := succ_sk
end
```

node $s > t$ to node t, and we repeat this subprocedure for each root node $t = j_0, \ldots, (n-2)$. Thus for each OD pair (s, t) satisfying $s > t \geq j_0$, we obtain the shortest distance in G'_L from s to t which in fact corresponds to the shortest distance in $H([1, s])$ from s to t. (see Corollary 4.2(a) in Section 4.2.5) Also, this procedure gives x^*_{nt}, the shortest distance in G from node n to any node $t \geq j_0$. (see Corollary 4.2(c) in Section 4.2.5)

Figure 5(b) in Section 4.2 illustrates the operations of *Acyclic_L*(j_0). Each application of the subprocedure *Get_D_L*(t) updates each entry (s, t) in column t of $[x_{ij}]$ and $[succ_{ij}]$ satisfying $s > t$, to represent the shortest distance in G'_L from node s to node t and the successor of node s in that shortest path.

For each node $k > j_0$, we introduce a new down-outward arc adjacency list (defined in Section 4.1) denoted $\hat{do}(k)$ which is initialized to be $do(k)$ obtained from procedure *G_LU*. Whenever we identify a path (not a single arc) in G'_L from node $k > t$ to node $t \geq j_0$, we add a new arc (k, t) to $\hat{do}(k)$. This connectivity information will be used by procedure *Reverse_LU* for sparse operation and complexity analysis.

The number of triple comparisons is bounded by $\sum\limits_{t=j_0}^{n-2} \sum\limits_{k=t}^{n-1} |di(k)|$, or $O((n - j_0)^3)$ in the worst case. Note that if we choose a node ordering such that j_0 is large, then we may decrease the computational work in this procedure, but such an ordering may make the first procedure *G_LU* and the next procedure *Acyclic_U*(i_0) inefficient. When solving an APSP problem, we have to set $j_0 = 1$; thus, a $\sum_{t=1}^{n-2}\sum_{k=t}^{n-1} |di(k)|$ bound is obtained, which will be $\frac{n(n-1)(n-2)}{6}$ on a complete graph. Therefore this procedure is $O(n^3)$ in the worst case.

4.2.3 Procedure *Acyclic_U*(i_0)

After obtaining the shortest paths in $H([1,s])$ from each node $s < t$ to each node t in procedure *G_LU*, *Acyclic_U*(i_0) computes their shortest paths in $H([1,t])$ from each node $s \geq i_0$ to each node $t > s$. The updated x_{st} will then be compared with the shortest distances in $H([1,r])$ for each $r = (t+1), \ldots, n$ from each node $s \geq i_0$ to each node $t > s$ by the last procedure.

```
Procedure Acyclic_U(i0)
begin
    Initialize: ∀ node k, ui^(k) := ui(k)
    for s = i0 to n − 2 do
        Get_D_U(s);
end


Subprocedure Get_D_U(s)
begin
    put node s in LIST
    while LIST is not empty do
        remove the lowest node k in LIST
        for each arc (k, t) ∈ uo(k) do
            if t ∉ LIST, put t into LIST
            if x_st > x_sk + x_kt then
                    if x_st = ∞ then
                        add a new arc (s, t) to ui^(t)
                x_st := x_sk + x_kt ; succ_st := succ_sk
end
```

This procedure is similar to *Acyclic_L*(j_0) except it is applied on the upper triangular part of $[x_{ij}]$ and $[succ_{ij}]$, which corresponds to shortest distance from i to j and the successor of i in that shortest path in G'_U, the acyclic subgraph of augmented graph G' that contains all of its upward arcs (see Section 4.2.1). Each application of subprocedure *Get_D_U*(s) gives the shortest distance in G'_U from each node s to each node $t > s$, and we repeat this subprocedure for each root node $s = i_0, \ldots, (n-2)$. Thus for each OD pair (s,t) satisfying $i_0 \leq s < t$, we obtain the shortest distance in G'_U from s to t which in fact corresponds to the shortest distance in $H([1,t])$ from s to t. (see Corollary 4.2(b) in Section 4.2.5) Also, this procedure gives x^*_{sn}, the shortest distance in G from any node $s \geq i_0$ to node n. (see Corollary 4.2(c) in Section 4.2.5)

Figure 5(b) in Section 4.2 illustrates the operations of $Acyclic\_U(i_0)$. Each application of the subprocedure $Get\_D\_U(s)$ updates each entry (s,t) in row s of $[x_{st}]$ and $[succ_{st}]$ satisfying $s < t$ which represents the shortest distance in G'_U from node $s < t$ to node t and the successor of node s in that shortest path.

For each node $k > i_0$, we introduce a new up-inward arc adjacency list (defined in Section 4.1) denoted $\hat{ui(k)}$ which is initialized to be $ui(k)$ obtained from procedure G_LU. Whenever we identify a path (not a single arc) in G'_U from node $s \geq i_0$ to node $k > s$, we add a new arc (s,k) to $\hat{ui(k)}$. This connectivity information will be used by procedure $Reverse\_LU$ for sparse operation and complexity analysis.

The number of triple comparisons is bounded by $\sum_{s=i_0}^{n-2} \sum_{k=s}^{n-1} |uo(k)|$, or $O((n-i_0)^3)$ in the worst case. Note that if we choose a node ordering such that i_0 is large, then we may decrease the computational work in this procedure, but such an ordering may make the first procedure G_LU and previous procedure $Acyclic\_L(j_0)$ inefficient. When solving an APSP problem, we have to set $i_0 = 1$; thus, a $\sum_{s=1}^{n-2} \sum_{k=s}^{n-1} |uo(k)|$ bound is obtained, which will be $\frac{n(n-1)(n-2)}{6}$ on a complete graph. Therefore this procedure is $O(n^3)$ in the worst case.

4.2.4 Procedure $Reverse\_LU(i_0, j_0, k_0)$

After previous two procedures, the algorithm will have determined the length of shortest path in $H(1, \max\{s,t\})$ from each node $s \geq i_0$ to each node $t \geq j_0$. $Reverse\_LU(i_0, j_0, k_0)$ then computes shortest distances in $H([1,r])$ for each $r = n, \ldots, (\max\{s,t\}+1)$ from each origin $s \geq k_0$ to each destination $t \geq j_0$ or from each origin $s \geq i_0$ to each destination $t \geq k_0$. Note that k_0 is set to be $\min_i\{\max\{s_i, t_i\}\}$ so that all the requested OD pairs in Q will be correctly updated by the procedure.

Procedure Reverse_LU($\mathbf{i_0, j_0, k_0}$)
begin
    for $k = n$ down to $k_0 + 1$ ***do***
        for each arc $(s, k) \in \hat{ui(k)}$ and $s \geq i_0$ ***do***
            for each arc $(k, t) \in \hat{do(k)}$ and $t \geq j_0$, $s \neq t$ ***do***
                if $x_{st} > x_{sk} + x_{kt}$ ***then***
                    if $x_{st} = \infty$ ***then***
                        if $s > t$ ***then*** add new arc (s, t) to $\hat{do(s)}$
                        if $s < t$ ***then*** add new arc (s, t) to $\hat{ui(t)}$
                    $x_{st} := x_{sk} + x_{kt}$; $succ_{st} := succ_{sk}$
end

Figure 5(c) in Section 4.2 illustrates the operations of $Reverse\_LU(i_0, j_0, k_0)$. It sequentially uses node $k = n, \ldots, (k_0 + 1)$ as an intermediate node to update each entry (s, t) of $[x_{ij}]$ and $[succ_{ij}]$ that satisfies $i_0 \leq s < k$ and $j_0 \leq t < k$ as long as $x_{sk} < \infty$, $x_{kt} < \infty$ and $x_{st} > x_{sk} + x_{kt}$.

Thus this procedure is similar to the first procedure, G_LU, but proceeds in reverse fashion. Since x^*_{sn} and x^*_{nt} for $s \geq i_0$ and $t \geq j_0$ will have been computed by $Acyclic\_U(i_0)$ and $Acyclic\_L(j_0)$ respectively, $Reverse\_LU(i_0, j_0, k_0)$ computes x^*_{sk} and x^*_{kt} for each s satisfying $i_0 \leq s < k$, each t satisfying $j_0 \leq t < k$, and for each $k = (n-1), \ldots, k_0$ where $k_0 := \min_i \{\max\{s_i, t_i\}\}$. In particular, this procedure computes length of shortest paths that must pass through node r in $H([1, r])$ for each $r = n, \ldots, (\max\{s, t\} + 1)$ from each origin $s \geq k_0$ to each destination $t \geq j_0$ or from each origin $s \geq i_0$ to each destination $t \geq k_0$. Since we will have obtained the shortest distances in $H([1, \max\{s, t\}])$ from each node $s \geq i_0$ to each node $t \geq j_0$ from previous procedures, the smaller of these two distances will be the x^*_{st} in G. This procedure stops when shortest distances for all the requested OD pairs are calculated.

Although this procedure calculates all shortest path lengths, not all of the paths themselves are known. To trace shortest paths for all the requested OD pairs by $[succ_{ij}]$, we must set $i_0 = 1$ and $k_0 = j_0$ in the beginning of the algorithm so that at iteration $k = j_0$ the successor columns $j_0, \ldots, n$ are valid for tracing the shortest path tree rooted at sink node k. Otherwise, there will exist some $succ_{st}$ with $s < i_0$ or $t < k_0$ that have not been updated by the algorithm and thus are not qualified to provide correct successor information for shortest path tracing purposes.

Note that for each node k, $\hat{do(k)}$ and $\hat{ui(k)}$ may be modified during this procedure, since they represent connectivity between node k and other nodes in G. In particular, if node $s \geq i_0$ can only reach node $t \geq j_0$ via some intermediate node $k > \max\{s,t\}$, this procedure will identify that path and add arc (s,t) to $\hat{do(s)}$ if $s > t$, or $\hat{ui(t)}$ if $s < t$.

When solving an APSP problem on a highly connected graph, their magnitude, $\left|\hat{do(k)}\right|$ and $\left|\hat{ui(k)}\right|$ tend to achieve their upper bound k for each node k. The number of triple comparisons is bounded by $\sum\limits_{k=k_0+1}^{n}(\left|\hat{ui(k)}\right| \cdot \left|\hat{do(k)}\right|)$, or $O((n-k_0)^3)$ in the worst case. Note that if we choose a node ordering such that k_0 is large, then we may decrease computational work in this procedure, but such an ordering may make the first procedure *G_LU* and one of the procedures *Acyclic_L*(j_0) and *Acyclic_U*(i_0) inefficient. This procedure has a time bound $\frac{n(n-1)(n-2)}{3}$ when solving an APSP problem on a complete graph. Therefore this procedure is $O(n^3)$ in the worst case.

4.2.5 Correctness and properties of algorithm $DLU1$

First we show the correctness of the algorithm, and then discuss some special properties of this algorithm. To prove its correctness, we will show how the procedures of $DLU1$ calculate shortest path lengths for various subsets of OD pairs, and then demonstrate that every OD pair must be in one such subset.

We begin by specifying the set of OD pairs whose shortest path lengths will be calculated by Procedure *G_LU*. In particular, *G_LU* will identify shortest path lengths for those requested OD pairs (s,t) whose shortest paths have all intermediate nodes with index lower than $\min\{s,t\}$.

Theorem 4.1. *A shortest path in G from s to t that has a highest node with index equal to* $\min\{s,t\}$ *will be reduced to arc (s,t) in G' by Procedure G_LU.*

Proof. Suppose such a shortest path in G from s to t contains p arcs. In the case where $p = 1$ and $r = s$ or t, the result is trivial. Let us consider the case where $p > 1$. That is, $s := v_0 \to v_1 \to v_2 \to \ldots \to v_{p-2} \to v_{p-1} \to v_p := t$ is a shortest path in G from s to t with $(p-1)$ intermediate nodes $v_k < \min\{s,t\}$ for $k = 1, \ldots, (p-1)$.

Let $v_\alpha < \min\{s,t\}$ be the lowest node in this shortest path. In the $k = v_\alpha$ iteration, *G_LU* will modify the length of arc $(v_{\alpha-1}, v_{\alpha+1})$ (or add this arc if it does not exist in G') to be sum of the arc lengths of $(v_{\alpha-1}, v_\alpha)$ and $(v_\alpha, v_{\alpha+1})$. Thus we obtain another path with $(p-1)$ arcs that is as short as the previous one. *G_LU* now repeats the same procedure that eliminates the new lowest node and constructs another path that is just as short but contains one fewer arc. By induction, in the $\min\{s,t\}$ iteration, *G_LU* eventually modifies (or adds if $(s,t) \notin A$) arc (s,t) with length equal to which of the shortest path from s to t in $H([1, \min\{s,t\}] \cup \max\{s,t\})$.

Therefore any arc (s,t) in G' corresponds to a shortest path from s to t with length x_{st} in $H([1, \min\{s,t\}] \cup \max\{s,t\})$. Since any shortest path in G from s to t that passes through only intermediate nodes $v_k < \min\{s,t\}$ corresponds to the same shortest path in $H([1, \min\{s,t\}] \cup \max\{s,t\})$, Procedure *G_LU* thus correctly computes the length of such a shortest path and stores it as the length of arc (s,t) in G'. □

Corollary 4.1. *(a) Procedure G_LU will correctly compute* $x^*_{n,n-1}$ *and* $x^*_{n-1,n}$.
(b) Procedure G_LU will correctly compute a shortest path for any node pair (s,t) *in* $H([1, \min\{s,t\}] \cup \max\{s,t\})$.

Proof. (a) This follows immediately from Theorem 4.1, because all other nodes have index less than $(n-1)$.

(b) This follows immediately from Theorem 4.1. □

Now, we specify the set of OD pairs whose shortest path lengths will be calculated by Procedure *Acyclic_L*(j_0) and *Acyclic_U*(i_0). In particular, these two procedures will give shortest path lengths for those requested OD pairs (s,t) whose shortest paths have all intermediate nodes with index lower than $\max\{s,t\}$.

Theorem 4.2. *(a) A shortest path in G from node* $s > t$ *to node* t *that has* s *as its highest node corresponds to a shortest path from* s *to* t *in* G'_L.
(b) A shortest path in G from node $s < t$ *to node* t *that has* t *as its highest node corresponds to a shortest path from* s *to* t *in* G'_U.

Proof. (a) Suppose such a shortest path in G from node $s > t$ to node t contains p arcs. In the case where $p = 1$, the result is trivial. Let us consider the case where $p > 1$. That is, $s \to v_1 \to v_2 \to \ldots \to v_{p-2} \to v_{p-1} \to t$ is a shortest path in G from node $s > t$ to node t with $(p-1)$ intermediate nodes $v_k < \max\{s,t\} = s$ for $k = 1, \ldots, (p-1)$.

In the case where every intermediate node $v_k < \min\{s,t\} = t < s$, Theorem 4.1 already shows that G_LU will compute such a shortest path and store it as arc (s,t) in G'_L. So, we only need to discuss the case where some intermediate node v_k satisfies $s > v_k > t$.

Suppose that r of the p intermediate nodes in this shortest path in G from s to t satisfy $s > u_1 > u_2 > \ldots > u_{r-1} > u_r > t$. Define $u_0 := s$, and $u_{r+1} := t$. We can break this shortest path into $(r+1)$ segments: u_0 to u_1, u_1 to u_2,..., u_r to u_{r+1}. Each shortest path segment $u_{k-1} \to u_k$ in G contains intermediate nodes that all have lower index than u_k. Since Theorem 4.1 guarantees that G_LU will produce an arc (u_{k-1}, u_k) for any such shortest path segment $u_{k-1} \to u_k$, the original shortest path that contains p arcs in G actually can be represented as the shortest path $s \to u_1 \to u_2 \to \ldots \to u_{r-1} \to u_r \to j$ in G'_L.

(b) Using a similar argument to (a) above, the result follows immediately. □

Corollary 4.2. *(a) Procedure Acyclic_L(j_0) will correctly compute shortest paths in $H([1,s])$ for all node pairs (s,t) such that $s > t \geq j_0$.*

(b) Procedure Acyclic_U(i_0) will correctly compute shortest paths in $H([1,t])$ for all node pairs (s,t) such that $i \leq s < t$.

(c) Procedure Acyclic_L(j_0) will correctly compute x^*_{nt} for each node $t \geq j_0$; Procedure Acyclic_U(i_0) will correctly compute x^*_{sn} for each node $s \geq i_0$

Proof. (a) This procedure computes sequences of shortest path tree in G'_L rooted at node $t = j_0, \ldots, (n-2)$ from all other nodes $s > t$. By Theorem 4.2(a), a shortest path in G'_L from node $s > t$ to node t corresponds to a shortest path in G from s to t where s is its highest node since all other nodes in this path in G'_L have lower index than s. In other words, such a shortest path corresponds to the same shortest path in $H([1,s])$.

Including the case of $t = (n-1)$ and $s = n$ as discussed in Corollary 4.1(a), the result follows directly.

(b) Using a similar argument as part (a), the result again follows directly.

(c) These follow immediately from part (a) and part (b). □

Finally, we demonstrate that procedure $Reverse\_LU(i_0, j_0, k_0)$ will correctly calculate all shortest path lengths. In particular, $Reverse\_LU(i_0, j_0, k_0)$ gives shortest path lengths for those requested OD pairs (s,t) whose shortest paths have some intermediate nodes with index higher than $\max\{s,t\}$.

Lemma 4.1. *(a) Any shortest path in G from s to t that has a highest node with index $h > \max\{s,t\}$ can be decomposed into two segments: a shortest path from s to h in G'_U, and a shortest path from h to t in G'_L.*
(b) Any shortest path in G from s to t can be determined by the shortest of the following two paths: (i) the shortest path from s to t in G that passes through only nodes $v \leq r$, and (ii) the shortest path from s to t in G that must pass through some node $v > r$, where $1 \leq r \leq n$.

Proof. (a) This follows immediately by combining Corollary 4.2(a) and (b).

(b) It is easy to see that every path from s to t must either passes through some node $v > r$ or else not. Therefore the shortest path from s to t must be the shorter of the minimum-length paths of each type. □

Theorem 4.3. *The k^{th} application of $Reverse\_LU(i_0, j_0, k_0)$ will correctly compute $x^*_{n-k,t}$ and $x^*_{s,n-k}$ for each s satisfying $i_0 \leq s < (n-k)$, and each t satisfying $j_0 \leq t < (n-k)$ where $k \leq (n-k_0)$.*

Proof. After procedures $Acyclic\_L(j_0)$ and $Acyclic\_U(i_0)$, we will have obtained shortest paths in $H([1, \max\{s,t\}])$ from each node $s \geq i_0$ to each node $t \geq j_0$. To obtain the shortest path in G from s to t, we need only to check those shortest paths that must pass through node h for each $h = (\max\{s,t\}+1), \ldots, n$. By Lemma 4.1(a), such a shortest path can be decomposed into two segments: from s to h and from h to t. Note that their shortest distances, x_{sh} and x_{ht}, will have been calculated by $Acyclic\_U(i_0)$ and $Acyclic\_L(j_0)$, respectively.

For each node $s \geq i_0$ and $t \geq j_0$, Corollary 4.2(c) shows that x^*_{nt} and x^*_{sn} will have been computed by procedures *Acyclic_L*(j_0) and *Acyclic_U*(i_0), respectively. Define x^k_{st} to be the shortest known distance from s to t after the k^{th} application of *Reverse_LU*(i_0, j_0, k_0), and x^0_{st} to be the best known distance from s to t before this procedure. Now we will prove this theorem by induction.

In the first iteration, *Reverse_LU*(i_0, j_0, k_0) updates $x^1_{st} := \min\{x^0_{st}, x^0_{sn} + x^0_{nt}\} = \min\{x_{st}, x^*_{sn} + x^*_{nt}\}$ for each node pair (s,t) satisfying $i_0 \leq s < n$ and $j_0 \leq t < n$. For node pairs $(n-1, t)$ satisfying $j_0 \leq t < (n-1)$, $x^*_{n-1,t} = \min\{x^0_{n-1,t}, x^*_{n-1,n} + x^*_{nt}\}$ by applying Lemma 4.1(b) with $r = (n-1)$. Likewise, $x^*_{s,n-1}$ is also determined for each s satisfying $i_0 \leq s < (n-1)$ in this iteration.

Suppose the theorem holds for iteration $k = \hat{k} < (n - \max\{s,t\})$. That is, at the end of iteration $k = \hat{k}$, *Reverse_LU*(i_0, j_0, k_0) gives $x^*_{n-\hat{k},t}$ and $x^*_{s,n-\hat{k}}$ for each s satisfying $i_0 \leq s < (n-\hat{k})$, and each t satisfying $j_0 \leq t < (n-\hat{k})$. In other words, we will have obtained $x^*_{n-r,t}$ and $x^*_{s,n-r}$ for each $r = 0, 1, \ldots, \hat{k}$, and for all $s \geq i_0$, $t \geq j_0$.

In iteration $k = (\hat{k}+1)$, for each t satisfying $j_0 \leq t < (n - \hat{k} - 1)$, $x^{\hat{k}+1}_{n-\hat{k}-1,t} := \min\{x^{\hat{k}}_{n-\hat{k}-1,t}, x^{\hat{k}}_{n-\hat{k}-1,n-\hat{k}} + x^{\hat{k}}_{n-\hat{k},t}\} = \min\{x^{\hat{k}}_{n-\hat{k}-1,t}, x^*_{n-\hat{k}-1,n-\hat{k}} + x^*_{n-\hat{k},t}\}$ by assumption of the induction. Note that the first term $x^{\hat{k}}_{n-\hat{k}-1,t}$ has been updated $\hat{k}$ times in the previous $\hat{k}$ iterations. In particular, $x^{\hat{k}}_{n-\hat{k}-1,t} = \min\limits_{0 \leq k \leq (\hat{k}-1)} \{x^0_{n-\hat{k}-1,t}, x^*_{n-\hat{k}-1,n-k} + x^*_{n-k,t}\}$ where $x^0_{n-\hat{k}-1,t}$ represents the length of a shortest path in G from node $(n-\hat{k}-1)$ to node t that has node $(n-\hat{k}-1)$ as its highest node. Substituting this new expression of $x^{\hat{k}}_{n-\hat{k}-1,t}$ into $\min\{x^{\hat{k}}_{n-\hat{k}-1,t}, x^*_{n-\hat{k}-1,n-\hat{k}} + x^*_{n-\hat{k},t}\}$, we obtain $x^{\hat{k}+1}_{n-\hat{k}-1,t} := \min\limits_{0 \leq k \leq \hat{k}} \{x^0_{n-\hat{k}-1,t}, x^*_{n-\hat{k}-1,n-k} + x^*_{n-k,t}\}$ whose second term $\min\limits_{0 \leq k \leq \hat{k}} \{x^*_{n-\hat{k}-1,n-k} + x^*_{n-k,t}\}$ corresponds to the length of a shortest path in G from node $(n-\hat{k}-1)$ to node t that must pass through some higher node with index $v > (n-\hat{k}-1)$ (v may be $(n-\hat{k}), \ldots, n$). By Lemma 4.1(b) with $r = (n-\hat{k}-1)$, we conclude $x^{\hat{k}+1}_{n-\hat{k}-1,t} = x^*_{n-\hat{k}-1,t}$ for each t satisfying $j_0 \leq t < (n-k)$. Likewise, $x^{\hat{k}+1}_{s,n-\hat{k}-1} = x^*_{s,n-\hat{k}-1}$ for each s satisfying $i_0 \leq s < (n-\hat{k}-1)$ will also be determined in the end of iteration $(\hat{k}+1)$.

By induction, we have shown the correctness of this theorem. □

Corollary 4.3. *(a) Procedure* $Reverse\_LU(i_0, j_0, k_0)$ *will terminate in* $(n - k_0)$ *iterations, and correctly compute* $x^*_{s_i,t_i}$ *for each of the requested OD pair* (s_i, t_i), $i = 1, \ldots, q$

(b) To trace shortest path for each requested OD pair (s_i, t_i) *in* Q*, we have to initialize* $i_0 := 1$ *and* $k_0 := j_0$ *in the beginning of Algorithm* $DLU1$.

(c) Any APSP problem can be solved by Algorithm $DLU1$ *with* $i_0 := 1$, $j_0 := 1$, *and* $k_0 := 2$.

Proof. (a) By setting $k_0 := \min_i\{\max\{s_i, t_i\}\}$, the set of all the requested OD pairs Q is a subset of node pairs $\{(s,t) : s \geq k_0, t \geq j_0\} \cup \{(s,t) : s \geq i_0, t \geq k_0\}$ whose x^*_{st} and $succ^*_{st}$ is shown to be correctly computed by Theorem 4.3. Therefore $Reverse\_LU(i_0, j_0, k_0)$ terminates in $n-(k_0+1)+1 = (n-k_0)$ iterations and correctly computed $x^*_{s_it_i}$ and $succ^*_{s_it_i}$ for each requested OD pair (s_i, t_i) in Q.

(b) The entries $succ_{st}$ for each $s \geq i_0$ and $t \geq j_0$ are updated in all procedures whenever a better path from s to t is identified. To trace the shortest path for a particular OD pair (s_i, t_i), we need the entire t_i^{th} column of $[succ^*_{ij}]$ which contains information of the shortest path tree rooted at sink node t_i. Thus we have to set $i_0 := 1$ so that procedures G_LU, $Acyclic\_L(j_0)$ and $Acyclic\_U(1)$ will update entries $succ_{st}$ for each $s \geq 1$ and $t \geq j_0$. However, $Reverse\_LU(1, j_0, k_0)$ will only update entries $succ_{st}$ for each $s \geq 1$ and $t \geq k_0$. Thus it only gives the t^{th} column of $[succ^*_{ij}]$ for each $t \geq k_0$ in which case some entries $succ_{st}$ with $1 \leq s < k_0$ and $j_0 \leq t < k_0$ may still contain incomplete successor information unless we set $k_0 := j_0$ in the beginning of this procedure.

(c) We set $i_0 := j_0 := 1$ because we need to update all entries of the $n \times n$ distance matrix $[x_{ij}]$ and successor matrix $[succ_{ij}]$ when solving any APSP problem. Setting $k_0 := 1$ will make the last iteration of $Reverse\_LU(1, 1, k_0)$ update x_{11} and $succ_{11}$, which is not necessary. Thus it suffices to set $k_0 := 2$ when solving any APSP problem. □

Algorithm $DLU1$ can easily identify a negative cycle. In particular, any negative cycle will be identified in procedure G_LU.

Theorem 4.4. *Suppose there exists a k-node cycle* C_k, $i_1 \to i_2 \to i_3 \to \ldots \to i_k \to i_1$, *with negative length. Then, procedure* G_LU *will identify it.*

Proof. Without loss of generality, let i_1 be the lowest node in the cycle C_k, i_r be the second lowest, i_s be the second highest, and i_t be the highest node. Let $length(C_k)$ denote the length function of cycle C_k. Assume that $length(C_k) = \sum_{(i,j)\in C_k} c_{ij} < 0$.

In G_LU, before we begin iteration i_1(using i_1 as the intermediate node), the length of some arcs of C_k might have already been modified, but no arcs of C_k will have been removed nor will $length(C_k)$ have increased. After we finish scanning $di(i_1)$ and $uo(i_1)$, we can identify a smaller cycle C_{k-1} by skipping i_1 and adding at least one more arc (i_k, i_2) to G. In particular, C_{k-1} is $i_k \to i_2 \to \ldots \to i_{k-1} \to i_k$, and $length(C_{k-1}) = length(C_k) - (x_{i_1 i_2} + x_{i_k i_1} - x_{i_k i_2})$. Since $x_{i_k i_2} \leq x_{i_1 i_2} + x_{i_k i_1}$ by the algorithm, we obtain $length(C_{k-1}) \leq length(C_k) < 0$. The lowest-index node in C_{k-1} is now $i_r > i_1$, thus we will again reduce the size of C_{k-1} by 1 in iteration $k = i_r$.

We iterate this procedure, each time processing the current lowest node in the cycle and reducing the cycle size by 1, until finally a 2-node cycle C_2, $i_s \to i_t \to i_s$, with $length(C_2) \leq length(C_3) \leq \ldots \leq length(C_k) < 0$ is obtained. Therefore, $x_{tt} < 0$ and a negative cycle in the augmented graph G' is identified with cycle length smaller than or equal to the original negative cycle C_k. □

To sum up, suppose the shortest path in G from s to t contains more than one intermediate node and let r be the highest intermediate node in that shortest path. There are only three cases: (1) $r < \min\{s,t\}$ (2) $\min\{s,t\} < r < \max\{s,t\}$ and (3) $r > \max\{s,t\}$. The first case will be solved by G_LU, second case by $Acyclic\_L$ and $Acyclic\_U$, and third case by $Reverse\_LU$. For any OD pair whose shortest path is a single arc, Algorithm $DLU1$ includes it in the very beginning and compares it with all other paths in the three cases discussed previously.

When solving an APSP problem on an undirected graph, Algorithm $DLU1$ can save half of the storage and computational work. In particular, since the graph is symmetric, $ui(k) = do(k)$, and $uo(k) = di(k)$ for each node k. Therefore storing only the lower (or upper) triangular part of the distance matrix $[x_{ij}]$ and successor matrix $[succ_{ij}]$ is sufficient for the algorithm. In addition, Procedure G_LU and $Reverse\_LU(i_0, j_0, k_0)$ can save half of their computation. In particular, procedure $Reverse\_LU(i_0, j_0, k_0)$ can be replaced by $Reverse\_LU(l_0, l_0, k_0)$ where $l_0 := \min\{i_0, j_0\}$ with the modification that it only updates entries of the lower (or upper) triangular part of $[x_{ij}]$ and $[succ_{ij}]$. We can also integrate procedures $Acyclic\_L(j_0)$ and $Acyclic\_U(i_0)$ into one procedure $Acyclic\_L(l_0)$ (or $Acyclic\_U(l_0)$). Thus half of the computational work can be saved. The SSSP algorithm, on the other hand, can not take advantage of this feature of undirected graphs. In particular, we still have to apply any SSSP algorithm $(n-1)$ times, each time for a different source node, to obtain APSP.

For problems on acyclic graphs, we can reorder the nodes so that the upper (or lower) triangle of $[x_{ij}]$ becomes empty. Then we can skip procedure $Acyclic\_U$ (or $Acyclic\_L$).

A good node ordering may practically reduce much computational work, but the best ordering is difficult (NP-complete) to obtain. In general, an ordering that reduces fill-in arcs in the first procedure G_LU may be beneficial for the other three procedures as well. On the other hand, an ordering that makes i_0 or j_0 as large as possible seems to be more advantageous for the last three procedures, but it may create more fill-in arcs in procedure G_LU, which in turn may worsen the efficiency of the last three procedures.

Overall, the complexity of Algorithm $DLU1$ is $O(\sum_{k=1}^{n-2}(|di(k)|\cdot|uo(k)|) + \sum_{t=j_0}^{n-2}\sum_{k=t}^{n-1}|di(k)| + \sum_{s=i_0}^{n-2}\sum_{k=s}^{n-1}|uo(k)| + \sum_{k=k_0+1}^{n}(\left|\hat{ui(k)}\right|\cdot\left|\hat{do(k)}\right|))$ which is $O(n^3)$ in the worst case. When solving an APSP problem on a complete graph, the four procedures G_LU, $Acyclic\_L(1)$, $Acyclic\_U(1)$ and $Reverse\_LU(1,1,2)$ will take $\frac{1}{3}, \frac{1}{6}, \frac{1}{6}$ and $\frac{1}{3}$ of the total $n(n-1)(n-2)$ triple comparisons respectively, which is as efficient as Carré's algorithm and Floyd-Warshall's algorithm in the worst case.

Algorithm $DLU1$ is more suitable for dense graphs than for sparse graphs, since the fill-in arcs we add in each procedure might destroy the graph's sparsity. In particular, in the beginning of procedure $Reverse\_LU(i_0, j_0, k_0)$, the distance matrix tends to be very dense, which makes its sparse implementation less efficient.

As stated in Corollary 4.3, Algorithm $DLU1$ obtains shortest distances for all node pairs (s,t) satisfying $s \geq k_0$, $t \geq j_0$ or $s \geq i_0$, $t \geq k_0$ where $k_0 := \min_i\{\max\{s_i, t_i\}\}$. This of course includes all the requested OD pairs in Q which makes it more efficient than other algebraic APSP algorithms since other APSP algorithms usually need to update all the $n \times n$ entries of $[x_{ij}]$ and $[succ_{ij}]$. However, $DLU1$ still does some redundant computations for many other unwanted OD pairs..Such computational redundancy can be remedied by our next algorithm $DLU2$. In addition, $DLU1$ has a major drawback in the computational dependence of x^*_{st} and $succ^*_{st}$ from higher nodes to lower nodes. In particular, to obtain $x^*_{s't'}$, we rely on the availability of $x^*_{st'}$ for each $s > s'$ and $x^*_{st'}$ for each $t > t'$. We propose two ways to overcome this drawback: one is the algorithm $DLU2$ presented in Section 4.3, and the other is a sparse implementation that will appear in Chapter 5.

Another drawback of this algorithm is the necessity of setting $i_0 := 1$ for the traceability of shortest paths. Unfortunately this is inevitable in our algorithms. However, Algorithm $DLU1$ still seems to have an advantage over other algebraic APSP algorithms because it can solve sequences of SSSP problems rather than having to solve an entire APSP problem. Although Algorithm $DLU1$ has to compute a higher rooted shortest path tree to obtain the lower rooted one, our second algorithm $DLU2$ and the other new sparse algorithm in Chapter 5 can overcome such difficulty.

4.3 Algorithm $DLU2$

Our second MPSP algorithm, $DLU2$, not only obtains shortest distances x^*_{st} faster but also traces shortest paths more easily than $DLU1$. Suppose Q is the set of q OD pairs (s_i, t_i) for $i = 1, \ldots, q$. Unlike $DLU1(Q)$ that not only computes x^*_{st} for all (s, t) in Q but also other unrequested OD pairs, $DLU2(Q)$ specifically attacks each requested OD pair in Q after the common LU decomposition procedure G_LU. Thus it should be more efficient, especially for problems where the requested OD pairs are sparsely distributed in the $n \times n$ OD matrix (i.e. only few OD pairs, but with their origin or destination indices scatteredly ordered among $[1, n]$).

To cite an extreme example, suppose we want to compute shortest path lengths on a graph with n nodes for OD pairs $Q = \{(1, 2), (2, 1)\}$. Suppose we are not allowed to alter the node ordering (otherwise it becomes easy since we can reorder them to be $\{(n-1, n), (n, n-$

2)} which can be solved by one run of G_LU). Then applying Algorithm $DLU1(Q)$, we will end up with solving an APSP problem, which is not efficient at all. On the other hand, Algorithm $DLU2(Q)$ only needs two runs of its procedure *Get_D* to directly calculate shortest distances for each requested OD pair individually after the common procedure G_LU.

Algorithm 6 DLU2($\mathbf{Q} := \{(\mathbf{s_i}, \mathbf{t_i}) : \mathbf{i} = \mathbf{1}, \ldots, \mathbf{q}\}$)

---

begin
  Initialize: $\forall\ s$, $x_{ss} := 0$ and $succ_{ss} := s$
    $\forall (s,t)$, ***if*** $(s,t) \in A$ ***then***
      $x_{st} := c_{st}$; $succ_{st} := t$
      if $s < t$ ***then*** add arc (s,t) to $uo(s)$
      if $s > t$ ***then*** add arc (s,t) to $di(t)$
    else $x_{st} := \infty$; $succ_{st} := 0$
    $opt_{ij} := 0\ \forall\ i = 1, \ldots, n,\ j = 1, \ldots, n$
    $subrow_i := 0$; $subcol_i := 0\ \forall\ i = 1, \ldots, n$
  G_LU;
  set $opt_{n,n-1} := 1$, $opt_{n-1,n} := 1$
  for $i = 1$ to q ***do***
    $Get\_D(s_i, t_i)$;
    if shortest paths need to be traced ***then***
      if $x_{s_i t_i} \neq \infty$ ***then***
        $Get\_P(s_i, t_i)$;
      else there exists no path from s_i to t_i
end

Algorithm $DLU2$ improves the efficiency of computing $x^*_{s_i t_i}$, $succ^*_{s_i t_i}$ and shortest paths. The first procedure G_LU and the two subprocedures, $Get\_D\_U(s_i)$ and $Get\_D\_L(t_i)$, are imported from Algorithm $DLU1$ as in Section 4.2.1, Section 4.2.3, and Section 4.2.2. The new procedure $Get\_D(s', t')$ gives $x^*_{s't'}$ directly without the need of $x^*_{st'}$ for each $s > s'$ and $x^*_{st'}$ for each $t > t'$ as required in Algorithm $DLU1$. Thus it avoids many redundant computations which would be done by $DLU1$.

Figure 7(b) illustrates how *Get_D* individually solves $x^*_{s_i t_i}$ for each requested OD pair (s_i, t_i). For example, to obtain x^*_{23}, it first applies $Get\_D\_U(2)$ to update x_{23}, x_{24}, and x_{25}, then updates x_{43}, and x_{53} by $Get\_D\_L(3)$. Finally it computes $\min\{x_{23}, (x_{24} + x_{43}), (x_{25} + x_{53})\}$ which corresponds to x^*_{23}. On the other hand, Algorithm $DLU1$ requires computations on x^*_{24}, x^*_{43}, x^*_{25} and x^*_{53} which requires more works than $DLU2$.

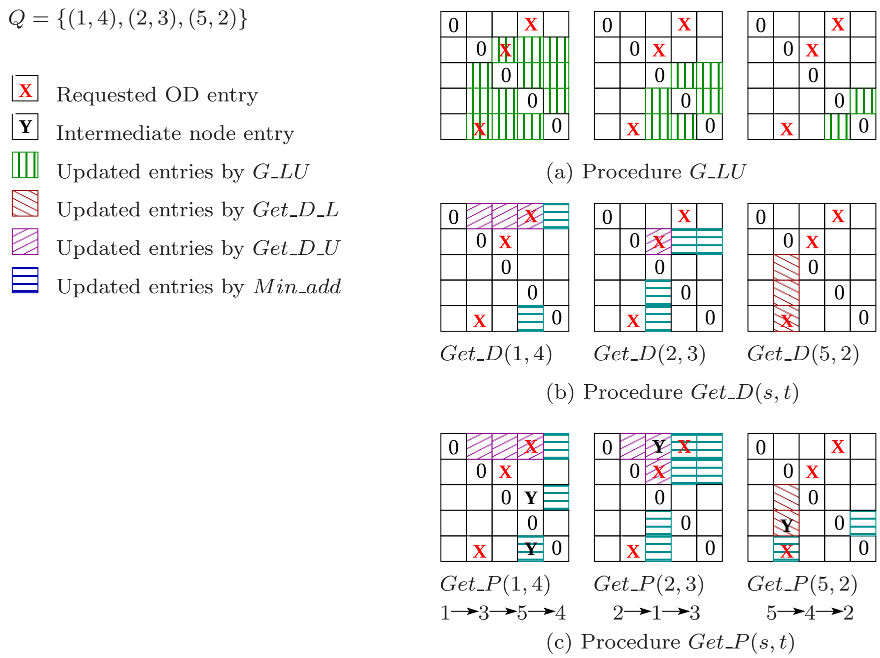


Figure 7: Solving a 3 pairs shortest path problem on a 5-node graph by Algorithm $DLU2(Q)$

If we need to trace the shortest path from s to t, procedure $Get\_P(s,t)$ will iteratively compute all the intermediate nodes in the shortest path from s to t. Therefore, $DLU2$ only does the necessary computations to get the shortest distance and path for the requested OD pair (s,t), whereas $DLU1$ needs to compute the entire shortest path tree rooted at t. For example, suppose $1 \to 3 \to 5 \to 4$ is the shortest path from node 1 to node 4 in Figure 7(c). $DLU2$ first computes x^*_{14} and $succ^*_{14}$. Based on $succ^*_{14} = 3$, which means node 3 is the successor of node 1 in that shortest path, it then computes x^*_{34} and $succ^*_{34} = 5$. Finally it computes x^*_{54} and $succ^*_{54} = 4$, which means node 5 is the last intermediate node in that shortest path. Thus procedure $Get\_P(1,4)$ gives all the intermediate nodes and their shortest distances to the sink node 4. On the other hand, Algorithm $DLU1$ requires additional computations on $succ^*_{24}$ and $succ^*_{25}$.

We introduce a new $n \times n$ optimality indicator array $[opt_{ij}]$ and two $n \times 1$ indicator arrays $[subrow_i]$ and $[subcol_j]$. $opt_{ij} = 1$ indicates that x^*_{ij} and $succ^*_{ij}$ are already obtained, and 0 otherwise. $subrow_i = 1$ if the subprocedure $Get\_D\_U(i)$ has already been run, and 0 otherwise. $subcol_j = 1$ if the subprocedure $Get\_D\_L(j)$ has already been run, and 0 otherwise. Each application of $Get\_D\_U(i)$ gives shortest distances in G'_U from node i to each node $t > i$, and $Get\_D\_L(j)$ gives shortest distances in G'_L from each node $s > j$ to node j. These indicator arrays are used to avoid repeated computations in procedure Get_D. For example, to compute x^*_{23} and x^*_{13} in Figure 7(c), we only need one application of $Get\_D\_L(3)$ which updates x_{43} and x_{53}. Since the updated x_{43} and x_{53} can also be used in computing x^*_{23} and x^*_{13}, setting $subcol_3 := 1$ will avoid repeated applications of $Get\_D\_L(3)$.

To obtain a shortest path tree rooted at sink node t, we set $Q := \{(i,t) : i \neq t,\ i = 1,\ldots,n\}$. Thus setting $Q := \{(i,j)\ i \neq j,\ i = 1,\ldots,n,\ j = 1,\ldots,n\}$ is sufficient to solve an APSP problem. To trace shortest paths for q specific OD pairs $Q := \{(s_i,t_i) : i = 1,\ldots,q\}$, $DLU2(Q)$ can compute these q shortest paths by procedure Get_P without building q shortest path trees as required in Algorithm $DLU1$. If, however, only shortest distances are requested, we can skip procedure Get_P and avoid many unnecessary computations. More details will be discussed in following sections.

4.3.1 Procedure $Get\_D(s_i,t_i)$

This procedure can be viewed as a decomposed version of the three procedures $Acyclic\_L(j_0)$, $Acyclic\_U(i_0)$, and $Reverse\_LU(i_0,j_0,k_0)$ in Algorithm $DLU1$. Given an OD pair (s_i,t_i), it will directly compute its shortest distance in G without the need of all entries x^*_{st} satisfying $s > s_i$ and $t > t_i$ as required by Algorithm $DLU1$.

Procedure Get_D($\mathbf{s_i,t_i}$)
begin
    if $opt_{s_it_i} = 0$ ***then***
        if $subcol_{t_i} = 0$ ***then*** $Get\_D\_L(t_i)$; $subcol_{t_i} := 1$
        if $subrow_{s_i} = 0$ ***then*** $Get\_D\_U(s_i)$; $subrow_{s_i} := 1$
        $Min\_add(s_i,t_i)$;
end

Subprocedure Min_add($\mathbf{s_i,t_i}$)
begin
    $r_i := \max\{s_i,t_i\}$
    for $k = n$ down to $r_i + 1$ ***do***
        if $x_{s_it_i} > x_{s_ik} + x_{kt_i}$ ***then***
            $x_{s_it_i} := x_{s_ik} + x_{kt_i}$; $succ_{s_it_i} := succ_{s_ik_i}$
    $opt_{s_it_i} := 1$
end

Subprocedure $Get\_D\_L(t_i)$ gives the shortest distance in G'_L from each node $s > t_i$ to t_i, which corresponds to entries x_{st_i} obtained by $Acyclic\_L(j_0)$ in column t_i for each $s > t_i$. $Get\_D\_U(s_i)$ gives shortest distance in G'_U from node s_i to each node $t > s_i$, which corresponds to entries x_{s_it} obtained by $Acyclic\_U(i_0)$ in row s_i for each $t > s_i$.

In $DLU1$, to compute $x^*_{s_it_i}$ for OD pair (s_i, t_i), we have to apply $Reverse\_LU(s_i, t_i, r_i)$ where $r_i := \max\{s_i, t_i\}$, which not only updates $x^*_{s_it_i}$ but also updates other entries in the $(n-s_i)\times(n-t_i)$ submatrix x^*_{st} for all (s, t) satisfying $s_i < s < n$ and $t_i < t < n$. On the other hand in $DLU2$, the subprocedure $Min\_add(s_i, t_i)$ gives $x^*_{s_it_i}$ by min-addition operations only on entries x_{s_ik} and x_{kt_i} for each $k = (r_i + 1), \ldots, n$. Therefore $Min\_add(s_i, t_i)$ avoids more unnecessary updates than $Reverse\_LU(i_0, j_0, k_0)$.

For each requested OD pair (s_i, t_i), the number of triple comparisons in $Get\_D(s_i, t_i)$ is bounded by $\sum\limits_{k=t_i}^{n-1} |di(k)| + \sum\limits_{k=s_i}^{n-1} |uo(k)| + \sum\limits_{k=\max\{s_i,t_i\}+1}^{n} (1)$, or $O((n - \min\{s_i, t_i\})^2)$ in the worst case. As discussed in $DLU1$, reordering the node indices such that s_i and t_i are as large as possible may potentially reduce the computational work of Get_D. However, this may incur more fill-ins and make both G_LU and Get_D less efficient. Overall, when solving a MPSP problem of q OD pairs, this procedure will take $\sum\limits_{i=1}^{q} (\sum\limits_{k=t_i}^{n-1} |di(k)| + \sum\limits_{k=s_i}^{n-1} |uo(k)| + (n-\max\{s_i, t_i\}))$ triple comparisons which is $O(qn^2)$ time in the worst case. Thus in general it is better than $O(n^3)$, the complexity of last three procedures of Algorithm $DLU1$. Note that in the case where $q \approx O(n^2)$, this procedure will take $O(\min\{qn^2, n^3\})$ since at most we have to apply Get_D_L and Get_D_U n times, which takes $O(n^3)$, and Min_add $n(n-1)$ times, which takes $O(n^3)$ as well.

In the worst case when solving an APSP problem on a complete graph, $DLU2$ will perform $\frac{n(n-1)(n-2)}{6}$ triple comparisons on each subprocedure $Get_D_$L and $Get_D_$U as does $DLU1$. Its subprocedure Min_add will have $2\cdot \sum\limits_{i=1}^{n-1} \sum\limits_{j=1,j<i}^{n-1} (n-\max\{i, j\}) = \frac{n(n-1)(n-2)}{3}$ times of triple comparisons, which is the same as $Reverse\_LU(i_0, j_0, k_0)$.

Therefore $DLU2$ is as efficient as $DLU1$ in the worst case, but should be more efficient in general when number of OD pairs q is not large.

4.3.2 Procedure $Get\_P(s_i, t_i)$

This procedure iteratively calls procedure $Get\_D(k, t_i)$ to update x_{kt_i} and $succ_{kt_i}$ for any node k that lies on the shortest path from s_i to t_i. Thus given an OD pair (s_i, t_i), it will directly compute the shortest distance label and successor for each intermediate node on its shortest path in G, avoiding computations to other nodes that would be required in Algorithm $DLU1$.

Procedure Get_P($\mathbf{s_i}, \mathbf{t_i}$)
begin
    let $k := succ_{s_i t_i}$
    while $k \neq t_i$ ***do***
        $Get\_D(k, t_i)$;
        let $k := succ_{kt_i}$
end

Starting from the successor of s_i, we check whether it coincides with the destination t_i. If not, we update its shortest distance and successor, and then visit the successor. We iterate this procedure until eventually the destination t_i is encountered. Thus the entire shortest path is obtained since each intermediate node on this path has correct shortest distance and successor by the correctness of procedure Get_D (see Theorem 4.5(a) in Section 4.3.3).

On the other hand, in order to trace shortest paths for OD pairs in Q by Algorithm $DLU1$, we have to obtain the shortest path trees rooted at distinct destination nodes in Q, which is better than applying the Floyd-Warshall algorithm, but is still an "over-kill". Here in $DLU2$, procedure Get_P simply does the necessary computational work to retrieve each intermediate node lying on the shortest path. Therefore it resolves the computational redundancy problem of $DLU1$ and should be more efficient.

For a particular OD pair (s_i, t_i), obtaining $x^*_{s_i t_i}$ takes $O((n - \min\{s_i, t_i\})^2)$. It also makes $subcol_{t_i} = 1$ so that later each application of $Get\_D(s, t_i)$ only takes $\sum_{k=s}^{n-1} |uo(k)| + (n - \max\{s_i, t_i\})$ which is $O((n - s)^2)$ for each node s lying on the path from s_i to t_i. Suppose this shortest path contains p arcs and has lowest intermediate node i_0. Then $Get\_P(s_i, t_i)$ takes at most $O(p(n - i_0)^2)$ time. This is better than $DLU1$ since $DLU1$ requires $Acyclic\_L(t_i)$, $Acyclic\_U(1)$ and $Reverse\_LU(1, t_i, t_i\})$ which are $O(\sum_{t=t_i}^{n-2} \sum_{k=t}^{n-1} |di(k)|$

$+\sum_{s=1}^{n-2}\sum_{k=s}^{n-1}|uo(k)| + \sum_{k=t_i+1}^{n}(\left|\hat{ui(k)}\right| \cdot \left|\hat{do(k)}\right|)) = O((n-1)^3)$ and is dominated by procedure *Acyclic_U*(1). In the worst case where $p = (n-1)$ and $i_0 = 1$, procedure $Get\_P(s_i, t_i)$ takes $O(n^3)$ time.

When solving an APSP problem, complexity of *Get_P* bound remains $O(n^3)$ since it at most applies *Get_D_L* and *Get_D_U* n times, which takes $O(n^3)$ time, while the $Min\_add(s,t)$ for each $s = 1, \ldots, n$ and $t = 1, \ldots, n$ takes $O(n^3)$ time as well.

4.3.3 Correctness and properties of algorithm $DLU2$

First we show the correctness of this algorithm, then discuss some special properties.

Theorem 4.5. *(a) Procedure $Get\_D(s_i, t_i)$ will correctly compute $x^*_{s_it_i}$ and $succ^*_{s_it_i}$ for a given OD pair (s_i, t_i)*

(b) Procedure $Get\_P(s_i, t_i)$ will correctly compute $x^*_{st_i}$ and $succ^*_{st_i}$ for any node s that lies on the shortest path in G from node s_i to node $t_i \geq j_0$.

Proof. (a) After subprocedures $Get\_D\_L(t_i)$ and $Get\_D\_U(s_i)$, we will have obtained shortest paths in $H([1, r_i])$ from s_i to t_i where $r_i := \max\{s_i, t_i\}$. To obtain the shortest path in G from s_i to t_i, we only need to check those shortest paths from s_i to t_i that have highest node h for each $h = (r_i + 1), \ldots, n$. By Lemma 4.1(a), such a shortest path can be decomposed into two segments: from s_i to h and from h to t_i. Note that their shortest distances, x_{s_ih} and x_{ht_i}, will have been calculated by $Get\_D\_U(s_i)$ and $Get\_D\_L(t_i)$, respectively. Note also that their sum, $x_{s_ih} + x_{ht_i}$, represents the shortest distance in $H([1, h])$ from s to t among paths that pass through h.

Procedure $Get\_D(s_i, t_i)$ updates $x_{s_it_i} := \min_{h>r}\{x_{s_it_i}, x_{s_ih} + x_{ht_i}\}$. Now we will show this value corresponds to $x^*_{s_it_i}$. First we consider the case where $h = (r_i + 1)$. Since $x_{s_it_i}$ is the length of a shortest path in $H([1, r_i])$ and $x_{s_ir_i} + x_{r_it_i}$ represents the shortest distance in $H([1, r_i])$ from s_i to t_i among paths that pass through r_i, $\min\{x_{s_it_i}, x_{s_ir_i} + x_{r_it_i}\}$ therefore corresponds to the shortest distance in $H([1, r_i])$ from s_i to t_i. Applying the same argument on $\min\{x_{s_it_i}, x_{s_ih} + x_{ht_i}\}$ for $h = (r_i + 2), \ldots, n$, shows that this value will correspond to the shortest distance in $H([1, n])$ from s to t, which is in fact $x^*_{s_it_i}$, the shortest distance in G from s to t.

Similarly we can show the successor $succ_{s_i t_i}$ is correctly updated in this procedure.

(b) Since we only apply $Get\_P(s_i, t_i)$ when $x^*_{s_i t_i} < \infty$, the shortest path from s_i to t_i is well defined. Whenever an intermediate node k that lies on the shortest path from s_i to t_i is encountered, $Get\_D(k, t_i)$ will return $x^*_{kt_i}$ and $succ^*_{kt_i}$. Then the procedure goes on to the successor of node k. By induction, when t_i is eventually encountered, we will have updated $x^*_{kt_i}$ and $succ^*_{kt_i}$ for each node k on the shortest path from s_i to t_i. □

Like in Algorithm $DLU1$, we can save half of the storage and computational burden when applying Algorithm $DLU2$ to solve an APSP problem on an undirected graph. Although $DLU2$ introduces one $n \times n$ array $[opt_{ij}]$ and two $n \times 1$ array $[subrow_i]$ and $[subcol_j]$ than $DLU1$, it does not need arc adjacency arrays $ui(i)$, $\hat{ui}(i)$, $do(i)$ and $\hat{do}(i)$ for each node i which saves up to $2n^2$ storage. Thus in terms of storage requirement, $DLU2$ requires approximately the same storage as does $DLU1$.

Overall, the complexity of Algorithm $DLU2(Q)$ is $O(\sum\limits_{k=1}^{n-2} (|di(k)| \cdot |uo(k)|) + \sum\limits_{i=1}^{q} (\sum\limits_{k=t_i}^{n-1} |di(k)| + \sum\limits_{k=s_i}^{n-1} |uo(k)| + (n - \max\{s_i, t_i\})) + O(qp(n-i_0)^2))$ where q is number of requested OD pairs, p is the maximal number of arcs contained among all the requested shortest paths, and i_0 is the lowest intermediate node appeared among all the requested shortest paths. This time bound is $O(n^3)$ in the worst case, mostly contributed by procedure G_LU when $q < n^2$. When solving an APSP problem on a complete graph, we can skip procedure Get_P. The other two procedures, G_LU and Get_D, will take $\frac{1}{3}$ and $\frac{2}{3}$ of the total $n(n-1)(n-2)$ triple comparisons respectively, which is as efficient as Algorithm $DLU1$ and the Floyd-Warshall algorithm in the worst case.

All the discussion of node ordering in $DLU1$ also applies to $DLU2$. In other words, techniques to reduce the fill-in arcs in G_LU or to make the indices of the requested origin and destination nodes as large as possible can similarly reduce the computational work of $DLU2$.

