Combining Shortest Paths, Bottleneck Paths and Matrix Multiplication

---

A thesis
submitted in partial fulfilment
of the requirements for the Degree
of
Doctor of Philosophy
in the
University of Canterbury
by
Tong-Wook Shinn

University of Canterbury
2014

To my parents, my wife and my children.

Acknowledgments

I thank my parents for bringing me into this world and raising me.

I thank my wife, Angela, for her loving support and being the best mum that our precious children could ask for.

I thank my primary supervisor, Prof. Takaoka, for his patience and guidance, without whom this thesis would never have eventuated.

I thank my colleague, Sung Bae, for being a great "Sun Bae"[1].

I thank the viva committee, especially Dr Guttmann, for the thorough review and constructive feedback that helped to polish up this thesis before submission.

I thank the examiners, especially Prof. Moffat, for spending the time to provide an incredibly in-depth review of the thesis.

[1] Korean for "senior colleague".

Abstract

We provide a formal mathematical definition of the Shortest Paths for All Flows (SP-AF) problem and provide many efficient algorithms. The SP-AF problem combines the well known Shortest Paths (SP) and Bottleneck Paths (BP) problems, and can be solved by utilising matrix multiplication. Thus in our research of the SP-AF problem, we also make a series of contributions to the underlying topics of the SP problem, the BP problem, and matrix multiplication.

For the topic of matrix multiplication we show that on an n-by-n two dimensional (2D) square mesh array, two n-by-n matrices can be multiplied in exactly $1.5n - 1$ communication steps. This halves the number of communication steps required by the well known Cannon's algorithm [10] that runs on the same sized mesh array.

We provide two contributions for the SP problem. Firstly, we enhance the breakthrough algorithm by Alon, Galil and Margalit (AGM) [4], which was the first algorithm to achieve a deeply sub-cubic time bound for solving the All Pairs Shortest Paths (APSP) problem on dense directed graphs. Our enhancement allows the algorithm by AGM to remain sub-cubic for larger upper bounds on integer edge costs. Secondly, we show that for graphs with n vertices, the APSP problem can be solved in exactly $3n - 2$ communication steps on an n-by-n 2D square mesh array. This improves on the previous result of $3.5n$ communication steps achieved by Takaoka and Umehara [72].

For the BP problem, we show that we can compute the bottleneck of the entire graph without solving the All Pairs Bottleneck Paths (APBP) problem, resulting in a much more efficient time bound.

Finally we define an algebraic structure called the distance/flow semi-ring to formally introduce the SP-AF problem, and we provide many algorithms for solving the Single Source SP-AF (SSSP-AF) problem and the All Pairs SP-AF (APSP-AF) problem. For the APSP-AF problem, algebraic algorithms are given that utilise faster matrix multiplication over a ring.

Table of Contents

Chapter 1: Introduction **1**

Chapter 2: Matrix Multiplication **5**

2.1 Faster Matrix Multiplication Over a Ring 6

2.2 Matrix Multiplication on a 2D Square Mesh Array 9

2.2.1 Definition of the 2D Square Mesh Array 9

2.2.2 Review of Cannon's Algorithm 9

2.2.3 Loading Data from Both Corners 13

2.2.4 Using Values from All Four Directions 18

Chapter 3: Shortest Paths (SP) **23**

3.1 Distance Semi-ring 25

3.2 Deeply Sub-cubic Time Complexity 27

3.3 Expansion and Contraction of Graphs 32

3.4 APSP Problem on the Mesh Array 35

3.4.1 Review of the Cascade Algorithm 36

3.4.2 Cascade Algorithm on the Mesh Array 43

Chapter 4: Bottleneck Paths (BP) **52**

4.1 Bottleneck Semi-ring 53

4.2 The Graph Bottleneck (GB) Problem 54

Chapter 5: Shortest Paths for All Flows (SP-AF) **56**

5.1 Distance/flow Semi-ring 58

5.2 Single Source SP-AF (SSSP-AF) 64

5.2.1 Unit Edge Costs 65

5.2.2 Integer Edge Costs 69

5.3 All Pairs SP-AF (APSP-AF) 76

5.3.1 Unit Edge Costs 77

5.3.2 Integer Edge Costs 80

Chapter 6: Conclusion **86**

Appendix A: Publications **89**

A.1 Conferences 89

A.2 Journals 91

References **92**

Chapter I

Introduction

The Shortest Paths (SP) problem is arguably one of the most widely studied problems in graph theory. Given a set of vertices and a set of edges with varying distances, the problem is to find the shortest paths between pairs of vertices.

A less well known, but still a widely studied problem, is the Bottleneck Paths (BP) problem. If edges have capacities, for any given path between vertices, there will obviously be a bottleneck given by the edge with the smallest capacity on the path. And hence the problem is to find the paths that would give us the maximum bottleneck values between pairs of vertices.

Matrix multiplication is one of the most fundamental mathematical operations, used as a tool to solve numerous problems in computer graphics, applied mathematics, physics and engineering, etc.

To understand how these seemingly unrelated topics are combined under a single graph path problem, let us consider a directed graph, $G = (V, E)$, such that V is the set of vertices and E is the set of edges. Let $|V| = n$ and $|E| = m$. We assume that vertices are numbered from 1 to n. Let $(i, j) \in E$ be the edge from vertex i to vertex j. Let $cost(i, j)$ and $cap(i, j)$ denote the cost and the capacity of (i, j), respectively. We use the terms 'cost' and 'distance' interchangeably. Note that the notations that have just been introduced will be used consistently throughout this thesis.

The SP problem is only concerned with the values of $cost(i, j)$, whereas the BP problem is only concerned with the values of $cap(i, j)$. If we were to solve the SP problem on the graph given in Figure 1.1 from vertex 1 to vertex 6, the shortest path would be via vertex 3 and 5, giving us the total path cost (or distance) of 4. On the other hand, if we were to solve the BP problem on the same graph from vertex 1 to vertex 6, the bottleneck path would be via vertex 3, 2, 4 and 5, giving us the bottleneck value of 7.

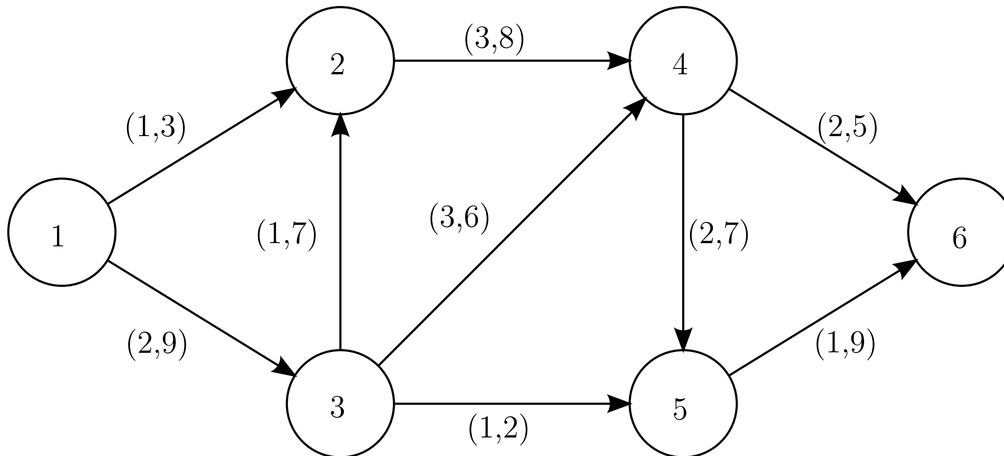

Figure 1.1: A example graph with with $n = 6$ and $m = 9$. The cost and capacity are shown beside each edge, respectively.

Let us now consider the shortest path that can flow a specific amount. In other words, the path has the bottleneck value of at least the required flow amount. Then intuitively, there may exist a shorter path that can only flow a smaller amount, and a longer path that can flow a larger amount. For example, on the graph given by Figure 1.1, if we wanted to flow 5 units from vertex 1 to vertex 6, the shortest path that can accommodate this flow amount is the path via vertices 3 and 4, giving us the total path cost of 7. If we only wanted to flow 3 units, however, then there exists a shorter path via vertices 2 and 4, giving us a lower path cost of 6.

If the flow requirements between vertices vary, then it is clearly beneficial to compute the shortest paths for all possible flow amounts, such that the shortest path can be chosen for any given flow amount. Thus we have the problem that is the main topic of this thesis, which we refer to as the Shortest Paths for All Flows (SP-AF) problem, that is concerned with both $cost(i, j)$ and $cap(i, j)$. The SP-AF problem can therefore be considered to be a combination of the SP and the BP problem, and we show that we can utilise matrix multiplication to solve this problem efficiently.

The main motivation for the SP-AF problem comes from its application in computer networking. If we model a computer network as a graph such that each router/host is represented as a vertex and each link is represented as

an edge, (i, j), then the bandwidth and latency of the link can be considered to be the capacity, $cap(i, j)$, and the cost of the edge, $cost(i, j)$, respectively. If the required data flow from a source host to a destination host can be predetermined, then it is clearly beneficial to find the path with the lowest total latency that has enough bandwidth for the required flow amount, such that the data can be transferred as quickly as possible without causing any network congestion. As we will discuss in later chapters, the SP-AF problem can be applied to other practical problems that can be modelled on a graph where each edge has the two properties, $cap(i, j)$ and $cost(i, j)$.

Even though the SP-AF problem is the main topic of this thesis, we were able to make a series of contributions to each of the three underlying topics. For the topic of matrix multiplication, in Chapter 2, we give a parallel distributed algorithm that halves the number of communication steps required by the well known Cannon's algorithm [10] on the same sized 2D mesh array. In Chapter 3, we make two contributions to the SP problem, firstly by enhancing the breakthrough algorithm by Alon *et al.* for solving the All Pairs Shortest Paths (APSP) problem [4], and secondly providing a parallel distributed algorithm on a mesh array to solve the APSP problem that is faster than the existing algorithm by Takaoka and Umehara [72]. In Chapter 4 we show that the bottleneck of the entire graph can be computed without solving the All Pairs Bottleneck Paths (APBP) problem, resulting in a very efficient time bound, thus also making a small contribution to the topic of BP problem.

Finally in Chapter 5, the SP-AF problem is presented as the main contribution of this thesis. We give a formal mathematical definition of the problem based on the theory of semi-rings and provide many algorithms that are faster than the straightforward methods of solving the various SP-AF problems.

There are many related problems to the aforementioned graph paths problems such as the minimum cost spanning tree problem on graphs with edge costs [53, 40], the maximum flow problem on graphs with edge capacities [23, 17, 45], and the minimum-cost flow problem on graphs with both edge costs and capacities [39, 27, 20, 28]. In-depth discussions of these related problems, however, are not part of this thesis.

Most graph paths problems discussed in this thesis can be based on the theory of semi-rings and the various representations of graphs as matrices. Condran and Minoux have discussed semi-rings in great depths alongside other algebraic structures such as fields, rings, dioids, etc. [29]. In our thesis, however, we use a more concise discussion of semi-rings by Aho, Hopcroft and Ullman [2]. The overall content of our thesis resembles the PhD thesis by Vassilevska [78] who utilised the semi-rings theory to provide algebraic algorithms for solving various graph paths problems, such as the shortest paths, maximum bottleneck paths and minimum non-decreasing paths problems.

A more in-depth review of relevant literatures for each topic is provided in the introduction of each chapter. A summary of all our contributions and the corresponding publications can be found in Chapter 6.

Chapter II

Matrix Multiplication

Let us make a start by defining what we mean by matrix multiplication. Let $X = \{x_{ij}\}$ be an n-by-n matrix[1] such that x_{ij} denotes the matrix element at row i and columnn j. Similarly, let $Y = \{y_{ij}\}$ and $Z = \{z_{ij}\}$ be n-by-n matrices. Then the matrix multiplication $Z = XY$ is given by:

$$z_{ij} = \sum_{k=1}^{n} x_{ik} y_{kj}$$

From this definition of matrix multiplication, it is clear that the operation can be performed in $O(n^3)$ time.

In 1969, Volker Strassen made an amazing breakthrough by showing that the product of two n-by-n matrices over a ring can be computed in $O(n^\omega)$ time bound, where $\omega < 2.808$ [65]. It is clear from the title of the paper "Gaussian elimination is not optimal" that Strassen was well aware of the magnitude of this discovery. To compute the value of one element, we must sum the products from n individual elements from a row of one matrix and the column of the other matrix. So how is it possible to break the $O(n^3)$ time barrier?

Before we provide an answer to this question in Section 2.1, let us discuss the parallelisation of matrix multiplication, as our contribution in this chapter actually comes in the form of a parallel algorithm. Parallelising matrix multiplication is quite straightforward as the matrices can be divided into rectangular sub-matrices of arbitrary sizes and the computation can be performed with the sub-matrices in parallel. For example, with n^2 processors, we can divide the matrices into n rows and n columns, such that each processor can compute the value of a single element in $O(n)$ time, giving us

[1] It is no co-incidence that we have defined the size of matrices to be n-by-n and the number of vertices in G is also denoted by n.

the total computational cost of $O(n^3)$ for this simple example of a parallel algorithm.

Lynn E. Cannon, also in 1969, gave a parallel distributed algorithm for matrix multiplication on a two dimensional (2D) square mesh array [10]. Mesh arrays are also commonly known as systolic arrays due to their rhythmical behaviour, and also as Very-Large-Scale-Integration (VLSI) circuits due to their ease of implementability onto Application Specific Integrated Circuits (ASIC) [41]. Cannon showed that with n^2 processors laid out in an n-by-n square 2D mesh array such that each processor can communicate with at most four neighbours, matrix multiplication can be performed in exactly $3n - 2$ communication steps between the processors.

Since then many different types of mesh arrays have been developed for matrix multiplication [12, 37, 47, 52]. In terms of the required number of communication steps, the most notable contributions were from Kak, who achieved $2n - 1$ communication steps [38], and Benaini and Robert, who achieved $1.5n$ communication steps [7]. Both of these achievements came as a result of a more complex 3D mesh array architecture, and in the case of the algorithm by Benaini and Robert, a more complex matrix multiplication method by Winograd was used [81].

In Section 2.2, we present our contribution to the topic of matrix multiplication by showing that the product of two n-by-n matrices can be computed in exactly $1.5n - 1$ communication steps on a strictly 2D square mesh array, effectively equalling the algorithm by Benaini and Robert in terms of the number of communication steps, but with two key advantages. Firstly, our mesh array definition is strictly 2D and simpler. Secondly, our algorithm itself is also simpler as it is based on Cannon's original algorithm rather than the more complex Winograd's algorithm.

2.1 Faster Matrix Multiplication Over a Ring

We abbreviate Faster Matrix Multiplication Over a Ring as FMMOR. Let X, Y and Z be n-by-n square matrices over a ring. We review Strassen's algorithm by using it to compute $Z = XY$. We ensure $n = 2^k$ for some integer $k > 0$ by adding rows and columns of zeroes if necessary. We divide

each matrix into four equal sized square matrices as follows:

$$X = \begin{bmatrix} X_{1,1} & X_{1,2} \\ X_{2,1} & X_{2,2} \end{bmatrix} Y = \begin{bmatrix} Y_{1,1} & Y_{1,2} \\ Y_{2,1} & Y_{2,2} \end{bmatrix} Z = \begin{bmatrix} Z_{1,1} & Z_{1,2} \\ Z_{2,1} & Z_{2,2} \end{bmatrix}$$

Then the product Z = XY can be computed as follows:

$$\begin{aligned} Z_{1,1} &= X_{1,1}Y_{1,1} + X_{1,2}Y_{2,1} \\ Z_{1,2} &= X_{1,1}Y_{1,2} + X_{1,2}Y_{2,2} \\ Z_{2,1} &= X_{2,1}Y_{1,1} + X_{2,2}Y_{2,1} \\ Z_{2,2} &= X_{2,1}Y_{1,2} + X_{2,2}Y_{2,2} \end{aligned}$$

However, since there are eight multiplications with $\frac{n}{2}$-by-$\frac{n}{2}$ matrices, the time complexity is still $O(n^3)$. But here comes the magical part with intermediate matrices $M_1, M_2, ..., M_7$:

$$\begin{aligned} M_1 &= (X_{1,1} + X_{2,2})(Y_{1,1} + Y_{2,2}) \\ M_2 &= (X_{2,1} + X_{2,2})Y_{1,1} \\ M_3 &= X_{1,1}(Y_{1,2} - Y_{2,2}) \\ M_4 &= X_{2,2}(Y_{2,1} - Y_{1,1}) \\ M_5 &= (X_{1,1} + X_{1,2})Y_{2,2} \\ M_6 &= (X_{2,1} - X_{1,1})(Y_{1,1} + Y_{1,2}) \\ M_7 &= (X_{1,2} - X_{2,2})(Y_{2,1} + Y_{2,2}) \end{aligned}$$

then the matrix Z can be computed as follows:

$$\begin{aligned} Z_{1,1} &= M_1 + M_4 - M_5 + M_7 \\ Z_{1,2} &= M_3 + M_5 \\ Z_{2,1} &= M_2 + M_4 \\ Z_{2,2} &= M_1 - M_2 + M_3 + M_6 \end{aligned}$$

Thus the number of multiplications with $\frac{n}{2}$-by-$\frac{n}{2}$ matrices has been reduced from eight to seven! By recursively using this method for the sub-matrices, the asymptotic time complexity becomes $O(n^\omega)$ where $\omega = \log_2 7 < 2.808$.

The most important thing to note, other than the amazing result itself, is that subtraction between matrices is performed in this algorithm. In other words, on a semi-ring where the inverse operation does not exist for the +

Year	ω	Author(s)
1969	2.808	Strassen [65]
1978	2.796	Pan [49]
1979	2.780	Bini, Capovani, Romani and Lotti [8]
1981	2.522	Schönhage [57]
1982	2.517	Romani [55]
1982	2.496	Coppersmith and Winograd [13]
1986	2.479	Strassen [66]
1990	2.376	Coppersmith and Winograd [14]
2010	2.374	Stothers [64]
2012	2.373	Williams [80]
2014	2.373	Le Gall [44]

Table 2.1: The exponent of the asymptotic time complexity of FMMOR.

operation, this algorithm will not work. This restriction is very important, as we will discover in later chapters of this thesis.

After the cubic barrier has been broken by Strassen, there have been many subsequent achievements in reducing the value of ω as shown in Table 2.1, at the expense of the algorithms getting more and more complex. Words such as "galactic" and "astronomical" are commonly used to describe some of the latter algorithms, which are widely known to be impractical to be implemented on modern day computers. The values of ω in Table 2.1 are rounded up to three decimal places. The last reduction to the value of ω was achieved by Le Gall [44], who firstly corrected William's result [80] from $\omega < 2.3727$ to $\omega < 2.37293$, then presented the new result of $\omega < 2.37286$

Throughout this thesis, we take the current best value of $\omega < 2.373$ to calculate the theoretical best worst-case time complexities of our algorithms that utilise FMMOR. Note that for all algorithms that utilise FMMOR, we can always revert back to the ordinary matrix multiplication method (or even Strassen's algorithm for larger matrices) to make the algorithm practical for implementation.

2.2 Matrix Multiplication on a 2D Square Mesh Array

Our contribution to the topic of matrix multiplication is essentially an enhancement to Cannon's parallel mesh array algorithm. We start by defining the 2D mesh array and show that Cannon's algorithm indeed takes exactly $3n-2$ communication steps to perform matrix multiplication. We then show how we can enhance this algorithm to achieve $1.5n-1$ steps, which is exactly half the number of steps required by Cannon's algorithm.

2.2.1 Definition of the 2D Square Mesh Array

Let $X = \{x_{ij}\}$ be an n-by-n matrix where x_{ij} denotes the element in row i and column j such that $1 \leq i, j \leq n$. Similarly let $Y = \{y_{ij}\}$ be an n-by-n matrix. Let $Z = XY$, that is, $Z = \{z_{ij}\}$ is the product of X and Y. We assume $x_{pq} = y_{pq} = 0$ for all $p, q < 0$ and $p, q > n$. We define 2D arrays $x[i][j]$ and $y[i][j]$ to store the values x_{ij} and y_{ij}, respectively.

We define a simple 2D square mesh array with n^2 cells (or processors), where each cell only communicates with its four neighbors, *Top*, *Bottom*, *Left* and *Right*. Let $cell(i, j)$ denote the cell at row i and column j, where $1 \leq i, j \leq n$.

Each cell has five registers, t, b, l, r and v, as shown in Figure 2.1. Let $t(i, j)$, $b(i, j)$, $l(i, j)$, $r(i, j)$ and $v(i, j)$ denote the five registers in $cell(i, j)$. Register t holds the data received from *Top* that is subsequently passed down. Register b holds the data received from *Bottom* that is subsequently passed up. Register l holds the data received from *Left* that is subsequently passed to the right. Register r holds the data received from *Right* that is subsequently passed to the left. Register $v(i, j)$ holds the partial sum for z_{ij}. We assume that all registers are initialized to 0.

2.2.2 Review of Cannon's Algorithm

Let the rectangular 2D array $L[p][q]$ where $1 \leq p \leq n$ and $1 \leq q \leq 2n$ hold the values for matrix $X = \{x_{ij}\}$ where the matrix is horizontally skewed and reversed. Let the rectangular 2D array $T[p][q]$ where $1 \leq p \leq 2n$ and $1 \leq q \leq n$ hold the values for matrix $Y = \{y_{ij}\}$ where the matrix is ver-

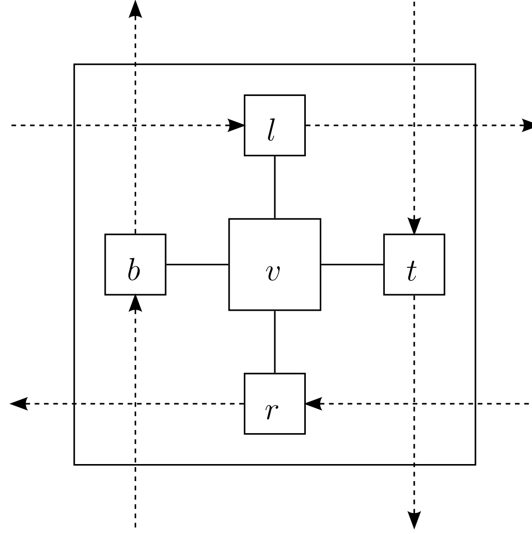

Figure 2.1: The structure of a cell in the 2D mesh array.

tically skewed and reversed. We refer to L and T as *loader* arrays because they hold the elements of the original matrices to be loaded onto the mesh array. These loader arrays are not essential for actual implementation of the parallel algorithms, but we use them to provide additional clarity. Algorithm 1 shows exactly how matrices X and Y are skewed and reversed in L and T, respectively. As mentioned in Section 2.2.1, $x[p][q] = 0$ and $y[p][q] = 0$ for $p, q < 0$ and/or $p, q > n$.

Algorithm 1 X and Y are skewed and reversed in L and T, respectively.

1: **for** $i = 1$ to n **do**
2:     **for** $j = 1$ to $2n - 1$ **do**
3:         $L[i][j] \leftarrow x[i][j - i + 1]$
4: **for** $i = 1$ to $2n - 1$ **do**
5:     **for** $j = 1$ to n **do**
6:         $T[i][j] \leftarrow y[i - j + 1][j]$

We use L and T in Cannon's algorithm, given by Algorithm 2. A close up of the upper left corner of the 2D mesh array at the start of Cannon's algorithm is shown in Figure 2.2, where the skewed and reversed matrices X and Y, stored in L and T, respectively, are illustrated.

Algorithm 2 Cannon's algorithm for computing $Z = XY$ in parallel.

```
1: for k = 1 to 3n − 2 do
2:     for all 1 ≤ i, j ≤ n in parallel do
3:         if j = 1 then
4:             l(i, 1) ← L[i][1]; /* load X from Left */
5:         else
6:             l(i, j) ← l(i, j − 1) /* receive data from Left */
7:         if i = 1 then
8:             t(1, j) ← T[1][j]; /* load Y from Top */
9:         else
10:            t(i, j) ← t(i − 1, j) /* receive data from Top */
11:        v(i, j) ← v(i, j) + l(i, j) · t(i, j) /* accumulate value for z_ij */
12:    Shift L horizontally by one (i.e. L[i][j] ← L[i][j + 1])
13:    Shift T vertically by one (i.e. T[i][j] ← T[i + 1][j])
```

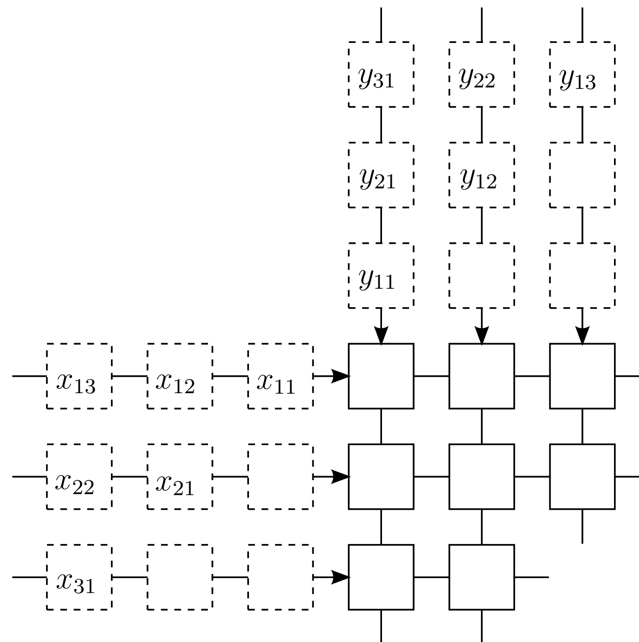

Figure 2.2: An illustration of Cannon's algorithm, showing a close up of the upper left corner of the 2D mesh array.

