2013

On the Design, Analysis, and Implementation of Algorithms for Selected Problems in Graphs and Networks

Matthew D. Williamson
West Virginia University

On the Design, Analysis, and Implementation of Algorithms for Selected Problems in Graphs and Networks

by

Matthew D. Williamson

Dissertation submitted to the
College of Engineering and Mineral Resources
at West Virginia University
in partial fulfillment of the requirements
for the degree of

Doctor of Philosophy
in
Computer Science

Elaine Eschen, Ph.D.
Hong-Jian Lai, Ph.D.
James Mooney, Ph.D.
Arun Ross, Ph.D.
K. Subramani, Ph.D., Chair

Lane Department of Computer Science and Electrical Engineering

Morgantown, West Virginia
2013

Keywords: Graph Theory, Algorithms, Minimum Spanning Tree Verification, Negative Cost Cycle Detection, Undirected Graphs, Girth

Abstract

On the Design, Analysis, and Implementation of Algorithms for Selected Problems in Graphs and Networks

by

Matthew D. Williamson
Doctor of Philosophy in Computer Science

West Virginia University

This thesis studies three problems in network optimization, viz., the minimum spanning tree verification (MSTV) problem, the undirected negative cost cycle detection (UNCCD) problem, and the negative cost girth (NCG) problem. These problems find applications in several domains including program verification, proof theory, real-time scheduling, social networking, and operations research.

The MSTV problem is defined as follows: Given an undirected graph $G = (V, E)$ and a spanning tree T, is T a minimum spanning tree of G? We focus on the case where the number of distinct edge weights is bounded. Using a bucketed data structure to organize the edge weights, we present an efficient algorithm for the MSTV problem, which runs in $O(|E| + |V| \cdot K)$ time, where K is the number of distinct edge weights. When K is a fixed constant, this algorithm runs in linear time. We also profile our MSTV algorithm with the current fastest known MSTV implementation. Our results demonstrate the superiority of our algorithm when $K \leq 24$.

The UNCCD problem is defined as follows: Given an undirected graph $G = (V, E)$ with arbitrarily weighted edges, does G contain a negative cost cycle? We discuss two polynomial time algorithms for solving the UNCCD problem: the b-matching approach and the T-join approach. We obtain new results for the case where the edge costs are integers in the range $\{-K \cdot \cdot K\}$, where K is a positive constant. We also provide the first extensive empirical study that profiles the discussed UNCCD algorithms for various graph types, sizes, and experiments.

The NCG problem is defined as follows: Given a directed graph $G = (V, E)$ with arbitrarily weighted edges, find the length, or number of edges, of the negative cost cycle having the least number of edges. We discuss three strongly polynomial NCG algorithms. The first NCG algorithm is known as the matrix multiplication approach in the literature. We present two new NCG algorithms that are asymptotically and empirically superior to the matrix multiplication approach for sparse graphs. We also provide a parallel implementation of the matrix multiplication approach that runs in polylogarithmic parallel time using a polynomial number of processors. We include an implementation profile to demonstrate the efficiency of the parallel implementation as we increase the graph size and number of processors. We also present an NCG algorithm for planar graphs that is asymptotically faster than the fastest topology-oblivious algorithm when restricted to planar graphs.

Acknowledgements

First and foremost, I thank my advisor, Professor K. Subramani, for introducing me to all the problems stated in this thesis and providing insight and feedback as I worked on these problems. Without his encouragement and support, I would not be where I am today. I appreciate the assistance from Dr. Pavlos Eirinakis for his involvement with implementing the UNCCD algorithms. I also thank Dr. Kamesh Madduri for his assistance with troubleshooting the NCG implementations, especially the parallel implementation. I am also indebted to Torben Hagerup and Valerie King for their discussions and feedback concerning our MSTV algorithm. Finally, I extend my gratitude towards my parents, Daniel and Pamela Williamson, and my brother, Aaron Williamson, for their love, support, and encouragement.

I also want to thank several agencies for supporting my research. First, I thank the West Virginia NASA Space Grant Consortium for its financial support through a NASA Grant and Cooperative Agreement Number NNX10AK62H. I also thank the National Science Foundation for its assistance with supplying the hardware required for our empirical study for the parallel NCG implementation through TeraGrid resources provided by NCSA under grant number TG-CCR100036. I also extend my gratitude towards the Lane Department of Computer Science and Electrical Engineering for financially supporting me through graduate teaching assistantships. Finally, I thank the National Science Foundation and Air Force of Scientific Research for its financial support through awards Awards CNS-0849735, CCF-0827397, and FA9550-12-1-0199.

Contents

List of Figures

List of Tables

List of Algorithms

Chapter 1

Statement of Contributions

In this thesis, we design and analyze algorithms for solving three different problems in network optimization. Specifically, we are interested in the minimum spanning tree verification (MSTV) problem, the undirected negative cost cycle detection (UNCCD) problem, and the negative cost girth (NCG) problem. These problems find applications in various domains including program verification, proof theory, real-time scheduling, social networking, certification, and operations research. We detail our contributions below.

1.1 The MSTV Problem

A minimum spanning tree (MST) is a spanning tree T of an undirected graph $G = (V, E)$, whose total weight is the minimum among all spanning trees of G. The problem of determining such an MST is known as the MST construction problem. Algorithms for constructing the MST can be found in [1, 2, 3, 4, 5]. A history of the MST construction problem is available in [6].

A related problem is the MSTV problem, which is defined as follows:

> *Given an undirected graph $G = (V, E)$, and a spanning tree T, determine whether T is a minimum spanning tree (MST) of G.*

One approach for solving the MSTV problem is as follows:

1. Use an efficient MST construction algorithm to construct an MST T.

2. Compare the total weight of T with the input spanning tree.

The running time of this approach depends on the running time of the MST construction algorithm. While this approach correctly verifies a tree, there exist MSTV algorithms that are both simple and independent of any MST construction algorithm. The current fastest algorithms for solving the MSTV problem are described in [7] and [8]. We focus on the case when the number of distinct edge weights is bounded by a fixed constant.

Suppose we are given a graph with n vertices, m edges, and K distinct edge weights. We are also given a spanning tree T. If T is not a spanning tree, then it is clear that T cannot be an MST. So, without loss of generality, we assume that T is spanning. Our MSTV algorithm, which we call DFS-Verify, verifies whether the spanning tree is an MST in $O(m + n \cdot K)$ time. Note that when K is constant, our algorithm runs in *linear time*.

We also profile the DFS-Verify algorithm against one of the fastest known MSTV algorithms in the literature [8]. Our experiments indicate the superiority of our algorithm when the number of distinct edge weights is at most 24.

1.2 The UNCCD Problem

The UNCCD problem is defined as follows:

> *Given an undirected graph $G = (V, E)$ with arbitrarily weighted edges, determine whether a negative cost cycle exists in G.*

For directed graphs, the problem of detecting a negative cost cycle (NCCD) has been widely studied [9, 10, 11, 12, 13, 14, 15, 16]. However, unlike directed graphs, detecting a negative cost cycle for undirected graphs is significantly more difficult. This is because we cannot transform an undirected graph into a directed graph by replacing each undirected edge with two directed edges going in opposite directions. In this case, if an undirected edge has a negative cost, then the two directed edges also have negative costs. This results in a negative cost cycle in the directed graph that did not exist in the undirected graph.

As per the literature, there are two known approaches for the UNCCD problem. The b-matching approach transforms G into a new graph G' and detects the presence of a negative cost cycle in G by finding a minimum weight perfect matching (MWPM) in G' [17, 18]. Given a graph with n vertices and m edges, the b-matching approach runs in $O((m + n)^3)$ time. The T-join

approach uses efficient all pairs shortest paths (APSP) and MWPM algorithms as subroutines to detect a negative cost cycle by utilizing T-joins [19, 20]. This approach runs in $O(n^3)$ time. We provide a detailed, formal presentation of the b-matching and T-join approaches, and we show that the b-matching approach runs in $O((m+n)^2 \cdot \log(m+n))$ time in the general case.

We improve the UNCCD algorithms when the edge costs are integers in the range $\{-K \cdots K\}$, where K is a positive integer constant. When $K = O(1)$, we show that the b-matching approach solves the UNCCD problem in $O((n+m)^{1.5} \cdot (\log(n+m))^{1.5} \cdot \sqrt{\alpha(n+m, n+m)})$ time, while the T-join approach solves the UNCCD problem in $O(n^{2.5} \cdot (\log n)^{1.5} \cdot \sqrt{\alpha(n^2, n)})$ time, where $\alpha(x, y)$ represents the inverse Ackermann function [21, 22]. Both algorithms improve upon the previously known time bounds for the case of fixed integer edge costs [23].

We also perform the first extensive empirical study of the algorithms for the UNCCD problem. In this study, we examine implementations for both the b-matching and T-join approaches for various graph families and sizes.

1.3 The NCG Problem

The girth of an unweighted graph $G = (V, E)$ is defined as the length (i.e., number of edges) of the shortest cycle. The negative cost girth is the length (i.e., number of edges) of the negative cost cycle with the fewest number of edges. In the NCG problem, we are interested in the negative cost girth. We define the NCG problem as follows:

> *Given a network (or directed graph) $G = (V, E)$ with arbitrarily weighted edges, find the negative cost girth of the network.*

In networks with arbitrary edge costs, the first polynomial time algorithm for this problem, known as the matrix multiplication (MM) approach, was proposed in [15]. Given a network with n vertices, m edges, and NCG k, this algorithm runs in $O(n^3 \cdot \log k)$ time. We present two new strongly polynomial NCG algorithms [24]. The first algorithm, which we call Edge-Progress (EP), runs in $O(n^2 \cdot k + m \cdot n \cdot k)$ time. The second algorithm, which we call Edge-Relax (ER), runs in $O(m \cdot n \cdot k)$ time. We profile the above NCG algorithms for various network types and sizes. Our results reinforce the asymptotic analysis and show that the new NCG algorithms are superior to the MM approach for sparse networks.

We extend the results of [15] and present a work-efficient parallel implementation of the MM approach that runs in $O(\log k \cdot \log n)$ parallel time using $O(n^3)$ processors. We include a detailed implementation profile that studies the efficiency of the parallel implementation as we increase the network size and the number of processors.

Finally, we present a new NCG algorithm for planar networks. The current algorithms for solving the NCG problem are topology-oblivious, which means the algorithms do not consider the topology of the graph. Our NCG algorithm in planar networks exploits the properties of planarity and runs in $O(n^{1.5} \cdot k)$ time. This is a substantial improvement over all previously known topology-oblivious NCG algorithms when restricted to planar networks. We also extend our algorithm to general networks that have a separator. In this case, the NCG algorithm runs in $O(n^{a+b} \cdot k + n^d \cdot \log n)$ time, where n^a is the size of the separator, n^b is the number of edges, and we can find the separator in $O(n^d)$ time.

1.4 Overview

The thesis is organized as follows:

Part I discusses the MSTV problem. Chapter 2 formally introduces the MSTV problem. Chapter 3 describes an algorithm for constructing the MST, which we call the Edge-Bucket algorithm. This is done by modifying the algorithm in [25]. We present the MSTV algorithm, which we call the DFS-Verify algorithm, in Chapter 4 and show that it runs in linear time when the number of distinct edge weights is a fixed constant. This chapter also provides an empirical study that profiles the DFS-Verify algorithm against one of the fastest known MSTV algorithms [8].

Part II discusses the UNCCD problem. We formally introduce the UNCCD problem in Chapter 5. Chapter 6 reviews the b-matching and T-join approaches, and we improve the asymptotic analysis for the b-matching approach. We also present improved UNCCD algorithms for both approaches when we restrict the edge costs to be integers in the range $\{-K \cdot \cdot K\}$, where K is a fixed positive constant. We provide a detailed implementation profile in Chapter 7 that studies both UNCCD algorithms for various graph types, sizes, and experiments.

Part III discusses the NCG problem. Chapter 8 introduces the NCG problem. Chapter 9 presents the EP and ER approaches for the NCG problem. We also provide an empirical study

that demonstrates the superiority of both algorithms when compared to the MM approach, for sparse networks. We describe a parallel implementation of the matrix multiplication approach in Chapter 10 and include an implementation profile that analyzes the efficiency of the parallel implementation as we increase the size of the network and the number of processors. Chapter 11 presents a new, efficient NCG algorithm in planar networks that is superior to the fastest topology-oblivious NCG algorithm, when restricted to planar networks.

Chapter 12 summarizes our conclusions and discusses avenues for future work.

Part I

The Minimum Spanning Tree Verification Problem

Chapter 2

Introduction

A spanning tree of a graph is defined as an acyclic subset such that all vertices are connected. A spanning tree is a minimum spanning tree (MST) if the total weight of the spanning is the minimum among all possible spanning trees of the graph. The problem of determining the MST is called the *MST construction problem*. Some of the more well-known approaches for constructing an MST are found in [2, 26]. Figure 2.1 provides an example of an MST.

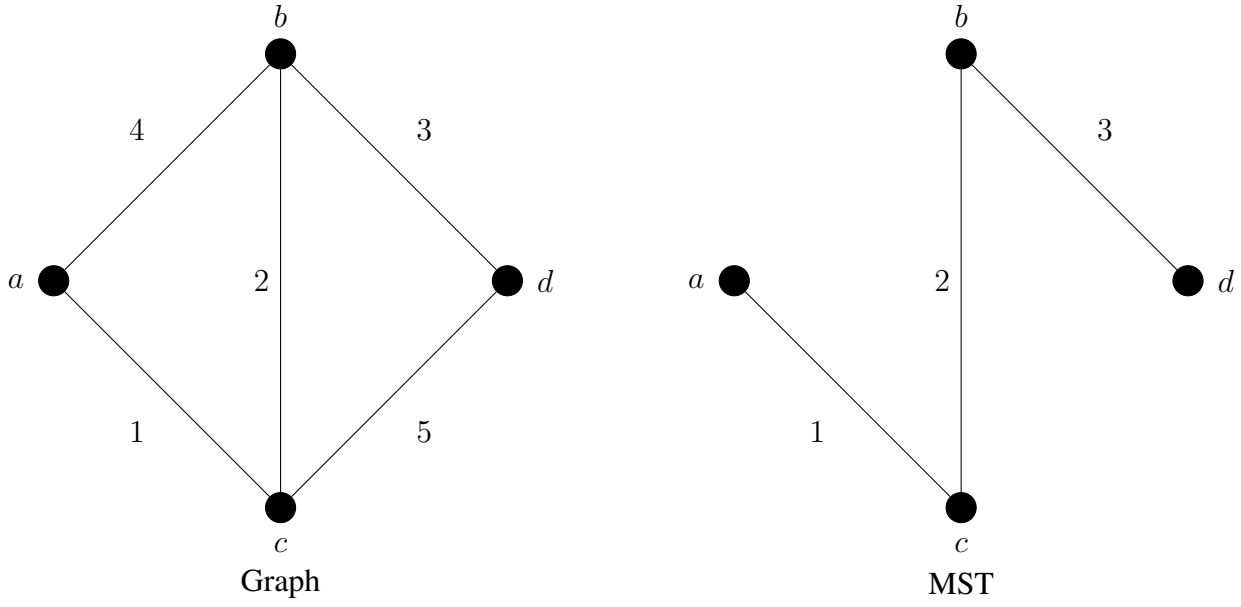


Figure 2.1: An MST of a connected graph.