In general, when solving a MPSP problem of $q < n^2$ OD pairs, Algorithm $DLU2$ saves more computational work in the last procedures than Algorithm $DLU1$. Unlike $DLU1$ which has to compute the full shortest path tree rooted at t so that the shortest path for a specific OD pair (s,t) can be traced, $DLU2$ can retrieve such a path by successively trespassing each intermediate node on that path, and thus it is more efficient.

4.4 Summary

In this chapter we propose two new algorithms called $DLU1$ and $DLU2$ that are suitable for solving MPSP problems. Although their worst case complexity $O(n^3)$ is no more efficient than other algebraic APSP algorithms such as Floyd-Warshall [113, 304] and Carré's [64, 65] algorithms, our algorithms can, in practice, avoid significant computational work in solving MPSP problems.

First obtaining the shortest distances from or to the last node n, algorithm $DLU1(i_0,$ $j_0,$ $k_0)$ systematically obtains shortest distances for all node pairs (s,t) such that $s \geq k_0$, $t \geq j_0$ or $s \geq i_0$, $t \geq k_0$ where $k_0 := \min_i\{\max\{s_i, t_i\}\}$. By setting $i_0 := 1$, it can be used to build the shortest path trees rooted at each node $t \geq k_0$. When solving MPSP problems, this algorithm may be sensitive to the distribution of requested OD pairs and the node ordering. In particular, when the requested OD pairs are closely distributed in the right lower part of the $n \times n$ OD matrix, algorithm $DLU1$ can terminate much earlier. On the other hand, scattered OD pairs might make the algorithm less efficient, although it will still be better than other APSP algorithms. A bad node ordering may require many "fill-ins." These fill-ins make the modified graph denser, which in turn will require more triple comparisons when applying our algorithms. Such difficulties may be resolved by reordering the node indices so that the requested OD pairs are grouped in a favorable distribution and/or the required number of fill-in arcs is decreased.

Algorithm $DLU2$ attacks each requested OD pair individually, so it is more suitable for problems with a scattered OD distribution. It also overcomes the computational inefficiency that algorithm $DLU1$ has in tracing shortest paths. It is especially efficient for solving a special MPSP problem of n OD pairs (s_i, t_i) that correspond to a matching. That is, each node appears exactly once in the source and sink node set but not the same time. Such an MPSP problem requires as much work as an APSP problem using most of the shortest path algorithms known nowadays, even though only n OD pairs are requested.

Our algorithms (especially $DLU2$) are advantageous when only the shortest distances between some OD pairs are required. For a graph with fixed topology, or a problem with fixed requested OD pairs where shortest paths have to be repeatedly computed with different numerical values of arc lengths, our algorithms are especially beneficial since we may do a preprocessing step in the beginning to arrange a good node ordering that favors our algorithms. These problems appear often in real world applications. For example, when solving the origin-destination multicommodity network flow problem (ODMCNF) using Dantzig-Wolfe decomposition and column generation[36], we generate columns by solving sequences of shortest path problems between some fixed OD pairs where the arc cost changes in each stage but the topology and requested OD pairs are both fixed. Also, in the computation of parametric shortest paths where arc length is a linear function of some parameter, we may solve shortest distances iteratively on the same graph to determine the critical value of the parameter.

Our algorithms can deal with graphs containing negative arc lengths and detect negative cycles as well. The algorithms save storage and computational work for problems with special structures such as undirected or acyclic graphs.

Although we have shown the superiority of our algorithms over other algebraic APSP algorithms, it is, however, still not clear how our algorithms perform when they are compared with modern SSSP algorithms empirically. Like all other algebraic algorithms in the literature, our algorithms require $O(n^2)$ storage which makes them more suitable for dense graphs. We introduced techniques of sparse implementation that avoid nontrivial triple comparisons, but they come with the price of extra storage for the adjacency data structures.

A more thorough computational experiment to compare the empirical efficiency of $DLU1$ and $DLU2$ with many modern SSSP and APSP algorithms is conducted in Chapter 5, in which we introduce additional sparse implementation techniques that lead to promising computational results.

CHAPTER V

IMPLEMENTING NEW MPSP ALGORITHMS

Many efficient SSSP and APSP algorithms and their implementations have been proposed in the literature. However, only few of them are targeted to solving the MPSP problems or problems with fixed topology but changeable arc lengths or requested OD pairs. These problems of course can be solved by repeated SSSP algorithms. Other methods such as the LP-based reoptimization algorithms (see Section 3.5.2) take advantage of previous optimal shortest paths and perform either primal network simplex methods (when arc lengths change) or dual network simplex methods (when OD pairs change). More recent computational experiments [58] indicate these reoptimization algorithms are still inferior to the repeated SSSP algorithms which repeatedly solve shortest path trees for different requested origins (or destinations, depending on which one has smaller cardinality).

A practical implementation of Carré's algorithm [65] by Goto et al. [150] tries to exploit the sparseness and topology of the networks. For networks with fixed topology, their implementation first does a preprocessing procedure to identify a good node pivoting order so that the fill-ins in the LU decomposition phase are decreased. To avoid unnecessary triple comparisons, they record all the nontrivial triple comparisons in the LU decomposition, forward elimination and backward substitution phases, and then generate an ad hoc APSP code. Their method only stores $O(m)$ data structures, which is the same as other combinatorial SSSP algorithms but is better than $O(n^2)$ as required in general algebraic algorithms. However, this comes with the price of storing the long codes of nontrivial triple comparisons, and may require more total hardware storage. Even worse, the long codes may not be compilable for some compilers.

In particular, we have tested a 1025-node, 6464-arc sparse graph and generated a $500MB$ long code using the code generation algorithm of Goto et al. [150]. The code we generated could not be compiled even on a fast Sun workstation with $1GB$ memory using *gcc*, a C compiler by GNU, with optimization tags. If instead we only store the arc index of all the triple comparisons, the code will be short but still need temporary storage around $200MB$ to store these indices. Therefore, code generation is not practical for large networks.

In this chapter, we observe that Carré's algorithm can be implemented combinatorially which resolves the need of a huge storage quota by the code generation. Furthermore, we can decompose and truncate Carré's algorithm according to the indices of the distinct origins/destinations of the requested OD pairs. Section 5.1 introduces some notation and definitions appearing in this chapter. Section 5.2 describes major procedures of our algorithm SLU, a sparse combinatorial implementation of Carré's algorithm. Section 5.3 gives detailed implementation issues, and techniques for speeding up SLU. Implementations of algorithm $DLU2$, our proposed MPSP algorithm that appeared in Section 4.3, are given in Section 5.4. Computational experiments including a sparse implementation of the Floyd-Warshall algorithm, many state-of-the-art SSSP codes written by Cherkassky et al. [74], and networks that we generate, are presented in Section 5.5. Section 5.6 shows results of our computational experiments and draws conclusions.

5.1 Notation and definition

Most of the notation and definitions appearing in this chapter can be found in previous chapters. In particular, see Section 4.2.1 for the definition of the augmented graph $G'$and its induced subgraphs G'_L and G'_U, triple comparison, and fill-in arcs. See Section 4.1 for the definition of arc adjacency lists $ui(i)$, $uo(i)$, $di(i)$, and $do(i)$, and the subgraph denoted by $H(S)$ induced on the node set S. See Section 3.4.1.1 for Carré's algorithm, Section 4.2 for algorithm $DLU1$, and Section 4.3 for algorithm $DLU2$.

Let A' denote the arc set of G' with cardinality $|A'| = m'$. Let $[a_{ij}]$ denote a $n \times n$ matrix with m' nonzero entries, each of which represents the arc index of an arc in G'. We create four arrays of size m', $head(a_{ij})$, $tail(a_{ij})$, $c(a_{ij})$ and $succ(a_{ij})$, to store the head, tail, arc length and successor for each arc $(i, j) \in A'$ with index a_{ij}.

Instead of maintaining the $n \times n$ distance matrix $[x_{ij}]$ and successor matrix $[succ_{ij}]$ as algorithm DLU does in Section 3.2, for each distinct destination node j, algorithm SLU uses two n dimensional vectors $d_n(i)$ and $succ_n(i)$ to denote the shortest distance from each node i to node j and the successor of each node i in the shortest path to node j, respectively. A $n \times 1$ indicator vector $label(i)$ indicates whether a node i is labeled ($label(i) = 1$) or not ($label(i) = 0$). After all the requested shortest distance lengths and successors for the requested OD pairs with the same destination node j have been computed, $d_n(i)$ and $succ_n(i)$ will be reset for the next destination node.

5.2 *Algorithm* SLU

Algorithm SLU can be viewed as an efficient sparse implementation of Carré's algorithm which resembles Gaussian elimination. The algorithm first performs a preprocessing procedure, $Preprocess$, to determine a good node ordering to reduce the number of fill-ins created by LU decomposition. Using the new node ordering, the topology (i.e., $ui(k)$, $uo(k)$, $di(k)$, and $do(k)$ for each node k) of the augmented graph G' is symbolically created. Suppose we want to compute shortest path lengths for a set of OD pairs $Q = \{(s_i, t_i) : i = 1, \ldots, q\}$ which contain $\hat{q}$ distinct destination nodes. For each distinct destination node $\hat{t}_i$, $Preprocess$ also determines the lowest indexed origin node $s_{\hat{t}_i}$ that appears in Q. That is, for each distinct $\hat{t}_i$, $s_{\hat{t}_i} := \min_i \{s_i : (s_i, \hat{t}_i) \in Q\}$.

Algorithm 7 SLU$(\mathbf{Q} := \{(\mathbf{s_i}, \mathbf{t_i}) : \mathbf{i} = \mathbf{1}, \ldots, \mathbf{q}\})$

```
begin
    Preprocess;
    G_LU0;
    for each distinct destination node t̂_i do
        reset label(k) = 0, d_n(k) = M, succ_n(k) = 0 ∀k ∈ N \ {t̂_i}; d_n(t̂_i) = 0
        G_Forward(t̂_i);
        G_Backward(s_{t̂_i}, t̂_i);
end
```

Based on the topology of G', SLU does a Gaussian elimination procedure G_LU0, and $\hat{q}$ iterations of procedure $G\_Forward(\hat{t}_i)$ followed by procedure $G\_Backward(s_{\hat{t}_i}, \hat{t}_i)$. All these three major procedures are "combinatorial" in a sense that they only operate on nodes and arcs of G'. In particular, for nodes $k = 1, \ldots, (n-2)$, G_LU0 scans each arc of $di(k)$ (with tail node s) and each arc of $uo(k)$ (with head node t) and updates the length and successor of arc (s, t) in G'. Procedure $G\_Forward(\hat{t}_i)$ can be viewed as computing shortest path lengths from each node $s > \hat{t}_i$ to node $\hat{t}_i$ on the acyclic induced subgraph G'_L. Based on the distance label computed by Procedure $G\_Forward(\hat{t}_i)$ for each node $s > \hat{t}_i$, procedure $G\_Backward(s_{\hat{t}_i}, \hat{t}_i)$ computes shortest distance lengths in G'_U for nodes $s = (n-1), \ldots, s_{\hat{t}_i}$. Thus, unlike other shortest path algorithms that work on the original graph G, algorithm SLU works on the augmented graph G'. The sparser the augmented graph G' is, the more efficient SLU becomes.

After application of $G\_Backward(s_{\hat{t}_i}, \hat{t}_i)$, we have computed the shortest distance $d^*_{\hat{t}_i}(s)$ for every node pair $(s, \hat{t}_i)$ satisfying $s \geq s_{\hat{t}_i}$. Therefore after $\hat{q}$ iterations of $G\_Forward(\hat{t}_i)$ and $G\_Backward(s_{\hat{t}_i}, \hat{t}_i)$ for each distinct destination node $\hat{t}_i$, algorithm SLU will give the shortest distance $d^*_{t_i}(s_i)$ for all the requested OD pairs (s_i, t_i) in Q.

5.2.1 Procedure *Preprocess*

This procedure determines the topology of the augmented graph G' by first determining a good node ordering and then performing a symbolic run of the procedure G_LU0 which determines all the arc adjacency data structures of G' (i.e., $di(k)$, $uo(k)$ and $ui(k)$ for each node k). In particular, G_LU0 requires $di(k)$ and $uo(k)$, $G\_Forward$ requires $di(k)$, and $G\_Backward$ requires $ui(k)$ for each node k. $di(k)$, $uo(k)$ and $ui(k)$ store the indices of the arcs that point up-inwards, down-inwards and up-outwards for each node k. These arc indices can be computed beforehand in *Preprocess* so that only $O(m')$ entries are required instead of $O(n^2)$ entries.

Procedure Preprocess
begin
    Decide a node ordering $perm(k)$ for each node k;
    Symbolic execution of procedure G_LU to determine the arc adjacency list
        $di(k)$, $uo(k)$, and $ui(k)$, for each node k of the augmented graph G' ;
    for each distinct destination node $\hat{t}_i$ ***do***
        if shortest paths need to be traced ***then***
            set $s_{\hat{t}_i} := 1$
        else set $s_{\hat{t}_i} := \min_i \{s_i : (s_i, \hat{t}_i) \in Q\}$
    Initialize: $\forall (s,t) \in A'$, ***if*** $(s,t) \in A$ with index a_{st} ***then***
                $c(a_{st}) := c_{st}$; $succ(a_{st}) := t$
            else $c(a_{st}) := M$; $succ(a_{st}) := 0$
end

The node ordering may be computed by many common techniques used in linear algebra to reduce fill-ins in LU decomposition. Most of the ordering techniques are based on the rationale of reducing fill-ins in the LU decomposition. In terms of path algebra, these methods try to create fewer artificial arcs when constructing the augmented graph G'. This rationale is especially effective for APSP problems since fewer artificial arcs will incur fewer triple comparisons in all of the three major procedures of SLU.

For MPSP problems, another ordering rationale based on the requested OD pairs may also be effective. In particular, the LU decomposition procedure does more triple comparisons for arcs with head and tail that are high-indexed. Thus, shortest distances between nodes with higher indices are usually computed earlier than nodes with lower indices. In other words, we may save some computational work by using an ordering that permutes nodes of the requested OD pairs to be as close and high as possible. Since such ordering rationale may in fact incur more fill-ins in the LU decomposition phase and is too problem-dependent and intractable, we do not use this ordering in our computational tests.

The node ordering is a permutation function $perm(k)$ that permutes node k in the original ordering to node $perm(k)$ in the new ordering. We also maintain an inverse permutation function, $iperm(k)$, where node k in the new ordering corresponds to node $iperm(k)$ in the original ordering.

After choosing a good ordering $perm$, we can construct the augmented graph G' by a symbolic run of the procedure G_LU0. For convenience, all the notation referring to node index in this chapter is in the new ordering. That is, when we say $i > j$, we actually mean $perm(i) > perm(j)$.

In the symbolic execution of G_LU0, for each arc (i, j) in G', we store its index (a_{ij}), head $(head(a_{ij}) = j)$, tail $(tail(a_{ij}) = i)$, and length $(c(a_{ij}) = c_{ij}$ if arc (i, j) is in the original graph, or otherwise $c(a_{ij}) = M$, a very large number). We also store the topology information of G' in $di(k)$, $ui(k)$ and $uo(k)$ for each node k.

The most two expensive operations in procedure $Preprocess$ are (1) identifying a fill-in reducing ordering, which may be NP-hard and (2) the symbolic run of procedure G_LU0.

We do not take the preprocessing time into consideration when comparing with other shortest path algorithms in our computational experiments. This may sound unfair; however, if we consider our problem as solving a MPSP on a graph with fixed topology but changeable arc lengths many times for days or even for years, the one-time execution of the preprocessing procedure would indeed be negligible.

In the initialization step, we first read the result of the preprocessing. That is, we read $perm(k)$, $iperm(k)$, $di(k)$, $ui(k)$ and $uo(k)$ for each node k, and $head(a_{ij})$, $tail(a_{ij})$, and $x(a_{ij})$ for each arc (i, j) in G'. We initiate $succ_n(a_{ij}) = j$ for each arc (i, j) in A.

5.2.2 Procedure G_LU0

This procedure is the same as the procedure G_LU introduced in Section 4.2.1, except it uses the $m' \times 1$ arrays $c(a_{ij})$ and $succ(a_{ij})$ to store the length and successor of arc a_{ij} instead of the $n \times n$ arrays x_{ij} and $succ_{ij}$ of algorithm DLU.

Procedure G_LU0
begin
    for $k = 1$ to $n - 2$ ***do***
        for each arc $(s, k) \in di(k)$ with index a_{sk} ***do***
            for each arc $(k, t) \in uo(k)$ with index a_{kt} ***do***
                if $s = t$ and $c(a_{sk}) + c(a_{kt}) < 0$ ***then***
                    Found a negative cycle; ***STOP***
                if $s \neq t$ ***then***
                    let arc (s, t) have index a_{st} ;
                    if $c(a_{st}) > c(a_{sk}) + c(a_{kt})$
                        $c(a_{st}) := c(a_{sk}) + c(a_{kt})$; $succ(a_{st}) := succ(a_{sk})$;
end

Note that this procedure can detect negative cycle. The total number of triple comparisons is bounded by $O(\sum_{k=1}^{n-2}(|di(k)| \cdot |uo(k)|))$, or $O(n^3)$ in the worst case. It is $\frac{n(n-1)(n-2)}{3}$ on a complete graph.

5.2.3 Procedure $G\_Forward(\hat{t}_i)$

This procedure is exactly the same as the subprocedure $Get\_D\_L(\hat{t}_i)$ introduced in Section 4.2.2 which gives the shortest distance $d_n(s)$ in G'_L from each node $s > \hat{t}_i$ to node $\hat{t}_i$. It is a sparse implementation of triple comparisons $s \to k \to \hat{t}_i$ for any node s and node k satisfying $s > k > \hat{t}_i$.

Procedure G_Forward$(\hat{t}_i)$
begin
  initialize $d_n(\hat{t}_i) := 0$, $d_n(k) := M \ \forall\ k \neq \hat{t}_i$
  put node $\hat{t}_i$ in $LIST$
  while $LIST$ is not empty ***do***
    remove the lowest node k in $LIST$
    $label(k) = 1$
    for each arc $(s,k) \in di(k)$ with index a_{sk} ***do***
      if $s \notin LIST$, put s into $LIST$
      if $d_n(s) > d_n(k) + c(a_{sk})$ ***then***
        $d_n(s) := d_n(k) + c(a_{sk})$; $succ_n(s) := succ(a_{sk})$
end

The distance label $d_n(s)$ actually corresponds to the shortest distance in $H([1,s])$ from each node $s > \hat{t}_i$ to node $\hat{t}_i$ (see Corollary 4.2(a) in Section 4.2.5). Suppose the highest labeled node in this procedure has index $\widetilde{s}_{\hat{t}_i}$. It can be shown that there exists no path in G from any node $s > \widetilde{s}_{\hat{t}_i}$ to node $\widetilde{s}_{\hat{t}_i}$. Also, the distance label $d_n(\widetilde{s}_{\hat{t}_i})$ computed by this procedure in fact corresponds to the shortest distance (i.e., $d^*_n(\widetilde{s}_{\hat{t}_i})$) in G from node $\widetilde{s}_{\hat{t}_i}$ to node $\hat{t}_i$ (see Corollary 5.2 in Section 5.2.5).

The total number of triple comparisons in $G\_Forward(\hat{t}_i)$ is bounded by $\sum\limits_{k} |di(k)|$, where k are the index of labeled nodes. In the worst case, this bound is $\sum\limits_{k=\hat{t}_i}^{n-1} |di(k)|$ and will be $O(n^2)$ for a complete graph. In general, for a MPSP problem, this procedure requires $\sum\limits_{\hat{t}_i \in Q} \sum\limits_{k=\hat{t}_i}^{n-1} |di(k)|$ triple operations. When we are solving an APSP problem on a complete graph, the total number of triple comparisons incurred by this procedure will be $\frac{n(n-1)(n-2)}{6}$.

The key to speeding up this procedure is to choose the lowest labeled node in $LIST$ as quickly as possible. We detail three different implementations in Section 5.3.3.

5.2.4 Procedure $G\_Backward(s_{\hat{t}_i}, \hat{t}_i)$

Since none of the nodes with index higher than $\widetilde{s}_{\hat{t}_i}$, the highest labeled node after $G\_Forward$ $(\hat{t}_i)$, can reach node $\hat{t}_i$ in G (see Corollary 5.2(b) in Section 5.2.5), we only need to check nodes with index lower than or equal to $\widetilde{s}_{\hat{t}_i}$. The distance label $d_n(k)$ computed by $G\_Forward(\hat{t}_i)$ represents the shortest distance in $H([1,k])$ from node $k \geq \hat{t}_i$ to node $\hat{t}_i$. Starting from the highest labeled node $\widetilde{s}_{\hat{t}_i}$ and based on the previously computed distance label, $G\_Backward(s_{\hat{t}_i}, \hat{t}_i)$ efficiently computes shortest path lengths in G from each node $s \geq s_{\hat{t}_i}$ to node $\hat{t}_i$. It is a sparse implementation of triple comparisons $s \rightarrow k \rightarrow \hat{t}_i$ for any s and k such that $s_{\hat{t}_i} \leq s < k \leq \widetilde{s}_{\hat{t}_i}$.

Procedure G_Backward($\mathbf{s_{\hat{t}_i}}, \mathbf{\hat{t}_i}$)
begin
  put all the labeled nodes with index higher than or equal to $s_{\hat{t}_i}$ into $LIST$
  while $LIST$ is not empty ***do***
    remove the highest node k in $LIST$
    $label(k) = 1$
    for each arc $(s, k) \in ui(k)$ with index a_{sk} ***do***
      if $s \geq s_{\hat{t}_i}$ and $s \neq \hat{t}_i$ ***then***
        if $s \notin LIST$, put s into $LIST$; $label(s) = 1$
        if $d_n(s) > d_n(k) + c(a_{sk})$ ***then***
          $d_n(s) := d_n(k) + c(a_{sk})$; $succ_n(s) := succ(a_{sk})$
end

In particular, for any labeled node $s \geq s_{\hat{t}_i}$, there exist paths in G to node $\hat{t}_i$. Let node $\widetilde{s}$ be the highest node in the shortest path from s to $\hat{t}_i$. Then this path can be decomposed into two parts: $s \rightarrow \widetilde{s}$ and $\widetilde{s} \rightarrow \hat{t}_i$. The shortest distance $d^*_n(\widetilde{s})$ from $\widetilde{s}$ to $\hat{t}_i$ in the second part is already computed by $G\_Forward(\hat{t}_i)$. The shortest distance from s to $\widetilde{s}$ in the first part will be computed by $G\_Backward(s_{\hat{t}_i}, \hat{t}_i)$. Thus, this procedure sequentially computes $d^*_n(s)$ for $s = \widetilde{s}$, $\widetilde{s} - 1, \ldots, s_{\hat{t}_i}$.

The total number of triple comparisons in $G\_Backward(s_{\hat{t}_i}, \hat{t}_i)$ is bounded by $\sum\limits_k |ui(k)|$, where k are the index of the set of nodes that have ever been labeled. In the worst case, this bound is $\sum\limits_{k=s_{\hat{t}_i}+1}^{n} |ui(k)|$, and will be $O(n^2)$ for a complete graph. In general, for a MPSP problem, this procedure requires $\sum\limits_{s_{\hat{t}_i} \in Q} \sum\limits_{k=s_{\hat{t}_i}+1}^{n} |ui(k)|$ triple operations. When we are solving an APSP problem on a complete graph, $s_{\hat{t}_i} = 1$ for each $\hat{t}_i$ ($\hat{t}_i = 1, \ldots, n$), and the total number of triple comparisons incurred by this procedure will be $\frac{n(n-1)(n-2)}{2}$.

The key to speeding up this procedure is to choose the highest labeled node in $LIST$ as quickly as possible. We give three different implementations to speed up both $G\_Forward(\hat{t}_i)$ and $G\_Backward(s_{\hat{t}_i}, \hat{t}_i)$ in Section 5.3.3.

5.2.5 Correctness and properties of algorithm SLU

First we show the correctness of the algorithm, and then discuss some special properties of this algorithm. To prove its correctness, we will show how the procedures of SLU calculate shortest path lengths for various subsets of the requested OD pairs Q, and then demonstrate that every requested OD pair must be in one such subset.

Without loss of generality, we only discuss the case with one OD pair $Q = \{(s_{\hat{t}_i}, \hat{t}_i)\}$ since if there is more than one distinct requested destination node $\hat{t}_j$, we simply repeat the same procedures for each $\hat{t}_j$. Also, if there is more than one distinct origin node, say, $s_1, \ldots s_q$, for the same destination node $\hat{t}_i$, we only require the one with lowest index $s_{\hat{t}_i} = \min\{s_1, \ldots s_q\}$ for $G\_Backward(s_{\hat{t}_i}, \hat{t}_i)$. Thus it suffices to show that algorithm $SLU((s_{\hat{t}_i}, \hat{t}_i))$ computes the shortest path lengths in G for all the node pairs $(s, \hat{t}_i)$ satisfying $s \geq s_{\hat{t}_i}$. To trace the shortest path from any node s to node $\hat{t}_i$, we have to set $s_{\hat{t}_i} = 1$ and apply algorithm $SLU((1, \hat{t}_i))$.

We begin by specifying the set of OD pairs whose shortest path lengths will be calculated by G_LU0. In particular, G_LU0 will identify shortest path lengths for those requested OD pairs (s, t) whose shortest paths have all intermediate nodes with index lower than $\min\{s, t\}$.

Theorem 5.1. *A shortest path in G from s to t that has a highest node with index equal to $\min\{s, t\}$ will be reduced to arc (s, t) in G' by Procedure G_LU0.*

Proof. Same as Theorem 4.1 in Section 4.2.5. □

Corollary 5.1. *Procedure G_LU0 will correctly compute a shortest path for any node pair (s, t) in $H([1, \min\{s, t\}] \cup \max\{s, t\})$.*

Proof. This follows immediately from Theorem 5.1. □

Now, we specify the set of OD pairs whose shortest path lengths will be calculated by Procedure $G\_Forward(\hat{t}_j)$. In particular, this procedure will give shortest path lengths for OD pairs $(s, \hat{t}_j)$ satisfying $s > \hat{t}_j$ and these shortest paths have all intermediate nodes with index lower than s.

Theorem 5.2. *(a) A shortest path in G from node $s > t$ to node t that has s as its highest node corresponds to a shortest path from s to t in G'_L.*

(b) A shortest path in G from node $s < t$ to node t that has t as its highest node corresponds to a shortest path from s to t in G'_U.

Proof. Same as Theorem 4.2 in Section 4.2.5. □

Lemma 5.1. *(a) Any shortest path in G from s to t that has a highest node with index $h > \max\{s,t\}$ can be decomposed into two segments: a shortest path from s to h in G'_U, and a shortest path from h to t in G'_L.*

(b) Any shortest path in G from s to t can be determined by the shortest of the following two paths: (i) the shortest path from s to t in G that passes through only nodes $v \le r$, and (ii) the shortest path from s to t in G that must pass through some node $v > r$, where $1 \le r \le n$.

Proof. (a) This follows immediately by combining Corollary 5.2(a) and (b).

(b) It is easy to see that every path from s to t must either pass through some node $v > r$ or else not. Therefore the shortest path from s to t must be the shorter of the minimum-length paths of each type. □

Corollary 5.2. *(a) Procedure $G\_Forward(\hat{t}_j)$ will correctly compute shortest paths in $H([1,s])$ for all node pairs $(s,\hat{t}_j)$ such that $s > \hat{t}_j$.*

(b) Suppose the highest labeled node has index $\widetilde{s}_{\hat{t}_i}$ after procedure $G\_Forward(\hat{t}_j)$. Then

(i) there exists no path in G from node $s > \widetilde{s}_{\hat{t}_i}$ to node $\hat{t}_j$, and

(ii) $d_n(\widetilde{s}_{\hat{t}_i})$ computed by $G\_Forward(\hat{t}_j)$ represents the shortest path length $d^*_n(\widetilde{s}_{\hat{t}_i})$ from node $\widetilde{s}_{\hat{t}_i}$ to node $\hat{t}_j$.

Proof. (a) $G\_Forward(\hat{t}_j)$ computes shortest path length in G'_L rooted at node $\hat{t}_j$ from all other nodes $s > \hat{t}_j$. By Theorem 5.2(a), a shortest path in G'_L from node $s > \hat{t}_j$ to node $\hat{t}_j$ corresponds to a shortest path in G from s to $\hat{t}_j$ where s is its highest node since all other nodes in this path in G'_L have lower index than s. In other words, such a shortest path corresponds to the same shortest path in $H([1,s])$.

(b.i) Suppose there exists at least one path in G from node $s > \widetilde{s}_{\hat{t}_i}$ to node $\hat{t}_j$. Let $\widetilde{s}$ be the highest indexed intermediate node in the shortest path from s to $\hat{t}_j$. By Lemma 5.1(a), there exists a path in G'_L from $\widetilde{s}$ to $\hat{t}_j$; thus $\widetilde{s}$ will be labeled by $G\_Forward(\hat{t}_j)$. Since $\widetilde{s} \geq s > \widetilde{s}_{\hat{t}_i}$, $\widetilde{s}_{\hat{t}_i}$ will not be the highest labeled node, a contradiction. Thus no node with index higher than $\widetilde{s}_{\hat{t}_i}$ can reach node $\hat{t}_j$ in G.

(b.ii) By (b.i) we know there exists no path from node $\widetilde{s}_{\hat{t}_i}$ to any other node with higher index. Thus the shortest path in $H([1, \widetilde{s}_{\hat{t}_i}])$ corresponds to the shortest path in G. Thus $d_n(\widetilde{s}_{\hat{t}_i}) = d^*_n(\widetilde{s}_{\hat{t}_i})$. □

Finally, we demonstrate that procedure $G\_Backward(s_{\hat{t}_i}, \hat{t}_i)$ will correctly calculate all shortest path lengths for node pairs $(s, \hat{t}_i)$ satisfying $s \geq s_{\hat{t}_i}$. In particular, $G\_Backward(s_{\hat{t}_i}, \hat{t}_i)$ gives shortest path lengths for those requested OD pairs $(s, \hat{t}_i)$ whose shortest paths have some intermediate nodes with index higher than $\max\{s, \hat{t}_i\}$.

Theorem 5.3. *Suppose in the p^{th} iteration of procedure $G\_Backward(s_{\hat{t}_i}, \hat{t}_i)$, when the highest labeled node $\widetilde{s}^p_{\hat{t}_i}$ in LIST is removed , $d_n(\widetilde{s}^p_{\hat{t}_i}) = d^*_n(\widetilde{s}^p_{\hat{t}_i})$ is determined.*

Proof. By Corollary 5.2(c), $d_n(\widetilde{s}_{\hat{t}_i}) = d^*_n(\widetilde{s}_{\hat{t}_i})$ for the highest labeled node $\widetilde{s}_{\hat{t}_i}$ in the beginning of procedure $G\_Backward(s_{\hat{t}_i}, \hat{t}_i)$.

When node $\widetilde{s}_{\hat{t}_i}$ is removed from $LIST$, $G\_Backward(s_{\hat{t}_i}, \hat{t}_i)$ update $d_n(s)$ for each node s that connects to node $s_{\hat{t}_i}$ by an arc $(s, \widetilde{s}_{\hat{t}_i})$ in G'_U. That is, $d_n(s) = \min\{d_n(s), c(a_{s\widetilde{s}_{\hat{t}_i}}) + d^*_n(\widetilde{s}_{\hat{t}_i})\}$. Let $\widetilde{s}'_{\hat{t}_i}$ be the current highest labeled node in $LIST$. $d_n(\widetilde{s}'_{\hat{t}_i})$ represents the shortest distance in $H([1, \widetilde{s}'_{\hat{t}_i}])$ from node $\widetilde{s}'_{\hat{t}_i}$ to node $\hat{t}_i$. If there exists no arc $(\widetilde{s}'_{\hat{t}_i}, \widetilde{s}_{\hat{t}_i})$ in G'_U, then obviously $d_n(\widetilde{s}'_{\hat{t}_i}) = d^*_n(\widetilde{s}'_{\hat{t}_i})$ since there exists no path from node $\widetilde{s}'_{\hat{t}_i}$ to node $\hat{t}_i$ that has intermediate node with index higher than $\widetilde{s}'_{\hat{t}_i}$.

If there exists arc $(\widetilde{s}'_{\hat{t}_i}, \widetilde{s}_{\hat{t}_i})$ in G'_U, then $c(a_{\widetilde{s}'_{\hat{t}_i}\widetilde{s}_{\hat{t}_i}}) + d^*_n(\widetilde{s}_{\hat{t}_i})$ represents the shortest distance in $H([1, \widetilde{s}'_{\hat{t}_i}] \cup \widetilde{s}_{\hat{t}_i})$. By Lemma 5.1(b) with $s = \widetilde{s}'_{\hat{t}_i}$, $t = \hat{t}_i$ and $r = \widetilde{s}'_{\hat{t}_i}$, we can conclude $d_n(\widetilde{s}'_{\hat{t}_i}) = \min\{d_n(\widetilde{s}'_{\hat{t}_i}), c(a_{\widetilde{s}'_{\hat{t}_i}\widetilde{s}_{\hat{t}_i}}) + d^*_n(\widetilde{s}_{\hat{t}_i})\} = d^*_n(\widetilde{s}'_{\hat{t}_i})$.

Using similar arguments, suppose in the p^{th} iteration, we remove the highest labeled node $\widetilde{s}^p_{\hat{t}_i}$ from $LIST$. $d_n(\widetilde{s}^p_{\hat{t}_i}) = \min\{d_n(\widetilde{s}^p_{\hat{t}_i}), c(a_{\widetilde{s}^p_{\hat{t}_i}\widetilde{s}^{p-1}_{\hat{t}_i}}) + d^*_n(\widetilde{s}^{p-1}_{\hat{t}_i}), c(a_{\widetilde{s}^p_{\hat{t}_i}\widetilde{s}^{p-2}_{\hat{t}_i}}) + d^*_n(\widetilde{s}^{p-2}_{\hat{t}_i}), \ldots, c(a_{\widetilde{s}^p_{\hat{t}_i}\widetilde{s}'_{\hat{t}_i}}) + d^*_n(\widetilde{s}'_{\hat{t}_i}), c(a_{\widetilde{s}^p_{\hat{t}_i}\widetilde{s}_{\hat{t}_i}}) + d^*_n(\widetilde{s}_{\hat{t}_i})\} = d^*_n(\widetilde{s}^p_{\hat{t}_i})$ by induction. □

Corollary 5.3. *(a) Procedure $G\_Backward(s_{\hat{t_i}}, \hat{t_i})$ terminates when node $s_{\hat{t_i}}$ is the only node in $LIST$, and correctly computes $d^*_n(s)$ for each node $s \geq s_{\hat{t_i}}$.*
(b) To trace the shortest path for OD pair $(s, \hat{t_i})$, we have to initialize $s_{\hat{t_i}} := 1$ in the beginning of Algorithm SLU.

Proof. (a) By Theorem 5.3, if node $s \geq s_{\hat{t_i}}$ has been labeled by $G\_Backward(s_{\hat{t_i}}, \hat{t_i})$, its label $d_n(s) = d^*_n(s)$. For a unlabeled node $s \geq s_{\hat{t_i}}$, there exists no path from s to $\hat{t_i}$ in G', which means no path exists from s to $\hat{t_i}$ in G; thus its distance label remains M (i.e., infinity).

(b) The entries $succ_n(s)$ for each $s \geq s_{\hat{t_i}}$ are updated in all procedures whenever a better path from s to $\hat{t_i}$ is identified. To trace the shortest path for a particular OD pair $(s_i, \hat{t_i})$, we need the entire $\hat{t_i}^{th}$ column of $[succ^*_{ij}]$ which contains information of the shortest path tree rooted at sink node $\hat{t_i}$. Thus we have to set $s_{\hat{t_i}} := 1$ so that procedure $G\_Backward(s_{\hat{t_i}}, \hat{t_i})$ will update entries $succ_n(s)$ for each $s \geq 1$. □

Algorithm SLU can easily identify a negative cycle. In particular, any negative cycle will be identified in procedure G_LU0.

Theorem 5.4. *Suppose there exists a k-node cycle C_k, $i_1 \to i_2 \to i_3 \to \ldots \to i_k \to i_1$, with negative length. Then, procedure G_LU0 will identify it.*

Proof. Same as Theorem 4.4 in Section 4.2.5 □

To summarize, suppose the shortest path in G from $s_{\hat{t_i}}$ to $\hat{t_i}$ contains more than one intermediate node and let r be the highest intermediate node in that shortest path. If $s_{\hat{t_i}} > \hat{t_i}$, there are three cases: (1) $r < \hat{t_i}$ (2) $\hat{t_i} < r < s_{\hat{t_i}}$ and (3) $r > s_{\hat{t_i}}$. The first case will be solved by G_LU0, second case by $G\_Forward(\hat{t_i})$, and third case by $G\_Backward(s_{\hat{t_i}}, \hat{t_i})$. If $s_{\hat{t_i}} < \hat{t_i}$, there are three cases: (1) $r < s_{\hat{t_i}}$ (2) $s < r < \hat{t_i}$ and (3) $r > \hat{t_i} > s$. The first case will be solved by G_LU0, and the second and third cases by $G\_Forward(\hat{t_i})$ and $G\_Backward(s_{\hat{t_i}}, \hat{t_i})$. In particular, $G\_Forward(\hat{t_i})$ only computes $d_n(k)$ for node $k > \hat{t_i}$. $G\_Backward(s_{\hat{t_i}}, \hat{t_i})$ sequentially computes $d^*_n(k)$ for node $k = n, \ldots, s_{\hat{t_i}}$.

To compute shortest path lengths for other OD pairs with the same destination $\hat{t_{i+1}} \neq \hat{t_i}$, we simply reset $d_n(k)$ and $succ_n(k)$ for each node k, and start $G\_Forward(\hat{t_{i+1}})$ and $G\_Backward(s_{\hat{t_{i+1}}}, \hat{t_{i+1}})$, without redoing G_LU0. Procedure G_LU0 only requires to be reapplied when arc lengths are changed.

When solving an APSP problem on an undirected graph, algorithm SLU becomes the same as Carré's algorithm [65], but it can save some storage and computational work compared with most shortest path algorithms. In particular, since the graph is symmetric, $ui(k) = do(k)$, and $uo(k) = di(k)$ for each node k. Therefore storing only the arcs (i, j) where $i > j$ in G' is sufficient. Special care may be required for paths using arcs in G'_U. For example, we need an additional m' dimensional successor vector, $succ'(a_{ij})$, to store the successor of arc (j, i) where $i > j$. In addition, Procedure G_LU0 can save half of its computation due to the symmetric structure.

For problems on acyclic graphs, we can reorder the nodes so that all the arcs after reordering point from higher nodes to lower nodes (or from lower nodes to higher nodes). Thus only the procedure $G\_Forward$ (or $G\_Backward$) is required, which is the same as the topological ordering method on acyclic graph.

Algorithm SLU is an efficient sparse implementation of Carré's algorithm. It depends on the topology of G'. The running time of these three procedures mainly depend on the m', the number of arcs in G'. In particular, the complexity of Algorithm SLU is $O(\sum\limits_{k=1}^{n-2}(|di(k)| \cdot |uo(k)|) + \sum\limits_{\hat{t_i} \in Q} \sum\limits_{k=\hat{t_i}}^{n-1} |di(k)| + \sum\limits_{s_{\hat{t_i}} \in Q} \sum\limits_{k=s_{\hat{t_i}}+1}^{n} |ui(k)|)$ which is $O(n^3)$ in the worst case. When solving an APSP problem on a complete graph, the three procedures G_LU0, $G\_Forward(k)$, and $G\_Backward(1, k)$ for $k = 1, \ldots, n$ will take $\frac{1}{3}$,$\frac{1}{6}$ and $\frac{1}{2}$ of the total $n(n-1)(n-2)$ triple comparisons respectively, which is as efficient as Carré's algorithm and Floyd-Warshall's algorithm in the worst case.

In some sense, computing the shortest distance for node pairs $(s, \hat{t_i})$ satisfying $s \geq s_{\hat{t_i}}$ can be viewed as computing the $(s, \hat{t_i})$ entry of the "inverse" of matrix $(I_n - C)$ in path algebra (see Section 3.4.1). After obtaining L and U from the LU decomposition (i.e., G_LU0), we can do the forward elimination (i.e., $G\_Forward(\hat{t_i})$) $L \otimes y_{\hat{t_i}} = b$, where the right hand side b is the $\hat{t_i}^{th}$ column of the identity matrix I_n. In particular, $y_{s\hat{t_i}}$ represents the shortest distance in G'_L from node s to node $\hat{t_i}$, where $s > \hat{t_i}$. The backward substitution (i.e., $G\_Backward(s_{\hat{t_i}}, \hat{t_i})$) $U \otimes x_{\hat{t_i}} = y_{\hat{t_i}}$ computes the shortest distance $x_{s\hat{t_i}}$ in G from node s to node $\hat{t_i}$, for $s = n, \ldots, s_{\hat{t_i}}$.

Carré's algorithm is algebraic and computes the ALL-ALL distance matrix. Algorithm SLU can be viewed as a truncated and decomposed sparse implementation of Carré's algorithm. In particular, SLU first "decomposes" Carré's algorithm columnwise by the requested destination nodes, and then the backward substitution is "truncated" as soon as all of the requested origins associated with the same destination are computed. Algorithm SLU terminates when all the requested OD pairs are computed; thus it skips many unnecessary triple comparisons that are required in Carré's algorithm, and is more efficient for solving MPSP problems.

Compared with algorithm $DLU2$ introduced in Section 4.3, algorithm SLU takes more advantage of the sparse topology of G'. Algorithm $DLU2$, on the other hand, requires more storage ($O(n^2)$ in general) than SLU ($O(m')$). Procedures G_LU and Get_D_L of algorithm $DLU2$ have the same number of total operations as the procedures G_LU0 and $G\_Forward$ of algorithm SLU. However, it is difficult to tell which of these two algorithms (i.e., algorithms $DLU2$ and SLU) is more efficient.

In particular, procedures Get_D_U and Min_add of algorithm $DLU2$ require operations on the dense matrix $[x_{ij}]$, but only do operations for requested OD entries. The procedure $G\_Backward$ of algorithm SLU requires operations on the sparse augmented graph G' only, but it requires the computation of shortest distances between higher-indexed node pairs before the computations on the requested OD entries. That is, to compute x^*_{st}, algorithm SLU must first compute x^*_{kt} for $k = n, \ldots, (s+1)$, but algorithm $DLU2$ can directly compute x^*_{st}. Another disadvantage of algorithm SLU is, when shortest paths have to be traced, it becomes "decomposed" like Carré's algorithm, which has to compute the whole shortest path tree like other SSSP algorithms. Algorithm $DLU2$, on the other hand, can trace any intermediate nodes very easily.

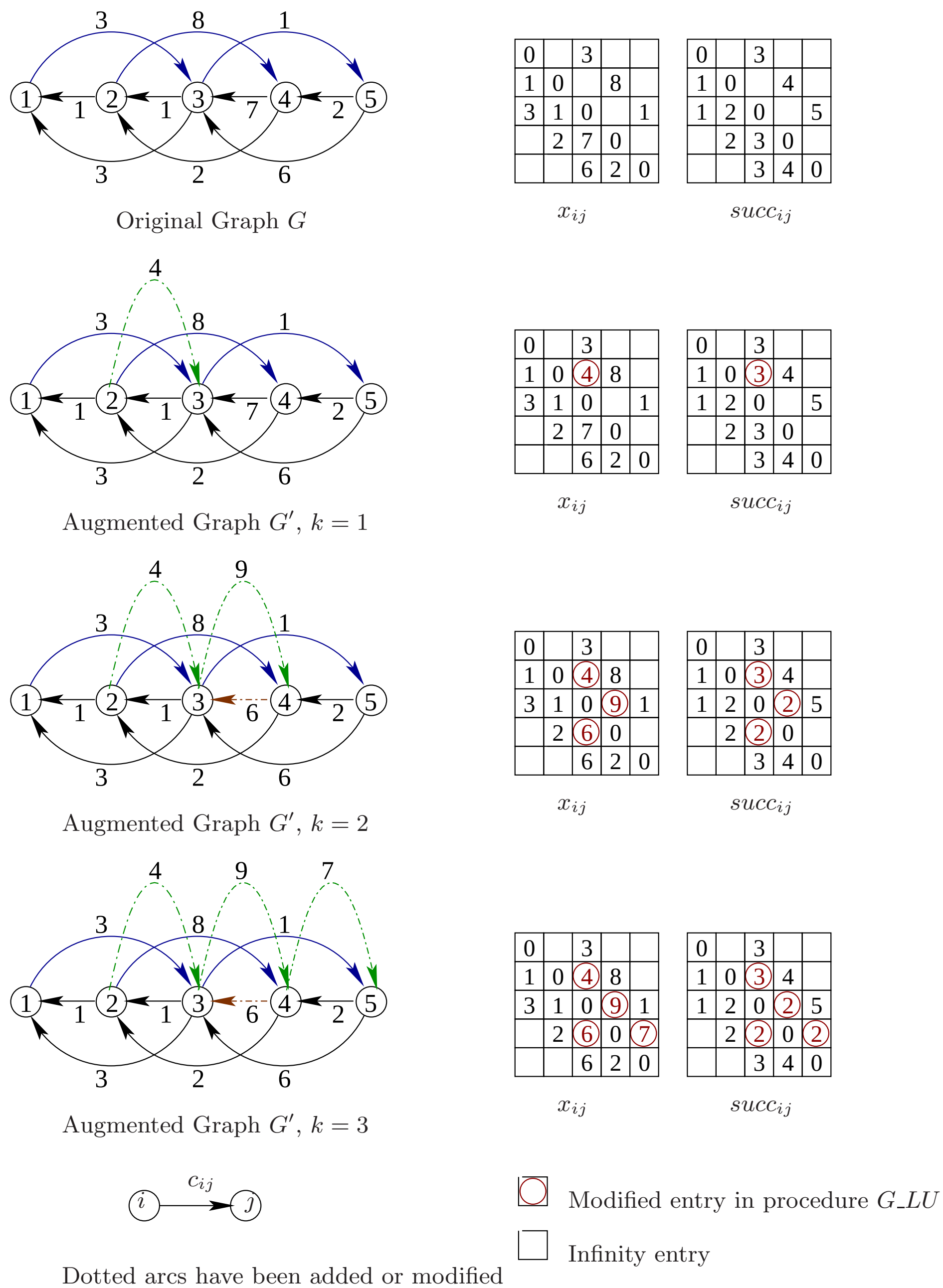
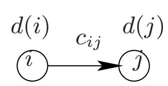


Figure 8: Illustration of procedure G_LU on a small example

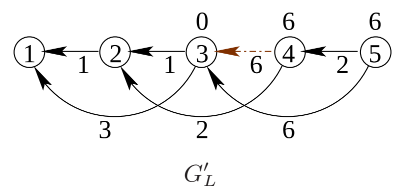
5.2.6 A small example

We cite a small example which computes x^*_{23} for a small network. Applying algorithm SLU, procedure G_LU0 creates the augmented graph G' which can be decomposed into G'_L and G'_U (see Figure 8). Procedure $G\_Forward(3)$ computes the shortest path length from each node $s > 3$ to node 3 in G'_L. Thus we obtain temporary distance label $d_n(2) = M$, $d_n(4) = 6$, and $d_n(5) = 6$. Note that $d^*_n(5) = d_n(5) = 6$ by Corollary 5.2(b.ii). Based on the current temporary distance labels, procedure $G\_Backward(2, 3)$ computes the shortest path length from each node $s \geq 2$ to node 3 in G'_U. In particular, $d^*_n(4) = \min\{d_n(4),\, c(a_{45}) + d^*_n(5)\} = \min\{6,\, 7+6\} = 6$, and $d^*_n(2) = \min\{d_n(2),\, c(a_{23}),\, c(a_{24}) + d^*_n(4),\, c(a_{25}) + d^*_n(5)\} = \min\{M,\, 4,\, 8 + 6,\, M + 6\} = 4$, where $c(a_{ij})$ is the length of arc (i, j) in G'. Figure 9 illustrates the operations of procedures $G\_Forward(3)$ and $G\_Backward(2, 3)$.



Node i	3	4	5
$d(i)$	0	6	6
$succ(i)$	-	2	3

$G\_Forward(3)$



Node i	2	4	5
$d(i)$	4	6	6
$succ(i)$	1	2	3

$G\_Backward(1, 3)$

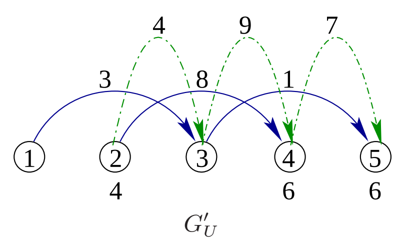


Figure 9: An example of all shortest paths to node 3

5.3 *Implementation of algorithm* SLU

Here we describe three efficient implementations of algorithm SLU.

5.3.1 Procedure *Preprocess*

Computing the optimal ordering that minimizes the fill-ins is NP-complete [272]. Many ordering techniques need to compute *degree production*, $i(k) \cdot o(k)$, where $i(k)$ ($o(k)$) denotes the in-degree (out-degree) of node k, for each node. The ordering techniques that minimize fill-ins have appeared in the literature of solving systems of linear equations and are applicable here since they are basically dealing with the same problem. In particular, we have implemented some of the following state-of-the-art ordering techniques:

1. **Natural ordering (NAT):**

   This ordering is the original one as read from the graph G (i.e., without any permutation).

2. **Dynamic Markowitz (DM):**

   This local minimum fill-in technique was first proposed by Markowitz [230], and has been proved very successful for general-purpose use.

   First we compute degree production for each node k. Choose the node $\hat{k}$ that has the smallest degree production from node 1 to node n, swap it with node 1, and update the degree production for the remaining $(n-1)$ nodes by removing arcs to/from $\hat{k}$. Then choose the node with smallest degree production among the remaining $(n-1)$ nodes, swap it with node 2, and update the degree production for the remaining $(n-2)$ nodes. Repeat the same procedure until finally no more node needs to be swapped.

   Note that if the graph is acyclic, the ordering obtained by Dynamic Markowitz rule will always make the nontrivial entries (i.e. finite entries) stored in the lower triangular part of the distance matrix. It is actually the so-called *topological ordering* for acyclic graph.

3. **Dynamic Markowitz with tie-breaking (DMT):**

   When using Markowitz criterion to choose node with the smallest degree production, we may often experience a tie. That is, two or more nodes may have the same degree production. Duff et al. [98] notice that a good tie-breaking strategy may substantially affect the quality of orderings. However, how to get the best tie-breaking strategy still remains unclear. Here we implement a simple tie-breaking strategy that probes fill-ins created by the two node candidates that are tied, and chooses the one with fewer fill-ins. In particular, suppose both node r and s have the same degree production. Our strategy is to first choose r, compute its fill-ins, then reset, and choose s to compute its fill-ins as well. Finally, we choose the one with fewer fill-ins as the pivot.

In our computational results, the quality of the ordering obtained by this tie-breaking strategy is generally only a little better than the one without the tie-breaking strategy.

4. **Static Markowitz (SM):**

   This is another simple local minimum fill-in technique by Markowitz [230], which is easy to implement but usually creates more fill-ins than its dynamic variants.

   In particular, this ordering is obtained by sorting the degree production for all nodes in the ascending order, without recalculation after each assignment.

5. **METIS_NodeND (MNDn):**

   METIS [195] is a software package for partitioning unstructured graphs and computing fill-reducing orderings of sparse matrices. It is based on *multilevel nested dissection* [196] that identifies a vertex-separator, moves the separator to the end of the matrix, and then recursively applies a similar process for each one of the other two parts.

   METIS_NodeND is a stand-alone function in this package which uses a multilevel paradigm to directly find a vertex separator.

6. **METIS_EdgeND (MNDe):**

   METIS_EdgeND is another stand-alone function of METIS. It first computes an edge separator using a multilevel algorithm to find a vertex separator. The orderings produced by METIS_EdgeND generally incur more fill-ins than those produced by METIS_NodeND.

7. **Multiple Minimum Degree on C^TC (MMDm):**

To solve a sparse system of linear equations $CX = b$, Liu [223] proposes a method called *multiple minimum degree* which computes a symmetric ordering that reduces fill-ins in the Cholesky factorization of C^TC. SuperLU, a software package of Demmel et al. [91], implements this method. We use SuperLU for our tests.

8. **Multiple Minimum Degree on $C^T + C$ (MMTa):**

   This method, also proposed by Liu [223], computes a symmetric ordering that reduces fill-ins in the Cholesky factorization of C^T+C. SuperLU also implements this method.

9. **Approximate Minimum Degree Ordering (AMD):**

   The multiple minimum degree method computes a symmetric ordering on either C^TC or $C^T + C$ which may be much denser than C, and time-consuming as well. Davis et al. [89] propose this approximate minimum degree ordering method which computes a better ordering in shorter time. We also imported this function from SuperLU for our tests.

In our tests, usually the dynamic Markowitz rule with tie-breaking produces the fewest fill-ins. Dynamic Markowitz rule without tie-breaking strategy usually performs as well as the one with tie-breaking. The advanced ordering techniques of METIS and SuperLU may get a good ordering very quickly, but the quality of the ordering they compute is not as good as the dynamic Markowitz rules. Since the time to compute a good ordering is not a major concern of our research, the dynamic Markowitz rules fit our needs better since they produce better orderings in reasonable time.

5.3.2 Procedure G_LU0

G_LU0 is the procedure to construct the augmented graph G' using arc adjacency lists $di(k)$ and $uo(k)$ for $k = 1 \ldots n-1$ in the new ordering. In particular, when we scan node k, we scan each arc (i,k) in $di(k)$ and each arc (k,j) in $uo(k)$, then check whether the length of arc (i,j) need to be modified or not. To do so, we need fast access to the index of each arc (i,j) in G'.

If we use the forward/backward star [3] implementation, it may take $O(n)$ time to search $di(i)$ and $ui(i)$ for a specific arc (i, j). Another alternative is to store an extra $n \times n$ index matrix, $[a_{ij}]$, whose (i, j) entry stores the index of arc (i, j). This implementation only takes $O(1)$ time to retrieve the index given i and j, but requires extra storage.

Since modern computers have lots of storage and G_LU0 does access arcs very frequently (up to $\frac{n(n-1)(n-2)}{3}$ for a complete graph), we choose the latter implementation for our tests. In particular, we store an $n \times n$ matrix $[a_{ij}]$ to record all the arc index of the augmented graph.

