Algorithms for Average Path Length Minimisation Through Edge Addition

Andrew Gozzard
University of Western Australia
email: 20948678@student.uwa.edu.au

This report is submitted as partial fulfilment of the requirements for the Honours Programme of the School of Computer Science and Software Engineering, The University of Western Australia, 2016

Abstract

The Average Path Length (APL) in a graph is a useful metric for assessing the connectivity of the graph. This report presents two efficient algorithms for minimising the APL in a weighted, directed graph $G = (V, E)$ by adding a single edge from the set of edges S. Time complexity analysis of these two algorithms gives theoretical run time complexities of $O(|V|^2|S|)$ and $O(|V|^3 \log|V| + |V||S| \log|V|)$, both of which are a significant improvement over the previous best known algorithm. Empirical analysis of the performance and correctness of these new algorithms validated the theoretical predictions. These algorithms have potential applications in a number of real-world systems, including microprocessor and network-on-chip design.

Keywords: Graph Theory, Average Path Length, Shortest Path
CR Categories: G.2.2

Acknowledgements

The author would like to thank Professor Amitava Datta and Max Ward for their invaluable advice and for providing the original inspiration for this work, and Tim Davies for his feedback that helped to improve the presentation of this report.

Contents

CHAPTER 1

Introduction

Many real-world systems, such as communication networks, can be modelled as a graph composed of a set of vertices and a set of edges. Using a computer network as an example, the set of vertices represents the devices in the network, and the edges represent the network connections between them.

The Average Path Length (APL) in a graph is useful as a metric for the connectivity and performance of any real-world systems that can be modelled as graphs, and minimising the APL is valuable as a method for optimising these real-world systems. Again using a computer network as an example, a network with low APL would also have low average transmission latency between computers. Minimising the APL can be valuable in optimising the performance of networks such as the inter-core communication networks found in modern multi-core microprocessors [1], [2] (see Section 3.2). Small-world networks are a type of graph characterised in part by their low APL [3]. The properties and synthesis of small-world networks have been the subject of some previous research [4], [5] (see Section 3.4). Efficient methods for minimising the APL of a graph can be used to synthesise small-world networks.

Previous work that has been done in this field, as well as on the relationship between graphs with low APL and small-world graphs, is covered in Chapter 3. Investigation of strategies for minimising the APL in a graph through single edge addition produced two efficient algorithms for finding an optimal solution to this problem. The algorithms were tested empirically to assess their correctness and efficiency.

CHAPTER 2

Problem Description

2.1 Average Path Length

Consider a directed graph $G = (V, E)$ in which edge weights are given by the function $w\colon E \to \mathbb{R}$. A path p in G from $s \in V$ to $t \in V$ is a sequence of vertices $p = \langle v_0, v_1, v_2, \ldots, v_k \rangle$ such that $v_0 = s$, $v_k = t$, and $(v_{i-1}, v_i) \in E$ for $i = 1, 2, \ldots, k$ [6, p. 1170]. The path p is said to *contain* the vertices $v_0, v_1, v_2, \ldots, v_k$ and the edges $(v_0, v_1), (v_1, v_2), (v_2, v_3), \ldots, (v_{k-1}, v_k)$. This can be expressed as $v_i \in p$ for $i = 0, 1, 2, \ldots, k$ and $(v_{i-1}, v_i) \in p$ for $i = 1, 2, \ldots, k$. The length of a path is the sum of the weights of all edges it contains. If a path exists between a given pair of vertices, there must therefore exist one or more such paths that have minimum length. Let $d_{s,t}$ be the length of the shortest (i.e., minimum length) path in G that begins at s and ends at t. Self-paths are considered to have zero length (i.e., $d_{s,s} = 0$) as they contain no edges, and pairs of vertices for which no path exists have a constant *disconnection cost* D (that is, $d_{s,t} = D$). In computer science the shortest path is considered undefined for graphs containing cycles (that is, paths that start and end at the same vertex) with negative total weight [6, p. 645]. Without loss of generality there exists a shortest path that is a simple path (that is, a path containing no cycles, and therefore no repeated vertices or edges) [6, p. 646]. The weighted Sum of Path Lengths (SPL) in G is defined as:

$$\mathrm{SPL} = \sum_{i,j \in V} c_{i,j} d_{i,j} \tag{2.1}$$

where $c_{i,j}$ is a constant non-negative weighting factor defined for every pair of vertices $i, j \in V$. A typical definition of $c_{i,j}$ would be $c_{i,i} = 0$ and $c_{i,j} = 1$ for all $i \neq j$. The APL in G is defined as:

$$\mathrm{APL} = \sum_{i,j \in V} c_{i,j} d_{i,j} \Big/ \sum_{i,j \in V} c_{i,j} = \mathrm{SPL}/\mathrm{SWF} \tag{2.2}$$

where $\mathrm{SWF} = \sum_{i,j \in V} c_{i,j}$ is the sum of all weighting factors. Note that the SWF is constant, and hence the SPL is a constant-factor multiple of the APL. Both

the SPL and the APL are undefined on graphs containing negative-weight cycles due to their definition in relation to the shortest path problem.

2.2 MinAPL

For a graph $G = (V, E)$ with weighting factors $c_{i,j}$, and a set of candidate edges S, the single edge addition APL minimisation problem (henceforth MinAPL or alternatively 1-MinAPL) is to select a single edge $e \in S$ such that the APL of $G' = (V, E \cup \{e\})$ is minimised. It is valid to make no selection if no edge in S reduces the APL. Any edge that introduces a negative-weight cycle is considered invalid, as its introduction would cause the APL to become undefined. Improving on the known polynomial-time solutions to this problem is the main focus of this report.

2.3 k-MinAPL

For a graph $G = (V, E)$ with weighting factors $c_{i,j}$, a set of candidate edges S, and an integer k, the multiple edge addition APL minimisation problem (henceforth k-MinAPL) is to select a set of edges $F \subseteq S$, $|F| \leq k$ such that the APL of $G' = (V, E \cup F)$ is minimised. Any set of edges that introduces a negative-weight cycle is considered invalid, as its introduction would cause the APL to become undefined. This problem has been shown to be NP-hard by reduction from the set cover problem [7], and hence is not equivalent to repeated application of 1-MinAPL.

2.4 Example

Consider the example graph shown in Fig. 2.1 with candidate edge set $S = \{e_1, e_2, e_3, e_4\}$ and $c_{i,j} = 1$ for all $i \neq j$ and $c_{i,i} = 0$ otherwise. Initially, the graph is disconnected and has a total APL of $(8D + 81)/16$. Let D have some large yet finite value. For $k = 1$ the optimal solution is to take either e_2 or e_3, as both give a new APL of $(4D + 89)/16$, and both other options give $(4D + 102)/16$. For $k = 2$ the optimal solution is to take both e_2 and e_3, as this gives a new APL of $40/16$, which is better than the result of taking any other subset of edges.

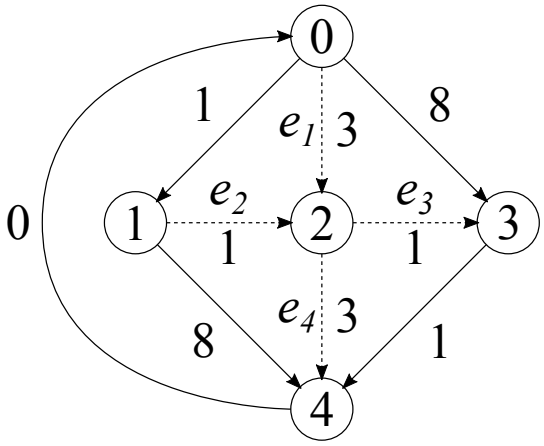

Figure 2.1: Example graph

CHAPTER 3

Literature Review

This section reviews work done in the field of APL minimisation and related problems. Section 3.1 reviews methods for computing shortest paths, Section 3.2 discusses the relationship between APL and network topology, specifically with regard to small-world networks, Section 3.3 considers what work has already been done in finding computational methods for minimising the APL of a graph, and Section 3.4 reviews applications of such methods.

3.1 Shortest Path Problems

Given the definition of APL in Section 2.1, it is readily apparent that computing $d_{i,j}$, the length of the shortest path from node i to node j, is fundamental to computing the APL. This section reviews previous work done on the problem of computing shortest paths in G as defined in Section 2.1, including the Single Source Shortest Paths (SSSP) and the All-Pairs Shortest Paths (APSP). The SSSP problem involves computing the shortest path from a single given source vertex $s \in V$ to each destination vertex $t \in V$. The APSP problem is to compute the shortest path from s to t for all pairs $s, t \in V$. Efficient algorithms for solving APSP are of particular relevance to computing the APL, as the APL is defined in terms of the shortest path lengths for all pairs of start and end vertices.

In 1959, Dijkstra [8] proposed an algorithm for computing the shortest path tree of a graph rooted at a given vertex s. The proposed algorithm is stated to only work for graphs with non-negative edge weights. A shortest path tree can be thought of as the union of the shortest paths from s to t for all $t \in V$. The construction of such a tree is equivalent to computing the SSSP of a graph. The algorithm works by starting with the singleton tree containing just s, and repeatedly adding vertices from V until the tree contains the entirety of V. With each addition, the vertex with minimum shortest path length from s and not already in the shortest path tree is the one chosen. It can be shown that this shortest path must be one edge longer than a shortest path already in the tree,

as any vertex closer to s must already be in the tree. Selecting the next vertex to add can therefore be accomplished by maintaining a record of the length of the shortest known path to each vertex, and updating this for all neighbours of the new vertex whenever a new vertex is added to the tree. The parent vertex is whichever intermediate vertex in the tree is responsible for the shortest known path to the new vertex, and whenever a new vertex is added to the tree the edge between the new vertex and its parent must also be added. In its original form, this algorithm performed a linear-time scan through all shortest known paths and selected the minimum, giving this algorithm time complexity $O(|V|^2)$. The algorithm can be improved using a minimum-priority queue and was improved upon in 1987 by Fredman and Tarjan [9], who used a Fibonacci-heap to implement a more efficient minimum-priority queue and achieved an overall time complexity of $O(|E| + |V| \log|V|)$.