A related problem is the *minimum spanning tree verification problem* (MSTV), which is concerned with determining whether a given spanning tree is an MST of the graph. This problem

finds applications in several domains including program verification and certification. The MSTV problem was introduced in [27], wherein the first near linear time algorithm was proposed. [3, 28] describe several approaches for the MSTV problem. The current fastest algorithms for solving the MSTV problem are described in [7, 8]. We focus on the case where the number of distinct edge weights, denoted as K, is bounded by a fixed constant.

The next two chapters present algorithms related to the MST. We first present an MST construction algorithm, known as the Edge-Bucket algorithm, that partitions the edges into distinct linked lists. This approach was used in [25] for solving the single source shortest path (SSSP) problem, and we discuss the SSSP algorithm in Appendix A. We then describe the MSTV algorithm, which we call the DFS-Verify algorithm. Both algorithms run in $O(m + n \cdot K)$ time, where n is the number of vertices, m is the number of edges, and K is the number of distinct edge weights. The key conclusion is that when K is a fixed constant, both algorithms run in linear time.

2.1 Preliminaries and Notation

We are given an undirected graph $G = (V, E, c)$, where V is the set of n vertices, E is the set of m edges, and $\mathbf{c} : E \to \mathbb{Z}^+$ is a cost function. For each edge $e_{ij} \in E$, we let c_{ij} be the weight of that edge. We assume $c_{ij} \geq 0$ is a real number. We represent G as an adjacency list Adj, where for each vertex $v \in V$, $Adj(v)$ is the set of outgoing edges from v in G.

We are also given a spanning tree $T = (V_T, E_T)$, where V_T is the set of n vertices in T, E_T is the set of $n - 1 \leq m$ edges in T. We also represent T as an adjacency list. Note that $V_T = V$ and $E_T \subseteq E$.

Finally, we let K be the number of distinct edge weights of the graph, where $1 \leq K \leq m$. $L = \{l_1, \ldots, l_K\}$ is the set of distinct edge weights, where $|L| = K$. We assume L is sorted in increasing order. Otherwise, we can sort L in $O(K \log K)$ time, which is $O(1)$ time if K is a constant. We provide an example of a graph with two distinct edge weights in Figure 2.2, where $L = \{3, 5\}$.

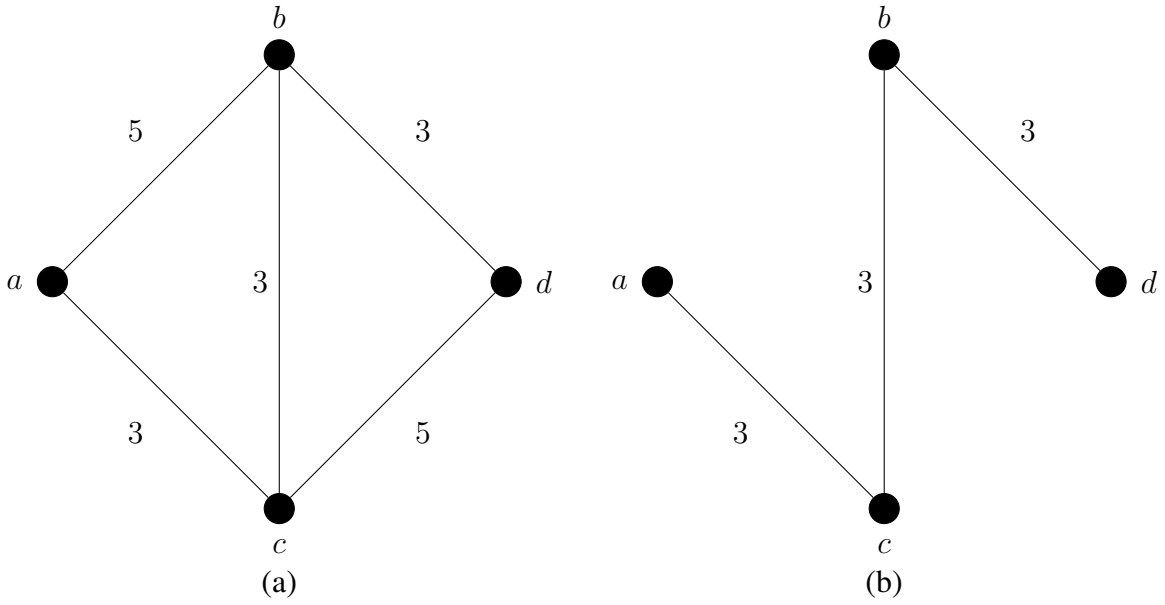


Figure 2.2: Example of a graph with two distinct edge weights. (a) is the graph G, and (b) is the MST of G.

Chapter 3

The MST Construction Algorithm

In this chapter, we present an algorithm for the MST construction problem in an undirected graph, where the number of distinct edge weights is bounded by a fixed constant. Recall that an MST is a spanning tree of a graph G, whose total weight is the minimum among all spanning trees of G. Suppose we are given a graph with n vertices, m edges, and K distinct edge weights. Our MST construction algorithm, which we call the Edge-Bucket algorithm, runs in $O(m + n \cdot K)$ time. Note that when K is a fixed constant, this algorithm runs in linear time.

3.1 Related Work

The problem of constructing an MST was first studied in [1]. Borůvka's algorithm contracted the lightest edges (i.e., edges with the smallest weight) from each vertex, and the contracted edges form the MST. This algorithm ran in $O(m \cdot \log n)$ time. Jarník [29] described the first greedy approach for the MST problem by selecting the lightest edges connecting to a "source" vertex. This algorithm is also known as Prim's algorithm [26], and the running time is dependent on the data structure used. Currently, the fastest known running time is $O(m + n \cdot \log n)$ time by using the Fibonacci heap data structure [30]. Kruskal [2] provided another greedy algorithm for the MST problem. This algorithm selected the edges with the smallest weight without creating any cycles and ran in $O(m \cdot \log n)$ time.

[31] and [32] independently developed the next significant MST construction algorithm that ran in $O(m \cdot \log \log n)$ time. With the assistance of Fibonacci heaps [30], Fredman and Tarjan

provided an algorithm that ran in $O(m \cdot \beta(m,n))$ time, where $\beta(m,n) = \min\{i : \log^i n \leq \frac{m}{n}\}$. Gabow et al. [33] improved the previous result with an algorithm that ran in $O(m \cdot \log \beta(m,n))$ time.

Recent MST algorithms involve non-greedy approaches by including the inverse Ackerman function $\alpha(m,n)$ [21]. The first of these algorithms was developed by Chazelle [34], whose algorithm ran in $O(m \cdot \alpha(m,n) \cdot \log \alpha(m,n))$ time. This was later improved by both Pettie [35] and Chazelle [5] independently. The idea with these algorithms was to compute suboptimal disjoint sets in a non-greedy fashion and improve these sets until an optimal solution is obtained. Both algorithms ran in $O(m \cdot \alpha(m,n))$ time.

Several advancements have been made towards the MST problem for special cases. [4] has a randomized MST construction algorithm that runs in linear expected time. This algorithm combines concepts from [1], random sampling, and a "black box" minimum spanning tree verification algorithm [7] to construct the MST. Fredman and Willard [36] describe an $O(m+n)$ time MST algorithm if the edge weights are b-bit integers, and we use a more powerful model of computation that involves manipulating the bits of the stored edge weights. For a pointer machine, Pettie and Ramachandran [37] provide an MST construction algorithm that runs in $O(\mathcal{T}^*(m,n))$ time, where $\mathcal{T}^*(m,n)$ is the number of edge weight comparisons needed to determine the MST. In most cases, the algorithm runs in linear time because it uses precomputed MST decision trees.

A detailed history of the above MST construction algorithms and additional advancements for the MST problem are available in in [6].

3.2 The Edge-Bucket Algorithm

We now discuss the MST construction algorithm that runs in $O(m + n \cdot K)$ time. This is a variation of Prim's $O(m + n \cdot \log n)$ time algorithm [26]. It is known that Prim's algorithm and Dijkstra's algorithm [38] are closely related [22]. Thus, we can modify the single source shortest path (SSSP) algorithm to compute the MST. Appendix A provides an $O(m + n \cdot K)$ time SSSP algorithm, where K is the number of distinct edge weights.

We let S be the set of permanently labeled vertices and $Temp = V - S$ be the set of temporary labeled vertices. Let $L = \{l_1, \ldots, l_K\}$ be the set of distinct edge weights. For each distinct edge

weight $l_t \in L = \{l_1, \ldots, l_K\}$, we have a linked list $E_t(S) = \{e_{ij} \in E : \ i \in S, c_{ij} = l_t\}$. This means the edges are partitioned into exactly one of these "buckets." Each edge e_{ij} in a bucket is sorted in the order j is added to S.

We let the pointer $CurrentEdge(t)$ be the first edge e_{ij} of $E_t(S)$, where $j \in Temp$. If $E_t(S)$ has such an edge, then we let $f(t) = l_t$. If $E_t(S)$ does not have such an edge, then $CurrentEdge(t) = \emptyset$. We can find the vertex with the current lightest edge by finding $argmin\{f(t) : \ 1 \leq t \leq K\}$, which runs in $O(K)$ time.

We use a subroutine UPDATE(t) to change $CurrentEdge(t)$ such that it either points to the first edge in $E_t(S)$, where the endpoint is in T, or sets $CurrentEdge(t)$ to $\emptyset$. If $CurrentEdge(t)$ points to an edge e_{ij}, then we set $f(t) = l_t$. Otherwise, we set $f(t) = \infty$. We denote $CurrentEdge(t).next$ as the operation that moves the pointer $CurrentEdge(t)$ to point to the next edge in $E_t(S)$.

The algorithm runs as follows: We start with the source vertex $s \in V$. Add all vertices in V into V_T. We then find $r = argmin\{f(t) : \ 1 \leq t \leq K\}$ and let edge e_{ij} be $CurrentEdge(r)$. We add e_{ij} to E_T and move j from $Temp$ to S. For each outgoing edge e_{jk}, we add e_{jk} to the end of $E_t(S)$, where $l_t = c_{jk}$. We then change $CurrentEdge(t)$ if it was initially null. For each distinct edge weight k, we run UPDATE(k), which updates the $CurrentEdge$ pointer for each link list if needed.

The procedure is shown in Algorithms 3.1, 3.2, and 3.3.

3.2.1 Resource Analysis

The algorithm runs the same procedure as [25] with a few modifications. The initialization process takes $O(n)$ time since we add all the vertices adjacent to s into their respective linked lists. For Algorithm 3.2, adding each of the vertices into V_T takes $O(n)$ time. Finding r takes $O(K)$ time, which is one of the major bottlenecks of the algorithm. Since we have $O(n)$ iterations, the total running time is $O(n \cdot K)$. The UPDATE(t) procedure remains unchanged except for a single notation change. Therefore, the running time of the procedure is $O(m + n \cdot K)$, which is the total running time of the algorithm. Note that when K is a small, fixed constant, this is a linear time algorithm.

Function INITIALIZE()
1: $S := \{s\}$; $Temp := V - \{s\}$
2: $pred(s) := \emptyset$
3: **for** (each vertex $v \in Temp$) **do**
4: $pred(v) = \emptyset$
5: **end for**
6: $T := \emptyset$
7: **for** ($t = 1$ **to** K) **do**
8: $E_t(S) := \emptyset$
9: $CurrentEdge(t) := NIL$
10: **end for**
11: **for** (each edge e_{sj}) **do**
12: Add e_{sj} to the end of the list $E_t(S)$, where $l_t = c_{sj}$
13: **if** ($CurrentEdge(t) = NIL$) **then**
14: $CurrentEdge(t) := e_{sj}$
15: **end if**
16: **end for**
17: **for** ($t = 1$ to K) **do**
18: UPDATE(t)
19: **end for**

Algorithm 3.1: MST Algorithm: Initialization

For the space requirement, we store G and T as adjacency lists of size $O(m + n)$. For E_t, where $1 \leq t \leq K$, recall that each edge is stored into one of the K linked lists. This means the total number of edges stored among all K lists is $O(m)$. Therefore, the total space required is $O(m + n) + O(m) = O(m + n)$.

3.2.2 Correctness

We now prove that the algorithm correctly forms a minimum spanning tree. Note that our algorithm is similar to Prim's algorithm. The difference is how the data is structured. At each iteration, instead of scanning all adjacent edges, we scan at most K adjacent edges to find the lightest edge. Also, each adjacent edge is placed in one of the K linked lists. This means scanning each list allows the algorithm to find the edge with the lightest weight.

After finding this edge e_{ij}, we add it to T, scan the edges e_{jk} adjacent to j, and place each edge e_{jk} in the appropriate linked list. This method is similar to noting all the adjacent edges outgoing from j from Prim's algorithm. The UPDATE(t) procedure simply changes the pointers of

Function EDGE-BUCKET()

1: INITIALIZE()
2: **for** (each vertex $v \in V$) **do**
3: $V_T := V_T \cup \{v\}$
4: **end for**
5: **while** ($Temp \neq \emptyset$) **do**
6: let $r = argmin\ \{f(t) : 1 \leq t \leq K\}$
7: let $e_{ij} = CurrentEdge(r)$
8: $E_T := E_T \cup \{e_{ij}\}$
9: $pred(j) := i$
10: $S = S \cup \{j\}; Temp := Temp - \{j\}$
11: **for** (each edge $e_{jk} \in Adj(j)$) **do**
12: Add the edge to the end of the list $E_t(S)$, where $l_t = c_{jk}$
13: **if** ($CurrentEdge(t) = NIL$) **then**
14: $CurrentEdge(t) := e_{jk}$
15: **end if**
16: **end for**
17: **for** ($t = 1$ to K) **do**
18: UPDATE(t)
19: **end for**
20: **end while**
21: **return** $T = (V_T, E_T)$

Algorithm 3.2: MST Algorithm: EDGE-BUCKET

Function UPDATE(t)

1: Let $e_{ij} = CurrentEdge(t)$
2: **if** ($j \in Temp$) **then**
3: $f(t) = c_{ij}$
4: **return**
5: **end if**
6: **while** (($j \notin Temp$) **and** ($CurrentEdge(t).next \neq NIL$)) **do**
7: Let $e_{ij} = CurrentEdge(t).next$
8: $CurrentEdge(t) = e_{ij}$
9: **end while**
10: **if** ($j \in Temp$) **then**
11: $f(t) = c_{ij}$
12: **else**
13: Set $CurrentEdge(t)$ to $\emptyset$
14: $f(t) = \infty$
15: **end if**

Algorithm 3.3: MST Algorithm: Update

each linked list.

Even with these modifications, we still select the edge with the lightest weight among the edges adjacent to the vertices in the set S and then scan the edges adjacent to the selected edge. Therefore, our algorithm operates identical to Prim's algorithm. Since Prim's algorithm is correct, Algorithm 3.2 correctly computes the MST.

3.3 Example of the Algorithm

We now provide an example of the Edge-Bucket algorithm on a graph with $n = 4$ vertices, $K = 2$, and $L = \{1, 3\}$. Figure 3.1 is the initial graph G. In line 1 of Algorithm 3.2, we first initialize $S = \{s\}$ and $Temp = \{v_1, v_2, v_3\}$. After scanning all edges adjacent from s, we have lists $E_1(S) = \{e_{sb}\}$ where $CurrentEdge(1) = e_{sb}$ and $E_2(S) = \{e_{sa}\}$ where $CurrentEdge(2) = e_{sa}$. We run the UPDATE(t) procedure for each list $E_t(S)$ and set $f(1) = 1$ and $f(2) = 3$. We then add all vertices into V_T.