This implementation of G_LU0 takes $O(\sum_{k=1}^{n-2}(|di(k)| \cdot |uo(k)|))$ time.

5.3.3 Procedures $G\_Forward(\hat{t}_i)$ and $G\_Backward(s_{\hat{t}_i}, \hat{t}_i)$

In terms of storage, $G\_Forward(\hat{t}_i)$ only requires $di(k)$, and $G\_Backward(s_{\hat{t}_i}, \hat{t}_i)$ requires $ui(k)$. We also use an indicator function $label(k)$ for each node k, where $label(k) = 1$ means node k is labeled and 0 otherwise. Let $N^F_{\hat{t}_i}$ denote the set of nodes that have ever entered $LIST$ in procedure $G\_Forward(\hat{t}_i)$, and $N^B_{s_{\hat{t}_i},\hat{t}_i}$ be the set of nodes that have ever entered $LIST$ in procedure $G\_Backward(s_{\hat{t}_i}, \hat{t}_i)$. Note that if $s_{\hat{t}_i} \leq \hat{t}_i$ then $N^F_{\hat{t}_i} \subseteq N^B_{s_{\hat{t}_i},\hat{t}_i}$; otherwise $N^F_{\hat{t}_i} \supset N^B_{s_{\hat{t}_i},\hat{t}_i}$.

When we *scan* a node k in $G\_Forward(\hat{t}_i)$ (or $G\_Backward(s_{\hat{t}_i}, \hat{t}_i)$), we are in fact doing triple comparison $s \to k \to \hat{t}_i$ for arcs (s, k) in $di(k)$ (or $ui(k)$). When we scan an arc (s, k), we mark its tail node s as labeled. The processes in these two procedures are very similar. That is, select a lowest (or highest) node k from $LIST$, scan and label the tails of arcs in $di(k)$ (or $ui(k)$), and then select the next lowest (or highest) node to scan.

Both of these two procedures contain a sorting process which selects the lowest (or highest) node from $LIST$ to scan. To efficiently select these lowest (or highest) nodes from $LIST$, we propose three different implementations. The first implementation uses $label(k)$ to check whether a node is labeled or not, which is similar to Dial's implementation [93] of Dijkstra's algorithm. The second one is to maintain a heap that stores nodes in ascending order in $G\_Forward(\hat{t}_i)$, and stores nodes in descending order in $G\_Backward(s_{\hat{t}_i}, \hat{t}_i)$. The last implementation is to maintain two heaps, a min-heap for $G\_Forward(\hat{t_i})$ and a max-heap for $G\_Backward(s_{\hat{t_i}}, \hat{t_i})$.

In our tests, both the first and the third implementations run faster than the second one. However, comparisons between the first implementation and the third one depend on the platforms and the compilers. In particular, the first implementation generally runs faster on a Sun Solaris workstation, while the result is reversed on a Intel PC running Linux.

5.3.3.1 Bucket implementation of $G\_Forward(\hat{t_i})$ and $G\_Backward(s_{\hat{t_i}}, \hat{t_i})$:

Although there is no data structure named 'bucket' in this implementation, we use this name since it resembles Dial's bucket implementation on Dijkstra's algorithm. In this implementation, we use an indicator function, $label(k)$ (i.e., a bucket), for each node k to indicate whether it is labeled or not (or in a sense, to indicate whether it is "in the bucket" or not). In the beginning, no nodes are labeled. That is, $label(k) = 0$ for each node k.

In $G\_Forward(\hat{t_i})$, starting from the destination node $k = \hat{t_i}$, we scan arcs in $di(k)$, update the distance labels for their tails, and mark these tails as labeled. That is, $d_n(s) = \min\{d_n(s), c(a_{sk}) + d_n(k)\}$, and $label(tail(a_{sk})) = 1$ for each arc (s, k) in $di(k)$. After node k is scanned, we then search for the next lowest labeled node to scan. We repeat these processes until finally all the labeled nodes higher than $\hat{t_i}$ are scanned. At this moment, any node whose distance label $d_n(s)$ is finite must have been labeled. In particular, any labeled node has a finite distance label $d_n(s)$ to represent its shortest path length in G'_L to node $\hat{t_i}$. $G\_Forward(\hat{t_i})$ starts from bucket $k = \hat{t_i}$, scans all its down-inward arcs (s, k) and puts each tail node s into its bucket if bucket s is still empty. It then searches for the next nonempty bucket (i.e., next labeled node) in ascending order. The operations are repeated until the last nonempty bucket $\widetilde{s}_{\hat{t_i}}$ (i.e., the highest labeled node) has been scanned.

In $G\_Backward(s_{\hat{t_i}}, \hat{t_i})$, starting from the highest labeled node $k = \widetilde{s}_{\hat{t_i}}$, we scan arcs in $ui(k)$, update the distance labels for their tails, and mark these tails as labeled if they are higher than $s_{\hat{t_i}}$. That is, $d_n(s) = \min\{d_n(s), c(a_{sk}) + d_n(k)\}$, and $label(tail(a_{sk})) = 1$ for each arc (s, k) in $ui(k)$ satisfying $s > s_{\hat{t_i}}$. Then we search for the next highest labeled node to scan. We repeat these operations until finally node $s_{\hat{t_i}}$ is scanned. At this moment, we have finished the triple comparisons on all the arcs that are in some paths to $\hat{t_i}$ in G'_U; thus $G\_Backward(s_{\hat{t_i}}, \hat{t_i})$ is terminated. $G\_Backward(s_{\hat{t_i}}, \hat{t_i})$ starts from bucket $k = \widetilde{s}_{\hat{t_i}}$ (i.e., the highest labeled node), scans all its up-inward arcs (s, k) and puts each tail node s into its bucket if bucket s is still empty. It then searches for the next nonempty bucket (i.e., next labeled node) in descending order. The operations are repeated until all the nonempty buckets with index higher than $s_{\hat{t_i}}$ (i.e., all the labeled nodes $s > s_{\hat{t_i}}$) have been scanned.

This bucket implementation has to check (1) each node $k = \hat{t_i}, \dots, \widetilde{s}_{\hat{t_i}}$ in $G\_Forward(\hat{t_i})$, and (2) each node $k = (s_{\hat{t_i}} + 1), \dots, \widetilde{s}_{\hat{t_i}}$ in $G\_Backward(s_{\hat{t_i}}, \hat{t_i})$, at least once. Therefore, the complexity of this implementation is $O(\sum_{k=\hat{t_i}}^{\widetilde{s}_{\hat{t_i}}} (1) + \sum_{k \in N^F_{\hat{t_i}}} |di(k)|)$ for $G\_Forward(\hat{t_i})$ and $O(\sum_{k=s_{\hat{t_i}}+1}^{\widetilde{s}_{\hat{t_i}}} (1) + \sum_{k \in N^B_{s_{\hat{t_i}}, \hat{t_i}}} |ui(k)|)$ for $G\_Backward(s_{\hat{t_i}}, \hat{t_i})$. This is an easy and very efficient implementation in our computational experiments. However, time spent in detecting unlabeled nodes (i.e., empty buckets) may be further saved. In particular, if we maintain all the labeled nodes in some sorted data structure such as a binary heap, then each iteration of $G\_Forward(\hat{t_i})$ only removes the top node of a min-heap, and each iteration of $G\_Backward(s_{\hat{t_i}}, \hat{t_i})$ only removes the top node of a max-heap. Such heaps avoid checking unlabeled nodes, but require overhead in heap sort.

Next, we propose the following two implementations based on heaps.

5.3.3.2 Single heap implementation of $G\_Forward(\hat{t_i})$ and $G\_Backward(s_{\hat{t_i}}, \hat{t_i})$:

In this implementation we use a single binary heap data structure to extract the lowest (or highest) labeled node. A node is inserted into the heap only when it is labeled. By maintaining the heap, first sorted as a min-heap in $G\_Forward(\hat{t_i})$ and then as a max-heap in $G\_Backward(s_{\hat{t_i}}, \hat{t_i})$, we skip the scanning operations required by the bucket implementation for unlabeled nodes.

In $G\_Forward(\hat{t_i})$, we maintain the heap as a min-heap (so that every node is lower than its children). Starting from the destination node $k = \hat{t_i}$, we scan arcs in $di(k)$, update the distance labels of their tails, mark these tails as labeled, and put them into the min-heap. Then we extract the top node from the min-heap and repeat these procedures until the min-heap becomes empty, when we terminate $G\_Forward(\hat{t}_i)$.

To start $G\_Backward(s_{\hat{t}_i}, \hat{t}_i)$, first we have to put all the labeled nodes higher than $s_{\hat{t}_i}$ into the max-heap. Then we extract the top node k (which corresponds to the highest labeled node), mark it as unlabeled, scan arcs in $ui(k)$, update the distance labels of their tails, mark these tails as labeled, and put them into the max-heap if they are higher than $s_{\hat{t}_i}$. We repeat the same procedures until finally the max-heap becomes empty. At this moment, we have finished the triple comparisons on all the arcs that are in some paths to $\hat{t}_i$ in G'_U; thus $G\_Backward(s_{\hat{t}_i}, \hat{t}_i)$ is terminated.

This implementation only checks labeled nodes, but its overhead in heap sorting may be time-consuming. In $G\_Forward(\hat{t}_i)$, each node in $N^F_{\hat{t}_i}$ is extracted and inserted into the min-heap exactly once. Both extracting and inserting a node on a min-heap may take $O(\log \left|N^F_{\hat{t}_i}\right|)$. Thus, an $O(\left|N^F_{\hat{t}_i}\right| \log \left|N^F_{\hat{t}_i}\right|)$ time bound is obtained for all the min-heap operations in $G\_Forward(\hat{t}_i)$. This time bound is for the worst case. The average time should be much better since on average the depth of the heap is smaller than $\left|N^F_{\hat{t}_i}\right|$. Including the inevitable $\sum\limits_{k \in N^F_{\hat{t}_i}} |di(k)|$ triple comparisons, the complexity of this single heap implementation on $G\_Forward(\hat{t}_i)$ is $O(\left|N^F_{\hat{t}_i}\right| \log \left|N^F_{\hat{t}_i}\right| + \sum\limits_{k \in N^F_{\hat{t}_i}} |di(k)|)$

In the beginning of $G\_Backward(s_{\hat{t}_i}, \hat{t}_i)$, we need a loop to identify labeled nodes either from node $\hat{t}_i$ to the highest labeled node $\widetilde{s}_{\hat{t}_i}$ (if $s_{\hat{t}_i} < \hat{t}_i$), or from node $s_{\hat{t}_i}$ to $\widetilde{s}_{\hat{t}_i}$ (if $s_{\hat{t}_i} > \hat{t}_i$), which will take $O(\widetilde{s}_{\hat{t}_i} - \max\{s_{\hat{t}_i}, \hat{t}_i\})$ time. Inserting all the labeled nodes that are higher than $s_{\hat{t}_i}$ into the max-heap will take $O(\left|N^B_{s_{\hat{t}_i},\hat{t}_i}\right| \log \left|N^B_{s_{\hat{t}_i},\hat{t}_i}\right|)$ time. Each node in $N^B_{s_{\hat{t}_i},\hat{t}_i}$ is extracted exactly once, for a total of $O(\left|N^B_{s_{\hat{t}_i},\hat{t}_i}\right| \log \left|N^B_{s_{\hat{t}_i},\hat{t}_i}\right|)$ time. Including the inevitable $\sum\limits_{k \in N^B_{s_{\hat{t}_i},\hat{t}_i}} |ui(k)|$ triple comparisons, the complexity of this single heap implementation on $G\_Backward(s_{\hat{t}_i}, \hat{t}_i)$ is $O(\widetilde{s}_{\hat{t}_i} - \max\{s_{\hat{t}_i}, \hat{t}_i\} + \left|N^B_{s_{\hat{t}_i},\hat{t}_i}\right| \log \left|N^B_{s_{\hat{t}_i},\hat{t}_i}\right| + \sum\limits_{k \in N^B_{s_{\hat{t}_i},\hat{t}_i}} |ui(k)|)$. Again this is for the worst case. On average, extracting a node from or inserting a node into the max-heap may take time shorter than $O(\log \left|N^B_{s_{\hat{t}_i},\hat{t}_i}\right|)$ since the depth of the max-heap is usually smaller than $\left|N^B_{s_{\hat{t}_i},\hat{t}_i}\right|$.

This implementation only maintains a single heap which becomes empty in the beginning of $G\_Backward(s_{\hat{t_i}}, \hat{t_i})$. To insert those nodes that are labeled by $G\_Forward(\hat{t_i})$ into the max-heap in the beginning of $G\_Backward(s_{\hat{t_i}}, \hat{t_i})$, an overhead that checks $\widetilde{s}_{\hat{t_i}} - \max\{s_{\hat{t_i}}, \hat{t_i}\}$ nodes has to be made. If only a few of these $\widetilde{s}_{\hat{t_i}} - \max\{s_{\hat{t_i}}, \hat{t_i}\}$ nodes are labeled, this overhead will become inefficient. Next, we propose another way to avoid this overhead, at the cost of maintaining an additional heap data structure.

5.3.3.3 Two-heap implementation of $G\_Forward(\hat{t_i})$ and $G\_Backward(s_{\hat{t_i}}, \hat{t_i})$:

Instead of using a single heap, this implementation maintains two heaps, one min-heap for $G\_Forward(\hat{t_i})$, and one max-heap for $G\_Backward(s_{\hat{t_i}}, \hat{t_i})$.

In particular, after we extract the lowest labeled node k from the min-heap in $G\_Forward$ $(\hat{t_i})$, we insert that node into the max-heap right away, if $k \geq s_{\hat{t_i}}$. Therefore, after $G\_Forward(\hat{t_i})$, the max-heap is ready to start $G\_Backward(s_{\hat{t_i}}, \hat{t_i})$. $G\_Forward(\hat{t_i})$ spends $O(\min\{ \left|N^F_{\hat{t_i}}\right| \log \left|N^F_{\hat{t_i}}\right|, \left|N^B_{s_{\hat{t_i}}, \hat{t_i}}\right| \log \left|N^B_{s_{\hat{t_i}}, \hat{t_i}}\right| \})$ time inserting nodes into the max-heap, and $O(\left|N^F_{\hat{t_i}}\right| \log \left|N^F_{\hat{t_i}}\right|)$ time inserting and extracting nodes on the min-heap. Including the $O(\sum_{k \in N^F_{\hat{t_i}}} |di(k)|)$ triple comparisons, the overall complexity of $G\_Forward(\hat{t_i})$ is $O(\left|N^F_{\hat{t_i}}\right| \log \left|N^F_{\hat{t_i}}\right| + \sum_{k \in N^F_{\hat{t_i}}} |di(k)|)$. Similarly, $G\_Backward(s_{\hat{t_i}}, \hat{t_i})$ will take $O(\left|N^B_{s_{\hat{t_i}}, \hat{t_i}}\right| \log \left|N^B_{s_{\hat{t_i}}, \hat{t_i}}\right| + \sum_{k \in N^B_{s_{\hat{t_i}}, \hat{t_i}}} |ui(k)|)$ time.

Therefore this two-heap implementation has a better time bound but needs more storage than the single heap implementation.

5.3.4 Summary on different *SLU* implementations

We give three implementations of algorithm SLU: a bucket implementation ($SLU1$), a single heap implementation ($SLU2$), and a two-heap implementation ($SLU3$). All of these three implementation have the same procedure G_LU0. They differ by techniques of implementing procedures $G\_Forward$ and $G\_Backward$.

Table 7 compares the running time for these three implementations.

It is difficult to tell which implementation is theoretically better than the others. However, the bucket implementation seems to be faster in practice, according to our experiments in Section 5.6.2.

Table 7: Running time of different SLU implementations

	G_LU0	**G_Forward**($\hat{t_i}$)	**G_Backward**($\mathbf{s}_{\hat{t_i}},\hat{t_i}$)
bucket	$\sum_{k=1}^{n-2}(\lvert di(k)\rvert \cdot \lvert uo(k)\rvert)$	$\sum_{k=\hat{t_i}}^{\tilde{s}_{\hat{t_i}}}(1) + \sum_{k\in N^F_{\hat{t_i}}} \lvert di(k)\rvert$	$\sum_{k=s_{\hat{t_i}}+1}^{\tilde{s}_{\hat{t_i}}}(1) + \sum_{k\in N^B_{s_{\hat{t_i}},\hat{t_i}}} \lvert ui(k)\rvert$
single heap	$\sum_{k=1}^{n-2}(\lvert di(k)\rvert \cdot \lvert uo(k)\rvert)$	$\lvert N^F_{\hat{t_i}}\rvert \log \lvert N^F_{\hat{t_i}}\rvert + \sum_{k\in N^F_{\hat{t_i}}} \lvert di(k)\rvert$	$\tilde{s}_{\hat{t_i}} - \max\{s_{\hat{t_i}},\hat{t_i}\} + \lvert N^B_{s_{\hat{t_i}},\hat{t_i}}\rvert \log \lvert N^B_{s_{\hat{t_i}},\hat{t_i}}\rvert + \sum_{k\in N^B_{s_{\hat{t_i}},\hat{t_i}}} \lvert ui(k)\rvert$
two-heap	$\sum_{k=1}^{n-2}(\lvert di(k)\rvert \cdot \lvert uo(k)\rvert)$	$\lvert N^F_{\hat{t_i}}\rvert \log \lvert N^F_{\hat{t_i}}\rvert + \sum_{k\in N^F_{\hat{t_i}}} \lvert di(k)\rvert$	$\lvert N^B_{s_{\hat{t_i}},\hat{t_i}}\rvert \log \lvert N^B_{s_{\hat{t_i}},\hat{t_i}}\rvert + \sum_{k\in N^B_{s_{\hat{t_i}},\hat{t_i}}} \lvert ui(k)\rvert$

5.4 Implementation of algorithm $DLU2$

Algorithm $DLU2$ is one of our new MPSP algorithms proposed in Section 4.3. Here we describe two efficient implementations of algorithm $DLU2$. Algorithm $DLU2$ (see Section 4.3) contains three major procedures: $Preprocess$, G_LU (see Section 4.2.1) and $Get\_D(s_i, t_i)$ (see Section 4.3.1). The preprocessing ($Preprocess$) and LU decomposition (G_LU) procedures are similar to those of algorithm SLU.

All the operations of SLU are based on the arc adjacency lists of G' (i.e., $di(k)$, $uo(k)$, and $ui(k)$ for each node k). Each iteration of SLU updates only two $n \times 1$ arrays $d_n(k)$ and $succ_n(k)$ to report the shortest path. On the other hand, although algorithm $DLU2$ performs most of its operations based on the arc adjacency lists of G' (i.e., $di(k)$ and $uo(k)$ for each node k), it also updates two $n \times n$ arrays $[x_{ij}]$ and $[succ_{ij}]$.

In particular, the procedure G_LU and subprocedures $Get\_D\_L(t_i)$ and $Get\_D\_U(s_i)$ update $[x_{ij}]$ and $[succ_{ij}]$ by operations on arcs $di(k)$ and $uo(k)$ for each node k in G'. We implement $Min\_add(s_i, t_i)$ algebraically since the updated $[x_{ij}]$ and $[succ_{ij}]$ become denser and make "graphical" implementation of $Min\_add(s_i, t_i)$ less efficient.

To speed up the acyclic operations of subprocedures $Get\_D\_L(t_i)$ and $Get\_D\_U(s_i)$, we propose two implementations: bucket implementation and heap implementation.

5.4.1 Bucket implementation of $Get\_D\_L(t_i)$ and $Get\_D\_U(s_i)$

This is similar to the bucket implementation of $G\_Forward(t_i)$ in Section 5.3.3.1.

In $Get\_D\_L(t_i)$, starting from node $k = t_i$, we scan arcs in $di(k)$, update the distance

Algorithm 8 DLU2($\mathbf{Q} := \{(\mathbf{s_i}, \mathbf{t_i}) : \mathbf{i} = \mathbf{1}, \ldots, \mathbf{q}\}$)

begin
  $Preprocess$;
  Initialize:
      $opt_{ij} := 0 \ \forall \ i = 1, \ldots, n, \ j = 1, \ldots, n$
      $subrow_i := 0$; $subcol_i := 0 \ \forall \ i = 1, \ldots, n$
  G_LU;
  set $opt_{n,n-1} := 1$, $opt_{n-1,n} := 1$
  for $i = 1$ to q ***do***
    $Get\_D(s_i, t_i)$;
    if shortest paths need to be traced ***then***
      if $x_{s_i t_i} \neq \infty$ ***then***
        $Get\_P(s_i, t_i)$;
      else there exists no path from s_i to t_i
end

Procedure Preprocess
begin
  Decide a node ordering $perm(k)$ for each node k;
  Symbolic execution of procedure G_LU to determine the arc adjacency list $di(k)$, and $uo(k)$, for each node k of the augmented graph G' ;
  Initialize: $\forall (s,t) \in G'$, $x_{st} := c_{st}$; $succ_{st} := t$
  Initialize: $\forall (s,t) \notin G'$, $x_{st} := M$; $succ_{st} := 0$
    $opt_{ij} := 0 \ \forall \ i = 1, \ldots, n, \ j = 1, \ldots, n$
    $subrow_i := 0$; $subcol_i := 0 \ \forall \ i = 1, \ldots, n$
end

Procedure G_LU
begin
  for $k = 1$ to $n - 2$ ***do***
    for each arc $(s,k) \in di(k)$ ***do***
      for each arc $(k,t) \in uo(k)$ ***do***
        if $x_{st} > x_{sk} + x_{kt}$
          if $s = t$ and $x_{sk} + x_{kt} < 0$ ***then***
            Found a negative cycle; ***STOP***
          if $x_{st} = \infty$ ***then***
            if $s > t$ ***then***
              add a new arc (s,t) to $di(t)$
            if $s < t$ ***then***
              add a new arc (s,t) to $uo(s)$
          $x_{st} := x_{sk} + x_{kt}$; $succ_{st} := succ_{sk}$
end

Procedure Get_D$(\mathbf{s_i}, \mathbf{t_i})$
begin
  if $opt_{s_i t_i} = 0$ ***then***
    if $subcol_{t_i} = 0$ ***then*** $Get\_D\_L(t_i)$; $subcol_{t_i} := 1$
    if $subrow_{s_i} = 0$ ***then*** $Get\_D\_U(s_i)$; $subrow_{s_i} := 1$
    $Min\_add(s_i, t_i)$;
end

Subprocedure Get_D_L$(\mathbf{t})$
begin
  put node t in $LIST$
  while $LIST$ is not empty ***do***
    remove the lowest node k in $LIST$
    for each arc $(s, k) \in di(k)$ ***do***
      if $s \notin LIST$, put s into $LIST$
      if $x_{st} > x_{sk} + x_{kt}$ ***then***
        $x_{st} := x_{sk} + x_{kt}$; $succ_{st} := succ_{sk}$
end

Subprocedure Get_D_U$(\mathbf{s})$
begin
  put node s in $LIST$
  while $LIST$ is not empty ***do***
    remove the lowest node k in $LIST$
    for each arc $(k, t) \in uo(k)$ ***do***
      if $t \notin LIST$, put t into $LIST$
      if $x_{st} > x_{sk} + x_{kt}$ ***then***
        $x_{st} := x_{sk} + x_{kt}$; $succ_{st} := succ_{sk}$
end

Subprocedure Min_add$(\mathbf{s_i}, \mathbf{t_i})$
begin
  $r_i := \max\{s_i, t_i\}$
  for $k = n$ down to $r_i + 1$ ***do***
    if $x_{s_i t_i} > x_{s_i k} + x_{k t_i}$ ***then***
      $x_{s_i t_i} := x_{s_i k} + x_{k t_i}$; $succ_{s_i t_i} := succ_{s_i k_i}$
  $opt_{s_i t_i} := 1$
end

Procedure Get_P($\mathbf{s_i}, \mathbf{t_i}$)
begin
let $k := succ_{s_i t_i}$
while $k \neq t_i$ ***do***
Get_D(k, t_i);
let $k := succ_{kt_i}$
end

labels for their tails, and mark these tails as labeled. That is, $x_{st_i} = \min\{x_{st_i}, x_{sk} + x_{kt_i}\}$, and $label(s) = 1$ for each arc (s, k) in $di(k)$. After node k is scanned, we then search for the next lowest labeled node to scan and repeat these procedures until finally all the labeled nodes higher than t_i are scanned. Then we reset $label(s) = 0$ for each node s.

In *Get_D_U*(s_i), starting from node $k = s_i$, we scan arcs in $uo(k)$, update the distance labels for their heads, and mark these heads as labeled. That is, $x_{s_it} = \min\{x_{s_it}, x_{s_ik} + x_{kt}\}$, and $label(t) = 1$ for each arc (k, t) in $uo(k)$. After node k is scanned, we then search for the next lowest labeled node to scan and repeat these procedures until finally all the labeled nodes higher than s_i are scanned. Then we reset $label(t) = 0$ for each node t.

Let $\widetilde{s}_{t_i}$ and $\widetilde{t}_{s_i}$ denote the highest labeled node obtained by *Get_D_L*(t_i) and *Get_D_U*(s_i), respectively. Let $N^L_{t_i}$ and $N^U_{s_i}$ be the set of labeled nodes by *Get_D_L*(t_i) and *Get_D_U*(s_i), respectively. The complexity of the bucket implementation is $O(\sum_{k=t_i}^{\widetilde{s}_{t_i}} (1) + \sum_{k \in N^L_{t_i}} |di(k)|)$ for *Get_D_L*(t_i) and $O(\sum_{k=s_i}^{\widetilde{t}_{s_i}} (1) + \sum_{k \in N^U_{s_i}} |uo(k)|)$ for *Get_D_U*(s_i). Note that we use the indicator arrays $subcol_{t_i}$ and $subrow_{s_i}$ for each distinct s_i and t_i to avoid redundant operations.

5.4.2 Heap implementation of *Get_D_L*(t_i) and *Get_D_U*(s_i)

This is similar to the bucket implementation of $G\_Forward(t_i)$ in Section 5.3.3.2.

In *Get_D_L*(t_i), we maintain a min-heap where any of its tree node is lower than its children. Starting from the destination node $k = t_i$, we scan arcs in $di(k)$, update the distance labels for their tails, mark these tails as labeled, and put them into the min-heap. Then we extract the top node from the min-heap and repeat these procedures until the min-heap becomes empty.

In *Get_D_U*(s_i), we use the same min-heap that has become empty after *Get_D_L*(t_i).

Starting from the origin node $k = s_i$, we scan arcs in $uo(k)$, update the distance labels for their heads, mark these heads as labeled, and put them into the min-heap. Then we extract the top node from the min-heap and repeat these procedures until the min-heap becomes empty.

Let $N_{t_i}^L$ and $N_{s_i}^U$ be the set of nodes labeled by $Get\_D\_L(t_i)$ and $Get\_D\_U(s_i)$, respectively. The complexity of the bucket implementation is $O(\left|N_{t_i}^L\right| \log \left|N_{t_i}^L\right| + \sum_{k \in N_{t_i}^L} |di(k)|)$ for $Get\_D\_L(t_i)$ where $\sum_{k \in N_{t_i}^L} |di(k)|$ is the total number of scans and $\left|N_{t_i}^L\right| \log \left|N_{t_i}^L\right|$ comes from inserting a node into and extracting a node from a binary heap. Similarly, $Get\_D\_U(s_i)$ has an $O(\left|N_{s_i}^U\right| \log \left|N_{s_i}^U\right| + \sum_{k \in N_{s_i}^U} |uo(k)|)$ time bound.

5.5 *Settings of computational experiments*

In this Section, we present the test conditions for our computational experiments on solving MPSP problems.

5.5.1 Artificial networks and real flight network

We use four network generators which produce artificial networks: SPGRID, SPRAND and SPACYC are designed by Cherkassky et al. [74]; NETGEN is by Klingman et al. [207, 185]. We also test our algorithms on a real Asia-Pacific flight network.

5.5.1.1 Asia-Pacific flight network (AP-NET):

As shown in Chapter 1, the AP-NET contains 112 nodes (48 center nodes in Table 2, and 64 rim nodes in Table 3) and 1038 arcs, where each node represents a chosen city and each arc represents a chosen flight. Among the 1038 arcs, 480 arcs connect center cities to center cities; 277 arcs connect rim cities to center cities; and 281 arcs connect center cities to rim cities. We use the great circle distance between the departure and arrival cities of each flight leg as the arc length.

5.5.1.2 SPGRID:

SPGRID generates a grid network defined by $XY + 1$ nodes. In particular, consider a plane with XY integer coordinates $[x, y]$, $1 \leq x \leq X$, $1 \leq y \leq Y$. An arc with tail $[x, y]$ is called "forward" if its head is $[x+1, y]$, "upward" if its head is $[x, (y+1) \bmod Y]$, and "downward" if its head is $[x, (y-1) \bmod Y]$ for $1 \leq x \leq X$, $1 \leq y \leq Y$. A "super source" node is connected to nodes $[1, y]$ for $1 \leq y \leq Y$. Let each "layer" be the subgraph induced by the nodes of the same x. For each layer, SPGRID can specify each layer to be either doubly connected (i.e. each layer is a doubly connected cycle) or singly connected. Among all of the adjustable parameters, we choose to adjust X, Y, layer connectivity, and the range of arc lengths for arcs inside each layer ($[c_l, c_u]$) and arcs between different layers ($[\hat{c}_l, \hat{c}_u]$). The arc lengths will be uniformly chosen in the range.

Following the settings used in [74], we generate the following four SPGRID families:

- SPGRID-SQ: (SQ stands for square grid)

  $X = Y \in \{10, 20, 30, 40, 50, 60, 70, 80, 90, 100\}$; each layer is double cycle;

  $[c_l, c_u] = [10^3, 10^4]$; $[\hat{c}_l, \hat{c}_u] = [10^3, 10^4]$

- SPGRID-WL: (WL stands for wide or long grid)

  Wide: $X = 16, Y \in \{64, 128, 256, 512\}$; Long: $Y = 16, X \in \{64, 128, 256, 512\}$; each layer is double cycle; $[c_l, c_u] = [10^3, 10^4]$; $[\hat{c}_l, \hat{c}_u] = [10^3, 10^4]$

- SPGRID-PH: (PH stands for positive arc lengths and hard problems)

  $Y = 32, X \in \{16, 32, 64, 128, 256\}$; each layer is single cycle;

  $[c_l, c_u] = [1, 1]$; $[\hat{c}_l, \hat{c}_u] = [10^3, 10^4]$; randomly assign 64 additional arcs within each layer

- SPGRID-NH: (NH stands for negative arc lengths and hard problems)

  $Y = 32, X \in \{16, 32, 64, 128\}$; each layer is single cycle;

  $[c_l, c_u] = [1, 1]$; $[\hat{c}_l, \hat{c}_u] = [-10^4, -10^3]$; randomly assign 64 additional arcs within each layer

5.5.1.3 SPRAND:

SPRAND first constructs a hamiltonian cycle, and then adds arcs with distinct random end points. Except for the SPRAND-LENS and SPRAND-LEND family, we set the length of the arcs in the hamiltonian cycle to be 1 and others to be uniformly chosen from the interval $[0, 10^4]$.

By adjusting the parameters, we generate the following six SPRAND families:

- SPRAND-S: (S stands for sparse graphs)
  $|N| \in \{128, 256, 512, 1024, 2048\}$; average degree $\in \{4, 16\}$

- SPRAND-D: (D stands for dense graphs)
  $|N| \in \{128, 256, 512, 1024, 2048\}$; $|A| = \frac{|N|(|N|-1)}{k}$ where $k \in \{4, 2\}$

- SPRAND-LENS: (LEN stands for different arc lengths; S stands for sparse graphs)
  $|N| \in \{256, 1024\}$; average degree $= 4$; range of arc lengths $[L, U] \in \{[1, 1], [0, 10], [0, 10^2], [0, 10^4], [0, 10^6]\}$

- SPRAND-LEND: (LEN stands for different arc lengths; D stands for dense graphs)
  $|N| \in \{256, 1024\}$; $|A| = \frac{|N|(|N|-1)}{4}$; range of arc lengths $[L, U] \in \{[1, 1], [0, 10], [0, 10^2], [0, 10^4], [0, 10^6]\}$

- SPRAND-PS: (P stands for changeable node potential; S stands for sparse graphs)
  $|N| \in \{256, 1024\}$; average degree $= 4$; lower bound of node potential $= 0$; upper bound of node potential $\in \{0, 10^4, 10^5, 10^6\}$

- SPRAND-PD: (P stands for changeable node potential; D stands for dense graphs)
  $|N| \in \{256, 1024\}$; $|A| = \frac{|N|(|N|-1)}{4}$; lower bound of node potential $= 0$; upper bound of node potential $\in \{0, 10^4, 10^5, 10^6\}$

5.5.1.4 NETGEN:

NETGEN is a network generator developed by Klingman et al. [207]. We use its C version to create our testing categories. Among many adjustable parameters, we choose to adjust $|N|$, $|A|$ and $[L, U]$ where L and U represent the lower and upper bounds on arc lengths.

By adjusting the parameters, we generate the following four NETGEN families:

- NETGEN-S: (S stands for sparse graphs)
  $|N| \in \{128, 256, 512, 1024, 2048\}$; average degree $\in \{4, 16\}$; $[L, U] = [0, 10^3]$

- NETGEN-D: (D stands for dense graphs)

  $|N| \in \{128, 256, 512, 1024, 2048\}$; $|A| = \frac{|N|(|N|-1)}{k}$ where $k \in \{4, 2\}$; $[L, U] = [0, 10^3]$

- NETGEN-LENS: (LEN stands for different arc lengths; S stands for sparse graphs)

  $|N| \in \{256, 1024\}$; average degree $= 4$; $L = 0$; $U \in \{0,\ 10^2,\ 10^4,\ 10^6\}$

- NETGEN-LEND: (LEN stands for different arc lengths; D stands for dense graphs)

  $|N| \in \{256, 1024\}$; $|A| = \frac{|N|(|N|-1)}{k}$ where $k \in \{4, 2\}$; $L = 0$; $U \in \{0,\ 10^2,\ 10^4,\ 10^6\}$

5.5.1.5 SPACYC:

SPACYC generates acyclic networks. It first constructs a central path starting from node 1 that visits every other node exactly once, and then randomly connects nodes. All arcs are oriented from nodes of smaller index to nodes of larger index. Among many adjustable parameters, we choose to adjust $|N|$, $|A|$ and $[L, U]$ where L and U represent the lower and upper bounds on arc lengths.

By adjusting the parameters, we generate the following five SPACYC families:

- SPACYC-PS: (P stands for positive arc length; S stands for sparse graphs)

  $|N| \in \{128, 256, 512, 1024, 2048\}$; average degree $\in \{4, 16\}$; arcs in the central path have length $= 1$, all other arcs have lengths uniformly distributed in $[L, U] = [0, 10^4]$

- SPACYC-NS: (N stands for negative arc length; S stands for sparse graphs)

  $|N| \in \{128, 256, 512, 1024, 2048\}$; average degree $\in \{4, 16\}$; arcs in the central path have length $= -1$, all other arcs have lengths uniformly distributed in $[L, U] = [-10^4, 0]$

- SPACYC-PD: (P stands for positive arc length; D stands for dense graphs)

  $|N| \in \{128, 256, 512, 1024, 2048\}$; $|A| = \frac{|N|(|N|-1)}{k}$ where $k \in \{4, 2\}$; arcs in the central path have length $= 1$, all other arcs have lengths uniformly distributed in $[L, U] = [0, 10^4]$

- SPACYC-NS: (N stands for negative arc length; D stands for dense graphs)

  $|N| \in \{128, 256, 512, 1024\}$; $|A| = \frac{|N|(|N|-1)}{k}$ where $k \in \{4, 2\}$; arcs in the central path have length $= -1$, all other arcs have lengths uniformly distributed in $[L, U] = [-10^4, 0]$

- SPACYC-P2N: (P2N stands for changing the arc lengths from positive to negative) $|N| \in \{128, 1024\}$; average degree $= 16$; arcs in the central path have length $= 1$, all other arcs have lengths uniformly distributed in $[L, U] \in 10^3 \times \{[0, 10]$, $[-1, 9]$, $[-2, 8]$, $[-3, 7]$, $[-4, 6]$, $[-5, 5]$, $[-6, 4]$, $[-10, 0]\}$

5.5.2 Shortest path codes

Using different node selection techniques, we have several implementations of our MPSP algorithms SLU and $DLU2$. We also implement a "combinatorial" (or graphical) Floyd-Warshall (FW) algorithm which is much faster than its naive algebraic implementation. All versions of SLU, DLU, and FW share the same preprocessing procedure which determines a sparse node ordering.

For other SSSP algorithms, we modify the label correcting and label setting codes written by Cherkassky et al. [74] for solving the MPSP problems.

5.5.2.1 SLU codes:

All versions $SLU1$, $SLU2$, and $SLU3$ have the same LU factorization subroutines. They differ by techniques of implementing the forward elimination and backward substitution procedures. In particular, $SLU1$ uses a technique similar to Dial's bucket implementation as described in Section 5.3.3.1; $SLU2$ uses single heap as introduced in Section 5.3.3.2; $SLU3$ uses two heaps as introduced in Section 5.3.3.3.

5.5.2.2 DLU2 codes:

Both $DLU21$ and $DLU22$ have the same LU factorization and min-addition subroutines. They differ by techniques of implementing the acyclic operations in $Acyclic\_L$ and $Acyclic\_U$ procedures. In particular, $DLU21$ uses a technique similar to Dial's bucket implementation as described in Section 5.4.1; $DLU22$ uses single heap as introduced in Section 5.4.2.

5.5.2.3 *Floyd-Warshall code:*

The graphical Floyd-Warshall algorithm FWG takes advantage of the sparse node ordering to avoid more trivial triple comparisons than its naive algebraic implementation FWA. In particular, in the k^{th} iteration of FWG, the nontrivial triple comparison $i \to k \to j$ computes $\min\{x_{ij}, x_{ik}+x_{kj}\}$ for $x_{ik} \neq \infty$, and $x_{kj} \neq \infty$. These nontrivial triple comparisons can be efficiently implemented if we maintain a $n \times 1$ outgoing arc list (k, l) and a $n \times 1$ incoming arc list (l, k) for $l = 1, \ldots, n$ for each node k. In the procedure, whenever $x_{ij} = \infty$ and $x_{ik}+x_{kj} < \infty$, we add a new arc (i, j) to the outgoing arc list of node i and incoming arc list of node j. Thus this implementation avoids trivial triple comparisons where $x_{ik} = \infty$ or $x_{kj} = \infty$ and is much faster.

5.5.2.4 *Label correcting SSSP codes:* $GOR1$, BFP, $THRESH$, $PAPE$, $TWOQ$

These five label correcting SSSP codes are chosen from Cherkassky et al. [74] due to their better performance than other implementations of label correcting algorithms. They differ with each other in the order of node selection for scanning.

In particular, $GOR1$, proposed by Goldberg and Radzik [140], does a topological ordering on the candidate nodes so that the node with more descendants will be scanned earlier. BFP is a modified Bellman-Ford algorithm [40] which scans a node only if its parent is not in the queue. $THRESH$ is the threshold algorithm by Glover et al. [134]. $PAPE$ is a dequeue implementation by Pape [264] and Levit [220] which maintains two candidate lists: one is a stack and the other is a queue. It always selects a node to scan from the stack, if it is not empty; otherwise, it scans a node from the queue. When it scans a node, that is, checks the heads of the outgoing arcs from that node, if the head has not been scanned yet, it will be put into the queue; otherwise, it will be put into the stack. $PAPE$ has been shown to have an exponential time bound [203, 290], but is practically efficient in most real-world networks. $TWOQ$ by Pallottino [260] is a similar algorithm which maintains two queues instead of one stack and one queue as $PAPE$.

When solving SSSP problems, $GOR1$ and BFP have complexity $O(nm)$, $THRESH$ has complexity $O(nm)$ for problems with nonnegative arc lengths, and $TWOQ$ has complexity

$O(n^2m)$.

5.5.2.5 *Label setting SSSP codes: $DIKH$, $DIKBD$, $DIKR$, $DIKBA$*

These four label setting SSSP codes are chosen from Cherkassky et al. [74] due to their better performance than other implementations of label setting algorithms. All of them are variants of Dijkstra's algorithm. They differ with each other in the way of selecting the node with smallest distance label.

In particular, $DIKH$ is the most common binary heap implementation. $DIKBD$, $DIKR$, and $DIKBA$ can be viewed as variants of Dial's bucket implementation [93]. $DIKBD$ is a double bucket implementation, $DIKBA$ is an approximate bucket implementation, and $DIKR$ is the radix-heap implementation. See Cherkassky et al. [74] for more detailed introduction on these implementations and their complexities.

5.5.2.6 *Acyclic code: (ACC)*

ACC is also written by Cherkassky et al. [74] to test the performance of different algorithms on acyclic graphs. It is based on topological ordering which has complexity $O(m)$ for solving SSSP problems.

5.5.3 Requested OD pairs

The indices of the requested OD pairs may affect the efficiency of our algorithms $DLU2$ and SLU, but will not affect the SSSP algorithms. In particular, algorithm $DLU2$ and SLU compute the shortest path lengths for node pairs with larger index earlier or faster than node pairs with smaller index since both $DLU2$ and SLU use the LU factorization, which does more triple comparisons for higher node pairs. The SSSP algorithms, on the other hand, find the shortest path tree at each iteration for different roots, and thus does not depend on the indices of the requested OD pairs.

To do a fair computational comparison, we first randomly choose several destination nodes (columns in the OD matrix). For each destination node, we randomly choose its associated origin node (row). We produce four sets: OD_{25}, OD_{50}, OD_{75}, and OD_{100} of OD pairs for each test family, in which OD_k means that the requested OD pairs cover $k\% * |N|$

Table 8: number of fill-ins created by different pivot rules

Pivoting rule	NAT	DM	DMT	SM	MNDn	MNDe	MMDm	MMTa	AMD
# fill-ins	1084	153	153	170	193	210	430	341	2172

destination nodes (columns). Using these four random OD sets for each test family, we can observe how our algorithms $DLU2$ and SLU perform compared with the repeated SSSP algorithms.

Intuitively, algorithm $DLU2$ and SLU should be advantageous for the OD_{100} case. The 100% random requested OD pairs will cover all the columns and make the MPSP problem an APSP problem for repeated SSSP algorithms. However, algorithm $DLU2$ and SLU save some computational work since they terminate when all the $|N|$ requested OD pairs are computed, instead of all $\frac{|N|(|N|-1)}{2}$ OD pairs.

5.6 Computational results

This section summarizes our comprehensive computational experiments. First we compare the different preprocessing rules, and then we compare different implementations of SLU, $DLU2$ and how they perform compared with the Floyd-Warshall algorithm. Finally we compare SLU and $DLU2$ with many SSSP algorithms on many different problem families using different percentages of covered columns in the set of requested OD pairs.

5.6.1 Best sparsity pivoting rule

Using the Asia-Pacific flight network (AP-NET) as an example, Table 8 gives the number of fill-ins induced by different pivot rules (see Section 5.3.1). None of these 9 pivoting rules is always superior to the others. However in our experience, usually the dynamic Markowitz rule (with or without tie breaking) produces fewer fill-ins than others. Thus in all of our experiments, we compare the number of fill-ins induced by NAT, DM, and DMT, then choose the smallest of these three rules as our pivoting rule.

5.6.2 Best SLU implementations

$SLU1$, $SLU2$, and $SLU3$ all use the same node ordering produced by the preprocessing procedure. In our tests (see Tables 35,...,91 in the appendix), $SLU1$ is always faster than $SLU2$ and $SLU3$. $SLU2$ performs a little better than $SLU3$, but in general they perform very similarly. We also have an interesting finding which indicates that the heap-oriented codes will perform better on Intel machines than Sun machines. In particular, the relative performance of the heap-oriented codes such as $SLU2$, and $SLU3$ will improve on Intel machines using Linux (Mandrake 8.2) or Windows (cygwin on Win2000). Nevertheless, $SLU1$ is still more efficient overall.

5.6.3 Best $DLU2$ implementations

Both $DLU21$ and $DLU22$ use the same node ordering produced by the preprocessing procedure. In our tests (see Tables 35,...,91 in the appendix), $DLU21$ is usually faster than $DLU22$. As discussed previously, the relative performance of the heap-oriented codes such as $DLU22$ will improve on Intel machines. Nevertheless, $DLU21$ is still more efficient asymptotically.

5.6.4 SLU and $DLU2$ vs. Floyd-Warshall algorithm

Algorithms SLU and $DLU2$ always perform much faster than the Floyd-Warshall algorithm. Here we use the AP-NET as an example. Suppose there are 112 requested OD pairs whose destination nodes span the whole node set (i.e. 112 nodes). Table 9 shows the relative running times of the naive algebraic implementation of Floyd-Warshall algorithm (FWA), the graphical implementation (FWG), and the SLU and $DLU2$ implementations on different platforms.[1] In this specific example, $DLU21$ is the fastest code. Although FWG significantly improves upon FWA, it is still worse than $SLU1$ and $DLU21$. We also observe that when the network becomes larger, the Floyd-Warshall algorithm becomes more inefficient. The reason may be due to more memory accessing operations since it is an algebraic algorithm and requires $O(n^2)$ storage.

[1] Sun is a Sun workstation; Mdk is Mandrake Linux on an Intel machine; Win is Win2000 on an Intel machine.

Table 9: Floyd-Warshall algorithms vs. SLU and $DLU2$

	FWA	FWG	SLU1	SLU2	SLU3	DLU21	DLU22
Sun	18.77	8.21	2.10	5.49	5.38	1.00	1.37
Mdk	12.25	5.62	3.87	6.25	5.06	1.00	1.29
Win	13.37	6.17	2.63	5.77	5.63	1.00	1.29

5.6.5 SLU and $DLU2$ vs. SSSP algorithms

In our computational experiments, we produce four sets: OD_{25}, OD_{50}, OD_{75}, and OD_{100} of OD pairs for each problem family. For the same problem family, the SSSP algorithms have consistent performance for different numbers of OD pairs. This is because the SSSP algorithms solve ALL-1 shortest path trees for each destination; thus increasing the number of distinct destinations simply increases the number of applications of these algorithms. On the other hand, our MPSP algorithms SLU and $DLU2$ will have better relative performance when number of distinct destinations increases. In particular, for the same problem set, SLU and $DLU2$ will perform relatively better on cases of 100% $|N|$ distinct destinations than on cases of 25% $|N|$ distinct destinations. The computationally burdensome LU factorization procedure of SLU and $DLU2$ only needs to be done once, after which the overall effort for solving APSP problems is less. Thus, for problems requesting OD pairs with more distinct destinations, the overhead in the LU factorization will not be wasted.

5.6.5.1 Flight networks

Table 10 lists the computational results of 15 algorithms on the AP-NET.

We observe that all versions of SLU and $DLU2$ perform better for cases with more distinct destinations. In this specific example, $DLU21$ is one of the fastest codes, especially for cases where more than 75% distinct requested destinations. Most label correcting methods like $PAPE$, $TWOQ$, $THRESH$, and BFP also perform well. $SLU1$ is not as fast as $DLU2$, but is still faster than all the Dijkstra codes. FWG, our modified Floyd-Warshall algorithm, is the slowest one even when all the SSSP codes are solving an APSP problem in the case with 100% distinct requested destinations.

It is also interesting that those codes which require more memory access (such as FWG,

Table 10: Relative performance of different algorithms on AP-NET

		FWG	SLU 1	SLU 2	SLU 3	DLU 21	DLU 22	GOR 1	BFP	THRESH	PAPE	TWOQ	DIKH	DIKBD	DIKR	DIKBA
25%	Sun	57.09	4.30	10.22	10.52	2.22	4.35	3.78	1.09	2.04	1.09	1.00	6.00	4.35	8.78	10.57
	Mdk	21.16	4.26	5.95	5.00	1.42	1.79	2.84	1.05	1.63	1.00	1.05	3.53	3.32	5.32	4.05
	Win	19.12	2.30	4.75	4.46	1.20	1.35	2.90	1.00	1.55	1.00	1.05	3.30	3.40	5.71	6.80
50%	Sun	27.52	3.71	9.46	9.12	2.33	3.06	3.40	1.08	1.96	1.02	1.00	5.67	4.27	8.35	10.31
	Mdk	10.92	3.81	5.92	4.81	1.03	1.70	2.70	1.11	1.65	1.00	1.05	3.49	3.27	5.35	4.51
	Win	9.54	2.10	5.00	4.38	1.12	1.30	2.95	1.02	1.62	1.05	1.00	3.56	3.50	5.83	6.58
75%	Sun	18.21	3.44	8.84	8.86	1.77	2.37	3.33	1.07	1.97	1.05	1.00	5.67	4.21	8.36	9.52
	Mdk	9.02	4.36	7.13	5.71	1.00	1.71	3.47	1.36	2.09	1.24	1.33	4.40	4.00	6.69	5.69
	Win	7.50	2.47	5.24	5.03	1.00	1.51	3.44	1.22	1.88	1.22	1.16	4.08	4.02	6.87	7.07
100%	Sun	13.63	3.49	9.12	8.93	1.66	2.27	3.55	1.08	1.99	1.05	1.00	5.66	4.43	8.52	10.29
	Mdk	5.62	3.87	6.25	5.06	1.00	1.29	3.01	1.17	1.78	1.09	1.14	3.78	3.48	5.75	4.80
	Win	6.17	2.63	5.77	5.63	1.00	1.29	3.72	1.35	2.23	1.39	1.27	4.21	4.55	7.26	5.58

Table 11: 75% SPGRID-SQ

Grid/deg	SLU 1	DLU 21	GOR 1	BFP	THRESH	PAPE	TWOQ	DIKH	DIKBD	DIKR	DIKBA
10x10/3	5.00	6.20	5.50	1.30	3.30	1.10	1.00	7.30	6.10	10.90	26.10
20x20/3	5.54	3.73	5.04	1.18	2.58	1.08	1.00	8.01	4.40	9.43	11.84
30x30/3	4.97	3.79	4.27	1.13	2.01	1.05	1.00	6.82	3.27	7.15	6.52
40x40/3	5.85	10.74	4.56	1.12	2.15	1.05	1.00	7.18	3.24	7.14	5.22
50x50/3	5.55	5.05	5.11	1.13	2.04	1.04	1.00	7.27	3.09	6.81	4.56
60x60/3	6.00	5.07	5.24	1.13	1.95	1.03	1.00	6.92	2.91	6.37	3.88
70x70/3	6.86	5.84	4.67	1.14	2.13	1.05	1.00	7.35	3.04	6.62	3.73
80x80/3	8.64	9.91	5.65	1.14	2.14	1.05	1.00	7.54	3.05	6.55	3.54
90x90/3	9.62	13.71	5.16	1.22	2.02	1.03	1.00	6.60	2.69	5.72	2.86
100x100/3	12.62	9.09	4.82	1.14	2.26	1.04	1.00	7.83	3.04	6.49	3.22

$SLU2$, $SLU3$, $DLU22$, $DIKH$, $DIKR$, and $DIKBA$) perform better on the Intel platform (Mdk and Win) than on Sun platform.

5.6.5.2 Random network generators

In this section, we choose the cases whose requested OD pairs have 75% $|N|$ distinct destinations for our discussion. Results for other cases using 25% $|N|$, 50% $|N|$, and 100% $|N|$ distinct destinations are listed in the appendix. All of these experiments are run on the Sun machine. The performances of $SLU2$ and $SLU3$ are always worse than $SLU1$. Similarly, $DLU22$ is worse than $DLU21$. Thus we only include $SLU1$ and $DLU21$ here for comparison.

SPGRID-SQ family: Table 11 shows that label-correcting codes $TWOQ$, $PAPE$ and BFP perform the best in this SPGRID-SQ family. Dijkstra-based codes such as $DIKBD$,

Table 12: 75% SPGRID-WL

Grid/deg	SLU 1	DLU 21	GOR 1	BFP	THR ESH	PA PE	TWO Q	DIK H	DIK BD	DIK R	DIK BA
16x64/3	5.66	3.50	4.38	1.09	2.05	1.05	1.00	6.22	3.88	8.42	9.69
16x128/3	5.70	3.59	5.15	1.12	1.99	1.05	1.00	5.82	3.61	8.20	8.12
16x256/3	5.65	4.46	4.90	1.10	2.03	1.05	1.00	5.76	3.70	8.49	7.83
16x512/3	7.11	3.18	5.60	1.09	2.04	1.03	1.00	5.41	3.53	8.33	7.24
64x16/3	3.93	3.69	4.81	1.13	2.22	1.06	1.00	8.39	3.25	6.78	5.60
128x16/3	3.38	3.76	4.97	1.15	2.02	1.03	1.00	8.46	2.90	5.78	3.68
256x16/3	3.39	4.85	4.86	1.12	1.93	1.03	1.00	9.61	2.98	5.77	3.27
512x16/3	3.51	5.00	5.06	1.15	1.80	1.04	1.00	10.62	2.97	5.64	3.02

Table 13: 75% SPGRID-PH

Grid/deg	SLU 1	DLU 21	GOR 1	BFP	THR ESH	PA PE	TWO Q	DIK H	DIK BD	DIK R	DIK BA
32x32/6	4.11	11.70	1.28	1.37	1.12	3.94	2.32	1.87	4.10	2.49	1.00
16x32/5	5.47	9.00	1.91	1.14	1.00	2.03	1.48	3.12	6.22	4.51	2.03
64x32/7	6.26	16.39	1.59	3.05	2.24	10.78	5.34	1.96	3.68	2.43	1.00
128x32/8	9.48	19.82	1.98	6.72	3.14	20.74	9.85	1.93	3.48	2.34	1.00
256x32/8	13.95	24.60	2.23	13.26	3.46	25.35	11.65	1.88	3.27	2.25	1.00

$DIKR$, and $DIKBA$ perform relatively worse for smaller networks. $DIKBD$ perform slightly worse than $THRESH$, but is the fastest Dijkstra's code. $SLU1$ and $DLU21$ perform similarly to $GOR1$ but become worse for larger networks.

SPGRID-WL family: Table 12 shows that label-correcting codes $TWOQ$, $PAPE$ and BFP perform the best in this SPGRID-WL family. $THRESH$ is only slightly worse than BFP. $DIKBD$ is the fastest Dijkstra's code, but $DIKBA$ catches up for larger LONG cases. $SLU1$ is faster in the LONG cases, $DLU21$ is faster in the WIDE cases, and they both are slightly better than $GOR1$. $DIKR$ performs the worst for the WIDE cases, but $DIKH$ performs the worst for the LONG cases.