In 1962, Floyd [10] proposed an algorithm for computing the lengths of the APSP of a graph. Floyd's work is very similar to an independent work published by Warshall [11] in the same year, and hence the algorithm is commonly referred to as the Floyd-Warshall algorithm. The proposed algorithm is stated to work for directed graphs with real-valued edge weights, but not for graphs containing negative-weight cycles. The algorithm works by maintaining an array $d_{i,j}$. Initially d is just the edge weights of the graph, $w_{i,j}$ (which are infinite if no edge exists). This can be thought of as the lengths of all shortest paths from i to j containing no intermediate vertices. The algorithm then iteratively grows the shortest paths by adding each $k \in V$ in turn to the set of allowable intermediate vertices and updating $d_{i,j}$ accordingly. For all pairs $i, j \in V$, if $d_{i,k} + d_{k,j} < d_{i,j}$, then $d_{i,j}$ is overwritten with the shorter length. As such $d_{i,j}$ always contains the length of the shortest path possible using only the current set of allowable intermediate vertices. When all vertices have been added, the set of allowable intermediate vertices is the entirety of the set V, and hence the values of $d_{i,j}$ are the lengths of the shortest paths in G for all $i, j \in V$. Time complexity analysis of the algorithm produces a theoretical time complexity of $O(|V|^3)$. For dense graphs (i.e.: where $|E| \in \Theta(|V|^2)$), this is the asymptotically fastest APSP algorithm in current literature.

More efficient APSP algorithms are known to exist for sparse graphs (where $|E| \in O(|V|)$). Repeated application of Dijkstra's SSSP algorithm for all source vertices can be used to compute the APSP in $O(|V||E| + |V|^2 \log|V|)$, or equivalently $O(|V|^2 + |V|^2 \log|V|) \sim O(|V|^2 \log|V|)$ due to the asymptotic limit on $|E|$. This is a clear improvement on the time complexity of the Floyd-Warshall algorithm, but its applications are constrained by the requirement for non-negative edge weights. However, in 1977 Johnson [12] published a method (based on the Bellman-Ford algorithm) for converting general graphs to graphs with non-

negative edge weights in $O(|V||E|)$ while retaining all shortest path information. Johnson combined this method with repeated application of Dijkstra's algorithm to produce an APSP algorithm with overall time complexity $O(|V||E| + |V|^2 \log|V|)$. In a more recent work from 2004, Pettie [13] succeeded in generalising the work of Thorup [14] and Hagerup [15] to produce an APSP algorithm for general graphs with overall time complexity $O(|V||E| + |V|^2 \log\log|V|)$. Both Johnson's and Pettie's algorithms are asymptotically as fast as the Floyd-Warshall algorithm on dense graphs, but on sparse graphs their overall time complexities become $O(|V|^2 \log|V|)$ and $O(|V|^2 \log\log|V|)$ respectively, both of which are a significant improvement over the Floyd-Warshall algorithm.

3.2 Small Worlds

The small-world phenomenon was observed and defined by Milgram in 1967 [3]. Many real-world networks have regular structure, meaning that all nodes in the network have an approximately equal number of incident links. The APL in a graph with regular structure is typically quite long. A small-world network is defined as having an APL that grows logarithmically with respect to the size of the network. This has lead to some research being done in converting arbitrary networks into small-world networks, which closely relates to the APL minimisation problem considered in this report.

In their 1998 work, Watts and Strogatz [4] investigated the properties of many real-world examples of small-world networks, such as the graph of actors who have been credited in the same films, and presented a method for the construction of a small-world network. They found that networks can be classified by their APL and clustering coefficient (a measure of how often two neighbouring nodes share a common neighbour). Using these measures they compared a number of naturally occurring small-world networks to networks with extremely regular structure as well as to networks with randomised edges. They found that small-world networks tend to have a very short APL relative to a structured network, but slightly longer than a randomised network, whereas the clustering coefficient for small-world networks tended to be nearly as large as that of regular networks, and far greater than that of a randomised network. Watts and Strogatz then proposed a method for constructing small-world networks by first forming a network with extremely regular structure and replacing randomly selected edges with a new edge to a randomly selected vertex. By varying the probability with which an edge is replaced, they found that replacing even a small number of random edges caused the APL to drop rapidly. A far greater number of edges had to be replaced in order to reduce the clustering coefficient significantly. This demonstrates that

an otherwise highly regular large-world network can be converted to a small-world network through the replacement of randomly selected edges with randomly generated edges.

In 2001, Comellas and Sampels [5] built on the aforementioned work of Watts and Strogatz by proposing two methods for the deterministic construction of small-world networks from an arbitrary initial network. The first method works by replacing each vertex in the initial graph with a wholly connected network of k nodes where k is the degree of the original node. Each edge attached to the original node is then attached to one of the newly created nodes such that every node in the subgraph has degree k. In this fashion, the size and clustering coefficient of the network are increased considerably, and the APL is significantly reduced. Comellas and Sampels go on to show that the resulting network has all the properties of a small-world network. The second method involves creating wholly connected subgraphs of arbitrary size for each node in the original network, and connecting that node to each newly created vertex so as to make it a part of the wholly connected subgraph. This method again significantly increases the clustering coefficient and size of the network while also reducing the APL, and can again be shown to produce small-world networks. This method is also much more flexible than the first method, and hence is likely more suited to real-world analysis and design applications. Both of these methods demonstrate that it is possible to deterministically convert arbitrary graphs to small-world graphs, thereby significantly reducing the APL. These methods do not, however, allow for the conversion of arbitrary networks to small-world networks of the same size.

3.3 APL Minimisation Through Edge Addition

While the above methods for conversion to small-world networks are highly effective, they have some shortcomings when it comes to solving MinAPL and k-MinAPL. Even though the small-world networks produced have lower APL than the original network from which they are derived, they are by no means the optimal solution. In addition to this, the method proposed by Watts and Strogatz is randomised [4], and both of Comellas' and Sampels' methods increase the size of the network [5]. All of these methods are therefore not suitable as solutions to MinAPL or k-MinAPL.

In their 2009 work, Meyerson and Tagiku [7] consider both the general problem of k-MinAPL, as well as several more constrained variants of the problem. They show that k-MinAPL and the variants they consider are all NP-hard for unbounded k by reduction from the set cover problem. This reduction constructs a directed acyclic graph containing a common root vertex s, a vertex v_S for each

subset $S \in C$, and a vertex v_x for each element $x \in U$, with edges of unit weight from the s to each v_S, and from each v_S to v_x for all $x \in S$. The sum of all shortest path lengths from s is therefore $|C|+2|U|$. The solution to this instance of the set cover problem can then be found by constructing a set of zero-weight candidate edges from the root to each subset, and iterating over all $k \leq |C|$ to find the minimum value of k for which k-MinAPL has a solution that reduces this shortest path length sum by at least $k + |U|$, as this would indicate that the shortest paths from the root to each of those k subsets has been shortened by 1, as have the shortest paths from the root to each element, and hence that these k subsets cover all the elements in U. This implies that the existence of a polynomial time solution to the single-source variant of k-MinAPL would therefore imply the existence of a polynomial time solution to the set cover problem, and hence k-MinAPL must itself be NP-hard. They go on to show that this implies a number of other variants of k-MinAPL, including the variant defined in Section 2.3, are also NP-hard. Meyerson and Tagiku propose a number of polynomial-time approximation algorithms for k-MinAPL and its variants, but do not propose any methods for computing exact solutions to MinAPL or k-MinAPL.

Gaur *et al.* consider the MinAPL problem in their 2014 work [16]. However, they constrain the problem by insisting the graph be unweighted and undirected. This allows them to compute the APSP and hence the APL in $O(|V|^2 \log|V|)$, presumably using Reddy and Iyer's [17] APSP algorithm for unweighted, undirected graphs, though they do not specify the APSP algorithm used. For each new edge being considered, Gaur *et al.* compute the APL and retain the single edge that most reduces the APL compared to the original graph. As there are $|S|$ new edges to consider, this algorithm has time complexity $O(|S||V|^2 \log|V|)$. The algorithm has a number of real-world applications, but its inability to handle weighted or directed graphs reduces its versatility and usefulness in many real-world problems.

3.4 Applications

This section covers two potential applications of the APL problem, but considering the huge variety of theoretical and real-world systems that can be modelled as graphs, it is likely there many other potential applications.

Chang *et al.* have produced two papers on their proposal to use radio frequency (RF) interconnects in multi-core processor design [1], [2]. The motivation for this proposal comes from the growing structural complexity of modern multi-core silicon processors. The complexity of the inter-core communication architecture increases in order to keep up with bandwidth requirements as the

number and complexity of the processor cores increases. These network-on-chip (NoC) architectures pose a challenging optimisation problem, as communication latency in NoC has a huge impact on the performance of the processor. Chang *et al.* propose the use of RF interconnects to allow communications signals to be transmitted at the speed of light to provide a shortcut between two locations on the chip to improve overall performance. However, the transceivers for these interconnects can be quite costly to manufacture and take up a lot of space. These cost and space constraints and the limits on usable RF bandwidth mean that only a very small number or perhaps only one RF interconnect can be added to a processor's network architecture. The ability to compute which edge or edges provide the greatest reduction in average latency would be valuable when trying to optimise NoC performance through the addition of RF interconnects.