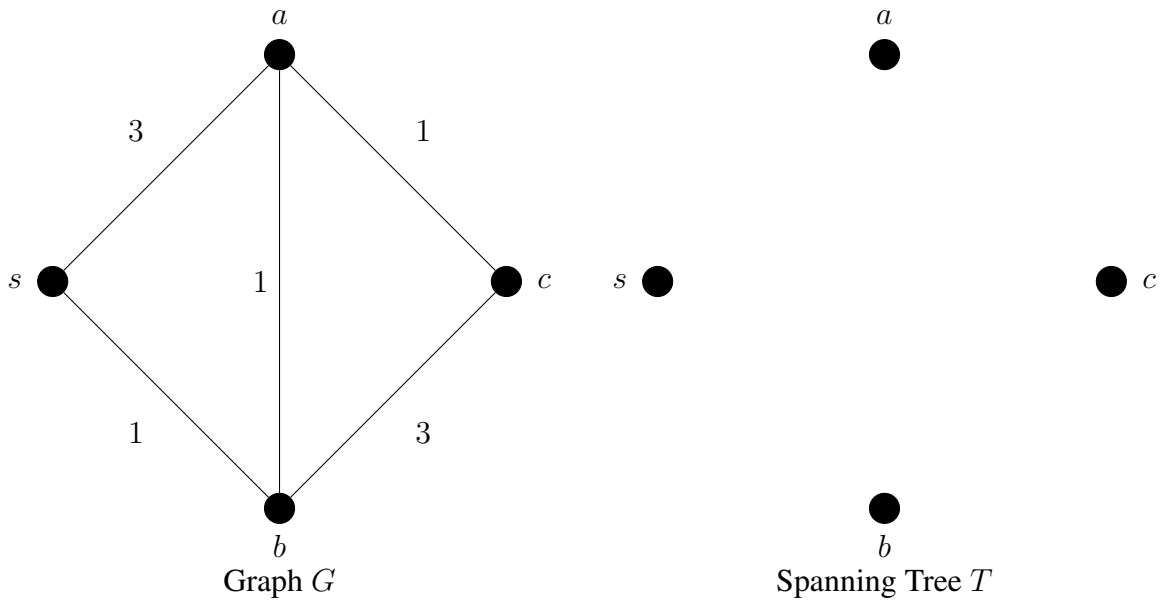


Figure 3.1: Example of the Edge-Bucket algorithm. Initial graph G and spanning tree T.

From line 6, we have $r = 1$, which means we set add e_{ab} to T and add b to S giving us $S = \{s, b\}$ and $Temp = \{a, c\}$. Lines 11 to 16 scan the edges adjacent to b and add them to $E_1(S)$ and $E_2(S)$ accordingly. This gives the lists $E_1(S) = \{e_{sb}, e_{ba}\}$ where $CurrentEdge(1) = e_{sb}$

and $E_2(S) = \{e_{sa}, e_{bc}\}$ where $CurrentEdge(2) = e_{sa}$. This is shown in Figure 3.2.

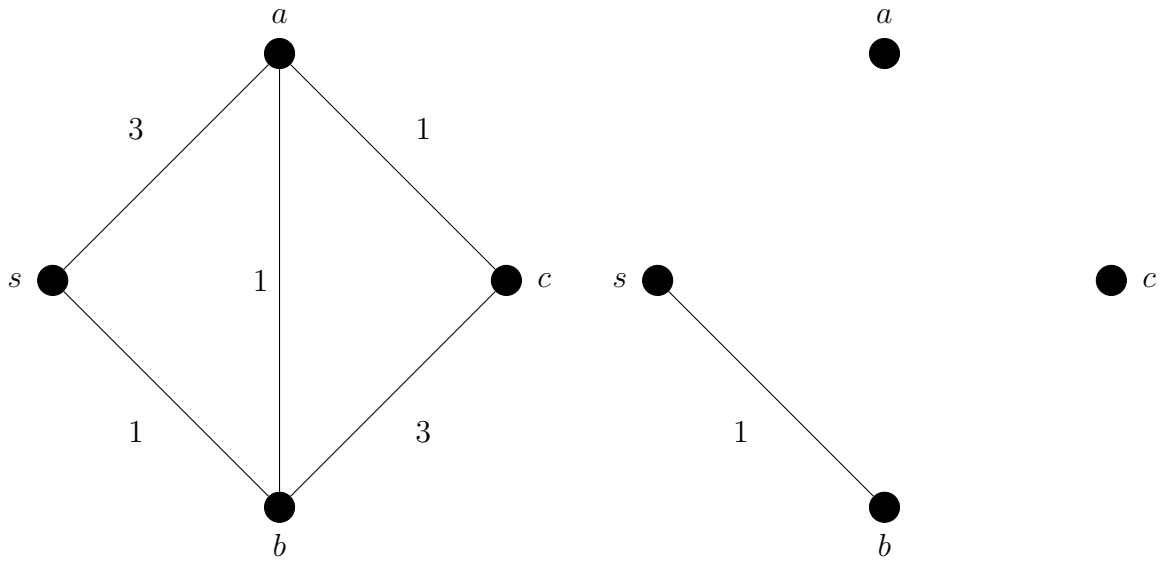


Figure 3.2: Example of the Edge-Bucket algorithm. $E_1(S) = \{e_{sb}, e_{ba}\}$ with $CurrentEdge(1) = e_{ba}$, and $E_2(S) = \{e_{sa}, e_{bc}\}$ with $CurrentEdge(2) = e_{sa}$.

We now run the UPDATE(t) procedure for each list $E_t(S)$. For $CurrentEdge(1)$, since $b \in S$, we update $CurrentEdge(1)$ to e_{ba} in lines 6 to 9 and let $f(1) = 1$ in lines 10 to 11. For $CurrentEdge(2)$, since $a \in Temp$, we set $f(2) = 3$, leave $CurrentEdge(2)$ alone, and end the procedure.

We run the next iteration since $Temp \neq \emptyset$. We find that $r = 1$, which means we add e_{ba} to E_T and add a to S giving us $S = \{s, a, b\}$ and $Temp = \{c\}$. We scan the edges adjacent to a and add them to our lists giving us $E_1(S) = \{e_{sb}, e_{ba}, e_{ac}\}$ where $CurrentEdge(1) = e_{ba}$ and $E_2(S) = \{e_{sa}, e_{bc}\}$ where $CurrentEdge(2) = e_{sa}$. When we run the UPDATE(t) procedure, we change $CurrentEdge(1)$ to e_{ac}, set $f(1) = 1$, change $CurrentEdge(2)$ to e_{bc}, and set $f(2) = 3$. We see this in Figure 3.3.

We run one more iteration since $Temp = \emptyset$. We have $r = 1$, which means we add e_{ac} to E_T and add c to S giving us $S = \{s, a, b, c\}$ and $Temp = \emptyset$. We scan the edges adjacent to c and find no other edges adjacent to c. This gives us $E_1(S) = \{e_{sb}, e_{ba}, e_{ac}\}$ where $CurrentEdge(1) = e_{ac}$ and $E_2(S) = \{e_{sa}, e_{bc}\}$ where $CurrentEdge(2) = e_{bc}$. When we run the UPDATE(t) procedure, we change both $CurrentEdge(1)$ and $CurrentEdge(2)$ to $\emptyset$ and set both $f(1)$ and $f(2)$ to ∞

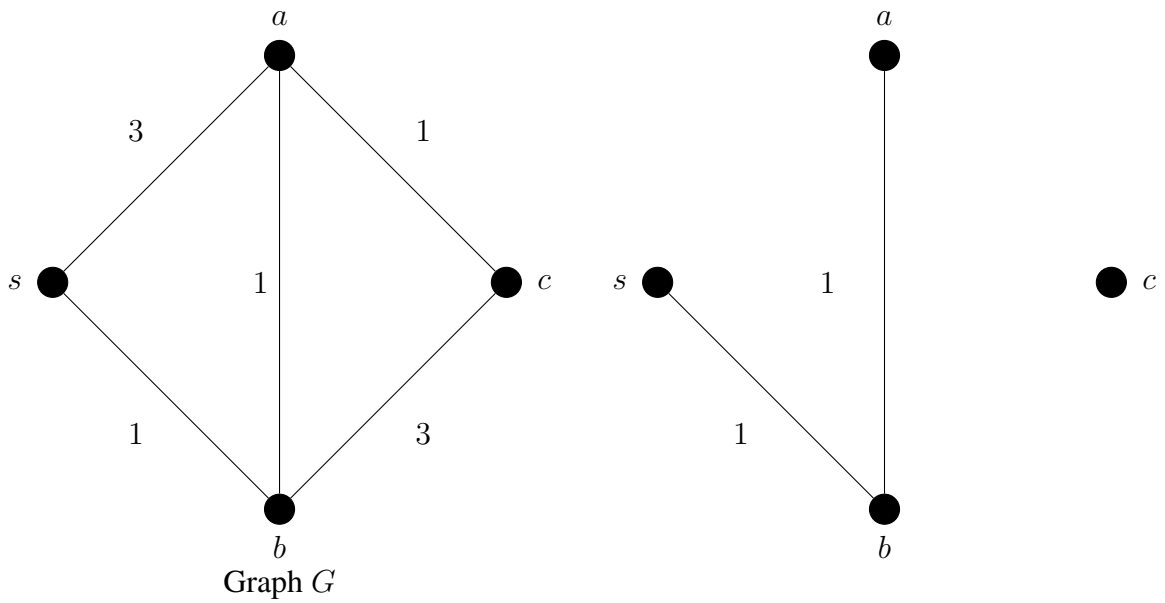


Figure 3.3: Example of the Edge-Bucket algorithm. $E_1(S) = \{e_{sb}, e_{ba}, e_{ac}\}$ with $CurrentEdge(1) = e_{ac}$, and $E_2(S) = \{e_{sa}, e_{bc}\}$ with $CurrentEdge(2) = e_{bc}$.

since $c \notin Temp$. The resulting graph and spanning tree are shown in Figure 3.4.

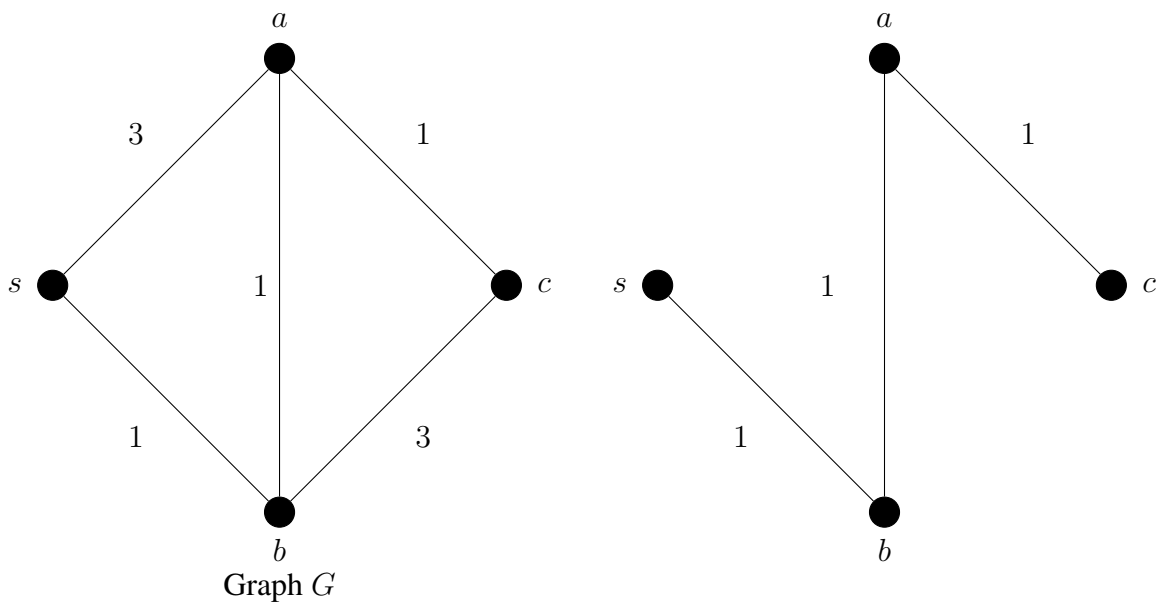


Figure 3.4: Example of the Edge-Bucket algorithm. $E_1(S) = \{e_{sb}, e_{ba}, e_{ac}\}$ with $CurrentEdge(1) = e_{ac}$, and $E_2(S) = \{e_{sa}, e_{bc}\}$ with $CurrentEdge(2) = e_{bc}$.

Chapter 4

The MSTV Algorithm

In this chapter, we present an algorithm for the MSTV problem in an undirected graph. We also provide an empirical study that analyzes the performance of our algorithm. Recall that an MST is a spanning tree of a graph G, whose total weight is the minimum among all spanning trees of G.

The MSTV problem is defined as follows:

> *Given an undirected graph $G = (V, E)$ and a spanning tree T, determine whether T is an MST of G.*

This problem finds applications in several domains including program verification and certification. Although we can verify a spanning tree T by comparing the total weight of T with the resulting MST of any MST construction algorithm, there exist MSTV algorithms that are both simple and independent of any MST construction algorithm. The MSTV problem was introduced in [27], wherein the first near linear time algorithm was proposed. [3] and [28] describe several approaches for the MSTV problem. The current fastest algorithms for solving the MSTV problem are described in [7] and [8]. In this paper, our focus is when the number of distinct edge weights is a fixed constant.

Suppose we are given a graph with n vertices, m edges, and K distinct edge weights. We are also given a spanning tree T. If T is not a spanning tree, then clearly T cannot be an MST. The DFS-Verify algorithm verifies whether the spanning tree is an MST in $O(m + n \cdot K)$ time. Note that when K is constant, our algorithm runs in *linear time*.

4.1 Related Work

The problem of verifying an MST is first studied in [27]. Tarjan's algorithm uses path compression to verify an input spanning tree. The running time of Tarjan's algorithm is almost linear, specifically $O(m \cdot \alpha(m, n))$ time, where $\alpha(m, n)$ is the inverse Ackermann function [21]. A revised and improved version of Tarjan's algorithm is provided in [39]. Komlós [3] shows that we can verify a minimum spanning tree using $O(m + n)$ binary comparisons. However, determining which comparisons to use takes non-linear time. The first linear time implementation of the MSTV problem is given in [28]. Dixon, Rauch, and Tarjan divides the spanning tree into small microtrees of size $O(\log \log n)$ and a single large subtree, computes the decision tree using [3] for each possible microtree, applies the decision trees to process the microtrees, and runs the algorithm in [39] for the remaining large subtree.

A simpler approach for implementing Komlós' algorithm is provided in [7] and runs in $O(m + n)$ time. King's algorithm first creates a full branching tree consisting of at most $2 \cdot n$ vertices and $2 \cdot n$ edges using Borůvka's algorithm [1]. The algorithm then determines the lowest common ancestor (LCA) for each pair of leaf vertices in the branching tree using a data structure. For each non-tree edge e_{ij}, the algorithm computes the heaviest edge in the path from i to z and j to z in the branching tree using a lookup table, where z is the LCA of i and j. King shows that the weight of the heaviest edge in the path between vertices x and y in the branching tree is weight of the heaviest edge in the path between x and y in the spanning tree. The linear time bound is achieved by performing several computations in a single computer word.