SPGRID-PH family: Table 13 shows that $DIKBA$, $DIKH$, and $GOR1$ perform the best. $DIKR$, $DIKBD$ and $THRESH$ perform slightly worse but are still relatively better than the remaining codes. BFP, $TWOQ$, $SLU1$ and $PAPE$ perform worse when the network become larger. $DLU21$ perform the worst overall.

Table 14: 75% SPGRID-NH

Grid/deg	SLU 1	DLU 21	GOR 1	BFP	THR ESH	PA PE	TWO Q	DIK H	DIK BD	DIK R	DIK BA
16x32/5	7.89	13.07	2.56	1.01	1.21	1.01	1.00	4.39	2.07	4.11	3.54
32x32/6	8.39	23.66	2.06	1.04	1.00	1.07	1.07	3.50	1.45	2.76	1.90
64x32/7	10.92	29.59	1.93	1.18	1.00	1.24	1.23	3.20	1.21	2.18	1.35
128x32/8	15.79	33.39	1.87	1.19	1.00	1.20	1.20	3.04	1.06	1.85	1.13

Table 15: 75% SPRAND-S4 and SPRAND-S16

$\|N\|$/deg	SLU 1	DLU 21	GOR 1	BFP	THR ESH	PA PE	TWO Q	DIK H	DIK BD	DIK R	DIK BA
128/4	9.00	7.00	4.00	1.50	1.50	1.00	1.00	6.50	5.50	10.50	12.00
256/4	12.00	12.60	3.90	1.00	1.30	1.00	1.00	5.90	5.10	8.80	4.10
512/4	16.37	22.27	3.40	1.21	1.00	1.19	1.23	4.45	3.79	6.16	2.73
1024/4	36.87	66.55	3.81	1.77	1.00	1.77	1.81	4.10	4.48	5.23	1.77
2048/4	103.60	144.95	3.11	1.83	1.00	2.06	2.09	2.98	3.96	3.46	1.03
128/16	10.43	14.14	3.00	1.00	1.00	1.14	1.00	2.57	2.14	3.57	4.29
256/16	25.42	26.36	3.06	1.24	1.00	1.27	1.27	2.64	2.33	3.36	2.30
512/16	78.31	59.22	3.21	1.38	1.00	1.59	1.59	2.49	2.24	3.04	1.63
1024/16	159.01	147.26	2.82	1.43	1.00	2.01	1.92	1.98	2.01	2.31	1.19
2048/16	270.46	262.17	2.80	1.68	1.22	2.89	2.73	1.73	1.95	1.90	1.00

SPGRID-NH family: Table 14 shows that all label-correcting codes perform the best, followed by the label-setting codes. $SLU1$ and $DLU21$ perform the worst, especially for larger networks.

SPRAND-S family: Table 15 shows that label-correcting codes perform the best. Label-setting codes are slightly worse than label-correcting codes, and tend to perform relatively better for cases with larger degree. $SLU1$ and $DLU21$ perform the worst, especially for networks with more nodes and larger degrees.

SPRAND-D family: Table 16 shows that label-setting codes perform the best. Label-correcting methods such as $THRESH$, $GOR1$, and BFP perform slightly worse than the label-setting codes. For smaller networks (e.g. $|N| \leq 512$), $SLU1$ and $DLU21$ perform the worst. $PAPE$ and $TWOQ$ perform very well for smaller networks, but become the worst for larger and denser cases.

SPRAND-LENS family: Table 17 shows that label-setting codes perform the best for cases with smaller arc length. For larger arc length, label-correcting codes perform the best.

Table 16: 75% SPRAND-D4 and SPRAND-D2

$\lvert N\rvert$/deg	SLU 1	DLU 21	GOR 1	BFP	THR ESH	PA PE	TWO Q	DIK H	DIK BD	DIK R	DIK BA
128/32	7.64	10.07	2.50	1.21	1.00	1.29	1.29	1.79	1.71	2.21	2.36
128/64	4.38	5.34	2.22	1.09	1.00	1.22	3.28	1.12	1.06	1.28	1.16
256/64	12.61	11.53	2.47	1.28	1.00	1.60	1.56	1.12	1.07	1.30	1.06
256/128	8.34	7.56	2.78	1.49	1.30	2.03	1.97	1.02	1.01	1.12	1.00
512/128	27.97	15.53	3.01	1.80	1.45	3.13	2.97	1.06	1.08	1.13	1.00
512/256	15.70	8.89	3.40	2.36	1.99	4.66	4.33	1.00	1.03	1.03	1.01
1024/256	30.46	22.95	3.81	3.14	2.61	8.41	7.02	1.00	1.10	1.05	1.05
1024/512	17.01	13.19	4.33	4.05	3.33	12.17	9.81	1.00	1.13	1.06	1.12
2048/512	19.60	15.53	4.35	5.29	4.01	19.17	13.31	1.00	1.07	1.01	1.11
2048/1024	10.30	8.10	4.99	7.62	5.46	30.65	19.84	1.00	1.06	1.00	1.15

Table 17: 75% SPRAND-LENS4

$\lvert N\rvert$/deg	$[L,U]$	SLU 1	DLU 21	GOR 1	BFP	THR ESH	PA PE	TWO Q	DIK H	DIK BD	DIK R	DIK BA
256/4	$[1,1]$	4.03	4.34	2.72	5.79	5.10	11.31	9.24	2.41	1.00	2.14	1.86
	$[0,10]$	3.53	3.94	2.03	2.61	2.33	2.75	2.75	1.94	1.00	1.86	1.39
	$[0,10^2]$	5.17	5.39	2.43	1.30	1.00	1.61	1.48	2.87	1.83	3.22	1.96
	$[0,10^4]$	12.80	13.70	3.60	1.00	1.30	1.00	1.00	5.80	5.20	8.70	4.60
	$[0,10^6]$	14.00	14.89	4.44	1.11	1.44	1.00	1.00	6.56	9.89	12.67	7.78
1024/4	$[1,1]$	20.00	35.44	4.67	35.38	22.67	49.16	57.99	2.59	1.00	2.05	3.55
	$[0,10]$	18.74	31.19	3.81	22.92	16.26	33.66	31.79	2.22	1.00	1.91	1.13
	$[0,10^2]$	21.19	36.03	3.52	4.98	3.38	6.61	5.07	2.63	1.65	2.58	1.00
	$[0,10^4]$	36.58	60.04	3.63	1.68	1.00	1.72	1.72	4.04	4.20	5.06	1.61
	$[0,10^6]$	40.03	68.74	3.60	1.14	1.10	1.01	1.00	4.76	4.86	7.42	1.96

Both $SLU1$ and $DLU21$ perform the worst, especially for cases with more nodes and larger ranges of arc length.

SPRAND-LEND family: Table 18 shows that label-setting codes perform the best. $GOR1$ performs slightly worse than label-setting codes. Label-correcting codes except $GOR1$ perform well for cases with larger range of arc length, but become the worst for

Table 18: 75% SPRAND-LEND4

$\lvert N\rvert$/deg	$[L,U]$	SLU 1	DLU 21	GOR 1	BFP	THR ESH	PA PE	TWO Q	DIK H	DIK BD	DIK R	DIK BA
256/64	$[1,1]$	8.43	7.57	3.19	7.57	7.33	36.57	24.05	1.00	1.10	1.05	1.76
	$[0,10]$	8.43	7.62	2.67	4.24	3.71	18.67	11.48	1.00	1.10	1.05	1.29
	$[0,10^2]$	9.16	8.42	2.37	2.05	1.74	6.16	4.47	1.00	1.00	1.05	1.21
	$[0,10^4]$	13.54	12.31	2.54	1.31	1.00	1.62	1.54	1.23	1.15	1.38	1.15
	$[0,10^6]$	12.29	11.36	2.43	1.21	1.14	1.57	1.57	1.14	1.50	1.43	1.00
1024/256	$[1,1]$	20.13	15.87	4.88	39.11	37.08	344.86	235.18	1.00	1.28	1.01	17.69
	$[0,10]$	21.49	16.24	4.49	20.73	17.26	226.33	122.45	1.00	1.31	1.01	4.36
	$[0,10^2]$	22.29	17.68	4.02	9.16	7.43	86.06	43.06	1.00	1.24	1.02	1.52
	$[0,10^4]$	30.57	23.29	3.84	3.29	2.66	8.18	6.87	1.00	1.12	1.05	1.07
	$[0,10^6]$	34.98	27.26	3.82	1.98	2.11	2.91	2.86	1.07	1.22	1.16	1.00

Table 19: 75% SPRAND-PS4

$\lvert N\rvert$/deg	P	SLU 1	DLU 21	GOR 1	BFP	THR ESH	PA PE	TWO Q	DIK H	DIK BD	DIK R	DIK BA
256/4	0	11.70	12.60	3.70	1.00	1.30	1.00	1.00	5.70	5.20	8.70	4.60
	10^4	13.22	14.11	4.00	1.11	1.44	1.11	1.00	7.00	5.11	9.33	5.56
	10^5	12.80	13.50	3.90	1.10	2.20	1.00	1.00	13.00	5.00	8.80	7.20
	10^6	12.60	13.70	3.80	1.00	3.20	1.00	1.00	21.70	8.00	10.30	7.10
1024/4	0	37.17	65.28	3.64	1.65	1.00	1.64	1.65	4.00	4.18	5.04	1.71
	10^4	36.95	60.61	3.62	1.67	1.00	1.69	1.70	4.07	3.88	4.76	1.66
	10^5	28.90	47.06	2.69	1.24	1.00	1.27	1.27	4.63	3.36	3.45	1.56
	10^6	19.78	34.31	2.19	1.00	1.94	1.02	1.02	11.32	3.49	4.40	2.83

Table 20: 75% SPRAND-PD4

$\lvert N\rvert$/deg	P	SLU 1	DLU 21	GOR 1	BFP	THR ESH	PA PE	TWO Q	DIK H	DIK BD	DIK R	DIK BA
256/64	0	13.62	11.92	2.54	1.23	1.00	1.54	1.54	1.15	1.15	1.38	1.15
	10^4	10.75	9.81	2.12	1.06	1.00	1.38	1.31	1.50	1.25	1.50	1.31
	10^5	10.29	9.24	1.94	1.00	1.65	1.24	1.18	2.65	1.53	1.88	1.65
	10^6	10.24	9.24	2.00	1.00	1.82	1.24	1.24	3.24	1.88	2.06	1.88
1024/256	0	28.76	22.75	3.75	3.21	2.66	8.31	6.97	1.00	1.10	1.05	1.06
	10^4	17.02	13.59	2.26	1.85	1.62	4.78	4.13	1.07	1.02	1.01	1.00
	10^5	9.26	6.98	1.18	1.00	1.56	2.49	2.11	2.07	1.70	1.68	1.66
	10^6	9.07	6.98	1.18	1.00	2.15	2.52	2.15	2.81	2.06	2.05	2.00

cases with smaller arc length. $SLU1$ and $DLU21$ perform worse for cases with more nodes and larger range of arc length.

SPRAND-PS family: Table 19 shows that the label-correcting codes perform the best, and are "potential-invariant" [74]. When the range of node potential P increases, there will be more arcs with negative lengths but not negative cycles which slow down the label-setting codes. Although $SLU1$ and $DLU21$ perform the worst overall, they perform relatively better when P increases.

SPRAND-PD family: Table 20 shows that both label-setting and label-correcting codes perform best for these dense families. $SLU1$ and $DLU21$ perform the worst, although their performances become relatively better when P increases.

NETGEN-S family: Table 21 shows that label-correcting codes are the best, followed by the label-setting codes. $SLU1$ and $DLU21$ perform better than label-setting codes for small networks with smaller degrees; they become significantly worse for larger cases.

Table 21: 75% NETGEN-S4 and NETGEN-S16

$\lvert N \rvert$/deg	SLU 1	DLU 21	GOR 1	BFP	THR ESH	PA PE	TWO Q	DIK H	DIK BD	DIK R	DIK BA
128/4	8.00	6.50	4.50	1.00	2.00	1.00	1.00	6.50	5.50	10.50	13.00
256/4	13.44	14.44	4.11	1.00	1.33	1.11	1.00	6.67	4.44	9.11	6.33
512/4	15.60	20.75	3.68	1.12	1.07	1.00	1.00	4.82	2.93	6.14	3.21
1024/4	44.18	71.50	3.79	1.18	1.04	1.00	1.00	4.50	2.43	5.21	2.00
2048/4	152.32	190.10	3.75	1.21	1.09	1.01	1.00	4.53	2.00	4.74	1.50
128/16	7.62	8.00	2.25	1.00	1.00	1.12	1.12	2.25	2.00	3.00	2.62
256/16	16.53	17.72	3.03	1.36	1.00	1.44	1.44	2.44	1.75	2.86	1.97
512/16	42.73	39.44	3.30	1.53	1.00	1.67	1.67	2.44	1.56	2.65	1.50
1024/16	141.41	136.49	3.36	1.59	1.00	1.80	1.82	2.32	1.30	2.33	1.28
2048/16	318.70	305.86	3.52	1.64	1.00	1.90	1.89	2.39	1.23	2.22	1.16

Table 22: 75% NETGEN-D4 and NETGEN-D2

$\lvert N \rvert$/deg	SLU 1	DLU 21	GOR 1	BFP	THR ESH	PA PE	TWO Q	DIK H	DIK BD	DIK R	DIK BA
128/32	5.24	6.06	2.35	1.12	1.00	1.24	1.24	1.47	1.24	1.82	1.65
256/64	11.84	10.71	2.75	2.31	1.52	2.07	1.99	1.27	1.00	1.32	1.13
512/128	30.63	18.02	3.55	1.93	2.12	2.97	3.88	1.20	1.00	1.18	1.06
1024/256	33.91	26.59	4.04	2.39	2.66	4.01	3.89	1.09	1.00	1.09	1.03
2048/512	22.19	18.20	4.31	2.64	3.08	3.97	3.86	1.03	1.00	1.03	1.01
128/64	3.55	4.27	2.24	1.18	1.21	1.48	1.52	1.06	1.00	1.24	1.12
256/128	9.33	8.08	3.38	1.92	2.14	2.91	4.81	1.15	1.00	1.20	1.13
512/256	17.46	10.30	3.87	2.21	2.55	3.71	3.57	1.11	1.00	1.09	1.02
1024/512	16.59	12.62	4.25	2.65	3.09	4.85	4.65	1.00	1.01	1.05	1.03

NETGEN-D family: Table 22 shows that label-setting codes perform the best, followed by the label-correcting codes. $SLU1$ and $DLU21$ perform the worst, but are relatively better for denser cases.

NETGEN-LENS family: Table 23 shows that label-correcting codes perform the best, followed by the label-setting codes. $SLU1$ and $DLU21$ perform the worst, especially for larger networks. Label-setting codes perform worse when the range of arc lengths becomes

Table 23: 75% NETGEN-LENS4

$\lvert N \rvert$/deg	$[L, U]$	SLU 1	DLU 21	GOR 1	BFP	THR ESH	PA PE	TWO Q	DIK H	DIK BD	DIK R	DIK BA
256/4	$[0, 10]$	13.67	13.44	4.44	1.11	1.44	1.00	1.00	6.22	1.89	3.56	1.67
	$[0, 10^2]$	15.25	16.62	5.12	1.25	1.62	1.00	1.12	7.25	2.75	6.38	2.25
	$[0, 10^4]$	14.44	15.11	4.33	1.11	1.44	1.00	1.00	6.33	4.33	9.33	7.44
	$[0, 10^6]$	13.22	13.56	4.22	1.11	1.33	1.00	1.00	6.67	10.44	12.67	9.00
1024/4	$[0, 10]$	47.88	72.66	3.60	1.21	1.02	1.00	1.00	4.24	1.23	2.11	1.16
	$[0, 10^2]$	43.86	71.94	3.64	1.18	1.05	1.01	1.00	4.40	1.48	2.99	1.28
	$[0, 10^4]$	46.57	75.32	3.65	1.18	1.06	1.01	1.00	4.51	2.40	5.13	1.96
	$[0, 10^6]$	61.76	94.82	3.64	1.19	1.03	1.00	1.01	4.31	4.27	6.70	2.10

Table 24: 75% NETGEN-LEND4

$\|N\|$/deg	$[L,U]$	SLU 1	DLU 21	GOR 1	BFP	THR ESH	PA PE	TWO Q	DIK H	DIK BD	DIK R	DIK BA
256/64	$[0,10]$	14.40	13.50	2.50	1.20	1.30	1.20	1.20	1.30	1.10	1.20	1.00
	$[0,10^2]$	13.18	12.18	3.09	1.73	1.73	2.18	2.27	1.36	1.00	1.18	1.00
	$[0,10^4]$	11.46	10.46	2.69	1.38	1.46	2.00	1.92	1.23	1.00	1.23	1.08
	$[0,10^6]$	9.53	8.87	2.40	1.27	1.27	1.67	1.60	1.00	1.60	1.27	1.00
1024/256	$[0,10]$	35.28	27.80	2.22	1.03	1.12	1.04	1.04	1.00	1.00	1.03	1.00
	$[0,10^2]$	33.84	27.00	3.77	2.40	2.78	2.90	2.90	1.06	1.01	1.04	1.00
	$[0,10^4]$	34.23	26.56	4.11	2.41	2.72	4.60	3.95	1.07	1.00	1.08	1.03
	$[0,10^6]$	33.04	25.63	3.88	2.23	2.51	3.77	3.65	1.04	1.10	1.13	1.00

Table 25: 75% SPACYC-PS4 and SPACYC-PS16

$\|N\|$/deg	$[L,U]$	ACC	SLU 1	DLU 21	GOR 1	BFP	THR ESH	PA PE	TWO Q	DIK H	DIK BD	DIK R	DIK BA
128/4	$[0,10^4]$	1.50	1.00	2.50	1.00	2.00	2.50	1.50	2.00	4.00	1.50	4.00	7.50
256/4	$[0,10^4]$	1.00	1.22	2.22	1.11	1.89	2.00	1.56	1.78	3.67	1.67	3.33	3.89
512/4	$[0,10^4]$	1.00	1.38	2.90	1.36	2.74	2.17	2.21	2.50	3.50	1.29	2.74	2.26
1024/4	$[0,10^4]$	1.00	1.47	4.18	1.41	3.06	2.18	2.47	2.82	3.82	1.24	2.59	1.71
2048/4	$[0,10^4]$	1.00	1.31	4.29	1.30	3.38	2.27	2.78	2.95	3.61	1.06	2.27	1.18
128/16	$[0,10^4]$	1.67	1.00	3.67	2.00	2.67	2.33	2.33	2.67	3.00	2.33	3.33	7.00
256/16	$[0,10^4]$	1.11	1.61	3.00	1.00	2.17	1.72	2.06	2.17	2.28	1.28	2.00	2.50
512/16	$[0,10^4]$	1.00	1.50	3.85	1.15	2.69	1.97	2.87	2.86	2.28	1.17	1.81	1.63
1024/16	$[0,10^4]$	1.00	1.51	5.94	1.14	3.20	2.17	3.43	3.54	2.40	1.03	1.71	1.37
2048/16	$[0,10^4]$	1.13	1.45	6.90	1.24	3.77	2.45	4.07	4.19	2.37	1.00	1.57	1.11

larger. It it not clear how the range of arc lengths affects $SLU1$ and $DLU21$.

NETGEN-LEND family: Table 24 shows that label-setting codes perform the best, followed by the label-correcting codes. $SLU1$ and $DLU21$ perform the worst, especially for larger networks. When the range of arc lengths becomes larger, $SLU1$ and $DLU21$ perform slightly better.

SPACYC-PS family: Table 25 shows that $SLU1$ and $GOR1$ perform the best. $DLU21$ performs relatively worse for cases with more nodes and larger degrees. On the other hand, label-setting codes perform relatively worse for cases with fewer nodes and smaller degrees.

SPACYC-NS family: Table 26 shows that $SLU1$ and $GOR1$ perform the best, followed by the $DLI21$. Other label-correcting algorithms perform much worse. All label-setting codes are even worse than the label-correcting algorithms.

Table 26: 75% SPACYC-NS4 and SPACYC-NS16

$\|N\|$/deg	$[L,U]$	ACC	SLU 1	DLU 21	GOR 1	BFP	THR ESH	PA PE	TWO Q	DIK H	DIK BD	DIK R	DIK BA
128/4	$[-10^4, 0]$	1.00	4.00	5.00	6.00	5.00	9.00	7.00	6.00	80.00	11.00	14.00	12.00
256/4	$[-10^4, 0]$	1.00	1.20	2.60	1.20	2.30	4.50	3.40	3.20	41.50	6.80	8.90	6.90
512/4	$[-10^4, 0]$	1.00	1.19	2.65	1.15	5.98	9.25	7.94	8.42	132.42	12.27	14.88	12.44
1024/4	$[-10^4, 0]$	1.00	1.60	4.80	1.47	16.93	26.13	17.40	22.13	363.00	34.20	40.60	34.67
2048/4	$[-10^4, 0]$	1.00	1.23	4.24	1.24	21.89	36.01	36.21	46.62	848.86	46.51	55.58	47.21
128/16	$[-10^4, 0]$	1.00	1.40	1.80	1.40	3.00	6.40	7.20	6.20	43.80	7.00	8.40	7.60
256/16	$[-10^4, 0]$	1.00	1.61	3.00	1.22	4.78	10.67	16.39	11.78	131.44	10.89	13.00	11.39
512/16	$[-10^4, 0]$	1.00	1.44	3.82	1.22	11.55	31.16	57.77	47.94	521.17	28.78	32.91	29.39
1024/16	$[-10^4, 0]$	1.00	1.42	5.95	1.05	17.84	44.16	72.13	61.92	569.24	41.95	47.55	42.76
2048/16	$[-10^4, 0]$	1.00	1.33	6.72	1.13	30.49	82.06	148.13	135.75	941.90	72.38	82.54	74.28

Table 27: 75% SPACYC-PD4 and SPACYC-PD2

$\|N\|$/deg	$[L,U]$	ACC	SLU 1	DLU 21	GOR 1	BFP	THR ESH	PA PE	TWO Q	DIK H	DIK BD	DIK R	DIK BA
128/32	$[0, 10^4]$	1.00	1.38	2.50	1.12	1.75	1.75	1.62	1.75	1.38	1.25	1.62	2.25
256/64	$[0, 10^4]$	1.20	1.55	3.29	1.12	2.51	2.00	2.78	2.82	1.29	1.00	1.24	1.49
512/128	$[0, 10^4]$	1.51	1.95	6.01	1.51	3.54	2.68	4.69	4.64	1.28	1.00	1.16	1.17
1024/256	$[0, 10^4]$	1.77	3.84	12.65	1.67	4.17	3.14	5.89	5.87	1.15	1.00	1.11	1.06
2048/512	$[0, 10^4]$	1.76	4.04	12.06	1.66	4.40	3.28	6.06	5.90	1.06	1.00	1.03	1.04
128/64	$[0, 10^4]$	1.00	1.36	2.27	1.27	1.91	1.73	2.00	1.91	1.36	1.09	1.36	2.18
256/128	$[0, 10^4]$	1.49	1.58	3.45	1.45	3.03	2.46	3.61	3.58	1.17	1.00	1.15	1.27
512/256	$[0, 10^4]$	1.75	1.81	5.61	1.64	3.61	3.00	4.73	4.64	1.15	1.00	1.09	1.12
1024/512	$[0, 10^4]$	1.88	3.87	11.15	1.73	3.83	3.73	5.56	5.35	1.06	1.00	1.04	1.06
2048/1024	$[0, 10^4]$	1.75	2.80	8.54	1.66	3.99	3.53	5.31	5.13	1.01	1.00	1.02	1.04

SPACYC-PD family: Table 27 shows that all label-setting codes and $GOR1$ perform the best, followed by the $SLU1$ which is slightly worse for larger cases. Label-correcting codes also perform well, less than 5 times slower than ACC. $DLU21$ performs worse when the network become larger. Both $SLU1$ and $DLU21$ perform relatively better for denser cases.

Table 28: 75% SPACYC-ND4 and SPACYC-ND2

$\|N\|$/deg	$[L,U]$	ACC	SLU 1	DLU 21	GOR 1	BFP	THR ESH	PA PE	TWO Q	DIK H	DIK BD	DIK R	DIK BA
128/32	$[-10^4, 0]$	1.00	1.38	2.38	1.12	3.25	8.00	12.50	7.25	62.00	8.12	9.38	8.38
256/64	$[-10^4, 0]$	1.00	1.24	2.71	1.06	5.96	16.13	44.00	19.93	250.69	14.38	15.76	14.82
512/128	$[-10^4, 0]$	1.00	1.26	3.85	1.05	11.63	32.61	179.11	58.14	943.02	27.44	28.58	27.74
1024/256	$[-10^4, 0]$	1.00	2.11	7.31	1.03	21.87	66.24	708.88	183.48	3312.01	55.39	56.49	55.65
128/64	$[-10^4, 0]$	1.00	1.21	2.07	1.14	3.79	10.07	15.07	8.71	66.00	8.36	9.43	8.71
256/128	$[-10^4, 0]$	1.00	1.00	2.25	1.04	6.24	17.86	49.41	20.86	177.84	15.42	16.36	15.80
512/256	$[-10^4, 0]$	1.00	1.01	3.10	1.04	12.66	40.90	267.88	82.04	992.15	33.17	33.93	33.40
1024/512	$[-10^4, 0]$	1.00	2.06	6.26	1.04	23.91	75.98	703.17	233.57	4211.82	64.92	65.05	64.90

Table 29: 75% SPACYC-P2N128 and SPACYC-P2N1024

$\lvert N\rvert$/deg	$[L,U]$ ×1000	ACC	SLU 1	DLU 21	GOR 1	BFP	THR ESH	PA PE	TWO Q	DIK H	DIK BD	DIK R	DIK BA
128/16	$[0,10]$	1.78	2.75	4.59	1.84	1.00	1.34	1.00	3.44	3.22	2.69	4.84	7.66
	$[-1,9]$	1.40	2.12	3.70	1.57	1.02	1.30	1.05	1.00	2.83	2.02	3.42	5.65
	$[-2,8]$	1.00	1.51	2.58	1.14	1.03	1.37	1.08	1.07	3.61	1.97	2.66	4.92
	$[-3,7]$	1.00	1.49	2.38	1.03	1.54	2.27	2.06	1.92	6.83	3.00	3.79	5.70
	$[-4,6]$	1.00	1.47	2.44	1.00	2.67	4.69	5.83	4.47	26.88	5.73	7.00	8.03
	$[-5,5]$	1.00	1.39	2.36	1.11	3.56	7.38	10.68	6.58	64.11	8.35	10.23	10.70
	$[-6,4]$	1.00	1.44	2.44	1.06	3.84	7.73	12.06	7.42	66.14	9.38	11.34	11.64
	$[-10,0]$	1.00	1.41	2.60	1.14	5.73	13.51	21.16	16.16	189.81	15.10	17.95	15.79
1024/16	$[0,10]$	1.76	2.31	8.55	1.66	1.03	1.03	1.00	1.00	2.83	1.90	3.48	2.00
	$[-1,9]$	1.29	1.97	7.35	1.44	1.09	1.00	1.09	1.06	2.85	1.53	2.62	1.71
	$[-2,8]$	1.00	1.39	5.16	1.04	1.22	1.29	1.37	1.39	4.69	1.67	2.37	2.12
	$[-3,7]$	1.00	1.35	4.96	1.04	2.73	3.84	3.82	4.18	24.27	4.90	5.98	5.39
	$[-4,6]$	1.00	1.45	5.55	1.26	7.72	14.04	16.51	14.40	90.43	17.43	20.72	18.51
	$[-5,5]$	1.00	1.44	6.23	1.10	16.69	39.44	58.19	57.38	479.52	45.75	52.42	46.75
	$[-6,4]$	1.00	1.39	5.41	1.10	28.39	72.41	137.86	113.71	959.76	82.71	95.02	85.14
	$[-10,0]$	1.00	1.42	5.48	1.15	39.25	97.81	162.62	88.94	877.62	118.56	135.33	121.50

SPACYC-ND family: Table 28 shows that $GOR1$ and $SLU1$ perform the best, followed by $DLU21$, which is a little worse for larger cases. All other codes (especially $DIKH$ and $PAPE$) perform significantly worse, especially for larger cases.

SPACYC-P2N family: Table 29 shows that $GOR1$ and $SLU1$ perform the best. When $[L,U]$ is larger than $[-3000,7000]$, the other label-correcting and label-setting codes perform well and slightly better than $DLU21$. However, when $[L,U]$ is smaller than $[-3000,7000]$, these SSSP codes (except $GOR1$) perform significantly worse. The more negative the arc lengths are, the worse they become.

5.7 Summary

After these comprehensive computational experiments, we have several observations.

1. Among the eight different pivoting rules for sparsity, the dynamic Markowitz and its variant that employs simple tie-breaking technique usually produce fewer fill-ins than other pivoting rules.

2. Among the three different SLU implementations, the bucket implementation $SLU1$ outperforms the other two implementations that are based on binary heaps, for all

cases. Similarly, the bucket implementation of $DLU2$, $DLU21$, has better performance than the heap implementation.

For general larger networks, $SLU1$ and $DLU21$ can not compete with the best of the state-of-the-art SSSP codes , except for acyclic networks. Nevertheless, $SLU1$ and $DLU21$ do not perform worst for all cases.

For dense graphs, usually $DLU21$ performs better than $SLU1$; for sparse graphs, usually $SLU1$ is better than $DLU21$. Unlike other SSSP algorithms which perform consistently regardless the number of distinct destinations, $SLU1$ and $DLU21$ perform relatively better for MPSP problems of more distinct requested destinations than for problems of fewer distinct requested destinations.

3. The Asia-Pacific flight network is the only real-world network tested in this thesis. Although it is sparse (112 nodes, 1038 arcs), $DLU21$ performs better than $SLU1$, which in turn outperforms all of the implemented label-setting codes. The label-correcting codes perform similarly to $SLU1$ and $DLU21$. The Floyd-Warshall algorithm is in no case competitive with other SSSP algorithms or our proposed MPSP algorithms.

4. In most SPGRID families, the label-correcting codes usually perform the best, except for the SPGRID-PH family for which the label-setting codes perform the best. $SLU1$ and $DLU21$ have better performance on the SPGRID-SQ and SPGRID-WL families than other SPGRID families.

   For each SPGRID family, label-setting codes usually perform relatively worse on smaller networks.

5. $SLU1$ and $DLU21$ are usually slower than other label-correcting and label-setting codes for most SPRAND families. Label-correcting codes perform better for most of the sparse SPRAND families, but label-setting codes perform better for most dense SPRAND families.

   When the range of arc lengths decreases (e.g., ≤ 10), label-correcting codes tend to perform much worse. When the range of arc lengths increases on sparse graphs, label-setting codes will perform only slightly worse.

6. $SLU1$ and $DLU21$ are usually slower than other label-correcting and label-setting codes for most NETGEN families. Label-correcting codes perform better for most sparse NETGEN families, but label-setting codes perform better for most dense NETGEN families.

   When the range of arc lengths increases on sparse graphs, label-setting codes tend to perform a little worse.

7. $SLU1$ and $GOR1$ usually outperform other codes for all SPACYC families except the SPACYC-PD family, for which the label-setting codes perform asymptotically the best. $DLU21$ performs asymptotically the worst for cases whose arc lengths are all positive. However, all label-correcting codes (except $GOR1$) and label-setting codes perform significantly worse for cases with negative arc lengths.

5.7.1 Conclusion and future research

Although our computational experiments are already extensive, more thorough tests may still be required to draw more solid conclusions. There are too many factors that may affect the performance of MPSP algorithms, such as requested OD pairs, arc lengths, network topologies, and node orderings.

In our experiments, for each test case, we generate only one set of requested OD pairs. Different requested OD pairs may affect the performance since the distribution of requested pairs in the OD matrix will not affect the SSSP algorithms but may affect our MPSP algorithms.

By specifying the same numbers of nodes and arcs but different random seeds, we may generate several different test cases with the same topology but different arc lengths. A more thorough experiment could generate several such networks and test all of the algorithms on these cases to compute their average performance. Due to time constraints, in this chapter we use only one random seed for each topology.

Different node orderings will significantly affect our MPSP algorithms. In this chapter, we choose a node ordering aimed at reducing the total fill-ins in the LU factorization. As discussed in Section 4.4, another ordering consideration is to group the requested OD pairs and assign them higher indices. This is difficult to test for randomly generated networks and requested OD pairs. It may be worth trying for some real-world applications where both the network topology and requested OD pairs are fixed.

Our limited results show that our MPSP algorithms seem to perform worse for general larger networks. This may make sense since after all SLU and DLU are algebraic algorithms, although we create graphical implementations for testing. In particular, SLU and DLU are more suitable for cases whose arc lengths are randomly distributed for all cases since they are designed to work independent of arc lengths. They are based on the ordering of triple comparisons. The sequence of triple comparisons is fixed and unavoidable in our MPSP algorithms. Thus, even if there exists some arc with a very large length, our algorithms could not avoid some trivial triple comparisons involving this arc, although intuitively we know in advance these operations are wasteful. Conventional SSSP algorithms, on the other hand, are based more on the comparison of arc lengths. For example, the greedy algorithm (Dijkstra's) only does triple comparisons on nodes with the smallest distance label in each iteration. Adding some "greedy" ideas to our MPSP algorithms might help them avoid many wasteful operations. However, it is still not clear how to integrate these two different methodologies.

If we suspect our MPSP algorithms may perform worse than conventional SSSP algorithms for most cases, we can design an experiment by generating the requested OD pairs most favorable to our MPSP algorithms. In particular, since both SLU and DLU can compute optimal distance faster for OD pairs with larger indices, the most favorable settings for our MPSP algorithms are: (a) find an optimal (or a very good) sparse node ordering, and (b) in that ordering, generate k requested OD pairs that span the k entries in the OD matrix in a line perpendicular to the diagonal entries. That is, entries $(n, n-k+1)$, $(n-1, n-k+2)$, $\ldots$, $(n-k+2, n-1)$, $(n-k+1, n)$ will be the set of requested OD pairs. Such a setting will force the conventional SSSP to solve k SSSP problems, but will be most advantageous for our MPSP algorithms since they are not only in the sparse ordering but are also the closest to the "southeastern" corner of the OD matrix in the new ordering.

If experiments on many random graphs with such settings still show that our MPSP algorithms do not outperform the conventional SSSP algorithms, we would conclude that our MPSP algorithms are indeed less efficient.

The experiments in this chapter are just a first step in evaluating algorithmic efficiency when solving general MPSP problems. More thorough experiments will be done in the future.

CHAPTER VI

PRIMAL-DUAL COLUMN GENERATION AND PARTITIONING METHODS

We have briefly introduced the primal-dual methods in Section 2.4 and key variable decomposition in Section 2.3.3. In this chapter, we will combine these two methods to solve MCNF problems.

6.1 Generic primal-dual method

The *primal-dual method* is a dual-ascent LP solution method which starts with a feasible dual solution, and then iteratively constructs a primal feasibility subproblem called the *restricted primal problem* (RPP) based on the complementary slackness (CS) conditions. It uses the optimal dual solution of the RPP to improve the current dual solution if primal infeasibility still exists. The algorithm terminates when all primal infeasibility disappears.

6.1.1 Constructing and solving the RPP

Suppose we are given a dual feasible solution $(\sigma, -\pi)$ where each σ_k is a free dual variable associated with the convexity constraint (or, in other words, flow balance constraint) for commodity k, and each $\pi_a \geq 0$ corresponds to the bundle constraint for arc a. Using the arc-path MCNF formulation as introduced in Section 2.3.2, we can identify the set of zero-reduced-cost columns $Q = \{Q^k, \forall k \in K\}$ and A^0, where $Q^k := \{p : PC_p^{c+\pi} = \sigma_k,\, p \in P^k\}$ for each commodity k and $A^0 := \{a : \pi_a = 0\}$ for each arc a. Define $A^+ := A \backslash A^0 = \{a : \pi_a > 0\}$. For each arc $a \in A$, we assign a nonnegative slack variable $s_a = u_a - \sum\limits_{k\in K} \sum\limits_{p\in Q^K} (B^k \delta_a^p) f_p$ and two nonnegative artificial variables w_a^+ and w_a^- representing the amount of primal infeasibility. The complementary slackness condition (CS.1) in Section 2.3.2 implies that $s_a = 0$ for each $a \in A^+$. The RPP is to minimize the sum of all artificial variables subject to the constraints whose variables have zero reduced cost in the dual feasible solution $(\sigma, -\pi)$.

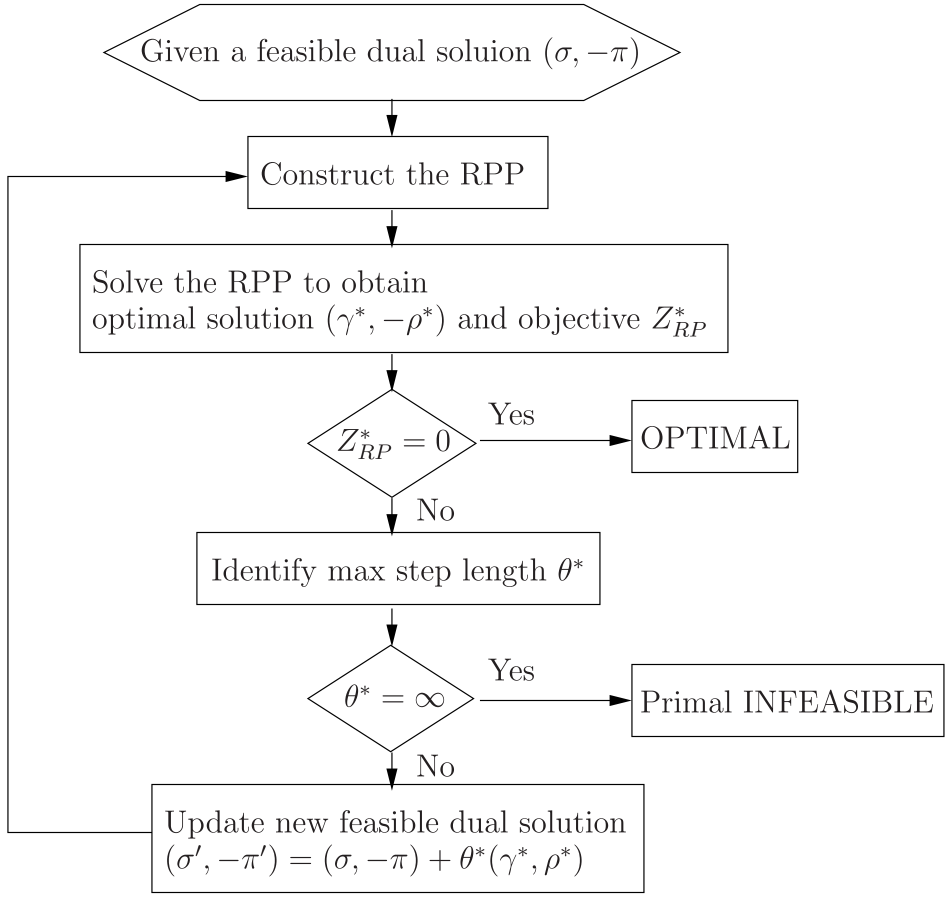


Figure 10: Generic primal-dual method for arc-path form of multicommodity flow problem

The RPP is expressed as follows:

$$\min \sum_{a \in A} (w_a^+ + w_a^-) = Z^*_{RP}(f, s, w) \qquad \text{(P\_RPP)}$$

$$s.t. \qquad \sum_{p \in Q^k} f_p = 1 \quad \forall k \in K \qquad (6.1)$$

$$\sum_{k \in K} \sum_{p \in Q^k} (B^k \delta_a^p) f_p + s_a + (w_a^+ - w_a^-) = u_a \quad \forall a \in A^0 \qquad (6.2)$$

$$\sum_{k \in K} \sum_{p \in Q^k} (B^k \delta_a^p) f_p \quad + (w_a^+ - w_a^-) = u_a \quad \forall a \in A^+ \qquad (6.3)$$

$$f_p \geq 0 \quad \forall p \in Q^k, \forall k \in K$$

$$s_a \geq 0 \quad \forall a \in A^0 \ ; \ w_a^+ \geq 0, \ w_a^- \geq 0 \quad \forall a \in A$$

whose dual is

$$\max \sum_{k \in K} \gamma_k + \sum_{a \in A} u_a(-\rho_a) = Z^*_{RD}(\gamma, \rho) \tag{D\_RPP}$$

$$s.t. \;\; \gamma_k + \sum_{a \in A} B^k \delta^p_a(-\rho_a) \leq 0 \;\; \forall p \in Q^k, \forall k \in K \tag{6.4}$$

$$0 \leq \rho_a \leq 1 \;\; \forall a \in A^0$$

$$-1 \leq \rho_a \leq 1 \;\; \forall a \in A^+$$

$$\gamma_k : \text{free} \;\; \forall k \in K$$

where γ_k are the dual variables for the convexity constraint (6.1) and $-\rho_a$ are the dual variables for the bundle constraints (6.2) and (6.3). By defining $PC^\rho_p = \sum_{a \in A} B^k \delta^p_a \rho_a$, the first constraint (6.4) of D_RPP can be rewritten as $PC^\rho_p \geq \gamma_k$, for each $p \in Q^k$ and each $k \in K$.

RPP is a LP, thus can be solved by any LP method. Section 6.1.4 discusses the degeneracy issues when generic primal-dual methods are used to solve RPP. Section 6.2 will discuss how to use Rosen's key variable decomposition method [273] to solve the RPP.

If the optimal objective value of the RPP is positive, which means there must exist primal infeasibility when using only the current column set $\{Q \cup A^0\}$ to solve the original problem, then the optimal dual solution of the RPP can be used as an improving direction for the current dual solution (this will be explained in Section 6.1.2). On the other hand, if the optimal objective value of the RPP is zero, we have achieved primal feasibility while maintaining dual feasibility and complementary slackness; thus, we have the optimal solution.

6.1.2 Improving the current feasible dual solution

In this section, we will explain why the optimal dual solution of the RPP is an improving direction for the current feasible dual solution $(\sigma, -\pi)$.

Suppose $(\gamma^*, -\rho^*)$ is an optimal dual solution for the RPP. Let $(\sigma', -\pi') := (\sigma, -\pi) + \theta(\gamma^*, -\rho^*)$. If the optimal objective value of RPP is positive, that is, $Z^*_{RP}(f, s, w) > 0$, then moving along the direction $(\gamma^*, -\rho^*)$ will increase the dual objective value since $Z_D(\pi', \sigma')$

$= Z_D(\pi, \sigma) + \theta Z_{RD}(\rho^*, \gamma^*) > Z_D(\pi, \sigma)$ where $Z_{RD}(\rho^*, \gamma^*) = Z^*_{RP}(f, s, w) > 0$.

Furthermore, for each commodity k and path $p \in Q^k$, the associated dual constraint $\sum\limits_{a \in A} B^k \delta^p_a(-\pi'_a) + \sigma'_k \leq PC^c_p + \theta(\sum\limits_{a \in A} B^k \delta^p_a(-\rho^*_a) + \gamma^*_k) \leq PC^c_p$ always holds for any positive θ. Similarly, for each arc $a \in A^0$, the associated dual constraint $\pi' = \pi + \theta\rho^* \geq 0$ holds for any positive θ. That is, if we move the current dual solution $(\sigma, -\pi)$ to $(\sigma', -\pi')$ along the direction of $(\gamma^*, -\rho^*)$, then for any positive step length θ the dual objective is improving and columns of $\{Q \cup A^0\}$ will have nonnegative reduced costs in the next iteration.

On the other hand, for column $p \in P^k \backslash Q^k$ or arc $a \in A^+$, if we choose a large step length θ, the associated dual constraint $PC^c_p + \theta(\sum\limits_{a \in A} B^k \delta^p_a(-\rho^*_a) + \gamma^*_k) \leq PC^c_p$ or $\pi' = \pi + \theta\rho^* \geq 0$ may not hold since $\sum\limits_{a \in A} B^k \delta^p_a(-\rho^*_a) + \gamma^*_k > 0$ may be true for $p \in P^k \setminus Q^k$, and $\rho^* < 0$ may be true for $a \in A^+$.

Thus, $(\gamma^*, -\rho^*)$ is a feasible direction, but the step length θ needs to be smaller than a specific threshold to maintain dual feasibility. If such a threshold does not exist, that is, if we can increase θ indefinitely while dual feasibility is always preserved, then the dual is unbounded and the primal is infeasible.

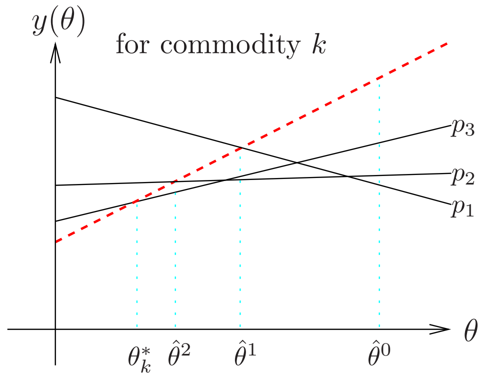
If there exists a finite optimal solution, then there must exist a finite threshold θ^* representing the maximum step length such that $(\sigma, -\pi) + \theta^*(\gamma^*, -\rho^*)$ is dual feasible but $(\sigma, -\pi) + \theta(\gamma^*, -\rho^*)$ is not, for any $\theta > \theta^*$. In particular, let $\theta_1 = \min\limits_{k \in K}\{\theta_k := \min\limits_{p \in P^k \backslash Q^k \text{ and } PC^{\rho^*}_p < \gamma^*_k} \frac{PC^{c+\pi}_p - \sigma_k}{\gamma^*_k - PC^{\rho^*}_p}\}$, and $\theta_2 = \min\limits_{a \in A^+ \text{ and } \rho^*_a < 0} \{\theta_a := \frac{\pi_a}{-\rho^*_a}\}$. Then, $\theta^* := \min\{\theta_1, \theta_2\}$. That is, θ_1 and θ_2 are the threshold for columns in $P^k \setminus Q^k$ and A^+ respectively, and the maximum step length is the smaller of these two values.

6.1.3 Computing the step length θ^*

The step length $\theta^* := \min\{\theta_1, \theta_2\}$. θ_2 can be easily computed by scanning each arc a in A^+ to find the one satisfying $\rho^*_a < 0$ with the smallest value of $\frac{\pi_a}{-\rho^*_a}$. Computing θ_1 requires more consideration since it is not clear how to choose the path $p \in P^k \setminus Q^k$ satisfying $PC^{\rho^*}_p < \gamma^*_k$ with the smallest value of $\frac{PC^{c+\pi}_p - \sigma_k}{\gamma^*_k - PC^{\rho^*}_p}$.

Let $sp(k)$ denote the shortest path for commodity k. If θ is large enough that $PC^{c+\pi+\theta\rho^*}_{sp(k)}$ $< \sigma_k + \theta\gamma^*_k$, then the dual vector $(\sigma', -\pi') = (\sigma, -\pi) + \theta(\gamma^*, -\rho^*)$ is infeasible for step length

θ. The relationship between step length θ and dual feasibility can be observed from Figure 11.



dased line: $y_k(\theta) = \sigma_k + \theta\gamma_k^*$

solid line: $y_p(\theta) = PC_p^{c+\pi} + \theta PC_p^{\rho^*}$ for path p

Figure 11: Illustraion on how to obtain θ_k^* for commodity k

In particular, we can draw a dashed line $y_k(\theta) = \sigma_k + \theta\gamma_k^*$ for commodity k, and draw a solid line $y_p(\theta) = PC_p^{c+\pi} + \theta PC_p^{\rho^*}$ for each path p (see Figure 11). Then, θ^1 makes $(\sigma', -\pi')$ $= (\sigma, -\pi) + \theta^1(\gamma^*, -\rho^*)$ infeasible if and only if there exists some path p and commodity k such that $y_p(\theta^1) = PC_{sp(k)}^{c+\pi+\theta^1\rho^*} < \sigma_k + \theta^1\gamma_k^* = y_k(\theta^1)$. In other words, to test whether a step length θ makes the dual solution infeasible, we simply compute the shortest path $sp(k)$ for each commodity k, and then compare the value of $y_{sp(k)}(\theta)$ with $y_k(\theta)$ for each commodity k. If there exists some commodity k such that $y_{sp(k)}(\theta) < y_k(\theta)$, then we know θ is still too large.

By setting $y_k(\theta) = y_p(\theta)$, we can solve for $\hat{\theta}_p$ so that $\sigma_k + \hat{\theta}_p\gamma_k^* = PC_p^{c+\pi} + \hat{\theta}_p PC_p^{\rho^*}$, and $\theta_1 = \min_p \hat{\theta}_p$. Since the number of paths connecting an OD pair k may be large, how to efficiently compute θ_1 is itself a challenging problem.

Polak [268] gives a Floyd-Warshall like algorithm to solve this problem, but its implementation is not clear. Barnhart and Sheffi [32, 38] use a shortest path procedure employing a multiple labeling scheme which first determines all possible value of $PC_p^{\rho^*}$ for each path $p \in P^k$ and for each OD pair k. Then, of all the possible paths with a specific value of $PC_p^{\rho^*}$, they determine the shortest one using arc length $c + \pi$. However, it is also not clear how to efficiently implement this multiple labeling scheme.

Here we propose a binary search method and an iterative method to compute θ^*.

6.1.3.1 Binary search method for finding θ^*

The idea of our binary search method is to identify the range $[\theta_l, \theta_u]$ for θ_1 such that $y_{sp(k)}(\theta_l) > y_k(\theta_l)$ for each commodity k, and $y_{sp(k)}(\theta_u) < y_k(\theta_u)$ for some commodity k. Then we iteratively shrink this range until its width is smaller than a threshold ϵ (the machine precision number, for example).

Since $\theta^* := \min\{\theta_1, \theta_2\}$, we can first identify θ_2, then use it to search for θ_1. In particular, let $sp(k)$ denote the shortest path for commodity k using $c + \pi'$ as the arc lengths. If $y_{sp(k)}(\theta_2) > y_k(\theta_2)$ for each commodity k, then it means $\theta_2 < \theta_1$ and thus $\theta^* := \theta_2$. Otherwise, we can use θ_2 as θ_u to start the binary search for $\theta^* := \theta_1$.

Let $K = \{(s_i, t_i)\}$ be the initial set of requested OD pairs. Procedure *bins_theta*$(K, \pi, \sigma, \rho^*, \gamma^*)$ first finds for $\theta' = \theta_2 = \min\limits_{a \in A^+ \text{ and } \rho_a^* < 0} \{\theta_a := \frac{\pi_a}{-\rho_a^*}\}$ if it exists. Furthermore, if $\theta' = \theta_2$ makes $(\sigma', -\pi') = (\sigma, -\pi) + \theta'(\gamma^*, -\rho^*)$ feasible, then $\theta^* = \theta_2$ has been identified.

If there exists no arc $a \in A^+$ such that $\rho_a^* < 0$, then we use any positive θ' to start the binary search. In particular, if θ' makes $(\sigma', -\pi')$ feasible (i.e., $y_{sp(k)}(\theta') > y_k(\theta')$ for each k), then we set $\theta_l = \theta'$, $\theta_u = 2\theta'$ and then test whether θ_u makes $(\sigma', -\pi')$ infeasible. If so, then we find a range $[\theta_l, \theta_u]$ for θ^* where $\theta_u = 2\theta_l$. Otherwise, we keep doubling θ_l and θ_u until θ_u makes $(\sigma', -\pi')$ infeasible.

If, on the other hand, the initial θ' makes $(\sigma', -\pi')$ infeasible, then we set $\theta_l = \frac{1}{2}\theta'$, $\theta_u = \theta'$ and then test whether θ_l makes $(\sigma', -\pi')$ feasible or not. Using a similar procedure as in the previous paragraph but searching along the opposite direction, a range $[\theta_l, \theta_u = 2\theta_l]$ contains θ^* can be identified.

After we obtain the range $[\theta_l, \theta_u]$ for θ^*, we can set $\theta' = \frac{\theta_l + \theta_u}{2}$ and check whether θ' can be used as a new upper bound θ_u (or a new lower bound θ_l) so that the range $[\theta_l, \theta_u]$ is cut in half. We iterate until the width is smaller than a threshold ϵ.

To guarantee that θ^* is sufficiently accurate, we may set the threshold ϵ to be the machine precision number, which is usually 10^{-9} or even smaller. However, such a small threshold value may require many iterations of binary search and makes the procedure inefficient. Next we propose an iterative method, which we will use for our implementation.

Procedure bins_theta($\mathbf{K} := \{(\mathbf{s_i}, \mathbf{t_i})\}, \pi, \sigma, \rho^*, \gamma^*$)
begin
  $[\theta_l, \theta_u, \overline{OD}] = FindRange(K, \pi, \sigma, \rho^*, \gamma^*)$;
  Let $\theta' = \theta_u$
  while $\theta_u > \theta_l + \epsilon$
    Let $\theta' = \frac{\theta_l + \theta_u}{2}$,
    $\overline{OD} = UpdateOD(\overline{OD}, \pi, \sigma, \theta')$;
    if $\overline{OD} \neq \emptyset$ ***then*** $\theta_u := \theta'$
    else $\theta_l := \theta'$
  return $\theta^* = \theta'$
end

Subprocedure $FindRange(K, \pi, \sigma, \rho^*, \gamma^*)$
begin
  Set an initial $\theta_0 > 0$, let $\hat{A}^+ = \{a : a \in A^+ \text{ and } \rho^*_a < 0\}$ and $\overline{OD} = K$
  if $\hat{A}^+ \neq \emptyset$ ***then*** $\theta' := \min_{a \in \hat{A}^+} \{\theta_a := \frac{\pi_a}{-\rho^*_a}\}$
  else $\theta' = \theta_0$
  Initialize $\theta_l = \theta_u = \theta'$
  $\overline{OD} = UpdateOD(\overline{OD}, \pi, \sigma, \theta')$;
  if $\overline{OD} \neq \emptyset$ ***then*** $\zeta_1 = 1$,
  else if $\hat{A}^+ \neq \emptyset$
    return $[\theta_l, \theta_u, \overline{OD}]$
  if $\zeta_1 = 1$ ***then*** $\zeta_2 = 1$ and $[\omega_l, \omega', \omega_u] := [\frac{1}{2}, 1, \frac{1}{2}]$
  else $\zeta_2 = 0$ and $[\omega_l, \omega', \omega_u] := [1, 2, 2]$
  while $\zeta_1 = \zeta_2$
    if $\zeta_2 = 0$ ***then*** reset $\overline{OD} = K$
    $\overline{OD} = UpdateOD(\overline{OD}, \pi, \sigma, \theta')$;
    if $\overline{OD} \neq \emptyset$ ***then*** $\zeta_2 = 1$
    else $\zeta_2 = 0$
    if $\zeta_1 = \zeta_2$ ***then*** $[\theta_l, \theta', \theta_u] := [\theta_l \times \omega_l, \theta' \times \omega', \theta_u \times \omega_u]$
  return $[\theta_l, \theta_u, \overline{OD}]$
end

Subprocedure $UpdateOD(\overline{OD}, \pi, \sigma, \theta')$
begin
  Compute $(\pi', \sigma') := (\pi, \sigma) + \theta'(\rho^*, \gamma^*)$
  Using $c + \pi'$ as arc lengths, use a MPSP algorithm to compuate $sp(k)$ $\forall k \in \overline{OD}$
  Remove those commodities k satisyfing $PC^{c+\pi'}_{sp(k)} \geq \sigma'_k$ from the set $\overline{OD}$
  return $\overline{OD}$
end

6.1.3.2 Iterative method for finding θ^*

From Figure 11, we know that for each commodity k, θ_k^* can be determined by the intersection of $y_k(\theta) = \sigma_k + \theta_p \gamma_k^*$ and $y_p(\theta) = PC_p^{c+\pi} + \theta_p PC_p^{\rho^*}$ for some path p from origin s_k to destination t_k. The challenge will be to efficiently determine the right path p for each commodity k.