Lemma 1. *After k steps of Algorithm 2, we have the following boundary conditions for $j = 1$ and $i = 1$, respectively:*

$$\begin{aligned} l(i, 1) &= x[i][k - i + 1] \\ t(1, j) &= y[k - j + 1][j] \end{aligned}$$

Proof. Since the array L is originally $L[i][j] = x[i][j-i+1]$ and the elements are horizontally shifted by one in each step, at the k-th step, $l(i,1) = L[i][1]$ is given by the initial value $L[i][k] = x[i][k-i+1]$. Similar argument can be made for $t(1, j)$. □

Lemma 2. *For all $k \geq 0$ and for all $1 \leq i, j \leq n$, we have the space/time invariants $P(i, j, k)$ where:*

$$\begin{aligned} P(i,j,k) \Leftrightarrow \quad l(i,j) &= x[i][k-i-j+2] \\ t(i,j) &= y[k-i-j+2][j] \\ v(i,j) &= x[i][1]y[1][j] + x[i][2]y[2][j] + ... + \\ & \quad x[i][k-i-j+2]y[k-i-j+2][j] \end{aligned}$$

Proof. Proof is based on induction on the three dimensional logical space indexed by (i, j, k). The basis for k is $P(i, j, 0)$. Since $i, j \geq 1$, array indices for x and y become out of range. Thus for $P(i, j, 0)$, we have $l(i, j) = t(i, j) = v(i, j) = 0$. As mentioned in Section 2.2.1, the values of all registers are initialized to 0 i.e. when $k = 0$ all registers are 0 by definition.

For general $P(i, j, k)$ for $k \geq 0$, we start with the two cases $j = 1$ and $j > 1$. In the first case, from Lemma 1, we have $l(i, 1) = x[i][k - i + 1] = x[i][k-i-j+2]$ at the end of the k-th iteration. In the second case, assume $P(i, j - 1, k - 1)$ for induction, that is:

$$l(i, j-1) = x[i][(k-1) - i - (j-1) + 2] = x[i][k-i-j+2]$$

When $j > 1$, the assignment statement $l(i, j) = l(i, j - 1)$ is performed (line 6). Thus at the end of the k-th iteration, from induction, we have $l(i, j) = x[i][k - i - j + 2]$ for $k \geq 0$.

Similarly, from Lemma 1, we have $t(1, j) = y[k-j+1][j] = y[k-i-j+2][j]$

for $i = 1$, and from $P(i-1, j, k-1)$ for $i > 1$, we have:

$$t(i-1, j) = y[(k-1) - (i-1) - j + 2][j] = y[k - i - j + 2][j]$$

When $i > 1$, the assignment statement $t(i, j) = t(i-1, j)$ is performed (line 10). Thus we have $t(i, j) = y[k - i - j + 2][j]$ for $k \geq 0$.

Also from $P(i, j, k-1)$ we have:

$$v(i, j) = x[i][1]y[1][j] + x[i][2]y[2][j] + ... + x[i][k - i - j + 1]y[k - i - j + 1][j]$$

Then the statement $v(i, j) = v(i, j) + l(i, j) \cdot t(i, j)$ is performed (line 11). Thus at the end of the k-th iteration, $P(i, j, k)$ holds. □

Theorem 1. *Cannon's algorithm takes exactly $3n - 2$ communication steps to compute the product of two n-by-n matrices on a 2D square mesh array with n^2 cells.*

Proof. Following on from Lemma 2, after iteration $k = 3n - 2$, that is, after $3n - 2$ communication steps, we have $v(i, j) = x[i][1]y[1][j] + x[i][2]y[2][j] + ... + x[i][3n - i - j]y[3n - i - j][j]$ for all $1 \leq i, j \leq n$. Since the maximum value for i and j is n, after $3n-2$ steps $v(i, j) = x[i][1]y[1][j] + x[i][2]y[2][j] + ... + x[i][n]y[n][j] = z_{ij}$ for all $v(i, j)$. □

As noted earlier, the loader arrays L and T are not required for actual implementation because the matrix element that needs to be loaded onto a specific register at a specific time can be derived from the location of the edge cells ($cell(1, j)$ and $cell(i, 1)$), and the time, k. In fact the formulae to compute exactly when and where loading of matrix elements must occur have already been specified by Lemma 1. Thus we do not include the time taken to initialize the loader arrays or any other operations performed on the loader arrays in our time analysis.

2.2.3 Loading Data from Both Corners

One shortcoming of Cannon's algorithm is that most cells on the bottom right hand corner of the mesh array are not performing useful computation in the beginning. In fact, $cell(n, n)$ only starts to accumulate the partial

sum for z_{nn} after $2n-2$ steps have already passed. We can reduce the idling time of cells on the bottom right corner of the mesh array by loading values of X and Y from *Right* and *Bottom* at the same time as loading the values from *Left* and *Top*, thereby reducing the total number of steps required to perform matrix multiplication. We note that the idea of loading values from all four directions has already been used previously to achieve a faster mesh algorithm for solving the APSP problem [77].

We define two more loader arrays R and B, similarly to L and T as defined in Section 2.2.2. Again, these arrays are not essential, but we define them to clarify how the matrices are skewed and reversed before the elements are loaded onto the mesh array. Algorithm 3 shows the exact contents of R and B. We then present Algorithm 4 that can compute the product of two n-by-n matrices in exactly $2n-1$ steps. An illustration of this algorithm is given in Figure 2.3, which clarifies the indexing for R and B.

Algorithm 3 X and Y are skewed and reversed in R and B, respectively.

for $i = 1$ to n **do**
  for $j = 1$ to $2n-1$ **do**
    $R[i][j] \leftarrow x[i][2n-i-j+1]$
for $i = 1$ to $2n-1$ **do**
  for $j = 1$ to n **do**
    $B[i][j] \leftarrow y[2n-i-j+1][j]$

Lemma 3. *After k steps of Algorithm 4, we have the following boundary conditions for $j = n$ and $i = n$, respectively:*

$$\begin{aligned} r(i,n) &= x[i][2n-i-k+1] \\ b(n,j) &= y[2n-j-k+1][j] \end{aligned}$$

Proof. Similarly to Lemma 1, since the array R is originally $R[i][j] = x[i][2n-i-j+1]$ and the elements are horizontally shifted by one in each step, at the k-th step, $r(i,n) = R[i][1]$ is given by the initial value $R[i][k] = x[i][2n-i-k+1]$. Similar argument can be made for $b(n,j)$. □

Algorithm 4 Computing $Z = XY$ in parallel in $2n-1$ steps.

```
1: for k = 1 to 2n − 1 do
2:     for all 1 ≤ i, j ≤ n in parallel do
3:         if i + j ≤ n then
4:             /* Region I */
5:             if j = 1 then
6:                 l(i, 1) ← L[i][1]; /* load X from Left */
7:             else
8:                 l(i, j) ← l(i, j − 1) /* receive data from Left */
9:             if i = 1 then
10:                t(1, j) ← T[1][j]; /* load Y from Top */
11:            else
12:                t(i, j) ← t(i − 1, j) /* receive data from Top */
13:            v(i, j) ← v(i, j) + l(i, j) · t(i, j)
14:        else
15:            /* Region II */
16:            if j = n then
17:                r(i, n) ← R[i][1]; /* load X from Right */
18:            else
19:                r(i, j) ← r(i, j + 1) /* receive data from Right */
20:            if i = n then
21:                b(n, j) ← B[1][j]; /* load Y from Bottom */
22:            else
23:                b(i, j) ← b(i + 1, j) /* receive data from Bottom */
24:            v(i, j) ← v(i, j) + r(i, j) · b(i, j)
25:    Shift L and R horizontally by one in opposite directions
26:    Shift T and B vertically by one in opposite directions
```

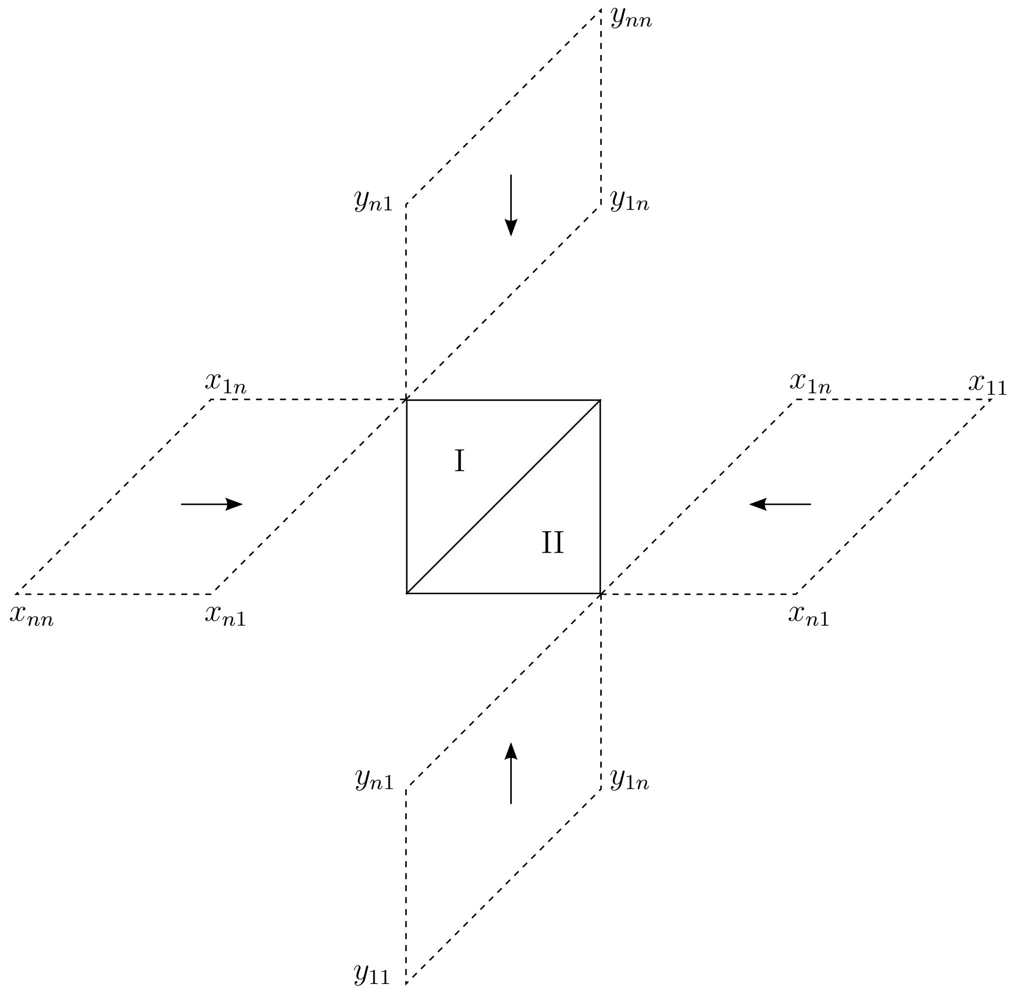

Figure 2.3: Loading values from both corners of the mesh array.

Lemma 4. *For all $k \geq 0$ and for all $1 \leq i, j \leq n$ such that $i + j \leq n + 1$ (Region I), we have the space/time invariants $P(i, j, k)$ where:*

$$\begin{array}{rcrcl} P(i,j,k) & \Leftrightarrow & l(i,j) & = & x[i][k-i-j+2] \\ & & t(i,j) & = & y[k-i-j+2][j] \\ & & v(i,j) & = & x[i][1]y[1][j] + x[i][2]y[2][j] + ...+ \\ & & & & x[i][k-i-j+2]y[k-i-j+2][j] \end{array}$$

Proof. Same as the proof of Lemma 2. □

Lemma 5. *For all $k \geq 0$ and for all $1 \leq i, j \leq n$ such that $i + j > n + 1$ (Region II), we have the space/time invariants $P(i, j, k)$ where:*

$$\begin{array}{rcrcl} P(i,j,k) & \Leftrightarrow & r(i,j) & = & x[i][3n-i-j-k+1] \\ & & b(i,j) & = & y[3n-i-j-k+1][j] \\ & & v(i,j) & = & x[i][n]y[n][j] + x[i][n-1]y[n-1][j] + ...+ \\ & & & & x[i][3n-i-j-k+1]y[3n-i-j-k+1][j] \end{array}$$

Proof. Proof is based on induction on the three dimensional logical space indexed by (i, j, k). The basis for k is $P(i, j, 0)$, where $r(i, j) = b(i, j) = v(i, j) = 0$. This is true by definition as all registers are initialized to zero as specified in Section 2.2.1.

For general $P(i, j, k)$ for $1 \leq i, j \leq n$ and $k \geq 0$, we start with the two cases $j = n$ and $j < n$. In the first case, from Lemma 3, we have $r(i, n) = x[i][2n - i - k + 1] = x[i][3n - i - j - k + 1]$ at the end of the k-th iteration. In the second case, assume $P(i, j + 1, k - 1)$ for induction, that is:

$$r(i, j + 1) = x[i][3n - i - (j + 1) - (k - 1) + 1] = x[i][3n - i - j - k + 1]$$

When $j < n$, the assignment statement $r(i, j) = r(i, j + 1)$ is performed. Thus at the end of the k-th iteration, from induction, we have $r(i, j) = x[i][3n - i - j - k + 1]$ for $k \geq 0$.

Similarly, from Lemma 3, we have $b(n, j) = y[2n - j - k + 1][j] = y[3n - i - j - k + 1][j]$ for $i = n$, and from $P(i + 1, j, k - 1)$ for $i < n$ and $k \geq 0$, we

have:

$$b(i+1,j) = y[3n-(i+1)-j-(k-1)+1][j] = y[3n-i-j-k+1][j]$$

When $i < n$, the assignment statement $b(i,j) = b(i+1,j)$ is performed. Thus we have $b(i,j) = y[3n-i-j-k+1][j]$ for $k \geq 0$.

Also from $P(i,j,k-1)$ we have:

$$v(i,j) = x[i][n]y[n][j] + x[i][n-1]y[n-1][j] + x[i][n-2]y[n-2][j] + ... \\ +x[i][3n-i-j-k+2]y[3n-i-j-k+2][j]$$

Then the statement $v(i,j) = v(i,j) + r(i,j) \cdot b(i,j)$ is performed. Thus at the end of the k-th iteration, $P(i,j,k)$ holds. □

Theorem 2. *Algorithm 4 correctly computes* $Z = X \times Y$ *in* $\{v(i,j)\}$ *in* $2n-1$ *steps.*

Proof. We divide the cells into two regions, Region I and Region II. All $cell(i,j)$ such that $i+j \leq n+1$ belongs to Region I, and all other cells belong to Region II. For Region I, when $k = 2n-1$, by Lemma 4, we have $v(i,j) = x[i][1]y[1][j] + x[i][2]y[2][j] + ... + x[i][2n-i-j+1]y[2n-i-j+1][j]$. Since the maximum value for $i+j$ in Region I is $n+1$, $v(i,j) = z_{ij}$ for all $v(i,j)$ in Region I. For Region II, by Lemma 5 we have $v(i,j) = x[i][n]y[n][j] + x[i][n-1]y[n-1][j] + ... + x[i][n-i-j+2]y[n-i-j+2][j]$. Since the minimum value for $i+j$ is $n+2$ for Region II, we have $v(i,j) = z_{ij}$ also for all $v(i,j)$ in Region II. □

2.2.4 Using Values from All Four Directions

We can observe that in Algorithms 2 and 4, each $cell(i,j)$ performs $x[i][k] \cdot y[k][j]$ for a single value of k in each step. Suppose at a given step, $x[i][k_1]$ and $y[k_1][j]$ arrives on $cell(i,j)$ from left and top, respectively, and $x[i][k_2]$ and $y[k_2][j]$ arrives from right and bottom, respectively, such that $k_1 \neq k_2$. Then we can perform the calculation $v(i,j) \leftarrow v(i,j) + x[i][k_1]y[k_1][j] + x[i][k_2]y[k_2][j]$ in the given step, which we call the quintuple operation. This allows us to accumulate the sum of two products instead of just one product

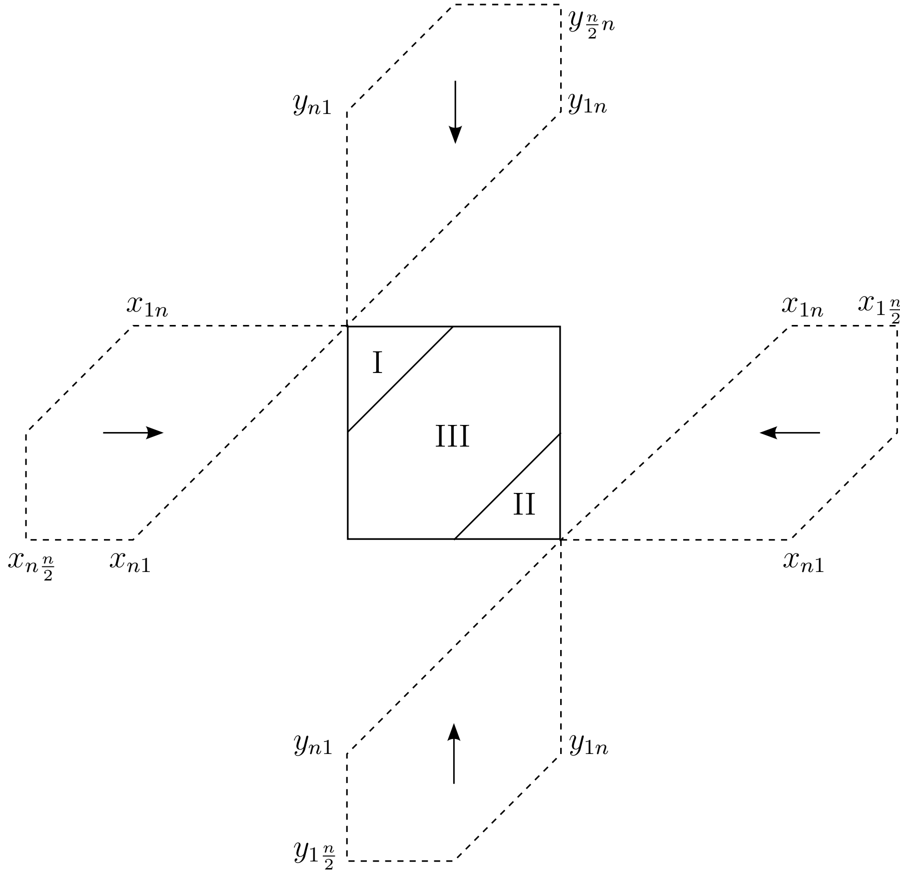

Figure 2.4: Utilising data from all four directions.

in a single step, thereby reducing the total number of communication steps even further.

We present Algorithm 5 to compute $Z = XY$ in exactly $1.5n - 1$ steps, where n is assumed to be even for simplicity. An illustration of the start of this algorithm is given in Figure 2.4. Note that there is no longer a need to load all n^2 values from both X and Y from the four different directions. Since the algorithm finishes after $1.5n - 1$ steps, as illustrated in Figure 2.4, we only load $\frac{7}{8}n^2$ matrix elements from each direction.

Lemma 6. *For all $k \geq 0$ and for all $1 \leq i, j \leq n$ such that $i + j \leq \frac{n}{2} + 1$*

Algorithm 5 Computing $Z = XY$ in parallel in $1.5n - 1$ steps.

```
1:  for k = 1 to 1.5n − 1 do
2:      for all 1 ≤ i, j ≤ n in parallel do
3:          if i + j ≤ n/2 + 1 then
4:              /* Region I */
5:              Same as Region I in Algorithm 4
6:          else if i + j > 3n/2 then
7:              /* Region II */
8:              Same as Region II in Algorithm 4
9:          else/* n/2 + 1 < i + j ≤ 3n/2 */
10:             /* Region III */
11:             if j = 1 then
12:                 l(i, 1) ← L[i][1]
13:             else
14:                 l(i, j) ← l(i, j − 1)
15:             if i = 1 then
16:                 t(1, j) ← T[1][j]
17:             else
18:                 t(i, j) ← t(i − 1, j)
19:             if j = n then
20:                 r(i, n) ← R[i][1]
21:             else
22:                 r(i, j) ← r(i, j + 1)
23:             if i = n then
24:                 b(n, j) ← B[1][j]
25:             else
26:                 b(i, j) ← b(i + 1, j)
27:             v(i, j) ← v(i, j) + l(i, j) · t(i, j) + r(i, j) · b(i, j)
28:     Shift L and R horizontally by one
29:     Shift T and B vertically by one
```

(Region I), we have the space/time invariants $P(i, j, k)$ where:

$$
\begin{array}{rcrcl}
P(i,j,k) & \Leftrightarrow & l(i,j) & = & x[i][k-i-j+2] \\
 & & t(i,j) & = & y[k-i-j+2][j] \\
 & & v(i,j) & = & x[i][1]y[1][j] + x[i][2]y[2][j] + ...+ \\
 & & & & x[i][k-i-j+2]y[k-i-j+2][j]
\end{array}
$$

Proof. Same as the proof of Lemma 2. □

Lemma 7. *For all $k \geq 0$ and for all $1 \leq i, j \leq n$ such that $i+j > \frac{3n}{2}$ (Region II), we have the space/time invariants $P(i, j, k)$ where:*

$$
\begin{array}{rcrcl}
P(i,j,k) & \Leftrightarrow & r(i,j) & = & x[i][3n-i-j-k+1] \\
 & & b(i,j) & = & y[3n-i-j-k+1][j] \\
 & & v(i,j) & = & x[i][n]y[n][j] + x[i][n-1]y[n-1][j] + ...+ \\
 & & & & x[i][3n-i-j-k+1]y[3n-i-j-k+1][j]
\end{array}
$$

Proof. Same as the proof of Lemma 5 □

Lemma 8. *For all $k \geq 0$ and for all $1 \leq i, j \leq n$ such that $\frac{n}{2}+1 < i+j \leq \frac{3n}{2}$ (Region III), we have the space/time invariants $P(i, j, k)$ where:*

$$
\begin{array}{rcrcl}
P(i,j,k) & \Leftrightarrow & l(i,j) & = & x[i][k-i-j+2] \\
 & & t(i,j) & = & y[k-i-j+2][j] \\
 & & r(i,j) & = & x[i][3n-i-j-k+1] \\
 & & b(i,j) & = & y[3n-i-j-k+1][j] \\
 & & v(i,j) & = & x[i][1]y[1][j] + x[i][2]y[2][j] + ... \\
 & & & & x[i][k-i-j+2]y[k-i-j+2][j] + ... \\
 & & & & x[i][3n-i-j-k+1]y[3n-i-j-k+1][j] + ... \\
 & & & & x[i][n-1]y[n-1][j] + x[i][n]y[n][j]
\end{array}
$$

Proof. Proof is straightforward by combining the proofs of Lemma 6 and Lemma 7. □

Theorem 3. *On a 2D square mesh array with n^2 cells where each cell is limited to four connections to its neighbours (*Top, Bottom, Left *and* Right*), the product of two n-by-n matrices can be computed in exactly $1.5n - 1$ steps.*

Proof. We divide the mesh array into three regions, Region I, II and III. All $cell(i,j)$ such that $i+j \leq \frac{n}{2}+1$ belong to Region I. All $cell(i,j)$ such that $i+j \geq \frac{3n}{2}+1$ belong to Region II. All other cells belong to Region III.

- Region I: At $k = 1.5n-1$, by Lemma 6, $v(i,j) = x[i][1]y[1][j] + x[i][2]y[2][j] + ... + x[i][1.5n-i-j+1]y[1.5n-i-j+1][j]$. Since the maximum value of $i+j$ is $0.5n+1$, $v(i,j) = z_{ij}$ for all (i,j) in Region I.

- Region II: At $k = 1.5n-1$, by Lemma 7, $v(i,j) = x[i][n]y[n][j] + x[i][n-1]y[n-1][j] + ... + x[i][1.5n-i-j+2]y[1.5n-i-j+2][j]$. Since the minimum value of $i+j$ is $1.5n+1$, $v(i,j) = z_{ij}$ for all (i,j) in Region II.

- Region III: At $k = 1.5n-1$, by Lemma 8, $v(i,j) = x[i][1]y[1][j] + x[i][2]y[2][j] + ... + x[i][1.5n-i-j+1]y[1.5n-i-j+1][j] + x[i][1.5n-i-j+2]y[1.5n-i-j+2][j] + ... + x[i][n-1]y[n-1][j] + x[i][n]y[n][j]$. Thus $v(i,j) = x[i][1]y[1][j] + x[i][2]y[2][j] + ... + x[i][n]y[n][j] = z_{ij}$.

□

We now conclude this chapter with our contribution to the topic of matrix multiplication given by Theorem 3. We make a final note that there is still room for improvements to Algorithm 5 in terms of the memory usage. As noted earlier, the loader matrices are not required for actual implementation hence no additional memory is required for storing skewed and reversed matrices. Also if the mesh array supports loading data from both *Left* and *Right* (and both *Top* and *Bottom*) from the same location, then again we do not need any additional memory for storing copies of matrices to load values of matrices from both corners.

Chapter III

Shortest Paths (SP)

The SP problem is often studied as two separate sub-topics, namely, the Single Source SP (SSSP) problem, and the All Pairs SP (APSP) problem. As the names of the problems suggest, the SSSP problem is to find the shortest paths from a single source vertex, s, to all other vertices in the graph, and the APSP problem is to find the shortest paths between all pairs of vertices.

Arguably the most famous algorithms for solving the SSSP and the APSP problems are Dijkstra's algorithm[1] [16] and Floyd's algorithm [22], respectively. Dijkstra's algorithm runs in $O(m + n \log n)$ time if enhanced with a priority queue such as the Fibonacci heap [25, 9]. Floyd's algorithm runs in $O(n^3)$ time. For dense graphs where $m = O(n^2)$, solving the APSP problem with Floyd's algorithm has the same time complexity as solving the problem by running Dijkstra's algorithm n times, once per each vertex as the source.