Hagerup [8] describes a simpler linear time MSTV algorithm by developing a linear time algorithm for the special tree path maxima problem, which is defined as follows:

> *Given a full branching tree T and a list of pairs (u, v) of vertices in T such that u is a proper ancestor of v, find for each pair (u, v) on the list a heaviest edge on the path in T between u and v.*

Hagerup's algorithm starts with a full branching tree and a list of paired vertices, where for each element (u, v) in the list, u is a proper ancestor of v. The algorithm then constructs a lookup table that is used for finding a heaviest edge on the path of the branching tree between each pair of vertices. A linear time tree path maxima algorithm implies a linear time MSTV algorithm. For

each non-tree edge e_{ij}, we find the LCA using King's approach from above. We then add the pairs (i, z) and (j, z) in the list, where z is the LCA (and proper ancestor) of i and j. Once the list of heaviest edges has been computed, we check the weight of e_{ij} with the weights of the heaviest edges of pairs (i, z) and (j, z). Note that Hagerup's algorithm is similar to King's approach, in that it is an implementation of Komlós' algorithm for full branching trees. However, the bit operations are more simplified and easier to implement.

All of the above algorithms take advantage of the following observation (also called the ***Red Rule*** in MST literature):

> *A spanning tree T is a minimum spanning tree if and only if the weight of each edge $e_{uv} \notin E_T$ is at least as large as the weight of the heaviest edge in the path from u to v containing only edges in T.*

Several advancements have been made towards the MSTV problem for special cases. [40] explores the distributed setting for the MSTV problem for weighted, undirected graphs. In this environment, every vertex "knows" which of its adjacent edges belongs to the MST. The resulting algorithm runs in $O(\log n \cdot \log W)$ time, where W is the largest edge weight. A parallel version of King's algorithm [7] is discussed in [41] that runs in $O(\log n)$ parallel time with $O(m + n)$ work. [42] shows that for the online MSTV problem, which consists of a sequence of queries of the form, "Is e_{ij} in the MST of $T \bigcup \{e_{ij}\}$?" for a fixed T, there are no linear time solutions. Pettie proves that the lower bound for this solution is $\Omega(m \cdot \alpha(m, n))$, where $\alpha(m, n)$ is the inverse Ackermann function. [43] provides an $O(m + n)$ time algorithm for a pointer machine by computing all of the path maxima. This was further improved in [44].

MSTV algorithms are used to solve other problems as well. [4] has a randomized MST construction algorithm that runs in linear expected time. This algorithm combines concepts from [1], random sampling, and a "black box" MSTV algorithm [7] to construct the MST. Approaches for the MSTV problem can be applied to the sensitivity analysis problem, which is defined as follows:

> *Suppose we are given an undirected, weighted graph G and a minimum spanning tree T. For each edge $e_{uv} \in E$, we want to determine how much c_{uv} can change without affecting the minimality of T.*

Tarjan [45] extends his MSTV algorithm to solve the sensitivity analysis problem in $O(m \cdot \alpha(m, n))$ time, where α is the functional inverse of Ackermann's function. For planar graphs, an $O(m)$ time algorithm for the sensitivity analysis of the MST is proposed in [46].

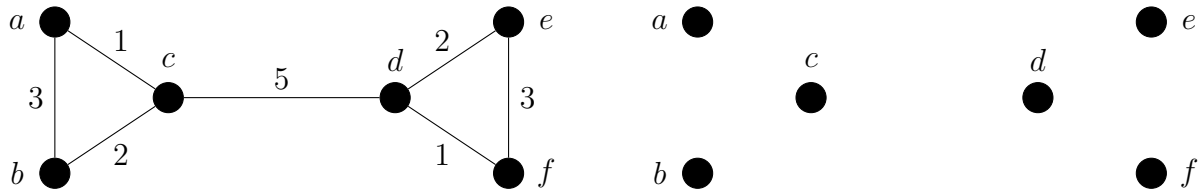
4.2 The DFS-Verify Algorithm

In this section, we determine whether a tree is an MST in $O(m + n \cdot K)$ time, where the graph contains K distinct edge weights. Briefly, our algorithm uses a modified version of Borůvka's algorithm to contract the lightest edges from the spanning tree T. For each edge contracted, we add it to a new graph G^*, where we determine the vertices that correspond to each contracted vertex in T. Using this information, we can check the vertices of the lightest non-tree edges to see if they correspond to the same contracted vertex. Finding an edge e_{ij} where i and j correspond to different contracted vertices implies that T is not a minimum spanning tree. The details of the algorithm are explained below.

4.2.1 Borůvka's Algorithm

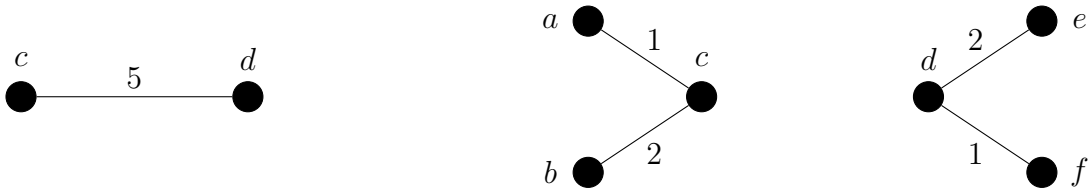
We first describe Borůvka's algorithm [1], which runs in $O(m \cdot \log n)$ time, for constructing the MST. For each vertex $u \in V$, we mark an edge e_{uv} such that $c_{uv} \leq c_{ux}$ for all vertices x that are adjacent to u. For each marked edge e_{uv}, we contract the edge by collapsing u and v into a single vertex, whose adjacent edges are the edges adjacent to both u and v except for e_{uv}. This process is called a Borůvka phase.

We demonstrate this process in Figures 4.1-4.3. Figure 4.1 gives us the graph G and a spanning tree T containing only the vertices. During the first Borůvka phase, a picks e_{ac}, b picks e_{bc}, c picks e_{ac}, d picks e_{df}, e picks e_{de}, and f picks e_{df}. Vertices a and b are contracted into c, and both e and f are contracted into d. The edges picked are added to T which is shown in Figure 4.2. For the ease of exposition, we remove edges e_{ab} and e_{ef} from Figure 4.2 since they are self-loops after contraction. During the second Borůvka phase, c picks e_{cd}, and d picks e_{cd}. These vertices contract into a single vertex, which we denote as c without loss of generality, and e_{cd} is added to T. This is shown in Figure 4.3. Since there are no more edges to contract in G, the algorithm terminates, and T is an MST.



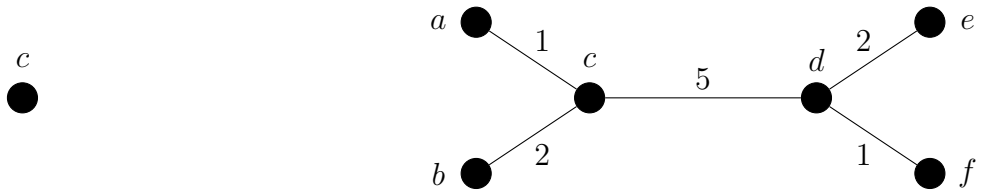
Graph G Tree T

Figure 4.1: Example of Borůvka's algorithm. Graph G and tree T with only the vertices.



Graph G Tree T

Figure 4.2: Example of Borůvka's algorithm. Graph G and tree T after the first Borůvka phase.



Graph G Tree T

Figure 4.3: Example of Borůvka's algorithm. Graph G and tree T after the second Borůvka phase.

From [1], each Borůvka phase runs in $O(m + n)$ time. After each phase, since the number of edges contracted is at least $n/2$, and each contracted edge removes exactly one vertex from G, the resulting graph contains as most $n/2$ vertices. This means we need $O(\log n)$ phases to merge all the vertices. Therefore, Borůvka's algorithm runs in $O((m + n) \cdot \log n) = O(m \cdot \log n)$ time.

4.2.2 The Verification Algorithm

Since T is connected, each pair of vertices i and j must have a unique path using edges in T. This means we can verify the spanning tree by comparing each non-tree edge e_{ij} with the heaviest edge in the corresponding path from i to j in T. This is the same approach used in [3], [7], and [8]. If the weight of the heaviest edge in this path is greater than the weight of e_{ij}, then the spanning tree is not a minimum spanning tree.

We start by partitioning all the non-tree edges into linked lists. For each distinct edge weight $l_r \in L = \{l_1, \ldots, l_K\}$, we have the linked list $E_r = \{e_{ij} \in E \backslash E_T : c_{ij} = l_r\}$. We also create a new graph $G^* = (V^*, E^*)$ that initially consists of the vertices in G. G^* is used to help label the vertices as we contract edges during each Borůvka phase.

We use a modified version of the Borůvka phase for each distinct edge weight. For each phase r, where $r = \{1, \ldots, K - 1\}$, we contract all edges in T whose weight is l_r, rather than contracting the lightest edge for each vertex. When we contract an edge $e_{ij} \in E_T$, we add e_{ij} in E^* and let i be the vertex in V_T that results from contracting edge e_{ij}.

After we contract all the edges of weight l_r, we need to keep track of which vertices in G correspond to each contracted vertex in V_T. We create an array A of n elements. For each element $A[v]$, v is a vertex in G, and $A[v]$ is the corresponding contracted vertex in T. For each vertex $v \in V_T$, we perform a depth-first search (DFS) [47, 22] in G^* with v as the starting vertex. For each vertex u that is visited, we set $A[u] = v$.

We now discuss the verification phase. For each edge $e_{ij} \in E_r$, we compare $A[i]$ with $A[j]$. If $A[i] = A[j]$, then the path from i to j using edges from T consists of edges whose weight is less than or equal to l_t. However, if $A[i] \neq A[j]$, then there exists at least one edge, denoted as e_{uv}, where $c_{uv} > l_r$ and e_{uv} is in the path from i to j using edges from T. If we remove e_{uv} from T and add e_{ij} into T, we get a new tree T' where the total weight of T' is less than the weight of T.

This means T is not a minimum spanning tree, and we can terminate.

If $A[i] = A[j], \forall e_{ij} \in E_r$, we proceed to phase $r+1$, contract all the edges in T whose weight is l_{r+1}, and repeat the DFS and verification processes. Note that in each phase r, we determine whether for each edge $e_{ij} \in E_r$, there is an edge e_{uv} in the path from i to j in T whose weight is greater than l_r. Since there are no edges whose weight is greater than l_K, we need only $K-1$ phases.

The procedure is shown in Algorithms 4.1, 4.2, and 4.3.

Function INITIALIZE()

1: **for** (each edge $e_{ij} \in E - E_T$) **do**
2: Add e_{ij} to the end of the list E_r, where $c_{ij} = l_r$.
3: **end for**
4: Create graph $G^* = (V^*, E^*)$ that contains only the vertices, V, of G.
5: **for** (each $v \in V$) **do**
6: $A[v] := \emptyset$.
7: **end for**

Algorithm 4.1: MSTV Algorithm: Initialization

Function DFS-VERIFY (G, T)

1: INITIALIZE()
2: **for** ($r = 1$ to $K-1$) **do**
3: **for** (each edge $e_{ij} \in E_T$, where $c_{ij} = l_r$) **do**
4: $Adj(i) := Adj(i) \cup Adj(j)$.
5: $V_T := V_T - \{j\}$.
6: $E_T := E_T - \{e_{ij}\}$.
7: Add edge e_{ij} to E^*.
8: **end for**
9: **for** (each vertex $v \in V_T$) **do**
10: MSTV-DFS(v, v).
11: **end for**
12: **for** (each edge $e_{ij} \in E_r$) **do**
13: **if** ($A[i] \neq A[j]$) **then**
14: **return** T is not a Minimum Spanning Tree.
15: **end if**
16: **end for**
17: **end for**

Algorithm 4.2: MSTV Algorithm: DFS-VERIFY

Function MSTV-DFS(u, v)
1: **if** (v is not visited) **then**
2: $A[v] := u$.
3: Mark v as visited.
4: **for** (each outgoing edge $e_{vw} \in E^*$) **do**
5: MSTV-DFS(u, w).
6: **end for**
7: **end if**

Algorithm 4.3: MSTV Algorithm: DFS

4.2.3 Resource Analysis

The initialization process takes $O(m + n)$ time since we add all the edges not in the spanning tree into distinct linked lists, and we initialize all the vertices. Contracting all edges in T of weight l_t takes $O(n)$ time since this is similar to a Borůvka phase, and we contract at most $O(n)$ edges because $|E_T| = n - 1$. For Algorithm 4.3, each vertex is scanned exactly once for all iterations of a phase. This is because each vertex $v \in V$ corresponds to exactly one vertex in V_T. Also, since we have at most $n - 1$ edges in G^*, we scan at most $O(n)$ edges for a single iteration of the DFS procedure. Therefore, the total running time for all iterations of Algorithm 4.3 for a phase is $O(n)$. Since we have $O(K)$ phases, the contraction and DFS phases take a total time of $O(n \cdot K)$.

We now analyze the verification phase. Since each edge e_{ij} belongs to exactly one linked list, when e_{ij} is checked, we never check that edge again. In other words, each edge is checked at most once. This means the total running time for the verification phase for all $O(K)$ phases is $O(m)$. Therefore, the total running time of the MSTV algorithm is $O(m + n \cdot K)$ time.

For the space requirement, we store G, T, and G^* as adjacency lists of size $O(m + n)$. We also store A as an array of size $O(n)$. Therefore, the total space required is $O(m + n) + O(n) = O(m + n)$.

4.2.4 Correctness

To prove the correctness of the algorithm, we show the following:

(1) Each iteration correctly contracts all the lightest edges from T.

(2) The DFS procedure correctly maps each vertex in G with a contracted vertex in T.

(3) If there is an edge not in T that should be in the MST, the verification phase correctly finds this edge, thus proving that T is not a minimum spanning tree.

Lemma 4.2.1 *For an iteration $1 \leq t \leq K-1$, the algorithm contracts all edges whose weight is l_t.*

Proof: At phase t, the smallest edge weight is l_t. Since we use Borůvka's algorithm to contract edges with the smallest weight, the algorithm correctly contracts all edges of weight l_t. □

Lemma 4.2.2 *For an iteration $1 \leq t \leq K-1$, the DFS procedure correctly maps each vertex in G with a contracted vertex in T.*

Proof: At phase t, V_T contains only the contracted vertices in T. Suppose we select vertex $u \in V_T$. The DFS procedure starts at $u \in V^*$, and we set $A[u] = u$. We scan the edges connected to u in G^* to find all the vertices that were contracted into u from the contraction procedure. Note that the DFS procedure is the same procedure in [47] and [22]. This means any vertex $v \in V^*$ visited during the DFS procedure must be connected to u, so we set $A[v] = u$. Therefore, the mapping is correct.

We now need to ensure that there is not a vertex $v \in V^*$ that corresponds to more than one vertex in V_T. Since we contracted all the edges whose weight is less than or equal to l_t, these are the only edges in E^*. Because $t \leq K-1$, there exists at least one edge of weight l_{t+1}. Assume this edge is in E_T. Otherwise, all the vertices in T would be contracted into a single vertex. After contracting all edges of weight l_t from T, the only edges remain have a weight l_{t+1} or greater. Since $|E_T| = n-1$, we contracted less than $n-1$ edges. This means G^* cannot be connected after adding the contracted edges.

Suppose vertices u and x remain in V_T after contraction, and assume we ran one iteration of the DFS procedure starting from u. When the next DFS procedure runs from x, any vertex in V^* visited must be connected to x. Suppose we visit a vertex $v \in V^*$ that is connected to u. This means any vertex connected to x is also connected to u through v which also means G^* is connected. However, since there is at least one edge of weight l_{t+1} that is not contracted, G^* cannot be connected, meaning we have a contradiction. Therefore, each vertex in V^* corresponds to exactly one contracted vertex. □

Lemma 4.2.3 *For an iteration $1 \leq t \leq K-1$, if there exists an edge e not in E_T that should be in E_T, the verification phase will find e.*

Proof: Suppose the algorithm is at iteration t, and assume we ran the contraction and DFS phases. This means all the edges $e_{ij} \in E \backslash E_T$ of weight l_t are in E_t. Given an arbitrary edge $e_{ij} \in E_t$, there exists a unique path from i to j using the edges in the spanning tree. To compare e_{ij} with these edges, we compare $A[i]$ with $A[j]$.