Procedure iter_theta($\mathbf{K} := \{(\mathbf{s_i}, \mathbf{t_i})\}, \pi, \sigma, \rho^*, \gamma^*$)
begin
  $[\theta_l, \theta_u, \overline{OD}] = FindRange(K, \pi, \sigma, \rho^*, \gamma^*)$;
  if $\theta_l = \theta_u$
    return $\theta^* = \theta_u$
  Set $\hat{\theta}^0 := \theta_u$, $t = 0$
  if $\overline{OD} = \emptyset$ ***then*** reset $\overline{OD} := K$
  while $\overline{OD} \neq \emptyset$
    Compute $(\pi', \sigma') := (\pi, \sigma) + \theta'(\rho^*, \gamma^*)$, $t := t + 1$
    Using $c + \pi'$ as arc lengths, solve the MPSP problem to find $sp(k)$ $\forall k \in \overline{OD}$
    Trace path $sp(k)$ to compute $PC_{sp(k)}^{\rho^*}$ for each commodity $k \in \overline{OD}$
    Compute $\hat{\theta}^t = \min\limits_{k \in \overline{OD}} \frac{PC_{sp(k)}^{c+\pi'} - \sigma_k}{\gamma_k^* - PC_{sp(k)}^{\rho^*}}$
    $\overline{OD} = UpdateOD(\overline{OD}, \pi, \sigma, \hat{\theta}^t)$;
  return $\theta^* = \hat{\theta}^t$
end

Our iterative method uses an initial $\hat{\theta}^0$ that produces an infeasible dual solution $(\sigma', -\pi')$ $= (\sigma, -\pi) + \hat{\theta}^0(\gamma^*, -\rho^*)$, and then solves the MPSP problem to find $sp(k)$ and $PC_{sp(k)}^{c+\pi'}$ for each commodity $k \in \overline{OD}$. Using the shortest path $sp(k)$ for commodity k, we can also compute $PC_{sp(k)}^{\rho}$ and determine a better $\hat{\theta}^1 = \frac{PC_{sp(k)}^{c+\pi'} - \sigma_k}{\gamma_k^* - PC_{sp(k)}^{\rho^*}}$ by finding the intersection of the line $y_k(\theta)$ and the line $y_{sp(k)}(\theta)$.

Using the new $\hat{\theta}^1$, we can update $\overline{OD}$, the set of commodities that still satisfies $y_k(\hat{\theta}^1) > y_{sp(k)}(\hat{\theta}^1)$. Let $\hat{\theta}^t$ denote the step length in the t^{th} iteration computed by this iterative method. It is easy to observe that $\hat{\theta}^1 < \hat{\theta}^0$, and that $\hat{\theta}^{t+1} < \hat{\theta}^t$. Thus, the sequence $\hat{\theta}^t$ is decreasing. Furthermore, if $y_{k'}(\hat{\theta}^t) < y_{sp(k')}(\hat{\theta}^t)$ for commodity k', then $y_{k'}(\hat{\theta}^{t+1}) < y_{sp(k')}(\hat{\theta}^{t+1})$. These properties can be easily verified from Figure 11. Thus, our iterative method iteratively shrinks $\overline{OD}$ and $\hat{\theta}^t$ until finally $\overline{OD}$ becomes empty.

We use Figure 11 to illustrate the procedure. Suppose commodity k has only three possible OD paths: p_1, p_2, and p_3. We start with an initial $\hat{\theta}^0$ that satisfies $y_k(\hat{\theta}^0) >$

$y_{sp(k)}(\hat{\theta}^0)$ for commodity k. Then, using $c + \pi + \hat{\theta}^0\rho^*$ as the arc lengths, we identify the shortest path $sp(k) = p_1$. We compute the intersection of $y_k(\theta)$ and $y_{sp(k)}(\theta)$ to determine a new step length $\hat{\theta}^1$. Using $c + \pi + \hat{\theta}^1\rho^*$ as the arc lengths, we identify the shortest path $sp(k) = p_2$ that still satisfies $y_k(\hat{\theta}^1) > y_{sp(k)}(\hat{\theta}^1)$. We compute the intersection of $y_k(\theta)$ and $y_{sp(k)}(\theta)$ to determine a new step length $\hat{\theta}^2$. From Figure 11, we find $y_k(\hat{\theta}^2) > y_{p_3}(\hat{\theta}^2)$ where path p_3 is the shortest path for commodity k using $c + \pi + \hat{\theta}^2\rho^*$ as the arc lengths. After computing $\hat{\theta}^3 = \frac{PC_{p_3}^{c+\pi'} - \sigma_k}{\gamma_k^* - PC_{p_3}^{\rho^*}}$ we find that $y_k(\hat{\theta}^3) = y_{sp(k)}(\hat{\theta}^3)$. Therefore, we have determined the step length $\theta_k^* = \hat{\theta}^3$ for commodity k.

In the worst case, this iterative method may have $\sum_k |P^k|$ iterations of MPSP computations, where P^k is the set of all possible paths for OD pair k. However, in practice, it requires only few iterations to converge to the exact θ^* since the size of $\overline{OD}$ is nonincreasing. We observe that it is faster than the binary search method because the binary search method is doing blind search and converges slowly near the end of the procedure. For the primal-dual method, we need to compute θ^* exactly so that the dual objective is strictly ascending. Numerical precision problems in the binary search method create difficulties in making improvements due to zero step length.

Both of our proposed methods for computing θ^* require many iterations of MPSP computation. Thus, an efficient MPSP algorithm will greatly improve the practical running time. There may exist other methods more efficient than our iterative method for computing the step length θ^*. We think this is an interesting topic for future research.

6.1.4 Degeneracy issues in solving the RPP

At each iteration, after identifying the step length θ^* a new, improved feasible dual solution $(\sigma', -\pi') = (\sigma, -\pi) + \theta^*(\gamma^*, -\rho^*)$ can be computed and used to construct a new RPP with only the zero-reduced-cost columns.

The RPP can be solved by any LP method. In our implementation, we use CPLEX 7.0 to solve the RPP. CPLEX has its own techniques to resolve degeneracy. However, our experiments show that sometimes degeneracy between different iterations of RPP may slow down the primal-dual methods even when CPLEX's techniques are used.

Suppose $Z^*_{RP_t}$ is the optimal objective for the t^{th} RPP (the RPP in the t^{th} primal-dual iteration). Since the primal infeasibility stays strictly positive until the last primal-dual iteration, the original dual objective will be strictly improving at each iteration. The total primal infeasibility $Z^*_{RP_t}$ should remain nonincreasing. That is, $Z^*_{RP_{t+1}} \leq Z^*_{RP_t}$. Equality holds whenever a primal degenerate pivot occurs between iteration t and iteration $t+1$.

Suppose we add columns to the RPP to construct a larger problem PP_A. That is, problem PP_A contains three sets of columns: (1) artificial columns with unit objective coefficients, (2) all shortest paths with zero objective coefficients for each commodity, and (3) all other paths which are not the shortest, but also have zero objective coefficients for each commodity. The original RPP can be thought as a "restricted master" problem of PP_A that uses only the first two groups of columns. A feasible basis and a basic feasible solution (b.f.s.) for RPP will also be a feasible basis and b.f.s. for PP_A. Therefore, when $Z^*_{RP_{t+1}} = Z^*_{RP_t}$, moving from iteration t to iteration $t+1$ in the primal-dual method can be thought as a degenerate pivot for the larger problem PP_A.

CPLEX does not avoid such degenerate pivots between primal-dual iterations since the new RPP is constructed from scratch and CPLEX solves the new RPP from scratch as well. Whatever degeneracy-resolving techniques CPLEX applies for solving the previous RPP do not carry over when solving the new RPP. In our experiments, we do experience such degeneracy. To illustrate the extent of the affect of degeneracy, we note a case in which the primal infeasibility remains unchanged for approximately 1000 iterations, causing the method to take several hours to terminate.

To avoid these degenerate pivots, we perturb the objective coefficients of the artificial variables. In particular, instead of using 1 as the objective coefficient for all artificial variables (as suggested in most textbooks), we assign random integers to be the new objective coefficients for artificial variables. This change will not affect the validity of the primal-dual method. The optimal dual solution for the new RPP will be different from the original one, but will still be a valid dual improving direction. All the other operations such as computing the step length θ^* and constructing the new RPP will still work. The algorithm will terminate since each primal-dual iteration strictly improves the primal infeasibility and the dual objective value, and dual feasibility and complementary slackness conditions are still maintained throughout the whole procedure.

In our implementation, we check whether the primal infeasibility decreases between primal-dual iterations. If it remains unchanged for two iterations, then we perturb the objective coefficients of artificial variables once. This change does resolve the degenerate pivots and shortens the computational time. For example, the previous case, which requires several hours to perform 1000 degenerate pivots, solves in several seconds.

In next section, we describe a new method for solving the RPP which may be efficient for cases with many commodities.

6.2 Primal-dual key path method

As introduced in Section 2.3.3, the key variable decomposition method, first proposed by Rosen [273], has been used by Barnhart et al. [36] in conjunction with column generation with success.

Barnhart et al. [36] use key variable decomposition with primal simplex column generation. In this section, we apply a similar technique to solve the RPP from the primal-dual method. In particular, the primal-dual key path method follows the generic primal-dual steps (see Figure 10), except that it uses the key variable decomposition method to solve the RPP. Thus in this section we focus on solving the RPP by key variable decomposition.

6.2.1 Cycle and Relax(i) RPP formulations

Given a feasible dual solution $(\sigma, -\pi)$, we can construct P_RPP, the path formulation of the RPP (see page 156). For each commodity k, we choose a shortest path $key(k) \in Q^k$, where $Q^k := \{p : PC_p^{c+\pi} = \sigma_k,\, p \in P^k\}$. After performing column operations to eliminate the key columns, we obtain the following key cycle formulation of the RPP:

$$\min \sum_{a\in A}(w_a^+ + w_a^-) = Z^*_{CRP}(f,s,w) \tag{C\_RPP}$$

$$s.t. \qquad \sum_{p\in Q^k} f_p = 1 \quad \forall k\in K \tag{6.5}$$

$$\sum_{k\in K}\sum_{p\in Q^k} B^k(\delta_a^p - \delta_a^{key(k)})f_p + s_a + (w_a^+ - w_a^-) = u_a - \sum_{k\in K} B^k\delta_a^{key(k,i)} \quad \forall a\in A^0 \tag{6.6}$$

$$\sum_{k\in K}\sum_{p\in Q^k} B^k(\delta_a^p - \delta_a^{key(k)})f_p \qquad + (w_a^+ - w_a^-) = u_a - \sum_{k\in K} B^k\delta_a^{key(k,i)} \quad \forall a\in A^+ \tag{6.7}$$

$$f_p \ge 0 \quad \forall p\in Q^k, \forall k\in K$$

$$s_a \ge 0 \quad \forall a\in A^0 \ ; \ w_a^+ \ge 0,\ w_a^- \ge 0 \quad \forall a\in A$$

Note that C_RPP has the same objective function as P_RPP since all the shortest path columns have zero objective coefficients. C_RPP should be no easier or harder than P_RPP since the transformation does not change the problem structure.

When the number of commodities (OD pairs) is huge, both C_RPP and P_RPP will have many constraints, which makes the RPP more difficult. The key variable transformation relaxes the nonnegativity constraints for each key path (i.e., it allows $f_{key(k)}$ to be negative) and results in an easier problem R_RPP(i), where i denotes the index of iteration:

$$\min \sum_{a\in A}(w_a^{i+} + w_a^{i-}) = Z^*_{RRP(i)}(f,s,w) \tag{R\_RPP(i)}$$

$$s.t. \quad \sum_{k\in K}\sum_{p\in Q^k} B^k(\delta_a^p - \delta_a^{key(k,i)})f_p^i + s_a^i + (w_a^{i+} - w_a^{i-}) = u_a - \sum_{k\in K} B^k\delta_a^{key(k,i)} \quad \forall a\in A^0 \tag{6.8}$$

$$\sum_{k\in K}\sum_{p\in Q^k} B^k(\delta_a^p - \delta_a^{key(k,i)})f_p^i \qquad + (w_a^{i+} - w_a^{i-}) = u_a - \sum_{k\in K} B^k\delta_a^{key(k,i)} \quad \forall a\in A^+ \tag{6.9}$$

$$f_p^i \ge 0 \quad \forall p\in Q^k \setminus f^i_{key(k,i)}, \forall k\in K;\ f^i_{key(k,i)} \text{ :free } \quad \forall k\in K$$

$$s_a^i \ge 0 \quad \forall a\in A^0 \ ; \ w_a^{i+} \ge 0,\ w_a^{i-} \ge 0 \quad \forall a\in A$$

whose dual is

$$\max \sum_{a \in A} (u_a - \sum_{k \in K} B^k \delta_a^{key(k,i)})(-\rho_a^i) = Z^*_{RRD(i)}(\rho) \qquad \text{(R\_RDP}(i))$$

$$\begin{aligned} s.t. \quad & \sum_{a \in A} B^k (\delta_a^p - \delta_a^{key(k,i)})(-\rho_a^i) \leq 0 \quad \forall p \in Q^k, \forall k \in K \\ & 0 \leq \rho_a^i \leq 1 \quad \forall a \in A^0 \\ & -1 \leq \rho_a^i \leq 1 \quad \forall a \in A^+ \end{aligned} \tag{6.10}$$

6.2.2 Key variable decomposition method for solving the RPP

The RPP can be solved by iteratively application of the key variable decomposition method as illustrated in Figure 12. In each iteration of the key variable decomposition method, an easier problem, R_RPP(i), is solved to optimality.

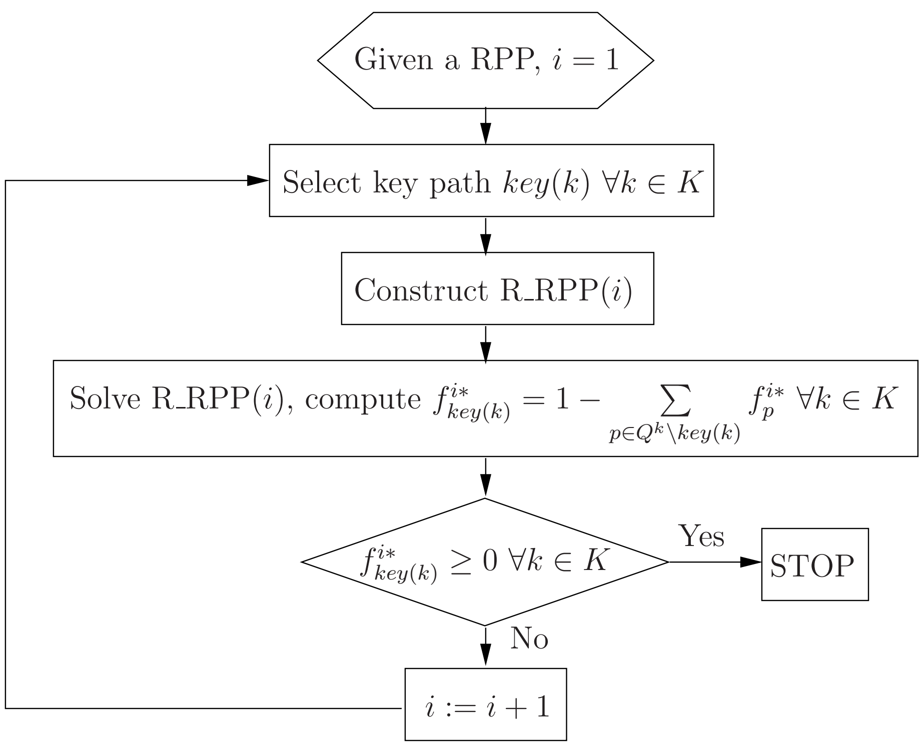


Figure 12: Key variable decomposition method for solving the RPP

After solving R_RPP(i), the algorithm will check the sign of key variables by calculating $f^{i*}_{key(k,i)} = 1 - \sum_{p \in Q^k \setminus key(k,i)} f_p^{i*}$. For those key variables with negative signs, the algorithm will perform a *key variable change* operation to replace the current key variable with a new one. Among all positive shortest path variables eligible for selection, the one with the largest value is usually chosen; intuitively, that path is more likely to have positive flow in the optimal solution. That is, $key(k, i+1) = \arg\max_{q \in Q^k} f_q^{i*}$.

The proof of finiteness and optimality for the key variable decomposition method can be found in Barnhart et al. [36]. In particular, after solving the R_RPP(i), if there exists some key variable $f_{key(k,i)}^{i*} < 0$, there must also exist a shortest path $q(k)$ such that $f_{q(k)}^{i*} > 0$, and column $q(k)$ is in the optimal basis. The optimal dual solution of RPP(i), $-\rho^{i*}$, remains as a basic feasible solution for R_RDP(i+1). The swap of key path from $key(k, i)$ to $q(k)$ does not affect using $-\rho^{i*}$ as an initial basic feasible solution for R_RDP(i+1). Furthermore, $Z^*_{RRD(i)}$ $= Z_{RRD(i+1)}$, and the complementary slackness conditions are maintained after the swap of the key path. Since $f_{key(k,i)}^{i+1} < 0$, a dual simplex pivot has to be performed to achieve optimality for R_RPP($i+1$). Using the dual simplex method with degeneracy resolving techniques, the dual objective is strictly improved in next iteration, and the algorithm will terminate in a finite number of iterations. If the key variable decomposition method takes $\hat{i}$ iterations to solve the RPP, then $Z^*_{RRP(\hat{i})} = Z^*_{RP}$.

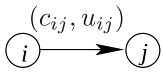
After the RPP is solved, we use the same procedure as introduced in Section 6.1.3 to identify the step length θ^*. Then, we can update $\pi' = \pi + \theta^* \rho^*$, compute $\sigma_k^* = PC_{sp(k)}^{c+\pi'}$ where $sp(k)$ is the shortest path of commodity k using $c + \pi'$ as the arc lengths, and finally construct a new RPP for next primal-dual iteration.

This algorithm maintains dual feasibility and complementary slackness conditions while trying to achieve primal feasibility (which will be attained when all key variables become nonnegative).

6.2.3 Degeneracy issues between key path swapping iterations

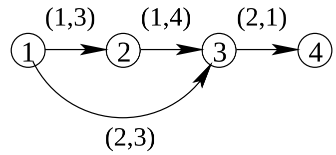
The finiteness of the key variable decomposition method for solving the relaxed problem R_RPP(i) relies on the assumption of techniques to resolve degeneracy. Degeneracy is not an issue when we solve the R_RPP(i), but it will become a crucial factor between iterations of solving the R_RPP(i).

Suppose we are to solve the ODMCNF problem shown in Figure 13, which requires us



c_{ij}: cost of arc (i,j)

u_{ij}: capacity of arc (i,j)



K^1: $(s^1, t^1) = (1,4)$, $B^1 = 2$

initial $\pi_a = 0 \; \forall a \in A$

arc (i,j)	index
$(1,3)$	1
$(1,3)$	2
$(2,3)$	3
$(3,4)$	4

$$Q^1: \quad \begin{array}{l} p_1: \; 1 \to 2 \to 3 \to 4 \\ p_2: \; 1 \to 3 \to 4 \end{array}$$

$$min \; \sum_{a=1}^{4} w_a^+ + \sum_{a=1}^{4} w_a^- \qquad \text{(RPP)}$$

subject to

$$\begin{array}{rrl} 2f_{p_1} & & + s_1 + w_1^+ - w_1^- = 3 \\ & 2f_{p_2} & + s_2 + w_2^+ - w_2^- = 3 \\ 2f_{p_1} & & + s_3 + w_3^+ - w_3^- = 4 \\ 2f_{p_1} & + 2f_{p_2} & + s_4 + w_4^+ - w_4^- = 1 \\ f_{p_1} & + \; f_{p_2} & = 1 \end{array}$$

$f_p \geq 0 \; \forall p \in Q^1$; $s_a \geq 0, w_a^+, w_a^-$: free $\forall a \in A$

Optimal solution:

$w_4^{-*} = 1$

$w_a^{-*} = 0$, $a = 1,2,3$

$w_a^{+*} = 0$, $a = 1,2,3,4$

both $f^*_{p_1}, f^*_{p_2} \in [0, 1.5]$, but $f^*_{p_1} + f^*_{p_2} = 1$

Figure 13: A small ODMCNF example and its RPP formulation

to send 2 units of flow from node 1 to node 4. Using $-\pi = 0$ as an initial dual feasible solution, the restricted network consists of 4 arcs, $(1,2)$, $(1,3)$, $(2,3)$, and $(3,4)$, because there are only 2 paths ($1 \to 2 \to 3 \to 4$ and $1 \to 3 \to 4$) that have zero reduced cost. Note that there should exist other paths from node 1 to node 4 using other arcs; otherwise the problem is primal infeasible. It is easy to see that the optimal solution of this RPP is $w_4^{+*} = 1$, $w_a^{+*} = 0$ for $a = 1,2,3$; $w_a^{-*} = 0$ for $a = 1,2,3,4$; and $f^{1*}_{p_1} \in [0, 1.5]$, $f^{1*}_{p2} \in [0, 1.5]$, with $f^{1*}_{p_1} + f^{1*}_{p_2} = 1$. That is, there exist infinitely many $f^{1*}_{p_1}$ and $f^{1*}_{p_2}$. The exact value of $f^{1*}_{p_1}$ and $f^{1*}_{p_2}$ will not affect the objective function.

Figure 14 illustrates the steps of the key variable decomposition method for solving this example. Suppose we choose p_1 to be the key path in the first iteration when solving R_RPP(1). There exist multiple optimal primal solutions to R_RPP(1); that is, $0 \leq f^{1*}_{p_2} \leq 1.5$. If we specify $f^{1*}_{p_2}$ to be any value in $[0, 1]$, then $f^{1*}_{p_1} = 1 - f^{1*}_{p_2}$ will be feasible. However, if we choose $1 < f^{1*}_{p_2} \leq 1.5$, then $-0.5 \leq f^{1*}_{p_1} < 0$ is not primal feasible for RPP, and we

$Q^1: \quad \begin{array}{l} p_1: 1 \to 2 \to 3 \to 4 : key(1,1) \\ p_2: 1 \to 3 \to 4 \end{array}$

$$min \sum_{a=1}^{4} w_a^+ + \sum_{a=1}^{4} w_a^- \qquad \text{(R\_RPP(1))}$$

subject to

$$\begin{aligned} -2f_{p_2}^1 + s_1 + w_1^+ - w_1^- &= 1 \\ 2f_{p_2}^1 + s_2 + w_2^+ - w_2^- &= 3 \\ -2f_{p_2}^1 + s_3 + w_3^+ - w_3^- &= 2 \\ s_4 + w_4^+ - w_4^- &= 1 \end{aligned}$$

$f_{p_1}^1$:free, $f_{p_2}^1 \geq 0$; s_a, w_a^+, w_a^-: free $\forall a \in A$

Optimal solution:

$$\begin{aligned} &w_4^{-*} = 1,\ f_{p_2}^{1*} \in [0, 1.5] \\ &w_a^{-*} = 0,\ a = 1,2,3 \\ &w_a^{+*} = 0,\ a = 1,2,3,4 \\ &f_{p_1}^{1*} \in [-0.5, 1] \end{aligned}$$

CPLEX gives $f_{p_2}^{1*} = 1.5$, thus $f_{p_1}^{1*} = -0.5 < 0$

Therefore, we have to swap $key(1,2) = p_2$

$Q^1: \quad \begin{array}{l} p_1: 1 \to 2 \to 3 \to 4 \\ p_2: 1 \to 3 \to 4 : key(1,2) \end{array}$

$$min \sum_{a=1}^{4} w_a^+ + \sum_{a=1}^{4} w_a^- \qquad \text{(R\_RPP(2))}$$

subject to

$$\begin{aligned} 2f_{p_1}^2 + s_2 + w_1^+ - w_1^- &= 3 \\ -2f_{p_1}^2 + s_2 + w_2^+ - w_2^- &= 1 \\ 2f_{p_1}^2 + s_3 + w_3^+ - w_3^- &= 4 \\ s_4 + w_4^+ - w_4^- &= -1 \end{aligned}$$

$f_{p_2}^2$:free, $f_{p_1}^2 \geq 0$; s_a, w_a^+, w_a^-: free $\forall a \in A$

Optimal solution:

$$\begin{aligned} &w_4^{-*} = 1,\ f_{p_1}^{2*} \in [0, 1.5] \\ &w_a^{-*} = 0,\ a = 1,2,3 \\ &w_a^{+*} = 0,\ a = 1,2,3,4 \\ &f_{p_2}^{2*} \in [-0.5, 1] \end{aligned}$$

CPLEX gives $f_{p_1}^{2*} = 0$, thus $f_{p_2}^{2*} = 1 > 0$, STOP

If the solver gives $f_{p_1}^{2*} = 1.5$, then $f_{p_1}^{2*} = -0.5 < 0$,
Therefore, we have to swap $key(1,3) = p_1$ again.

Figure 14: Infinite iterations of key path swapping due to dual degeneracy

have to swap key paths.

Unfortunately, CPLEX (with default settings) will give $f_{p_2}^{1*} = 1.5$, and thus we have to set the key path for the second iteration, $key(1,2)$, to be path p_2 and construct R_RPP(2).

Again, R_RPP(2) has multiple optimal primal solutions $0 \leq f_{p_1}^{2*} \leq 1.5$, and thus the situation is exactly the same as the first iteration. Fortunately, CPLEX with default settings will give us $f_{p_1}^{2*} = 0$. However, there is no guarantee that CPLEX or any other solver will have good luck all the time. If the solver happens to choose $1 < f_{p_1}^{2*} \leq 1.5$ as its optimal solution, then $-0.5 \leq f_{p_2}^{2*} < 0$ is not primal feasible for RPP, and we have to swap the key path back to p_1. For CPLEX, since it will only choose the one that are basic feasible, so $f_{p_2}^{2*} = 0$ or $f_{p_2}^{2*} = 1.5$, a 50% chance to terminate earlier. In other words, since the LP solver does not distinguish between multiple optimal solutions, it is possible that a "bad" optimal solution will be chosen, resulting in infinite loops of key path swapping. Indeed, in our computational experiments, this happens very frequently because the RPP is very dual degenerate.

Using 1 as an upperbound for each path flow f_p^i may avoid some cycling, since such upperbound forces each non-key-path flow less than 1 which in turn makes the sum of their flows smaller, and thus more likely to make the flow on key path nonnegative. However, it comes with the difficulty that we will have extra dual variables for these upper bound constraints. Bookkeeping for these extra dual variables, one for each path, may be difficult since there may be many of them and for each we require a record of its associated path. Furthermore, it is still possible that each $f_p^i < 1$ but their sum exceeds 1. Thus, cycling may still occur.

Because the objective function in RPP and R_RPP(i) contains only artificial variables with no other restriction on the path flows, the LP solver may give the optimal path flow "blindly" even if there does exist a better optimal flow solution. Here, we propose a perturbation method that assigns a very small random positive objective coefficient for all path variables. This idea is inspired by observing the example in Figure 14. In particular, after we relax the nonnegativity constraints for the key variables, we will prefer the optimal path flow solution to be as small as possible so that the convexity constraints will be more likely to make the key variable positive.

Therefore, instead of assigning a zero objective coefficient for each path flows, we give a very small positive random cost coefficient for each path flow. Thus, the optimal flow solution will tend to be smaller and key path flows will be more likely to be nonnegative. Note that such a change will affect dual feasibility of the R_RDP(i), the dual of the R_RPP(i). Thus we have to add an extra loop to compensate for the effect of the perturbation.

In particular, the optimal dual solution of R_RPP(i), $-\rho^*$, will be in the set $\{-1, 0, 1\}$. After the perturbation, $-\rho^*$ may become $-1 \pm \epsilon$, $\pm\epsilon$, or $1 \pm \epsilon$, where ϵ is a very small number. Thus we reset each deviated $-\rho_a^*$ to be its nearest integer, -1, 0, or 1, so that the affect of perturbation to $-\rho^*$ is compensated and corrected.

Table 30: Four methods for solving ODMCNF

Algorithm	Problem formulation	Solution method
PD	arc-path P_RPP formulation (see page 156)	generic primal-dual method
KEY	arc-path R_RPP(i) formulation (see page 166)	primal-dual key path method
DW	arc-path P_PATH formulation (see page 21)	Dantzig-Wolfe decomposition
NA	node-arc formulation (see page 13)	CPLEX

Table 31: Problem characteristics for 17 randomly generated test problems

Problem	$\|N\|$	$\|A\|$	$\|K\|$	$\|N\|\,\|K\|+\|A\|$	$\|M\|\,\|K\|$
$\mathbf{P}_1$	200	517	323	65,117	166,991
$\mathbf{P}_2$	200	501	323	65,117	161,823
$\mathbf{P}_3$	200	508	323	65,117	164,084
$\mathbf{P}_4$	200	520	317	63,917	164,840
$\mathbf{P}_5$	300	760	561	168,817	426,360
$\mathbf{P}_6$	300	811	530	159,517	429,830
$\mathbf{P}_7,\ldots,\mathbf{P}_{17}$	49	130	427	21,440	55,510

6.3 Computational results

We implemented four algorithms for solving the multicommodity flow problems. Each algorithm solves a different multicommodity flow formulation (see Table 30). All of our tests were run using CPLEX 7.0 on a Sun Sparc machine with 512MB RAM. We tested 17 problem sets obtained from Hane [165] (see Table 31). $P_1,\ldots,P_6$ are directed graphs with different topologies and commodities, and $P_7,\ldots,P_{17}$ are undirected graphs with different commodities but the same topology. All of these 17 problems use OD pairs as commodities, but the commodity costs are uniform on each arc. That is, $c_a^k = c_a$, for each arc a in A and each commodity k in K. Our MPSP algorithm $DLU2$ are used for computing shortest paths in MCNF algorithms PD, KEY, and DW.

6.3.1 Algorithmic running time comparison

Table 32 lists the overall running time of each algorithm on each problem, as well as percentage of time spent by different procedures. In particular, t_{all} denotes the total running time. We divide the running time of algorithm PD into three components: t_{sp}, t_θ and $\bar{t}$, where t_{sp} represents the time spent on generating all shortest paths for constructing the RPP, t_θ is the time spent on computing the step length θ^*, and $\bar{t}$ is the remaining time mostly spent

Table 32: Total time (ms) of four algorithms on problems $P_7, \ldots, P_{17}$

	PD				KEY					DW			NA
	$\mathbf{t}_{sp}\%$	$\mathbf{t}_\theta\%$	$\bar{t}\%$	$\mathbf{t}_{all}$	$\mathbf{t}_{sp}\%$	$\mathbf{t}_\theta\%$	$\mathbf{t}_{key}\%$	$\bar{t}\%$	$\mathbf{t}_{all}$	$\mathbf{t}_{sp}\%$	$\bar{t}\%$	$\mathbf{t}_{all}$	$\mathbf{t}_{all}$
$\mathbf{P}_1$	54%	28%	18%	**291**	43%	22%	26%	9%	**366**	31%	69%	**57**	**5532**
$\mathbf{P}_2$	39%	46%	15%	**1444**	19%	22%	56%	3%	**2584**	31%	69%	**65**	**5934**
$\mathbf{P}_3$	42%	44%	15%	**987**	24%	25%	48%	4%	**1713**	29%	71%	**58**	**5611**
$\mathbf{P}_4$	42%	42%	16%	**699**	22%	22%	53%	4%	**1334**	27%	73%	**62**	**5479**
$\mathbf{P}_5$	50%	34%	16%	**2276**	12%	9%	78%	1%	**9769**	28%	72%	**124**	**34753**
$\mathbf{P}_6$	44%	46%	10%	**3214**	25%	26%	46%	3%	**5630**	36%	64%	**144**	**30460**
$\mathbf{P}_7$	67%	20%	13%	**65**	70%	22%	3%	5%	**59**	57%	43%	**13**	**363**
$\mathbf{P}_8$	61%	26%	12%	**102**	66%	28%	4%	3%	**97**	51%	49%	**15**	**369**
$\mathbf{P}_9$	59%	29%	11%	**1075**	60%	30%	8%	2%	**1038**	43%	57%	**27**	**592**
$\mathbf{P}_{10}$	59%	29%	12%	**1416**	58%	29%	12%	2%	**1397**	40%	60%	**27**	**812**
$\mathbf{P}_{11}$	61%	28%	12%	**209**	64%	30%	4%	2%	**195**	51%	49%	**18**	**416**
$\mathbf{P}_{12}$	61%	28%	11%	**311**	65%	30%	4%	2%	**292**	52%	48%	**20**	**437**
$\mathbf{P}_{13}$	60%	29%	11%	**384**	64%	30%	4%	2%	**359**	56%	44%	**19**	**426**
$\mathbf{P}_{14}$	60%	29%	11%	**451**	63%	30%	5%	2%	**434**	56%	44%	**22**	**465**
$\mathbf{P}_{15}$	61%	28%	11%	**533**	64%	29%	5%	2%	**517**	52%	48%	**17**	**445**
$\mathbf{P}_{16}$	61%	28%	11%	**697**	63%	29%	7%	2%	**653**	54%	46%	**20**	**504**
$\mathbf{P}_{17}$	60%	29%	11%	**805**	62%	29%	8%	2%	**780**	45%	55%	**23**	**518**

on CPLEX operations. The total running time of algorithm KEY can be divided into four parts: t_{sp}, t_θ, t_{key} and $\bar{t}$. Three of them, t_{sp}, t_θ and $\bar{t}$, are as described before. t_{key} is the time spent for key path swapping and for CPLEX to solve the relaxed RPP. Algorithm DW spent t_{sp} in computing shortest paths for generating columns with negative reduced cost. Algorithm NA simply uses CPLEX to directly solve the problem in node-arc form.

In all of the tests, DW is the fastest algorithm. For larger cases $P_1, \ldots, P_6$, NA is the slowest algorithm and usually is many times slower than the other algorithms. PD is the second fast algorithm. KEY usually spends no more than twice the time of PD, with the exception of problem P_5.

For smaller cases $P_7, \ldots, P_{17}$, KEY performs slightly better than PD. It is difficult to conclude whether KEY is faster than NA or not from our limited tests. However, it is observed that the performance of NA seems to be consistent when the topology and number of commodities are fixed. For example, we may roughly classify the 17 problems into 3 groups: $P_1, \ldots, P_4$, P_5 and P_6, and $P_7, \ldots, P_{17}$, by their number of nodes, arcs and commodities. We observe that NA performs similarly for problems in the same group. KEY and PD, on the other hand, seem to be more sensitive to what the commodities are, instead of how many commodities there are, and may perform drastically differently even on the same network with the same number of commodities.

Table 33: Comparisons on number of iterations for PD, KEY and DW

	PD	KEY		DW
	number of iterations	number of iterations	number of key changes	number of iterations
$\mathbf{P}_1$	8	8	100	7
$\mathbf{P}_2$	29	25	1770	7
$\mathbf{P}_3$	21	21	973	6
$\mathbf{P}_4$	15	15	797	7
$\mathbf{P}_5$	21	21	3102	6
$\mathbf{P}_6$	24	24	1788	6
$\mathbf{P}_7$	4	4	0	4
$\mathbf{P}_8$	6	6	4	5
$\mathbf{P}_9$	61	60	142	7
$\mathbf{P}_{10}$	79	77	391	6
$\mathbf{P}_{11}$	12	12	5	5
$\mathbf{P}_{12}$	18	18	5	6
$\mathbf{P}_{13}$	22	22	11	6
$\mathbf{P}_{14}$	26	26	26	7
$\mathbf{P}_{15}$	31	31	29	5
$\mathbf{P}_{16}$	40	39	71	6
$\mathbf{P}_{17}$	46	46	122	6

6.3.1.1 Generic primal-dual method (PD) vs. key path decomposition method (KEY)

In this section we take a closer look at algorithms PD and KEY. KEY and PD should have a similar number of primal-dual iterations since they only differ in using different techniques to solving the RPP, where KEY uses the key path decomposition method and PD uses CPLEX. Table 33 compares the number of iterations. The results show that both PD and KEY do take almost same number of primal-dual iterations. They are not identical since there exist multiple optimal dual solutions (i.e., feasible dual improving directions) for the RPP.

The reason KEY performs much worse than PD for problems $P_1,\ldots,P_6$ is that there are many iterations of key path swapping when solving the relaxed RPP. Table 32 shows that t_{key}, the time spent on swapping key paths and solving the relaxed RPP, is usually the most time-consuming component in KEY. This fact can also be confirmed by observing that both PD and KEY have about the same amount of t_{sp} and t_θ. In other words, the key path decomposition method that iteratively solves smaller relaxed LPs seems to be less efficient than the ordinary LP methods that solve a larger and more difficult LP, at least for the larger cases we have tested.

However, this does not mean that KEY is always less efficient than PD. Indeed, for example, in all problems $P_7, \ldots, P_{17}$, KEY performs slightly better than PD. In particular, the key path decomposition method is designed to be more efficient for problems with a large number of OD commodities so that the time saved by solving the relaxed RPP will outweigh the time spent in swapping key paths. Problems $P_7, \ldots, P_{17}$ have more commodities than problems $P_1, \ldots, P_6$. Thus, it is not surprising that algorithm KEY is more efficient in solving problems $P_7, \ldots, P_{17}$, than algorithm PD.

To improve the relative efficiency of algorithm KEY, techniques to shorten t_{key} are required. In this thesis, we have proposed the method of adding positive perturbations on the objective coefficients of the path flows in the relaxed RPP to avoid degenerate pivots and cycling problems, but more work is necessary.

6.3.1.2 Generating shortest paths for Dantzig-Wolfe decomposition

Because DW performs so efficient, in all of our tests, we try to find out the best of four implementations of DW: DW_{OO}, DW_{OA}, DW_{AO} and DW_{AA}: DW_{OO} generates only a single shortest path for each commodity in all iterations; DW_{OA} generates only a single shortest path for each commodity in the initial iteration, but then generates all shortest paths for each commodity in every other iterations; DW_{AO} generates all shortest paths for each commodity in the initial iteration, but then generates a single shortest path for each commodity in the latter iterations; and DW_{AA} generates all shortest paths in all iterations.

The goal here is to see whether generating all shortest paths for each commodity in some iterations will shorten the total running time. In other words, we try to see how many good columns should be generated to shorten the overall running time in the column generation scheme of DW. Intuitively, if we can generate many good columns earlier, we may achieve optimality earlier. However, generating all of the columns with negative reduced cost (i.e., all shortest paths, in our case) can be time consuming. Compromises in the number of columns generated (i.e., generating at most a specific number of columns) might be worth investigating.

Note that in our DW implementation, we initially add $|N| + |K|$ artificial variables with

Table 34: Total time (ms) of four DW implementations

	$\mathbf{DW}_{OO}$	$\mathbf{DW}_{OA}$	$\mathbf{DW}_{AO}$	$\mathbf{DW}_{AA}$
$\mathbf{P}_1$	57	92	63	96
$\mathbf{P}_2$	65	104	79	122
$\mathbf{P}_3$	58	102	73	107
$\mathbf{P}_4$	62	114	80	136
$\mathbf{P}_5$	124	294	155	308
$\mathbf{P}_6$	144	277	208	325
$\mathbf{P}_7$	13	21	22	30
$\mathbf{P}_8$	15	28	25	36
$\mathbf{P}_9$	27	47	35	55
$\mathbf{P}_{10}$	27	48	33	56
$\mathbf{P}_{11}$	18	28	24	36
$\mathbf{P}_{12}$	20	34	27	40
$\mathbf{P}_{13}$	19	33	28	43
$\mathbf{P}_{14}$	22	39	32	48
$\mathbf{P}_{15}$	17	28	25	38
$\mathbf{P}_{16}$	20	35	29	44
$\mathbf{P}_{17}$	23	39	31	48

large cost so that initial basic feasible solutions can be easily identified by CPLEX.

Table 34 compares running time for four different DW implementations. It shows that DW_{OO}, the implementation that only generates single shortest path for each commodity in all iterations, is the most efficient one, DW_{AO} is the second fastest, DW_{OA} is third, and DW_{AA} is the slowest implementation, taking more than twice the time of DW_{OO}. Because of the results, we use DW_{OO} for the comparisons in Table 32 and 33.

Table 34 shows that, at least for our limited tests, generating all shortest paths is not a good idea. Generating all shortest paths may reduce the number of total iterations, but DW_{OO} requires fewer than 10 iterations for all test problems (as shown in Table 33). Generating all shortest paths may indeed be a time-consuming operation. This may also explain why the other methods, PD and KEY, which requires all shortest paths, may be less efficient.

6.3.2 Impact of good shortest path algorithms

Three of the algorithms, especially PD and KEY, require extensive computations of shortest paths. Thus, an efficient shortest path algorithm is crucial for practical efficiency.

In particular, both PD and KEY generate all shortest paths to construct the RPP (this takes time t_{sp}). Also, they need to solve many iterations of MPSP to determine the step length θ^* (this takes time t_θ). Table 33 shows that PD spends approximately 85% of its total running time doing shortest path computations. KEY spends about the same amount of time as PD in shortest path computations. DW spends approximately 30% to 50% of its time computing shortest paths.

There are at least two ways to shorten the time spent on shortest path computations in PD and KEY: one is to find a better dual improving direction so that the total number of primal-dual iterations can be reduced, and the other is to design more-efficient shortest path algorithms.

All of the shortest path problems encountered in these algorithms are, in fact, MPSP problems. In particular, each commodity corresponds to an OD pair, and shortest paths between several OD pairs are repeatedly computed either to test whether they are dual infeasible (when determining θ^*) or to generate all shortest paths for each requested OD pair (when constructing the RPP). These MPSP problems share the same topology. All of these issues point to opportunities in the design of efficient MPSP algorithms for problems with fixed topology, which is the topic we have discussed in Chapter 2, Chapter 3, and Chapter 4 of this thesis.

6.4 Summary

In this chapter we first discussed generic primal-dual methods (PD) for solving origin-destination multicommodity network flow (ODMCNF) problems. We proposed two methods to determine the step length θ^* and choose the latter one for our implementation. We perturbed the objective coefficients of the artificial variables to resolve the degenerate pivots caused by primal degeneracy.

We then proposed a new method, the primal-dual key path method (KEY), to solve ODMCNF problems. We discussed its properties and difficulties, and proposed techniques to resolve cycling problems caused by dual degeneracy.

We compared our algorithm KEY with other three algorithms, PD, DW, and NA. We found DW, the Dantzig-Wolfe decomposition method, was the fastest algorithm in all of our tests. As expected, solving the node-arc formulation (NA) without any special techniques such as resource-directive methods or the basis-partitioning methods introduced in Chapter 2, will be time-consuming and very inefficient, except for small networks.

KEY follows the generic PD steps but uses the key path decomposition method to solve a relaxed RPP at each iteration. It is designed for problems with large number of commodities. In our limited tests, KEY does perform better than PD for cases with more commodities. However, for larger cases with few commodities, it performs much worse than PD. The reason for its inefficiency is that it requires many iterations of key path swapping operations. Although our proposed technique, which perturbs the objective coefficients for path flows, has reduced the chances of key path swapping and resolved the cycling problem caused by dual degeneracy, it is still not efficient enough for some larger cases.

More tests should be performed, especially using cases with a large number of commodities, to draw more solid conclusions on the efficiency of KEY.

We also found that generating all shortest paths for the same commodity may not save time in Dantzig-Wolfe decomposition with a column generation scheme. That is, generating a single shortest path for each commodity is good enough.

To explain why DW performs so well compared to PD and KEY, we give the following reasons:

First, DW generates fewer shortest paths than DW and KEY. In particular, DW will not generate a path twice, if we keep all of the previously generated paths in the restricted master problem. PD and KEY, on the other hand, generate a new set of all shortest paths at every iteration using the new dual solutions. Note that between two primal-dual iterations, many paths may remain the shortest and do not need to be altered. The reason that we remove the entire previous set of shortest paths and add a new set of shortest paths is that keeping old paths would require bookkeeping on each specific path, which is difficult and inefficient. In particular, to check whether a path is in the current set takes exponential time when there are (exponentially) many paths to be added or removed. Furthermore, using CPLEX to solve the RPP usually requires the sparse representation of the constraint matrix. In this case, each path corresponds to a column. Removing and adding columns will require a new sparse representation. Although we can add easily columns at the end of the matrix, identifying the columns to be removed requires sparse matrix operations. Thus, PD and KEY invoke many more operations than DW does, making the algorithms less efficient.

Degeneracy is an important issue, especially for primal-dual based methods like PD and KEY. Multiple optimal dual or primal solutions may cause implementation problems. Techniques that choose a ”good” optimal solution are desired, and may drastically shorten the total running time.

KEY has one more issue of concern than PD: the memory requirement. To implement the key path swapping operations more efficiently, we allocate memory to store the initial relaxed RPP at each primal-dual iteration. Inside a primal-dual iteration, KEY may perform many iterations of key path swapping. The column corresponding to the key path is empty, and an empty column has no sparse representation at all. Thus, it is clumsy to do the key path swapping and column operations only using the original sparse representation of the RPP. Although we have an efficient way to swap key paths and update each column for the commodities that have new key paths, it still requires much time, especially when there are many iterations of key path swapping operations. Table 33 and Table 34 show that more iterations of swapping key paths results in a longer running time compared to PD.

To make KEY more competitive, new techniques that identify better dual improving directions can be developed. Our results show that efficient MPSP algorithms play a crucial role in improving all of the path-oriented multicommodity network flow algorithms, especially primal-dual based algorithms like PD and KEY which 85% (PD) or 95% (KEY) of their total running time computing shortest paths.

CHAPTER VII

CONCLUSION AND FUTURE RESEARCH

This chapter concludes the thesis by highlighting our contributions in Section 7.1, and then suggesting some potential directions for future research in shortest path, multicommodity flow and their related problems in Section 7.2.

7.1 Contributions

This thesis contains extensive surveys of both the applications and solution methods of multicommodity network flow problems. The previous survey papers in multicommodity network flow were written more than two decades ago. During the past twenty years, many new methods and applications in the field have been proposed and researched. We survey over 200 references and summarize the applications in Chapter 1, and solution methods in Chapter 2.

The shortest path problem is a classic combinatorial optimization problem. It is considered to be easy but is very important because it appeared as a subproblem in many difficult problems. For example, solving origin-destination multicommodity network flow (ODMCNF) problems by the arc-path formulation will usually require extensive computations of shortest paths between multiple pairs of nodes.

Although the shortest path problem has been researched for more than 50 years, to the best of our knowledge, there exist no combinatorial algorithms designed specifically for solving multiple pairs shortest path problems (MPSP), until now. In Chapter 3 we survey over 100 references and summarize most of the shortest path algorithms in the literature, discuss their pros and cons, and then demonstrate that a new method called the Least Squares Primal-Dual method (LSPD), when used to solve the 1-1 and ALL-1 shortest path problems with nonnegative arc lengths, performs identical steps to the ”classic” Dijkstra's algorithm. We also discuss the relationship between LSPD, Dijkstra, and the original primal-dual algorithm in solving the 1-1 and ALL-1 shortest path problems.

Inspired by an APSP algorithm proposed by Carré [64, 65], in Chapter 4 we propose two new MPSP algorithms, called $DLU1$ and $DLU2$, which are the first shortest path algorithms designed specifically for solving multiple pairs shortest path problems. They do not need to compute a single source shortest path (SSSP) tree multiple times, nor do they need to compute all pairs shortest paths (APSP) as many algorithms in the literature do. In particular, $DLU2$ can specifically compute the shortest paths or distances for the requested OD pairs without wasting time on computing other unrequested OD entries. Our algorithms are applicable to cases with negative arc lengths but not negative cycles. Like the Floyd-Warshall algorithm, our algorithms can also detect negative cycles.

For real world problems where the network topology is fixed but arc lengths or requested OD pairs are variable, our MPSP algorithms may take more advantage of similarities than other shortest path algorithms. In particular, our algorithms rely on a good node pivot ordering, which can be computed in advance by many fill-in reducing techniques used for the sparse Gaussian method in numerical linear algebra. For problems with fixed topology, this special node ordering needs to be determined only once, to produce an "ad hoc" efficient shortest path algorithm that is designed specifically for the problem topology. This property is advantageous, for problems such as ODMCNF in which shortest paths between multiple pairs of OD nodes have to be repeatedly computed for the same network topology but different arc lengths.

We show correctness and give the theoretical complexity of our algorithms. Although theoretically both $DLU1$ and $DLU2$ have $O(n^3)$ time bounds, we expect practical efficiency because of the fill-in minimizing techniques from numerical linear algebra.

Chapter 5 gives a detailed introduction to a sparse implementation, SLU, of Carré's algorithm. We have performed many computational experiments to compare several implementations of our algorithms $DLU1$, $DLU2$, and SLU, with other 5 label-correcting and 4 label-setting SSSP shortest path codes [74] which are considered to be the state-of-the-art. We have done thorough computational tests on 19 problem families of MPSP problems, each with 4 different percentages (25%, 50%, 75%, and 100%) of OD pairs requested. This is the first computational study in the literature for solving the MPSP problems. Unfortunately, our proposed algorithms do not perform more efficiently than most of the state-of-the-art shortest path codes for the cases we tested. However, it does give evidence that our MPSP algorithms perform absolutely better than Floyd-Warshall algorithms for solving MPSP problems. Also, on a real flight network in the Asia-Pacific region which contains 112 nodes and 1038 arcs, our new MPSP algorithms do appear to be competitive, and are even faster than many of the state-of-the-art SSSP codes. Our MPSP algorithms especially take advantage of problems where the distribution of OD pairs correspond to a matching in the OD matrix. That is, our MPSP algorithms perform relatively better for MPSPs with n requested OD pairs (s_i, t_i) where each node appears exactly once in both of the origin and destination sets but not at the same time. In this case, most of the known shortest path algorithms have to compute shortest paths for all pairs but our algorithms only compute the necessary shortest paths rather than solving the APSP problem.

Finally, in Chapter 6 we propose a new primal-dual method named the "primal-dual key path method" to solve ODMCNF problems. We describe the details and give two methods to compute the step length θ^*. We also explain how primal and dual degeneracy may affect our algorithms and propose perturbation methods to resolve the degenerate pivots and cycling. In limited computational tests on 17 test problems, we compare our primal-dual key path method (KEY) with the generic primal-dual method (PD), Dantzig-Wolf decomposition method (DW), and the CPLEX LP solver that solves the node-arc (NA) formulation. We conclude DW to be the most efficient implementation and give reasons why primal-dual based algorithms (PD and KEY) may perform worse than DW. Our computational results also confirm the fact that KEY is more suitable for cases with many OD commodities. Also, we discover that generating a single shortest path, rather than all shortest paths, for each commodity is good enough for the DW method.

Further breakdown of the total running time for PD, KEY and DW reconfirms the importances of a good MPSP algorithm in speeding up those path-based algorithms. In particular, up to 85% total running time of PD, 95% total running time of KEY and 55% total running time of DW may be spent on shortest path computations. Thus designing an efficient MPSP algorithm, as we have researched in Chapter 3, 4, and 5, is very important.

7.2 Future research

In this section, we propose some interesting problems for future research.

7.2.1 Shortest path

- Although we have proposed new MPSP algorithms, in our algorithms, sequence of operations is based on the node ordering, and does not consider the affect of individual arc lengths as most SSSP algorithms do. A good research topic is integration of arc length into our algorithm. In particular, it may be possible that some triple-comparisons can be avoided, and such techniques may speed up our algorithms.

- In Chapter 5, we chose test problems based on the computational experiments on SSSP algorithms by Cherkassky et al. [74]. Although the computational results are not so encouraging, it is possible that our MPSP algorithms may perform better on some specific classes of graphs. Complete graphs, for example, are a class of networks in which our MPSP algorithms should run faster than other algorithms not only theoretically, but also computationally. Note that our algorithms may have worse performance for larger networks since they require more memory, and accessing so much memory will slow down their performance. Hub-and-spoke networks may also be a class where our algorithms are advantageous since a careful node ordering which permutes the hub node and its nearby nodes to larger indices will avoid many fill-ins.

  What classes of graphs may be suitable for our MPSP algorithms? Why do our algorithms run so quickly for the Aisa-Pacific flight network? What are the factors that slow down or speed up our algorithms? Answers to these questions may lead us to design new and better MPSP algorithms, or help us to identify appropriate applications for our MPSP algorithms.

- Theoretically, the running time of a shortest path algorithm should be proportional to the total number of triple comparisons. However, due to the computer architecture, the running time may be "nonlinearly" affected by the memory caching and accessing which makes the algebraic algorithms less efficient for larger network. Another good research is to study the relationship between the total running time and total number of triple comparisons, or, to benchmark algorithms using total number of triple comparisons, instead of using the practical running time.

- Our MPSP algorithms are inspired by the Gaussian method applied in a path algebra system. In other words, computing shortest path distances for q OD pairs (s_i, t_i) is identical to computing the q specific entries $x_{s_i t_i}$ of the matrix $X = (I_n - C)^{-1}$ where C is the measure matrix and I_n is the identity matrix (see Section 3.4).