Many algorithms exist that exploit certain properties of the input graphs to achieve faster time bounds. For example, for graphs with integer edge costs bounded by c, Thorup gave an $O(m + n \log \log c)$ algorithm [75] for solving the SSSP problem and Takaoka gave an $O(mn + n^2 \log (c/n))$ algorithm [71] for solving the APSP problem. For nearly acyclic graphs, Takaoka gave an $O(m + n \log k)$ algorithm [69] for solving the SSSP problem where k is the maximum cardinality of the strongly connected components, Abuaiadh and Kingston gave an $O(m+n \log t)$ algorithm [1] where t is the number of delete-min operations performed in the priority queue manipulation, and Saunders and Takaoka gave an $O(m + r \log r)$ algorithm [56] where the graph can be decomposed into r trees (or more generally, r 1-dominator sets, including trees).

There has also been active research in breaking the $O(n^3)$ barrier for

[1] Dijkstra's algorithm only works on graphs with non-negative edge costs. We do not discuss graphs with negative edge costs in this thesis.

Year	Time Complexity	Author(s)
1976	$O(n^3(\log\log n/\log n)^{1/3})$	Fredman [24]
1990	$O(n^3/\sqrt{\log n})$	Dobosiewicz [18]
1992	$O(n^3\sqrt{\log\log n/\log n})$	Takaoka [67]
2004	$O(n^3(\log\log n/\log n)^{5/7})$	Han [31]
2005	$O(n^3\log\log n/\log n)$	Takaoka [70]
2006	$O(n^3\sqrt{\log\log n}/\log n)$	Zwick [84]
2008	$O(n^3/\log n)$	Chan [11]
2008	$O(n^3(\log\log n/\log n)^{5/4})$	Han [32]
2012	$O(n^3\log\log n/\log^2 n)$	Han and Takaoka [33]

Table 3.1: Breaking the cubic barrier of the APSP problem.

solving the APSP problem on dense graphs with real edge costs. Table 3.1 shows the progress in reducing the asymptotic worst-case time complexity. The main idea used in most of the listed algorithms is to pre-compute a lookup table that can be used to speed up the main computation of finding the shortest paths. All time complexities given in Table 3.1 are only slightly sub-cubic, achieving speed-ups of only polylog[2] factor.

In 1991, the three authors Alon, Galil and Margalit made a breakthrough by providing a deeply sub-cubic algorithm [4], albeit only for small integer edge costs. We refer to this algorithm as the AGM algorithm. In Section 3.2, we provide a review of the AGM algorithm, which was the first algorithm to successfully utilise FMMOR to solve the APSP problem. Our first contribution to the SP problem is to enhance the AGM algorithm such that the running time of the algorithm remains sub-cubic for larger integer edge costs.

In terms of parallel algorithms, in 1967, Hu gave a parallel distributed algorithm for solving the APSP problem on an n-by-n mesh array [36] based on a serial algorithm called the cascade algorithm, which was invented by Farby, Land and Murchland earlier in the same year [21]. Later in 1987, Lakhani and Dorairaj gave a different parallel algorithm for the n-by-n mesh array [42], based on Floyd's algorithm [22] for solving the APSP problem. On a related note, in 1979, Guibas, Kung and Thompson gave a parallel

[2] A polylog factor is a logarithmic factor that has been raised to a power greater than 1.

algorithm for solving the transitive closure on the n-by-n mesh array [30], and Ullman showed that this algorithm for computing the transitive closure can be easily adapted to solve the APSP problem [76].

More recently, in 1992, Takaoka and Umehara enhanced the algorithm by Lakhani and Dorairaj such that the APSP problem can be solved in just $3.5n$ communication steps [72], which was a big improvement from the $5n$ communication steps required by both Hu's algorithm and the original algorithm by Lakhani and Dorairaj. Our second contribution to the SP problem, given in Section 3.4, is to achieve exactly $3n-2$ communication steps for solving the APSP problem on the n-by-n mesh array.

Before we present our two contributions to the SP problem in Sections 3.3 and 3.4, we describe the algebraic structure called the distance semi-ring in Section 3.1 to show how the theory of semi-rings and matrices can relate to the SP problem.

3.1 Distance Semi-ring

We define a matrix called the distance matrix, $D = \{d_{ij}\}$, where d_{ij} is the edge cost from vertex i to vertex j. That is, $d_{ij} = cost(i, j)$. If $(i, j) \notin E$, then $d_{ij} = \infty$. Naturally for graphs with n vertices, D is an n-by-n matrix. $d_{ii} = 0$ for all $1 \leq i \leq n$, that is, the distance from a vertex to itself is zero.

We now introduce the $(min, +)$-product of matrices denoted as $\star$, such that the $(min, +)$-product $Z = X \star Y$ is given by:

$$z_{ij} = \min_{k=1}^{n}\{x_{ik} + y_{kj}\}$$

The meaning of the $(min, +)$-product becomes clear when we consider computing $D^2 = D \star D$. We can think of k as the "via" vertex on a path from vertex i to vertex j, as shown in Figure 3.1. The distance from vertex i to vertex j via vertex k is given by $d_{ik} + d_{kj}$. After computing the distances for all $1 \leq k \leq n$ via vertices, we take the minimum among them. Hence d^2_{ij} is the shortest distance possible from vertex i to vertex j with paths of lengths up to 2, where the path length is the number of edges on the path. (It is important to clearly distinguish the path length from the path dis-

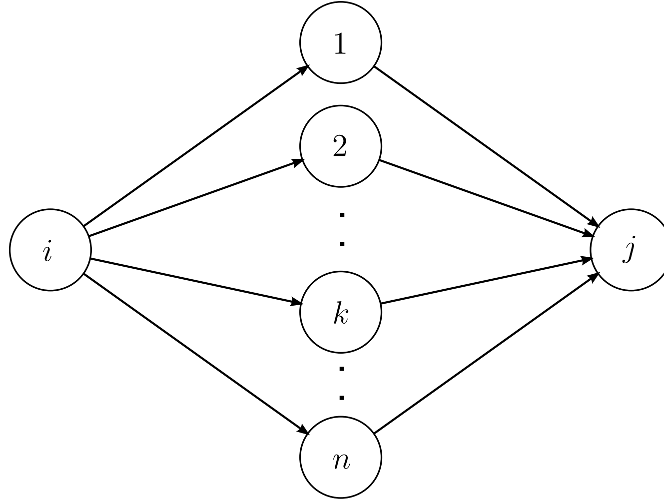

Figure 3.1: Which via vertex, k, gives us the best path from i to j?

tance/cost.) Clearly, D^{n-1} is the solution to the APSP problem. If $T(n)$ is the time taken to perform the $(min, +)$-product, by repeated squaring, we can solve the APSP problem in $O(T(n) \log n)$ time.

In fact, we can solve the APSP problem in just $O(T(n))$ time based on the theory of semi-rings. $(\mathbb{R}, \min, +, \infty, 0)$ is a closed semi-ring [2], which we refer to as the distance semi-ring. Then it follows that $(M, +, \cdot, O, I)$ is also a semi-ring, where M is the set of all possible n-by-n distance matrices, O is the zero distance matrix and I is the identity distance matrix. We refer to $(M, +, \cdot, O, I)$ as the distance matrix semi-ring. In the distance matrix semi-ring, $+$ is a component-wise min operation and $\cdot$ is the $(min, +)$-product defined above. The zero distance matrix, O, has ∞ for all elements and the identity distance matrix, I, has 0 for the diagonals and ∞ for all other elements.

The closure of the distance matrix D in the distance matrix semi-ring, denoted by D^*, is given by the following equation:

$$D^* = \sum_{k \in \mathbb{N}} D^k$$

Since the maximum number of edges of any shortest path is $n - 1$, we can

stop at D^{n-1} to compute the closure in the distance matrix semi-ring [82]:

$$D^* = \sum_{k=0}^{n-1} D^k$$

Clearly, D^* is the solution to the APSP problem. A general algorithm for computing the closure of a semi-ring matrix was given by McNaughton and Yamada [46]. In fact, Floyd's algorithm can be thought of as a specific instance of the more generic algorithm by McNaughton and Yamada, and using Floyd's algorithm, D^* can be computed in $O(n^3)$ time.

It is also known that the closure of a semi-ring matrix can be computed in the same asymptotic time complexity as computing the product in the semi-ring if the equation $T(2n) \geq 4(T(n))$ is satisfied [2]. This is clearly true in the distance matrix semi-ring, hence the APSP problem can be solved in $O(T(n))$. Using the straightforward method to compute the $(min, +)$-product yields $T(n) = O(n^3)$, which equals the time bound given by Floyd's algorithm.

Can we claim $T(n) = O(n^\omega)$ where $\omega < 2.373$? Unfortunately FMMOR introduced in Section 2.1 only works over a ring. More specifically, FMMOR requires the inverse of the $+$ operation. In the distance matrix semi-ring the $+$ operation is a component-wise *min* operation, and no inverse operation exists for the *min* operation.

Thus even though the APSP problem translates nicely onto a matrix, FMMOR cannot be used directly to compute the closure of the distance matrix semi-ring. However, it is possible to utilise FMMOR to achieve a sub-cubic time bound for the APSP problem, as we will discuss in Section 3.2.

3.2 Deeply Sub-cubic Time Complexity

In this section we provide a review of the AGM algorithm. We start by defining the reachability matrix $R = \{r_{ij}\}$, where $r_{ij} = 1$ if an edge exists from vertex i to vertex j, and 0 otherwise. A vertex is reachable from itself, hence $r_{ii} = 1$ for all $1 \leq i \leq n$. Clearly R is an n-by-n Boolean matrix. We define the Boolean-product, denoted as $\bullet$, such that the Boolean product

$Z = X \bullet Y$ is given by:

$$z_{ij} = \bigvee_{k=1}^{n} \{x_{ik} \wedge y_{kj}\}$$

Similarly to the $(min, +)$-product that was introduced in Section 3.1, the meaning of the Boolean-product becomes clear when we consider computing $R^2 = R \bullet R$. Again k can be thought of as the via vertex. $r^2_{ij} = 1$ if vertex j is reachable from vertex i with paths of lengths up to 2. If $r^{n-1}_{ij} = 1$ for all i and j, then the graph must be strongly connected. In fact R^{n-1} is called the reflexive-transitive closure.

Since Boolean algebra does not form a ring, FMMOR cannot be used directly to compute the Boolean-product. It is quite straightforward, however, to utilise FMMOR to perform the Boolean-product as follows: we simply treat all 0s and 1s in the Boolean matrices as integers and use FMMOR to compute the matrix product over the ring of integer matrices, then we scan the resulting matrix and convert all elements that are greater than 1 to 1.

Let us now simplify the APSP problem and consider only directed graphs with unit edge costs. Then we can use the reachability matrix to determine the shortest distances. For example, if $r^{\ell}_{ij} = 1$ but $r^{\ell-1}_{ij} = 0$ for some path length ℓ, then this means vertex j is only reachable from vertex i with paths of lengths ℓ or greater. Since on graphs with unit edge costs path lengths and path distances are equivalent, the shortest distance from i to j in this example must be ℓ. Algorithm 6 shows how the distances can be retrieved from the reachability matrix.

Algorithm 6 Determine the shortest distances from the reachability matrix.

1: **for** $\ell = 2$ to r **do**
2: $\quad R^{\ell} \leftarrow R^{\ell-1} \bullet R$
3: $\quad D^{\ell} \leftarrow D^{\ell-1}$
4: $\quad$ **for** $i \leftarrow 1$ to n; $j \leftarrow 1$ to n **do**
5: $\quad\quad$ **if** $r^{\ell}_{ij} = 1$ and $r^{\ell-1}_{ij} = 0$ **then**
6: $\quad\quad\quad d^{\ell}_{ij} \leftarrow \ell$

In Algorithm 6, r is an integer constant such that $1 < r < n$. If $r = n-1$, at the end of the algorithm we have D^{n-1} and hence the APSP problem has been solved. This method of solving the APSP problem is inefficient,

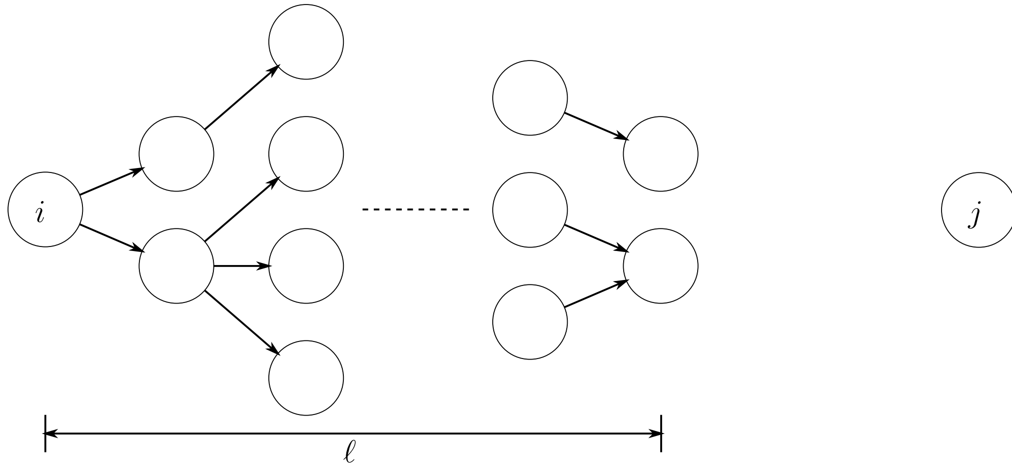

Figure 3.2: To get to vertex j from i, must we check $O(n)$ via vertices?

however, as the total time complexity of this method is $O(n^{\omega+1})$, which is slower than the $O(n^3)$ running time of Floyd's algorithm [22].

This is where Alon *et al.*'s main theorem comes in. Let us suppose that R^ℓ, and thus D^ℓ has been computed for some $\ell < n$. Suppose we wish to compute $D^{2\ell}$ by squaring D^ℓ using the $(min, +)$-product. Then do we really need to consider all $1 \leq k \leq n$ via vertices to compute each $d_{ij}^{2\ell}$?

Let us consider vertex i as the source vertex, and let j be a vertex such that the shortest distance from i to j is between ℓ and 2ℓ. See Figure 3.2. To get to vertex j from i, we need to pass at least one vertex at path length 1, and another vertex at path length 2, and another vertex at path length 3, and so on until we finally have to pass a vertex at path length of ℓ. In the worst case there may already be $O(n)$ vertices that are reachable from i with path lengths up to ℓ. Then by the pigeon hole principle, we can pick a path length p between 1 and ℓ, such that the number of vertices at path length of p is at most $O(n/\ell)$. Since at least one vertex whose path length is p must be passed from vertex i to get to vertex j, this means we only need to check $O(n/\ell)$ as our via vertices rather than all n vertices! We refer to this key theorem as the bridging set theorem, which gives rise to Algorithm 7.

We use the same car analogy as the review of the same algorithm given by Takaoka [68]. We refer to the first part of the algorithm as the acceleration

Algorithm 7 The AGM algorithm for solving the APSP problem.

```
    /* Acceleration phase*/
 1: Same as Algorithm 6

    /* Cruising phase*/
 2: ℓ = r
 3: while ℓ < n do
 4:     ℓ' ← ⌈3ℓ/2⌉
 5:     for i ← 1 to n do
 6:         scan i^th row of D^ℓ with j and find the smallest set of equal d^ℓ_ij
 7:             such that ⌈ℓ/2⌉ ≤ d^ℓ_ij ≤ ℓ and let the set of j be S_i
 8:         /* S_i is the bridging set for row i */
 9:     for i ← 1 to n; j ← 1 to n do
10:         m_ij ← min_{k∈S_i}{d^ℓ_ik + d^ℓ_kj} /* squaring D^ℓ with the help of S_i */
11:         if m_ij < d^ℓ_ij and m_ij < ℓ' then
12:             d^ℓ'_ij ← m_ij
13:         else
14:             d^ℓ'_ij ← d^ℓ_ij
15:     ℓ ← ℓ'
```

phase, where we slowly build up the path length one by one up to some integer constant $1 < r < n$. Once we have reached a certain speed (path length), we change gear and move onto the cruising phase, where we perform repeated squaring of the distance matrix with the help of the bridging set, S_i for row i, which is the set of via vertices to check from the source vertex i to the destination vertex j, for $1 \leq j \leq n$. Alon *et al.* chose to increase the path length by a factor of 1.5 in each iteration of the cruising phase. This is somewhat arbitrary, as any factor greater than 1 and less than 2 will work.

The time complexity of the acceleration phase is $O(rn^{\omega})$. The time complexity of the cruising phase is not so straightforward. As explained above we only need to consider $O(n/r)$ via vertices to compute each of the n^2 distances ($|S_i| = O(n/r)$) thus we start with $O(n^2 \cdot n/r)$. No logarithmic factor is required for repeated squaring because the path length increases by a constant factor in each iteration, thereby decreasing the size of the bridging set accordingly. This means we end up with a geometric series if we add up the size of the bridging set in each iteration and the first term dominates the

time complexity. Therefore the time complexity of the cruising phase is just $O(n^2 \cdot n/r)$. We choose the best value for r by balancing the time complexities of the two phases. $rn^\omega = n^3/r$ gives us $r = n^{(3-\omega)/2}$, and hence the asymptotic worst case time complexity of Algorithm 7 is $O(n^{(3+\omega)/2}) < O(n^{2.687})$.

But have we actually solved the APSP problem? We have computed the shortest distances between vertices, but how can we retrieve the explicit paths? After all, the name of the problem is "Shortest Paths", not "Shortest Distances". Storing all explicit paths would require $O(n^3)$ time, which defeats the purpose of sub-cubic algorithms. We get around this problem with the help of *witness* vertices and *successor* vertices.

Suppose in an iteration $d_{ik} + d_{kj}$ gives us the minimum distance from i to j for some via vertex k. Then k is the witness vertex. In other words, the vertex k proves to us that there exists a path from i to j with the given distance. In the cruising phase, retrieving the witness vertex is trivial. In the acceleration phase, however, retrieving the witness vertex is not a simple matter as FMMOR is used to perform the Boolean-product. Fortunately, it is known that a witnessed Boolean-product can be performed with an additional polylog factor [26], such that in each iteration of the acceleration phase, we are given the witness matrix $W = \{w_{ij}\}$ for all pairs of vertices as a by-product.

From the witness vertices, we can derive the successor vertices. A successor vertex on a path from i to j is the vertex that comes immediately after i on the path. Thus simply by following the successor vertices one by one, we can derive explicit paths in time linear to the path length. The algorithm for deriving the successor vertices from the witness vertices in $O(n^2)$ time bound has been given by Zwick [83].

And thus, the APSP problem on directed graphs with unit edge costs can be solved in $\tilde{O}(n^{2.687})$ time, where $\tilde{O}$ is a common notation used to omit all polylog factors in the time complexity. The result of this algorithm is the matrix D^{n-1} that contains the shortest distances for all pairs of vertices, and another n-by-n matrix of successor vertices that we can use to retrieve explicit paths.

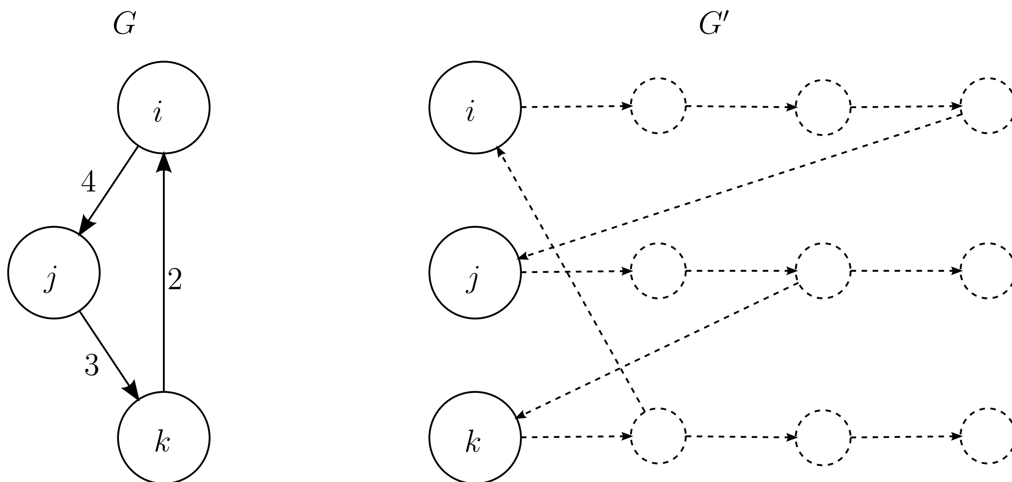

Figure 3.3: Expanding G to G' with $c = 4$.

3.3 Expansion and Contraction of Graphs

Alon *et al.* then applied their deeply sub-cubic algorithm to graphs with integer edge costs. In order to do this, they provided a mechanism to transform the graph G that has integer edge costs bounded by c, to another graph, G', with unit edge costs with a total of cn vertices, such that solving the APSP problem on G' solves the problem on G. We refer to this transformation process as graph expansion. Expansion of G to G' involves creating $c-1$ artificial vertices for each real vertex, and linking the artificial vertices and real vertices to ensure that the distances between the real vertices stays consistent with G. An example of the graph expansion process is shown in Figure 3.3.

As the example shows, after creating a chain of artificial vertices, we can link the vertices together with edges of unit costs such that the distances between real vertices in G' is consistent with the distances in G. Therefore the APSP problem on directed graphs with integer edge costs bounded by c can be solved with Algorithm 7 in $\tilde{O}((cn)^{(3+\omega)/2})$ time, which remains sub-cubic for $c < n^{(3-\omega)/(3+\omega)} < n^{0.117}$.

Note that we can optimize G' from the observation that not all real vertices require the chain of $c-1$ artificial vertices created for them. If we let k

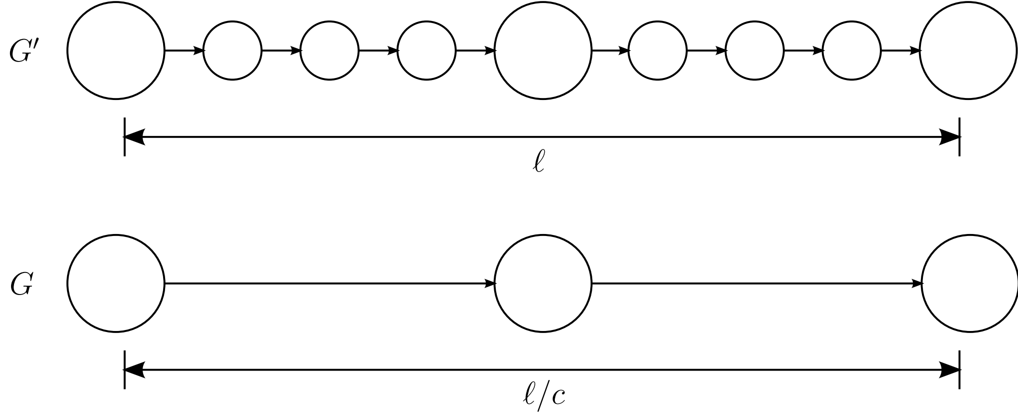

Figure 3.4: Path length of ℓ in G' is equivalent to path length of ℓ/c in G.

be the maximum edge cost out of all out-going edges from the real vertex v such that $1 \leq k \leq c$, then it is sufficient to create just $k-1$ artificial vertices for v rather than $c-1$ artificial vertices. This optimization during graph expansion can be significant if the variance in edge costs in G is large.

Our first contribution to the SP problem is to enhance the AGM algorithm such that the algorithm remains sub-cubic for larger values of c. We achieve this with the key observation that only the acceleration phase of Algorithm 7 is restricted to graphs with unit edge costs, and the cruising phase has no such limitations. In summary, we use G' for the acceleration phase, gather the information, then switch back to G for the cruising phase. We refer to the process of going from G' back to G as graph contraction.

Note that the distinction between path lengths and path distances has now become very important. The path length of G' is actually equivalent to the path distance in G, as illustrated in Figure 3.4. The bridging set theorem used in the cruising phase is based on the path length. Therefore in the acceleration phase, while we are building up the path length in G' one by one, we need to keep track of what the equivalent path length is in G, such that we can use this information in the cruising phase to determine the bridging sets, S_i. We define the path length matrix $H = \{h_{ij}\}$ to store this information in the acceleration phase. Algorithm 8 is the enhanced AGM algorithm.

Algorithm 8 Enhanced AGM algorithm.

```
    /* Graph expansion and initialisation */
 1: Expand G to G', initialize D and R based on G'
 2: for i ← 1 to cn; j ← 1 to cn do
 3:     h_ij = 0

    /* Acceleration phase */
 4: for ℓ = 2 to r do
 5:     R^ℓ ← R^(ℓ−1) • R /* witnesses given as W = {w_ij} */
 6:     for i ← 1 to cn; j ← 1 to cn do
 7:         if r^ℓ_ij = 1 and r^(ℓ−1)_ij = 0 then
 8:             d^ℓ_ij ← ℓ
 9:             k = w_ij
10:             if j ∈ G /* j is a real vertex */ then
11:                 h_ij = h_ik + 1 /* path length in G is incremented */
12:             else
13:                 h_ij = h_ik
14:         if d^(ℓ−1)_ij < ℓ then
15:             d^ℓ_ij ← d^(ℓ−1)_ij

    /* Graph contraction */
16: Remove all rows/columns for artificial vertices from D and H

    /* Cruising phase*/
17: ℓ = r
18: while ℓ < n do
19:     ℓ' ← ⌈3ℓ/2⌉
20:     for i ← 1 to n do
21:         scan i^th row of H with j and find the smallest set of equal h_ij such
22:             that ⌈ℓ/2⌉ ≤ h_ij ≤ ℓ and let the set of corresponding j be S_i
23:     for i ← 1 to n; j ← 1 to n do
24:         m_ij ← min_{k∈S_i}{d^ℓ_ik + d^ℓ_kj} /* squaring D^ℓ with S_i */
25:         k ← the witness vertex
26:         if m_ij < d^ℓ_ij then
27:             d^ℓ'_ij ← m_ij
28:             h_ij ← h_ik + h_kj /* path length is updated accordingly */
29:         else
30:             d^ℓ'_ij ← d^ℓ_ij
31:     ℓ ← ℓ'
```

Theorem 4. *Algorithm 8 solves the APSP problem on directed graphs with integer edge costs bounded by c in $\tilde{O}(c^{(1+\omega)/2}n^{(3+\omega)/2})$ time.*

Proof. For proof of correctness it is sufficient to show that the correct path length in G is kept in H since the correctness of Algorithm 7 has already been proven [4]. In the acceleration phase, when j first becomes reachable from i (i.e. when the shortest distance is found from i to j), h_{ij} is incremented only when j is a real vertex. This ensures that we are not counting the artificial vertices in the path length, hence h_{ij} is correct after the acceleration phase. In the cruising phase it is straightforward to keep h_{ij} up to date, simply by updating h_{ij} whenever a shorter path is found between i and j.