Suppose $A[i] = A[j]$. This means both $i, j \in V^*$ are connected to some vertex u that is in both V^* and V_T after contracting all the edges of weight l_t. Note that u could be i or j, but this is not necessary. This means there exists a path from u to i in T and a path from u to j in T prior to contracting any edge whose weight is less than or equal to l_t. Consequently, there exists a path from i to j in T, before contraction, using only edges whose weight is no greater than l_t. Replacing any of these edges with e_{ij} will not reduce the total weight of T, meaning e_{ij} is not in the MST.

Suppose $A[i] \neq A[j]$. This means $i \in V^*$ is connected to some vertex u that is in both V^* and V_T, and $j \in V^*$ is connected to some vertex $v \neq u$ that is in both V^* and V_T. It is possible for $i = u$ or $j = v$, but neither is required. Since $u \neq v$, u and v belong in two separate components in G^*. If u and v are not connected in G^*, then there exists an edge e_{wx} in the path from u to v in T whose weight is strictly greater than l_t. Let T' be the new spanning tree that is identical to T except it contains e_{ij} instead of e_{wx}. Since $c_{wx} > c_{ij}$, the total weight of T' is strictly less than the total weight of T. Therefore, T cannot be a minimum spanning tree. □

Theorem 4.2.1 *The verification algorithm correctly determines whether or not an input spanning tree is a minimum spanning tree.*

Proof: By Lemma 4.2.1, lines 3 to 8 in Algorithm 4.2 are correct. From Lemma 4.2.2, lines 9 to 11 in Algorithm 4.2 and all of Algorithm 4.3 are correct. And, by Lemma 4.2.3, lines 12 to 16 in Algorithm 4.2 are correct. The **for** loop in lines 2 to 17 is correct since each iteration $1 \leq t \leq K-1$ only uses edges whose weight is l_t. □

4.3 Example of the Algorithm

We now provide an example of the DFS-Verify algorithm. Suppose we are given a graph G containing three distinct edge weights (i.e., $K = 3$). We want to determine if the spanning tree T is a minimum spanning tree. Figure 4.4 provides us with graph G and spanning tree T.

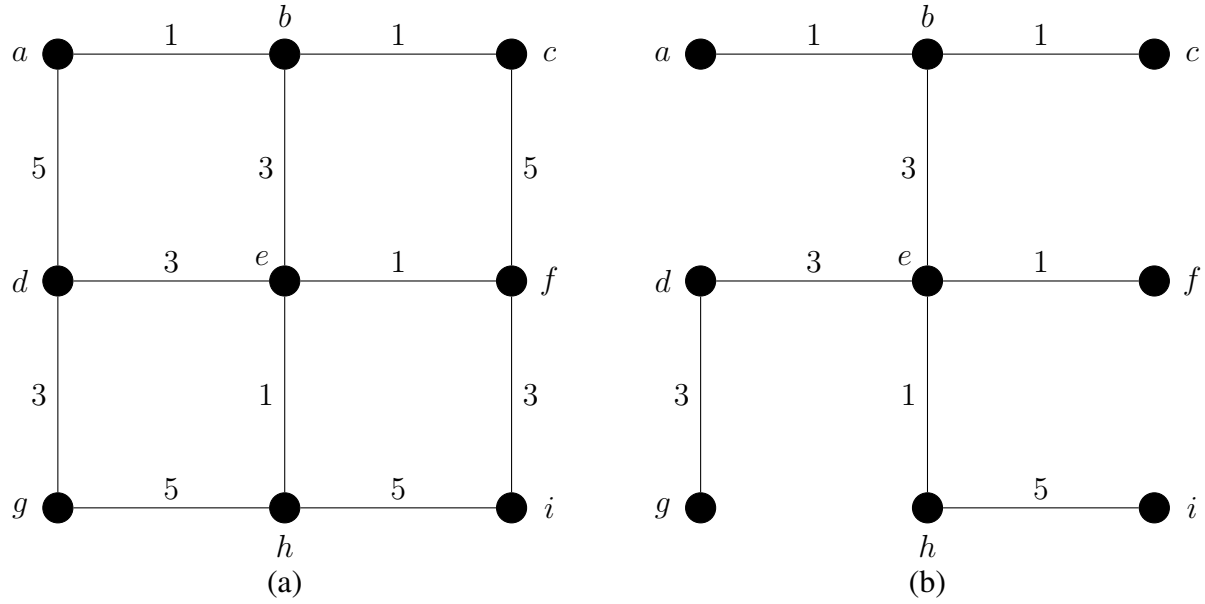


Figure 4.4: Example of the DFS-Verify algorithm. (a) is the graph G. (b) is the spanning tree T.

The first step is to add each edge in $E - E_T$ into either E_1, E_2, or E_3. E_1 is the set of edges with weight 1, E_2 is the set containing the edges with weight 3, and E_3 consists of the edges with weight 5. In this example,

$$\begin{aligned} E_1 &= \emptyset \\ E_2 &= \{e_{fi}\} \\ E_3 &= \{e_{ad}, e_{cf}, e_{gh}\}. \end{aligned}$$

We now construct graph $G^* = (V^*, E^*)$, where $V^* = V$ and $E^* = \emptyset$. We also initialize $A[v]$ to $\emptyset$ for all $v \in V$.

The next step is to contract all edges in T with the lightest weight. In this case, the weight is 1. For each edge contracted, we add the edge to G^*. After we contract these edges, we have the contracted tree T and graph G^* shown in Figure 4.5.

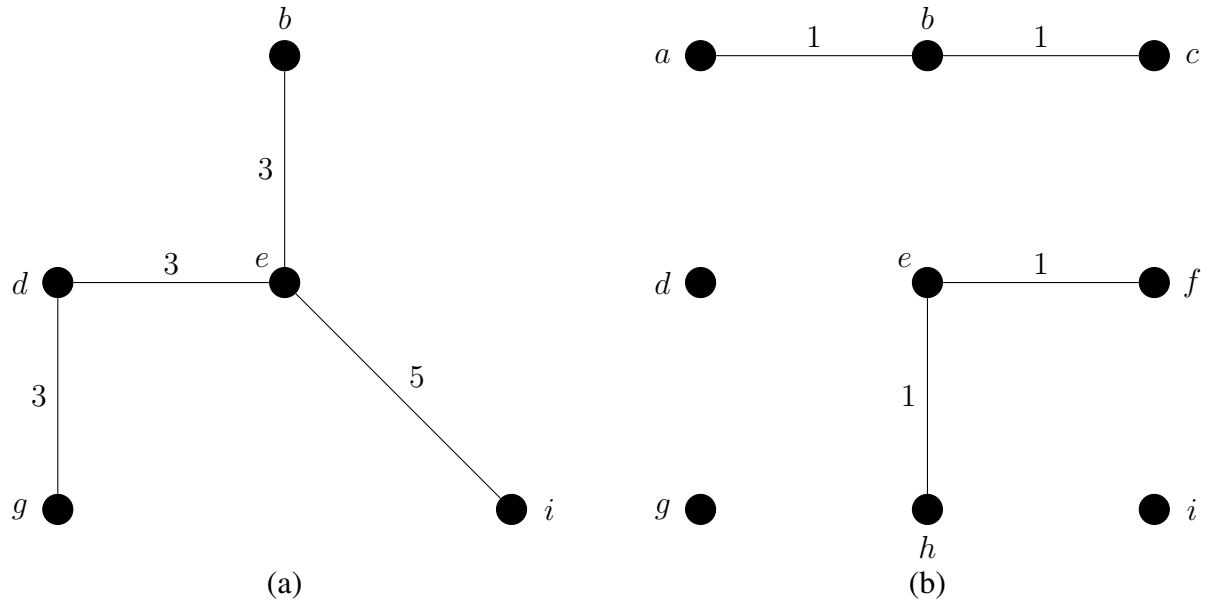


Figure 4.5: Example of the DFS-Verify algorithm. After the first contraction, (a) is the contracted tree T, and (b) is graph G^*.

We now perform a DFS for each component in G^*. In this case, we have components $\{a, b, c\}$, $\{d\}$, $\{e, f, h\}$, $\{g\}$, and $\{i\}$. We update A accordingly, as shown in Table 4.1.

Table 4.1: Example of the DFS-Verify algorithm. DFS Array A after the first DFS.

v	a	b	c	d	e	f	g	h	i
$A[v]$	a	a	a	d	e	e	g	e	i

The next step is to check if any edges in E_1 should be in the MST. Since $E_1 = \emptyset$, we know this is not the case. Therefore, we proceed with the algorithm and contract all edges in T with the next lightest weight, which is 3. After we contract these edges and add them to G^*, we have the new contracted T and graph G^* given in Figure 4.6.

We perform another DFS for each component in G^*. In this case, we have components $\{a, b, c, d, e, f, g, i\}$, and $\{i\}$. We update A accordingly, as shown in Table 4.2.

Table 4.2: Example of the DFS-Verify algorithm. DFS Array A after the second DFS.

v	a	b	c	d	e	f	g	h	i
$A[v]$	a	a	a	a	a	a	a	a	i

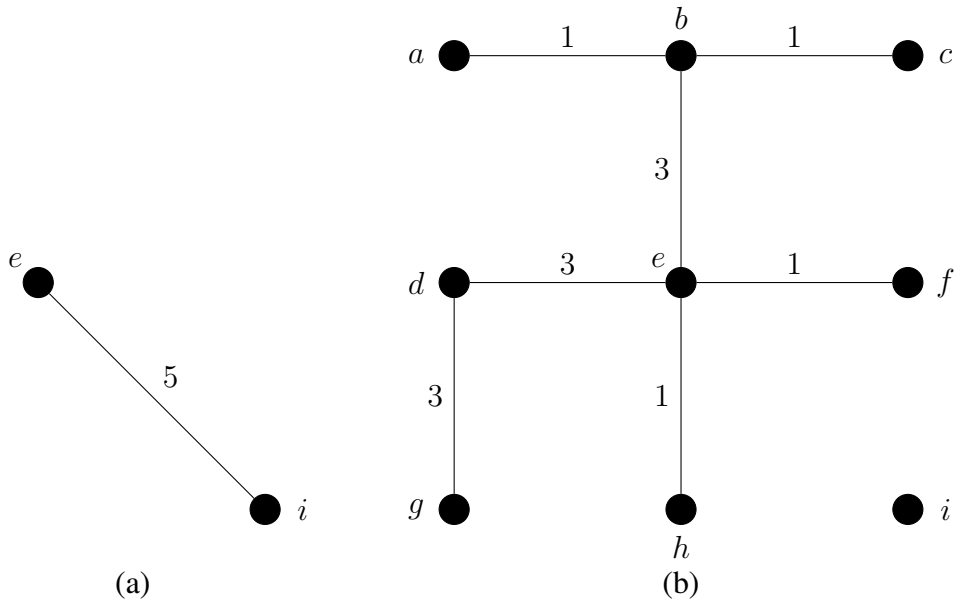


Figure 4.6: Example of the DFS-Verify algorithm. After the second contraction, (a) is the contracted tree T, and (b) is graph G^*.

The next step is to check if any edges in E_2 should be in the MST. In this case, the only edge in E_2 is e_{fi}. This means we compare $A[f]$ and $A[i]$. Since $A[f] = a$, $A[i] = i$, and $A[f] \neq A[i]$, we can replace e_{fi} with an edge in T to get a spanning tree T' whose total weight is less than T. Therefore, T is not an MST.

4.4 Empirical Study

We profile two algorithms for our empirical study. The first algorithm is our $O(m + n \cdot K)$ time algorithm. For the ease of exposition, we refer to this algorithm as the DFS-Verify algorithm. The second algorithm is Hagerup's MSTV algorithm [8]. This algorithm consists of two main components: constructing a full branching tree [3, 7] and finding the tree path maxima for a set of paired vertices [8]. Along with the total time to run Hagerup's MSTV algorithm, our results include the times of both components, which we refer to as the Branching Tree algorithm and the Tree Path Maxima algorithm, to observe how much time is spent running each subroutine.

4.4.1 Experimental Setup

Our experiments study the performance of these algorithms on graphs with varying parameters. We study four graph families that are produced by two generators. These families are chosen because they are natural and have been used in previous empirical studies [11, 48, 49]. The graph generators used are part of the 9th DIMACS Shortest Path Implementation Challenge benchmark package [50].

The first generator (SPRAND [51]) creates random graphs with n vertices and $m \geq n$ edges. The generator first constructs a Hamiltonian cycle to ensure the graph is connected. The remaining $m - n$ edges are added by randomly selecting a pair of distinct vertices. Note that the generator can produce parallel edges and/or self-loops. Our experiments include both sparse and dense random graphs of varying sizes. For sparse graphs, we set $m = 4 \cdot n$, where 4 is an arbitrary constant to represent sparse graphs since $m = O(n)$. For dense graphs, we set $m = n^{1.4}$ to ensure m is large enough to represent dense graphs but small enough to satisfy the memory allocation limitations of our testing platform.

The second generator (SPGRID [51]) creates mesh (grid) graphs. The vertices of these graphs produce a two-dimensional $x \times y$ mesh. With the exception of boundary vertices, each vertex is connected to four neighbors that are above, below, to the right, and to the left of the original vertex. For boundary vertices, some of these neighbors are absent. Our experiments include two types of mesh graphs: long and square. For long mesh graphs, y is fixed to a small constant, and x grows with the number of vertices. In our experiments, we set $x = \frac{n}{50}$ and $y = 50$. Square mesh graphs are fully connected. This means $x = y = \sqrt{n}$. Note that both mesh graphs are sparse, and the generator sets $m \approx 4 \cdot n$.

Both algorithms are written in D [52], and they are compiled and run in identical experimental settings. We use the adjacency list data structure for the graph and spanning tree. We construct the spanning tree using the algorithm in Chapter 3. Since we study how the number of incorrect edges impacts the execution times, we make a slight modification to our construction algorithm. Suppose x is the number of incorrect edges we want in the input spanning tree. In Algorithm 3.2, along with $r = argmin\{f(t) : 1 \leq t \leq K\}$, we find $r' = argmax\{f(t) : 1 \leq t \leq K\}$. If $x > 0$ and $f(r) < f(r')$, we set $e_{ij} = CurrentEdge(r')$, giving us the heaviest available edge instead

of the lightest, and decrease x by one. This ensures the incorrect spanning tree is connected and contains several edges that are part of the MST. For the ease of exposition, we refer to the case where the spanning tree has zero incorrect edges as a "Yes" instance, and we refer to the case where the spanning tree contains at least one incorrect edge as a "No" instance. Although both MSTV implementations can process graphs with real or integer weights, we use integer weights in our experiments.

For random sparse, long mesh, and square mesh graphs, we allow the sizes to be 10000, 50000, 100000, 500000, and 1 million vertices. For dense graphs, we allow the sizes to be 10000, 20000, 30000, 40000, and 50000. This allows us to set $m = n^{1.4}$ for dense graphs and not exceed the memory allocation limitation of our testing platform. We also let the ratio of the largest to the smallest edge weight in the graph be 10000. We let K, number of distinct edge weights, range from 2 to 32.