  Using similar arguments to the numerical linear algebra, we should be able to give a "twin" algorithm similar to our MPSP algorithms which compute q specific entries for some matrix B^{-1} without inverting the whole matrix B (as does the APSP algorithm), and without computing whole columns of the matrix that cover the requested OD pairs (as does by SSSP algorithm). We even do not need to compute the entries $B^{-1}_{n,j}, \ldots, B^{-1}_{i+1,j}$ to obtain the entry $B^{-1}_{i,j}$ (as does by Carré's algorithm). It remains an open question whether efficiently computing specific entries for the inverse of a matrix is difficult in most cases.

- We have discovered the new Least Squares Primal-Dual method (LSPD), when used to solve shortest path problems with nonnegative arc lengths, actually corresponds to the Dijkstra's algorithm (see Section 3.6). What happens in the case of negative arc lengths? Will LSPD perform better than other SSSP algorithms, and in which cases?

  In our survey and computational tests, the LP-based methods such as primal simplex methods, dual simplex methods, or primal-dual simplex methods are usually considered to be practically less efficient than the combinatorial label-setting and label-correcting methods. LSPD can also be classified as a LP-based method. Although it has special properties such as being imperious to degenerate pivots, whether it will solve MPSP faster or solve SSSP faster for cases with negative arc lengths require more investigation.

7.2.2 Multicommodity network flow

- Our limited computational tests show that our primal-dual key path method (KEY) is not very competitive. However, more tests should be done, especially using the problems with many OD commodities of which KEY can take more advantage.

- Both KEY and PD are primal-dual based algorithms, and perform worse than DW. From our computational results, we think the key to speeding up KEY and PD lies in the techniques used to obtain optimal dual solutions of the RPP. In particular, there usually exist multiple optimal dual solutions of the RPP. Among these multiple choices of dual improving directions, which one might help to reduce the total number of primal-dual iterations? Which improving direction might reduce the number of degenerate pivots? If we can reduce the number of primal-dual iterations, we should save much time in shortest path computations.

  The LSPD may be a good direction to study, since it is also primal-dual based and can avoid degenerate pivots. More research can be done in applying LSPD to solve the RPP, either in the node-arc form or in the arc-path form. The reason we cite the node-arc form here is because LSPD already has had success in solving the single commodity min-cost network flow problem [149, 30]. In that case, the node-arc form of the single commodity problem helps to shorten the time spent in least squares computations. We suspect similar methods may also be available for the multicommodity cases. We will continue to investigate the possibility of applying LSPD in solving the ODMCNF problems.

- Some new methods such as the volume algorithm of Barahona and Anbil [24], as introduced in Section 2.3.2, have not yet been applied to solve ODMCNF problems. We think it is a good research direction to try this new algorithm and compare its performance with other similar algorithms such as bundle methods and subgradient methods.

- Although DW has been shown to be very efficient in our tests, other research in the airline crew partitioning and cutting-stock problems [169] shows that a primal-dual subproblem simplex method might also be competitive. Since we already develop techniques for generating all of the shortest paths for our primal-dual algorithms, it should not be difficult to generate all of the paths with length within a small threshold value of the length of shortest paths, as required for the primal-dual subproblem simplex method to proceed. However, one disadvantage of this method is the necessity of path index bookkeeping, as encountered in our PD implementation.

APPENDIX A

COMPUTATIONAL EXPERIMENTS ON SHORTEST PATH ALGORITHMS

This appendix lists computational results of shortest path algorithms on networks generated by four generators: SPGRID, SPRAND, NETGEN, and SPACYC. The first 19 tables (from Table 35 to Table 53) are cases where the requested OD pairs have distinct $\frac{1}{4}|N|$ destinations. The second 19 tables (from Table 54 to Table 72) are cases where the requested OD pairs have distinct $\frac{1}{2}|N|$ destinations. The last 19 tables (from Table 73 to Table 91) are cases where the requested OD pairs have distinct $|N|$ destinations.

Table 35: 25% SPGRID-SQ

Grid/deg	SLU 1	SLU 2	SLU 3	DLU 21	DLU 22	GOR 1	BFP	THR ESH	PA PE	TWO Q	DIK H	DIK BD	DIK R	DIK BA
10x10/3	6.67	18.33	18.33	8.33	12.00	5.00	1.33	4.00	1.00	1.33	8.67	7.00	12.67	29.67
20x20/3	6.04	14.82	14.44	4.62	6.36	5.10	1.16	2.58	1.12	1.00	7.88	4.64	9.70	13.14
30x30/3	4.96	12.46	12.19	4.08	5.18	4.46	1.12	2.67	1.04	1.00	6.75	3.30	7.24	6.33
40x40/3	5.92	15.43	15.58	10.67	13.04	4.54	1.12	2.14	1.05	1.00	7.05	3.23	7.16	5.41
50x50/3	5.80	13.83	13.13	6.03	8.23	5.17	1.10	2.07	1.03	1.00	7.40	3.17	6.83	4.63
60x60/3	6.03	14.77	14.22	6.06	6.69	5.09	1.12	1.98	1.02	1.00	7.00	3.18	6.45	3.94
70x70/3	6.78	15.98	15.38	6.45	7.21	4.46	1.12	2.12	1.03	1.00	7.30	3.01	6.54	3.71
80x80/3	8.75	19.47	18.64	9.91	10.88	5.67	1.12	2.13	1.03	1.00	7.61	3.01	6.54	3.57
90x90/3	9.89	20.11	19.36	15.16	13.78	5.21	1.25	2.03	1.05	1.00	6.81	2.72	5.76	2.95
100x100/3	13.06	23.99	22.35	11.04	11.64	4.71	1.14	2.25	1.04	1.00	7.69	3.05	6.48	3.29

Table 36: 25% SPGRID-WL

Grid/deg	SLU 1	SLU 2	SLU 3	DLU 21	DLU 22	GOR 1	BFP	THR ESH	PA PE	TWO Q	DIK H	DIK BD	DIK R	DIK BA
16x64/3	6.09	14.22	13.96	4.30	5.17	4.70	1.17	2.17	1.09	1.00	6.13	3.96	8.70	9.83
16x128/3	5.70	13.99	13.48	4.37	5.07	5.09	1.10	2.02	1.04	1.00	5.67	3.69	8.28	8.55
16x256/3	5.77	15.51	15.02	4.87	4.95	4.63	1.10	2.02	1.05	1.00	5.69	3.75	8.60	7.87
16x512/3	6.64	16.61	15.92	3.37	3.20	5.32	1.12	2.04	1.04	1.00	5.39	3.54	8.35	7.35
64x16/3	4.14	11.09	10.73	4.00	5.05	4.77	1.18	2.23	1.18	1.00	8.64	3.41	7.09	5.41
128x16/3	3.47	10.35	9.08	3.74	4.96	4.83	1.15	2.01	1.03	1.00	8.73	2.94	5.89	3.86
256x16/3	3.31	9.29	9.05	5.78	7.78	5.06	1.15	1.91	1.06	1.00	9.62	2.99	5.69	3.21
512x16/3	3.56	10.61	10.44	5.97	7.47	4.90	1.14	1.78	1.04	1.00	10.58	2.97	5.66	3.02

Table 37: 25% SPGRID-PH

Grid/deg	SLU 1	SLU 2	SLU 3	DLU 21	DLU 22	GOR 1	BFP	THR ESH	PA PE	TWO Q	DIK H	DIK BD	DIK R	DIK BA
16x32/5	6.03	12.20	12.55	10.62	15.10	2.01	1.12	1.00	2.09	1.49	3.17	6.39	4.59	2.04
32x32/6	4.17	7.67	7.80	11.48	14.47	1.22	1.26	1.01	3.83	2.13	1.79	4.09	2.40	1.00
64x32/7	6.29	10.01	10.23	16.85	20.43	1.70	3.23	2.38	11.46	5.49	1.97	3.59	2.42	1.00
128x32/8	9.87	13.84	14.23	21.42	24.91	1.96	6.87	3.29	22.28	10.00	2.00	3.42	2.39	1.00
256x32/8	14.30	17.80	18.26	24.81	28.56	2.20	14.09	3.73	26.41	12.41	1.95	3.27	2.32	1.00

Table 38: 25% SPGRID-NH

Grid/deg	SLU 1	SLU 2	SLU 3	DLU 21	DLU 22	GOR 1	BFP	THR ESH	PA PE	TWO Q	DIK H	DIK BD	DIK R	DIK BA
16x32/5	8.81	17.89	18.40	15.64	22.10	2.50	1.00	1.24	1.03	1.00	4.55	2.07	4.28	3.59
32x32/6	8.61	16.01	16.35	24.31	30.60	2.05	1.05	1.00	1.10	1.09	3.52	1.49	2.81	1.95
64x32/7	11.33	18.05	18.48	30.55	36.96	1.98	1.17	1.00	1.22	1.22	3.20	1.23	2.19	1.37
128x32/8	16.43	22.87	23.29	34.11	40.37	1.84	1.20	1.00	1.22	1.21	3.03	1.06	1.86	1.14

Table 39: 25% SPRAND-S4 and SPRAND-S16

$\|N\|$/deg	SLU 1	SLU 2	SLU 3	DLU 21	DLU 22	GOR 1	BFP	THR ESH	PA PE	TWO Q	DIK H	DIK BD	DIK R	DIK BA
128/4	8.00	14.00	12.00	10.00	12.00	3.00	1.00	2.00	1.00	1.00	5.00	4.00	8.00	9.00
256/4	18.00	27.00	28.33	22.00	26.67	4.00	1.33	1.33	1.00	1.00	6.67	6.00	9.67	4.33
512/4	20.95	28.45	28.75	27.00	31.45	3.60	1.25	1.00	1.30	1.25	4.70	3.85	6.35	2.85
1024/4	54.91	65.73	65.00	85.09	88.36	3.55	1.64	1.00	1.73	1.73	3.91	4.27	5.00	1.64
2048/4	163.80	178.74	177.65	191.89	197.60	3.09	1.86	1.00	2.02	2.02	3.02	4.03	3.54	1.05
128/16	23.00	27.00	27.50	35.00	38.50	3.50	1.00	1.00	1.50	1.50	3.00	2.00	4.00	3.50
256/16	46.17	52.58	51.17	48.08	52.33	2.75	1.17	1.00	1.17	1.17	2.33	2.25	3.08	2.00
512/16	157.28	169.79	166.79	109.04	117.11	3.17	1.43	1.00	1.57	1.57	2.51	2.53	3.02	1.53
1024/16	307.66	318.94	319.94	249.47	260.41	2.81	1.44	1.00	2.03	1.97	1.97	2.12	2.28	1.19
2048/16	574.87	592.39	575.73	453.32	463.43	2.85	1.68	1.23	3.00	2.77	1.72	1.93	1.91	1.00

Table 40: 25% SPRAND-D4 and SPRAND-D2

$\|N\|$/deg	SLU 1	SLU 2	SLU 3	DLU 21	DLU 22	GOR 1	BFP	THR ESH	PA PE	TWO Q	DIK H	DIK BD	DIK R	DIK BA
128/32	13.80	16.20	16.00	21.60	23.20	2.60	1.00	1.00	1.20	1.20	1.60	1.60	2.00	1.80
128/64	10.10	11.40	11.50	14.20	15.30	2.40	1.10	1.00	1.30	1.20	1.10	1.10	1.30	1.30
256/64	30.09	32.02	32.00	27.62	29.64	2.49	1.27	1.00	1.58	1.56	1.16	1.11	1.31	1.02
256/128	19.53	20.71	20.35	17.68	18.40	2.86	1.63	1.38	2.28	2.21	1.03	1.00	1.10	1.01
512/128	61.20	62.79	62.30	34.51	36.53	3.08	1.82	1.44	3.30	3.07	1.05	1.07	1.12	1.00
512/256	38.24	39.14	38.52	20.03	21.19	3.46	2.34	1.95	4.57	4.14	1.00	1.03	1.04	1.00
1024/256	68.65	70.66	69.41	45.40	46.22	3.87	3.13	2.61	7.90	6.74	1.00	1.10	1.06	1.04
1024/512	39.07	40.02	39.71	26.18	26.45	4.53	4.19	3.49	13.44	10.38	1.00	1.09	1.03	1.07
2048/512	46.66	47.75	47.09	31.02	31.37	4.18	4.92	3.75	16.59	12.22	1.00	1.07	1.00	1.09
2048/1024	23.78	24.73	24.27	16.31	16.60	4.90	7.31	5.24	29.81	19.41	1.01	1.06	1.00	1.14

Table 41: 25% SPRAND-LENS4

$\|N\|$/deg	$[L, U]$	SLU 1	SLU 2	SLU 3	DLU 21	DLU 22	GOR 1	BFP	THR ESH	PA PE	TWO Q	DIK H	DIK BD	DIK R	DIK BA
256/4	$[1, 1]$	4.55	7.36	7.27	5.55	7.00	2.64	4.82	4.82	8.36	9.09	2.18	1.00	1.91	1.18
	$[0, 10]$	4.64	7.00	7.09	5.27	6.45	2.18	3.09	2.73	3.09	3.82	2.09	1.00	2.00	1.18
	$[0, 10^2]$	7.43	11.86	12.00	9.86	12.14	2.43	1.43	1.00	1.86	1.71	3.14	1.86	3.57	2.43
	$[0, 10^4]$	17.67	27.67	28.33	22.67	28.33	4.00	1.00	1.33	1.33	1.33	6.67	5.67	9.67	4.00
	$[0, 10^6]$	16.67	26.33	26.00	20.67	25.67	4.33	1.33	1.33	1.00	1.00	6.33	10.00	12.67	7.67
1024/4	$[1, 1]$	29.83	35.61	35.40	41.39	44.14	4.35	34.42	23.01	64.21	55.01	2.49	1.00	2.00	3.58
	$[0, 10]$	29.24	34.80	34.53	42.12	45.01	3.65	22.52	15.50	26.70	26.43	2.26	1.00	1.92	1.06
	$[0, 10^2]$	34.22	40.58	40.18	48.76	51.74	3.62	5.01	3.34	6.87	5.07	2.64	1.62	2.61	1.00
	$[0, 10^4]$	55.66	66.42	65.50	80.10	86.15	3.66	1.78	1.00	1.79	1.79	4.01	4.19	5.08	1.59
	$[0, 10^6]$	68.49	81.94	81.08	99.50	105.95	3.59	1.19	1.10	1.02	1.00	4.74	4.92	7.41	1.81

Table 42: 25% SPRAND-LEND4

$\lvert N\rvert$/deg	$[L, U]$	SLU 1	SLU 2	SLU 3	DLU 21	DLU 22	GOR 1	BFP	THR ESH	PA PE	TWO Q	DIK H	DIK BD	DIK R	DIK BA
256/64	$[1, 1]$	19.57	20.71	20.71	18.14	19.29	3.00	8.14	7.29	36.14	23.14	1.00	1.14	1.00	1.57
	$[0, 10]$	19.43	20.71	20.43	17.86	18.71	2.57	4.14	3.57	16.86	10.71	1.00	1.14	1.14	1.14
	$[0, 10^2]$	22.67	24.33	23.50	21.00	22.33	2.17	2.00	1.67	6.50	4.50	1.00	1.17	1.17	1.17
	$[0, 10^4]$	33.25	35.00	34.50	29.75	31.75	2.75	1.50	1.00	1.75	1.75	1.25	1.25	1.50	1.25
	$[0, 10^6]$	27.20	28.80	28.00	25.00	26.20	2.40	1.00	1.00	1.40	1.60	1.00	1.40	1.20	1.00
1024/256	$[1, 1]$	49.31	50.60	50.13	32.22	32.69	5.15	38.06	37.77	341.06	226.39	1.00	1.27	1.01	18.06
	$[0, 10]$	45.75	47.08	46.00	31.41	31.76	4.35	20.48	17.04	229.44	125.74	1.00	1.28	1.01	4.41
	$[0, 10^2]$	51.96	53.09	52.10	35.77	36.03	4.07	9.12	7.39	86.20	42.96	1.00	1.28	1.02	1.53
	$[0, 10^4]$	63.82	65.89	65.39	44.12	44.82	3.81	3.12	2.60	8.38	7.00	1.00	1.11	1.06	1.07
	$[0, 10^6]$	78.79	80.67	79.93	54.62	55.69	3.74	1.96	2.08	2.88	2.84	1.06	1.19	1.15	1.00

Table 43: 25% SPRAND-PS4

$\lvert N\rvert$/deg	P	SLU 1	SLU 2	SLU 3	DLU 21	DLU 22	GOR 1	BFP	THR ESH	PA PE	TWO Q	DIK H	DIK BD	DIK R	DIK BA
256/4	0	16.67	26.67	26.67	21.67	25.67	3.33	1.00	1.33	1.33	1.00	6.67	5.67	9.67	5.00
	10^4	17.67	27.33	27.33	23.00	28.33	4.00	1.00	1.33	1.33	1.00	6.67	5.33	9.33	5.33
	10^5	17.67	27.67	27.67	22.33	27.00	4.33	1.00	2.33	1.00	1.00	14.33	6.33	10.00	7.67
	10^6	16.67	27.00	26.67	20.00	25.67	4.00	1.00	3.67	1.00	1.00	25.33	10.00	11.67	8.67
1024/4	0	57.79	68.35	68.19	78.56	82.98	3.60	1.69	1.00	1.69	1.74	4.04	4.23	5.03	1.63
	10^4	58.36	69.02	68.55	82.31	87.43	3.64	1.65	1.00	1.66	1.68	4.06	3.74	4.71	1.68
	10^5	41.16	48.86	48.59	59.55	63.47	2.74	1.26	1.00	1.31	1.33	4.67	3.33	3.46	1.60
	10^6	34.21	40.93	40.33	50.09	53.18	2.27	1.01	1.99	1.00	1.04	11.67	3.57	4.50	2.89

Table 44: 25% SPRAND-PD4

$\lvert N\rvert$/deg	P	SLU 1	SLU 2	SLU 3	DLU 21	DLU 22	GOR 1	BFP	THR ESH	PA PE	TWO Q	DIK H	DIK BD	DIK R	DIK BA
256/64	0	33.50	36.25	35.25	31.00	34.00	3.00	1.25	1.25	1.75	1.75	1.25	1.25	1.50	1.00
	10^4	22.83	24.00	23.83	20.50	22.00	1.83	1.00	1.00	1.17	1.17	1.33	1.17	1.33	1.17
	10^5	26.20	28.00	27.20	24.20	25.40	2.20	1.00	2.00	1.40	1.40	3.20	2.20	2.20	2.20
	10^6	23.00	23.83	23.83	20.83	22.17	1.67	1.00	1.67	1.17	1.17	2.83	1.67	2.00	1.50
1024/256	0	70.18	71.82	70.78	46.31	47.16	3.75	3.03	2.49	7.49	6.49	1.00	1.10	1.06	1.06
	10^4	39.21	40.51	40.38	27.70	28.17	2.34	1.92	1.68	4.96	4.31	1.05	1.04	1.03	1.00
	10^5	20.47	20.94	20.66	14.22	14.45	1.21	1.00	1.62	2.64	2.21	2.15	1.82	1.79	1.76
	10^6	20.88	21.57	21.20	14.61	14.86	1.23	1.00	2.46	2.62	2.17	3.27	2.31	2.29	2.29

Table 45: 25% NETGEN-S4 and NETGEN-S16

$\lvert N\rvert$/deg	SLU 1	SLU 2	SLU 3	DLU 21	DLU 22	GOR 1	BFP	THR ESH	PA PE	TWO Q	DIK H	DIK BD	DIK R	DIK BA
128/4	6.67	13.33	14.44	11.11	12.22	3.33	1.00	1.11	1.11	1.11	4.44	4.44	7.78	8.89
256/4	17.33	26.33	27.00	22.33	27.67	4.67	1.00	1.33	1.00	1.00	7.00	4.00	9.33	6.00
512/4	24.50	33.89	33.78	32.00	37.33	4.06	1.17	1.11	1.06	1.00	5.22	3.17	6.50	3.28
1024/4	70.78	83.78	82.44	99.78	106.44	3.89	1.22	1.11	1.00	1.00	4.78	2.44	5.33	2.11
2048/4	258.66	281.34	280.71	290.05	301.90	3.76	1.20	1.10	1.07	1.00	4.66	2.05	4.83	1.61
128/16	10.33	13.67	13.33	15.67	17.00	2.33	1.00	1.00	1.00	1.00	2.00	1.33	2.67	2.33
256/16	26.92	31.33	31.17	30.50	33.83	2.83	1.33	1.00	1.42	1.42	2.33	1.75	2.83	2.00
512/16	120.02	127.04	124.87	97.38	102.92	3.38	1.48	1.00	1.71	1.67	2.50	1.62	2.71	1.58
1024/16	275.44	290.68	293.08	236.52	242.84	3.32	1.64	1.00	1.84	1.84	2.36	1.36	2.44	1.32
2048/16	579.17	606.23	603.01	490.16	502.83	3.53	1.63	1.00	1.88	1.87	2.34	1.20	2.18	1.16

Table 46: 25% NETGEN-D4 and NETGEN-D2

$\lvert N\rvert$/deg	SLU 1	SLU 2	SLU 3	DLU 21	DLU 22	GOR 1	BFP	THR ESH	PA PE	TWO Q	DIK H	DIK BD	DIK R	DIK BA
128/32	9.00	11.00	11.00	14.50	16.00	2.17	1.17	1.00	1.17	1.17	1.50	1.17	1.67	1.67
256/64	25.95	28.10	27.69	23.69	25.50	2.79	1.48	1.48	2.02	1.98	1.21	1.00	1.31	1.17
512/128	67.90	69.42	69.05	40.13	41.88	3.62	1.96	2.15	3.02	2.92	1.20	1.00	2.32	1.04
1024/256	75.30	77.38	75.59	51.39	52.45	4.05	2.39	2.69	4.05	4.01	1.08	1.00	1.10	1.03
2048/512	51.10	52.90	51.65	35.04	35.04	4.27	2.61	3.07	3.96	3.85	1.03	1.00	1.01	1.00
128/64	6.80	8.20	8.20	10.50	11.80	2.60	1.50	1.50	1.90	1.90	1.30	1.00	1.30	1.60
256/128	22.07	23.15	23.26	19.40	20.47	3.19	1.78	1.99	2.71	2.65	1.13	1.00	1.18	1.07
512/256	39.69	41.02	40.30	23.71	24.47	3.91	2.30	2.65	3.91	3.81	1.13	1.00	1.10	1.06
1024/512	39.41	40.72	40.41	27.26	27.99	4.18	2.50	2.89	4.56	4.34	1.00	1.02	1.06	1.02

Table 47: 25% NETGEN-LENS4

$\lvert N\rvert$/deg	$[L,U]$	SLU 1	SLU 2	SLU 3	DLU 21	DLU 22	GOR 1	BFP	THR ESH	PA PE	TWO Q	DIK H	DIK BD	DIK R	DIK BA
256/4	$[0,10]$	19.67	29.33	29.33	25.33	30.67	4.00	1.00	1.33	1.00	2.33	6.33	1.67	3.67	1.33
	$[0,10^2]$	17.33	27.00	27.00	21.00	28.00	3.67	1.33	1.33	1.33	1.00	6.33	2.00	5.33	2.00
	$[0,10^4]$	18.33	28.00	28.00	23.33	29.00	3.67	1.00	1.33	1.00	1.00	6.33	4.33	9.33	7.33
	$[0,10^6]$	17.00	27.33	27.67	19.67	26.00	4.67	1.33	1.33	1.00	1.00	6.67	10.67	13.00	7.00
1024/4	$[0,10]$	72.59	84.86	83.95	95.57	102.33	3.66	1.22	1.05	1.00	1.00	5.64	1.25	2.17	1.19
	$[0,10^2]$	72.10	84.91	84.59	101.51	108.41	3.85	1.20	1.11	1.02	1.00	4.58	1.52	3.09	1.33
	$[0,10^4]$	74.57	88.84	87.92	102.36	109.20	3.82	1.21	1.11	1.01	1.00	4.63	2.43	5.30	2.08
	$[0,10^6]$	67.33	79.18	78.51	91.86	98.38	3.62	1.21	1.04	1.01	1.00	4.38	4.26	6.79	2.17

Table 48: 25% NETGEN-LEND4

$\lvert N\rvert$/deg	$[L,U]$	SLU 1	SLU 2	SLU 3	DLU 21	DLU 22	GOR 1	BFP	THR ESH	PA PE	TWO Q	DIK H	DIK BD	DIK R	DIK BA
256/64	$[0,10]$	34.33	37.00	36.00	32.00	34.00	2.67	1.33	1.33	1.33	1.33	1.33	1.33	1.33	1.00
	$[0,10^2]$	36.33	38.00	37.67	33.67	35.00	4.00	2.33	2.00	2.67	2.33	1.67	1.00	1.33	1.00
	$[0,10^4]$	25.75	28.25	27.50	24.50	26.00	3.00	1.50	1.50	2.00	2.00	1.25	1.00	1.50	1.00
	$[0,10^6]$	21.20	22.60	22.60	20.00	21.20	2.40	1.20	1.20	1.80	1.60	1.00	1.60	1.20	1.00
1024/256	$[0,10]$	77.94	80.32	79.12	53.52	54.51	2.21	1.02	1.14	1.04	1.03	1.00	1.00	1.03	1.00
	$[0,10^2]$	74.75	77.34	77.04	52.76	53.41	3.75	2.34	2.74	2.86	2.86	1.06	1.00	1.05	1.02
	$[0,10^4]$	75.77	78.45	77.10	51.38	52.61	4.04	2.44	2.62	4.04	3.89	1.09	1.00	1.09	1.02
	$[0,10^6]$	74.98	77.14	76.79	52.11	53.03	3.96	2.31	2.57	3.83	3.78	1.05	1.13	1.16	1.00

Table 49: 25% SPACYC-PS4 and SPACYC-PS16

$\lvert N\rvert$/deg	$[L,U]$	ACC	SLU 1	SLU 2	SLU 3	DLU 21	DLU 22	GOR 1	BFP	THR ESH	PA PE	TWO Q	DIK H	DIK BD	DIK R	DIK BA
128/4	$[0,10^4]$	1.11	1.11	6.67	6.67	2.22	3.33	1.11	1.11	2.22	1.11	1.11	3.33	1.00	3.33	7.78
256/4	$[0,10^4]$	1.00	2.50	10.50	10.50	4.50	9.00	2.50	3.00	5.50	3.00	2.50	5.50	2.00	4.50	6.00
512/4	$[0,10^4]$	1.00	1.29	6.50	6.64	2.57	5.64	1.14	2.29	2.00	2.00	2.14	3.29	1.00	2.57	2.29
1024/4	$[0,10^4]$	1.00	2.00	10.75	11.00	5.50	10.00	1.75	4.00	3.00	3.75	4.00	5.00	1.75	3.50	2.50
2048/4	$[0,10^4]$	1.04	1.22	7.37	7.52	4.00	6.96	1.30	3.11	2.07	2.41	2.59	3.44	1.00	2.11	1.11
128/16	$[0,10^4]$	1.00	3.33	5.56	6.67	1.11	4.44	2.22	2.22	3.33	2.22	3.33	2.22	2.22	4.44	6.67
256/16	$[0,10^4]$	1.00	1.14	3.86	4.14	2.86	4.14	1.00	2.00	1.71	2.00	2.00	2.00	1.29	1.86	2.00
512/16	$[0,10^4]$	1.00	1.52	4.63	4.89	3.81	6.04	1.19	2.56	1.89	2.63	2.67	2.22	1.15	1.78	1.74
1024/16	$[0,10^4]$	1.09	1.45	5.09	5.18	6.18	7.82	1.27	3.18	2.18	3.45	3.64	2.36	1.00	1.64	1.45
2048/16	$[0,10^4]$	1.04	1.46	5.00	5.13	6.79	8.60	1.15	3.79	2.48	4.02	4.17	2.33	1.00	1.56	1.19

Table 50: 25% SPACYC-NS4 and SPACYC-NS16

$\|N\|$/**deg**	$[L,U]$	**ACC**	**SLU 1**	**SLU 2**	**SLU 3**	**DLU 21**	**DLU 22**	**GOR 1**	**BFP**	**THR ESH**	**PA PE**	**TWO Q**	**DIK H**	**DIK BD**	**DIK R**	**DIK BA**
128/4	$[-10^4, 0]$	2.00	1.00	6.00	6.00	2.00	4.00	1.00	2.00	3.00	2.00	2.00	19.00	4.00	6.00	5.00
256/4	$[-10^4, 0]$	1.00	1.33	7.00	8.67	2.00	5.67	1.00	3.33	6.00	5.33	4.33	59.00	8.00	10.67	8.33
512/4	$[-10^4, 0]$	1.00	1.20	5.93	6.20	2.40	5.07	1.13	5.33	10.40	7.13	7.40	98.67	13.53	16.40	13.87
1024/4	$[-10^4, 0]$	1.00	1.33	7.17	7.33	3.83	6.67	1.17	13.33	23.00	16.50	21.17	238.0	30.33	36.67	30.67
2048/4	$[-10^4, 0]$	1.00	1.31	7.65	7.85	4.27	7.38	1.31	29.19	47.19	72.35	51.46	543.2	58.50	69.31	59.54
128/16	$[-10^4, 0]$	2.00	2.00	6.00	6.00	2.00	6.00	1.00	4.00	9.00	11.00	8.00	72.00	10.00	13.00	11.00
256/16	$[-10^4, 0]$	1.00	1.43	3.86	4.14	2.57	4.29	1.00	5.00	10.71	18.86	11.14	162.7	11.71	13.71	12.14
512/16	$[-10^4, 0]$	1.00	1.46	4.64	4.82	3.82	5.89	1.25	8.86	19.43	39.32	22.75	294.1	21.11	24.43	21.68
1024/16	$[-10^4, 0]$	1.00	1.64	5.18	5.36	6.73	8.09	1.18	16.82	41.36	122.5	57.45	760.6	38.64	44.45	39.73
2048/16	$[-10^4, 0]$	1.00	1.46	5.12	5.37	7.50	8.85	1.35	31.96	90.15	185.3	148.5	1518	81.35	92.13	83.60

Table 51: 25% SPACYC-PD4 and SPACYC-PD2

$\|N\|$/**deg**	$[L,U]$	**ACC**	**SLU 1**	**SLU 2**	**SLU 3**	**DLU 21**	**DLU 22**	**GOR 1**	**BFP**	**THR ESH**	**PA PE**	**TWO Q**	**DIK H**	**DIK BD**	**DIK R**	**DIK BA**
128/32	$[0, 10^4]$	2.00	2.50	5.00	5.50	5.00	6.00	1.00	3.00	2.50	3.00	3.00	2.50	2.00	2.50	4.50
256/64	$[0, 10^4]$	1.21	1.86	3.36	3.64	3.79	5.21	1.43	2.86	2.29	3.29	3.21	1.43	1.00	1.43	1.64
512/128	$[0, 10^4]$	1.52	1.95	3.33	3.47	6.01	7.51	1.48	3.52	2.67	4.45	4.58	1.25	1.00	1.16	1.22
1024/256	$[0, 10^4]$	1.73	3.74	4.88	4.96	12.44	13.05	1.66	4.22	3.12	5.95	5.93	1.12	1.00	1.07	1.04
2048/512	$[0, 10^4]$	1.77	4.05	4.81	4.88	12.07	12.35	1.66	4.31	3.26	5.99	5.89	1.06	1.00	1.04	1.04
128/64	$[0, 10^4]$	1.00	1.33	2.33	2.33	3.33	3.33	1.33	1.67	1.67	2.00	2.00	1.33	1.33	1.33	2.33
256/128	$[0, 10^4]$	1.57	1.57	2.78	2.91	3.43	4.74	1.48	3.17	2.57	3.52	3.57	1.22	1.00	1.26	1.22
512/256	$[0, 10^4]$	1.71	1.86	2.85	2.98	5.67	6.74	1.70	3.62	3.02	4.71	4.56	1.16	1.00	1.11	1.11
1024/512	$[0, 10^4]$	1.87	3.88	4.72	4.80	11.00	11.43	1.73	3.90	3.37	5.55	5.44	1.06	1.00	1.04	1.08
2048/1024	$[0, 10^4]$	1.79	2.88	3.36	3.41	8.76	8.92	1.69	4.00	3.47	5.38	5.19	1.02	1.00	1.02	1.05

Table 52: 25% SPACYC-ND4 and SPACYC-ND2

$\|N\|$/**deg**	$[L, U]$	**ACC**	**SLU 1**	**SLU 2**	**SLU 3**	**DLU 21**	**DLU 22**	**GOR 1**	**BFP**	**THR ESH**	**PA PE**	**TWO Q**	**DIK H**	**DIK BD**	**DIK R**	**DIK BA**
128/32	$[-10^4, 0]$	1.00	1.25	2.75	2.75	1.50	3.00	1.00	3.00	7.25	10.75	13.75	100.00	7.25	8.75	7.50
256/64	$[-10^4, 0]$	1.00	1.27	2.32	2.36	2.64	3.50	1.00	5.73	15.27	42.09	19.41	205.55	13.77	15.09	14.18
512/128	$[-10^4, 0]$	1.00	1.30	2.17	2.30	3.85	4.70	1.04	11.15	32.37	176.42	59.31	717.18	27.45	28.65	27.81
1024/256	$[-10^4, 0]$	1.00	2.05	2.72	2.84	7.13	6.89	1.04	21.99	65.76	730.31	178.5	1808.6	55.58	56.53	55.64
128/64	$[-10^4, 0]$	1.00	1.00	2.25	2.25	2.25	2.50	1.00	2.75	6.50	11.00	6.50	55.25	6.25	7.50	6.75
256/128	$[-10^4, 0]$	1.00	1.02	1.68	1.78	2.20	2.83	1.00	6.22	16.22	44.44	17.27	123.73	14.34	15.17	14.63
512/256	$[-10^4, 0]$	1.00	1.04	1.63	1.66	3.13	3.61	1.02	11.18	36.49	211.82	62.79	1124.2	30.46	31.06	30.53
1024/512	$[-10^4, 0]$	1.00	1.96	2.46	2.55	5.91	5.68	1.01	23.18	74.65	848.39	244.2	3195.8	63.54	64.27	63.64

Table 53: 25% SPACYC-P2N128 and SPACYC-P2N1024

$\|N\|$/**deg**	$[L, U]$ $\times 1000$	**ACC**	**SLU 1**	**SLU 2**	**SLU 3**	**DLU 21**	**DLU 22**	**GOR 1**	**BFP**	**THR ESH**	**PA PE**	**TWO Q**	**DIK H**	**DIK BD**	**DIK R**	**DIK BA**
128/16	$[0, 10]$	1.54	2.54	7.46	7.62	4.69	8.31	1.85	1.08	1.23	1.00	1.00	3.08	2.38	4.62	6.69
	$[-1, 9]$	1.35	2.00	5.71	6.06	3.59	5.94	1.35	1.00	1.35	1.06	1.06	3.12	2.12	3.12	4.88
	$[-2, 8]$	1.00	1.36	3.92	4.16	2.44	3.88	1.00	1.00	1.44	1.28	1.16	3.96	2.04	2.52	4.24
	$[-3, 7]$	1.00	1.44	3.96	4.12	2.56	4.16	1.08	1.68	2.60	2.60	2.20	11.44	3.20	4.20	5.44
	$[-4, 6]$	1.00	1.65	4.30	4.57	2.78	4.43	1.04	2.57	4.35	4.61	5.30	22.52	5.61	6.70	7.52
	$[-5, 5]$	1.00	1.54	4.08	4.33	2.75	4.38	1.12	3.17	6.29	9.92	5.79	67.38	7.50	9.17	9.33
	$[-6, 4]$	1.00	1.64	4.55	4.59	2.95	4.86	1.23	4.23	9.82	11.27	9.55	77.18	11.41	13.91	14.18
	$[-10, 0]$	1.00	1.64	4.56	4.56	2.64	4.64	1.04	5.56	12.80	17.80	12.32	105.1	14.44	17.16	15.04
1024/16	$[0, 10]$	1.78	2.33	7.67	7.89	8.33	11.33	1.78	1.11	1.00	1.00	1.00	2.89	1.89	3.44	1.89
	$[-1, 9]$	1.50	2.10	7.00	7.20	7.60	10.10	1.70	1.20	1.00	1.10	1.10	2.90	1.60	2.80	2.10
	$[-2, 8]$	1.06	1.31	4.31	4.50	4.81	6.38	1.00	1.44	1.50	1.62	1.62	5.81	1.94	2.62	2.31
	$[-3, 7]$	1.00	1.40	4.60	4.80	5.20	6.67	1.20	3.80	5.87	7.53	6.53	43.93	7.53	9.00	8.07
	$[-4, 6]$	1.00	1.83	5.75	6.17	7.25	9.25	1.50	11.42	21.75	44.42	24.00	295.9	26.00	30.42	27.50
	$[-5, 5]$	1.00	1.62	5.38	5.69	6.62	8.62	1.31	20.69	53.08	82.77	73.38	448.0	62.23	71.62	64.15
	$[-6, 4]$	1.00	1.47	4.80	4.87	5.73	7.13	1.07	29.60	74.73	113.0	109.5	608.6	86.73	100.8	88.73
	$[-10, 0]$	1.00	1.62	5.46	5.54	6.15	7.85	1.38	42.69	104.5	190.9	104.9	1488	130.7	148.9	132.5

Table 54: 50% SPGRID-SQ

Grid/deg	SLU 1	SLU 2	SLU 3	DLU 21	DLU 22	GOR 1	BFP	THR ESH	PA PE	TWO Q	DIK H	DIK BD	DIK R	DIK BA
10x10/3	4.57	12.71	12.86	6.86	8.29	4.57	1.00	3.00	1.00	1.00	6.71	5.71	9.86	26.43
20x20/3	5.49	13.74	13.43	3.93	5.38	5.05	1.18	2.60	1.11	1.00	8.04	4.50	9.66	12.13
30x30/3	5.02	12.45	12.08	3.86	4.83	4.56	1.11	2.01	1.05	1.00	6.77	3.31	7.26	6.54
40x40/3	5.88	15.49	15.63	10.82	13.36	4.45	1.12	2.17	1.05	1.00	7.16	3.26	7.21	5.26
50x50/3	5.56	13.31	12.77	5.11	7.10	4.87	1.13	2.02	1.05	1.00	7.25	3.13	6.80	4.59
60x60/3	6.04	14.75	13.97	5.34	6.28	5.19	1.12	2.26	1.04	1.00	7.07	2.95	6.41	3.87
70x70/3	6.81	16.18	15.61	5.96	6.64	4.67	1.14	2.12	1.04	1.00	7.40	3.09	6.58	3.79
80x80/3	8.77	19.48	18.63	9.68	10.67	5.34	1.14	2.15	1.04	1.00	7.39	3.08	6.46	3.45
90x90/3	9.94	20.19	19.44	13.81	15.26	5.11	1.21	2.02	1.05	1.00	6.65	2.66	5.66	2.86
100x100/3	12.66	23.08	21.83	7.93	8.96	4.81	1.14	2.21	1.02	1.00	7.76	3.03	6.50	3.19

Table 55: 50% SPGRID-WL

Grid/deg	SLU 1	SLU 2	SLU 3	DLU 21	DLU 22	GOR 1	BFP	THR ESH	PA PE	TWO Q	DIK H	DIK BD	DIK R	DIK BA
16x64/3	5.77	13.83	13.25	3.60	4.46	4.27	1.15	2.12	1.04	1.00	5.94	3.81	8.33	9.60
16x128/3	5.66	14.15	13.64	3.59	4.14	5.12	1.10	2.00	1.03	1.00	5.75	3.61	8.25	8.00
16x256/3	5.60	15.16	14.72	4.39	4.62	4.69	1.11	2.02	1.04	1.00	5.73	3.64	8.51	7.84
16x512/3	6.53	16.47	15.80	3.19	3.12	5.34	1.13	2.02	1.04	1.00	5.42	3.55	8.35	7.29
64x16/3	3.96	10.83	10.44	3.85	4.90	4.44	1.06	2.19	1.00	1.38	8.06	3.15	6.48	5.19
128x16/3	3.42	9.27	8.88	3.52	4.55	4.67	1.15	1.98	1.02	1.00	8.57	2.91	5.83	3.76
256x16/3	3.31	9.42	9.12	4.83	6.61	4.85	1.16	1.88	1.02	1.00	9.48	2.95	5.66	3.13
512x16/3	3.50	10.66	10.44	5.80	6.93	5.00	1.16	1.80	1.04	1.00	10.67	3.02	5.66	3.05

Table 56: 50% SPGRID-PH

Grid/deg	SLU 1	SLU 2	SLU 3	DLU 21	DLU 22	GOR 1	BFP	THR ESH	PA PE	TWO Q	DIK H	DIK BD	DIK R	DIK BA
16x32/5	5.62	11.82	12.07	9.59	14.01	1.82	1.13	1.00	1.92	1.47	3.15	6.28	4.57	2.11
32x32/6	4.15	7.85	7.99	11.73	14.87	1.27	1.39	1.12	4.07	2.27	1.89	4.16	2.50	1.00
64x32/7	6.26	10.01	10.22	16.82	20.42	1.58	3.02	2.18	10.71	5.11	1.96	3.73	2.42	1.00
128x32/8	9.59	13.37	13.64	19.93	23.67	1.93	6.50	3.10	20.32	9.18	1.97	3.45	2.33	1.00
256x32/8	14.45	18.33	18.54	25.43	29.56	2.16	14.12	3.76	25.42	12.22	1.95	3.35	2.31	1.00

Table 57: 50% SPGRID-NH

Grid/deg	SLU 1	SLU 2	SLU 3	DLU 21	DLU 22	GOR 1	BFP	THR ESH	PA PE	TWO Q	DIK H	DIK BD	DIK R	DIK BA
16x32/5	8.08	16.87	17.38	13.72	20.08	2.54	1.00	1.21	1.01	1.00	4.38	2.11	4.23	3.48
32x32/6	8.23	15.51	15.83	23.40	29.88	2.02	1.04	1.00	1.08	1.07	3.48	1.47	2.77	1.90
64x32/7	11.46	18.34	18.76	30.99	37.49	1.97	1.18	1.00	1.24	1.23	3.26	1.24	2.21	1.41
128x32/8	16.06	22.71	23.18	33.70	39.78	1.82	1.18	1.00	1.20	1.20	3.04	1.06	1.85	1.10

Table 58: 50% SPRAND-S4 and SPRAND-S16

$\|N\|$/deg	SLU 1	SLU 2	SLU 3	DLU 21	DLU 22	GOR 1	BFP	THR ESH	PA PE	TWO Q	DIK H	DIK BD	DIK R	DIK BA
128/4	15.00	25.00	23.00	15.00	18.00	5.00	2.00	3.00	2.00	1.00	9.00	8.00	14.00	18.00
256/4	13.83	23.33	24.00	15.00	19.33	4.17	1.00	1.50	1.17	1.17	6.83	5.67	9.67	5.17
512/4	17.78	26.29	26.66	24.02	28.83	3.63	1.24	1.00	1.17	1.24	4.61	5.07	6.27	2.32
1024/4	39.50	49.64	50.27	67.59	73.41	3.55	1.64	1.00	1.64	1.64	3.95	4.23	4.95	1.59
2048/4	124.34	138.07	138.06	160.98	165.03	3.04	1.85	1.00	2.10	2.04	2.96	3.89	3.43	1.00
128/16	12.80	16.80	17.20	16.80	20.60	2.40	1.00	1.00	1.00	1.00	2.20	2.20	3.20	3.20
256/16	29.00	34.83	34.57	30.61	34.96	3.04	1.30	1.00	1.35	1.30	2.57	2.30	3.22	2.04
512/16	99.02	109.41	108.14	73.37	80.96	3.19	1.41	1.00	1.62	1.62	2.48	2.51	3.04	1.61
1024/16	203.76	219.13	216.26	181.89	190.94	2.89	1.44	1.00	2.05	1.97	2.02	2.16	2.94	1.19
2048/16	362.50	386.97	380.05	317.39	325.64	2.89	1.74	1.26	3.13	2.89	1.79	1.97	1.96	1.00

Table 59: 50% SPRAND-D4 and SPRAND-D2

$\|N\|$/deg	SLU 1	SLU 2	SLU 3	DLU 21	DLU 22	GOR 1	BFP	THR ESH	PA PE	TWO Q	DIK H	DIK BD	DIK R	DIK BA
128/32	9.00	11.50	11.30	12.80	14.20	2.60	1.10	1.00	1.10	1.20	1.70	1.40	2.10	2.10
128/64	5.81	7.00	7.00	7.43	8.38	2.24	1.14	1.00	1.24	1.29	1.14	1.10	1.33	1.24
256/64	16.71	18.71	18.34	15.17	16.80	2.43	1.24	1.00	1.53	1.49	1.12	1.06	1.26	1.04
256/128	11.09	12.11	11.85	10.15	10.96	2.82	1.56	1.32	2.17	2.10	1.00	1.01	1.10	1.02
512/128	35.47	37.78	37.23	19.66	21.62	3.01	1.88	1.53	3.30	3.08	1.03	1.07	1.11	1.00
512/256	21.14	22.04	21.69	11.88	12.66	3.59	2.27	1.95	4.50	4.14	1.01	1.03	1.04	1.00
1024/256	37.85	39.98	39.68	27.80	28.63	3.72	3.10	2.56	8.02	6.70	1.00	1.09	1.05	1.04
1024/512	22.24	23.27	22.95	16.47	16.98	4.23	3.99	3.36	12.32	9.70	1.00	1.08	1.02	1.06
2048/512	26.71	27.56	26.55	18.91	19.42	4.10	4.84	3.70	16.60	12.28	1.01	1.08	1.00	1.09
2048/1024	13.77	14.39	14.19	10.03	10.24	4.95	7.34	5.32	30.17	19.73	1.01	1.06	1.00	1.15

Table 60: 50% SPRAND-LENS4

$\|N\|$/deg	$[L,U]$	SLU 1	SLU 2	SLU 3	DLU 21	DLU 22	GOR 1	BFP	THR ESH	PA PE	TWO Q	DIK H	DIK BD	DIK R	DIK BA
256/4	$[1,1]$	4.23	6.82	6.86	4.86	6.14	2.45	5.18	4.27	7.86	6.45	2.14	1.00	1.91	1.59
	$[0,10]$	3.38	5.58	5.71	3.83	4.92	1.96	2.71	2.62	2.88	3.33	1.88	1.00	1.88	1.46
	$[0,10^2]$	5.86	10.00	10.07	6.43	8.07	2.57	1.50	1.00	1.79	1.64	3.14	2.00	3.50	2.29
	$[0,10^4]$	12.29	21.00	21.14	14.00	18.14	3.57	1.00	1.14	1.00	1.00	5.57	4.71	8.43	4.57
	$[0,10^6]$	15.33	25.83	26.00	17.17	22.50	4.67	1.17	1.33	1.17	1.00	6.67	9.00	12.67	7.50
1024/4	$[1,1]$	22.63	28.59	28.60	35.98	38.99	4.63	31.30	20.38	53.50	49.86	2.56	1.00	2.03	3.14
	$[0,10]$	21.50	27.14	27.03	33.66	36.20	3.65	21.71	14.45	30.37	26.44	2.28	1.00	1.94	1.04
	$[0,10^2]$	24.58	31.10	31.04	38.64	41.61	3.48	5.08	3.68	7.45	5.86	2.63	1.66	2.58	1.00
	$[0,10^4]$	41.20	52.79	52.34	68.88	73.36	3.74	1.67	1.00	1.65	1.69	4.10	4.28	5.20	1.68
	$[0,10^6]$	51.06	64.07	63.83	79.02	85.76	3.61	1.18	1.09	1.00	1.01	4.68	4.85	7.28	1.82

Table 61: 50% SPRAND-LEND4

$\|N\|$/deg	$[L,U]$	SLU 1	SLU 2	SLU 3	DLU 21	DLU 22	GOR 1	BFP	THR ESH	PA PE	TWO Q	DIK H	DIK BD	DIK R	DIK BA
256/64	$[1,1]$	11.85	13.23	13.08	11.00	12.00	3.31	8.38	7.62	38.69	24.62	1.00	1.15	1.15	1.77
	$[0,10]$	12.00	13.38	13.31	11.15	12.08	2.77	4.54	3.77	20.38	11.92	1.00	1.08	1.08	1.38
	$[0,10^2]$	11.85	13.46	13.38	10.77	12.00	2.15	2.00	1.77	5.69	4.15	1.00	1.00	1.00	1.23
	$[0,10^4]$	16.78	18.44	18.56	15.67	17.11	2.56	1.22	1.00	1.56	1.44	1.22	1.11	1.33	1.11
	$[0,10^6]$	16.89	18.67	18.44	15.44	17.22	2.67	1.22	1.22	1.56	1.56	1.22	1.78	1.44	1.00
1024/256	$[1,1]$	27.50	29.18	28.84	20.23	20.76	5.25	39.58	38.77	348.38	232.20	1.00	1.29	1.01	18.44
	$[0,10]$	27.09	28.22	27.90	20.07	20.59	4.29	20.55	17.05	220.60	120.29	1.00	1.32	1.02	4.42
	$[0,10^2]$	29.47	31.34	31.02	21.82	22.33	3.86	8.61	6.96	78.44	40.10	1.01	1.23	1.00	1.44
	$[0,10^4]$	38.12	40.27	40.00	28.71	29.39	3.80	3.33	2.70	8.42	7.00	1.00	1.08	1.04	1.03
	$[0,10^6]$	44.04	45.64	45.08	32.51	33.32	3.65	1.87	2.02	2.74	2.71	1.05	1.18	1.12	1.00

Table 62: 50% SPRAND-PS4

$\|N\|$/deg	P	SLU 1	SLU 2	SLU 3	DLU 21	DLU 22	GOR 1	BFP	THR ESH	PA PE	TWO Q	DIK H	DIK BD	DIK R	DIK BA
256/4	0	15.00	24.50	25.33	17.83	22.17	4.67	1.17	1.33	1.17	1.00	6.67	5.67	9.83	4.67
	10^4	13.83	23.33	23.67	15.17	19.33	4.50	1.17	1.50	1.17	1.00	7.00	5.00	9.33	5.67
	10^5	14.83	24.17	24.67	15.83	21.00	4.50	1.17	2.33	1.17	1.00	14.83	5.83	9.83	7.33
	10^6	15.33	26.17	26.50	17.33	21.67	4.50	1.17	3.33	1.17	1.00	24.00	8.83	11.50	7.33
1024/4	0	42.78	53.33	53.26	67.20	72.35	3.65	1.68	1.00	1.70	1.70	4.04	4.20	5.06	1.66
	10^4	43.63	55.77	55.08	68.68	73.85	3.57	1.66	1.00	1.68	1.70	4.16	3.96	4.86	1.67
	10^5	31.77	40.37	39.85	49.80	52.95	2.64	1.19	1.00	1.21	1.22	4.64	3.33	3.45	1.55
	10^6	25.57	32.11	32.06	39.88	42.61	2.18	1.00	2.03	1.01	1.03	12.07	3.60	4.50	2.90

Table 63: 50% SPRAND-PD4

$\|N\|$/deg	P	SLU 1	SLU 2	SLU 3	DLU 21	DLU 22	GOR 1	BFP	THR ESH	PA PE	TWO Q	DIK H	DIK BD	DIK R	DIK BA
256/64	0	16.89	19.11	18.67	15.67	17.22	2.44	1.22	1.00	1.56	1.56	1.22	1.11	1.22	1.22
	10^4	14.36	15.64	15.55	12.73	14.27	2.00	1.09	1.00	1.27	1.27	1.45	1.09	1.36	1.18
	10^5	14.00	15.36	15.18	12.64	13.73	2.09	1.00	1.73	1.27	1.27	2.73	1.73	2.00	1.91
	10^6	13.73	15.45	15.27	12.73	13.82	2.00	1.00	1.82	1.27	1.18	3.18	2.00	2.09	1.91
1024/256	0	38.25	40.38	39.75	28.20	29.00	3.75	3.01	2.49	7.88	6.56	1.00	1.09	1.04	1.05
	10^4	23.37	24.45	24.26	17.18	17.67	2.27	1.83	1.61	4.80	4.01	1.05	1.04	1.02	1.00
	10^5	11.45	12.02	11.91	8.66	8.78	1.13	1.00	1.56	2.53	2.11	2.00	1.67	1.66	1.63
	10^6	12.12	12.66	12.50	9.10	9.34	1.21	1.00	2.14	2.52	2.12	2.84	2.00	2.02	1.99

Table 64: 50% NETGEN-S4 and NETGEN-S16

$\|N\|$/deg	SLU 1	SLU 2	SLU 3	DLU 21	DLU 22	GOR 1	BFP	THR ESH	PA PE	TWO Q	DIK H	DIK BD	DIK R	DIK BA
128/4	13.00	21.00	24.00	13.00	14.00	6.00	2.00	3.00	2.00	1.00	8.00	7.00	14.00	14.00
256/4	14.50	24.00	25.00	16.67	21.33	4.67	1.00	1.33	1.17	1.00	6.33	4.33	9.17	7.17
512/4	18.14	27.19	27.16	23.84	29.14	3.68	1.14	1.11	1.00	1.00	5.03	2.97	6.30	3.27
1024/4	48.32	60.74	59.32	77.47	81.26	3.79	1.16	1.05	1.00	1.00	4.47	2.37	5.11	1.79
2048/4	166.42	194.21	192.00	214.20	221.45	3.79	1.25	1.07	1.00	1.01	4.58	2.00	4.71	1.52
128/16	9.60	13.20	13.80	13.00	15.40	2.60	1.00	1.00	1.20	1.20	2.40	2.00	3.20	3.00
256/16	18.87	23.87	23.61	20.91	24.52	3.09	1.39	1.00	1.39	1.43	2.43	1.83	2.91	2.17
512/16	82.71	93.73	90.68	67.17	73.68	3.47	1.53	1.00	1.71	1.71	2.58	1.66	2.82	1.64
1024/16	193.24	206.50	205.64	180.90	188.74	3.34	1.58	1.00	1.78	1.80	2.32	1.34	2.36	1.32
2048/16	363.44	386.25	378.62	337.20	348.86	3.58	1.65	1.00	1.95	1.95	2.42	1.24	2.24	1.17

Table 65: 50% NETGEN-D4 and NETGEN-D2

$\lvert N\rvert$/deg	SLU 1	SLU 2	SLU 3	DLU 21	DLU 22	GOR 1	BFP	THR ESH	PA PE	TWO Q	DIK H	DIK BD	DIK R	DIK BA