The time complexity of the acceleration phase is $\tilde{O}(r(cn)^{\omega})$. For the time complexity of the cruising phase we must consider the lower bound of h_{ij} at the start of the cruising phase, for any i and j, because this determines the upper bound on the size of the bridging set. In the worst case, all edge costs are c, which gives us the lower bound of r/c for any path length h_{ij}. Therefore the upper bound on the size of the bridging set at the start of the cruising phase is $|S_i| = O(cn/r)$, and hence the time complexity of the cruising phase becomes $O(cn^3/r)$. Balancing the two time complexities gives us $r = c^{(1-\omega)/2}n^{(3-\omega)/2}$, which results in the total worst case time complexity of $\tilde{O}(c^{(1+\omega)/2}n^{(3+\omega)/2})$. □

We can observe that $\tilde{O}(c^{(1+\omega)/2}n^{(3+\omega)/2}) \leq \tilde{O}((cn)^{(3+\omega)/2})$. In fact, Algorithm 8 remains sub-cubic for $c < n^{0.186}$. Although there exists more efficient algorithms for the APSP problem that remain sub-cubic for a larger upper bound on integer edge costs, up to $c < n^{0.624}$ [68, 83], our contribution given by Theorem 4 remains significant. Firstly, it enhances a breakthrough algorithm that was widely celebrated, and secondly, as we will discuss later in the thesis, we can use the enhancement to derive efficient algorithms for solving other graph path problems, such as the SP-AF problem.

3.4 APSP Problem on the Mesh Array

Our second contribution to the SP problem comes in the form of a parallel distributed algorithm on an n-by-n 2D square mesh array to solve the

APSP problem. The definition of the mesh array used for this section is almost exactly the same as the mesh array defined in Section 2.2 for matrix multiplication. The only differences are that all registers are initialized to ∞ (instead of 0), and the set of operations performed by each cell in each communication step is different.

Our parallel algorithm is derived from the serial cascade algorithm invented by Farby, Land and Murchland [21]. Thus we start with an in-depth review of the cascade algorithm in Section 3.4.1 based on Umehara's masters thesis [77] and its translation by Takaoka [73]. We then present our new parallel algorithm in Section 3.4.2.

3.4.1 Review of the Cascade Algorithm

The name "cascade" comes from the fact that previously computed distances (d_{ij}) are used in later computations. The algorithm consists of two distinct phases, which we refer to as the Forward Process (FWP) and the Backward Process (BWP). Note that all vertices in V are numbered from 1 to n. In the FWP, the vertices are inspected in increasing order, whereas in the BWP, the vertices are inspected in decreasing order. The pseudo code for the original cascade algorithm is given by Algorithm 9.

Algorithm 9 The original cascade algorithm.

```
   /* Forward Process (FWP) */
1: for i = 1 to n do
2:     for j = 1 to n do
3:         for k = 1 to n do
4:             d_ij = min {d_ij, d_ik + d_kj}
   /* Backward Process (BWP) */
5: for i = n down to 1 do
6:     for j = n down to 1 do
7:         for k = n down to 1 do
8:             d_ij = min {d_ij, d_ik + d_kj}
```

Let us now compare the cascade algorithm to Floyd's algorithm given by Algorithm 10. We refer to the operation of performing "$\min\{d_{ij}, d_{ik} + d_{kj}\}$" as the triple operation. We can observe that the cascade algorithm in its

original form performs twice as many triple operations as Floyd's algorithm. Perhaps this is the reason why the cascade algorithm has been slowly forgotten while Floyd's algorithm remains one of the most well known algorithms in graph theory.

Algorithm 10 Floyd's algorithm.

```
1: for k = 1 to n do
2:     for i = 1 to n do
3:         for j = 1 to n do
4:             d_ij = min {d_ij, d_ik + d_kj}
```

We can revive the cascade algorithm, however, by limiting the number of vertices that are inspected in the inner-most loop, as shown in Algorithm 11. We refer to the modified FWP as the Short Forward Process (SFWP) and the modified BWP as the Long Backward Process (LBWP). Similarly we can define the Long Forward Process (LFWP) where k sweeps from 1 to $\max\{i, j\}$, and the Short Backward Process (SBWP) where k sweeps from n down to $\max\{i, j\}$.

Algorithm 11 Improved cascade algorithm.

```
   /* Short Forward Process (SFWP) */
1: for i = 1 to n do
2:     for j = 1 to n do
3:         for k = 1 to min {i, j} do
4:             d_ij = min {d_ij, d_ik + d_kj}
   /* Long Backward Process (LBWP) */
5: for i = n down to 1 do
6:     for j = n down to 1 do
7:         for k = n down to min {i, j} do
8:             d_ij = min {d_ij, d_ik + d_kj}
```

In fact it is possible to optimize both Floyd's algorithm and the improved cascade algorithm (Algorithm 11) by skipping the triple operation if $i = j$ or $i = k$ or $j = k$. Then the total number of triple operations performed in both of the optimized algorithms can be shown to be exactly $n(n-1)(n-2)$. In other words, the cascade algorithm can be modified from its original version

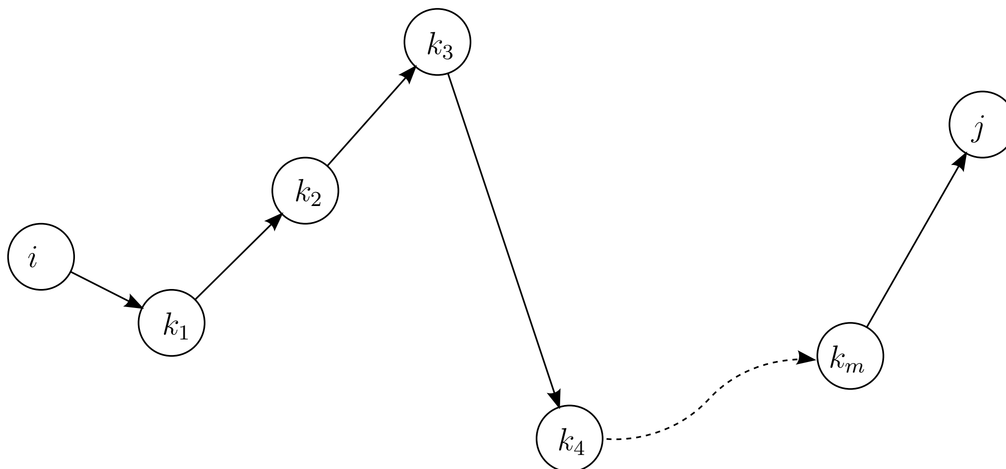

Figure 3.5: The shortest path from i to j.

to be on a par with the better known Floyd's algorithm. But of course, we need to show that the modified cascade algorithm does indeed solve the APSP problem correctly.

Vertex numbers play an important role in proving the correctness of Algorithm 11. Let the shortest path from vertex i to vertex j be denoted by vertices $(i, k_1, k_2, k_3, ..., k_m, j)$. An illustration of such a path is shown in Figure 3.5, where the relative vertical positioning of the vertices is used to visualize the relative vertex numbers. That is, in the path from i to j shown in Figure 3.5, $k_3 > j > k_2 > i > k_1 > k_m > k_4$. With this visualization in mind, we define some terminologies for various possible paths.

Definition 1. *The path $(i, k_1, k_2, ..., k_m, j)$ is:*

- *an up sequence, if $i < k_1 < k_2 < ... < k_m < j$*
- *a down sequence, if $i > k_1 > k_2 > ... > k_m > j$*
- *a valley sequence, if $i > k_\ell$ and $j > k_\ell$ for all ℓ*
- *a hill sequence, if $i < k_\ell$ and $j < k_\ell$ for all ℓ*
- *a short valley sequence, if $m = 1$ and $i, j > k_1$*

- *a short hill sequence, if $m = 1$ and $i, j < k_1$*

Lemma 9. *If the shortest path from i to j, $(i, k_1, k_2, ..., k_m, j)$, is a valley sequence, the shortest distance from i to j is given by d_{ij} at the end of the SFWP.*

Proof. Proof is by induction on the path length, where the path length is defined to be the number of edges on the path. For our basis we assume $m = 1$, that is, the shortest path from i to j is a short valley sequence. Then the edges (i, k_1) and (k_1, j) are the shortest paths from i to k_1 and k_1 to j, respectively. Since the triple operation $\min\{d_{ij}, d_{ik_1} + d_{k_1j}\}$ is performed in the SFWP when $k = k_1$, the basis is correct.

Assume that the lemma holds for some m. Let the shortest path from i to j be $(i, k_1, k_2, ..., k_m, k_{m+1}, j)$. Let $k_\ell = \max\{k_1, k_2, ..., k_m, k_{m+1}\}$. Then the first sub-path $(i, k_1, ..., k_\ell)$ is a valley sequence if $\ell > 2$, a short valley sequence if $\ell = 2$, and a single edge if $\ell = 1$. Similarly, the second sub-path of $(k_\ell, k_{\ell+1}, ...k_m, k_{m+1}, j)$ is either a valley sequence, or a short valley sequence, or an edge. Since the path lengths of both sub-paths are less than or equal to m, by the induction hypothesis, d_{ik_ℓ} and $d_{k_\ell j}$ give the shortest distances from i to k_ℓ and k_ℓ to j, respectively. Thus $d_{ik_\ell} + d_{k_\ell j}$ gives the shortest distance from i to j by the argument in the basis above (see Figure 3.6). □

Lemma 10. *If the shortest path from i to j, $(i, k_1, k_2, ..., k_m, j)$, is a hill sequence, the shortest distance from i to j is given by d_{ij} at the end of the SBWP.*

Proof. Similar to the proof of Lemma 9. □

Whenever d_{ij} is updated by the triple operation, we assume a hypothetical edge (i, j) that has the cost d_{ij}. We describe this process as *reducing* the shortest path from i to j to the hypothetical edge (i, j). Note that if $(i, j) \in E$ then this simply means that $cost(i, j)$ is not the shortest possible distance from i to j and the new hypothetical edge represents a path of a shorter distance.

We now define two additional sequences. The visualization of these sequences are shown in Figure 3.7.

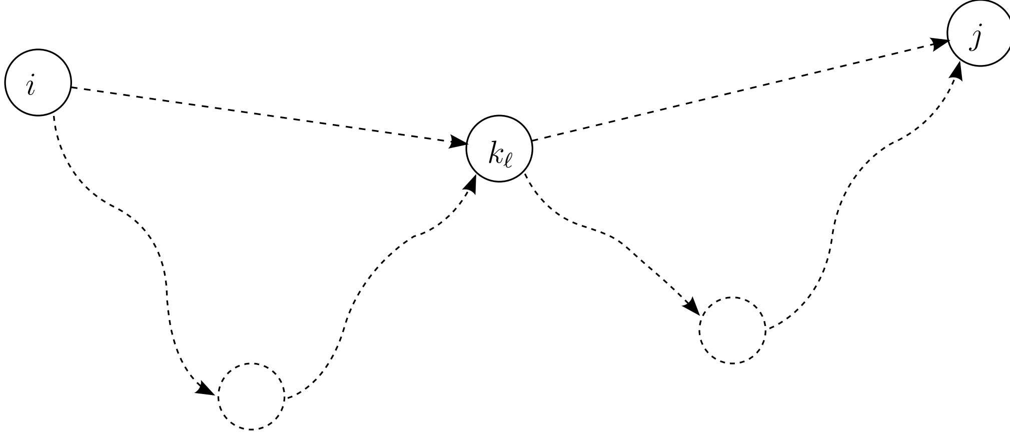

Figure 3.6: The path from i to j is a valley sequence that consists of sub-paths that are also valley sequences.

Definition 2. *The path* $(i, k_1, k_2, ..., k_\ell, ..., k_m, j)$ *is:*

- *an extended valley sequence, if* $(i, ..., k_\ell)$ *is a valley sequence and* $(k_\ell, ..., j)$ *is an up sequence such that* $i < k_\ell$*, or if* $(i, ..., k_\ell)$ *is a down sequence and* $(k_\ell, ..., j)$ *is a valley sequence such that* $k_\ell > j$*.*

- *an extended hill sequence, if* $(i, ..., k_\ell)$ *is a hill sequence and* $(k_\ell, ..., j)$ *is down sequence such that* $i > k_\ell$*, or if* $(i, ..., k_\ell)$ *is an up sequence and* $(k_\ell, ..., j)$ *is a hill sequence such that* $k_\ell < j$*.*

Lemma 11. *If the shortest path from* i *to* j*,* $(i, k_1, k_2, ..., k_m, j)$*, is one of the following three sequences, the value of* d_{ij} *gives the shortest distance from* i *to* j *at the end of the LFWP.*

(1) an up sequence

(2) a down sequence

(3) an extended valley sequence

Proof. We prove (1) by induction. To prove the basis we assume that the up sequence is simply the edge (i, j). For the induction hypothesis, assume

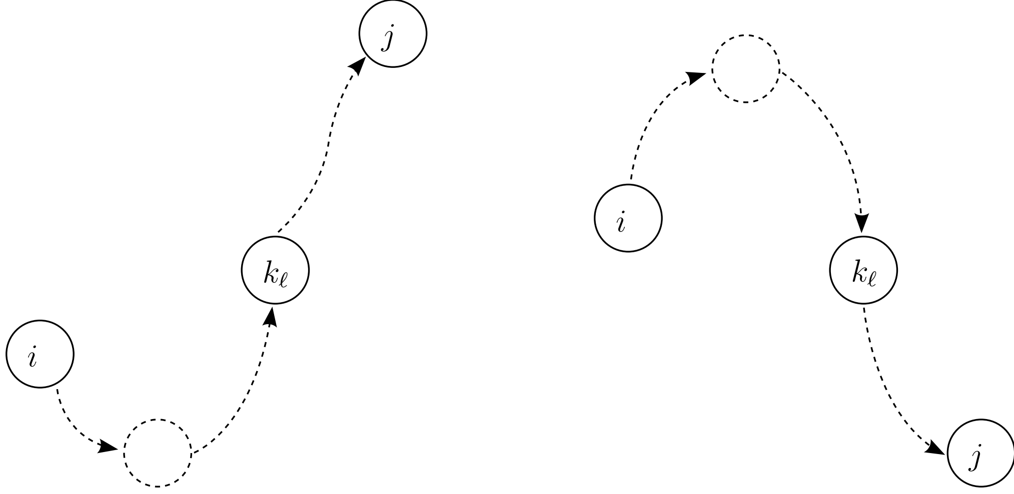

Figure 3.7: An extended valley sequence and an extended hill sequence.

that d_{ik_m} gives the shortest distance from i to k_m after the FWP. Then d_{ij} is given by $d_{ik_m} + d_{k_mj}$, which is obviously true since d_{k_mj} is the edge cost of (k_m, j) and the edge from k_m to j is on the shortest path. The proof for (2) is similar. Note that for (1), the shortest distances are calculated in the order of $d_{ik_1}, d_{ik_2}, ..., d_{ik_m}, d_{ij}$, and for (2), the shortest distances are calculated in the order of $d_{k_mj}, d_{k_{m-1}j}, ..., d_{k_1j}, d_{ij}$.

To prove (3) we first prove the case of $i < j$. Let k be the minimum vertex on the final up sequence such that $i < k$. Then the path from i to k is a valley sequence, which is reduced to the hypothetical edge (i, k) during the FWP by Lemma 9. Then the path $(i, k, ..., j)$ is an up sequence. Since higher vertices are processed later in the FWP, the computation process is the same as the up sequence. Clearly, the case of $i > j$ is symmetric. □

Lemma 12. *If the shortest path from i to j, $(i, k_1, k_2, ..., k_m, j)$, is one of the following three sequences, the value of d_{ij} gives the shortest distance from i to j at the end of the LBWP.*

(1) a down sequence

(2) an up sequence

(3) an extended hill sequence

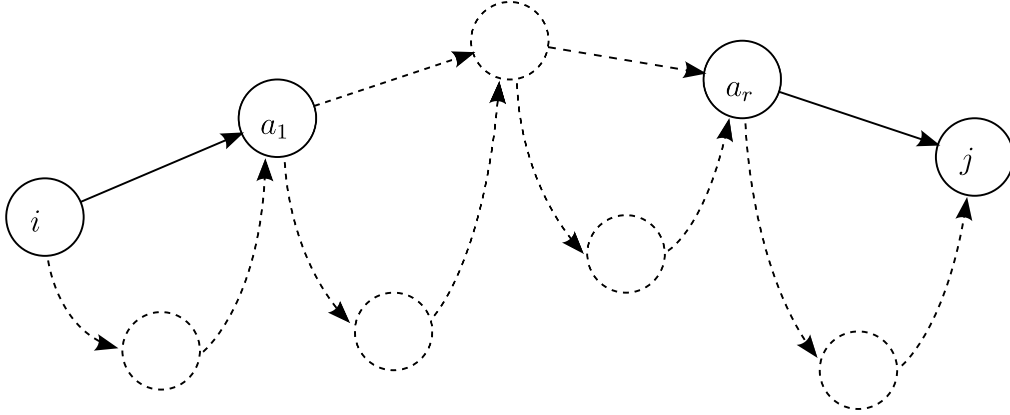

Figure 3.8: Valley sequences and extended valley sequences are reduced to hypothetical edges by the SFWP.

Proof. Similar to the proof of Lemma 11. □

Lemma 13. *The execution of the SFWP followed by the LBWP, symbolized by (SFWP, LBWP), computes the shortest distance from i to j in d_{ij}.*

Proof. By Lemma 9, all valley sequences within the shortest path from i to j are reduced to hypothetical edges after the SFWP. In other words, the shortest path is reduced to $(i, a_1, a_2, ..., a_r, j)$ where $a_1, a_2, ..., a_r$ are the end points of the valley sequences (see Figure 3.8). Clearly, the reduced path of $(i, a_1, a_2, ..., a_r, j)$ is an extended hill sequence, an up sequence or a down sequence. Thus by Lemma 12, performing LBWP on the path $(i, a_1, a_2, ..., a_r, j)$ computes the shortest distance from i to j. □

Lemma 14. *The execution of the SBWP followed by the LFWP, denoted by (SBWP, LFWP), computes the shortest distance from i to j in d_{ij}.*

Proof. Similar to the proof of Lemma 13. □

With Lemma 13, we have shown that the modified cascade algorithm given by Algorithm 11 correctly solves the APSP problem, and our review of the serial cascade algorithm is now complete. In the next section we show that the serial cascade algorithm can be mapped onto the square mesh array to give an efficient parallel algorithm.

3.4.2 Cascade Algorithm on the Mesh Array

In order to perform the FWP on the mesh array, we skew the distance matrix, D, both vertically and horizontally, then load from the top and the left, respectively. The distance matrix is skewed and reversed in the exact same manner as described in Section 2.2, as shown in Figure 3.9. Clearly d_{ik} and d_{kj} will meet at $cell(i, j)$ for all $1 \leq k \leq n$. Thus we can perform the triple operation by performing $\min\{v(i, j), l(i, j) + t(i, j)\}$. Additionally, to achieve the cascade effect, when d_{ij} arrives on $cell(i, j)$ either from the left on $l(i, j)$ or the top on $t(i, j)$, we release the value stored on $v(i, j)$ onto $l(i, j)$ and $t(i, j)$, respectively, such that the reduction of the shortest path from i to j that has previously occurred is used for subsequent computation on cells in higher columns and/or rows. This idea of releasing $v(i, j)$ to achieve the cascade effect originates from the algorithm given by Guibas, Kung and Thompson for computing the transitive closure [30]. The pseudo code for the FWP on the mesh array is given by Algorithm 12.

Algorithm 12 The FWP on the mesh array.

1: **for** $s = 1$ to $3n - 1$ **do**
2:     **for all** $1 \leq i, j \leq n$ **in parallel do**
3:         **if** $j = 1$ **then**
4:             $l(i, 1) \leftarrow d_{i,s-i+1}$ /* load data from the left */
5:         **else**
6:             $l(i, j) \leftarrow l(i, j - 1)$ /* data is transferred to the right */
7:         **if** $i = 1$ **then**
8:             $t(1, j) \leftarrow d_{s-j+1,j}$ /* load data from the top */
9:         **else**
10:            $t(i, j) \leftarrow t(i - 1, j)$ /* data is transferred to the bottom */
11:        $v(i, j) \leftarrow min\{v(i, j), l(i, j) + t(i, j)\}$ /* the triple operation */
12:        **if** $s = i + 2j - 2$ **then** /* d_{ij} arrived from the left */
13:            $l(i, j) \leftarrow v(i, j)$ /* release $v(i, j)$ as the new d_{ij} */
14:        **if** $s = 2i + j - 2$ **then** /* d_{ij} arrived from the top */
15:            $t(i, j) \leftarrow v(i, j)$ /* release $v(i, j)$ as the new d_{ij} */

Lemma 15. *d_{ij} arrives on $cell(i, j)$ from the left at step $s = i + 2j - 2$.*

Proof. D is horizontally skewed and reversed such that d_{ij} arrives on the

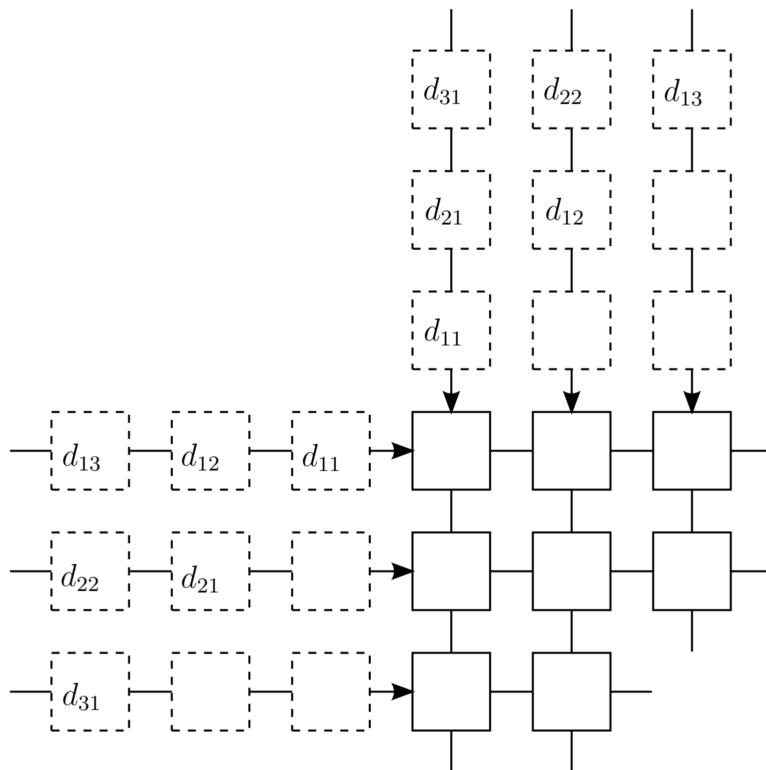

Figure 3.9: Performing the FWP on the mesh array.

left edge of the mesh array (i.e. $cell(i,1)$) at step $s = i + j - 1$. To reach cell $cell(i,j)$, it takes another $j - 1$ communication steps. (Refer to Section 2.2.) □

Lemma 16. *d_{ij} arrives on $cell(i,j)$ from the top at step $s = 2i + j - 2$.*

Proof. Similar to the proof of Lemma 15. □

Lemma 17. *The FWP can be computed in $3n - 1$ communication steps on the mesh array.*

Proof. d_{nn} from the top and d_{nn} from the left takes $3n - 1$ steps to reach $cell(n,n)$. □

Note that for all cells in Algorithm 12, k actually iterates from 1 to n. In terms of the cascade effect, however, we release the value of $v(i,j)$ twice, once onto $l(i,j)$ at the end of the step when the triple operation $min\{d_{ij}, d_{ij} + d_{jj}\}$ is performed, and once onto $t(i,j)$ at the end of the step when $min\{d_{ij}, d_{ii} + d_{ij}\}$ is performed. Thus the cascade effect only holds for k such that $k \leq max\{i,j\}$. If $i < j$, releasing $v(i,j)$ onto $l(i,j)$ corresponds to the SFWP and releasing $v(i,j)$ onto $t(i,j)$ corresponds to the LFWP. If $i > j$ then clearly the SFWP and the LFWP are swapped.