Ogras and Marculescu [18] discuss a similar application in NoC design for Very Large Scale Integration (VLSI) design. They investigated the problem of congestion in highly regular mesh networks with the goal of improving the overall capacity of the graph by increasing the required volume of traffic to cause congestion. By adding a very few long-range links to the network they were able to improve the capacity of the network significantly and prevent congestion even at far higher network utilisation. Ogras and Marculescu present an algorithm for selecting long-range links to add to the mesh network. This algorithm works by iteratively selecting the single most beneficial link to add and adding it to the network. Which link is most beneficial is determined by evaluating the network with each link added to produce a congestion measure called the critical load, λ_c, and selecting the link that maximises this value. Ogras and Marculescu show that λ_c is inversely proportional to the APL in the network, and hence select the edge that minimises the APL. This algorithm is therefore equivalent to the repeated application of MinAPL, which means the development of efficient methods for solving the MinAPL problem has valuable applications in optimising mesh networks through long-range link addition.

The dynamic APSP problem and related problems have been the focus of some investigation [19]. The research aims to find ways of maintaining shortest path information about a graph such that the graph itself can be updatedand the shortest paths in the graph queried more efficiently than simply recomputing the shortest path for each query. The dynamic APSP problem is closely related to MinAPL, as any efficient dynamic APSP algorithm has the potential to be adapted into an efficient solution to MinAPL. The reverse is likely to also be true, as any methods that efficiently solve MinAPL must be computing the change in APSP lengths. As such the dynamic APSP problem is a promising avenue for further investigation related to MinAPL, and has the potential to be a valuable application of research relating to MinAPL.

CHAPTER 4

Efficient Algorithms for MinAPL

This section proposes two new and highly efficient algorithmic solutions to the MinAPL problem, and compares their theoretical performance with the brute force algorithm and with each other.

4.1 Solving MinAPL by Brute Force

Consider the brute force solution to MinAPL. To find the single edge $e \in S$ that minimises the APL of $G' = (V, E \cup \{e\})$, simply consider each possible e, construct G', and compute the APL. By keeping track of which option for e produced the minimum APL, the optimal solution can be found exhaustively. Algorithm 1 shows a method for computing the APL of a graph given the graph and its weighting factors. This method is used by Algorithm 2 to find an optimal solution to MinAPL by brute force.

The time complexity of Algorithm 2 is a factor of $O(|S|)$ greater than Algorithm 1, which itself has time complexity $O(apsp(|V|, |E|)+|V|^2)$ where $apsp(|V|, |E|)$ is the run time of the APSP algorithm used. The time complexity of Algorithm 1 is dominated by the APSP algorithm used. Using Pettie's $O(|V||E| + |V|^2 \log\log|V|)$ APSP algorithm [13] in Algorithm 1 gives an overall time complexity for Algorithm 2 of $O(|S||V||E| + |S||V|^2 \log\log|V|)$. Using the Floyd-Warhsall algorithm instead gives $O(|S||V|^3)$. For dense graphs (where $|E| \in \Omega(|V|^2)$[1]) these complexities are equivalent. Algorithm 2 requires $O(|V|^2)$ memory in all cases.

[1]Big Omega notation: $f(n) \in \Omega(g(n)) \iff g(n) \in O(f(n))$

Algorithm 1 Algorithm for computing the c-weighted APL of a graph G

```
1: procedure APL(c, G)
2:     (V, E) ← G
3:     d ← APSP(G)                          ▷ Compute the All-Pairs Shortest Paths
4:     SPL ← 0
5:     SWF ← 0
6:     for i ∈ V do
7:         for j ∈ V do
8:             SPL ← SPL + c_{i,j} d_{i,j}      ▷ Undefined if d_{i,j} is undefined
9:             SWF ← SWF + c_{i,j}
10:        end for
11:    end for
12:    APL ← SPL/SWF
13:    return APL
14: end procedure
```

Algorithm 2 Brute force algorithm for solving MinAPL

```
1: procedure MinAPL(c, G, S)
2:     (V, E) ← G
3:     best ← APL(c, G)                              ▷ Trivial solution
4:     choice ← ∅
5:     for e ∈ S do
6:         G′ ← (V, E ∪ e)
7:         APL′ ← APL(c, G′)          ▷ APL is undefined for negative cycles
8:         if APL′ < best then                ▷ False if APL′ is undefined
9:             best ← APL′
10:            choice ← {e}
11:        end if
12:    end for
13:    return choice
14: end procedure
```

4.2 Ward-Datta Algorithm

Ward and Datta proposed an algorithm that solves MinAPL more efficiently than the brute force method [20]. This section describes and explains in full the methods used in this algorithm. Lemmas 1 and 2 demonstrate a property of edge addition (see Fig. 4.1) that is fundamental to this algorithm.

Lemma 1. *Given a graph $G = (V, E)$ with weight function $w\colon E \to \mathbb{R}$, let $p = \langle v_0, v_1, v_2, \ldots, v_k \rangle$ be a shortest path from vertex v_0 to vertex v_k and, for any i and j such that $0 \leq i \leq j \leq k$, let $p_{ij} = \langle v_i, v_{i+1}, \ldots, v_j \rangle$ be the subpath of p from vertex v_i to v_j. Then, p_{ij} is a shortest path from v_i to v_j.*

Proof. See Cormen *et al.* [6, p. 645]. □

Lemma 2. *Given a graph $G = (V, E)$ with weight function $w\colon E \to \mathbb{R}$, and a directed edge e from $u \in V$ to $v \in V$, let $d_{i,j}$ be the length of the shortest path from i to j in G, $G' = (V, E \cup \{e\})$, and $d'_{i,j}$ be the length of the shortest path from i to j in G'. Then, $d'_{i,j} = \min(d_{i,j}, d_{i,u} + w(e) + d_{v,j})$.*

Proof. Let $p'_{i,j}$ be the shortest simple path from i to j in G', and $p_{i,j}$ be the shortest simple path from i to j in G. If $p'_{i,j}$ does not contain e, then an identical path $p_{i,j}$ must exist in G, and so $d'_{i,j} = d_{i,j}$. Otherwise, if $p'_{i,j}$ contains e, then $p'_{i,j}$ must contain subpaths from i to u and from v to j. By Lemma 1, any subpath of a shortest path must itself be a shortest path, meaning $p'_{i,u}$ and $p'_{v,j}$ are both subpaths of $p'_{i,j}$. As $p'_{i,j}$ is a simple path, and therefore must not contain a cycle, neither $p'_{i,u}$ nor $p'_{v,j}$ may contain e, and so identical paths $p_{i,u} = p'_{i,u}$ and $p_{v,j} = p'_{v,j}$ must exist in G. Therefore $p'_{i,j}$ is the concatenation of $p_{i,u}$, e, and $p_{v,j}$, and has length $d'_{i,j} = d_{i,u} + w(e) + d_{v,j}$. Combining the above cases for whether or not $p'_{i,j}$ contains e gives the relation $d'_{i,j} = \min(d_{i,j}, d_{i,u} + w(e) + d_{v,j})$. □

Given $d_{i,j}$ for all $i, j \in V$ and e, the result demonstrated in Lemma 2 gives a simple arithmetic expression for $d'_{i,j}$. By applying this expression for all i and j, the result of computing the APSP lengths in G can therefore be used to compute the APSP lengths, and hence APL, in G'. Algorithm 3 shows a method for solving MinAPL by applying this result for each candidate edge $e \in S$. The method excludes edges that would otherwise introduce a negative-weight cycle, because a negative-weight cycle can be thought of as a self-path with negative length, and this means $d'_{u,u} = \min(d_{u,u}, d_{u,u} + w(e) + d_{v,u}) = w(e) + d_{v,u} < 0$.

Using the analysis results for Algorithm 1 from Section 4.1, time complexity analysis of Algorithm 3 gives a run time complexity of $O(|S||V|^2 + apsp(|V|, |E|))$, where $apsp(|V|, |E|)$ is the run time of the APSP algorithm used. Using Pettie's

Algorithm 3 Ward-Datta algorithm for solving MinAPL

```
1: procedure MinAPL(c, G, S)
2:     (V, E) ← G
3:     d ← APSP(G)                          ▷ Compute the All-Pairs Shortest Paths
4:     best ← APL(c, G)                                          ▷ Trivial solution
5:     choice ← ∅
6:     for e ∈ S do
7:         (u, v) ← e
8:         if w(e) + d_{v,u} ≥ 0 then                  ▷ Avoid negative-weight cycles
9:             SPL ← 0
10:            SWF ← 0
11:            for i ∈ V do
12:                for j ∈ V do
13:                    d'_{i,j} ← min(d_{i,j}, d_{i,u} + w(e) + d_{v,j})
14:                    SPL ← SPL + c_{i,j} d'_{i,j}      ▷ Undefined if d'_{i,j} is undefined
15:                    SWF ← SWF + c_{i,j}
16:                end for
17:            end for
18:            APL ← SPL/SWF
19:            if APL < best then                          ▷ False if APL is undefined
20:                best ← APL
21:                choice ← {e}
22:            end if
23:        end if
24:    end for
25:    return choice
26: end procedure
```

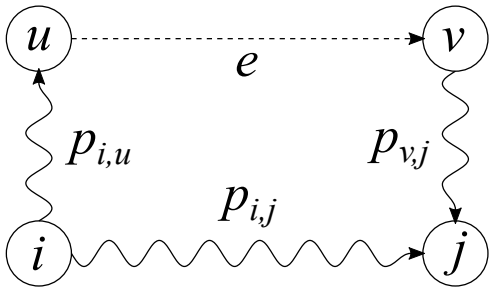

Figure 4.1: Shown are the shortest paths $p_{i,j} = \langle i, \ldots, j \rangle$, $p_{i,u} = \langle i, \ldots, u \rangle$, and $p_{v,j} = \langle v, \ldots, j \rangle$ with lengths $d_{i,j}$, $d_{i,u}$, and $d_{v,j}$ respectively. After adding the edge e to the graph, the shortest path from i to j is either unchanged or is now $p'_{i,j} = \langle i, \ldots, u, v, \ldots, j \rangle$, and has length $d'_{i,j} = \min(d_{i,j}, d_{i,u} + w(e) + d_{v,j})$.