128/32	6.08	8.08	8.08	7.67	8.92	2.17	1.08	1.00	1.33	1.33	1.42	1.25	1.67	1.75
256/64	15.66	17.80	17.49	14.47	16.18	2.75	1.48	1.47	1.94	1.92	1.24	1.00	1.31	1.16
512/128	40.84	43.27	42.84	23.86	25.58	3.59	1.96	2.17	3.11	3.03	1.20	1.00	1.19	1.06
1024/256	44.24	46.31	45.87	32.54	33.45	4.06	2.33	2.61	3.98	3.85	1.07	1.00	1.08	1.03
2048/512	30.73	31.95	31.62	22.52	23.08	4.32	2.66	3.05	4.04	3.93	1.04	1.00	1.04	1.01
128/64	4.43	5.62	5.62	5.76	6.62	2.38	1.24	1.29	1.57	1.48	1.10	1.00	1.29	1.29
256/128	12.37	13.65	13.54	10.80	11.87	3.20	1.76	1.93	2.59	2.50	1.12	1.00	1.18	1.07
512/256	23.10	24.45	24.21	13.51	14.36	3.94	2.29	2.61	3.86	3.68	1.12	1.00	1.10	1.05
1024/512	24.25	25.24	25.16	17.66	17.78	4.27	2.61	3.04	4.67	4.48	1.00	1.02	1.04	1.04

Table 66: 50% NETGEN-LENS4

$\lvert N\rvert$/deg	$[L,U]$	SLU 1	SLU 2	SLU 3	DLU 21	DLU 22	GOR 1	BFP	THR ESH	PA PE	TWO Q	DIK H	DIK BD	DIK R	DIK BA
256/4	$[0,10]$	13.14	22.14	22.29	14.29	19.00	4.00	1.14	1.29	1.00	1.00	5.71	1.43	3.14	1.43
	$[0,10^2]$	14.83	24.50	24.83	16.83	21.67	4.17	1.17	1.50	1.00	1.00	6.83	2.33	5.50	1.83
	$[0,10^4]$	14.83	25.00	25.33	17.50	22.50	4.33	1.00	1.33	1.00	1.00	6.67	4.33	9.33	6.67
	$[0,10^6]$	14.83	25.00	24.50	15.83	21.00	3.83	1.17	1.50	1.17	1.00	6.67	10.67	12.50	8.33
1024/4	$[0,10]$	54.87	67.82	67.49	82.24	87.44	3.69	1.19	1.06	1.01	1.00	4.38	1.27	2.19	1.20
	$[0,10^2]$	67.37	80.91	80.89	102.75	109.62	3.74	1.18	1.06	1.00	1.01	4.52	1.50	3.03	1.27
	$[0,10^4]$	52.83	65.92	65.77	83.31	88.98	3.73	1.20	1.09	1.00	1.01	4.60	2.45	5.30	2.00
	$[0,10^6]$	58.95	72.70	71.68	86.54	92.75	3.75	1.19	1.09	1.00	1.01	4.60	4.56	7.12	2.28

Table 67: 50% NETGEN-LEND4

$\lvert N\rvert$/deg	$[L,U]$	SLU 1	SLU 2	SLU 3	DLU 21	DLU 22	GOR 1	BFP	THR ESH	PA PE	TWO Q	DIK H	DIK BD	DIK R	DIK BA
256/64	$[0,10]$	16.86	19.14	19.00	16.14	18.14	2.29	1.14	1.14	1.14	1.00	1.29	1.00	1.14	1.00
	$[0,10^2]$	18.00	20.57	20.29	17.00	19.14	3.29	1.71	1.71	2.14	2.14	1.43	1.00	1.29	1.14
	$[0,10^4]$	15.12	17.12	16.88	14.25	15.62	3.00	1.50	1.50	1.88	1.88	1.25	1.00	1.38	1.25
	$[0,10^6]$	11.00	12.64	12.45	10.36	11.64	2.18	1.18	1.18	1.55	1.55	1.00	1.36	1.18	1.00
1024/256	$[0,10]$	46.82	49.30	48.85	34.36	35.22	2.21	1.02	1.12	1.03	1.04	1.01	1.00	1.03	1.01
	$[0,10^2]$	45.03	47.16	46.79	33.38	34.25	3.77	2.36	2.72	2.85	2.78	1.06	1.00	1.04	1.01
	$[0,10^4]$	44.76	46.89	46.29	32.82	33.87	4.10	2.37	2.69	4.03	3.91	1.09	1.00	1.10	1.02
	$[0,10^6]$	41.81	43.94	43.57	31.92	32.64	3.81	2.29	2.56	3.83	3.76	1.02	1.08	1.12	1.00

Table 68: 50% SPACYC-PS4 and SPACYC-PS16

$\lvert N\rvert$/deg	$[L,U]$	ACC	SLU 1	SLU 2	SLU 3	DLU 21	DLU 22	GOR 1	BFP	THR ESH	PA PE	TWO Q	DIK H	DIK BD	DIK R	DIK BA
128/4	$[0,10^4]$	1.00	1.00	10.00	9.00	3.00	8.00	2.00	2.00	3.00	2.00	2.00	5.00	3.00	5.00	10.00
256/4	$[0,10^4]$	1.00	1.29	5.71	6.29	1.71	5.00	1.43	1.57	1.71	1.14	1.43	2.71	1.14	2.71	3.57
512/4	$[0,10^4]$	1.00	1.29	6.82	7.04	2.86	5.93	1.00	2.89	2.39	2.25	2.64	3.46	1.29	2.75	2.29
1024/4	$[0,10^4]$	1.00	1.45	7.91	8.09	4.09	11.55	1.36	3.09	2.36	2.73	3.00	3.64	1.18	2.64	1.64
2048/4	$[0,10^4]$	1.00	1.27	7.75	8.55	4.22	7.47	1.27	3.43	2.35	2.76	3.02	3.59	1.00	2.22	1.12
128/16	$[0,10^4]$	1.00	1.25	3.25	3.50	2.50	3.25	1.00	1.50	1.25	1.50	1.50	1.75	1.00	1.50	3.00
256/16	$[0,10^4]$	1.00	1.55	4.91	5.09	3.18	5.27	1.18	2.27	2.00	2.45	2.45	2.45	1.45	2.18	3.09
512/16	$[0,10^4]$	1.00	1.53	4.73	4.89	3.84	6.13	1.07	2.82	2.22	2.82	3.04	2.25	1.07	1.84	1.73
1024/16	$[0,10^4]$	1.00	1.64	5.36	5.50	6.50	8.36	1.41	3.36	2.32	3.68	3.64	2.36	1.18	1.82	1.36
2048/16	$[0,10^4]$	1.07	1.44	5.02	5.21	6.94	8.65	1.16	3.79	2.45	4.08	4.18	2.37	1.00	1.56	1.17

Table 69: 50% SPACYC-NS4 and SPACYC-NS16

$\lvert N\rvert$/deg	$[L,U]$	ACC	SLU 1	SLU 2	SLU 3	DLU 21	DLU 22	GOR 1	BFP	THR ESH	PA PE	TWO Q	DIK H	DIK BD	DIK R	DIK BA
128/4	$[-10^4,0]$	1.00	2.22	8.89	11.11	3.33	6.67	2.22	3.33	6.67	4.44	3.33	33.33	10.00	12.22	8.89
256/4	$[-10^4,0]$	1.00	1.60	8.40	8.80	3.00	6.00	1.00	3.80	7.40	4.60	4.80	54.00	10.40	13.60	10.80
512/4	$[-10^4,0]$	1.00	1.37	7.04	7.22	2.96	6.15	1.04	7.07	13.81	12.52	11.59	129.7	18.07	21.93	18.44
1024/4	$[-10^4,0]$	1.00	1.50	8.60	8.90	4.70	8.20	1.20	12.70	22.80	21.70	18.70	248.5	28.60	34.30	29.10
2048/4	$[-10^4,0]$	1.00	1.33	8.29	8.50	4.65	8.00	1.21	24.81	39.08	52.46	32.71	648.2	50.58	60.17	51.44
128/16	$[-10^4,0]$	1.00	2.50	6.50	7.00	3.50	7.50	1.00	5.00	10.00	15.00	10.50	89.50	11.00	13.50	12.00
256/16	$[-10^4,0]$	1.00	2.00	6.00	6.44	4.11	6.44	1.67	7.33	15.89	19.22	17.56	280.8	16.78	20.00	17.56
512/16	$[-10^4,0]$	1.00	1.35	4.26	4.39	3.65	5.52	1.15	8.87	21.35	29.08	24.65	331.4	22.32	25.71	23.03
1024/16	$[-10^4,0]$	1.00	1.46	5.04	5.21	6.50	7.88	1.33	17.75	44.67	102.7	68.04	669.0	43.54	49.38	44.33
2048/16	$[-10^4,0]$	1.00	1.37	4.83	4.97	6.96	8.20	1.21	32.22	88.21	225.3	155.9	1182	74.09	83.46	75.59

Table 70: 50% SPACYC-PD4 and SPACYC-PD2

$\lvert N\rvert$/deg	$[L,U]$	ACC	SLU 1	SLU 2	SLU 3	DLU 21	DLU 22	GOR 1	BFP	THR ESH	PA PE	TWO Q	DIK H	DIK BD	DIK R	DIK BA
128/32	$[0,10^4]$	1.00	1.50	3.75	4.00	3.00	4.75	1.25	2.00	2.00	2.00	2.25	2.00	1.50	2.00	3.75
256/64	$[0,10^4]$	1.13	1.61	3.00	3.19	3.26	4.74	1.29	2.68	2.10	2.87	3.00	1.35	1.00	1.26	1.55
512/128	$[0,10^4]$	1.51	1.97	3.31	3.44	6.07	7.66	1.52	3.59	2.67	4.58	4.66	1.26	1.00	1.15	1.19
128/64	$[0,10^4]$	1.00	1.12	2.62	2.75	2.12	3.62	1.12	2.00	1.75	2.12	2.12	1.12	1.00	1.25	2.12
256/128	$[0,10^4]$	1.43	1.55	2.62	2.81	3.32	4.68	1.43	3.04	2.62	3.51	3.40	1.21	1.00	1.17	1.26
512/256	$[0,10^4]$	1.75	1.83	2.83	2.94	5.68	6.79	1.67	3.65	3.10	4.70	4.60	1.16	1.00	1.10	1.13
1024/256	$[0,10^4]$	1.74	3.83	4.98	5.09	12.52	13.13	1.65	4.12	3.15	6.06	5.84	1.16	1.00	1.10	1.07
2048/512	$[0,10^4]$	1.75	3.98	4.71	4.78	11.91	12.24	1.65	4.31	3.25	6.08	5.95	1.07	1.00	1.03	1.05
1024/512	$[0,10^4]$	1.88	3.90	4.71	4.79	11.17	11.58	1.74	3.98	3.35	5.55	5.38	1.06	1.00	1.06	1.07
2048/1024	$[0,10^4]$	1.79	2.87	3.37	3.42	8.72	8.88	1.69	4.04	3.47	5.33	5.16	1.02	1.00	1.02	1.05

Table 71: 50% SPACYC-ND4 and SPACYC-ND2

$\|N\|$/deg	$[L,U]$	ACC	SLU 1	SLU 2	SLU 3	DLU 21	DLU 22	GOR 1	BFP	THR ESH	PA PE	TWO Q	DIK H	DIK BD	DIK R	DIK BA
128/32	$[-10^4,0]$	1.00	1.75	3.75	4.25	3.00	4.50	1.25	4.25	10.50	23.25	10.50	202.0	10.25	11.75	10.75
256/64	$[-10^4,0]$	1.00	1.24	2.34	2.51	2.71	3.59	1.07	5.85	15.15	38.02	17.10	207.5	13.02	14.27	13.39
512/128	$[-10^4,0]$	1.00	1.25	2.12	2.19	3.90	4.70	1.04	11.64	35.15	263.7	64.63	785.4	29.22	30.40	29.56
128/64	$[-10^4,0]$	1.00	1.00	1.91	2.00	2.00	2.45	1.09	3.09	7.91	19.00	8.18	122.5	6.91	7.73	7.18
256/128	$[-10^4,0]$	1.00	1.05	1.68	1.84	2.33	2.93	1.03	6.07	17.13	52.51	20.59	215.7	14.99	15.93	15.39
512/256	$[-10^4,0]$	1.02	1.00	1.57	1.63	3.11	3.58	1.06	12.18	38.93	348.5	81.39	1644	33.30	33.87	33.37
1024/256	$[-10^4,0]$	1.00	2.00	2.70	2.80	7.08	6.84	1.03	21.09	64.89	744.7	190.8	2611	53.89	55.09	54.07

Table 72: 50% SPACYC-P2N128 and SPACYC-P2N1024

$\|N\|$/deg	$[L,U]$ ×1000	ACC	SLU 1	SLU 2	SLU 3	DLU 21	DLU 22	GOR 1	BFP	THR ESH	PA PE	TWO Q	DIK H	DIK BD	DIK R	DIK BA
128/16	$[0,10]$	1.70	2.80	7.60	7.95	4.70	7.75	1.95	1.05	1.45	1.00	1.00	3.10	2.75	4.95	8.00
	$[-1,9]$	1.58	2.38	6.42	6.67	3.67	6.38	1.88	1.08	1.38	1.12	1.00	3.08	2.12	3.62	6.38
	$[-2,8]$	1.00	1.61	4.33	4.44	2.67	4.31	1.11	1.00	1.39	1.06	1.03	3.36	1.97	2.78	5.53
	$[-3,7]$	1.00	1.41	3.73	3.93	2.37	3.95	1.15	1.22	1.71	1.54	1.41	5.49	2.44	3.10	5.17
	$[-4,6]$	1.00	1.51	3.97	4.21	2.38	3.90	1.18	2.23	3.79	3.59	3.05	17.87	4.79	5.77	7.28
	$[-5,5]$	1.00	1.54	4.19	4.43	2.65	4.35	1.19	3.03	5.51	5.51	4.38	36.65	6.62	8.30	9.38
	$[-6,4]$	1.00	1.66	4.43	4.69	2.80	4.49	1.20	4.37	8.63	9.71	7.54	45.60	10.26	12.49	13.37
	$[-10,0]$	1.00	1.53	4.13	4.26	2.61	4.03	1.18	5.71	12.03	15.18	10.76	87.18	14.32	17.37	16.37
1024/16	$[0,10]$	1.72	2.33	7.89	8.00	8.72	11.67	1.56	1.06	1.06	1.00	1.06	3.00	1.89	3.56	2.28
	$[-1,9]$	1.36	1.95	6.45	6.68	7.09	9.41	1.45	1.14	1.00	1.14	1.14	2.82	1.45	2.59	1.77
	$[-2,8]$	1.00	1.34	4.44	4.59	4.97	6.47	1.06	1.53	1.66	1.81	1.72	6.59	2.16	2.91	2.69
	$[-3,7]$	1.00	1.57	5.14	5.25	5.79	7.61	1.18	4.18	6.68	8.21	6.64	41.39	8.07	10.07	8.75
	$[-4,6]$	1.12	1.77	5.69	5.73	6.46	8.04	1.00	10.92	23.08	32.73	23.19	161.7	28.15	31.92	28.62
	$[-5,5]$	1.00	1.67	5.33	5.59	6.52	8.26	1.22	22.70	48.74	72.70	55.59	419.4	57.96	66.48	60.07
	$[-6,4]$	1.00	1.50	4.77	5.00	5.73	7.40	1.17	27.53	66.10	110.3	84.73	617.2	76.73	87.87	79.80
	$[-10,0]$	1.00	1.56	5.33	5.52	6.19	7.78	1.22	43.04	106.8	211.1	121.3	2267	129.2	147.8	132.2

Table 73: 100% SPGRID-SQ

Grid/deg	SLU 1	SLU 2	SLU 3	DLU 21	DLU 22	GOR 1	BFP	THR ESH	PA PE	TWO Q	DIK H	DIK BD	DIK R	DIK BA
10x10/3	5.67	15.92	16.00	5.83	10.00	4.50	1.42	3.67	1.17	1.00	8.25	6.58	11.83	31.75
20x20/3	5.55	13.97	13.71	3.80	5.24	4.95	1.18	2.56	1.06	1.00	8.13	4.54	9.63	12.69
30x30/3	4.94	12.51	12.21	3.74	4.73	4.56	1.13	2.02	1.05	1.00	6.83	3.30	7.21	6.61
40x40/3	5.87	16.23	15.63	10.67	13.25	4.58	1.11	2.15	1.05	1.00	7.25	3.27	7.20	5.32
50x50/3	5.65	13.43	12.82	4.45	6.02	5.15	1.14	2.02	1.03	1.00	7.10	3.12	6.83	4.54
60x60/3	5.99	14.65	14.10	4.86	5.58	4.97	1.12	1.94	1.03	1.00	6.90	2.90	6.35	3.80
70x70/3	6.73	16.09	15.43	5.50	6.14	4.81	1.14	2.12	1.04	1.00	7.38	3.03	6.61	3.72
80x80/3	8.81	19.94	18.51	9.44	10.44	5.53	1.15	2.10	1.04	1.00	7.52	2.96	6.57	3.49
90x90/3	9.48	19.46	18.55	14.10	14.64	5.07	1.20	1.97	1.02	1.00	6.58	2.59	5.61	2.87
100x100/3	12.08	22.39	21.10	8.71	9.01	4.72	1.14	2.26	1.04	1.00	7.79	3.05	6.54	3.24

Table 74: 100% SPGRID-WL

Grid/deg	SLU 1	SLU 2	SLU 3	DLU 21	DLU 22	GOR 1	BFP	THR ESH	PA PE	TWO Q	DIK H	DIK BD	DIK R	DIK BA
16x64/3	5.56	13.58	13.07	3.42	4.20	4.40	1.12	2.05	1.02	1.00	6.01	3.76	8.34	9.33
16x128/3	5.62	14.08	14.05	3.35	3.80	5.26	1.11	1.98	1.16	1.00	5.72	3.63	8.25	8.04
16x256/3	5.67	15.47	15.04	4.36	4.59	4.89	1.11	2.05	1.04	1.00	5.78	3.72	8.60	7.96
16x512/3	7.00	17.12	16.48	3.68	3.43	5.36	1.12	2.01	1.04	1.00	5.50	3.58	8.39	7.24
64x16/3	3.95	10.42	10.25	3.36	4.37	6.09	1.14	2.25	1.04	1.00	8.60	3.24	6.71	5.48
128x16/3	3.38	9.29	8.88	3.22	4.08	4.76	1.15	2.00	1.03	1.00	8.51	2.92	5.79	3.70
256x16/3	3.39	9.70	9.36	4.46	5.87	4.97	1.16	1.92	1.04	1.00	9.73	3.01	5.75	3.28
512x16/3	3.35	10.02	9.76	4.49	5.60	5.03	1.16	1.78	1.03	1.00	10.99	3.08	5.68	3.08

Table 75: 100% SPGRID-PH

Grid/deg	SLU 1	SLU 2	SLU 3	DLU 21	DLU 22	GOR 1	BFP	THR ESH	PA PE	TWO Q	DIK H	DIK BD	DIK R	DIK BA
16x32/5	5.52	11.76	12.04	9.27	13.76	1.93	1.15	1.00	1.95	1.50	3.19	6.40	4.58	2.02
32x32/6	3.97	7.50	7.69	11.37	14.46	1.39	1.44	1.19	4.20	2.46	1.84	4.06	2.44	1.00
64x32/7	6.03	9.72	9.95	16.31	19.86	1.54	2.98	2.15	11.14	5.12	1.97	3.65	2.41	1.00
128x32/8	9.54	13.27	13.56	19.90	23.48	1.79	6.49	3.03	21.13	9.76	1.95	3.46	2.36	1.00
256x32/8	14.31	18.16	18.33	25.41	28.80	2.29	13.78	3.68	25.86	12.06	1.94	3.30	2.31	1.00

Table 76: 100% SPGRID-NH

Grid/deg	SLU 1	SLU 2	SLU 3	DLU 21	DLU 22	GOR 1	BFP	THR ESH	PA PE	TWO Q	DIK H	DIK BD	DIK R	DIK BA
16x32/5	7.97	16.99	17.48	13.55	19.98	2.54	1.02	1.22	1.02	1.00	4.50	2.12	4.20	3.58
32x32/6	8.42	15.66	15.98	23.41	30.08	2.13	1.07	1.00	1.10	1.10	3.50	1.45	2.73	1.91
64x32/7	10.80	17.46	17.86	29.46	35.69	1.94	1.17	1.00	1.23	1.22	3.17	1.21	2.17	1.35
128x32/8	16.15	22.69	23.01	34.92	40.37	1.83	1.18	1.00	1.20	1.20	3.00	1.04	1.83	1.11

Table 77: 100% SPRAND-S4 and SPRAND-S16

$\lvert N\rvert$/deg	SLU 1	SLU 2	SLU 3	DLU 21	DLU 22	GOR 1	BFP	THR ESH	PA PE	TWO Q	DIK H	DIK BD	DIK R	DIK BA
128/4	8.33	15.33	16.00	6.67	10.00	3.67	1.00	1.67	1.00	1.00	6.00	4.67	9.33	11.33
256/4	12.77	22.00	22.69	13.54	17.54	4.15	1.08	1.31	1.08	1.00	6.15	5.15	9.08	4.46
512/4	13.91	22.06	22.32	17.91	21.86	3.56	1.25	1.00	1.23	1.23	4.65	5.07	6.33	2.28
1024/4	38.67	49.98	49.45	66.60	70.43	3.79	1.67	1.00	1.71	1.74	4.14	4.29	5.17	1.86
2048/4	100.32	116.10	114.46	135.51	141.57	3.10	1.88	1.00	2.18	2.10	3.09	4.00	3.55	1.05
128/16	11.11	16.00	15.89	11.44	15.00	2.78	1.00	1.00	1.11	1.11	2.67	2.33	3.67	3.67
256/16	21.20	27.38	27.22	21.29	25.49	3.02	1.31	1.00	1.36	1.36	2.58	2.31	3.29	2.20
512/16	76.60	87.97	80.42	52.28	59.71	3.22	1.43	1.00	1.67	1.65	2.53	2.51	3.08	1.62
1024/16	140.01	152.01	152.90	132.95	139.23	2.88	1.44	1.00	1.96	1.94	1.98	2.03	2.31	1.19
2048/16	243.73	259.72	258.62	242.47	250.13	2.82	1.66	1.21	2.91	2.71	1.72	1.90	1.91	1.00

Table 78: 100% SPRAND-D4 and SPRAND-D2

$\lvert N\rvert$/deg	SLU 1	SLU 2	SLU 3	DLU 21	DLU 22	GOR 1	BFP	THR ESH	PA PE	TWO Q	DIK H	DIK BD	DIK R	DIK BA
128/32	6.89	9.47	9.74	8.26	9.95	2.63	1.11	1.00	1.21	1.16	1.74	1.53	2.26	2.00
128/64	4.17	5.34	5.24	4.73	5.66	2.37	1.15	1.00	1.32	1.29	1.15	1.02	1.37	1.37
256/64	10.64	12.58	12.42	9.55	11.10	2.48	1.25	1.00	1.58	1.55	1.14	1.07	1.30	1.04
256/128	6.98	8.06	8.07	5.97	6.83	2.79	1.53	1.31	2.14	2.10	1.03	1.00	1.14	1.03
512/128	23.52	25.69	25.69	13.29	14.10	3.01	1.84	1.50	3.15	2.99	1.05	1.07	1.13	1.00
512/256	13.78	14.75	14.66	7.58	8.30	3.47	2.29	1.95	4.76	4.22	1.01	1.03	1.04	1.00
1024/256	26.85	28.61	28.18	20.93	21.53	3.81	3.13	2.56	8.48	6.76	1.00	1.09	1.04	1.03
1024/512	13.81	15.08	14.92	11.31	11.50	4.43	4.24	3.45	13.07	10.18	1.00	1.11	1.04	1.08
2048/512	16.31	17.39	16.97	13.29	13.51	3.94	4.95	3.72	17.29	13.01	1.01	1.07	1.00	1.09
2048/1024	9.18	9.65	9.61	7.42	7.42	5.25	8.03	5.61	31.10	20.28	1.03	1.07	1.00	1.14

Table 79: 100% SPRAND-LENS4

$\lvert N\rvert$/deg	$[L,U]$	SLU 1	SLU 2	SLU 3	DLU 21	DLU 22	GOR 1	BFP	THR ESH	PA PE	TWO Q	DIK H	DIK BD	DIK R	DIK BA
256/4	$[1,1]$	3.64	6.52	6.55	3.50	4.76	2.74	5.19	4.48	9.02	7.83	2.24	1.00	2.00	1.67
	$[0,10]$	3.41	6.09	6.11	3.16	4.39	2.18	3.48	2.86	3.00	3.25	2.09	1.00	2.05	1.41
	$[0,10^2]$	5.57	9.57	9.87	5.77	7.43	2.43	1.37	1.00	1.67	1.50	2.90	1.93	3.30	2.03
	$[0,10^4]$	12.46	21.92	22.00	12.46	16.31	3.92	1.08	1.31	1.00	1.00	6.15	5.31	9.00	4.00
	$[0,10^6]$	12.31	21.00	21.46	12.38	16.15	4.23	1.08	1.31	1.08	1.00	6.23	9.08	11.62	6.23
1024/4	$[1,1]$	18.16	24.14	24.17	30.27	32.95	4.43	34.86	22.71	53.96	65.31	2.52	1.00	2.00	3.69
	$[0,10]$	17.13	22.77	22.69	28.88	30.86	3.59	21.40	14.09	26.51	26.75	2.20	1.00	1.88	1.04
	$[0,10^2]$	19.70	26.31	26.28	33.31	35.75	3.50	4.63	3.11	6.57	5.44	2.60	1.60	2.55	1.00
	$[0,10^4]$	35.72	47.20	46.93	61.46	65.95	3.72	1.71	1.00	1.74	1.74	4.11	4.28	5.13	1.67
	$[0,10^6]$	42.92	56.79	56.65	72.57	78.18	3.58	1.17	1.10	1.00	1.00	4.72	4.93	7.38	1.86

Table 80: 100% SPRAND-LEND4

$\|N\|$/deg	$[L,U]$	SLU 1	SLU 2	SLU 3	DLU 21	DLU 22	GOR 1	BFP	THR ESH	PA PE	TWO Q	DIK H	DIK BD	DIK R	DIK BA
256/64	$[1,1]$	6.96	8.21	8.14	6.29	7.29	3.14	7.68	7.43	36.68	23.82	1.00	1.07	1.07	1.64
	$[0,10]$	7.04	8.32	8.29	6.32	7.29	2.64	4.54	3.64	17.71	11.61	1.00	1.11	1.07	1.18
	$[0,10^2]$	7.88	9.24	9.20	7.08	8.16	2.40	2.16	1.88	6.00	4.60	1.00	1.08	1.08	1.16
	$[0,10^4]$	10.72	12.72	12.78	9.72	11.11	2.50	1.28	1.00	1.67	1.61	1.11	1.06	1.33	1.06
	$[0,10^6]$	9.14	10.90	10.67	8.29	9.67	2.14	1.10	1.05	1.38	1.33	1.00	1.52	1.24	1.00
1024/256	$[1,1]$	17.76	19.19	18.98	14.06	14.49	5.21	40.43	38.78	357.35	239.22	1.00	1.29	1.02	18.35
	$[0,10]$	17.38	18.73	18.58	14.10	14.43	4.35	20.26	17.01	222.59	122.42	1.00	1.29	1.01	4.25
	$[0,10^2]$	19.69	21.08	20.88	15.25	15.73	4.06	9.57	7.37	83.21	41.99	1.00	1.27	1.02	1.51
	$[0,10^4]$	25.20	26.86	26.89	20.03	20.62	3.89	3.16	2.58	8.03	6.79	1.00	1.10	1.04	1.05
	$[0,10^6]$	29.12	31.43	30.74	23.26	24.16	3.80	1.96	2.08	2.81	2.82	1.07	1.21	1.15	1.00

Table 81: 100% SPRAND-PS4

$\|N\|$/deg	P	SLU 1	SLU 2	SLU 3	DLU 21	DLU 22	GOR 1	BFP	THR ESH	PA PE	TWO Q	DIK H	DIK BD	DIK R	DIK BA
256/4	0	11.85	21.08	21.00	11.00	15.38	3.92	1.08	1.31	1.00	1.00	6.08	5.38	9.08	5.08
	10^4	12.69	22.15	22.85	13.38	17.31	4.23	1.08	1.31	1.00	1.00	6.38	4.46	8.46	5.38
	10^5	12.54	21.92	22.38	12.23	16.38	4.00	1.08	2.15	1.08	1.00	12.85	5.46	8.92	6.92
	10^6	13.08	23.00	23.17	13.33	17.50	4.42	1.08	3.50	1.00	1.08	24.92	8.00	11.33	7.92
1024/4	0	34.80	45.83	45.86	61.31	61.96	3.63	1.72	1.00	1.77	1.78	4.04	4.23	5.08	1.66
	10^4	35.03	46.26	45.97	57.61	62.45	3.63	1.66	1.00	1.72	1.73	4.11	3.90	4.77	1.69
	10^5	28.36	37.07	36.95	48.31	51.55	2.79	1.27	1.00	1.29	1.30	4.66	3.36	3.49	1.59
	10^6	21.45	28.26	28.23	36.19	38.56	2.14	1.00	1.96	1.00	1.01	11.42	3.54	4.42	2.84

Table 82: 100% SPRAND-PD4

$\|N\|$/deg	P	SLU 1	SLU 2	SLU 3	DLU 21	DLU 22	GOR 1	BFP	THR ESH	PA PE	TWO Q	DIK H	DIK BD	DIK R	DIK BA
256/64	0	10.72	12.67	12.56	9.61	11.11	2.50	1.22	1.00	1.50	1.50	1.17	1.11	1.33	1.00
	10^4	8.43	10.00	9.91	7.65	8.78	2.00	1.00	1.00	1.26	1.22	1.48	1.17	1.35	1.22
	10^5	8.77	10.36	10.27	7.82	8.91	2.05	1.00	1.82	1.32	1.27	2.95	1.95	2.09	1.91
	10^6	8.77	10.23	10.18	7.95	9.18	2.09	1.00	1.91	1.27	1.23	3.27	2.14	2.18	2.00
1024/256	0	25.00	27.04	26.82	19.97	21.84	3.89	3.21	2.64	8.54	7.13	1.00	1.09	1.05	1.04
	10^4	15.35	16.51	16.41	12.69	12.29	2.29	1.85	1.61	5.01	4.10	1.05	1.03	1.01	1.00
	10^5	7.92	8.55	8.49	6.31	6.53	1.24	1.00	1.63	2.57	2.15	2.12	1.76	1.74	1.72
	10^6	7.67	8.32	8.26	6.23	6.42	1.20	1.00	2.14	2.69	2.22	2.89	2.05	2.04	2.02

Table 83: 100% NETGEN-S4 and NETGEN-S16

$\|N\|$/deg	SLU 1	SLU 2	SLU 3	DLU 21	DLU 22	GOR 1	BFP	THR ESH	PA PE	TWO Q	DIK H	DIK BD	DIK R	DIK BA
128/4	12.50	24.00	24.50	11.00	13.50	6.00	1.50	2.50	2.00	1.00	8.50	6.50	14.00	17.50
256/4	12.42	22.08	22.25	11.17	15.00	4.67	1.08	1.42	1.08	1.00	6.50	4.25	9.33	7.75
512/4	17.21	26.94	27.17	22.93	27.38	4.11	1.13	1.14	1.00	1.01	5.15	3.13	6.56	3.54
1024/4	49.29	63.71	62.79	82.68	88.45	3.58	1.18	1.05	1.00	1.00	4.50	2.34	5.11	1.97
2048/4	149.55	172.20	167.88	190.08	201.69	3.79	1.20	1.08	1.00	1.01	4.62	2.02	4.75	1.54
128/16	8.44	12.44	12.56	8.89	11.78	3.00	1.33	1.00	1.44	1.33	2.78	2.33	3.44	3.56
256/16	14.44	19.25	19.42	15.58	18.85	2.94	1.38	1.00	1.48	1.48	2.40	1.75	2.81	2.04
512/16	53.09	63.26	63.15	45.51	52.70	3.53	1.55	1.00	1.76	1.74	2.57	1.67	2.82	1.69
1024/16	130.49	138.54	139.15	125.44	132.02	3.38	1.62	1.00	1.85	1.83	2.35	1.36	2.39	1.30
2048/16	263.59	290.99	288.43	278.51	288.71	3.59	1.66	1.00	1.92	1.93	2.43	1.24	2.24	1.17

Table 84: 100% NETGEN-D4 and NETGEN-D2

$\|N\|$/deg	SLU 1	SLU 2	SLU 3	DLU 21	DLU 22	GOR 1	BFP	THRESH	PAPE	TWOQ	DIKH	DIKBD	DIKR	DIKBA
128/32	4.57	6.39	6.48	5.30	6.39	2.17	1.09	1.00	1.30	1.30	1.43	1.30	1.78	1.70
256/64	9.95	11.99	11.84	9.03	10.44	2.74	1.47	1.45	1.95	1.95	1.25	1.00	1.33	1.11
512/128	26.02	28.35	28.15	15.07	16.66	3.60	1.98	2.16	3.02	2.96	1.20	1.00	1.19	1.08
1024/256	28.65	30.73	30.29	22.82	23.44	4.09	2.41	2.70	4.13	4.00	1.07	1.00	1.07	1.02
2048/512	19.42	20.25	20.27	15.96	16.31	4.31	2.62	3.02	3.92	3.80	1.05	1.04	1.04	1.00
128/64	2.98	4.02	4.05	3.37	4.05	2.37	1.30	1.33	1.65	1.63	1.09	1.00	1.26	1.33
256/128	7.92	9.26	9.28	6.69	7.66	3.33	1.84	2.08	2.79	2.70	1.16	1.00	1.21	1.14
512/256	14.59	17.17	15.92	8.66	9.54	3.96	2.30	2.62	3.87	3.72	1.12	1.00	1.10	1.03
1024/512	13.47	14.58	14.76	10.97	11.40	4.14	2.50	2.92	4.49	4.30	1.00	1.00	1.04	1.02

Table 85: 100% NETGEN-LENS4

$\|N\|$/deg	$[L,U]$	SLU 1	SLU 2	SLU 3	DLU 21	DLU 22	GOR 1	BFP	THRESH	PAPE	TWOQ	DIKH	DIKBD	DIKR	DIKBA
256/4	$[0,10]$	13.17	22.67	23.42	11.33	16.50	4.33	1.17	1.42	1.00	1.08	6.42	1.83	3.58	1.67
	$[0,10^2]$	12.50	22.25	22.58	11.00	15.42	4.25	1.08	1.42	1.08	1.00	6.42	2.42	5.67	2.08
	$[0,10^4]$	12.92	22.50	22.83	12.08	16.33	4.50	1.17	1.42	1.08	1.00	6.42	4.33	9.33	7.08
	$[0,10^6]$	13.82	24.82	25.00	13.64	18.55	4.55	1.27	1.55	1.00	1.18	7.27	11.09	13.73	8.91
1024/4	$[0,10]$	41.72	54.82	55.00	70.94	76.31	3.66	1.20	1.07	1.00	1.01	4.40	1.27	2.21	1.20
	$[0,10^2]$	39.40	52.07	52.25	62.98	68.22	3.68	1.20	1.04	1.01	1.00	4.40	1.46	2.93	1.24
	$[0,10^4]$	43.67	57.15	56.94	68.91	76.35	3.70	1.19	1.07	1.00	1.01	4.46	2.39	5.14	2.00
	$[0,10^6]$	42.30	55.59	55.63	65.38	70.22	3.72	1.19	1.08	1.00	1.00	4.55	4.45	7.03	2.21

Table 86: 100% NETGEN-LEND4

$\|N\|$/deg	$[L,U]$	SLU 1	SLU 2	SLU 3	DLU 21	DLU 22	GOR 1	BFP	THRESH	PAPE	TWOQ	DIKH	DIKBD	DIKR	DIKBA
256/64	$[0,10]$	12.29	14.86	14.79	11.29	13.14	2.43	1.14	1.29	1.14	1.14	1.29	1.00	1.07	1.00
	$[0,10^2]$	10.87	13.07	12.87	10.07	11.47	3.20	1.80	1.80	2.33	2.20	1.33	1.00	1.13	1.00
	$[0,10^4]$	10.00	12.07	12.13	8.93	10.47	3.07	1.73	1.73	2.33	2.27	1.40	1.07	1.40	1.00
	$[0,10^6]$	8.10	9.70	9.80	7.30	8.55	2.25	1.30	1.30	1.75	1.70	1.00	1.55	1.30	1.05
1024/256	$[0,10]$	30.56	32.76	32.49	24.12	24.93	2.21	1.03	1.13	1.05	1.05	1.00	1.00	1.04	1.00
	$[0,10^2]$	28.89	31.53	31.29	23.33	24.05	3.84	2.40	2.79	2.93	2.88	1.06	1.00	1.04	1.01
	$[0,10^4]$	28.62	30.48	30.14	22.55	23.43	4.02	2.28	2.59	3.86	3.78	1.08	1.00	1.09	1.04
	$[0,10^6]$	27.97	30.20	29.78	22.81	23.29	3.84	2.19	2.52	3.78	3.71	1.06	1.11	1.16	1.00

Table 87: 100% SPACYC-PS4 and SPACYC-PS16

$\|N\|$/deg	$[L,U]$	ACC	SLU 1	SLU 2	SLU 3	DLU 21	DLU 22	GOR 1	BFP	THRESH	PAPE	TWOQ	DIKH	DIKBD	DIKR	DIKBA
128/4	$[0,10^4]$	1.00	1.33	6.67	6.67	2.00	4.67	2.00	1.33	2.33	1.00	1.33	3.00	2.33	3.67	7.33
256/4	$[0,10^4]$	1.00	1.21	6.14	6.43	2.14	5.00	1.43	1.57	1.71	1.50	1.50	3.00	1.29	2.79	3.43
512/4	$[0,10^4]$	1.00	1.33	7.00	7.16	2.79	5.91	1.35	2.61	2.21	2.33	2.53	3.53	1.26	2.82	2.33
1024/4	$[0,10^4]$	1.00	1.32	7.20	7.40	3.76	6.80	1.12	2.84	2.08	2.40	2.60	3.16	1.08	2.36	1.60
2048/4	$[0,10^4]$	1.00	1.28	7.78	8.08	4.25	7.45	1.26	3.45	2.30	2.72	3.00	3.53	1.05	2.25	1.17
128/16	$[0,10^4]$	1.00	1.60	5.00	5.20	2.80	5.40	1.40	2.20	2.00	2.40	2.40	2.60	2.00	2.60	5.00
256/16	$[0,10^4]$	1.00	1.58	4.83	5.04	3.04	5.29	1.21	2.42	2.08	2.58	2.67	2.33	1.38	2.21	2.71
512/16	$[0,10^4]$	1.00	1.41	4.44	4.57	3.52	5.71	1.11	2.66	1.93	2.76	2.96	2.12	1.07	1.71	1.53
1024/16	$[0,10^4]$	1.00	1.44	4.68	4.84	5.60	7.30	1.14	3.02	2.00	3.30	3.38	2.16	1.00	1.60	1.30
2048/16	$[0,10^4]$	1.09	1.43	5.06	5.20	6.82	8.73	1.17	3.69	2.41	3.97	4.16	2.33	1.00	1.56	1.15

Table 88: 100% SPACYC-NS4 and SPACYC-NS16

$\lvert N\rvert$/**deg**	$[L,U]$	**ACC**	**SLU 1**	**SLU 2**	**SLU 3**	**DLU 21**	**DLU 22**	**GOR 1**	**BFP**	**THR ESH**	**PA PE**	**TWO Q**	**DIK H**	**DIK BD**	**DIK R**	**DIK BA**
128/4	$[-10^4, 0]$	1.00	1.00	6.00	5.67	2.67	4.00	1.33	2.33	4.67	2.33	3.00	35.33	5.33	7.33	5.67
256/4	$[-10^4, 0]$	1.00	1.55	8.00	8.09	2.82	6.27	1.55	3.45	6.91	5.00	5.64	62.55	9.64	12.55	9.82
512/4	$[-10^4, 0]$	1.00	1.39	7.41	7.61	2.94	6.20	1.26	7.91	14.30	12.26	10.89	83.15	18.93	22.87	19.31
1024/4	$[-10^4, 0]$	1.00	1.45	8.23	8.41	4.50	7.77	1.41	12.59	21.14	22.64	20.50	327.77	27.55	33.36	28.18
2048/4	$[-10^4, 0]$	1.00	1.41	8.47	8.75	4.67	8.19	1.29	27.19	44.07	62.52	43.85	842.48	55.19	65.66	56.04
128/16	$[-10^4, 0]$	1.00	1.67	4.33	4.50	3.00	4.50	1.17	4.00	8.83	9.67	8.50	78.50	9.50	11.17	9.83
256/16	$[-10^4, 0]$	1.00	1.46	4.18	4.43	2.79	4.68	1.07	5.82	12.14	19.39	13.43	215.89	13.29	15.75	13.82
512/16	$[-10^4, 0]$	1.00	1.47	4.63	4.72	3.84	5.93	1.31	7.98	20.49	38.71	26.52	442.92	20.57	23.79	21.20
1024/16	$[-10^4, 0]$	1.00	1.43	4.73	4.84	5.84	7.37	1.10	16.88	45.86	119.8	69.61	1242.4	42.27	48.12	43.20
2048/16	$[-10^4, 0]$	1.00	1.35	4.82	4.94	6.84	8.18	1.14	30.53	82.70	179.0	129.8	1755.7	73.36	83.11	75.13

Table 89: 100% SPACYC-PD4 and SPACYC-PD2

$\lvert N\rvert$/**deg**	$[L,U]$	**ACC**	**SLU 1**	**SLU 2**	**SLU 3**	**DLU 21**	**DLU 22**	**GOR 1**	**BFP**	**THR ESH**	**PA PE**	**TWO Q**	**DIK H**	**DIK BD**	**DIK R**	**DIK BA**
128/32	$[0, 10^4]$	1.00	1.67	3.56	3.67	2.78	4.00	1.11	2.11	2.11	2.00	2.33	1.67	1.22	1.89	2.89
256/64	$[0, 10^4]$	1.16	1.49	2.94	3.12	3.25	4.70	1.15	2.58	1.99	3.03	2.99	1.27	1.00	1.24	1.49
512/128	$[0, 10^4]$	1.54	1.94	3.30	3.44	6.00	7.65	1.53	3.51	2.69	4.58	4.56	1.27	1.00	1.15	1.17
1024/256	$[0, 10^4]$	1.77	3.85	5.03	5.15	12.69	13.29	1.68	4.19	3.12	5.96	5.90	1.15	1.00	1.09	1.07
2048/512	$[0, 10^4]$	1.75	3.95	4.70	4.76	11.88	12.21	1.65	4.30	3.23	5.97	5.85	1.07	1.00	1.03	1.05
128/64	$[0, 10^4]$	1.12	1.31	2.44	2.50	2.38	3.38	1.00	1.94	1.81	2.06	2.12	1.19	1.12	1.31	2.12
256/128	$[0, 10^4]$	1.50	1.50	2.63	2.72	3.37	4.58	1.40	2.94	2.47	3.37	3.37	1.19	1.00	1.17	1.29
512/256	$[0, 10^4]$	1.75	1.81	2.80	2.93	5.58	6.81	1.66	3.81	3.13	5.00	4.84	1.17	1.00	1.09	1.11
1024/512	$[0, 10^4]$	1.88	3.88	4.70	4.77	11.11	11.54	1.74	3.86	3.34	5.51	5.40	1.06	1.00	1.05	1.09
2048/1024	$[0, 10^4]$	1.78	2.85	3.32	3.36	8.67	8.83	1.69	4.09	3.51	5.32	5.11	1.02	1.00	1.01	1.05

Table 90: 100% SPACYC-ND4 and SPACYC-ND2

$\|N\|$/deg	$[L,U]$	ACC	SLU 1	SLU 2	SLU 3	DLU 21	DLU 22	GOR 1	BFP	THR ESH	PA PE	TWO Q	DIK H	DIK BD	DIK R	DIK BA
128/32	$[-10^4, 0]$	1.00	1.18	3.00	3.00	2.27	3.64	1.00	3.36	7.55	11.00	7.27	70.82	7.36	8.73	7.82
256/64	$[-10^4, 0]$	1.00	1.26	2.41	2.52	2.74	3.67	1.08	6.31	17.16	42.41	23.00	341.90	15.18	16.60	15.64
512/128	$[-10^4, 0]$	1.00	1.26	2.12	2.20	4.00	4.62	1.04	10.63	29.93	200.8	54.10	616.51	25.34	26.45	25.71
1024/256	$[-10^4, 0]$	1.00	2.09	2.80	2.85	7.24	7.01	1.05	21.11	67.16	1091	232.6	5269	54.55	55.51	54.76
128/64	$[-10^4, 0]$	1.00	1.05	2.00	2.05	1.86	2.57	1.00	3.48	8.24	12.90	8.38	78.62	7.62	8.48	7.95
256/128	$[-10^4, 0]$	1.04	1.03	1.74	1.83	2.28	2.94	1.00	6.75	19.99	59.65	24.48	231.81	17.08	18.07	17.47
512/256	$[-10^4, 0]$	1.00	1.02	1.58	1.61	2.99	3.52	1.02	11.70	37.64	166.4	76.76	613.97	30.51	31.27	30.73
1024/512	$[-10^4, 0]$	1.00	1.95	2.43	2.48	5.78	5.99	1.03	22.09	74.89	761.8	206.6	4031	66.13	66.85	66.17

Table 91: 100% SPACYC-P2N128 and SPACYC-P2N1024

$\|N\|$/deg	$[L,U]$ $\times 1000$	ACC	SLU 1	SLU 2	SLU 3	DLU 21	DLU 22	GOR 1	BFP	THR ESH	PA PE	TWO Q	DIK H	DIK BD	DIK R	DIK BA
128/16	$[0, 10]$	1.90	2.78	7.73	8.10	4.71	7.71	2.10	1.05	1.37	1.00	1.00	3.02	2.95	4.90	7.66
	$[-1, 9]$	1.39	2.27	6.20	6.51	3.63	6.35	1.78	1.00	1.27	1.00	1.02	2.96	2.20	3.53	6.22
	$[-2, 8]$	1.00	1.51	4.17	4.34	2.56	4.14	2.90	1.00	1.34	1.08	1.09	3.86	2.00	2.61	5.35
	$[-3, 7]$	1.00	1.48	4.01	4.19	2.46	4.11	1.09	1.70	2.41	1.95	1.94	9.78	3.29	4.03	6.08
	$[-4, 6]$	1.00	1.53	4.10	4.31	2.54	4.19	1.05	2.47	3.90	3.56	3.28	13.37	5.12	6.38	7.85
	$[-5, 5]$	1.00	1.51	4.06	4.24	2.57	4.16	1.09	3.05	5.38	4.43	4.72	34.03	6.78	8.41	9.53
	$[-6, 4]$	1.00	1.48	4.11	4.27	2.62	4.15	1.16	3.99	7.80	7.34	6.85	43.59	9.27	11.34	12.03
	$[-10, 0]$	1.00	1.64	4.49	4.67	2.83	4.58	1.18	6.04	12.72	14.60	11.99	92.69	15.28	18.31	15.89
1024/16	$[0, 10]$	1.71	2.34	7.79	8.03	8.71	11.50	1.66	1.08	1.05	1.05	1.00	2.89	1.84	3.47	2.00
	$[-1, 9]$	1.38	1.98	6.60	6.76	7.80	9.73	1.49	1.13	1.00	1.13	1.13	2.84	1.53	2.62	1.73
	$[-2, 8]$	1.00	1.43	4.68	4.86	5.25	6.97	1.14	1.38	1.46	1.70	1.67	5.57	1.86	2.63	2.37
	$[-3, 7]$	1.00	1.61	4.70	4.81	5.25	6.91	1.11	3.69	5.67	6.55	6.17	30.73	7.33	8.67	7.84
	$[-4, 6]$	1.00	1.54	5.12	5.24	5.88	7.73	1.20	10.07	19.15	38.80	20.66	129.2	23.03	26.75	23.95
	$[-5, 5]$	1.00	1.53	5.20	5.25	6.53	7.90	1.27	20.66	49.14	88.80	55.68	542.2	54.08	62.07	55.54
	$[-6, 4]$	1.00	1.40	4.47	4.65	6.03	7.07	1.06	27.40	72.26	121.1	101.7	667.5	82.01	93.62	84.31
	$[-10, 0]$	1.00	1.42	4.71	4.78	5.78	7.37	1.06	37.97	94.51	167.4	91.68	814.4	114.8	131.1	117.9

REFERENCES

[1] Aggarwal, A., Oblak, M., and Vemuganti, R., "A heuristic solution procedure for multicommodity integer flows," *Computers & Operations Research*, vol. 22, pp. 1075–1087, Dec 1995.

[2] Aho, A., Hopcroft, J., and Ullman, J., *The design and analysis of computer algorithms*. Addison-Wesley, 1974.

[3] Ahuja, R., Magnanti, T., and Orlin, J., *Network flows: theory, algorithms and applications*. Englewood Cliffs, NJ: Prentice Hall, 1993.

[4] Ahuja, R., Mehlhorn, K., Orlin, J., and Tarjan, R., "Faster algorithms for the shortest path problem," *Journal of ACM*, vol. 37, no. 2, pp. 213–223, 1990.

[5] Aikens, C., "Facility location models for distribution planning," *European Journal of Operational Research*, vol. 22, no. 3, pp. 263–279, 1985.

[6] Albers, S., Garg, N., and Leonardi, S., "Minimizing stall time in single and parallel disk systems," *Journal of the ACM*, vol. 47, no. 6, pp. 969–986, 2000.

[7] Albers, S. and Witt, C., "Minimizing stall time in single and parallel disk systems using multicommodity network flows," in *Approximation, Randomization and Combinatorial Optimization: Algorithms and Techniques* (Goemans, M., Jansen, K., Rolim, J., and Trevisan, L., eds.), vol. 2129, (Berlin Heidelberg), pp. 12–23, Springer, Aug 2001.

[8] Albrecht, C., "Global routing by new approximation algorithms for multicommodity flow," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 20, pp. 622–632, May 2001.

[9] Ali, A., Barnett, D., Farhangian, K., Kennington, J., Patty, B., Shetty, B., McCarl, B., and Wong, P., "Multicommodity network problems: Applications and computations," *IIE Transactions*, vol. 16, no. 2, pp. 127–134, 1984.

[10] Ali, A., Helgason, R., Kennington, J., and Lall, H., "Computational comparison among three multicommodity network flow algorithms," *Operations Research*, vol. 28, no. 4, pp. 995–1000, 1980.

[11] Ali, A. and Kennington, J., "Mnetgen program documentation," technical report 77003, Department of IEOR, Southern Methodist University, Dallas, 1977.

[12] Alon, N., Galil, Z., and Margalit, O., "On the exponent of the all pairs shortest paht problem," *Journal of Computer and System Sciences*, vol. 54, pp. 255–262, April 1997.

[13] Alvelos, F. and Valério de Carvalho, J., "Solving the multicommodity flow problem by branch-and-price," tech. rep., Departamento de Produção e Sistemas, Universidade do Minho, Braga, Portugal, 2000.

[14] Asano, Y. and Imai, H., "Practical efficiency of the linear-time algorithm for the singale source shortest path problem," *Journal of the Operations Research Society of Japan*, vol. 43, pp. 431–447, December 2000.

[15] Assad, A., "Multicommodity network flows-a survey," *Networks*, vol. 8, no. 1, pp. 37–91, 1978.

[16] Assad, A., "Modelling of rail networks: toward a routing/makeup model," *Transportation Research, Part B (Methodological)*, vol. 14B, no. 1-2, pp. 101–114, 1980.

[17] Assad, A., "Solving linear multicommodity flow problems," in *Proceedings of the IEEE International Conference on Circuits and Computers ICCC 80*, pp. 157–161, Octobor 1980.

[18] Atkinson, D. and Vaidya, P., "A cutting plane algorithm for convex programming that uses analytic centers," *Mathematical Programming*, vol. 69, no. 1, Ser.B, pp. 1–43, 1995.

[19] Awerbuch, B. and Leighton, T., "A simple local-control approximation algorithm for multicommodity flow," in *Proceedings of the 34th annual IEEE Symposium on Foundations of Computer Science*, pp. 459–468, IEEE, 1993.

[20] Awerbuch, B. and Leighton, T., "Improved approximations for the multi–commodity flow problem and local competitive routing in networks," in *Proceedings of the twenty-sixth annual ACM symposium on Theory of computing*, (New York, NY, USA), pp. 487–496, ACM Press, 1994.

[21] Backhouse, R. and Carré, B., "Regular algebra applied to path finding problems," *Journal of the Institute of Mathematics and Its Applications*, vol. 15, pp. 161–186, April 1975.

[22] Backhouse, R. and Carré, B., "A comparison of gaussian and gauss-jordan elimination in regular algebra," *International Journal of Computer Mathematics*, vol. 10, no. 3-4, pp. 311–325, 1982.

[23] Bahiense, L., Barahona, F., and Porto, O., "Solving steiner tree problems in graphs with lagrangian relaxation," rc 21846, Research Division, IBM T.J. Watson Research Center, Yorktown Heights, NY, 2000.

[24] Barahona, F. and Anbil, R., "The volume algorithm: producing primal solutions with a subgradient method," *Mathematical Programming*, vol. 87, no. 3, Ser. A, pp. 385–399, 2000.

[25] Barahona, F. and Anbil, R., "On some difficult linear programs coming from set partitioning," *Discrete Applied Mathematics*, vol. 118, pp. 3–11, Apr 2002.

[26] Barahona, F. and Chudak, F., "Near-optimal solutions to large scale facility location problems," rc 21606, IBM Research Division, T.J. Watson Research Center, Yorktown Heights, New York, 1999.

[27] Barahona, F. and Chudak, F., "Solving large scale uncapacitated facility location problems," in *Approximation and complexity in numerical optimization : continuous*

and discrete problems (v.42) (Pardalos, P., ed.), pp. 48–62, Dordrecht: Kluwer Academic Publishers, 2000.

[28] Barnes, E., Chen, V., Gopalakrishnan, B., and Johnson, E., "A least squares primal-dual algorithm for solving linear programming problems," *Operations Research Letters*, vol. 30, pp. 289–294, Oct 2002.

[29] Barnes, E., Gopalakrishnan, B., Johnson, E., and Sokol, J., "A combined objective least-squares algorithm." in preparation.

[30] Barnes, E., Gopalakrishnan, B., Johnson, E., and Sokol, J., "A least-squares network flow algorithm." in preparation.

[31] Barnett, D., Binkley, J., and McCarl, B., "Port elevator capacity and natioinal and world grain shipments," *Western Journal of Agricultural Economics*, vol. 9, pp. 77–84, 1984.

[32] Barnhart, C., *A network-based primal-dual solution methodology for the multicommodity network flow problem.* PhD thesis, Transportation Systems Division, M.I.T., Cambridge, MA, 1988.

[33] Barnhart, C., "Dual-ascent methods for large-scale multicommodity flow problems," *Naval Research Logistics*, vol. 40, pp. 305–324, April 1993.