Our testing platform is a 2.0 GHz 32-bit Intel Core 2 Duo machine with a 4 GB RAM and a 2 MB cache which runs Ubuntu (version 12.04). The implementations are compiled with the DMD compiler [52] version 2.059. We report the average execution time of ten independent trials for each test.

4.5 Results and Analysis

We compare the performance of the DFS-Verify algorithm to the performance of Hagerup's algorithm based on the following domains:

(i) Suite A: Size of the graph (Figures 4.7-4.10 and Table 4.3).

(ii) Suite B: Number of distinct edge weights (Figures 4.11-4.14 and Tables 4.4-4.5).

(iii) Suite C: Number of incorrect edges for small K (Figures. 4.15-4.18 and Table 4.6).

(iv) Suite D: Number of incorrect edges for large K (Figures 4.19-4.22 and Table 4.7).

For each experiment, we provide the complete data in its respective table. Each entry in the table consists of the average execution time (the top number) and the standard deviation (the bottom number).

4.5.1 Graph Size

We first study the execution times as we increase the problem size for "Yes" instances. We let $K = 4$. Because of the variation of the execution time, we use a logarithmic scale on the Y-axis in the figures in Suite A.

We find that all four graph types produce similar results. When $K = 4$, the DFS-Verify algorithm runs faster than Hagerup's algorithm for all n. For sparse, long mesh, and square mesh graphs, the improvement is approximately 82%. We can see that the bottleneck of Hagerup's algorithm is constructing the full branching tree since it takes about twice as long to construct the branching tree compared to finding the tree path maxima.

The interesting observation is that for dense graphs, the improvement is approximately 87%. In this case, the bottleneck of Hagerup's algorithm is finding the tree path maxima instead of constructing the full branching tree. This is because the execution time of Hagerup's implementation depends on m tree path maxima queries. Since $m = n^{1.4}$ in our experiments, the number of vertex pairs is substantially more for dense graphs than the other three sparse graphs.

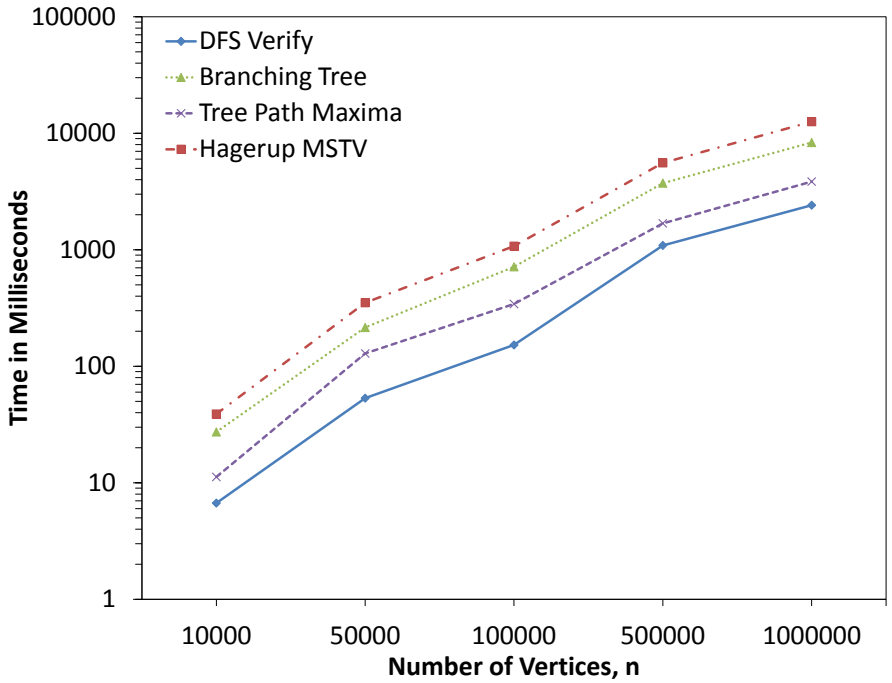


Figure 4.7: MSTV performance for sparse graphs as the number of vertices is varied and $K = 4$.

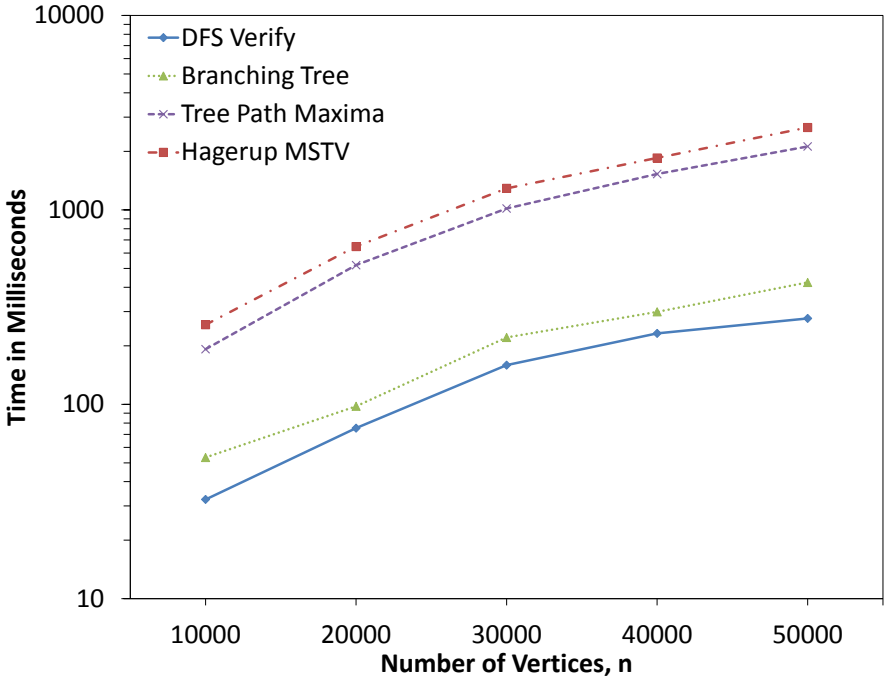


Figure 4.8: MSTV performance for dense graphs as the number of vertices is varied and $K = 4$.

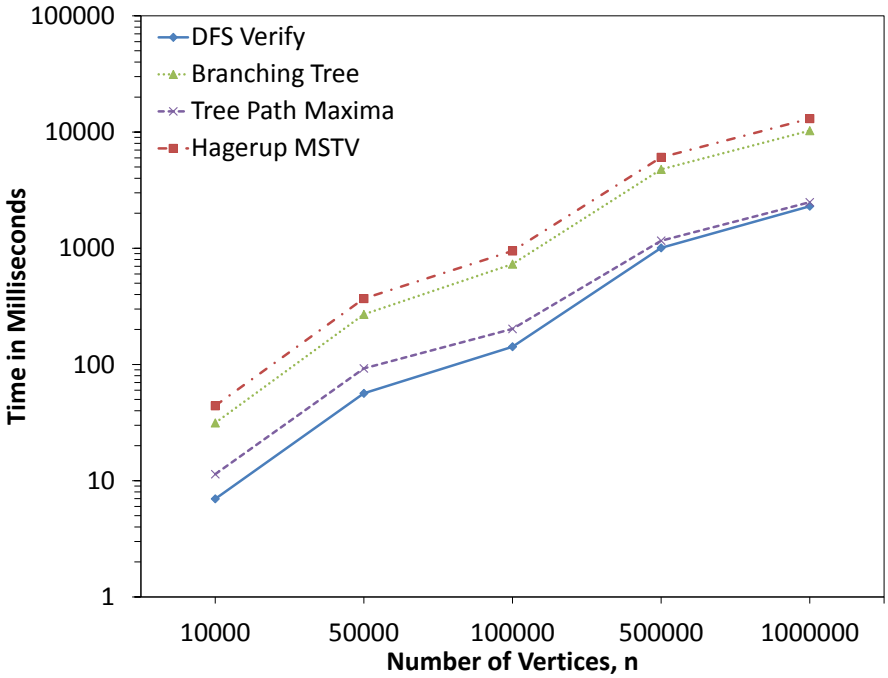


Figure 4.9: MSTV performance for long mesh graphs as the number of vertices is varied and $K = 4$.

Table 4.3: Experiment Results for Graph Size and $K = 4$ (in Milliseconds)

Graph Family	n	DFS-Verify	Branching Tree	Tree Path Maxima	Hagerup MSTV
Random Sparse	10000	6.701 0.262	27.337 0.412	11.234 0.389	38.968 0.463
	50000	53.279 0.665	215.279 2.466	128.852 2.550	351.278 5.243
	100000	152.525 0.893	714.424 7.081	342.781 3.024	1074.611 8.296
	500000	1090.315 27.232	3735.811 18.228	1687.155 21.218	5571.176 48.081
	1000000	2414.629 57.763	8326.844 93.028	3849.656 49.040	12575.886 150.467
Random Dense	10000	32.414 0.293	53.248 0.112	192.135 0.667	256.618 0.708
	20000	75.391 0.466	97.720 0.681	519.790 2.512	647.546 2.849
	30000	158.968 0.131	220.633 5.489	1016.467 23.250	1290.216 23.526
	40000	231.576 0.994	299.004 0.253	1528.781 8.515	1848.462 8.437
	50000	276.532 0.395	423.265 0.935	2119.440 19.098	2651.944 19.536
Long Mesh	10000	6.980 0.300	31.416 0.594	11.373 0.866	44.114 1.234
	50000	56.433 0.758	269.706 1.082	92.445 6.902	369.002 6.845
	100000	142.115 1.271	730.523 22.244	202.244 6.123	948.201 28.038
	500000	1007.064 24.248	4771.639 1.940	1160.698 24.921	6057.476 24.334
	1000000	2301.335 46.060	10276.019 87.818	2488.311 44.870	13050.114 129.320
Square Mesh	10000	6.746 0.252	26.817 0.395	10.562 0.440	38.655 0.687
	50000	57.590 1.475	257.092 2.837	88.652 2.076	352.482 3.528
	100000	146.909 2.122	641.678 4.295	204.912 8.824	864.060 11.795
	500000	1055.621 2.838	4544.824 19.478	1244.180 22.418	5922.473 40.997
	1000000	2359.340 3.049	9294.312 52.226	2427.942 42.635	11994.295 82.618

Note: For each entry, the top number is the average execution time, and the bottom number is the standard deviation.

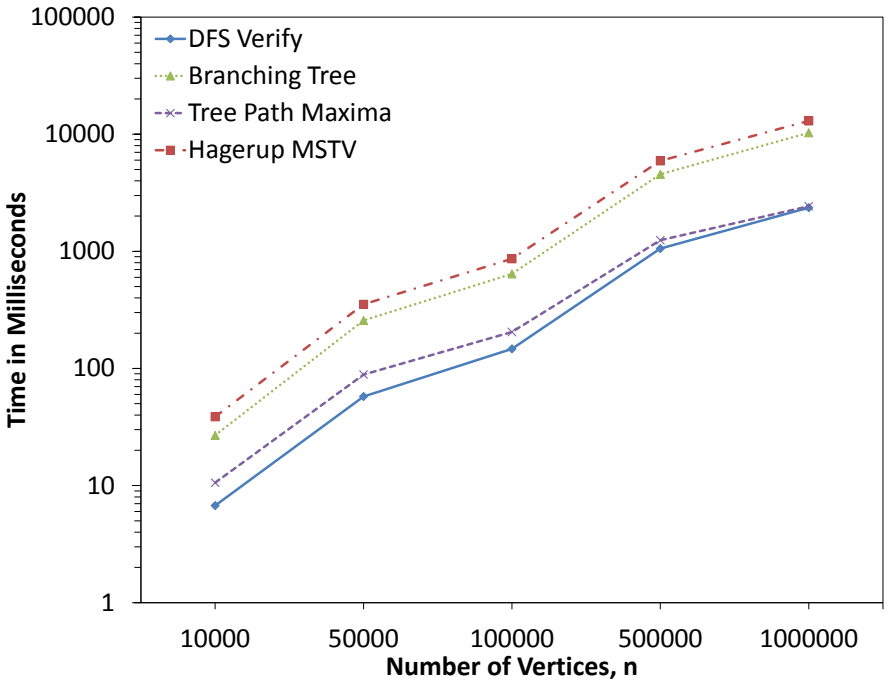


Figure 4.10: MSTV performance for square mesh graphs as the number of vertices is varied and $K = 4$.

4.5.2 Distinct Edge Weights

We next study the execution times for all graph families as we increase K. In this study, we let $n = 1$ million for sparse, long mesh, and square mesh graphs, and we let $n = 50000$ for dense graphs. Note that all experiments in this study are "Yes" instances.

For all graph types, we observe that when $K \leq 24$, the DFS-Verify algorithm is faster than Hagerup's algorithm for "Yes" instances. It is not surprising that the DFS-Verify algorithm runs slower than Hagerup's algorithm when $K \geq 28$. Recall that Hagerup's algorithm runs in $O(m+n)$ time, while our algorithm runs in $O(m + n \cdot K)$ time. This means there exists some threshold for K where Hagerup's algorithm is superior to our algorithm once that threshold is surpassed.

A surprising observation is the superiority of our algorithm for dense graphs. When $K \leq 32$, our algorithm runs faster than Hagerup's algorithm. Further, it appears the DFS-Verify algorithm is superior for larger values of K. However, the exact value of K is currently unknown.

We also observe that as K increases, the rate of the execution time of our algorithm is slower for dense graphs than for sparse graphs. This is because $n \cdot K > m = 4 \cdot n$ for sparse graphs, but

$n \cdot K < m = n^{1.4}$ for dense graphs in this study. This means increasing K does not significantly affect the execution time of the algorithm for dense graphs.

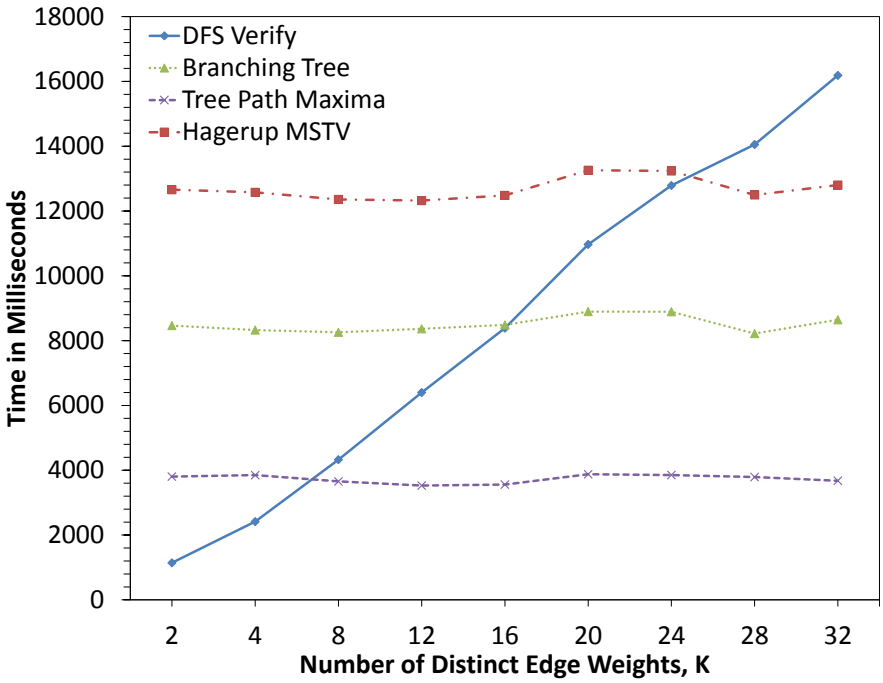


Figure 4.11: MSTV performance for sparse graphs with 1 million vertices as the value of K is varied.