$O(|V||E| + |V|^2 \log\log|V|)$ APSP algorithm [13] gives an overall time complexity for Algorithm 3 of $O(|S||V|^2 + |V||E| + |V|^2 \log\log|V|)$. Using the Floyd-Warhsall algorithm instead gives $O(|S||V|^2 + |V|^3)$. For dense graphs (defined by $|E| \in \Omega(|V|^2)$) these complexities are equivalent. This is a significant improvement in performance compared to the brute force method discussed in Section 4.1. It eliminates a full factor of $|V|$ in dense graphs, and performs at least as well as the brute force method in all cases. Memory complexity remains $O(|V|^2)$ in all cases, as it does in the brute force method.

4.3 Threshold Algorithm

The threshold algorithm is a novel algorithm based on the same underlying logic as the Ward-Datta algorithm described in Section 4.2, and affords significant improvements in performance for large $|S|$. This section describes the methods used to achieve these results. Recall the relation from Lemma 2:

$$d'_{i,j} = \min(d_{i,j}, d_{i,u} + w(e) + d_{v,j}) \tag{4.1}$$

where $d_{i,j}$ is the length of the shortest path from i to j in $G = (V, E)$, e is an edge from u to v, and $d'_{i,j}$ is the length of the shortest path from i to j in $G' = (V, E \cup \{e\})$. Using this result, the change in path length $\Delta d_{i,j}$ is defined as:

$$\Delta d_{i,j} = d'_{i,j} - d_{i,j} = \min(0, d_{i,u} + w(e) + d_{v,j} - d_{i,j}) \tag{4.2}$$

which has piecewise definition:

$$\Delta d_{i,j} = \begin{cases} d_{i,u} + w(e) + d_{v,j} - d_{i,j} & \text{if } d_{i,u} + w(e) + d_{v,j} - d_{i,j} < 0, \\ 0 & \text{else} \end{cases} \tag{4.3}$$

Note that for this formulation to be well-defined, $d_{i,u} + w(e) + d_{v,j}$ and $d_{i,j}$ must not both be infinite, and ideally should be strictly real-valued. This precludes an infinite disconnection cost D (as described in Section 2.1). However, the same results are achieved by selecting a sufficiently large finite value for D such that it is greater than all values that will be compared to it. A threshold value $T_{i,v,j}$ is defined, and Eq. (4.3) is rearranged as follows:

$$T_{i,v,j} = d_{i,j} - d_{v,j} \tag{4.4}$$

$$\Delta d_{i,j} = \begin{cases} d_{i,u} + w(e) - T_{i,v,j} & \text{if } d_{i,u} + w(e) < T_{i,v,j}, \\ 0 & \text{else} \end{cases} \tag{4.5}$$

Figure 4.2 gives a graphical representation of the threshold value. The total change in the SPL as a result of adding edge e to E is:

$$\Delta\text{SPL} = \sum_{i \in V} \sum_{j \in V} c_{i,j} \Delta d_{i,j} \tag{4.6}$$

Observe that only values of i and j such that $d_{i,u} + w(e) < T_{i,v,j}$ contribute to the sum in Eq. (4.6), and that the left-hand side of this condition is independent of j. Therefore the subset $V_i' \subseteq V$ can be defined and substituted into Eq. (4.6) as follows:

$$V_i' = \{j \in V \mid d_{i,u} + w(e) < T_{i,v,j}\} \tag{4.7}$$

$$\begin{aligned} \Delta\text{SPL} &= \sum_{i \in V} \sum_{j \in V_i'} c_{i,j} \Delta d_{i,j} = \sum_{i \in V} \sum_{j \in V_i'} c_{i,j} (d_{i,u} + w(e) - T_{i,v,j}) \\ &= \sum_{i \in V} \left((d_{i,u} + w(e)) \sum_{j \in V_i'} c_{i,j} - \sum_{j \in V_i'} c_{i,j} T_{i,v,j} \right) \end{aligned} \tag{4.8}$$

If the elements of $\{T_{i,v,j} \mid j \in V\}$ are sorted by ascending value, the elements of $\{T_{i,v,j} \mid j \in V_i'\}$ form a contiguous suffix of the sorted list. A binary search on the sorted elements gives the bounds of this suffix. A pair of cumulative sum arrays can then be used to compute the two sums over V_i' in Eq. (4.8). A cumulative sum array is a simple data structure that allows the sum of a contiguous range of elements in a list to be computed in constant time. Let C be a sum array for some given numeric list. The n-indexed element C_n of the sum array is the sum of the first n elements of the original list. The sum of all elements in the index

Algorithm 4 Threshold algorithm for solving MinAPL

procedure MinAPL(c, G, S)
  $(V, E) \leftarrow G$
  $d \leftarrow \mathrm{APSP}(G)$ ▷ Compute the All-Pairs Shortest Paths
  $\Delta\mathrm{SPL}_e \leftarrow 0 \; \forall \, e \in S$ ▷ Initialise change in SPL for each edge
  for $i \in V$ **do**
    for $v \in V$ **do**
      for $j \in V$ **do**
        $T_{i,v,j} \leftarrow d_{i,j} - d_{v,j}$ ▷ Compute thresholds as in (4.4)
      end for
      $J_{i,v} \leftarrow \textsc{SortInds}(V, T)$ ▷ Sort $j \in V$ by value $T_{i,v,j}$
      $W_{v,0} \leftarrow 0$ ▷ Cumulative sum for c-weighted T ordered by J
      $C_{v,0} \leftarrow 0$ ▷ Cumulative sum for weights c ordered by J
      for $ind \in [0, |V|)$ **do**
        $j \leftarrow J_{i,v,ind}$
        $T'_{i,v,ind} \leftarrow T_{i,v,j}$ ▷ T sorted by ascending value
        $W_{v,ind+1} \leftarrow W_{v,ind} + c_{i,j} T_{i,v,j}$
        $C_{v,ind+1} \leftarrow C_{v,ind} + c_{i,j}$
      end for
    end for
    for $e \in S$ **do**
      $(u, v) \leftarrow e$
      if $w(e) + d_{v,u} \geq 0$ **then** ▷ Avoid negative-weight cycles
        $L \leftarrow d_{i,u} + w(e)$ ▷ Path length from i to v through e
        $l \leftarrow \textsc{BinarySearchUpperBound}(T'_{i,v}, L)$
        $\Delta\mathrm{SPL}_e \leftarrow \Delta\mathrm{SPL}_e + (C_{v,|V|} - C_{v,l})L - (W_{v,|V|} - W_{v,l})$
      end if
    end for
  end for
  best $\leftarrow 0$ ▷ Trivial solution
  choice $\leftarrow \emptyset$
  for $e \in S$ **do**
    if $\Delta\mathrm{SPL}_e <$ *best* **then**
      best $\leftarrow \Delta\mathrm{SPL}_e$
      choice $\leftarrow \{e\}$
    end if
  end for
  return *choice*
end procedure

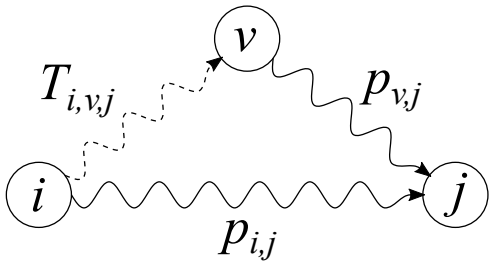

Figure 4.2: Shown are the shortest paths $p_{i,j} = \langle i, \ldots, j \rangle$ and $p_{v,j} = \langle v, \ldots, j \rangle$ with lengths $d_{i,j}$ and $d_{v,j}$ respectively. The threshold value $T_{i,v,j} = d_{i,j} - d_{v,j}$ is the required length of the hypothetical shortest path $p_{i,v} = \langle i, \ldots, v \rangle$ such that the shortest path $p_{i,v,j} = \langle i, \ldots, v, \ldots, j \rangle$ has length equal to that of $p_{i,j}$.

range $[i, j)$ is then $C_j - C_i$. The cumulative sum array for any given list can be computed in linear time in the size of the input list. Algorithm 4 shows a method for solving MinAPL using this result.

The function SortInds(V, T) on Line 10 of Algorithm 4 is a function that sorts the elements $j \in V$ in ascending order according to the corresponding value of $T_{i,v,j}$. This gives an ordering $J_{i,v}$ of V for which $T_{i,v,j}$ increases monotonically, which can be used to compute the cumulative sum arrays W_v and C_v, as well as a monotonically increasing ordering $T'_{i,v}$ of $T_{i,v}$. The function BinarySearchUpperBound$(T'_{i,v}, L)$ on Line 24 of Algorithm 4 is a function that performs a binary search over the ordering $T'_{i,v}$ and returns an index l, the minimum 0-based index for which $T'_{i,v,l} > L$, or $|T'_{i,v}|$ if no such l exists. This index is then used on Line 25 of Algorithm 4 in conjunction with the aforementioned cumulative sum arrays to compute a partial sum from Eq. (4.8) for a particular i and e. By looping over all $i \in V$ and all candidate edges $e \in S$ it is possible to compute the change in SPL as a result of adding edge e, ΔSPL_e, and e can therefore be selected to minimise the SPL and hence the APL in $G' = (V, E \cup \{e\})$.

Time complexity analysis of Algorithm 4 gives a run time complexity of $O(apsp(|V|, |E|) + |V|^3 \log|V| + |V||S| \log|V|)$, where $apsp(|V|, |E|)$ is the run time of the APSP algorithm used. Using Pettie's $O(|V||E| + |V|^2 \log\log|V|)$ APSP algorithm [13] gives an overall time complexity for Algorithm 3 of $O(|V|^3 \log|V| + |V||S| \log|V|)$. Using the Floyd-Warhsall algorithm instead gives the same result, as both APSP algorithms are dominated by the $O(|V|^3 \log|V|)$ component of the algorithm. For large $|S|$ (i.e., $|S| \in \Omega(|V|^2)$), this is a significant improvement in performance over the Ward-Datta algorithm described in Section 4.2, giving run time $O(|V|^3 \log|V|)$ as opposed to $O(|V|^4)$. For small $|S|$, however, the Ward-

Table 4.1: Example weighting factors $c_{i,j}$ for $i = 0$.

j	0	1	2	3	4	5	6
$c_{i,j}$	0	2	1	2	1	1	2

Datta algorithm will outperform this algorithm. Memory complexity remains $O(|V|^2)$ in all cases, as in both the brute force method and the Ward-Datta algorithm.