[34] Barnhart, C., Hane, C., and Vance, P., "Integer multicommodity flow problems," in *Integer programming and combinatorial optimization, Proceedings of the 5th International Conference (IPCO V) held in Vancouver, BC, Canada* (Cunningham, W., McCormick, S., and Queyranne, M., eds.), vol. 1084, (Berlin Heidelberg), pp. 58–71, Springer, Jun 1996.

[35] Barnhart, C., Hane, C., and Vance, P., "Using branch-and-price-and-cut to solve origin-destination integer multicommodity flow problems," *Operations Research*, vol. 48, no. 2, pp. 318–326, 2000.

[36] Barnhart, C., Johnson, E., Hane, C., and Sigismondi, G., "An alternative formulation and solution strategy for multicommodity network flow problems," *Telecommunication Systems*, vol. 3, pp. 239–258, 1995.

[37] Barnhart, C., Johnson, E., Nemhauser, G., Savelsbergh, M., and Vance, P., "Branch-and-price: column generation for solving huge integer programs," *Operations Research*, vol. 46, no. 3, pp. 316–329, 1998.

[38] Barnhart, C. and Sheffi, Y., "A network-based primal-dual heuristic for the solution of multicommodity network flow problems," *Transportation Science*, vol. 27, pp. 102–117, May 1993.

[39] Bazarra, M., Hurley, J., Johnson, E., Nemhauser, G., Sokol, J., Wang, I.-L., Chew, E., Huang, H., Mok, I., Tan, K., and Teo, C., "The asia pacific air cargo system," Tech. Rep. TLI-AP/00/01, The Logistics Institue-Asia Pacific, 2000.

[40] Bellman, R., "On a routing problem," *Quarterly of Applied Mathematics*, vol. 16, pp. 87–90, 1958.

[41] Bellmore, M., Bennington, G., and Lubore, S., "A multivehicle tanker scheduling problem," *Transportation Science*, vol. 5, pp. 36–47, 1971.

[42] Benders, J., "Partitioning procedures for solving mixed-variables programming problems," *Numerische Mathematik*, vol. 4, pp. 238–252, 1962.

[43] Bertsekas, D., "A unified framework for primal-dual methods in minimum cost network flow problems," *Mathematical Programming*, vol. 32, no. 2, pp. 125–145, 1985.

[44] Bertsekas, D., "A simple and fast label correcting algorithm for shortest paths," *Networks*, vol. 23, pp. 703–709, December 1993.

[45] Bertsekas, D., *Network ptimization: continuous and discrete models*. P.O. Box 391, Belmont, MA 02178-9998: Athena Scientific, 1998.

[46] Bertsekas, D., Hossein, P., and Tseng, P., "Relaxation methods for network flow problems with convex arc costs," *SIAM Journal on Control and Optimization.*, vol. 25, no. 5, pp. 1219–1243, 1987.

[47] Bertsekas, D., Pallottino, S., and Scutellà, M., "Polynomial auction algorithms for shortest paths," *Computational Optimization and Applications*, vol. 4, pp. 99–125, April 1995.

[48] Bienstock, D., "Approximately solving large-scale linear programs. i: Strengthening lower bounds and accelerating convergence," Tech. Rep. CORC Report 1999-1, Department of IEOR, Columnbia University, New York, NY, 1999.

[49] Bienstock, D., "Potential function methods for approximately solving linear programming problems: Theory and practice," tech. rep., CORE, Université catholique de Louvain, Belgium, 2001.

[50] Bienstock, D., Chopra, S., Günlük, O., and Tsai, C.-Y., "Minimum cost capacity installation for multicommodity network flows," *Mathematical Programming*, vol. 81, no. 2, Ser. B, pp. 177–199, 1998.

[51] Bienstock, D. and Günlük, O., "Computational experience with a difficult mixed-integer multicommodity flow problem," *Mathematical Programming*, vol. 68, no. 2, Ser. A, pp. 213–237, 1995.

[52] Bienstock, D. and Muratore, G., "Strong inequalities for capacitated survivable network design problems," *Mathematical Programming*, vol. 89, no. 1, Ser. A, pp. 127–147, 2000.

[53] Bienstock, D. and Saniee, I., "Atm network design: traffic models and optimization-based heuristics," *Telecommunication Systems - Modeling, Analysis, Design and Management*, vol. 16, no. 3-4, pp. 399–421, 2001.

[54] Björck, Å., "Iterative refinement of linear least squares solution," *BIT (Nordisk Tidskrift for Informationsbehandling)*, vol. 7, pp. 257–278, 1967.

[55] Björck, Å., "Iterative refinement of linear least squares solutions," *BIT (Nordisk Tidskrift for Informationsbehandling)*, vol. 8, no. 1, pp. 8–30, 1968.

[56] Bourbeau, B., Crainic, T., and Gendron, B., "Branch-and-bound parallelization strategies applied to a depot location and container fleet management problem," *Parallel Computing*, vol. 26, no. 1, pp. 27–46, 2000.

[57] Brunetta, L., Conforti, M., and Fischetti, M., "A polyhedral approach to an integer multicommodity flow problem," *Discrete Applied Mathematics*, vol. 101, pp. 13–36, Apr 2000.

[58] Burton, D., "On the inverse shortest path problem," *Département de Mathématique, Faculté des Sciences, Facultés Universitaires Notre-Dame de la Paix de Namur*, 1993.

[59] Cappanera, P. and Frangioni, A., "Symmetric and asymmetric parallelization of a cost-decomposition algorithm for multicommodity flow problems," *INFORMS Journal on Computing*, vol. to appear, 2002.

[60] Cappanera, P. and Gallo, G., "On the airline crew rostering problem," tech. rep., Dipartimento di Informatica, Università di Pisa, Oct 2001.

[61] Carden, R. and Cheng, C, K., "A global router using an efficient approximate multicommodity multiterminal flow algorithm," in *28th ACM/IEEE Design Automation Conference*, pp. 316–321, ACM/IEEE, 1991.

[62] Carolan, W., Hill, J., Kennington, J., Niemi, S., and Wichmann, S., "An empirical evaluation of the korbx algorithms for military airlift applications," *Operations Research*, vol. 38, no. 2, pp. 240–248, 1990.

[63] Carré, B., *Graphs and networks.* Walton Street, Oxford OX2 6DP: Oxford University Press, 1979.

[64] Carré, B., "A matrix factorization method for finding optimal paths through networks," in *I.E.E. Conference Publication (Computer-Aided Design)*, no. 51, pp. 388–397, 1969.

[65] Carré, B., "An algebra for network routing problems," *Journal of Institute of Mathematics and Its Applications*, vol. 7, pp. 273–294, 1971.

[66] Castro, J., "Solving difficult multicommodity problems through a specialized interior-point algorithm," Tech. Rep. DR20000-14, Statistics and Operations Research Dept., Universitat Politècnica de Catalunya, Pau Gargallo 5, 08028 Barcelona, Spain, Aug 2000.

[67] Castro, J., "A specialized interior-point algorithm for multicommodity network flows," *SIAM Journal on Optimization*, vol. 10, no. 3, pp. 852–877, 2000.

[68] Castro, J., "Solving quadratic multicommodity problems through an interior-point algorithm," Tech. Rep. DR20001-14, Statistics and Operations Research Dept., Universitat Politècnica de Catalunya, Pau Gargallo 5, 08028 Barcelona, Spain, Aug 2001.

[69] Castro, J. and Frangioni, A., "A parallel implementation of an interior-point algorithm for multicommodity network flows," in *Vector and parallel processing–VECPAR 2000 : 4th International Conference, Porto, Portugal, June 21-23, 2000* (J.M.L.M. Palma, J. Dongarra, V. H., ed.), vol. 1981, pp. 301–315, Springer, 2001.

[70] Castro, J. and Nabona, N., "An implementation of linear and nonlinear multicommodity network flows," *European Journal of Operational Research*, vol. 92, pp. 37–53, Jul 1996.

[71] Cerulli, R., Festa, P., and Raiconi, G., "Graph collapsing in shortest path auction algorithms," *Computational Optimization and Applications*, vol. 18, no. 3, pp. 199–220, 2001.

[72] Chardaire, P. and Lisser, A., "Minimum cost multicommodity flow," in *Handbook of Applied Optimization* (Pardalos, P. and Resende, M., eds.), pp. 404–421, Oxford University Press, 2002.

[73] Chardaire, P. and Lisser, A., "Simplex and interior point specialized algorithms for solving non-oriented multicommodity flow problems," *Operations Research*, to appear.

[74] Cherkassky, B., Goldberg, A., and Radzik, T., "Shorthest paths algorithms: theory and experimental evaluation," *Mathematical Programming*, vol. 73, pp. 129–174, June 1996.

[75] Chifflet, J., Mahey, P., and Reynier, V., "Proximal decomposition for multicommodity flow problems with convex costs," *Telecommunication Systems*, vol. 3, no. 1, pp. 1–10, 1994.

[76] Choi, I. and Goldfarb, D., "Solving multicommodity network flow problems by an interior point method," in *Large-scale numerical optimization (Ithaca, NY, 1989),* (Coleman, T. and Li, Y., eds.), (Philadelphia, PA), pp. 58–69, SIAM, 1990.

[77] Clarke, S. and Surkis, J., "An operations research approach to racial desegregation of school systems," *Socio-Economic Planning Sciences*, vol. 1, pp. 259–272, 1968.

[78] Crainic, T., Ferland, J.-A., and Rousseau, J.-M., "A tactical planning model for rail freight transportation," *Transportation Science*, vol. 18, pp. 165–184, May 1984.

[79] Crainic, T., "Service network design in freight transportation," *European Journal of Operational Research*, vol. 122, pp. 272–288, Apr 2000.

[80] Crainic, T., Dejax, P., and Delorme, L., "Models for multimode multicommodity location problems with interdepot balancing requirements," *Annals of Operations Research*, vol. 18, no. 1-4, pp. 279–302, 1989.

[81] Crainic, T. and Delorme, L., "Dual-ascent procedures for multicommodity location-allocation problems with balancing requirements," *Transportation Science*, vol. 27, pp. 90–101, May 1993.

[82] Crainic, T., Delorme, L., and Dejax, P., "A branch-and-bound method for multicommodity location with balancing requirements," *European Journal of Operational Research*, vol. 65, no. 3, pp. 368–382, 1993.

[83] Crainic, T., Gendreau, M., Soriano, P., and Toulouse, M., "A tabu search procedure for multicommodity location/allocation with balancing requirements," *Annals of Operations Research*, vol. 41, pp. 359–383, May 1993.

[84] D'Amours, S., Montreuil, B., and Soumis, F., "Price-based planning and scheduling of multiproduct orders in symbiotic manufacturing networks," *European Journal of Operational Research*, vol. 96, no. 1, pp. 148–166, 1996.

[85] Dantzig, G., "On the shortest route through a network," *Management Science*, vol. 6, pp. 187–190, 1960.

[86] Dantzig, G., *Linear Programming and Extensions*. Princeton University Press, 1963.

[87] Dantzig, G., "All shortest routes in a graph," in *Theory of Graphs (International Symposium., Rome, 1966)*, pp. 91–92, New York: Gordon and Breach, 1967.

[88] Dantzig, G. and Wolfe, P., "The decomposition algorithm for linear programs," *Econometrica*, vol. 29, pp. 767–778, 1961.

[89] Davis, T., Gilbert, J., Larimore, S., and Ng, E., "A column approximate minimum degree ordering algorithm," technical report tr-00-005, Department of Computer and Information Science and Engineering, University of Florida, October 2000.

[90] De Leone, R., Meyer, R., and Kontogiorgis, S., "Alternating directions methods for the parallel solution of large-scale block-structured optimization problems," tech. rep., Computer Sciences Technical Report 1217, Computer Science, University of Wisconsin-Madison, Feb 1994.

[91] Demmel, J., Gilbert, J., and Li, X., "Superlu user's guide," September 1999.

[92] Detlefsen, N. and Wallace, S., "The simplex algorithm for multicommodity networks," *Networks*, vol. 39, no. 1, pp. 15–28, 2001.

[93] Dial, R., "Algorithm 360 shortest path forest with topological ordering," *Communications of the ACM*, vol. 12, pp. 632–633, 1965.

[94] Dial, R., Glover, F., Karney, D., and Klingman, D., "A computational analysis of alternative algorithms and labeling techniques for finding shortest path trees," *Networks*, vol. 9, no. 3, pp. 215–248, 1979.

[95] Dijkstra, E., "A note on two problems in connection with graphs," *Numerische Mathematik*, vol. 1, pp. 269–271, 1959.

[96] Divoky, J. and Hung, H., "Performance of shortest path algorithms in network flow problems," *Management Science*, vol. 36, pp. 661–673, June 1990.

[97] Druckerman, J., Silverman, D., and Viaropulos, K., *IBM optimization subroutine library guide and reference,*. IBM, Kingston, NY, 1991.

[98] Duff, I., Erisman, A., and Reid, J., *Direct methods for sparse matrices*. New York: Oxford University Press Inc., 1989.

[99] Eckstein, J. and Fukushima, M., "Some reformulations and applications of the alternating direction method of multipliers," in *Large scale optimization : State of the Art (Gainesville, FL, 1993)* (Hager, W., Hearn, D., and Pardalos, P., eds.), pp. 115–134, Kluwer Academic Publishers, 1994.

[100] Evans, J., "A combinatorial equivalence between a class of multicommodity flow problems and the capacitated transportation problem," *Mathematical Programming*, vol. 10, pp. 401–404, Jun 1976.

[101] Evans, J., "On equivalent representations of certain multicommodity networks as single commodity flow problems," *Mathematical Programming*, vol. 15, pp. 92–99, Jul 1978.

[102] Evans, J., "The simplex method for integral multicommodity networks," *Naval Research Logistics Quarterly*, vol. 25, pp. 31–37, Mar 1978.

[103] Evans, J., "A single-commodity transformation for certain multicommodity networks," *Operations Research*, vol. 26, no. 4, pp. 673–680, 1978.

[104] Evans, J., "The multicommodity assignment problem: a network aggregation heuristic," *Computers & Mathematics with Applications*, vol. 7, no. 2, pp. 187–194, 1981.

[105] Evans, J., "A network decomposition/aggregation procedure for a class of multicommodity transportation problems," *Networks*, vol. 13, no. 2, pp. 197–205, 1983.

[106] Evans, J., Jarvis, J., and Duke, R., "Graphic matroids and the multicommodity transportation problem," *Mathematical Programming*, vol. 13, pp. 323–328, Dec 1977.

[107] Even, S., Itai, A., and Shamir, A., "On the complexity of timetable and multicommodity flow problems," *SIAM Journal on Computing*, vol. 5, no. 4, pp. 691–703, 1976.

[108] Farbey, B. A., Land, A. H., and Murchland, J. D., "The cascade algorithm for finding all shorest distances in a directed graph," *Management Science*, vol. 14, pp. 19–28, September 1967.

[109] Farvolden, J., Powell, W., and Lustig, I., "A primal partitioning solution for the arc-chain formulation of a multicommodity network flow problem," *Operations Research*, vol. 41, no. 4, pp. 669–693, 1993.

[110] Ferris, M., Meeraus, A., and Rutherford, T., "Computing wardropian equilibria in a complementarity framework," *Optimization Methods and Software*, vol. 10, no. 5, pp. 669–685, 1999.

[111] Fleischer, L., "Approximating fractional multicommodity flow independent of the number of commodities," *SIJDM: SIAM Journal on Discrete Mathematics*, vol. 13, no. 4, pp. 505–520, 2000.

[112] Florian, M., Nguyen, S., and Pallottino, S., "A dual simplex algorithm for finding all shortest paths," *Networks*, vol. 11, no. 4, pp. 367–378, 1981.

[113] Floyd, R., "Algorithm 97, shortest path," *Comm. ACM*, vol. 5, p. 345, 1962.

[114] Ford Jr., L., *Network flow theory*, pp. –923. Santa Monica, California: The RAND Corp., 1956.

[115] Ford Jr., L. and Fulkerson, D., "A suggested computation for maximal multicommodity network flows," *Management Science*, vol. 5, pp. 97–101, 1958.

[116] Ford Jr., L. and Fulkerson, D., *Flows in networks.* Princeton, NJ: Princeton University Press, 1962.

[117] Ford Jr., L. and Fulkerson, D., *Flows in networks.* Princeton, NJ: Princeton University Press, 1962.

[118] Frangioni, A., *Dual ascent methods and multicommodity flow problems.* PhD thesis, Dipartimento di Informatica, Università di Pisa, May 1997.

[119] Frangioni, A. and Gallo, G., "A bundle type dual-ascent approach to linear multicommodity min-cost flow problems," *INFORMS Journal on Computing*, vol. 11, no. 4, pp. 370–393, 1999.

[120] Frank, M. and Wolfe, P., "An algorithm for quadratic programming," *Naval Research Logistics Quarterly*, vol. 3, pp. 95–110, 1956.

[121] Fredman, M. and Tarjan, R., "Fibonacci heaps and their uses in improved network optimization algorithms," *Journal of the ACM*, vol. 34, no. 3, pp. 596–615, 1987.

[122] Fredman, M., "New bounds on the complexity of the shortest path problems," *SIAM Journal on Computing*, vol. 5, pp. 83–89, March 1976.

[123] Fujishige, S., "A note on the problem of updating shortest paths," *Networks*, vol. 11, no. 3, pp. 317–319, 1981.

[124] Gabrela, V., Knippelb, A., and Minoux, M., "Exact solution of multicommodity network optimization problems with general step cost functions," *Operations Research Letters*, vol. 25, pp. 15–23, Aug 1999.

[125] Galil, Z. and Margalit, O., "All pairs shortest distances for graphs with small integer length edges," *Information and Computation*, vol. 134, pp. 103–139, May 1997.

[126] Galil, Z. and Margalit, O., "All pairs shortest paths for graphs with small integer length edges," *Journal of Computer and System Sciences*, vol. 54, pp. 243–254, April 1997.

[127] Gallo, G. and Pallottino, S., "A new algorithm to find the shortest paths between all pairs of nodes," *Discrete Applied Mathematics*, vol. 4, no. 1, pp. 23–35, 1982.

[128] Garg, N. and Könemann, J., "Faster and simpler algorithms for multicommodity flow and other fractional packing problems," in *39th Annual Symposium on Foundations of Computer Science: proceedings: November 8–11, 1998, Palo Alto, California* (IEEE, ed.), (1109 Spring Street, Suite 300, Silver Spring, MD 20910, USA), pp. 300–309, IEEE Computer Society Press, 1998.

[129] Gendron, B. and Crainic, T., "A parallel branch-and-bound algorithm for multicommodity location with balancing requirements," *Computers & Operations Research*, vol. 24, no. 9, pp. 829–847, 1997.

[130] Gendron, B., Crainic, T., and Frangioni, A., "Multicommodity capacitated network design," in *Telecommunications network planning* (Soriano, P. and Sansò, B., eds.), Boston: Kluwer Academic Publisher, 1999.

[131] Geoffrion, A., "Primal resource-directive approaches for optimizing nonlinear decomposable systems," *Operations Research*, vol. 18, pp. 375–403, 1970.

[132] Geoffrion, A. and Graves, G., "Multicommodity distribution system design by benders decomposition," *Management Science*, vol. 20, no. 5, pp. 822–844, 1974.

[133] Girard, A. and Sansò, B., "Multicommodity flow models, failure propagation, and reliable loss network design," *IEEE/ACM Transactions on Networking*, vol. 6, pp. 82–93, Feb 1998.

[134] Glover, F., Glover, R., and Klingman, D., "Computational study of an improved shortest path algorithm," *Networks*, vol. 14, no. 1, pp. 25–36, 1984.

[135] Glover, F., Klingman, D., and Phillips, N., "A new polynomially bounded shortest paths algorithm," *Operations Research*, vol. 33, no. 1, pp. 65–73, 1985.

[136] Goffin, J.L.and Haurie, A. and Vial, J.-P., "Decomposition and nondifferentiable optimization with the projective algorithm," *Management Science*, vol. 38, pp. 284–302, Feb 1992.

[137] Goffin, J.-L., Gondzio, J., Sarkissian, R., and Vial, J.-P., "Solving nonlinear multicommodity flow problems by the analytic center cutting plane method," *Mathematical Programming*, vol. 76, no. 1, Ser.B, pp. 131–154, 1997.

[138] Goldberg, A., "A natural randomization strategy for multicommodity flow and related algorithms," *Information Processing Letters*, vol. 42, no. 5, pp. 249–256, 1992.

[139] Goldberg, A., Oldham, J., Plotkin, S., and Stein, C., "An implementation of a combinatorial approximation algorithm for minimum-cost multicommodity flow," in *Integer programming and combinatorial optimization, Proceedings of the 6th International Conference (IPCO VI) held in Houston, TX, 1998* (Bixby, R., Boyd, E., and Ríos-Mercado, R., eds.), pp. 338–352, Berlin: Springer, 1998.

[140] Goldberg, A. and Radzik, T., "A heuristic improvement of the bellman-ford algorithm," *Applied Mathematics Letters*, vol. 6, no. 3, pp. 3–6, 1993.

[141] Goldberg, A. and Tarjan, R., "A new approach to the maximum-flow problem," *Journal of the ACM*, vol. 35, no. 4, pp. 921–940, 1988.

[142] Goldberg, A. and Tarjan, R., "Finding minimum-cost circulations by canceling negative cycles," *Journal of the Association for Computing Machinery*, vol. 36, no. 4, pp. 873–886, 1989.

[143] Goldberg, A. and Tarjan, R., "Solving minimum-cost flow problems by successive approximation," *Mathematics of Operations Research*, vol. 15, no. 3, pp. 430–466, 1990.

[144] Golden, B., "A minimum-cost multicommodity network flow problem concerning imports and exports," *Networks*, vol. 5, pp. 331–356, Oct 1975.

[145] Goldfarb, D., Hao, J., and Kai, S., "Efficient shortest path simplex algorithms," *Operations Research*, vol. 38, no. 4, pp. 624–628, 1990.

[146] Goldfarb, D., Hao, J., and Kai, S., "Shortest path algorithms using dynamic breadth-first search," *Networks*, vol. 21, no. 1, pp. 29–50, 1991.

[147] Goldfarb, D. and Idnani, A., "A numerically stable dual method for solving strictly convex quadratic programs," *Mathematical Programming*, vol. 27, pp. 1–33, Sep 1983.

[148] Goldfarb, D. and Jin, Z., "An o(nm)-time network simplex algorithm for the shortest path problem," *Operations Research*, vol. 47, no. 3, pp. 445–448, 1999.

[149] Gopalakrishnan, B., *Least-Squares Methods in Linear Programming*. PhD thesis, School of Industrial and Systems Engineering, Georgia Institute of Technology, 2002.

[150] Goto, S., Ohtsuki, T., and Yoshimura, T., "Sparse matrix techniques for the shortest path problem," *IEEE Transactions on Circuits and Systems*, vol. CAS-23, pp. 752–758, Dec 1976.

[151] Goto, S. and Sangiovanni-Vincentelli, A., "A new shortest path updating algorithm," *Networks*, vol. 8, no. 4, pp. 341–372, 1978.

[152] Gouveia, L., "Multicommodity flow models for spanning trees with hop constraints," *European Journal of Operational Research*, vol. 95, no. 1, pp. 178–190, 1996.

[153] Gratzer, F. and Steiglitz, K., "A heuristic approach to large multicommodity flow problems," in *22nd international symposium on computer-communications networks and teletraffic*, pp. 311–324, 1972.

[154] Graves, G. and McBride, R., "The factorization approach to large-scale linear programming," *Mathematical Programming*, vol. 10, no. 1, pp. 91–110, 1976.

[155] Grigoriadis, M. D. and White, W. W., "Computational experience with a multicommodity network flow algorithm," in *Optimization methods for resource allocation (Proc. NATO Conf., Elsinore)* (Cottle, R. and Krarup, J., eds.), pp. 205–226, English Univsity Press, London, 1972.

[156] Grigoriadis, M. D. and White, W. W., "A partitioning algorithm for the multicommodity network flow problem," *Mathematical Programming*, vol. 3, no. 3, pp. 157–177, 1972.

[157] Grigoriadis, M. and Khachiyan, L., "Fast approximation schemes for convex programs with many blocks and coupling constraints," *SIAM Journal on Optimization*, vol. 4, no. 1, pp. 86–107, 1994.

[158] Grigoriadis, M. and Khachiyan, L., "An exponential-function reduction method for block-angular convex programs," *Networks*, vol. 26, no. 2, pp. 59–68, 1995.

[159] Grigoriadis, M. and Khachiyan, L., "Approximate minimum-cost multicommodity flows in $\tilde{O}(\epsilon^{-2}KNM)$ time," *Mathematical Programming*, vol. 75, no. 3, Ser. A, pp. 477–482, 1996.

[160] Grigoriadis, M. and Khachiyan, L., "Coordination complexity of parallel price-directive decomposition," *Mathematics of Operations Research*, vol. 21, no. 2, pp. 321–340, 1996.

[161] Grinold, R., "Steepest ascent for large scale linear programs," *SIAM Review*, vol. 14, no. 3, pp. 447–464, 1972.

[162] Grötschel, M., Lovász, L., and Schrijver, A., *Geometric algorithms and combinatorial optimization*. Berlin: Springer-Verlag, second ed., 1993.

[163] Hadjiat, M., Maurras, J.-F., and Vaxès, Y., "A primal partitioning approach for single and non-simultaneous multicommodity flow problems," *European Journal of Operational Research*, vol. 123, pp. 382–393, Jun 2000.

[164] Haghani, A. and Oh, S.-C., "Formulation and solution of a multi-commodity, multi-modal network flow model for disaster relief operations," *Transportation research. Part A, Policy and practice.*, vol. 30A, pp. 231–250, May 1996.

[165] Hane, C., "Personal communication," 2001.

[166] Hane, C., Barnhart, C., Johnson, E., Marsten, R., Nemhauser, G., and Sigismondi, G., "The fleet assignment problem: solving a large-scale integer program," *Mathematical Programming*, vol. 70, pp. 211–232, 1995.

[167] Hartman, J. and Lasdon, L., "A generalized upper bounding algorithm for multicommodity network flow problems," *Networks*, vol. 1, no. 4, pp. 333–354, 1972.

[168] Held, M., Wolfe, P., and Crowder, H., "Validation of subgradient optimization," *Mathematical Programming*, vol. 6, no. 1, pp. 62–88, 1974.

[169] Hu, J. and Johnson, E., "Computational results with a primal-dual subproblem simplex method," *Operations Research Letters*, vol. 25, pp. 149–157, 1999.

[170] Hu, T., "Multi-commodity network flows," *Operations Research*, vol. 11, pp. 344–360, 1963.

[171] Hu, T., "Revised matrix algorithms for shortest paths," *SIAM Journal of Applied Mathematics*, vol. 15, pp. 207–218, January 1967.

[172] Hu, T., "A decomposition algorithm for shortest paths in a network," *Operations Research*, vol. 16, pp. 91–102, 1968.

[173] Hung, M. and Divoky, J., "A computational study of efficient shortest path algorithms," *Computers and Operations Research*, vol. 15, no. 6, pp. 567–576, 1988.

[174] IATA, *Air Cargo Annual, A Statistical Overview of the Market in 1998*. IATA, 1999.

[175] IATA, *Asia Pacific Air Transport Forecast 1980-2010*. IATA, 1999.

[176] IATA, *Freight Forecast 1999-2003*. IATA, 1999.

[177] IATA, *World Air Transport Statistics*. IATA, 1999.

[178] Ibaraki, S., Fujishima, M., and Ibaraki, T., "Primal-dual proximal point algorithm for linearly constrained convex programming problems," *Computational Optimization and Applications*, vol. 1, no. 2, pp. 207–226, 1992.

[179] Ibaraki, S. and Fukushima, M., "Primal-dual proximal point algorithm for multicommodity network flow problems," *Journal of the Operations Research Society of Japan*, vol. 37, pp. 297–309, Dec 1994.

[180] Jewell, W., "Warehousing and distribution of a seasonal product," *Naval Research Logistics Quarterly*, vol. 4, no. 4, pp. 29–34, 1957.

[181] Jewell, W., *Optimal flow through networks*. PhD thesis, Interim Tech. Report No. 8, OR Center, M.I.T., Cambridge, MA, Jun 1958.

[182] Jewell, W., "A primal-dual multicommoidyt flow algorithm," Tech. Rep. ORC 66-24, Operations Research Center, University of California, Berkeley, 1966.

[183] Johnson, D., "Efficient algorithms for shortest paths in sparse networks," *Journal of ACM*, vol. 24, no. 1, pp. 1–13, 1977.

[184] Johnson, D., "A priority queue in which initialization and queue operations take $o(loglogd)$ time," *Mathematical Systems Theory*, vol. 15, no. 4, pp. 295–309, 1982.

[185] Johnson, D. and McGeoch, C., *Network Flows and Matching: First DIMACS Implementation Challenge*. AMS, Providence, 1993.

[186] Johnson, E., "On shortest paths and sorting," in *Proceedings of the ACM 25th annual conference*, pp. 510–517, 1972.

[187] Jones, K., Lustig, I., Farvolden, J., and Powell, W., "Multicommodity network flows: the impact of formulation on decomposition," *Mathematical Programming*, vol. 62, no. 1, pp. 95–117, 1993.

[188] Kallio, M. and Ruszczyński, A., "Parallel solution of linear programs via nash equilibria," tech. rep., WP-94-015, IIASA, Laxenburg, Austria, March 1994.

[189] Kamath, A. and Palmon, O., "Improved interior point algorithms for exact and approximate solution of multicommodity flow problems," in *Proceedings of the sixth annual ACM-SIAM Symposium on Discrete Algorithms (San Francisco, CA, 1995)*, pp. 502–511, ACM, SIAM, ACM, SIAM, 1995.

[190] Kamath, A., Palmon, O., and Plotkin, S., "Fast approximation algorithm for minimum cost multicommodity flow," in *Proceedings of the sixth annual ACM-SIAM symposium on Discrete algorithms*, (New York, NY, USA), pp. 493–501, ACM Press, 1995.

[191] Kapoor, S. and Vaidya, P., "Speeding up karmarkar's algorithm for multicommodity flows," *Mathematical Programming*, vol. 73, no. 1, Ser. A, pp. 111–127, 1996.

[192] Karger, D. and Plotkin, S., "Adding multiple cost constraints to combinatorial optimization problems, with applications to multicommodity flows," in *Proceedings of the Twenty-Seventh Annual ACM Symposium on the Theory of Computing*, (New Yourk, NY, USA), pp. 18–25, ACM, ACM Press, 1995.

[193] Karmarkar, N., "A new polynomial-time algorithm for linear programming," *Combinatorica*, vol. 4, no. 4, pp. 373–395, 1984.

[194] Karp, R., "On the computational complexity of combinatorial problems," *Networks*, vol. 5, no. 1, pp. 45–68, 1975.

[195] Karypis, G. and Kumar, V., "Metis, a software package for partitioning unstructured graphs, partitioning meshes, and computing fill-reducing orderings of sparse matrices." http://www-users.cs.umn.edu/ karypis/metis/metis/download.html, 1998.

[196] Karypis, G. and Kumar, V., "A fast and high quality multilevel scheme for partitioning irregular graphs," *SIAM Journal on Scientific Computing*, vol. 20, no. 1, pp. 359–392, 1999.

[197] Kazanov, A., "Minimum cost multiflows in undirected networks," *Mathematical Programming*, vol. 66, no. 3, Ser. A, pp. 313–325, 1994.

[198] Kazanov, A., "Multiflows and disjoint paths of minimum total cost," *Mathematical Programming*, vol. 78, no. 2, Ser. B, pp. 219–242, 1997.

[199] Kennington, J. and Shalaby, M., "An effective subgradient procedure for minimal cost multicommodity flow problems," *Management science*, vol. 23, pp. 994–1004, May 1977.

[200] Kennington, J., "Solving multicommodity transportation problems using a primal partitioning simplex technique," *Naval Research Logistics Quarterly*, vol. 24, pp. 309–325, Jun 1977.

[201] Kennington, J., "A survey of linear cost multicommodity network flows," *Operations Research*, vol. 26, no. 2, pp. 209–236, 1978.

[202] Kennington, J. and Helgason, R., *Algorithms for Network Programming*. New York: Wiley-Interscience, 1980.

[203] Kershenbaum, A., "A note on finding shortest paths trees," *Networks*, vol. 11, no. 4, pp. 399–400, 1981.

[204] Klein, P., Agrawal, A., Ravi, R., and Rao, S., "Approximation through multicommodity flow," in *Proceedings of the 31th annual IEEE Symposium on Foundations of Computer Science*, vol. 2, pp. 726–737, IEEE, 1990.

[205] Klein, P., Plotkin, S., Stein, C., and Tardos, É., "Faster approximation algorithms for the unit capacity concurrent flow problem with applications to routing and finding sparse cuts," *SIAM Journal on Computing*, vol. 23, no. 3, pp. 466–487, 1994.

[206] Klein, P., Rao, S., Agrawal, A., and Ravi, R., "An approximate max-flow min-cut relation for undirected multicommodity flow, with applications," *Combinatorica*, vol. 15, no. 2, pp. 187–202, 1995.

[207] Klingman, D., Napier, A., and Stutz, J., "Netgen: a program for generating large scale capacitated assignment, transportation and minimum cost flow network problems," *Management Science*, vol. 20, pp. 814–821, Jan 1974.

[208] Könemann, J., "Faster and simpler algorithms for multicommodity flow and other fractional packing problems," Master's thesis, Computer Science, Universität des Saarlandes, Saarbrücken, Germany, 1998.

[209] Kontogiorgis, S., *Alternating directions methods for the parallel solution of large-scale block-structured optimization problems.* PhD thesis, Computer Science, University of Wisconsin-Madison, 1995.

[210] Kwon, O., Martland, C., and Sussman, J., "Routing and scheduling temporal and heterogeneous freight car traffic on rail networks," *Transportation Research Part E: Logistics and Transportation Review*, vol. 34, pp. 101–115, Jun 1998.

[211] Land, A. and Stairs, S., "The extension of the cascade algorithm to large graphs," *Management Science*, vol. 14, pp. 29–33, 1967.

[212] Larsen, J. and Pedersen, I., "Experiments with the auction algorithm for the shortest path problem," *Nordic Journal of Computing*, vol. 6, no. 4, pp. 403–421, 1999.

[213] Lasdon, S., *Optimization theory for large systems.* New York: MacMillan, 1970.

[214] Lawson, C. and Hanson, R., *Solving Least-Squares Problems.* Englewood Cliffs, N.J.: Prentice-Hall, 1974.

[215] LeBlanc, L., Morlok, E., and Pierskalla, W., "An efficient approach to solving the road network equilibrium traffic assignment problem," *Transportation Research*, vol. 9, pp. 309–318, Oct 1975.

[216] Leichner, S., Dantzig, G., and Davis, J., "A strictly improving linear programming phase i algorithm," *Annals of Operations Research*, vol. 46-47, pp. 409–430, Dec 1993.

[217] Leighton, F. and Rao, S., "An approximate max-flow min-cut theorem for uniform multicommodity flow problems with applications to approximation algorithms," in *Proceedings of the 29th annual IEEE Symposium on Foundations of Computer Science*, pp. 422–431, IEEE, 1988.

[218] Leighton, T., Makedon, F., Plotkin, S., Stein, C., Tardos, É., and Tragoudas, S., "Fast approximation algorithms for multicommodity flow problems," *Journal of Computer and System Sciences*, vol. 50, pp. 228–243, Apr 1995.

[219] Leighton, T. and Rao, S., "Multicommodity max-flow min-cut theorems and their use in designing approximation algorithms," *Journal of the ACM*, vol. 46, pp. 787–832, Nov 1999.

[220] Levit, B. and Livshits, B., "Neleneinye setevye transportnye zadachi," *Transport*, p. in Russian, 1992.

[221] Lin, F. and Yee, J., "A new multiplier adjustment procedure for the distributed computation of routing assignments in virtual circuit data networks," *ORSA Journal on Computing*, vol. 4, no. 3, pp. 250–266, 1992.

[222] Lin, S.-Y. and Lin, C, H., "A computationally efficient method for nonlinear multicommodity network flow," *Networks*, vol. 29, no. 4, pp. 225–244, 1997.

[223] Liu, J. W., "Modification of the minimum degree algorithm by multiple elimination," *ACM Trans. Math. Software*, vol. 11, pp. 141–153, 1985.

[224] Lomonosov, M., "Combinatorial approaches to multiflow problems," *Discrete Applied Mathematics*, vol. 11, no. 1, pp. 1–93, 1985.

[225] Lustig, I. and Rothberg, E., "Gigaflops in linear programming," *Operations Research Letters*, vol. 18, pp. 157–165, Feb 1996.

[226] Lustig, I.J.and Marsten, R. and Shanno, D., "Computational experience with a primal-dual interior point method for linear programming," *Linear Algebra and its Applications*, vol. 152, pp. 191–222, 1991.

[227] Mahey, P., Ouorou, A., LeBlanc, L., and Chifflet, J., "A new proximal decomposition algorithm for routing in telecommunication networks," *Networks*, vol. 31, pp. 227–238, Jul 1998.

[228] Maier, S., "A compact inverse scheme applied to a multicommodity network with resource constraints," in *Optimization methods for resource allocation (Proc. NATO Conf., Elsinore)* (Cottle, R. and Krarup, J., eds.), pp. 179–203, English Univsity Press, London, 1972.

[229] Mamer, J. and McBride, R., "A decomposition-based pricing procedure for large-scale linear programs: an application to the linear multicommodity flow problem," *Management Science*, vol. 46, pp. 693–709, May 2000.

[230] Markowitz, H., "The elimination form of the inverse and its application to linear programming," *Management Science*, vol. 3, pp. 255–269, April 1957.

[231] Marsten, R., Subramanian, R., Saltzman, M., Lustig, I., and Shanno, D., "Interior point methods for linear programming: just call newton, lagrange, and fiacco and mccormick!," *Interfaces*, vol. 20, no. 4, pp. 105–116, 1990.

[232] Maurras, J.-F. and Vaxès, Y., "Multicommodity network flow with jump constraints," *Discrete Mathematics*, vol. 165-166, pp. 481–486, 1997.

[233] McBride, R., "Solving embedded generalized network problems," *European Journal of Operational Research*, vol. 21, no. 1, pp. 82–92, 1985.

[234] McBride, R., "Advances in solving the multicommodity-flow problem," *Interfaces*, vol. 28, no. 2, pp. 32–41, 1998.

[235] McBride, R., "Progress made in solving the multicommodity flow problem," *SIAM Journal on Optimization*, vol. 8, pp. 947–955, Nov. 1998.

[236] McBride, R. and Mamer, J., "Solving multicommodity flow problems with a primal embedded network simplex algorithm," *INFORMS Journal on Computing*, vol. 9, no. 2, pp. 154–163, 1997.

[237] McBride, R. and Mamer, J., "Solving the undirected multicommodity flow problem using a shortest path-based pricing algorithm," *Networks*, vol. 38, pp. 181–188, Dec. 2001.

[238] McCallum, C.J., J., "A generalized upper bounding approach to a communications network planning problem," *Networks*, vol. 7, no. 1, pp. 1–23, 1977.

[239] Meyer, R. and Zakeri, G., "Multicoordination methods for solving convex block-angular programs," *SIAM Journal on Optimization*, vol. 10, no. 1, pp. 121–131, 1999.

[240] Mills, G., "A decomposition algorithm for the shortest-route problem," *Operations Research*, vol. 14, pp. 279–291, 1966.

[241] Mondou, J., Crainic, T., and Nguyen, S., "Shortest path algorithms: a computational study with the c programming language," *Computers and Operations Research*, vol. 18, pp. 767–786, 1991.

[242] Monteiro, D. and Adler, I., "Interior path following primal-dual algorithms. part i: Linear programming.," *Mathematical Programming*, vol. 44, no. 1, Ser. A, pp. 27–41, 1989.

[243] Moore, E., "The shortest path through a maze," in *Proceedings of the International Symposium on theory of Switching , Part II*, pp. 285–292, The Annals of the Computation Laboratory of Harvard University, 1957.

[244] Morris, J., "An escalator process for the solution of linear simultaneous equations," *Philos. Mag. (7)*, vol. 37, pp. 106–120, 1946.

[245] Murray, S., *An interior point approach to the generalized flow problem with costs and related problems*. PhD thesis, Department of Operations Research, Stanford University, Aug 1992.

[246] Murtagh, B. and Saunders, M., "Large-scale linearly constrained optimization," *Mathematical Programming*, vol. 14, no. 1, pp. 41–72, 1978.

[247] Murtagh, B. and Saunders, M., "Minos 5.4 release notes, appendix to minos 5.1 user's guide," tech. rep., Stanford University, 1992.

[248] Muthukrishnan, S. and Suel, T., "Second-order methods for distributed approximate single- and multicommodity flow," in *Randomization and Approximation Techniques in Computer Science. Second International Workshop, RANDOM'98. Proceedings* (Luby, M., Rolim, J., and Serna, M., eds.), vol. 1518, (Berlin), pp. 369–383, Springer, 1998.

[249] Nagamochi, H., Fukushima, M., and Ibaraki, T., "Relaxation methods for the strictly convex multicommodity flow problem with capacity constraints on individual commodities," *Networks*, vol. 20, pp. 409–426, Jul 1990.

[250] Nagamochi, H. and Ibaraki, T., "On max-flow min-cut and integral flow properties for multicommodity flows in directed networks," *Information Processing Letters*, vol. 31, pp. 279–285, Jun 1989.

[251] Nakamori, M., "A note on the optimality of some all-shortest-path algorithms," *Journal of the Operations Research Society of Japan*, vol. 15, pp. 201–204, December 1972.

[252] Nemhauser, G., "A generalized permanent label setting algorithm for the shortest path between specified nodes," *Journal of Mathematical Analysis and Applications*, vol. 38, pp. 328–334, 1972.

[253] Nguyen, S., Pallottino, S., and Scutellà, M., "A new dual algorithm for shortest path reoptimization," in *Transportation and Network Analysis - Current Trends* (Gendreau, M. and Marcotte, P., eds.), Kluwer Academic Publishers, 2001.

[254] Okamura, H., "Multicommodity flows in graphs," *Discrete Applied Mathematics*, vol. 6, pp. 55–62, May 1983.

[255] Okamura, H. and Seymour, P., "Multicommodity flows in planar graphs," *Journal of Combinatorial Theory*, vol. 31, no. 1, Ser. B, pp. 75–81, 1981.

[256] Oldham, J., *Multicommodity and generalized flow algorithms: theory and practice.* PhD thesis, Department of Computer Science, Stanford University, 1999.

[257] Ouorou, A., "A primal-dual algorithm for monotropic programming and its application to network optimization," *Computational Optimization and Applications*, vol. 15, pp. 125–143, Feb 2000.

[258] Ouorou, A. and Mahey, P., "A minimum mean cycle cancelling method for nonlinear multicommodity flow problems," *European Journal of Operational Research*, vol. 121, pp. 532–548, Mar 2000.

[259] Ouorou, A., Mahey, P., and Vial, J.-P., "A survey of algorithms for convex multicommodity flow problems," *Management Science*, vol. 46, pp. 126–147, Jan 2000.

[260] Pallottino, S., "Shortest-path methods: compelxity, interrelations and new propositions," *Networks*, vol. 14, pp. 257–267, 1984.

[261] Pallottino, S. and Scutellà, M., "Dual algorithms for the shortest path tree problem," *Networks*, vol. 29, no. 2, pp. 125–133, 1997.

[262] Palmon, O., *Optimization issues in network routing.* PhD thesis, Department of Computer Science, Stanford University, Jun 2001.

[263] Papadimitriou, C. and Steiglitz, K., *Combinatorial optimization: algorithms and complexity.* Englewood Cliffs, NJ: Prentice-Hall, 1982.

[264] Pape, U., "Implementation and efficiency of moore algorithms for the shortest root problem," *Mathematical Programming*, vol. 7, pp. 212–222, 1974.

[265] Pettie, S. and Ramachandran, V., "Computing shortest paths with comparisons and additions," utcs tr-01-12, Department of Computer Science, The University of Texas at Austin, 2001.

[266] Pinar, M. and Zenios, S., "Parallel decomposition of multicommodity network flows using a linear-quadratic penalty algorithm," *ORSA Journal on Computing*, vol. 4, no. 3, pp. 235–249, 1992.

[267] Plotkin, S., Shmoys, D., and Tardos, É., "Fast approximation algorithms for fractional packing and covering problems," *Mathematics of Operations Research*, vol. 20, pp. 257–301, May 1995.

[268] Polak, G., "On a parametric shortest path problem from primal-dual multicommodity network optimization," *Networks*, vol. 22, pp. 283–295, May 1992.

[269] Radzik, T., "Fast deterministic approximation for the multicommodity flow problem," *Mathematical Programming*, vol. 78, no. 1, Ser. A, pp. 43–58, 1995.

[270] Raghavan, P., "Integer programming in vlsi design," *Discrete Applied Mathematics*, vol. 40, pp. 29–43, Nov 1992.

[271] Rockafellar, R., *Network flows and monotropic optimization*. New York, NY: John Wiley, 1984.

[272] Rose, D. and Tarjan, R., "Algorithmic aspects of vertex elimination on directed graphs.," *SIAM Journal on Applied Mathematics*, vol. 34, no. 1, pp. 176–197, 1978.

[273] Rosen, J., "Primal partition programming for block diagonal matrices," *Numerische Mathematik*, vol. 6, pp. 250–260, 1964.

[274] Rote, G., "A systolic array algorithm for the algebraic path problem," *Computing*, vol. 34, no. 3, pp. 191–219, 1985.

[275] Rote, G., "Path problems in graphs," in *Computational graph theory*, pp. 155–189, Vienna: Springer, 1990.

[276] Rothschild, B. and Whinston, A., "On two commodity network flows," *Operations Research*, vol. 14, pp. 377–387, 1966.

[277] Saigal, R., "Multicommodity flows in directed networks," Tech. Rep. ORC Report 66-25, Operations Research Center, University of California, Berkeley, 1967.

[278] Sakarovitch, M., "Two commodity network flows and linear programming," *Mathematical Programming*, vol. 4, no. 1, pp. 1–20, 1973.

[279] Sarrafzadeh, M. and Wong, C., *An introduction to VLSI physical design*. New York: McGraw Hill, 1996.

[280] Schneur, R., *Scaling algorithms for multicommodity flow problems and network flow problems with side constraints*. PhD thesis, M.I.T., Cambridge, MA, 1991.

[281] Schneur, R. and Orlin, J., "A scaling algorithm for multicommodity flow problems," *Operations Research*, vol. 146, no. 2, pp. 231–246, 1998.

[282] Schrijver, A., "Short proofs on multicommodity flows and cuts," *Journal of Combinatorial Theory*, vol. 53, no. 1, Ser. B, pp. 32–39, 1991.

[283] Schultz, G. and Meyer, R., "An interior point method for block angular optimization," *SIAM Journal on Optimization*, vol. 1, no. 4, pp. 583–602, 1991.

[284] Seidel, R., "On the all-pairs-shortest-path problem in unweighted undirected graphs," *Journal of Computer and System Sciences*, vol. 51, pp. 400–403, December 1995.

[285] Sensen, N., "Lower bounds and exact algorithms for the graph partitioning problem using multicommodity flows," in *Algorithms-ESA 2001 : 9th annual European symposium, Aarhus, Denmark, August 28-31, 2001 : proceedings* (auf der Heide, F. M., ed.), vol. 2161, (Berlin Heidelberg), pp. 391–403, Springer, Aug 2001.

[286] Seymour, P., "Matroids and multicommodity flows," *European Journal of Combinatorics*, vol. 2, no. 3, pp. 257–290, 1981.

[287] Shahrokhi, F. and Matula, D., "The maximum concurrent flow problem," *Journal of the ACM*, vol. 37, pp. 318–334, Apr 1990.

[288] Shepherd, B. and Zhang, L., "A cycle augmentation algorithm for minimum cost multicommodity flows on a ring," *Discrete Applied Mathematics*, vol. 110, no. 2-3, pp. 301–315, 2001.

[289] Shetty, B. and Muthukrishnan, R., "A parallel projection for the multicommodity network model," *Journal of the Operational Research Society*, vol. 41, pp. 837–842, Sep 1990.

[290] Shier, D. and Witzgall, C., "Properties of labeling methods for determining shortest paths trees," *Journal of Research of the National Bureau of Standards*, vol. 86, no. 3, pp. 317–330, 1981.

[291] Shimbel, A., "Applications of matrix algebra to communication nets," *Bulletin of Mathematical Biophysics*, vol. 13, pp. 165–178, 1951.

[292] Soun, Y. and Truemper, K., "Single commodity representation of multicommodity networks," *SIAM Journal on Algebraic and Discrete Methods*, vol. 1, no. 3, pp. 348–358, 1980.

[293] Takaoka, T., "A new upper bound on the complexity of the all pairs shortest path problem," *Information Processing Letters*, vol. 43, no. 4, pp. 195–199, 1992.

[294] Takaoka, T., "Subcubic cost algorithms for the all pairs shortest path problem," *Algorithmica*, vol. 20, no. 3, pp. 309–318, 1998.

[295] Thorup, M., "Undirected single-source shortest paths with positive integer weights in linear time," *Jounral of ACM*, vol. 46, no. 3, pp. 362–394, 1999.

[296] Thorup, M., "Floats, integers, and single source shortest paths," *Journal of Algorithms*, vol. 35, no. 2, pp. 189–201, 2000.

[297] Tomlin, J., "Minimum-cost multicommodity network flows," *Operations Research*, vol. 14, no. 1, pp. 45–51, 1966.

[298] Truemper, K., "Unimodular matrices of flow problems with additional constraints," *Networks*, vol. 7, no. 4, pp. 343–358, 1977.

[299] Truemper, K. and Soun, Y., "Minimal forbidden subgraphs of unimodular multicommodity networks," *Mathematics of Operations Research*, vol. 4, no. 4, pp. 379–389, 1979.

[300] Vaidya, P., "Speeding-up linear programming using fast matrix multiplication," in *30th Annual Symposium on Foundations of Computer Science*, pp. 332–337, IEEE, 1989.

[301] Van de Panne, C. and Whinston, A., "The symmetric foundation of the simplex method for quadratic programming," *Econometrica*, vol. 37, pp. 507–527, 1969.

[302] Vance, P., Barnhart, C., Johnson, E., and Nemhauser, G., "Solving binary cutting stock problems by column generation and branch-and-bound," *Computational Optimization and Applications*, vol. 3, no. 2, pp. 111–130, 1994.

[303] Wardrop, J., "Some theoretical aspects of road traffic research," in *Proceeding of the Institute of Civil Engineers, Part II*, vol. 1, pp. 325–378, 1952.

[304] Warshall, S., "A theorem on boolean matrics," *Journal of ACM*, vol. 9, pp. 11–12, 1962.

[305] Yamakawa, E., Matsubara, Y., and Fukushima, M., "A parallel primal-dual interior point method for multicommodity flow problems with quadratic costs," *Journal of the Operations Research Society of Japan*, vol. 39, pp. 566–591, Dec 1996.

[306] Yang, L. and Chen, W., "An extension of the revised matrix algorithm," in *IEEE international Symposium on Circuits and Systems*, (Portland, Oregon), pp. 1996–1999, IEEE, May 1989.

[307] Ye, Y., "Toward probabilistic analysis of interior-point algorithms for linear programming," *Mathematics of Operations Research*, vol. 19, no. 1, pp. 38–52, 1994.

[308] Yen, J., "An algorithm for finding shortest routes from all source nodes to a given destination in general networks," *Quarterly of Applied Mathematics*, vol. 27, pp. 526–530, 1970.

[309] Young, N., "Randomized rounding without solving the linear program," in *Proceedings of the sixth annual ACM-SIAM symposium on Discrete algorithms*, (New Your, NY, USA), pp. 170–178, ACM Press, 1995.

[310] Zakeri, G., *Multi-coordination methods for parallel solution of block-angular programs.* PhD thesis, Computer Science, University of Wisconsin, May 1995.

[311] Zenios, S., Pinar, M., and Dembo, R., "A smooth penalty function algorithm for network-structured problems," *European Journal of Operational Research*, vol. 83, pp. 229–236, May 1995.

[312] Zhan, F. and Noon, C., "Shortest path algorithms: an evaluation using real road networks," *Transportation Science*, vol. 32, pp. 65–73, February 1998.

[313] Zwick, U., "All pairs shortest pahts in weighted directed graphs - exact and almost exact algorithms," in *Proceedings of the 39th Annual IEEE Symposium on Foundations of Computer Science*, (Palo Alto, California), pp. 310–319, November 1998.

VITA

I-Lin Wang was born and raised in Tainan city, Taiwan, on August 29, 1969. He attended Cheng Kung Elementary school, Li Ming private Junior High school, Tainan First High School, and National Cheng Kung University, where he majored in Aeronautical and Astronautical Engineering.

I-Lin lived in Tainan city until he was 22 years of age, he then served in the Air Force for 2 years at the Aero-Industry-Development-Center in Taichung City. There he was an SAS programmer for Integrated Logistics Support. He had been quite interested in astronomy and planned to study Orbital Mechanics for his masters at the Department of Aeronautical and Astronautical Engineering at M.I.T.

Before going to M.I.T. in 1993, his Father died of liver cancer. After that he decided to change his major, and successfully transferred to the OR center of M.I.T., there he learned Operations Research. After obtaining his masters degree in O.R. from M.I.T., he found a foreign research position at the Fujitsu Research Lab at Kawasaki, Japan via the M.I.T. Japanese Program. His job was to design and implement wavelength assignment and routing algorithms for optical networks.

After one year in Japan, he went to Stanford University and later decided to continue his Ph.D. at Georgia Tech's School of Industrial and Systems Engineering (ISyE), beginning fall 1998. He passed the comprehensive exam in 9 months and worked with Professor Ellis Johnson on the Air Cargo projects of TLI-Asia Pacific. He has been doing research in network optimization.

I-Lin married Hsin-Yi Yao in the summer of 1999. On September 29, 2000, their first son Chun-Han Wang was born. During I-Lin's four and half years at ISyE, he spent about one year in Singapore. He passed his thesis defense on January 21, 2003, and will receive his Ph.D. degree in May, 2003. His plans are to seek a faculty position at a university in Taiwan.