4.5.3 "No" Instances for Small K

We now study the execution times for all graph families as we increase the number of incorrect edges when $K = 4$. Similar to the previous experiment, we let $n = 1$ million for sparse, long mesh, and square mesh graphs, and we let $n = 50000$ for dense graphs. In this study, we let the number of incorrect edges be 0, 1, 10, and $0.1 \cdot n$ to simulate various situations. 0 incorrect edges represents the "Yes" instance, where the spanning tree is the MST. 1 incorrect edge corresponds to the spanning tree being the MST except for a single edge. 10 incorrect edges refers to having a small number of incorrect edges. Finally, $0.1 \cdot n$ incorrect edges exemplifies having a spanning tree with many incorrect edges.

We find that when $K = 4$, the DFS-Verify algorithm runs substantially faster than Hagerup's algorithm for all "No" instances. For sparse, long mesh, and square mesh graphs, the improvement

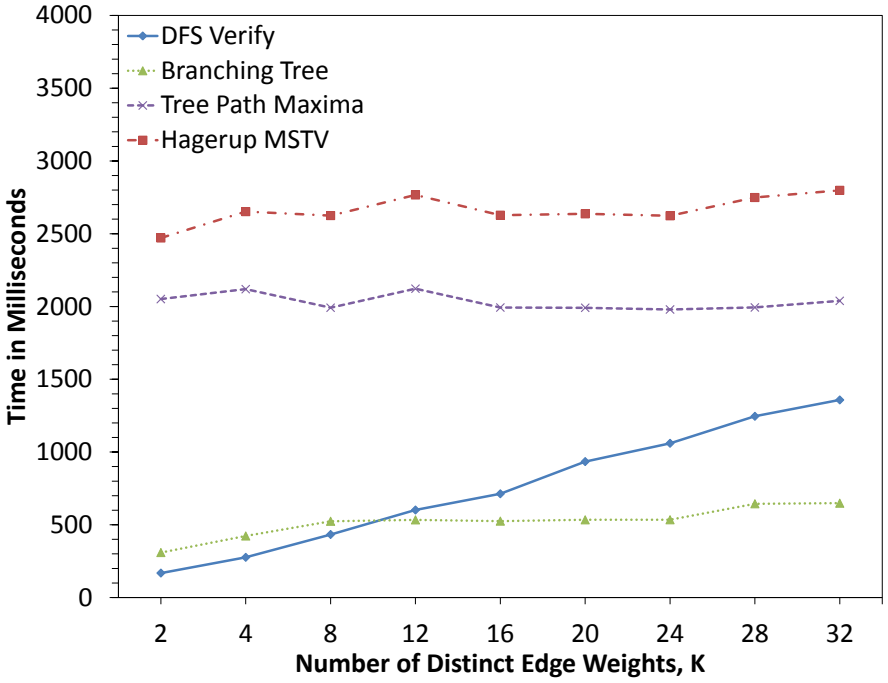


Figure 4.12: MSTV performance for dense graphs with 50000 vertices as the value of K is varied.

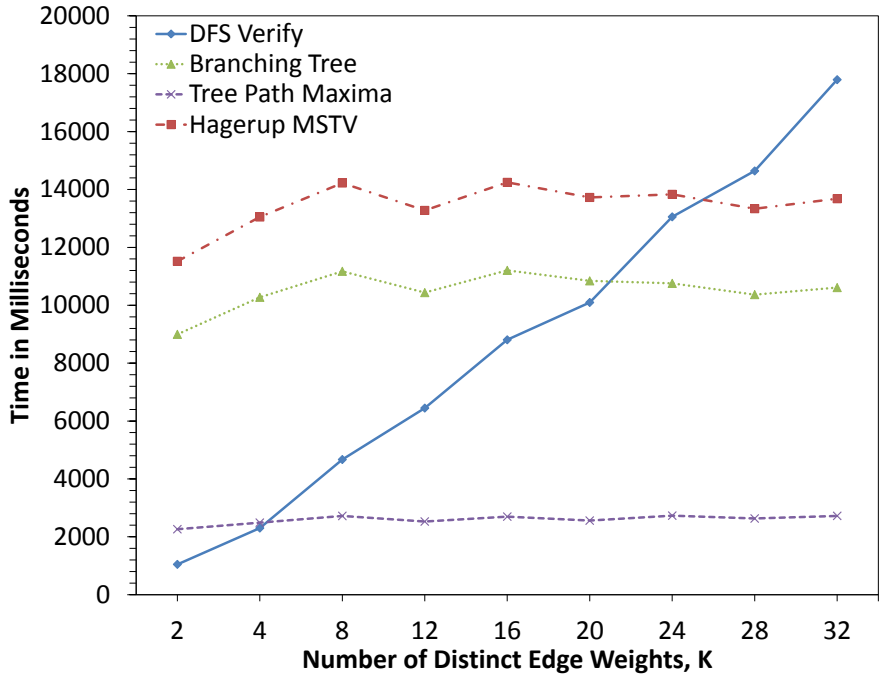


Figure 4.13: MSTV performance for long mesh graphs with 1 million vertices as the value of K is varied.

Table 4.4: Experiment Results for Distinct Edge Weights in Random Graphs (in Milliseconds)

Graph Family	K	DFS-Verify	Branching Tree	Tree Path Maxima	Hagerup MSTV
Random Sparse	2	1142.348 31.025	8465.118 176.459	3804.071 72.305	12657.006 188.900
	4	2414.629 57.763	8326.844 93.028	3849.656 49.040	12575.886 150.467
	8	4327.094 8.306	8258.244 85.478	3655.250 60.830	12356.898 136.222
	12	6397.879 60.317	8364.862 27.554	3527.734 48.883	12321.195 48.725
	16	8385.526 113.760	8484.542 54.441	3558.086 11.185	12481.266 54.404
	20	10968.399 50.501	8896.910 255.658	3877.873 64.202	13258.267 301.461
	24	12787.028 24.764	8892.347 241.426	3852.831 71.169	13235.453 276.979
	28	14051.977 249.210	8216.809 616.999	3790.210 228.995	12495.742 558.279
	32	16184.538 46.899	8643.672 36.750	3676.436 85.083	12802.212 55.944
Random Dense	2	168.402 0.107	309.455 0.644	2051.235 37.973	2470.983 37.788
	4	276.532 0.395	423.265 0.935	2119.440 19.098	2651.944 19.536
	8	433.410 2.243	524.450 1.100	1991.477 9.829	2624.907 10.122
	12	601.732 10.938	533.904 1.254	2122.211 0.554	2766.571 1.410
	16	713.115 1.351	525.416 0.852	1992.992 44.660	2627.533 44.381
	20	934.368 40.689	534.246 8.442	1991.182 36.423	2636.777 41.045
	24	1060.188 8.132	534.631 1.347	1979.592 1.827	2623.351 2.266
	28	1245.588 2.488	644.512 5.664	1993.573 18.624	2748.882 20.479
	32	1357.959 35.882	648.607 8.933	2038.494 43.157	2798.289 46.406

Note: For each entry, the top number is the average execution time, and the bottom number is the standard deviation.

Table 4.5: Experiment Results for Distinct Edge Weights in Mesh Graphs (in Milliseconds)

Graph Family	K	DFS-Verify	Branching Tree	Tree Path Maxima	Hagerup MSTV
Long Mesh	2	1047.992 5.416	8995.560 105.227	2264.408 52.414	11518.697 144.082
	4	2301.335 46.060	10276.019 87.818	2488.311 44.870	13050.114 129.320
	8	4669.892 65.796	11174.352 139.514	2721.650 46.938	14224.751 41.405
	12	6447.356 45.258	10437.315 170.123	2528.620 18.170	13276.687 176.216
	16	8806.310 31.193	11205.928 166.344	2697.621 10.999	14246.187 171.621
	20	10097.292 47.295	10843.650 199.745	2559.952 32.615	13724.930 212.268
	24	13054.662 34.012	10756.701 69.132	2730.702 15.638	13831.937 63.746
	28	14641.079 54.409	10368.599 317.676	2632.384 7.168	13336.820 316.325
	32	17793.836 47.057	10611.177 109.254	2721.808 17.097	13681.168 97.983
Square Mesh	2	1140.780 2.821	9613.015 48.688	2472.790 52.188	12360.878 73.088
	4	2359.340 3.049	9294.312 52.226	2427.942 42.635	11994.295 82.618
	8	4713.669 5.235	11014.091 109.853	2687.769 22.472	14026.244 3.745
	12	6612.330 48.909	10757.437 119.197	2628.753 51.587	13712.444 127.380
	16	8677.986 49.749	10975.507 130.991	2638.542 37.411	13948.217 90.652
	20	10849.766 43.940	11387.892 73.607	2725.273 7.838	14457.771 76.467
	24	12112.351 47.525	10877.700 189.404	2580.852 10.978	13785.732 197.692
	28	14089.816 71.830	10998.235 266.257	2566.161 3.512	13905.850 270.946
	32	16212.266 439.507	11289.858 92.373	2603.514 26.859	14223.861 76.086

Note: For each entry, the top number is the average execution time, and the bottom number is the standard deviation.

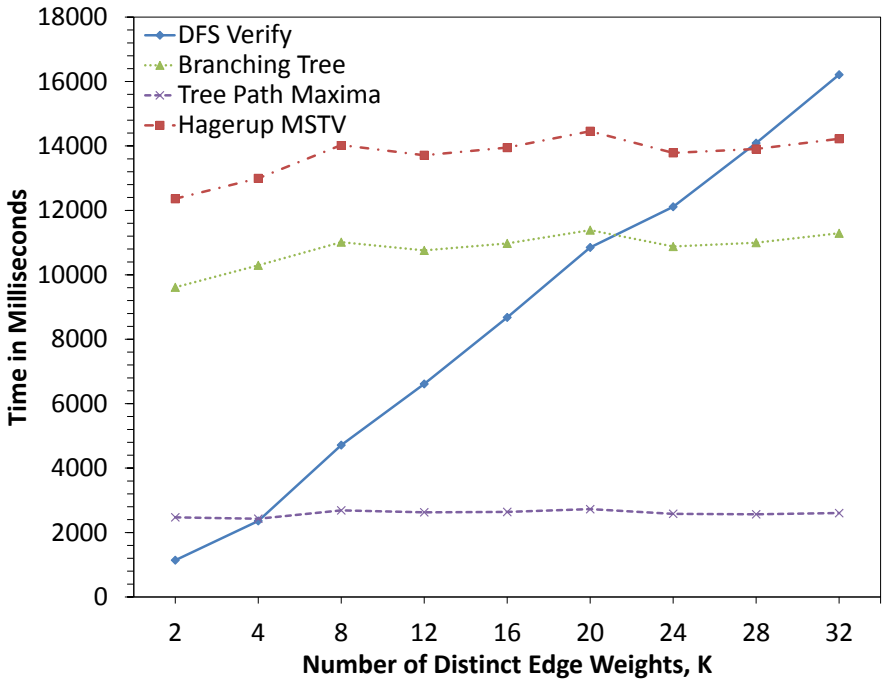


Figure 4.14: MSTV performance for square mesh graphs with 1 million vertices as the value of K is varied.

is approximately 92% and 98% for dense graphs. This is because our algorithm checks the non-tree edges in increasing weight as we process the spanning tree. If the spanning tree is incorrect, our algorithm halts as soon as the edge is detected which means fewer iterations are required. Hagerup's algorithm processes the entire spanning tree before checking non-tree edges. Also, Hagerup's algorithm cannot assume the edges are sorted by increasing weight, as sorting the edges requires $O(m \cdot \log m)$ space, which is non-linear.

We also analyze the impact of increasing the number of incorrect edges. For the DFS-Verify algorithm, there is a slight improvement when there are more incorrect edges. With more incorrect edges, it is more likely a non-tree edge should be in the MST with a lighter weight. This means we can detect an incorrect edge with fewer iterations. For Hagerup's algorithm, there does not appear to be a clear correlation between the number of incorrect edges and the execution time.

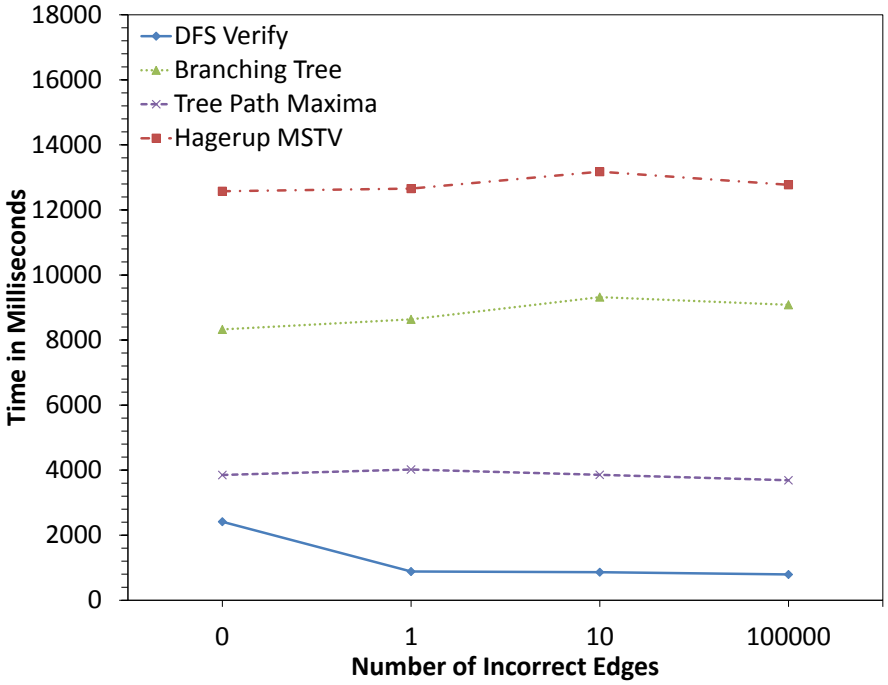


Figure 4.15: MSTV performance for sparse graphs with 1 million vertices as the number of incorrect edges is varied and $K = 4$d.

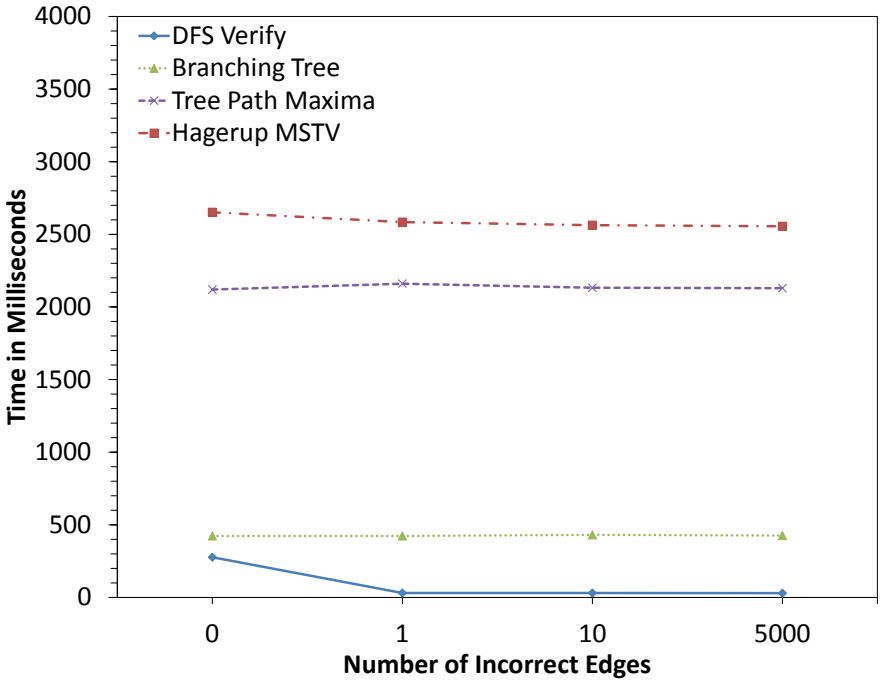


Figure 4.16: MSTV performance for dense graphs with 50000 vertices as the number of incorrect edges is varied and $K = 4$.