4.3.1 Example

This example is a partial demonstration of the threshold algorithm for the graph shown in Fig. 4.3, with weighting factors $c_{i,j}$ from Table 4.1. For the sake of brevity, the algorithm is demonstrated only for the subset of paths starting at $i = 0$. Threshold values for this example are computed from the shortest path lengths in Table 4.2, and listed in Table 4.3. To compute the result of adding the single edge $e = (1, 4) = (u, v)$ to the graph, consider the $v = 4$ row of Table 4.3. Table 4.4 gives this row sorted into ascending order, with corresponding weighting factors $c_{i,j}$ and weighted threshold values $c_{i,j}T_{i,v,j}$. Table 4.5 lists the cumulative sum arrays for these values. A binary search for the upper bound of $d_{i,u} + w(e) = 2$ finds the highlighted column from Table 4.5. Subtracting these values from their corresponding overall sums gives the values of the sums $\sum_{j \in V_i'} c_{i,j}$ and $\sum_{j \in V_i'} c_{i,j}T_{i,v,j}$. Substituting these values into Eq. (4.8) gives the total change in shortest path length as a result of adding e for all paths starting at i:

$$\begin{aligned} \Delta\mathrm{SPL}_i &= (d_{i,u} + w(e)) \sum_{j \in V_i'} c_{i,j} - \sum_{j \in V_i'} c_{i,j}T_{i,v,j} \\ &= 2(9 - 4) - (15 - -22) \\ &= -27 \end{aligned} \tag{4.9}$$

Repeating this process for each starting vertex $i \in V$ and summing the results gives the total change in SPL as a result of adding e to the graph. Note that it is not necessary to recompute the cumulative sum arrays for different e. Applying this method for all $e \in S$ finds the edge that minimises the APL.

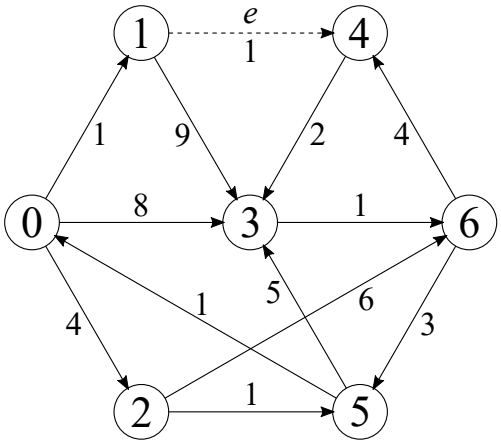

Figure 4.3: Example graph with edge weights.

Table 4.2: Shortest path lengths $d_{i,j}$

i \ j	0	1	2	3	4	5	6
0	0	1	4	8	13	5	9
1	14	0	18	9	14	13	10
2	2	3	0	6	10	1	6
3	5	6	9	0	5	4	1
4	7	8	11	2	0	6	3
5	1	2	5	5	10	0	6
6	4	5	8	6	4	3	0

Table 4.3: Threshold values $T_{i,v,j} = d_{i,j} - d_{v,j}$ for $i = 0$.

v \ j	0	1	2	3	4	5	6
0	0	0	0	0	0	0	0
1	-14	-1	-14	-1	-1	-8	-1
2	-2	-2	-4	2	3	4	3
3	-5	-5	-5	8	8	1	8
4	-7	-7	-7	6	13	-1	6
5	-1	-1	-1	3	3	5	3
6	-4	-4	-4	2	9	2	9

Table 4.4: Sorted threshold values $T_{i,v,j}$ for $i = 0$, $(u, v) = e = (1, 4)$, with corresponding $c_{i,j}T_{i,v,j}$ and $c_{i,j}$.

j	0	1	2	5	3	6	4
$T_{i,v,j}$	-7	-7	-7	-1	6	6	13
$c_{i,j}T_{i,v,j}$	0	-14	-7	-1	12	12	13
$c_{i,j}$	0	2	1	1	2	2	1

Table 4.5: Sorted threshold values $T_{i,v,j}$ for $i = 0$, $(u, v) = e = (1, 4)$, with corresponding cumulative sums for $c_{i,j}T_{i,v,j}$ and $c_{i,j}$.

$T_{i,v,j}$	-7	-7	-7	-1	6	6	13	
$\sum c_{i,j}T_{i,v,j}$	0	0	-14	-21	-22	-10	2	15
$\sum c_{i,j}$	0	0	2	3	4	6	8	9

CHAPTER 5

Empirical Analysis

This section reviews the efforts made to empirically analyse the correctness and performance of the algorithms described in Chapter 4. The algorithms were implemented in C++11 and were compiled and run using the GNU C++ Compiler version 5.4.0 with `-O2` optimisation enabled under Kubuntu Linux version 16.04 running on an Intel Core i7-6700K processor. The Floyd-Warshall algorithm was used to compute the APSP in this implementation.

5.1 Random Graph Generation

To test these algorithms thoroughly requires a source of pseudorandom weighted directed graphs without negative-weight cycles—negative-weight cycles would trivialise the problem. For the sake of this analysis, these graphs are constructed programmatically, given $|V|$, $|E|$ and a minimum and maximum edge weight, by selecting a random ordering of all $|V|^2 - |V|$ possible edges. Edges are then added to the graph one by one from the random ordering, with weights selected randomly such that they remain within the specified bounds and do not form a negative cycle with any edges already in the graph. If no such weight is possible the edge is discarded. This continues until either the graph contains the desired number of edges or the random ordering is exhausted, in which case the graph generator has failed. The likelihood of this method failing is dependent on the size of the graph and the weight range given, but failure is very unlikely for appropriate weight limits.

5.2 Correctness Validation

The three MinAPL algorithms described in Chapter 4 were run repeatedly on randomly generated input, and their outputs were compared. If any two algorithms disagreed on the minimum achievable APL the test case was considered

to have failed. The algorithms were tested for $|V| = 50$, randomly selected $|E| \in [250, 2000]$, and $|S| = 1000$, with weights in the range $w(e) \in [-50, 150]$, weighting factors in the range $c_{i,j} \in [0, 5]$, and $D = 1000|V|^3 c_{\max} w_{\max} = 1000(50^3)(5)(150)$. All three MinAPL algorithms were validated against more than 30,000 test cases for integer-valued edge weights and weighting factors, and over 120,000 test cases for real-valued edge weights and weighting factors. No tests failed.

5.3 Run Time Analysis

The three MinAPL algorithms described in Chapter 4 were run repeatedly on randomly generated input, and their run time recorded. Timing data was collected using C++'s `chrono` library, and processes were given maximum priority to minimise noise due to external processes. The algorithms were tested for $|S| = 0, 100, 200, \ldots, 10000$, and for $|V|$ automatically selected for each value of $|S|$ and for each algorithm to cover all run times up to 500 milliseconds to a time resolution of no more than 20 milliseconds. As $|E|$ does not affect the run time performance of any of these implementations, all tests were performed with $|E| = \lfloor (3/4)|V|(|V| - 1) \rfloor$. Similarly, these tests were performed with uniform weighting factors $c_{i,j} = 1$ for all $i, j \in V$, as the values of these factors have no effect on run time.

Figures 5.1–5.6 show the results of these tests plotted jointly for comparison over $|S| = 100, 500, 1000, 2000, 5000, 10000$. It is readily apparent from these figures that the run time of the the brute force algorithm far exceeds that of the Ward-Datta and threshold algorithms for any significant value of $|S|$. They also show that while the Ward-Datta algorithm grows more slowly with $|V|$, and hence initially outperforms the threshold algorithm, this does not hold true as $|S|$ increases. From close inspection it appears that the run time of both the Ward-Datta and the threshold algorithm grow linearly with $|S|$, but the run time of the Ward-Datta algorithm grows much faster than that of the threshold algorithm. These results agree with the time complexity analyses from Chapter 4. Some noise is present in the run time for the threshold algorithm, though this is suspected to be a result of sorting randomized data, and does not appear to affect performance significantly.

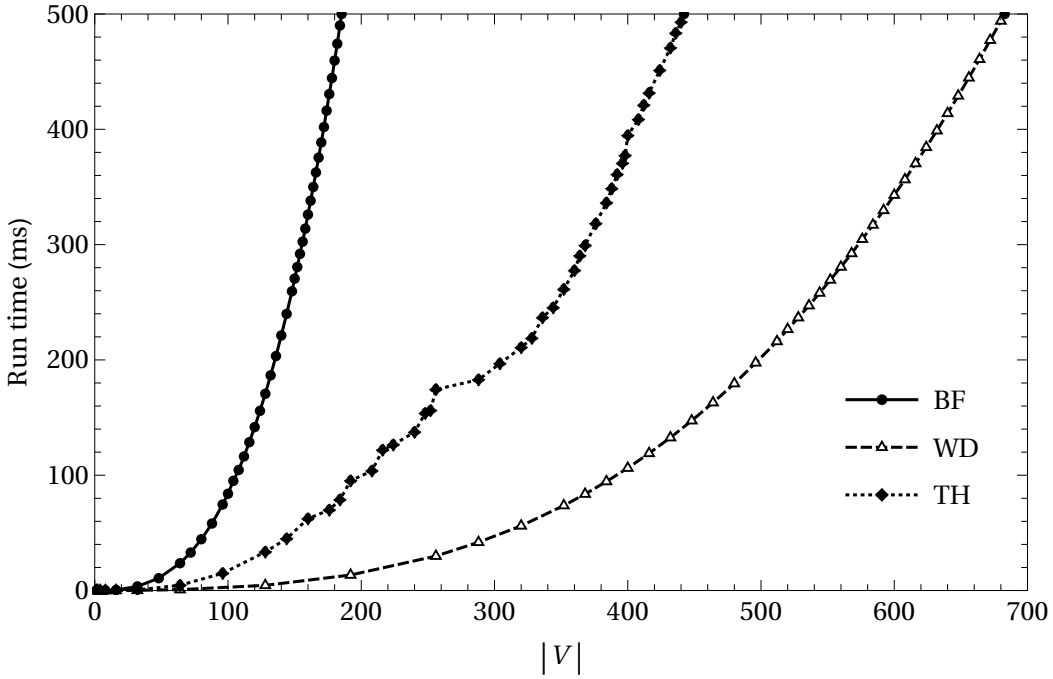

Figure 5.1: Run times for brute force (BF), Ward-Datta (WD), and threshold (TH) algorithms for $|S| = 100$