Lemma 18. *The BWP can be computed in $3n - 1$ communication steps on the mesh array.*

Proof. The BWP is clearly symmetrical to the FWP. Skewed and reversed D is loaded from the left and from the bottom. d_{11} from the bottom and d_{11} from the left takes $3n - 1$ steps to reach $cell(1,1)$. □

Theorem 5. *The APSP problem can be solved in $6n - 2$ communication steps on an n-by-n mesh array.*

Proof. The FWP takes $3n - 1$ steps as given in Lemma 17. After the FWP is complete, using the values in $v(i,j)$ as d_{ij}, we perform the BWP on a re-initialized mesh array in another $3n - 1$ steps. By Lemma 13, all shortest distances are stored in $v(i,j)$ after $6n - 2$ communication steps. □

The parallel algorithm described in the proof of Theorem 5 is as close to the original serial cascade algorithm (Algorithm 9) as we can get on the square mesh array. This parallel algorithm is far from optimal, as many cells near the bottom right corner of the mesh array are not performing useful computation at the start of the FWP, and also many cells on the top left corner of the mesh array are not performing useful computation at the start of the BWP. Can we start the computation from both corners of the mesh array at the same time to reduce the number of communication steps, similarly to the parallel matrix multiplication algorithm given in Section 2.2? The answer is "yes", but the difficulty is in proving the correctness of such an algorithm.

In Algorithm 13, the FWP and the BWP are started at the same time from both corners of the mesh array. In each step, instead of the triple operation, we perform the operation $\min\{v(i,j), l(i,j) + t(i,j), r(i,j) + b(i,j)\}$. This is very similar to the idea used in Section 2.2.4 to reduce the number of communication steps required for matrix multiplication. For Algorithm 13 we also make a small enhancement to the mesh array by adding *wraparounds*. That is, we add direct connections from the cells on the top of the mesh array to the bottom of the mesh array and vice versa, such that $t(i,n)$ can be transferred directly to $t(i,1)$ and $b(i,1)$ can be transferred directly to $b(i,n)$. Similarly, we add wraparounds from/to the left edge of the mesh array to/from the right edge of the mesh array. Note that each cell is still limited to just four neighbours, although it can be argued that the mesh array is no longer strictly 2D. Lines 12 and 14 in Algorithm 12 were due to Lemma 15 and 16, respectively. In Algorithm 13, lines 32, 34, 36 and 38 are based on the same set of principles, but with additional modulo operations required because of the wraparounds.

In order to prove the correctness of Algorithm 13, we start with the following definitions:

Definition 3. *$cell(i,j)$ is on the:*

- *FWP frontier, if* $(i+j) \bmod n = (s+1) \bmod n$
- *BWP frontier, if* $(i+j) \bmod n = (2n-s+1) \bmod n$

Algorithm 13 Solve the APSP problem in $3n-1$ steps.

1: **for** $s = 1$ to $3n-1$ **do**
2: **for all** $1 \le i, j \le n$ **in parallel do**
3:   **if** $j = 1$ **then**
4:    **if** $s < n + i$ **then**
5:     $l(i,1) \leftarrow d_{i,s-i+1}$
6:    **else**
7:     $l(i,1) \leftarrow l(i,n)$ /* wraparound from the right edge */
8:   **else**
9:    $l(i,j) \leftarrow l(i,j-1)$
10:   **if** $i = 1$ **then**
11:    **if** $s < n + j$ **then**
12:     $t(1,j) \leftarrow d_{s-j+1,j}$
13:    **else**
14:     $t(1,j) \leftarrow t(n,j)$ /* wraparound from the bottom edge */
15:   **else**
16:    $t(i,j) \leftarrow t(i-1,j)$
17:   **if** $j = n$ **then**
18:    **if** $s < 2n - i + 1$ **then**
19:     $r(i,n) \leftarrow d_{i,2n-i-s+1}$
20:    **else**
21:     $r(i,n) \leftarrow r(i,1)$ /* wraparound from the left edge */
22:   **else**
23:    $r(i,j) \leftarrow r(i,j-1)$
24:   **if** $i = n$ **then**
25:    **if** $s < 2n - j + 1$ **then**
26:     $b(n,j) \leftarrow d_{2n-j-s+1,j}$
27:    **else**
28:     $b(n,j) \leftarrow b(1,j)$ /* wraparound from the top edge */
29:   **else**
30:    $b(i,j) \leftarrow b(i-1,j)$
31:   $v(i,j) \leftarrow min\{v(i,j), l(i,j) + t(i,j), r(i,j) + b(i,j)\}$
32:   **if** $s \equiv (i + 2j - 2) \bmod n$ **then**
33:    $l(i,j) \leftarrow v(i,j)$
34:   **if** $s \equiv (2i + j - 2) \bmod n$ **then**
35:    $t(i,j) \leftarrow v(i,j)$
36:   **if** $s \equiv (3n - i - 2j + 1) \bmod n$ **then**
37:    $r(i,j) \leftarrow v(i,j)$
38:   **if** $s \equiv (3n - 2i - j + 1) \bmod n$ **then**
39:    $b(i,j) \leftarrow v(i,j)$

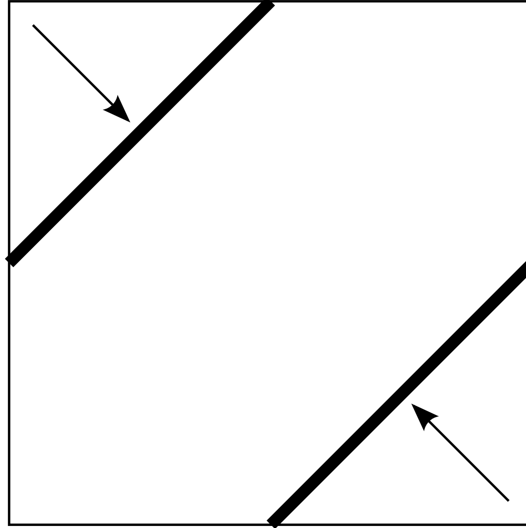

Figure 3.10: The FWP frontier and the BWP frontier on the mesh array.

All cells on the FWP frontier have just started the FWP, and all cells on the BWP frontier have just started the BWP. Figure 3.10 shows both frontiers at $s = n/2$. At $s = n$ the two frontiers will meet at the diagonal given by $i+j = n+1$, and a second set of frontiers will begin at $cell(1,1)$ and $cell(n,n)$ due to the wraparounds on the mesh array. We can observe that the modulo operations in Definition 3 are required due to the wraparounds.

We now consider cells that complete the SFWP in each step. Obviously cells closer to the top left corner of the mesh array finish the SFWP earlier. However, the completion rate of SFWP on the cells do not follow the FWP frontier exactly. Since k sweeps from 1 to $min\{i,j\}$ in the SFWP, cells near the diagonal connecting $cell(1,1)$ to $cell(n,n)$ take longer to complete the SFWP than cells near the top edge or the right edge. This is illustrated in Figure 3.11, which shows the FWP frontier at $s = n$, and the shaded region contains the cells that have completed the SFWP.

Lemma 19. *At $s = 6n/5$, $cell(2n/5, 2n/5)$ has completed the SFWP and is on the first BWP frontier.*

Proof. For simplicity we assume that n is divisible by 5. At step s, for $cell(i,i)$ to have completed the SFWP, i must be less than the shortest dis-

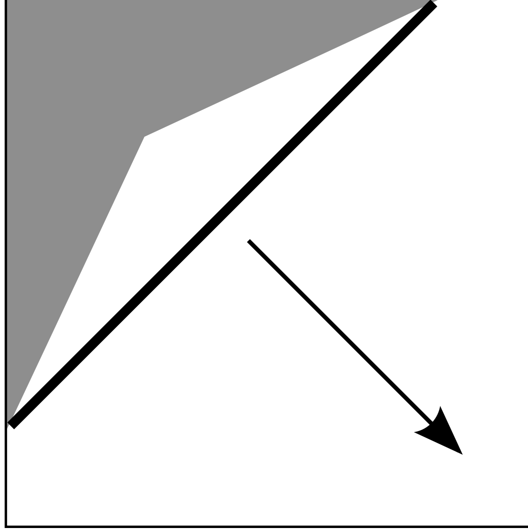

Figure 3.11: The FWP frontier and cells that have completed the SFWP.

tance between $cell(i, i)$ and any point on the FWP frontier since the SFWP would sweep from $k = 1$ to i. At step s, $cell(i, s - i)$ is on the FWP frontier. Thus the maximum i such that $cell(i, i)$ has completed the SFWP at step s is $i = s/3$. Also at step s, $cell(n - s/2, n - s/2)$ is on the first BWP frontier. We can simply solve the simultaneous equation $s/3 = n - s/2$ to retrieve the value for s and the cell location given in the Lemma. □

Lemma 20. *At $s = 9n/5$, $cell(3n/5, 3n/5)$ has completed the SFWP and is on the second BWP frontier.*

Proof. Similarly to the proof of Lemma 19, we can show that at step s, $cell(3n/2 - s/2, 3n/2 - s/2)$ is on the second BWP frontier, and $cell(s/3, s/3)$ has completed the SFWP. Solving the equation $s/3 = 3n/2 - s/2$ yields $s = 9n/5$. □

Finally, to prove the correctness of Algorithm 13, we divide the mesh array into four regions, as illustrated in Figure 3.12:

- Region I: all $cell(i, j)$ such that $i + j \leq 4n/5$

- Region II: all $cell(i, j)$ such that $4n/5 < i + j \leq n$

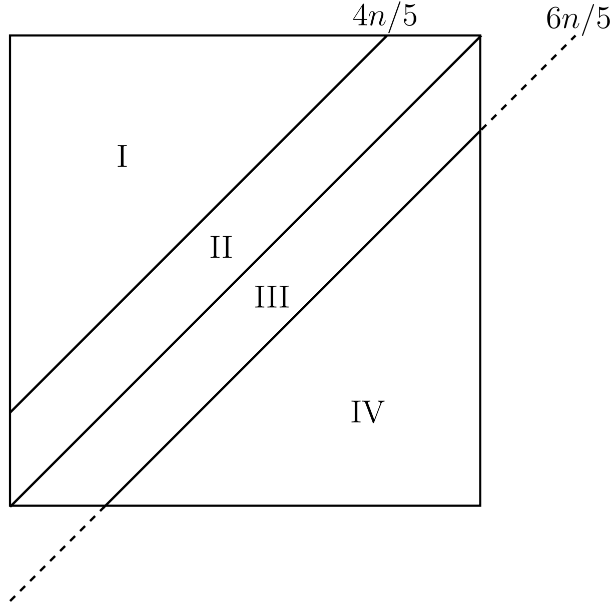

Figure 3.12: The mesh array divided into four regions.

- Region III: all $cell(i, j)$ such that $n < i + j \leq 6n/5$
- Region IV: all $cell(i, j)$ such that $6n/5 < i + j \leq 2n$

Theorem 6. *The APSP problem can be solved in $3n - 1$ communication steps on an n-by-n mesh array with wraparounds.*

Proof. At $s = 6n/5$, by Lemma 19, all cells in Region I have completed the SFWP, and the first BWP frontier is on the border of Region I and II. At $s = 3n - 1$, when $cell(1, 1)$ finally finishes the LBWP, all $v(i, j)$ in Region I have finished (SFWP, LBWP). Thus by Lemma 13, all $v(i, j)$ in Region I holds the shortest distances from i to j at $s = 3n - 1$.

At $s = 9n/5$, by Lemma 20, all cells in Region III have completed the SFWP, and the second BWP frontier is on the border of Region III and IV. Thus after another $n/5 + n - 1$ steps, all cells in Region III have completed (SFWP, LBWP). Since $9n/5 + n/5 + n - 1 = 3n - 1$, at $s = 3n - 1$, all $v(i, j)$ in Region III holds the shortest distances from i to j.

Clearly, Region II and Region IV are symmetrical, with (SBWP, LFWP) being performed in these two regions within $3n - 1$ communication steps. □

Our second contribution to the SP problem is given by Theorem 6. It is actually possible to save one communication step by stopping at $3n - 2$, since $v(i, i) = 0$ for all $1 \leq i \leq n$. Now let us consider an example graph such that the shortest path from vertex 1 to vertex 2 is via vertex n and then vertex $n-1$ i.e. a hill sequence of $\{1, n, n-1, 2\}$. At $s = n+1$, $cell(1, 2)$ has completed the FWP (k sweeps from 1 to n) but this does not reduce the path in any way because the shortest path from vertex 1 to $n-1$ and the shortest path from vertex n to 2 is still unknown. After another n steps, $cell(1, 2)$ has completed another FWP but still no reduction in path has occurred because the shortest paths from 1 to $n - 1$ and from n to 2 are now known, but the information have not yet cascaded to $cell(1, 2)$. Only when d_{1n} and d_{n2} arrives as part of the BWP after another $n - 3$ steps, the shortest distance from vertex 1 to vertex 2 can be computed. Thus our analysis of $3n - 2$ communication steps for Algorithm 13 is sharp.

Chapter IV

Bottleneck Paths (BP)

For the SP problem in Chapter 3, we were only concerned with the cost (or distance) of edges. In this chapter we concern ourselves with the capacity of edges. We can think of each edge as a water pipe. The length of the pipe represents the edge cost, and the thickness of the pipe represents the edge capacity. Consider flowing as much water as possible down a single path from vertex i to vertex j. Clearly, the amount of water that we can flow down a single path from i to j will be restricted by the thinnest pipe on the path. In other words, the thinnest pipe on the path is the bottleneck for the path.

The problem of finding the path that gives us the biggest possible bottleneck value is commonly known as the Bottleneck Paths (BP) problem, which was first introduced by Pollack [51] as the "maximum capacity" problem. Hu proved that on undirected graphs, the All Pairs BP (APBP) problem can be solved in $O(n^2)$ time bound [35]. It has also been shown that FMMOR can be utilised to solve the APBP problem on graphs with real edge capacities in deeply sub-cubic time complexities [59, 79]. Duan and Pettie provided the current best time bound of $\tilde{O}(n^{2.687})$ for solving the APBP problem [19]. The Single Source BP (SSBP) problem can be solved with a simple modification to the Dijkstra's algorithm, whereby in each iteration the vertex that is reachable with the highest capacity is chosen, rather than the vertex that is reachable with the shortest distance. Thus the SSBP problem can be solved in $O(m + n \log n)$ time, and subsequently the APBP problem can also be solved in $O(mn + n^2 \log n)$ time.

Our contribution to the BP problem comes in the form of a new problem, which we call the Graph Bottleneck (GB) problem. As evident from the name of the problem, the GB problem is to find the bottleneck of the entire graph. We define the graph bottleneck as the smallest bottleneck out of all bottleneck paths. Clearly, it is straightforward to solve the GB problem

by first solving the APBP problem, then simply identifying the smallest bottleneck path. We show that we can solve the GB problem much faster than solving the APBP problem with a simple binary search.

Although the new algorithm that we provide for the GB problem is simple, the improvement in time complexity is significant and the practical meaning for system maintenance is important. The bottleneck of the entire graph highlights the link that may experience the greatest amount of pressure when the network is under heavy load, and also edges with smaller capacities can be considered to be useless.

Before we present our contribution to the topic of BP problem in Section 4.2, we review the bottleneck semi-ring in Section 4.1, which is similar to the distance semi-ring that we discussed earlier in Section 3.1. The bottleneck semi-ring is to the BP problem as the distance semi-ring is to the SP problem.

4.1 Bottleneck Semi-ring

Let $B = \{b_{ij}\}$ be the bottleneck matrix, where $b_{ij} = cap(i, j)$ if $(i, j) \in E$, and 0 otherwise. There is no limit on the amount of flow from a vertex to itself, hence $b_{ii} = \infty$ for all $1 \leq i \leq n$.

We then define the (max, min)-product, denoted by $*$, such that the (max, min)-product of $Z = X * Y$ is given by:

$$z_{ij} = \max_{k=1}^{n}\{\min \{x_{ik}, y_{kj}\}\}$$

If we were to compute $B^2 = B * B$, clearly, b^2_{ij} is the maximum possible bottleneck value from vertex i to vertex j with paths of lengths up to 2, and it follows that B^{n-1} will give us the solution to the APBP problem.

Similarly to the distance semi-ring in Section 3.1, $(\mathbb{R}, max, min, 0, \infty)$ is a semi-ring, which we refer to as the bottleneck semi-ring. If M is the set of all possible n-by-n bottleneck matrices, O is the zero bottleneck matrix that has 0 for all elements, and I is the identity bottleneck matrix that has ∞ for all diagonal elements and 0 for all other elements, then $(M, +, \cdot, O, I)$ is a semi-ring, which we refer to as the bottleneck matrix semi-ring. In the bottleneck matrix semi-ring, $+$ is the component-wise max operation and $\cdot$

is the (max, min)-product. The closure of the bottleneck matrix B in the bottleneck matrix semi-ring, denoted by B^*, is given by:

$$B^* = \sum_{k=0}^{n-1} B^k$$

B^* is obviously the solution to the APBP problem, and can be computed in the same time bound as computing the (max, min)-product, as explained in Section 3.1. The current best time bound of $\tilde{O}(n^{2.687})$ for solving the APBP problem given by Duan and Pettie is actually the time taken to perform the witnessed (max, min)-product [19].

4.2 The Graph Bottleneck (GB) Problem

For a network designer it is useful to know which edge(s) in the network is the cause of the smallest bottleneck value out of all bottleneck paths. Note that this is not just a simple matter of taking the edge with the smallest capacity, as edges of small capacities may never be used for any of the bottleneck paths. In fact, finding the solution to the GB problem can highlight the redundant edges, that is, once we know the graph bottleneck, we know that all edges with capacities less than the graph bottleneck are simply not part of any of the bottleneck paths.

As discussed before, the APBP problem, and hence the GB problem, can be solved in $\tilde{O}(n^{2.687})$ or $O(mn + n^2 \log n)$ time bounds. Intuitively, it is wasteful to compute the bottleneck paths for all pairs of vertices when we are only interested in the bottleneck path(s) with the smallest bottleneck value. If we define the graph bottleneck to be 0 for graphs that are not strongly connected, then we show that the GB problem can be solved with a simple binary search.

Let t be the number of distinct edge capacities. We sort the distinct edge capacities and map them to integers from 1 to t. We can do this without any loss of generality as the only operations performed with the edge capacities are comparisons. Then with Algorithm 14 we perform binary search between 1 and t to find the graph bottleneck.

Lemma 21. *Algorithm 14 correctly solves the GB problem on directed graphs*

Algorithm 14 Solve the GB problem without solving the APBP problem.

```
1: α ← 0
2: β ← t + 1
3: while β − α > 0 do
4:     h ← ⌊(α + β)/2⌋
5:     if α = h then
6:         break
7:     G′ ← remove all edges from the graph with capacities less than h
8:     if G′ is strongly connected then
9:         α ← h
10:    else
11:        β ← h
12: α is the graph bottleneck
```

with real edge capacities.

Proof. If G' is not strongly connected, this means the graph bottleneck must be less than h. If G' is strongly connected, this means the graph bottleneck must be greater than or equal to h. Correctness follows immediately. If the original graph is not strongly connected, the value of α will never change and $\alpha = 0$ will be returned. □

Theorem 7. *The GB problem on directed graphs with real edge capacities can be solved in $O(m \log n)$ time.*

Proof. In each iteration of Algorithm 14 we use Tarjan's algorithm to determine strongly connected components in $O(m)$ time [74]. Binary search takes $\log t$ steps, resulting in $O(m \log t)$. In the worst case $t = O(n^2)$ hence the worst case time complexity of Algorithm 14 is $O(m \log n)$. □

We conclude this chapter with our small contribution to the topic of BP problem given by Theorem 7. Note that the time complexity of $O(m \log n)$ is significantly less than both $\tilde{O}(n^{2.687})$ and $O(mn + n^2 \log n)$.

Chapter V

Shortest Paths for All Flows (SP-AF)

In Chapter 1, we provided an example of a practical application for the SP-AF problem by modelling a computer network as a graph. In fact, the SP-AF problem can be easily applied to any optimization problems that involve some form of networks, where both the flow amount and the cost of the flow need to be considered. For example, let us consider transporting raw wood trunks from a forest to the timber mill on trucks of various sizes. The cost of transportation corresponds to the distance travelled. Then for all truck sizes, finding the shortest path that can legally accommodate each truck size will assist in reducing the overall transportation cost of wood trunks. Clearly computer networks and transportation/logistics are just two examples of many potential real life situations where there may exist varying flow requirements from one location to another, and the problem of finding the shortest path for a given flow value may arise.

The SP-AF problem is related to the Bi-objective Shortest Paths (BSP) problem. Instead of considering just the distance in the SP problem, or just the capacity in the BP problem, we are considering both objectives in the SP-AF problem. In the most commonly studied BSP problems the aim is to minimize both objectives [5, 54, 48]. A practical application for such a BSP problem would be road travel. Taking the motorway could get us to a destination faster, but we may have to travel a longer distance and end up using more fuel. We could save on fuel by taking a short-cut via some narrower roads, but the overall time taken for the trip could be longer. Thus we wish to minimize both time taken and fuel spent, but in order to minimize one we must sacrifice the other. BSP problems where the aim is to minimize one objective while maximizing the second objective have also been studied in the past [34, 50]. The key difference between such BSP problems and the SP-AF problem is that in the SP-AF problem we wish to compute and

explicitly list the shortest paths for all possible flow values, whereas in BSP problems the focus is on finding just one or some pre-defined number of solutions as per the problem definition.

For the SP-AF problem, we represent each path by the (d, f) pair, where d is the total distance of the path and f is the bottleneck of the path. We refer to such a pair as the *df*-pair. Then to solve the SP-AF problem, we must find all maximal[1] *df*-pairs between pairs of vertices. That is, if (d, f) and (d', f') are two *df*-pairs for two distinct paths between the same two vertices such that $d < d'$, we keep (d', f') *iff* $f < f'$. In other words, a longer path is only kept if it can accommodate a greater flow.

Let t be the number of distinct edge capacities in our input graph, G. Assume that the edge capacities are sorted in increasing order. Let b_1 and b_2 be two consecutive edge capacities such that $b_1 < b_2$. If the flow requirement, f, is such that $b_1 < f < b_2$, then b_2 is the minimum edge capacity that can flow f. Hence to solve the SP-AF problem, it suffices to solve the SP problem just for all t capacities. In other words, for any pair of vertices, there will be $O(t)$ maximal *df*-pairs. We refer to these t distinct capacities as maximal flows. As explained in Section 4.2 we can treat the maximal flows as integers ranging from 1 to t without any loss of generality.

Algorithm 15 A straightforward method of solving the SP-AF problem.

1: **for all** maximal flow values as f **do**
2: Remove all edges (i, j) from E such that $cap(i, j) < f$
3: Solve the SP problem on the remaining sub-graph
4: Retrieve maximal *df*-pairs from all results

A very straightforward method of solving the SP-AF problem is given by Algorithm 15. Clearly we can take any algorithm for solving the SP problem and execute it t times, once for each maximal flow value, to solve the SP-AF problem.

Before introducing the algorithms for solving a variety of SP-AF problems more efficiently than the straightforward method given by Algorithm 15, we first give a formal mathematical definition of the SP-AF problem in Section

[1] Also known as Pareto optimal.

5.1. In this section we define a new semi-ring called the distance/flow semi-ring and show that this new semi-ring not only helps us in defining the problem but also can be used to solve the SP-AF problem. Then we divide the SP-AF problem into the Single Source SP-AF (SSSP-AF) problem and the All Pairs SP-AF (APSP-AF) problem, as is commonly done in graph path problems, and we provide a range of efficient algorithms.

5.1 Distance/flow Semi-ring

In Sections 3.1 and 4.1 we described the distance semi-ring and the bottleneck semi-ring, respectively. In this section, we combine them to make a composite semi-ring which we call the distance/flow semi-ring. Let us start by defining the ordering between *df*-pairs.

Definition 4. *Let (d, f) and (d', f') be two df-pairs. Then the merit order is defined as:*

$$(d, f) \leq_m (d', f') \Leftrightarrow d \geq d' \wedge f \leq f'$$

Note that $(d, f) <_m (d', f') \Leftrightarrow ((d, f) \leq_m (d', f')) \wedge ((d, f) \neq (d', f'))$. Hence from Definition 4 we can derive the meanings of $=_m, \neq_m$ and $\|_m$ as follows:

$$\begin{aligned}
(d, f) \;=_m\; (d', f') &\Leftrightarrow ((d, f) \geq_m (d', f')) \wedge ((d, f) \leq_m (d', f')) \\
&\Leftrightarrow d = d' \wedge f = f' \\
(d, f) \;\neq_m\; (d', f') &\Leftrightarrow d \neq d' \vee f \neq f' \\
(d, f) \;\|_m\; (d', f') &\Leftrightarrow ((d, f) \not\leq_m (d', f')) \wedge ((d, f) \not\geq_m (d', f')) \\
&\Leftrightarrow (d < d' \vee f > f') \wedge (d > d' \vee f < f') \\
&\Leftrightarrow (d < d' \wedge f < f') \vee (d > d' \wedge f > f')
\end{aligned}$$

Intuitively, $(d, f) >_m (d', f')$ means that the path represented by (d, f) is strictly better since the path can support flow amounts of at least f' while being shorter, or can support a greater flow with path distance at most d'. Clearly all maximal *df*-pairs are incomparable under the merit order, denoted by $\|_m$.

Definition 5. *Let (d, f) and (d', f') be two df-pairs. Then the natural order is defined as:*

$$(d, f) \leq_n (d', f') \Leftrightarrow d \leq d' \wedge f \leq f'$$

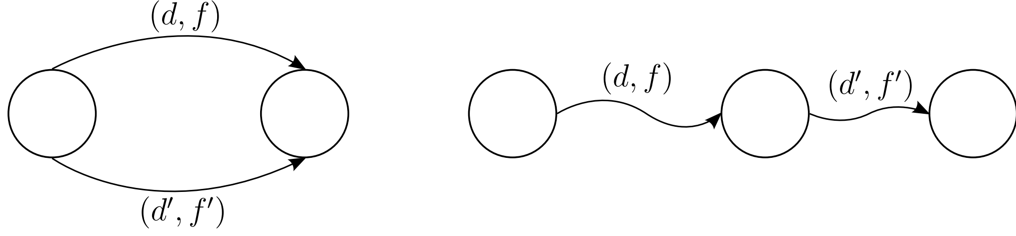

Figure 5.1: An illustration of addition and multiplication of *df*-pairs.