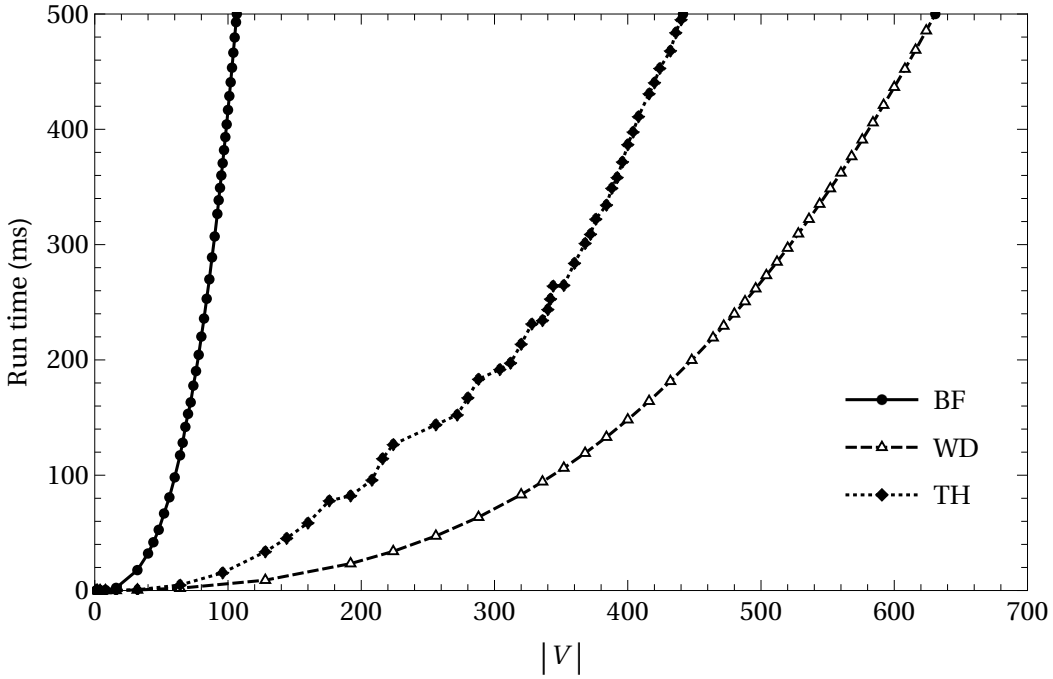

Figure 5.2: Run times for brute force (BF), Ward-Datta (WD), and threshold (TH) algorithms for $|S| = 500$

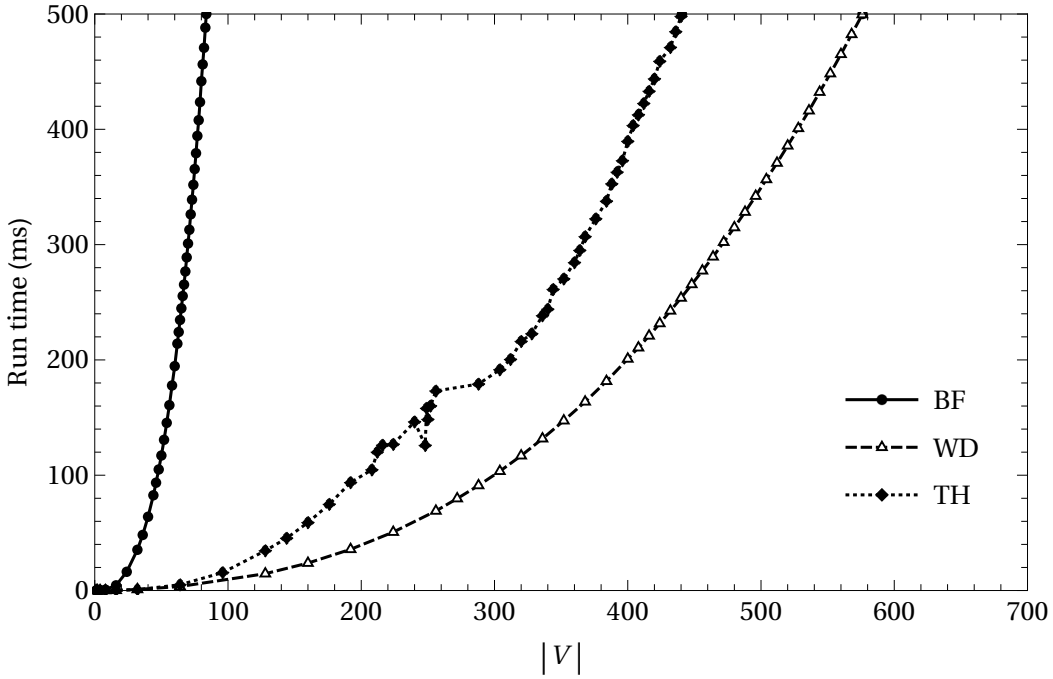

Figure 5.3: Run times for brute force (BF), Ward-Datta (WD), and threshold (TH) algorithms for $|S| = 1000$

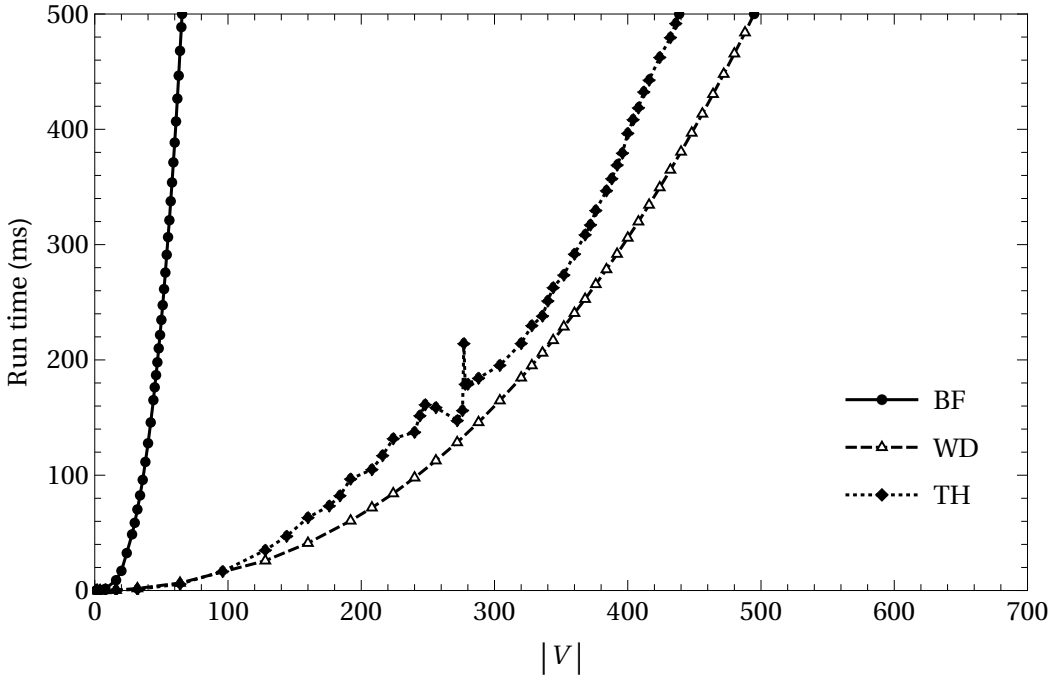

Figure 5.4: Run times for brute force (BF), Ward-Datta (WD), and threshold (TH) algorithms for $|S| = 2000$

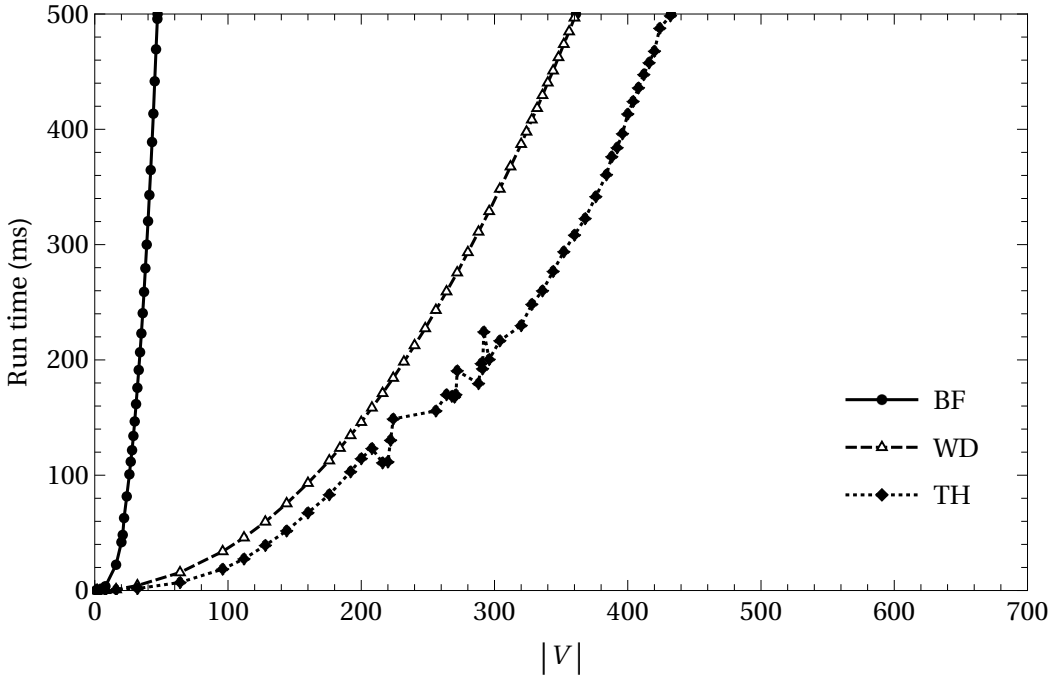

Figure 5.5: Run times for brute force (BF), Ward-Datta (WD), and threshold (TH) algorithms for $|S| = 5000$

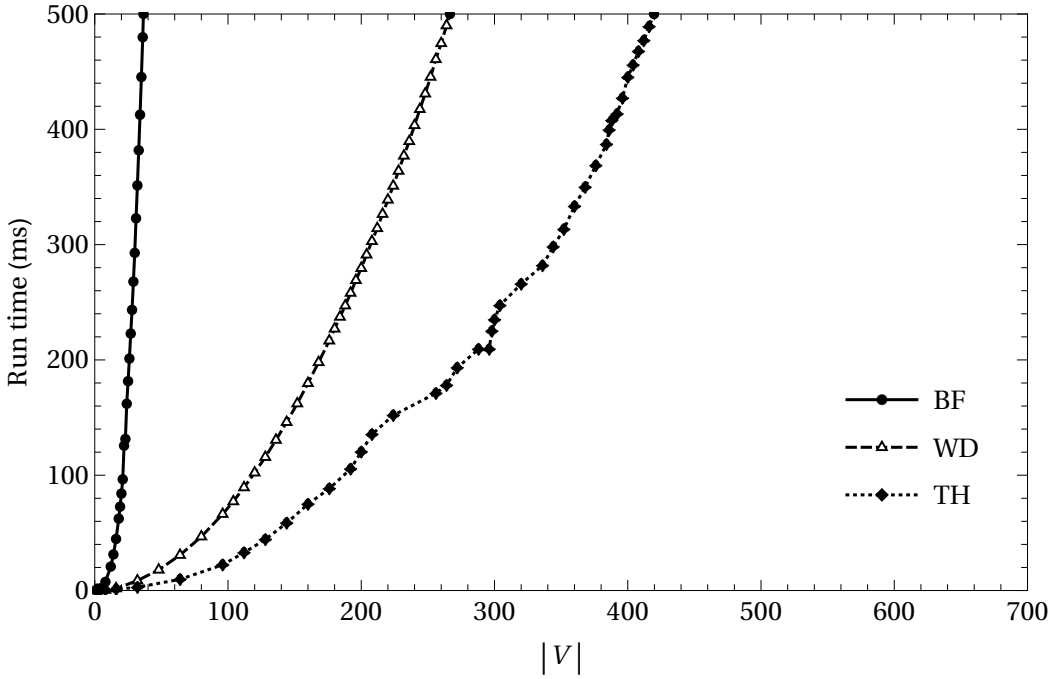

Figure 5.6: Run times for brute force (BF), Ward-Datta (WD), and threshold (TH) algorithms for $|S| = 10000$

CHAPTER 6

Algorithm Extensions

This section explores extensions and modifications that can be made to the algorithms described in Chapter 4 in order to solve related problems.

6.1 k-MinAPL

It is possible to adapt the Ward-Datta MinAPL algorithm described in Section 4.2 to solve k-MinAPL. The fundamental principal behind the Ward-Datta algorithm is its ability to take a precomputed set of APSP lengths $d_{i,j}$ for a graph $G = (V, E)$ and use this to generate the APSP lengths $d'_{i,j}$ for a graph $G' = (V, E \cup \{e\})$ for some edge e in $O(|V|^2)$ time. The same method can be used to add another edge to the graph repeatedly in order to add any arbitrary set of edges F to the graph in $O(|F||V|^2)$ time. The number of subsets $F \subseteq S$ of size exactly $|F| = f$ is $\binom{|S|}{f} = |S|!/f!/(|S| - f)!$, and the minimum APL can be computed as a result of adding exactly f edges from S in $O(\binom{|S|}{f} f|V|^2)$ time, which has a safe upper bound of $O(|S|^f f|V|^2)$. The minimum APL value is a result of adding up to k edges from S in $O(\sum_{f=0}^{k} \binom{|S|}{f} f|V|^2)$, which has a safe upper bound of $O(|S|^k k|V|^2)$.

6.2 Undirected Edges

Both the Ward-Datta and threshold algorithms can readily be adapted to solve MinAPL in undirected graphs. First, the input graph is converted to a directed graph by replacing each undirected edge between u and v with a pair of directed edges (u, v) and (v, u), each with the same weight as the original edge. Note that to prevent negative-weight cycles, undirected graphs must not have negative edge weights, because even a single negative-weight edge introduces a trivial negative cycle between the two ends of the edge. Adding a new undirected edge to the

graph is equivalent to adding a pair of opposite-facing directed edges of the same weight. It is simple to adapt the Ward-Datta algorithm to handle this, because the result from Lemma 2 on which the algorithm is based can be adapted to undirected graphs by considering traversing the edge in either direction, which gives $d'_{i,j} = \min(d_{i,j}, d_{i,u} + w(e) + d_{v,j}, d_{i,v} + w(e) + d_{u,j})$. Adapting the threshold algorithm, however, requires a less intuitive observation.

Observe that the threshold algorithm only counts a path towards the total change in SPL if $d_{i,u} + w(e) < T_{i,v,j} = d_{i,j} - d_{v,j}$. If two edges are simply added at once, there is a risk of counting the same path twice if both $d_{i,u} + w(e) < d_{i,j} - d_{v,j}$ and $d_{i,v} + w(e) < d_{i,j} - d_{u,j}$. Observe that for all i, j, and u, $d_{i,j} \leq d_{i,u} + d_{u,j}$, because otherwise $d_{i,j}$ cannot be the length of the shortest path, as $d_{i,u} + d_{u,j}$ is shorter. Substituting this relation into the conditions above implies that in order to count the same path twice, both $d_{i,u} + w(e) < d_{i,v}$ and $d_{i,v} + w(e) < d_{i,u}$ must hold. It is readily apparent that there is no valid (that is, non-negative) value for the edge weight $w(e)$ that satisfies both of these conditions, and hence the threshold algorithm will count each path at most once. This means that the threshold algorithm can be adapted to solve undirected MinAPL by simply computing and summing the change in SPL as a result of traversing the new edge in either direction.

6.3 Candidate Edge Set Reduction

In most cases, the size of the candidate edge set S will be $|S| \in O(|V|^2)$. In the event that this is not the case it is possible to reduce the set of candidate edges S so as to further optimise the Ward-Datta algorithm and the threshold algorithm for large S. Firstly, discard all edges that would otherwise introduce a negative-weight cycle (That is, $w(e) + d_{v,u} < 0$). Now consider the case where S contains multiple edges from u to v with a range of weights. Lemma 2 gives the relation $d'_{i,j} = \min(d_{i,j}, d_{i,u} + w(e) + d_{v,j})$. Hence, for a pair of edges $e, f \in S$, both going from u to v and with $w(e) \leq w(f)$, $\min(d_{i,j}, d_{i,u} + w(e) + d_{v,j}) \leq \min(d_{i,j}, d_{i,u} + w(f) + d_{v,j})$. It follows from this that the new shortest path length $d'_{i,j}$, and hence the APL produced by adding e to G, is no greater than that produced by adding f, and therefore e is at least as good a solution to the MinAPL problem as f. For each pair of vertices $u, v \in V$, there exists a single edge from u to v that has weight no greater than all other edges from u to v, and it is sufficient to consider only this edge. Algorithm 5 shows a method for performing this reduction.

Analysis of Algorithm 5 shows that it is able to reduce S to size $|S| \leq |V|^2 - |V|$ using $O(|S|)$ time and $O(|V|^2)$ memory. The algorithm does require the set of

Algorithm 5 Algorithm for eliminating unnecessary elements in S

```
1: procedure ReduceS(d, S)
2:     for e ∈ S do
3:         (u, v) ← e
4:         if w(e) + d_{v,u} ≥ 0 then                ▷ Avoid negative-weight cycles
5:             if w(e) < w(S'_{u,v}) then
6:                 S'_{u,v} ← e
7:             end if
8:         end if
9:     end for
10:    return S'
11: end procedure
```

shortest path lengths as input, but computing this is already part of both the Ward-Datta algorithm and the threshold algorithm. Performing this algorithm as a preprocessing step gives an overall time complexity for the Ward-Datta algorithm of $O(|V|^4+|S|)$, and an overall time complexity for the threshold algorithm of $O(|V|^3 \log|V|+|S|)$. For both algorithms this is a significant improvement only for $|S| \in \Omega(|V|^2)$.

CHAPTER 7

Conclusion

Since their discovery by Milgram [3], small-world networks and their properties have been a topic of significant interest in graph theory. Their helpful properties of low APL and high clustering coefficient have driven attempts to construct methods for converting arbitrary graphs to small-world graphs [4], [5]. These methods have their failings however, and minimising the APL in a fixed-size graph through edge addition proved to be a challenging optimisation problem, with the general problem of k-MinAPL being shown to be NP-hard [7]. The simpler MinAPL problem of selecting a single edge to minimise the APL in a graph has recently become a topic of interest [16]. The research described in this report is motivated by the applications of this problem in network optimisation for high-performance multi-core processor design [1], [2], [18], as well as potential applications resulting from efficiently solving the dynamic APSP and related problems [19]. However, much work remains to be done in investigating efficient methods for solving MinAPL. One potential avenue of enquiry is to develop methods for solving this problem in constrained, non-general graphs, where the additional properties of the graph may simplify the problem. This report presents a pair of efficient algorithms for solving MinAPL in general graphs, providing significant improvements in performance over the previous best known methods.