Note that if $(d, f) \parallel_m (d', f')$, then $((d, f) <_n (d', f')) \vee ((d, f) >_n (d', f'))$. The natural order provides ordering of maximal *df*-pairs by their distances. This ordering is useful in solving the SP-AF problem as we will see later on when we introduce various algorithms that rely on this ordering for correctness. Let us now define the addition and multiplication of *df*-pairs.

Definition 6. *The addition of two df-pairs is defined as:*

$$(d, f) + (d', f') = \begin{cases} \{(d, f)\} & \text{if } (d, f) \geq_m (d', f') \\ \{(d', f')\} & \text{if } (d, f) <_m (d', f') \\ \{(d, f), (d', f')\} & \text{if } (d, f) \parallel_m (d', f') \end{cases}$$

The multiplication of two df-pairs is defined as:

$$(d, f) \cdot (d', f') = (d + d', \min \{f, f'\})$$

As shown in Figure 5.1, the addition of two *df*-pairs is equivalent to comparing two parallel paths. We either take the path that is better (i.e. higher in the merit order), or we take both paths. The multiplication of two *df*-pairs is equivalent to combining two serial paths into one. Note that by adding two *df*-pairs, we end up with a set of *df*-pairs, that either contains one *df*-pair that is higher in the merit order, or two *df*-pairs that are incomparable under the merit order. A set of *df*-pairs simply represents one or more parallel paths from a source vertex to a destination vertex. We can now give a formal definition for the SP-AF problem based on sets of *df*-pairs.

Definition 7. *The SP-AF problem is to find the sets of all maximal df-pairs (paths) between pairs of vertices.*

We now define the addition of two sets of *df*-pairs, denoted by $+$. Let x and y be two sets of *df*-pairs. Let $z = x + y$. Then z is the set of all maximal *df*-pairs from $x \cup y$. We can compute $z = x + y$ in $O(|x| + |y|)$ as shown in Algorithm 16. In this algorithm, and in all subsequent algorithms, a set of *df*-pairs is represented by a list of *df*-pairs sorted by the natural order. The operation $a \Leftarrow x$ in the algorithm means the first *df*-pair in x is removed and assigned to a. If x is an empty set, $a \Leftarrow x$ results in $a = null$.

Algorithm 16 Add two sets of *df*-pairs.

```
1:  z ← {}
2:  a ⇐ x, b ⇐ y
3:  while (a ≠ null) and (b ≠ null) do
4:      if a ||_m b then
5:          if a <_n b then
6:              append a to z, a ⇐ x
7:          else
8:              append b to z, b ⇐ x
9:      else if a >_m b then
10:         b ⇐ y
11:     else if b >_m a then
12:         a ⇐ x
13:     else /* a =_m b */
14:         append a to z, a ⇐ x, b ⇐ y
15: if a ≠ null then
16:     append a to z, append x to z (if x ≠ {})
17: if b ≠ null then
18:     append b to z, append y to z (if y ≠ {})
```

Theorem 8. *Algorithm 16 computes $x + y$ in $O(|x| + |y|)$ time, where x and y are both sets of df-pairs.*

Proof. We prove that the set of *df*-pairs, z, accumulates incomparable *df*-pairs in natural order. Observe that *df*-pairs that are lower in the merit order are discarded at lines 10 and 12. We discard *df*-pairs until $a \parallel_m b$ or

$a =_m b$, at which point we append one *df*-pair to z (at lines 6 or 8 or 14). This ensures that all resulting *df*-pairs in z are incomparable under the merit order. When appending a *df*-pair to z we ensure that the smaller *df*-pair in natural order is appended (line 5). This ensures that all *df*-pairs in z are sorted in natural order. $O(|x| + |y|)$ is obvious since at least one of x or y becomes smaller in each iteration and the while loop finishes when either one becomes empty. □

Let us now define the product of sets of *df*-pairs, denoted by $\cdot$. Let $z = x \cdot y$ where x and y are sets of *df*-pairs. The $\cdot$ operation can be performed as follows. We multiply each *df*-pair in x with each *df*-pair in y then discard all non-maximal *df*-pairs. Then z is the resulting set of maximal *df*-pairs. A straightforward method to compute $x \cdot y$ would take $O(|x||y|)$, but we show that this can be performed in $O(|x| + |y|)$ time by using Algorithm 17.

Algorithm 17 Multiply two sets of *df*-pairs.

```
1:  z ← {}
2:  a ⇐ x, b ⇐ y
3:  while (a ≠ null) and (b ≠ null) do
4:      append a · b to z
5:      let a = (d_a, f_a)
6:      let b = (d_b, f_b)
7:      if f_a < f_b then
8:          a ⇐ x
9:      else if f_b < f_a then
10:         b ⇐ y
11:     else/* f_a = f_b */
12:         a ⇐ x, b ⇐ y
```

Theorem 9. *Algorithm 17 computes $x \cdot y$ in $O(|x| + |y|)$ time, where x and y are sets of df-pairs.*

Proof. Suppose we have some accumulation of incomparable *df*-pairs in natural order. If $f_a < f_b$, a can no longer multiply with remaining *df*-pairs in y to generate an incomparable *df*-pair, hence a is discarded (line 8). Similar reasoning applies to the case of $f_a > f_b$, and it follows that if $f_a = f_b$, both

can be discarded. Time complexity is obvious. Note that we can base the comparisons on the d values of *df*-pairs instead of the f values. □

Note that both Algorithms 16 and 17 work even if x and y contain non-maximal *df*-pairs as long as both are sorted in natural order. In the SP-AF problem, however, we can assume that all *df*-pairs in x and y are maximal.

Having defined the addition and multiplication of sets of *df*-pairs, we can finally move onto defining the distance/flow semi-ring, and show that the distance/flow semi-ring is in fact a closed semi-ring.

Theorem 10. *Let S be the set of all possible sets of df-pairs. Let $o = \{(\infty, 0)\}$ and $i = \{(0, \infty)\}$. Then $(S, +, \cdot, o, i)$ is a closed semi-ring.*

Proof. To prove that $(S, +, \cdot, o, i)$ is a closed semi-ring, we start by defining the operator $\times$ and the function p. Let $x \in S$ and $y \in S$. Then $x \times y$ is the Cartesian product of the sets x and y, that is, the set of *df*-pairs resulting from all products of all *df*-pairs from x and y. Note that $x \times y$ may contain non-maximal *df*-pairs. p is a function of the set of *df*-pairs that retrieves all maximal *df*-pairs from the given set. Thus $p(x \times y) = x \cdot y$ and $p(x \cup y) = x + y$. The commutative, distributive and associative natures of the Cartesian product, $\times$, and the union, $\cup$, of sets are widely known. Also note that $p(p(x)+y) = p(x+y)$ and $p(p(x)\cdot y) = p(x\cdot y)$, since a non-maximal *df*-pair in a set simply cannot combine with another *df*-pair to result in a maximal *df*-pair during the $+$ or $\cdot$ operations between sets of *df*-pairs. We now prove that $(S, +, \cdot, o, i)$ is a closed semi-ring by showing that each of the individual properties of a closed semi-ring is satisfied:

- $(S, \cdot, i)$ is a monoid: The product of two sets of *df*-pairs result in a set of *df*-pairs. Since by definition S is the set of all possible *df*-pairs, $x \cdot y \in S$. Hence $(S, \cdot, i)$ is closed. The $\cdot$ operation is associative as shown by the equation $(x \cdot y) \cdot z = p((x \cdot y) \times z) = p(p(x \times y) \times z) = p(x \times y \times z) = p(x \times p(y \times z)) = p(x \times (y \cdot z)) = x \cdot (y \cdot z)$. From the equation $(d, f) \cdot (0, \infty) = (d + 0, \min \{f, \infty\}) = (d, f)$, it is clear that i serves as the identity for the $\cdot$ operation. Thus all three conditions for $(S, \cdot, i)$ to be a monoid have been satisfied.

- $(S, +, o)$ is a monoid: By similar reasoning to the above, $(S, +, o)$ is closed, the $+$ operation is associative, and o serves as the identity for the $+$ operation.

- The $+$ operation is commutative: This can be shown by the equation $x + y = p(x \cup y) = p(y \cup x) = y + x$.

- The $+$ operation is idempotent: This can be shown by the equation $x + x = p(x \cup x) = p(x) = x$.

- o is the annihilator for the $\cdot$ operation: From the equation $(d, f) \cdot (\infty, 0) = (d + \infty, \min\{f, 0\}) = (\infty, 0)$, o is clearly the annihilator for the $\cdot$ operation.

- The $\cdot$ operation distributes over the $+$ operation: This can be shown by the equation $x \cdot (y + z) = p(x \times p(y \cup z)) = p(x \times (y \cup z)) = p((x \times y) \cup (x \times z)) = p(p(x \times y) \cup p(x \times z)) = (x \cdot y) + (x \cdot z)$.

- The closure of x, $x^* = i + x + x^2 + ...$ exists in S: From the equation $(0, \infty) + (d, f) = \{(0, \infty)\}$, it is clear that $x^* = i$ and hence exists in S.

□

We refer to the semi-ring defined in Theorem 10 as the distance/flow semi-ring. Similarly to the distance matrix semi-ring and the bottleneck matrix semi-ring in Sections 3.1 and 4.1, respectively, we can also define the distance/flow matrix semi-ring. Let $P = \{p_{ij}\}$ be a distance/flow matrix where each element is a set of *df*-pairs. Let M be the set of all possible n-by-n distance/flow matrices. The zero distance/flow matrix, O, has $\{(\infty, 0)\}$ for all elements. The identity distance/flow matrix, I, has $\{(0, \infty)\}$ for the diagonals and $\{(\infty, 0)\}$ for all other elements. Let $+$ be the component-wise addition of sets of *df*-pairs, and $\cdot$ be the distance/flow matrix multiplication based on Definition 6. Then $(M, +, \cdot, O, I)$ is a semi-ring, which we refer to as the distance/flow matrix semi-ring.

Clearly the closure of P in the distance/flow matrix semi-ring, denoted by P^*, is the solution to the APSP-AF problem. Distance/flow matrix multiplication on an n-by-n distance/flow matrix takes $O(tn^3)$ time, since there are $O(t)$ maximal *df*-pairs in any set of *df*-pairs. Thus the APSP-AF problem can be solved in $O(tn^3)$ time as explained in Section 3.1. Note that this time bound is equivalent to executing Floyd's algorithm t times to solve the APSP problem for each maximal flow value, as given by Algorithm 15, or simply executing Floyd's algorithm once on P to compute the closure in the distance/flow semi-ring, spending $O(t)$ time in each iteration for the $+$ and $\cdot$ operations.

For the parallelisation of the APSP-AF problem, we can utilise our second contribution to the SP problem as given in Section 3.4.2, which was to solve the APSP problem in $3n-2$ communication steps on an n-by-n mesh array. In each communication step, each cell transmits $O(t)$ data (sets of *df*-pairs) to its neighbours, and performs the $+$ and $\cdot$ operations in $O(t)$ time. Thus we can also solve the APSP-AF problem on a square 2D mesh array in exactly $3n-2$ communication steps, resulting in the total computational cost of $O(tn^3)$ for this parallel algorithm.

The main theme of subsequent sections is to improve the general time complexity of $O(tn^3)$ for the APSP-AF problem and also to provide more efficient algorithms for the SSSP-AF problem than Algorithm 15. We start with the SSSP-AF problem in Section 5.2, then move onto the APSP-AF problem in Section 5.3.

5.2 Single Source SP-AF (SSSP-AF)

For the SSSP-AF problem, as per Definition 7, we wish to find the set of all maximal *df*-pairs from a single source vertex, s, to all other vertices in V. In other words, we wish to fully populate an entire row of the distance/flow matrix, $P = \{p_{ij}\}$, where $i = s$ and $1 \leq j \leq n$. In this section we provide algorithms for solving the SSSP-AF problem, firstly on graphs with unit edge costs then on graphs with integer edge costs bounded by c.

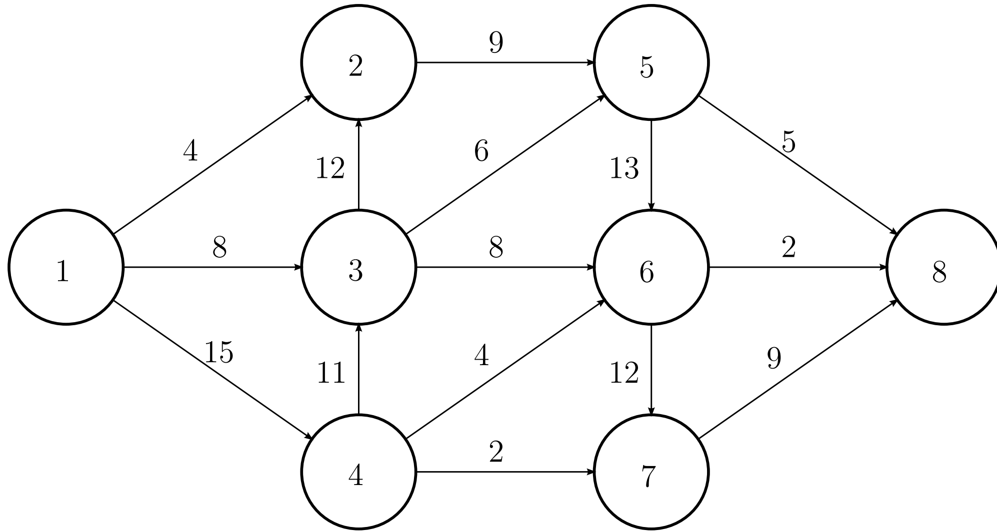

Figure 5.2: A directed graph with unit edge costs. $n = 8$, $m = 15$ and $t = 10$. Capacities are shown beside each edge.

5.2.1 Unit Edge Costs

Solving the SSSP problem on directed graphs with unit edge costs is very straightforward, achieved with a simple breadth-first-search taking $O(m)$ time. Thus the SSSP-AF problem can be solved in $O(tm)$ time using Algorithm 15. In the worst case $t = m = O(n^2)$, which results in the worst case time complexity of $O(n^4)$ for this straightforward method.

Example 1. *Solving the SSSP-AF problem on the example graph given in Figure 5.2 with $s = 4$ will result in the following set of df-pairs to vertex 7:* $p_{4,7} = \{(1, 2), (2, 4), (3, 8), (5, 9)\}$.

Algorithm 18 solves the SSSP-AF problem on graphs with unit edge costs in $O(mn)$ worst case time complexity. This is achieved by exploiting the fact that on graphs with unit edge costs there are $O(n)$ possible distances from s to any other vertices in V. Therefore we design an algorithm around this by visiting each possible distance once, spending $O(m)$ time for edge inspection in each distance, resulting in the $O(mn)$ time bound.

Let us introduce the notation used in Algorithm 18. Let B be an array such that $B[v]$ is the currently known largest bottleneck value from s to v.

Similarly, let L be an array that keeps track of the shortest possible distance such that $L[v]$ is the currently possible shortest distance from s to v. Let T be the Shortest Paths Spanning Tree (SPT) that is an incremental data structure, such that incremental changes are applied to T as we progress through the algorithm. Let Q be an array of sets of vertices such that $Q[i]$ is the set of vertices that may be added to T at distance i from s, where $1 \leq i \leq n$.

In summary, Algorithm 18 starts with s as the root of T. We iterate through all maximal flows as f in increasing order. In each iteration, we remove v from T that can no longer accommodate f and re-add it to T at the next shortest possible distance that can accommodate f.

Lemma 22. *If v exists in T on line 22 of Algorithm 18, $(L[v], B[v])$ is a maximal df-pair for p_{sv}.*

Proof. Proof is by contradiction. Suppose $(L[v], B[v])$ is a non-maximal *df*-pair. Then there must be a *df*-pair $(d, B[v])$ such that $d < L[v]$, or $(L[v], f)$ such that $f > B[v]$. The first case is not possible because we inspect each possible distance from s one by one by incrementing $L[v]$ in each iteration. The second case is not possible because in the inner *for* loop starting from line 17, we inspect all incoming edges such that v is added as the child of the parent that can provide the maximum flow. Thus $(L[v], B[v])$ must be a maximal *df*-pair. □

Theorem 11. *Algorithm 18 correctly solves the SSSP-AF problem in $O(mn)$ worst case time complexity.*

Proof. Correctness follows from Lemma 22. We perform lifetime analysis for determining the time complexity. We start with the key observation that there are $O(n)$ possible distances from s, and each destination vertex can be observed at each possible distance from s exactly once since $L[v]$ is monotonically increasing.

Cutting the vertex v from T or adding it to T both take $O(1)$ time, achieved simply by setting the parent of v to *null* or u, respectively. Since each $O(n)$ vertices can be removed then re-added in each iteration, we have

Algorithm 18 Solve the SSSP-AF problem on graphs with unit edge costs.

```
 1: for i ← 1 to n do
 2:     B[i] ← 0
 3:     L[i] ← 0
 4:     Q[i] ← φ
 5:     p_si ← φ
 6: T ← s
 7: B[s] ← ∞
 8: for all maximal flow f in increasing order do
 9:     for all v ∈ V such that B[v] < f do
10:         if v exists in T then
11:             cut v from T
12:         L[v] ← L[v] + 1
13:         push v to Q[L[v]]
14:     for ℓ ← 1 to n − 1 do
15:         while Q[ℓ] is not empty do
16:             pop v from Q[ℓ]
17:             for all (u, v) ∈ E do
18:                 if L[u] = L[v] − 1 then
19:                     if MIN(cap(u, v), B[u]) > B[v] then
20:                         B[v] ← MIN(cap(u, v), B[u])
21:                         add v to T with u as the parent
22:             if v exists in T then
23:                 append (L[v], B[v]) as a df-pair to p_sv
24:             else
25:                 L[v] ← L[v] + 1
26:                 push v to Q[L[v]]
```

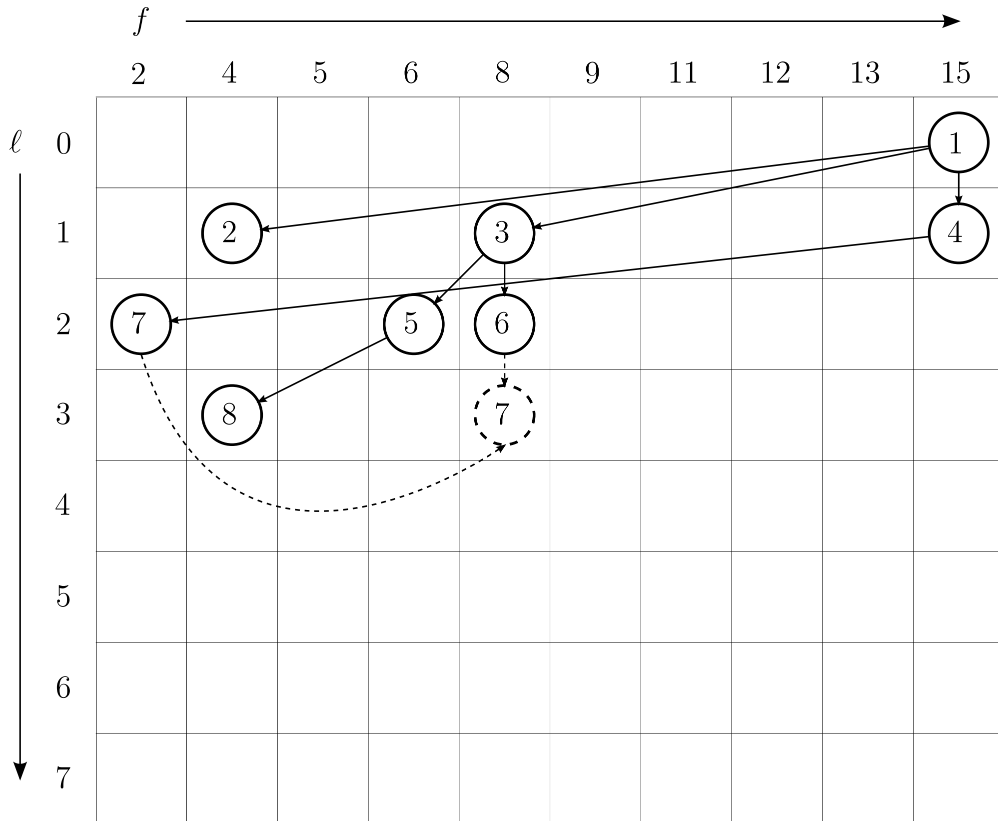

Figure 5.3: Changes to T at iteration $f = 4$.

$O(n^2)$ for all operations involving T. Q can be implemented with a simple linked list data structure, resulting in the total time complexity of all operation involving Q of $O(n^2)$.

Finally for edge inspections starting from line 17, we have a total of $O(m)$ edge inspections for each possible distance from s, and thus $O(mn)$ total edge inspections for the whole algorithm, which subsequently becomes the worst case time complexity of Algorithm 18. □

Example 2. *Figure 5.3 is a visualisation of Algorithm 18 with the example graph shown in Figure 5.2 as the input with* $s = 1$. *At iteration* $f = 4$, *the edge* $(4, 7)$ *is cut from* T *because the path to vertex 7 cannot push flow of* $f = 4$. *The next shortest possible distance from vertex 1 to vertex 7 is 3, with the new bottleneck value of 8.* $L[7]$ *is increased from 2 to 3 and* $B[7]$ *is*

increased from 2 to 8. At the end of the iteration $p_{1,7} = \{(2, 2), (3, 8)\}$.

5.2.2 Integer Edge Costs

Let us now consider solving the SSSP-AF problem on graphs with integer edge costs bounded by c. As mentioned in Chapter 3, the SSSP problem on directed graphs with integer edge costs bounded by c can be solved in $O(m + n \log\log c)$ time [75]. Hence the SSSP-AF problem can be solved in $O(tm+tn \log\log c)$ time. We can also use Algorithm 18 to solve the problem in $O(mnc)$ time, since there are now $O(nc)$ possible distances from s. Our contribution to the SSSP-AF problem is an algorithm with the worst case time complexities of either $O(tm + nc)$ or $O(tm + tn \log (c/t))$ depending on the underlying data structure used to implement the priority queue. We briefly visit Dijkstra's algorithm for solving the SSSP problem since our new algorithm for solving the SSSP-AF problem can be thought of as an extension to the well known Dijkstra's algorithm.

Algorithm 19 Dijkstra's algorithm.

1: $Q, P \leftarrow \phi$
2: insert $(s, 0)$ into Q
3: **while** $Q \neq \phi$ **do**
4:     delete (v, d) from Q such that d is the minimum /* *delete-min* */
5:     **for all** $w \in OUT(v)$ such that $w \neq s$ **do**
6:         $d' \leftarrow d + cost(v, w)$
7:         **if** (w, d^*) in Q for w and some d^* **then**
8:             **if** $d' < d^*$ **then**
9:                 update (w, d^*) to (w, d') in Q /* *decrease-key* */
10:         **else**
11:             insert (w, d') into Q /* *insert* */
12:     append (v, d) to P

Q and P in Algorithm 19 are commonly known as the *frontier set* and the *solution set*, respectively. In each iteration we find and remove the vertex v from Q with the minimum distance d from the source vertex s. This operation is known as the *delete-min* operation. We then inspect all outgoing edges from v and for each edge either update the distances of vertices

that are already in Q, or insert new vertices into Q. The former operation is known as the *decrease-key* operation and the latter is known as the *insert* operation.

The time complexity of Dijkstra's algorithm depends on the time taken to perform these three key operations. The total number of *insert* and *delete-min* operations performed in the algorithm is n, since all n vertices are added then removed from the *frontier set*, Q. The total number of *decrease-key* operations performed is $O(m)$ in the worst case. If a simple linear array is used to implement Q, then each *insert* and *decrease-key* operations will take $O(1)$ time, whereas the *delete-min* operation takes $O(n)$ time, resulting in the total worst case time complexity of $O(m + n + n^2) = O(n^2)$. If Q is implemented with a priority queue such as the Fibonacci heap [25, 9], such that the *insert* and *decrease-key* operations take $O(1)$ time and the *delete-min* operation takes $O(\log n)$ time, then $O(m + n \log n)$ can be achieved.

We now present Algorithm 20 for solving the SSSP-AF problem. For this algorithm we define the (v, d, f) triplet where v is the destination vertex and d and f corresponds to the *df*-pair of the path from s to v. Let Q be the priority queue for the (v, d, f) triplets with d as the key such that the operations performed on Q are limited to *insert*, *decrease-key* and *delete-min* operations. $OUT(v)$ denotes all vertices that are directly reachable from v via a single edge. In summary, we start by initializing Q with $(s, 0, \infty)$. In each iteration we take the (v, d, f) triplet with the smallest d from Q and inspect all out going edges from v. We update Q as necessary, and add (d, f) to the solution set p_{sv} if it is a maximal *df*-pair.

Lemma 23. *Algorithm 20 correctly solves the SSSP-AF problem on directed graphs with integer edge costs and real edge capacities.*

Proof. Proof is by induction. Our induction hypothesis is that in the beginning of each iteration:

1. p_{sv} for all $v \in V$ such that $v \neq s$ contains maximal *df*-pairs

2. For any (v, d, f) triplet in Q, d is the distance of the shortest path possible from s to v that can push f, using only the paths whose vertices are already in p_{sv} for all $v \in V$, except for the end point v.