APPENDIX A

Original Honours Proposal

To: Whomsoever it may concern

From: Andrew Gozzard, 20948678@student.uwa.edu.au

Subject: Honours Project Proposal

Title: Investigation of Average Shortest Path Distance Minimisation via Shortcut Edge Addition in Non-General Graphs

Date: 2016-10-22

In accordance with the requirements of my course, I would like this document to be considered as a formal project proposal for my Honours in Computer Science and Software Engineering. I hope to undertake this project under the supervision of Professor Amitava Datta and his PhD candidate, Max Ward.

1 Introduction

Many real-world networks suffer from end-to-end delays. These delays are the result of the additive effect of long, multi-hop paths through the network. Fall [2] has previously shown that the addition of a few random links to the network can significantly reduce the average end-to-end delay in a network. By modelling these networks as graphs, where the weight of an edge corresponds to the link transmission latency, the average end-to-end delay becomes the average shortest path length (APL) across all pairs of vertices. The problem of selecting which links to add so as to minimise the average end-to-end delay in the network now becomes a graph theoretic problem with an algorithmic solution. Meyerson and Tagiku [4] have shown that the general problem of adding k edges to a general graph in order to minimise the APL is NP-hard, and have designed a number of approximation algorithms aimed at achieving good solutions to this problem in reasonable time. Recently, Gaur *et al.* studied the related problem of converting an arbitrary, unweighted, undirected network to a *small-world* network through link addition [3]. A small-world network is defined to be a graph where the APL grows proportionally with the logarithm of the number of vertices in the graph, and hence the shortest path between two nodes tends to be relatively small. Gaur *et al.* propose an $O(V^4 \log V)$ algorithm for selecting the best single edge to add to an unweighted, undirected graph $G = (V, E)$ to minimise the APL in the graph. Ward and Datta have proposed the existence of an $O(V^4)$ algorithm, though their work awaits peer review [6].

These works have real-world applications including the optimisation of computer networks [2, 3], as well as in microprocessor design [4], and in Network-on-Chip (NoC) development in Very-Large Scale Integration (VLSI) design [5]. In a number of these applications, the network in question can be modelled as some class of non-general graph; for instance, VLSI design problems can be modelled as circle graphs

[1], and wireless communication networks can be modelled as Euclidean graphs due to the nature of the electromagnetic transmission medium. Despite this, preliminary readings suggest no or very little work has been done on APL minimisation in these classes of non-general graphs.

The aim of this project, therefore, is the investigation of average shortest path distance minimisation via shortcut edge addition in non-general graphs. Specifically, to explore the possibility of there being significantly faster algorithms for solving this problem in those classes of non-general graphs that have real-world networking applications, such as those given above.

2 Aims and Open Problems

In my work I aim to investigate in particular the single edge addition APL minimisation problem (MinAPL) and the k edge addition version of the same (k-MinAPL) in different classes of non-general graphs. A full literature review would allow me to determine which classes of non-general graph are most applicable to the real-world and so allow me to prioritise particular avenues of enquiry. Preliminary readings suggest that circle and Euclidean graphs hold the most promise and so are the most immediately worthy of attention. As such, the open problems I hope to address include:

- Does there exist an algorithmic solution to MinAPL or k-MinAPL on circle graphs that is of lower time complexity than the best known solution on general graphs?
- Does there exist an algorithmic solution to MinAPL or k-MinAPL on 2D Euclidean graphs that is of lower time complexity than the best known solution on general graphs?
- (Extension) Does there exist an algorithmic solution to MinAPL or k-MinAPL on higher-dimensional Euclidean graphs that is of lower time complexity than the best known solution on general graphs?
- (Extension) Does there exist an algorithmic solution to MinAPL or k-MinAPL on other classes of non-general graphs (or combinations thereof) that is of lower time complexity than the best known solution on general graphs?

Problems marked with (Extension) may be subject to progress and developments made on other related problems. A complete literature review and further investigation may reveal other promising open problems. Hence this list may be subject to change.

3 Time-line and Planning

Of course the first phase in my work for this project would be to complete a full literature review. Doing so would allow me to prioritise the various open problems

given above and construct a schedule for working on them. As a requirement of my course this literature review is required to be completed by the end of first semester, though I hope to have it completed well ahead of time to allow myself enough time to investigate as many open problems as possible. I then intend to spend some time investigating the common background of the problems and the theory of MinAPL and k-MinAPL in general graphs to ensure a proper understanding of the problems. The aforementioned schedule would then allow some time to each problem. The schedule will include an overflow period to leave time for any problems that require extra work, and time to prepare the work for submission before the required deadlines. With this plan I believe myself capable of addressing most if not all of the open problems given above.

4 Closing Note

With what investigation I have been able to complete since learning of this project, I believe I have the ability to do good work in this topic. I also believe that this work will be quite valuable in the real-world applications mentioned above. I hope this proposal appears as promising to you as it does to me. Please do not hesitate to contact me with any enquiry related to this proposal, and I will answer to the best of my ability.

References

[1] Jingsheng Cong and C L Liu. Over-the-cell channel routing. In *Proceedings of International Conference on Computer-Aided Design*, pages 80–83. Ieee, 1988.

[2] Kevin Fall. A delay-tolerant network architecture for challenged internets. In *Proceedings of the 2003 conference on Applications, technologies, architectures, and protocols for computer communications*, pages 27–34. ACM, 2003.

[3] Nidhi Gaur, Arpan Chakraborty, and BS Manoj. Delay optimized small-world networks. *Communications Letters, IEEE*, 18(11):1939–1942, 2014.

[4] Adam Meyerson and Brian Tagiku. Minimizing average shortest path distances via shortcut edge addition. In *Approximation, Randomization, and Combinatorial Optimization. Algorithms and Techniques*, pages 272–285. Springer, 2009.

[5] Umit Y Ogras and Radu Marculescu. "It's a small world after all": NoC performance optimization via long-range link insertion. *Very Large Scale Integration (VLSI) Systems, IEEE Transactions on*, 14(7):693–706, 2006.

[6] Max Ward and Amitava Datta. Converting a network into a small-world network: A fast algorithm for minimizing average path length through link addition. Unpublished.

Bibliography

[1] M.-C. F. Chang, E. Socher, S.-W. Tam, J. Cong, and G. Reinman, "Rf interconnects for communications on-chip," in *Proceedings of the 2008 international symposium on Physical design*, ACM, 2008, pp. 78–83.

[2] M. F. Chang, J. Cong, A. Kaplan, M. Naik, G. Reinman, E. Socher, and S.-W. Tam, "Cmp network-on-chip overlaid with multi-band rf-interconnect," in *High Performance Computer Architecture, 2008. HPCA 2008. IEEE 14th International Symposium on*, IEEE, 2008, pp. 191–202.

[3] S. Milgram, "The small world problem," *Psychology today*, vol. 2, no. 1, pp. 60–67, 1967.

[4] D. J. Watts and S. H. Strogatz, "Collective dynamics of small-world networks," *Nature*, vol. 393, no. 6684, pp. 440–442, 1998.

[5] F. Comellas and M. Sampels, "Deterministic small-world networks," *Physica A: Statistical Mechanics and its Applications*, vol. 309, no. 1, pp. 231–235, 2002.

[6] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein, *Introduction to algorithms*. MIT press Cambridge, 2001, vol. 6.

[7] A. Meyerson and B. Tagiku, "Minimizing average shortest path distances via shortcut edge addition," in *Approximation, Randomization, and Combinatorial Optimization. Algorithms and Techniques*, Springer, 2009, pp. 272–285.

[8] E. W. Dijkstra, "A note on two problems in connexion with graphs," *Numerische mathematik*, vol. 1, no. 1, pp. 269–271, 1959.

[9] M. L. Fredman and R. E. Tarjan, "Fibonacci heaps and their uses in improved network optimization algorithms," *Journal of the ACM (JACM)*, vol. 34, no. 3, pp. 596–615, 1987.

[10] R. W. Floyd, "Algorithm 97: Shortest path," *Communications of the ACM*, vol. 5, no. 6, p. 345, 1962.

[11] S. Warshall, "A theorem on boolean matrices," *Journal of the ACM (JACM)*, vol. 9, no. 1, pp. 11–12, 1962.

[12] D. B. Johnson, "Efficient algorithms for shortest paths in sparse networks," *Journal of the ACM (JACM)*, vol. 24, no. 1, pp. 1–13, 1977.

[13] S. Pettie, "A new approach to all-pairs shortest paths on real-weighted graphs," *Theoretical Computer Science*, vol. 312, no. 1, pp. 47–74, 2004.

[14] M. Thorup, "Undirected single-source shortest paths with positive integer weights in linear time," *Journal of the ACM (JACM)*, vol. 46, no. 3, pp. 362–394, 1999.

[15] T. Hagerup, "Improved shortest paths on the word ram," in *Automata, Languages and Programming*, Springer, 2000, pp. 61–72.

[16] N. Gaur, A. Chakraborty, and B. Manoj, "Delay optimized small-world networks," *Communications Letters, IEEE*, vol. 18, no. 11, pp. 1939–1942, 2014.

[17] U. K.R. K. R. and K. V. Iyer, "All-pairs shortest-paths problem for unweighted graphs in $O(n^2 \log n)$ time," *World Academy of Science, Engineering and Technology, International Journal of Computer, Electrical, Automation, Control and Information Engineering*, vol. 3, no. 2, pp. 320–326, 2009.

[18] U. Y. Ogras and R. Marculescu, ""It's a small world after all": NoC performance optimization via long-range link insertion," *Very Large Scale Integration (VLSI) Systems, IEEE Transactions on*, vol. 14, no. 7, pp. 693–706, 2006.

[19] C. Demetrescu and G. F. Italiano, "A new approach to dynamic all pairs shortest paths," *Journal of the ACM (JACM)*, vol. 51, no. 6, pp. 968–992, 2004.

[20] M. Ward and A. Datta, Personal Communication, 2016.