Algorithm 20 Solve the SSSP-AF problem for integer edge costs.

```
1: Q ← φ
2: insert (s, 0, ∞) into Q
3: while Q ≠ φ do
4:     delete (v, d, f) from Q such that d is the minimum /* delete-min */
5:     for all w ∈ OUT(v) such that w ≠ s do
6:         f' ← MIN(f, cap(v, w))
7:         d' ← d + cost(v, w)
8:         if (w, d*, f') in Q for w, f' and some d* then
9:             if d' < d* then
10:                update (w, d*, f') to (w, d', f') in Q /* decrease-key */
11:        else
12:            insert (w, d', f') into Q /* insert */
13:    if p_sv = φ then
14:        append (d, f) to p_sv
15:    else
16:        let (d_0, f_0) be the last pair in p_sv
17:        if f_0 < f then
18:            if d_0 = d then
19:                delete (d_0, f_0) from p_sv
20:            append (d, f) to p_sv
```

For the induction basis we observe that both are correct before the main *while* loop begins. Suppose both are correct at the beginning of some iteration. Then:

1. Let (d_0, f_0) be the last pair in p_{sv}. Note that *df*-pairs are sorted in increasing order of d. Since we remove the triplet with the minimum value of d in each iteration, $d_0 \leq d$. We append (d, f) to p_{sv} only when $f_0 < f$. Hence the new *df*-pair is also maximal.

2. After (v, d, f) is removed from Q, we inspect all $w \in OUT(v)$. Thus when Q is updated with (w, d', f'), the *df*-pair of (d', f') represents a path from s to w via v. Since (d, f) is added as a maximal *df*-pair into p_{sv} at the end of the iteration, d' is the shortest possible path from s to w that can push f', using only the paths whose vertices are already in p_{sv} for all $v \in V$ except for the end point w.

Thus both of our induction hypotheses are preserved after the iteration. □

Theorem 12. *The SSSP-AF problem can be solved in $O(tm+nc)$ worst case asymptotic time complexity.*

Proof. We use a simple one dimensional bucket structure to implement Q in Algorithm 20, such that the *insert* and *decrease-key* operations can both be performed in $O(1)$ time, and the *delete-min* operation involves scanning the buckets one by one in increasing order of d until a non-empty bucket is found. There are $O(tn)$ (v, d, f) triplets, resulting in $O(tn)$ time bound for the *insert* operation for the whole algorithm. For the *delete-min* operation we have the total time complexity of $O(cn)$ for the whole algorithm since the maximum distance from s to any destination vertex is $O(cn)$. The *decrease-key* operation can occur once for each edge inspection in line 5. We can observe that for each v, $OUT(v)$ can be inspected for each maximal flow value, resulting in the total of $O(tm)$ edge inspections, thus giving us the total time complexity of $O(tm)$ for the *decrease-key* operation. Checking whether a (v, d, f) triplet exists in Q or not in line 8 can be performed in $O(1)$ time simply by maintaining a two dimensional array of size $O(tn)$, for example $I[v][f]$, for each destination vertex v and for each maximal flow f,

such that $I[v][f]$ is set to True when the triplet (v, d, f) is inserted into Q for some d, and set to False when the triplet is deleted from Q. Note that the delete operation performed in line 19 does not propagate further inside p_{sv} due to the strictly increasing property of p_{sv}. Hence for the entire algorithm we have $O(tm + tn + nc) = O(tm + nc)$. □

It is clear that the time complexity of Algorithm 20 depends largely on the data structure used to implement Q. As mentioned earlier, Algorithm 20 can also run in $O(tm + tn \log (c/t))$ time bound by using the data structure called the k-level Cascading Bucket System (CBS) [3, 15] to implement Q. The one dimensional bucket system used in Theorem 12 can be thought of as a 1-level CBS. We now give a brief review of the k-level CBS data structure. An extensive review of this data structure has been given by Takaoka [71].

The k-level CBS is a data structure that supports the required *insert*, *decrease-key* and *delete-min* operations for integer key values. There are k levels, where each level has p buckets, except the highest level which has M/p^{k-1} buckets, such that M is the highest integer key value that the CBS supports. For the SP-AF problem we can set $M = cn$ since we know that the furthest distance between any pair of vertices will be less than cn.

Before any *delete-min* operations are performed on the CBS, we can find the bucket to insert an item with the key value of d with the following equation:

$$d = x_{k-1}p^{k-1} + x_{k-2}p^{k-2} + ... + x_1 p + x_0 \tag{5.1}$$

which we use to find the largest i such that x_i is non-zero. Then the item can be inserted into the x_i-th bucket at level i.

To perform the *delete-min* operation, an active pointer a_i is maintained for each level, for $0 \leq i < k$, such that a_i points to the minimum non-empty bucket at level i. $a_i = p$ means level i is empty. We scan from a_0 to find the lowest level with a non-empty bucket. If $0 \leq a_0 < p$, this means level 0 is non-empty and the *delete-min* operation simply involves removing an item in the bucket pointed to by a_0. The *delete-min* operation gets more complex when the lowest level with a non-empty bucket is higher than level 0. Let j be the lowest non-empty level. Then we must re-distribute the elements in

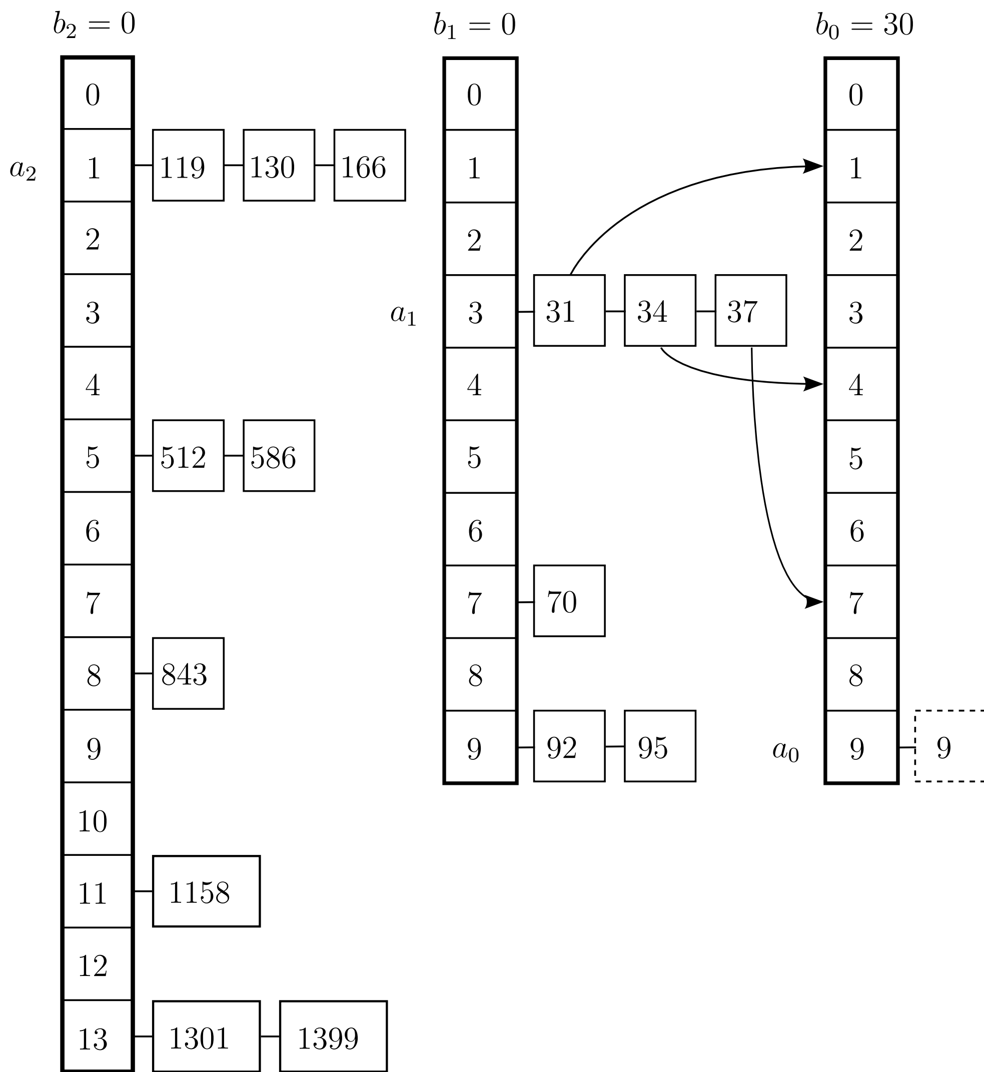

Figure 5.4: Visualization of the k-level CBS with k = 3, p = 10 and M = 1399.

the a_j-th bucket into level $j-1$, then re-distribute the elements in the a_{j-1}-th bucket into level $j-2$ and so on until level 0 is non-empty. This process of repeated redistribution is referred to as cascading. We can observe that once a cascade operation is performed on the CBS we can no longer use equation 5.1 to insert new elements. Therefore we must maintain the base, b_i, for each level i. Before any cascade operations are performed $b_i = 0$ for all $0 \leq i < k$. When a cascade operation is performed, b_i is updated for $0 \leq i \leq j$, where j is the lowest non-empty level found at the start of the cascade operation. The bases are updated with the following recurrence formula:

$$b_{i-1} = b_i + a_i p^i \tag{5.2}$$

The base values can then be used for the *insert* operation since all key values in level i must be greater than or equal to b_i.

Finally, the *decrease-key* operation can be performed by removing the element from the CBS in $O(1)$ time, updating the key value, then performing the *insert* operation with the new key value.

Example 3. *Figure 5.4 shows an illustration of the cascade operation. Once the item with the key value of 9 is removed, the next* delete-min *operation will trigger the cascade operation, resulting in the three items stored in the bucket pointed to by a_1 to be redistributed to level 0. a_1 and a_0 will be updated to point to buckets 7 and 1, respectively. The base value b_0 will be updated to 30. Note that Figure 5.4 shows the old values of a_i and the new values for b_i.*

Let us now move onto the time complexities of the three operations. For the *insert* operation we have $O(k)$ to find the location based on equation 5.1, or to scan the base values b_i from the highest level down to the lowest level. The *decrease-key* operation can be performed in $O(l)$ time, where l is the difference between the initial level and the new level. Observing the scanning effort over the levels to find a non-empty level and then to find a non-empty bucket in the found level, the *delete-min* operation takes $O(k+p)$ if $j < k-1$ such that j is the lowest non-empty level, and $O(k + M/p^{k-1})$ if $j = k-1$.

Theorem 13. *The SSSP-AF problem can be solved in $O(tm + tn \log(c/t))$ worst case asymptotic time complexity.*

Proof. We use the k-level CBS to implement Q in Algorithm 20. There are $O(tn)$ (v, d, f) triplets to be inserted into Q hence we have $O(ktn)$ for the *insert* operations. There can be up to $O(tm)$ *decrease-key* operations but each triplet can move a maximum of k levels, resulting in $O(ktn)$ also for the *decrease-key* operations. For the *delete-min* operation we have $O(ktn + ptn + cn/p^{k-1})$ since the data structure only needs to support key values up to $O(cn)$. We choose $p = (c/t)^{1/k}$ and $k = \log(c/t)$ to implement the CBS, resulting in $O(tn \log(c/t))$ as the time bound for all operations involving Q. Thus we have $O(tm + tn \log(c/t))$ as the time bound. □

For explicit path retrieval, we store the *predecessor* vertex whenever a triplet gets inserted or updated in Q. A predecessor vertex on the path from s to w is the vertex that comes immediately before w. We store v alongside (w, d', f') in Algorithm 20 since we know that v is the vertex that comes immediately before vertex w. Then we can retrieve the explicit path by simply following the predecessor vertices.

There is no clear winner between $O(tm + tn \log\log c)$, $O(tm + nc)$ and $O(tm + tn \log(c/t))$. For $c = O(t)$, $O(tm + nc)$ is the most efficient. For $c = O(t \log t)$, $O(tm + tn \log(c/t))$ gives the best time complexity. For larger values of c, $O(tm + tn \log\log c)$ given by the straightforward method based on Thorup's SSSP algorithm gives the best time complexity. The data structure used in Thorup's algorithm, however, is known to be very complex and impractical for implementation [75]. On the other hand, the CBS (of any number of levels) is practical for implementation. Therefore we conclude this section with a final claim that for practical purposes, we can use the $O(tm + nc)$ algorithm for $c = O(t)$, and for larger values of c, we can use the $O(tm + tn \log(c/t))$ algorithm and still enjoy the logarithmic function over the value of c.

5.3 All Pairs SP-AF (APSP-AF)

We now present a series of efficient algorithms for solving the APSP-AF problem. To solve the APSP-AF problem, we must find the set of maximal df-pairs for all elements in the distance/flow matrix $P = \{p_{ij}\}$ such that

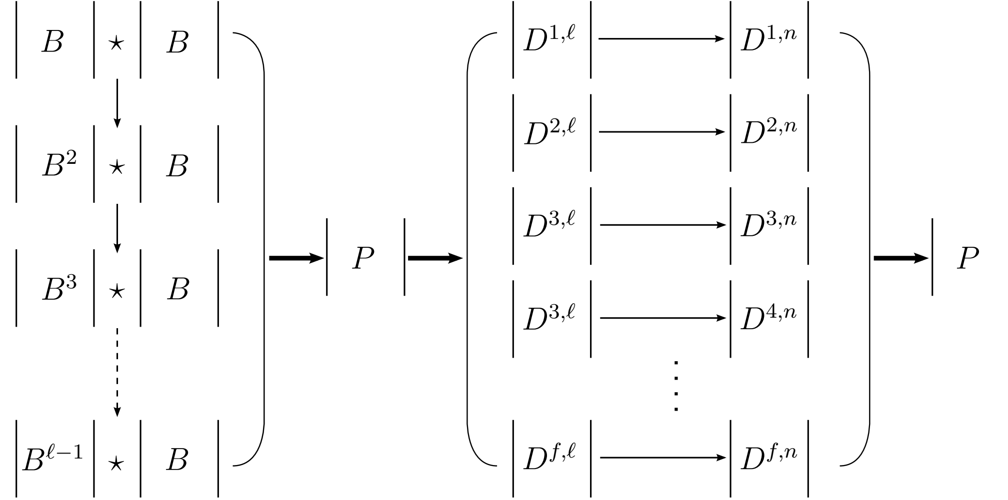

Figure 5.5: An illustration of Algorithm 21.

$1 \leq i, j \leq n$. Similarly to the SSSP-AF problem in Section 5.2, we will consider both unit edge costs and integer edge costs bounded by c.

5.3.1 Unit Edge Costs

We can solve the APSP-AF problem on graphs with unit edge costs in $O(tmn)$ or $O(mn^2)$ time bounds, by executing the $O(tm)$ or the $O(mn)$ SSSP-AF algorithm n times, respectively. For dense graphs where $m = O(n^2)$, the time complexities become $O(tn^3)$ and $O(n^4)$. Here we present an algebraic algorithm that is more efficient than both $O(tn^3)$ and $O(n^4)$.

Our algorithm is derived from two key observations. Firstly, on graphs with unit edge costs, by multiplying the bottleneck matrix, B, one by one, we can retrieve *df*-pairs in each iteration. Suppose we have B^ℓ and we compute $B^{\ell+1}$ by performing the (max, min)-product with B. If $b_{ij}^{\ell} < b_{ij}^{\ell+1}$, then we can retrieve $(\ell+1, b_{ij}^{\ell+1})$ as a *df*-pair because we know that $\ell+1$ is the minimum path distance (length) that can accommodate the flow value of $b_{ij}^{\ell+1}$. Secondly, we can utilise the bridging set theorem, as explained in Section 3.2, by using the framework provided by the AGM algorithm. We present Algorithm 21 that utilises these two key ideas to achieve $\tilde{O}(\sqrt{t}n^{(\omega+9)/4}) = \tilde{O}(\sqrt{t}n^{2.843})$ time bound.

Algorithm 21 Solve the APSP-AF problem on graphs with unit edge costs.

```
    /* Initialization for the acceleration phase */
 1: for i ← 1 to n; j ← 1 to n do
 2:     p_ij ← φ

    /* Acceleration phase */
 3: for ℓ ← 1 to r do
 4:     B^ℓ ← B^{ℓ−1} ⋆ B
 5:     for i ← 1 to n; j ← 1 to n such that i ≠ j do
 6:         f ← b^ℓ_ij
 7:         if f > b^{ℓ−1}_ij then
 8:             append (ℓ, f) to p_ij

    /* Initialization for the cruising phase */
 9: for i ← 1 to n; j ← 1 to n such that i ≠ j do
10:     for all x in p_ij do
11:         if x ≠ φ then
12:             let x = (d, f)
13:             d^{f,ℓ}_ij ← d
14:         else
15:             d^{f,ℓ}_ij ← ∞

    /* Cruising phase */
16: for all maximal flow values as f do
17:     perform the cruising phase of Algorithm 7 on D^{f,ℓ}

    /* Finalization */
18: let D^f be D^{f,n} from the result of the cruising phase
19: for i ← 1 to n; j ← 1 to n such that i ≠ j do
20:     for all maximal flow f in increasing order do
21:         d ← d^f_ij
22:         let the last pair of p_ij be x = (d′, f′)
23:         if x = φ or (f > f′ and d < ∞) then
24:             if d = d′ then
25:                 replace x with (d, f)
26:             else
27:                 append (d, f) to p_ij
```

An illustration of Algorithm 21 is shown in Figure 5.5. In each iteration of the acceleration phase, we perform the (max, min)-product with the bottleneck matrix, B, to retrieve all maximal *df*-pairs up to the path length of $\ell = r$, for some integer constant $1 < r < n$. The maximal *df*-pairs are stored in the distance/flow matrix, P. Then from all *df*-pairs gathered thus far in P, we can initialise t distance matrices, $D^{f,\ell}$, for each maximal flow value f. (Note that both the path length, ℓ, and the maximal flow value, f, can now appear on the superscript.) Then we enter the cruising phase, where we perform repeated squaring of all t distance matrices based on the $(min, +)$-product to effectively solve the APSP problem for all maximal flow values, from which we can retrieve the remaining maximal *df*-pairs.

Lemma 24. *At the end of the acceleration phase of Algorithm 21 we can initialize D^f for each maximal flow value such that the matrix contains the shortest distances possible for all pairs of vertices with path lengths up to r that can push flow of f.*

Proof. In each iteration of the acceleration phase the (max, min)-product computes the maximum bottleneck value possible with the path length up to ℓ. After computing the product, if the bottleneck value for a pair of vertices was increased from the previous iteration, then ℓ and the new bottleneck value b_{ij}^{ℓ} must be a maximal *df*-pair since it is not possible for a shorter path to accommodate a flow value that is greater than or equal to b_{ij}^{ℓ}. Hence at the end of the acceleration phase, for all pairs of vertices, all maximal *df*-pairs have been computed that can be derived with paths of lengths up to r. From the *df*-pairs, it is straightforward to initialise the distance matrices as D^f for each maximal flow value, f, such that each D^f contains the shortest distances possible with paths of lengths up to r that can push flow of f for all pairs of vertices. □

Theorem 14. *Algorithm 21 correctly solves the APSP-AF problem on graphs with unit edge costs in $\tilde{O}(\sqrt{t}n^{(\omega+9)/4}) = \tilde{O}(\sqrt{t}n^{2.843})$ time bound.*

Proof. Following on from Lemma 24, we can observe that at the end of the cruising phase, we have solved the APSP problem in D^f for each maximal flow value f. Thus the end result is the same as the straightforward method of

solving the SP-AF problem given by Algorithm 15. Retrieving all remaining *df*-pairs from all D^f is simply a reverse process of the initialization for the cruising phase.

Using the current fastest algorithm known for the (max, min)-product by Duan and Pettie [19], the time bound for the acceleration phase is $O(rn^{(\omega+3)/2})$. For the cruising phase we have $O(tn^3/r)$ where $O(n/r)$ is the size of the bridging set. We balance the two time complexities by setting $r = \sqrt{t}n^{(3-\omega)/4}$ to achieve $O(\sqrt{t}n^{(\omega+9)/4})$ for the whole algorithm.

Since there are $O(\min\{n, t\})$ *df*-pairs in each p_{ij}, explicitly storing all paths takes $O(\min\{tn^3, n^4\})$, which is clearly too expensive. As explained in Section 3.2, we get around this issue with the help of successor vertices. We can extend the *df*-pair to the (d, f, s) triplet to store the successor vertex, s, for each *df*-pair. In the acceleration phase we can perform witnessed (max, min)-product with an additional polylog factor [19], from which we can derive the successor vertices in $O(n^2)$ time in each iteration [83]. In the cruising phase retrieving the successor vertex can be performed without any additional costs since ordinary matrix multiplication is performed. Then the explicit path can be retrieved by simply following the successor vertices. The additional polylog factor required in the acceleration phase is omitted in the total worst case time complexity of the algorithm to give $\tilde{O}(\sqrt{t}n^{(\omega+9)/4})$. □

Note that the value we choose for r to balance the two time complexities in the proof of Theorem 14 must be less than n, otherwise ℓ iterates all the way up to n in the acceleration phase and the APSP-AF problem is solved before the start of the cruising phase. The time complexity for staying in the acceleration phase until $\ell = n$ is $O(n^{(\omega+5)/2})$. $r = \sqrt{t}n^{(3-\omega)/4} \geq n$ when $t \geq n^{(\omega+1)/2}$. Therefore a more accurate worst case time complexity for Algorithm 21 is actually $\tilde{O}(\min\{n^{(\omega+5)/2}, \sqrt{t}n^{(\omega+9)/4}\}) = \tilde{O}(\min\{n^{3.687}, \sqrt{t}n^{2.843}\})$.

5.3.2 Integer Edge Costs

On graphs with integer edge costs, the APSP-AF problem can be solved in $O(tmn + n^2c)$ or $O(tmn + n^2\log(c/t))$ time bounds simply by running Algorithm 20 n times. We show that by sharing the same priority queue for all *df*-pairs for all pairs of vertices, we can actually solve the APSP-AF

problem more efficiently in $O(tmn + nc)$ or $O(tmn + n^2 \log(c/(tn)))$ time bounds.

In Section 5.2.2 we basically extended Dijkstra's algorithm to solve the SSSP-AF problem by inserting $O(tn)$ (v, d, f) triplets into the frontier set (priority queue), Q. To solve the APSP-AF problem, we extend the algorithm one step further by inserting $O(tn^2)$ (u, v, d, f) quadruples into Q, where u is the source vertex, v is the destination vertex, and (d, f) is the *df*-pair for the path from u to v. Then by using d as the key in the priority queue, we can solve the APSP-AF problem using just one priority queue. Q is initialised with $(v, v, 0, \infty)$ for all $v \in V$, and the algorithm for solving the APSP-AF problem on directed graphs with integer edge costs bounded by c is given by Algorithm 22.

One can think of this approach as sharing a common resource to solve n instances of the SSSP-AF problem at the same time. We note that this idea of sharing resources was already used by Takaoka to solve the APSP problem in $O(mn + n^2 \log(c/n))$ time [71]. Our contribution is to extend this idea further to the more complex APSP-AF problem.

Lemma 25. *Algorithm 22 correctly solves the APSP-AF problem on directed graphs with integer edge costs and real edge capacities.*

Proof. Similar to the proof of Lemma 23. □

Theorem 15. *The APSP-AF problem can be solved in the worst case asymptotic time complexity of* $O(tmn + nc)$.

Proof. We use the one dimensional bucket system to implement Q in Algorithm 22. There are $O(tn^2)$ quadruples, resulting in $O(tn^2)$ time for the *insert* operation. $O(tmn)$ edge inspections can occur, which is also the time complexity for the *decrease-key* operation. The *delete-min* operation is still bounded by $O(nc)$, since we only have to scan through the one dimensional bucket system once. Thus we have $O(tmn + tn^2 + nc) = O(tmn + nc)$ □

Theorem 16. *The APSP-AF problem can be solved in the worst case asymptotic time complexity of* $O(tmn + tn^2 \log(c/(tn)))$.

Algorithm 22 Solve the APSP-AF problem for integer edge costs.

```
1: Q ← φ
2: for all v ∈ V do
3:     insert (v, v, 0, ∞) into Q
4: while Q ≠ φ do
5:     delete (u, v, d, f) from Q such that d is the minimum
6:     for all w ∈ OUT(v) such that w ≠ u do
7:         f' ← MIN(f, cap(v, w))
8:         d' ← d + cost(v, w)
9:         if (u, w, d*, f') in Q for u, w, f' and some d* then
10:            if d' < d* then
11:                update (u, w, d*, f') to (u, w, d', f') in Q
12:        else
13:            insert (u, w, d', f') into Q
14:    if p_uv = φ then
15:        append (d, f) to p_uv
16:    else
17:        let (d_0, f_0) be the last pair in p_uv
18:        if f_0 < f then
19:            if d_0 = d then
20:                delete (d_0, f_0) from p_uv
21:            append (d, f) to p_uv
```

Proof. We use the k-level CBS to implement Q in Algorithm 22 such that the *delete-min* operation takes $O(kmn^2 + pmn^2 + cn/p^{k-1})$ time. We choose $p = (c/(tn))^{1/k}$ and $k = \log(c/(tn))$ such that all operations involving Q is bounded by $O(tn^2 \log(c/(tn)))$. Adding $O(tmn)$ for the total number of edge inspections gives us $O(tmn + tn^2 \log(c/(tn)))$. □

We now present the final, and the most complex algorithm of this thesis, which is an algebraic algorithm for solving the APSP-AF problem on directed graphs with integer edge costs.

In Algorithm 23, the original graph of integer edge costs, G, is expanded to a graph with unit edge costs, G', and B' is the corresponding cn-by-cn bottleneck matrix based on G'. Note that the edge capacities are retained in G'. The *df*-pair is extended to (h, d, f) triplet, where h is the path length in G, and d and f continues to correspond to the *df*-pair. We let $P' = \{p'_{ij}\}$ be a cn-by-cn matrix such that p'_{ij} holds the set of (h, d, f) triplets. Simply put, P' is the matrix of *df*-pairs for G', with the path length information added to each *df*-pair. Similarly, we extend the definition of P to hold (h, d, f) triplets for G. Finally we let $D^f = \{d^f_{ij}\}$ be the distance matrix for the maximal flow value of f, and let $H^f = \{h^f_{ij}\}$ be the matrix of path lengths such that h^f_{ij} is the length of the path that has the distance of d^f_{ij}. As noted in Section 5.3.1, both the path length, ℓ, and the maximal flow, f, can now appear on the superscripts.

Theorem 17. *Algorithm 23 correctly solves the APSP-AF problem with the worst case time complexity of* $\tilde{O}(\sqrt{t}c^{(5+\omega)/4}n^{(9+\omega)/4}) = \tilde{O}(\sqrt{t}c^{1.843}n^{2.843})$.

Proof. Correctness follows from Theorems 4 and 14. The time complexity of the acceleration phase is $\tilde{O}(r(cn)^{(3+\omega)/2})$. As mentioned in the proof of Theorem 4, $|S_i| = O(cn/r)$ in the worst case. Therefore for the cruising phase we have the time complexity of $O(tcn^3/r)$. For both the initialization for the cruising phase and finalization, we have $O(tn^2)$, which is absorbed in the time complexity of the cruising phase since we can assume that $r < cn$. We choose $r = \sqrt{t}c^{(-1-\omega)/4}n^{(3-\omega)/4}$, which gives us the total worst case time complexity of $\tilde{O}(\sqrt{t}c^{(\omega+5)/4}n^{(\omega+9)/4})$. □

Depending on the value of t and c, the value for r may turn out to be greater than cn, which means the APSP-AF problem is solved entirely in the

Algorithm 23 Solve the APSP-AF problem for integer edge costs.

/* Initialization for the acceleration phase */

1: expand G to G', initialize B' based on G'

2: **for** $i \leftarrow 1$ to cn; $j \leftarrow 1$ to cn **do**

3: $\quad p'_{ij} \leftarrow \phi$

/* Acceleration phase */

4: **for** $\ell \leftarrow 2$ to r **do**

5: $\quad B'^{(\ell)} \leftarrow B'^{(\ell-1)} * B'$ /* witnesses given as $W = \{w_{ij}\}$ */

6: $\quad$ **for** $i \leftarrow 1$ to cn; $j \leftarrow 1$ to cn; $i \neq j$ **do**

7: $\quad\quad$ **if** $b'^{(\ell)}_{ij} > b'^{(\ell-1)}_{ij}$ **then**

8: $\quad\quad\quad k = w_{ij}$

9: $\quad\quad\quad (h, d, f) \leftarrow$ last triplet in p'_{ik} /* if empty, $h = 0$ */

10: $\quad\quad\quad$ **if** $j \in G$ **then** append $(h+1, \ell, b'^{(\ell)}_{ij})$ to p'_{ij}

11: $\quad\quad\quad$ **else** append $(h, \ell, b'^{(\ell)}_{ij})$ to p'_{ij}

/* Initialization for the cruising phase, $\ell = r$ */

12: $P \leftarrow$ rows/columns in P' for real vertices /* G' is contracted to G */

13: $D^f, H^f \leftarrow I$ for all maximal flows f /* I is the identity matrix */

14: **for** $i \leftarrow 1$ to n; $j \leftarrow 1$ to n; $i \neq j$ **do**

15: $\quad$ let $p_{ij} = \{(h_1, d_1, f_1), ..., (h_s, d_s, f_s)\}$ for some s /* skip empty p_{ij} */

16: $\quad k \leftarrow 1$ /* k iterates from 1 to s */

17: $\quad$ **for all** maximal flows f in increasing order **do**

18: $\quad\quad$ **if** $f > f_k$ **then** $k \leftarrow k + 1$ /* the next d_k value is needed */

19: $\quad\quad$ **if** $k > s$ **then** break /* we proceed to the next p_{ij} */

20: $\quad\quad d^{f,\ell}_{ij} \leftarrow d_k$; $h^{f,\ell}_{ij} \leftarrow h_k$

/* Cruising phase */

21: **for all** maximal flow values f **do**

22: $\quad$ perform the cruising phase of Algorithm 7 on $D^{f,\ell}$

/* Finalization */

23: let D^f be $D^{f,n}$ from the result of the cruising phase

24: **for** $i \leftarrow 1$ to n; $j \leftarrow 1$ to n; $i \neq j$ **do**

25: $\quad$ **for all** maximal flows f in increasing order **do**

26: $\quad\quad d \leftarrow d^f_{ij}$

27: $\quad\quad$ let the last pair of p_{ij} be $x = (d', f')$ /* if p_{ij} is empty, $x = \phi$ */

28: $\quad\quad$ **if** $x = \phi$ or $(f > f'$ and $d < \infty)$ **then**

29: $\quad\quad\quad$ **if** $d = d'$ **then** replace x with (d, f)

30: $\quad\quad\quad$ **else** append (d, f) to p_{ij}

acceleration phase. Thus a more accurate time complexity for Algorithm 23 is actually $\tilde{O}(\min\{(cn)^{(5+\omega)/2}, \sqrt{t}c^{(\omega+5)/4}n^{(\omega+9)/4}\})$.

We conclude the final chapter of our thesis with Theorem 17 giving us the highlight of our contribution to the field of graph theory. We make a final note regarding the practicality of Algorithms 21 and 23. By reverting to the ordinary matrix multiplication method to compute the (max, min)-product in the acceleration phases of the algorithms, both algorithms become implementable with time complexities of $O(\sqrt{t}n^3)$ and $O(\sqrt{t}c^2n^3)$, respectively. On graphs with relatively larger values of t and smaller values of c, these asymptotic time bounds are still more efficient compared to the straightforward methods.

Chapter VI

Conclusion

We have performed an in-depth study of the problem called the SP-AF problem that has many practical applications where the given problems can be modelled on graphs with edges with both costs and capacities. In our research for finding efficient algorithms to solve the SP-AF problem, we were also able to contribute to the individual topics of matrix multiplication, the SP problem and the BP problem, all of which can be considered to be subtopics of the SP-AF problem. The following list is a summary and the citation for each of our contributions:

- A faster parallel algorithm for matrix multiplication on a 2D mesh array (Algorithm 5): We have halved the number of communication steps from the well known Cannon's algorithm, from $3n$ to $1.5n$ communication steps, achieved by a better overall utilisation of the mesh array. Our mesh array definition remains strictly 2D and the algorithm remains simple [6].

- An enhancement to the breakthrough algorithm by AGM for solving the APSP problem on graphs with integer edge costs (Algorithm 8): From an in-depth analysis of the breakthrough algorithm by AGM that has the worst case time complexity remaining sub-cubic for integer edge costs bounded by $c < n^{0.117}$, we have improved the algorithm such that the worst case time complexity of the algorithm remains sub-cubic for integer edge costs bounded by $c < n^{0.186}$ [61].

- A faster parallel algorithm for solving the APSP problem (Algorithm 13): By translating the serial cascade algorithm onto the 2D mesh array and optimizing the resulting parallel algorithm, we have achieved $3n$ communication steps to solve the APSP problem, which improves upon

the $3.5n$ communication steps that was achieved more than two decades ago. (This contribution has been submitted to the journal of Parallel and Distributed Computing and is currently under review.)

- An efficient algorithm for solving the GB problem (Algorithm 14): By combining Tarjan's algorithm for determining strongly connected components with a simple binary search, we have significantly reduced the worst case time complexity of solving the GB problem from $\tilde{O}(n^{2.687})$ or $O(mn + n^2 \log n)$ down to $O(m \log n)$ [63].

- A formal mathematical definition of the SP-AF problem (Theorem 10): We have defined a new instance of the semi-ring algebraic structure called the distance/flow semi-ring, which not only serves as a formal definition for the new SP-AF problem, but also shows that existing algorithms that are generally applicable to semi-rings can be applied to the new SP-AF problem [60].

- Efficient non-algebraic algorithms for both the SSSP-AF and APSP-AF problems (Algorithms 18, 20 and 22): We have shown that Dijkstra's algorithm can be extended to solve both the SSSP-AF and the APSP-AF problems with efficient time bounds by utilising an advanced data structure called the CBS, as well as utilising the key concept of resource sharing [62].

- Efficient algebraic algorithms for the APSP-AF problem that utilise FMMOR (Algorithms 21 and 23): Using our enhancement to the AGM algorithm as the basis, we have derived an algorithm to solve the APSP-AF problem on graphs with integer edge costs in $\tilde{O}(\sqrt{t}c^{1.844}n^{2.844})$ worst case time bound that utilises both the (max, min)-product of the bottleneck matrix semi-ring and $(min, +)$-product of the distance matrix semi-ring [63, 61].

We conclude our thesis with some open questions that would make good candidates for future research:

- Are there more efficient algorithms for solving the SP-AF problems on undirected graphs? For example, the APSP problem on undirected unweighted graphs can be solved in $\tilde{O}(n^{2.373})$ time bound [58], whereas for directed unweighted graphs, the best time bound remains at $\tilde{O}(n^{2.530})$ [43, 83]. Likewise, for the SP-AF problems, are there faster algorithms for undirected graphs?

- Are there efficient algorithms for solving the SP-AF problems on graphs with real edge costs? In this thesis we have provided algorithms that are faster than the straightforward methods of solving the SP-AF problems on graphs with unit edge costs and integer edge costs. The problem seems to be much harder for real edge costs, as evidenced by the fact that the best known algorithm for solving the APSP problem on dense graphs with real edge costs is only able to achieve a speed up of a polylog factor [33].

- What is the lower bound for the SP-AF problems? The thesis has focused on reducing the worse case upper bounds for the problems. Can we provide sharper lower bounds than the trivial lower bounds of $O(tn)$ and $O(tn^2)$ for the SSSP-AF and APSP-AF problems, respectively?

Appendix A

Publications

A.1 Conferences

- The 1st International Conference on Resource Efficiency in Interorganizational Networks [60]
  - Title: Efficient Graph Algorithms for Network Analysis
  - Authors: Tong-Wook Shinn and Tadao Takaoka
  - Date: 13th – 14th of November, 2013
  - Location: Georg-August-Universitaet, Goettingen, Germany
  - Editors: Jutta Geldermann and Matthias Schumann
  - ISBN: 978-3-86395-142-9
  - Pages: 236 – 247
- The 37th Australasian Computer Science Conference [62]
  - Title: Combining the Shortest Paths and the Bottleneck Paths Problems
  - Authors: Tong-Wook Shinn and Tadao Takaoka
  - Date: 20th – 23rd of January, 2014
  - Location: Auckland University of Technology, Auckland, New Zealand
  - Editors: Bruce Thomas and Dave Parry
  - ISBN: 978-1-921770-30-2
  - Pages: 13 – 18

- The 8th International Workshop on Algorithms and Computation [63]
  - Title: Some Extensions of the Bottleneck Paths Problem
  - Authors: Tong-Wook Shinn and Tadao Takaoka
  - Date: 13th – 15th of February, 2014
  - Location: Indian Institute of Technology, Chennai, India
  - Editors: Sudebkumar Prasant Pal and Kunihiko Sadakane
  - ISBN: 978-3-319-04656-3
  - Pages: 176 – 187
- The 12th Latin American Theoretical INformatics Symposium [61]
  - Title: Combining All Pairs Shortest Paths and All Pairs Bottleneck Paths Problems
  - Authors: Tong-Wook Shinn and Tadao Takaoka
  - Date: 31st of March – 4th of April, 2014
  - Location: Montevideo, Uruguay
  - Editors: Alberto Pardo and Alfredo Viola
  - ISBN: 978-3-642-54423-1
  - Pages: 226 – 237
- The 14th International Conference on Computational Science [6]
  - Title: A Faster Parallel Algorithm for Matrix Multiplication on a Mesh Array
  - Authors: Sung Eun Bae, Tong-Wook Shinn and Tadao Takaoka
  - Date: 10th – 12th of June, 2014
  - Location: Cairns, Australia
  - Editors: David Abramson, Michael Lees, Valeria Krzhizhanovskaya, Jack Dongarra and Peter M.A. Sloot
  - ISSN: 1877-0509
  - Pages: 2230 – 2240

A.2 Journals

- Journal of Theoretical Computer Science
  - Title: Variations on the Bottleneck Paths Problem
  - Authors: Tong-Wook Shinn and Tadao Takaoka
  - Status: accepted, waiting to be published
- Journal of Parallel and Distributed Computing
  - Title: A Faster Parallel Algorithm for Solving the APSP Problem on a Mesh Array
  - Authors: Sung Eun Bae, Tong-Wook Shinn, and Tadao Takaoka
  - Status: under review
- SIAM Journal on Computing
  - Title: Shortest Paths for All Flows on Graphs with Integer Edge Costs
  - Authors: Tong-Wook Shinn and Tadao Takaoka
  - Status: under review

References

[1] D. Abuaiadh and J. H. Kingston. Are Fibonacci heaps optimal? In *ISAAC*, volume 834, pages 442–450. Springer, 1994.

[2] A. V. Aho, J. E. Hopcroft, and J. D. Ullman. *The Design and Analysis of Computer Algorithms*. Addison-Wesley, 1974.

[3] R. K. Ahuja, K. Mehlhorn, J. Orlin, and R. E. Tarjan. Faster algorithms for the shortest path problem. *Journal of ACM*, 37(2):213–223, 1990.

[4] N. Alon, Z. Galil, and O. Margalit. On the exponent of the all pairs shortest path problem. In *FOCS*, pages 569–575. IEEE Computer Society, 1991.

[5] Y. P. Aneja, V. Aggarwal, and K. P. K. Nair. On a class of quadratic programs. *European Journal of Operational Research*, 18(1):62–70, 1984.

[6] S. E. Bae, T. Shinn, and T. Takaoka. A faster parallel algorithm for matrix multiplication on a mesh array. In *ICCS*, pages 2230–2240, 2014.

[7] A. Benaini and Y. Robert. An even faster systolic array for matrix multiplication. *Parallel computing*, 12(2):249–254, 1989.

[8] D. Bini, M. Capovani, F. Romani, and G. Lotti. $O(n^{2.7799})$ complexity for $n * n$ approximate matrix multiplication. *Information Processing Letters*, 8(5):234–235, 1979.

[9] G. S. Brodal, G. Lagogiannis, and R. E. Tarjan. Strict fibonacci heaps. In *STOC*, pages 1177–1184. ACM, 2012.

[10] L. E. Cannon. *A cellular computer to implement the Kalman filter algorithm*. PhD thesis, Montana State University, Bozeman, MT, USA, 1969.

[11] T. M. Chan. All-pairs shortest paths with real weights in $O(n^3/\log n)$ time. *Algorithmica*, 50(2):236–243, 2008.

[12] K. H. Cheng and S. Sahni. VLSI systems for band matrix multiplication. *Parallel computing*, 4(3):239–258, 1987.

[13] D. Coppersmith and S. Winograd. On the asymptotic complexity of matrix multiplication. *SIAM Journal on Computing*, 11(3):472–492, 1982.

[14] D. Coppersmith and S. Winograd. Matrix multiplication via arithmetic progressions. *Journal of Symbolic Computation*, 9(3):251–280, 1990.

[15] E. V. Denardo and B. L. Fox. Shortest-route methods: 1. reaching, pruning, and buckets. *Operations Research*, 27(1):161–186, 1979.

[16] E. W. Dijkstra. A note on two problems in connexion with graphs. *Numerische Mathematik*, 1(1):269–271, 1959.

[17] Y. A. Dinitz. Algorithm for solution of a problem of maximum flow in a network with power estimation. *Doklady Akademii Nauk SSSR*, 194(4):754, 1970.

[18] W. Dobosiewicz. A more efficient algorithm for the min-plus multiplication. *International Journal of Computer Mathematics*, 32(1-2):49–60, 1990.

[19] R. Duan and S. Pettie. Fast algorithms for (max, min)-matrix multiplication and bottleneck shortest paths. In *SODA*, pages 384–391. SIAM, 2009.

[20] J. Edmonds and R. M. Karp. Theoretical improvements in algorithmic efficiency for network flow problems. *Journal of the ACM*, 19(2):248–264, 1972.

[21] B. A. Farby, A. H. Land, and J. D. Murchland. The cascade algorithm for finding all shortest distances in a directed graph. *Management Science*, 14(1):19–28, 1967.

[22] R. W. Floyd. Algorithm 97: Shortest path. *Communications of the ACM*, 5(6):345, 1962.

[23] L. R. Ford and D. R. Fulkerson. Maximal flow through a network. *Canadian journal of Mathematics*, 8(3):399–404, 1956.

[24] M. L. Fredman. New bounds on the complexity of the shortest path problem. *SIAM Journal on Computing*, 5(1):83–89, 1976.

[25] M. L. Fredman and R. E. Tarjan. Fibonacci heaps and their uses in improved network optimization algorithms. In *FOCS*, pages 338–346. IEEE Computer Society, 1984.

[26] Z. Galil and O. Margalit. Witnesses for Boolean matrix multiplication and for transitive closure. *Journal of Complexity*, 9(2):201–221, 1993.

[27] A. V. Goldberg and R. E. Tarjan. Finding minimum-cost circulations by canceling negative cycles. *Journal of the ACM*, 36(4):873–886, 1989.

[28] A. V. Goldberg and R. E. Tarjan. Finding minimum-cost circulations by successive approximation. *Mathematics of Operations Research*, 15(3):430–466, 1990.

[29] M. Gondran and M. Minoux. *Graphs, dioids and semirings: new models and algorithms*, volume 41. Springer, 2008.

[30] L. J. Guibas, H. T. Kung, and C. D. Thompson. Direct VLSI implementation of combinatorial algorithms. *Caltech Conference on VLSI*, pages 509–525, 1979.

[31] Y. Han. Improved algorithm for all pairs shortest paths. *Information Processing Letters*, 91(5):245–250, 2004.

[32] Y. Han. An $O(n^3(\log\log n/\log n)^{5/4})$ time algorithm for all pairs shortest path. *Algorithmica*, 51(4):428–434, 2008.

[33] Y. Han and T. Takaoka. An $O(n^3 \log\log n/\log^2 n)$ time algorithm for all pairs shortest paths. In *SWAT*, volume 7357, pages 131–141. Springer Berlin Heidelberg, 2012.

[34] P. Hansen. Bicriterion path problems. In *Multiple criteria decision making theory and application*, pages 109–127. Springer, 1980.

[35] T. C. Hu. Letter to the editor-the maximum capacity route problem. *Operations Research*, 9(6):898–900, 1961.

[36] T. C. Hu. Revised matrix algorithms for shortest paths. *SIAM Journal on Applied Mathematics*, 15(1):207–218, 1967.

[37] H. V. Jagadish and T. Kailath. A family of new efficient arrays for matrix multiplication. *IEEE Transactions on Computers*, 38(1):149–155, 1989.

[38] S. C. Kak. A two-layered mesh array for matrix multiplication. *Parallel Computing*, 6(3):383–385, 1988.

[39] M. Klein. A primal method for minimal cost flows with applications to the assignment and transportation problems. *Management Science*, 14(3):205–220, 1967.

[40] J. B. Kruskal. On the shortest spanning subtree of a graph and the travelling salesman problem. *Proceedings of the American Mathematical society*, 7(1):48–50, 1956.

[41] H. T. Kung and C. E. Leiserson. Systolic arrays (for VLSI). *Society for Industrial & Applied*, page 256, 1979.

[42] G. Lakhani and R. Dorairaj. A VLSI implementation of all-pair shortest path problem. In *ICPP*, pages 207–209, 1987.

[43] F. Le Gall. Faster algorithms for rectangular matrix multiplication. In *FOCS*, pages 514–523. IEEE, 2012.

[44] F. Le Gall. Powers of tensors and fast matrix multiplication. In *ISSAC*, pages 296–303, 2014.

[45] V. M. Malhotra, M. P. Kumar, and S. N. Maheshwari. An $O(|V|^3)$ algorithm for finding maximum flows in networks. *Information Processing Letters*, 7(6):277–278, 1978.

[46] R. McNaughton and H. Yamada. Regular expressions and state graphs for automata. *IRE Transactions on Electronic Computers*, EC-9(1):39–47, 1960.

[47] L. Melkemi and M. Tchuente. Complexity of matrix product on a class of orthogonally connected systolic arrays. *IEEE Transactions on Computers*, 100(5):615–619, 1987.

[48] J. C. Namorado Climaco and E. Queiros Vieira Martins. A bicriterion shortest path algorithm. *European Journal of Operational Research*, 11(4):399–404, 1982.

[49] V. Y. Pan. Strassen's algorithm is not optimal trilinear technique of aggregating, uniting and canceling for constructing fast algorithms for matrix operations. In *FOCS*, pages 166–176. IEEE Computer Society, 1978.

[50] B. Pelegrın and P. Fernández. On the sum-max bicriterion path problem. *Computers & operations research*, 25(12):1043–1054, 1998.

[51] M. Pollack. Letter to the editor-the maximum capacity through a network. *Operations Research*, 8(5):733–736, 1960.

[52] F. P. Preparata and J. E. Vuillemin. Area-time optimal VLSI networks for multiplying matrices. *Information Processing Letters*, 11(2):77–80, 1980.

[53] R. C. Prim. Shortest connection networks and some generalizations. *Bell system technical journal*, 36(6):1389–1401, 1957.

[54] E. Queiros Vieira Martins. On a multicriteria shortest path problem. *European Journal of Operational Research*, 16(2):236–245, 1984.

[55] F. Romani. Some properties of disjoint sums of tensors related to matrix multiplication. *SIAM Journal on Computing*, 11(2):263–267, 1982.

[56] S. Saunders and T. Takaoka. Improved shortest path algorithms for nearly acyclic graphs. *Electronic Notes in Theoretical Computer Science*, 42:232–248, 2001.

[57] A. Schönhage. Partial and total matrix multiplication. *SIAM Journal on Computing*, 10(3):434–455, 1981.

[58] R. Seidel. On the all-pairs-shortest-path problem in unweighted undirected graphs. *Journal of computer and system sciences*, 51(3):400–403, 1995.

[59] A. Shapira, R. Yuster, and U. Zwick. All-pairs bottleneck paths in vertex weighted graphs. In *SODA*, pages 978–985. Society for Industrial and Applied Mathematics, 2007.

[60] T. Shinn and T. Takaoka. Efficient graph algorithms for network analysis. In *ResEff*, pages 236–247, 2013.

[61] T. Shinn and T. Takaoka. Combining all pairs shortest paths and all pairs bottleneck paths problems. In *LATIN*, pages 226–237, 2014.

[62] T. Shinn and T. Takaoka. Combining the shortest paths and the bottleneck paths problems. In *ACSC*, pages 13–18, 2014.

[63] T. Shinn and T. Takaoka. Some extensions of the bottleneck paths problem. In *WALCOM*, pages 176–187, 2014.

[64] A. J. Stothers. *On the complexity of matrix multiplication.* PhD thesis, The University of Edinburgh, 2010.

[65] V. Strassen. Gaussian elimination is not optimal. *Numerische Mathematik*, 13(4):354–356, 1969.

[66] V. Strassen. The asymptotic spectrum of tensors and the exponent of matrix multiplication. In *FOCS*, pages 49–54, 1986.

[67] T. Takaoka. A new upper bound on the complexity of the all pairs shortest path problem. *Information Processing Letters*, 43(4):195–199, 1992.

[68] T. Takaoka. Sub-cubic cost algorithms for the all pairs shortest path problem. In *WG*, volume 1017, pages 323–343. Springer, 1995.

[69] T. Takaoka. Shortest path algorithms for nearly acyclic directed graphs. *Theoretical Computer Science*, 203(1):143–150, 1998.

[70] T. Takaoka. An $O(n^3 \log\log n/\log n)$ time algorithm for the all-pairs shortest path problem. *Information Processing Letters*, 96(5):155–161, 2005.

[71] T. Takaoka. Efficient algorithms for the all pairs shortest path problem with limited edge costs. In *CATS*, volume 128, pages 21–26, 2012.

[72] T. Takaoka and K. Umehara. An efficient VLSI algorithm for the all pairs shortest path problem. *Journal of Parallel and Distributed Computing*, 16(3):265–270, 1992.

[73] T. Takaoka and K. Umehara. Cascade algorithm revisited. Technical Report TR-COSC 01/14, University of Canterbury, 2014.

[74] R. E. Tarjan. Depth-first search and linear graph algorithms. *SIAM Journal on Computing*, 1(2):146–160, 1972.

[75] M. Thorup. Integer priority queues with decrease key in constant time and the single source shortest paths problem. In *STOC*, pages 149–158. ACM, 2003.

[76] J. D. Ullman. *Computational aspects of VLSI*, volume 11. Computer Science Press Rockville, MD, 1984.

[77] K. Umehara. The Design and Analysis of Sequential and Parallel Algorithms for Shortest Path Problem. Master's thesis, Department of Computer and Information Science, Ibaraki University, Ibaraki, Japan, 1990.

[78] V. Vassilevska. *Efficient Algorithms for Path Problems in Weighted Graphs*. PhD thesis, School of Computer Science, Carneige Mellon University, Pittsburgh, PA 15213, 2008.

[79] V. Vassilevska, R. Williams, and R. Yuster. All pairs bottleneck paths and max-min matrix products in truly subcubic time. *Theory Of Computing*, 5(1):173–189, 2009.

[80] V. V. Williams. Multiplying matrices faster than coppersmith-winograd. In *STOC*, pages 887–898. ACM, 2012.

[81] S. Winograd. A new algorithm for inner product. *IEEE Transactions on Computers*, 100(7):693–694, 1968.

[82] M. Yoeli. A note on a generalization of Boolean matrix theory. *American Mathematical Monthly*, pages 552–557, 1961.

[83] U. Zwick. All pairs shortest paths using bridging sets and rectangular matrix multiplication. *Journal of the ACM*, 49(3):289–317, 2002.

[84] U. Zwick. A slightly improved sub-cubic algorithm for the all pairs shortest paths problem with real edge lengths. *Algorithmica*, 46(2):181–192, 2006.