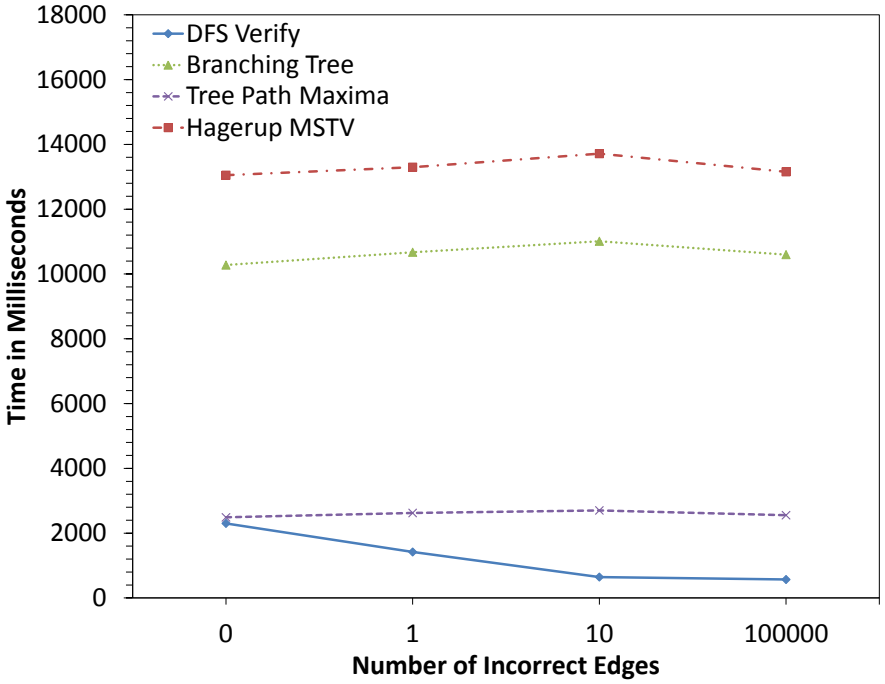


Figure 4.17: MSTV performance for long mesh graphs with 1 million vertices as the number of incorrect edges is varied and $K = 4$.

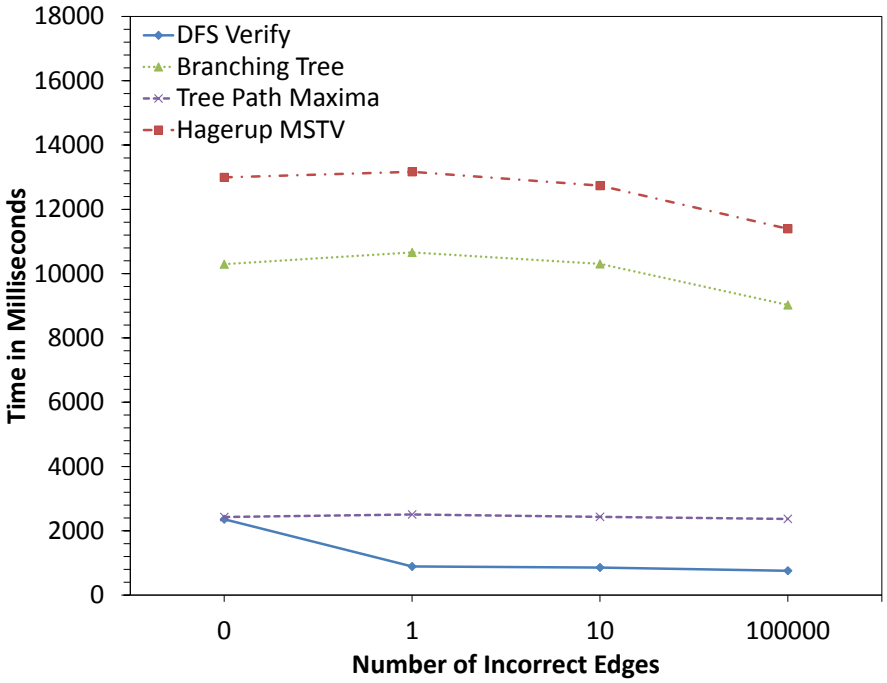


Figure 4.18: MSTV performance for square mesh graphs with 1 million vertices as the number of incorrect edges is varied and $K = 4$.

Table 4.6: Experiment Results for Incorrect Edges and $K = 4$ (in Milliseconds)

Graph Family	Incorrect Edges	DFS-Verify	Branching Tree	Tree Path Maxima	Hagerup MSTV
Random Sparse	0	2414.629 57.763	8326.844 93.028	3849.656 49.040	12575.886 150.467
	1	882.413 11.395	8636.568 87.975	4019.027 56.430	12655.691 129.133
	10	861.342 28.962	9321.191 106.724	3854.498 49.311	13175.712 151.915
	100000	791.853 11.574	9084.229 55.537	3688.891 57.644	12773.143 99.417
Random Dense	0	276.532 0.395	423.265 0.935	2119.440 19.098	2651.944 19.536
	1	30.161 0.783	423.382 0.904	2160.481 45.769	2583.992 46.020
	10	30.269 0.961	431.040 13.070	2131.901 30.447	2563.011 35.330
	5000	29.393 0.393	425.970 2.951	2129.417 4.512	2555.412 5.006
Long Mesh	0	2301.335 46.060	10276.019 87.818	2488.311 44.870	13050.114 129.320
	1	1419.031 4.099	10671.563 47.709	2623.980 61.538	13297.716 77.299
	10	645.297 2.609	11010.902 44.895	2702.582 43.321	13713.505 52.265
	100000	569.368 7.409	10598.034 104.460	2555.240 48.606	13153.296 47.278
Square Mesh	0	2359.340 3.049	9294.312 52.226	2427.942 42.635	11994.295 82.618
	1	889.060 1.366	10658.754 64.203	2508.582 46.742	13167.379 88.540
	10	857.780 0.858	10301.865 50.421	2433.871 44.501	12735.758 80.483
	100000	757.959 1.217	9024.737 88.256	2370.048 51.561	11395.014 123.408

Note: For each entry, the top number is the average execution time, and the bottom number is the standard deviation.

4.5.4 "No" Instances for Large K

For our last study, we explore what happens when we increase the number of incorrect edges for a larger K. We use the same parameters as the study in Chapter 4.5.3 except $K = 20$.

The results are very similar to the study in Chapter 4.5.3. The DFS-Verify algorithm runs faster than Hagerup's algorithm regardless of the number of incorrect edges. The key difference is we can clearly see the improvement in the DFS-Verify algorithm as we increase the number of incorrect edges. As with the previous study, this is because we can detect an incorrect edge faster if there are more incorrect edges in the spanning tree.

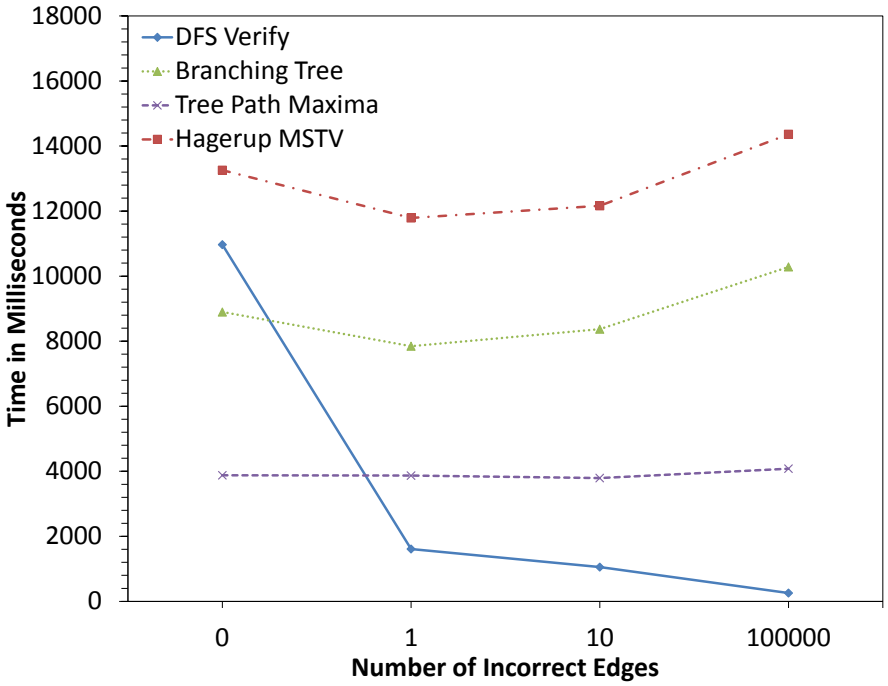


Figure 4.19: MSTV performance for sparse graphs with 1 million vertices as the number of incorrect edges is varied and $K = 20$d.

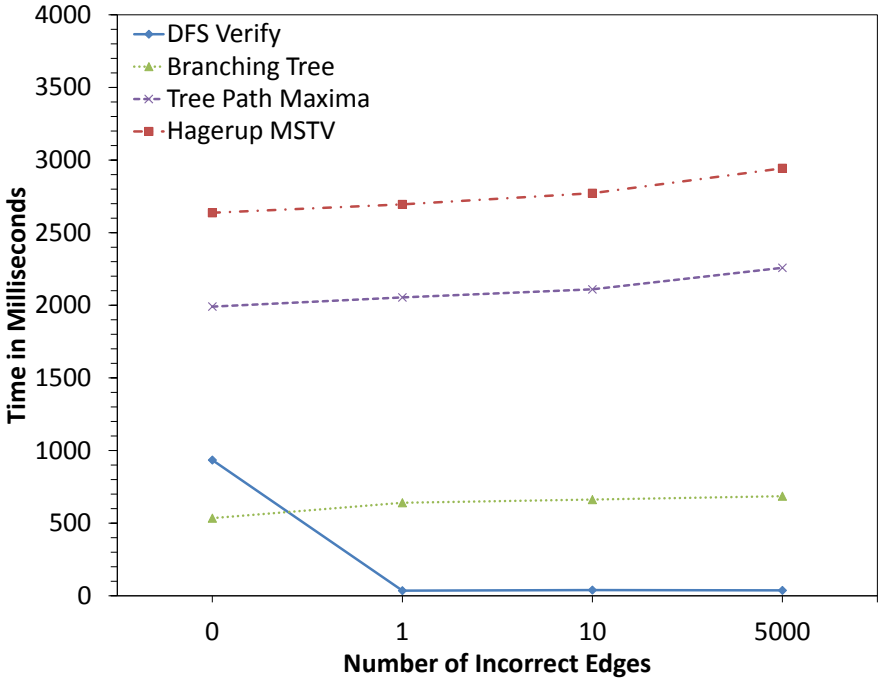


Figure 4.20: MSTV performance for dense graphs with 50000 vertices as the number of incorrect edges is varied and $K = 20$.

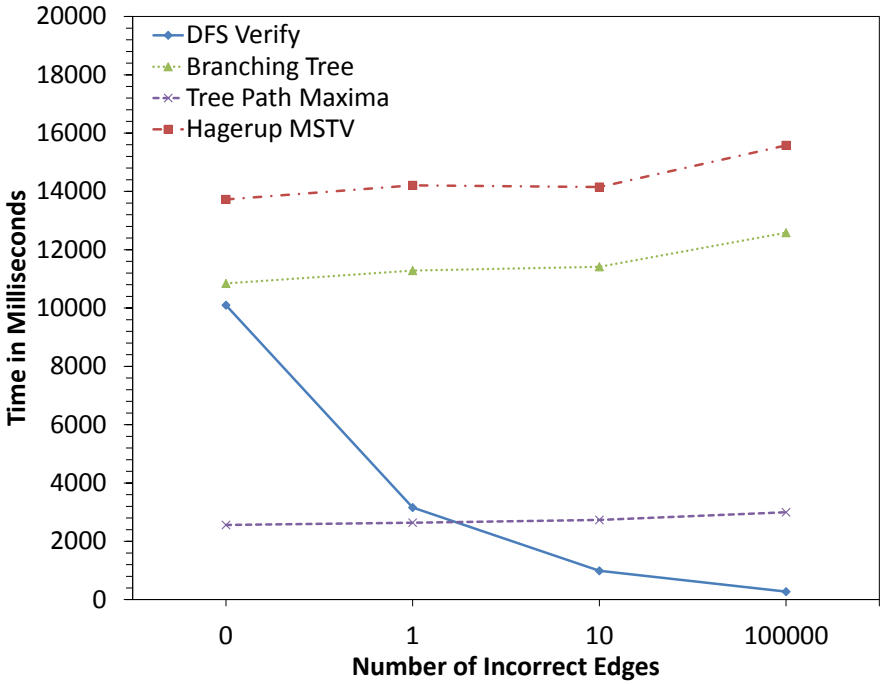


Figure 4.21: MSTV performance for long mesh graphs with 1 million vertices as the number of incorrect edges is varied and $K = 20$.

Table 4.7: Experiment Results for Incorrect Edges and $K = 20$ (in Milliseconds)

Graph Family	Incorrect Edges	DFS-Verify	Branching Tree	Tree Path Maxima	Hagerup MSTV
Random Sparse	0	10968.399 50.501	8896.910 255.658	3877.873 64.202	13258.267 301.461
	1	1610.180 2.050	7846.480 221.254	3867.246 52.418	11788.940 271.337
	10	1053.705 0.950	8369.830 82.268	3791.907 90.324	12162.570 54.644
	100000	256.466 0.237	10283.704 421.813	4079.794 153.148	14363.520 472.273
Random Dense	0	934.368 40.689	534.246 8.442	1991.182 36.423	2636.777 41.045
	1	35.004 0.147	640.342 8.037	2054.102 34.148	2694.466 33.752
	10	39.096 4.270	661.960 23.876	2109.573 86.315	2771.555 103.993
	5000	36.943 1.463	685.664 19.572	2257.655 80.657	2943.343 86.135
Long Mesh	0	10097.292 47.295	10843.650 199.745	2559.952 32.615	13724.930 212.268
	1	3158.220 11.638	11284.356 198.566	2638.174 33.118	14207.015 218.639
	10	987.014 36.093	11415.967 47.940	2733.496 28.187	14149.484 73.574
	100000	273.376 24.432	12583.817 495.061	2996.000 179.286	15579.840 610.487
Square Mesh	0	10849.766 43.940	11387.892 73.607	2725.273 7.838	14457.771 76.467
	1	952.253 32.920	11008.541 816.491	2599.065 53.340	13607.629 297.534
	10	292.243 17.578	11953.929 469.635	2869.879 102.028	14823.832 484.550
	100000	255.868 0.819	12556.474 198.108	2746.781 13.063	15303.298 197.260

Note: For each entry, the top number is the average execution time, and the bottom number is the standard deviation.

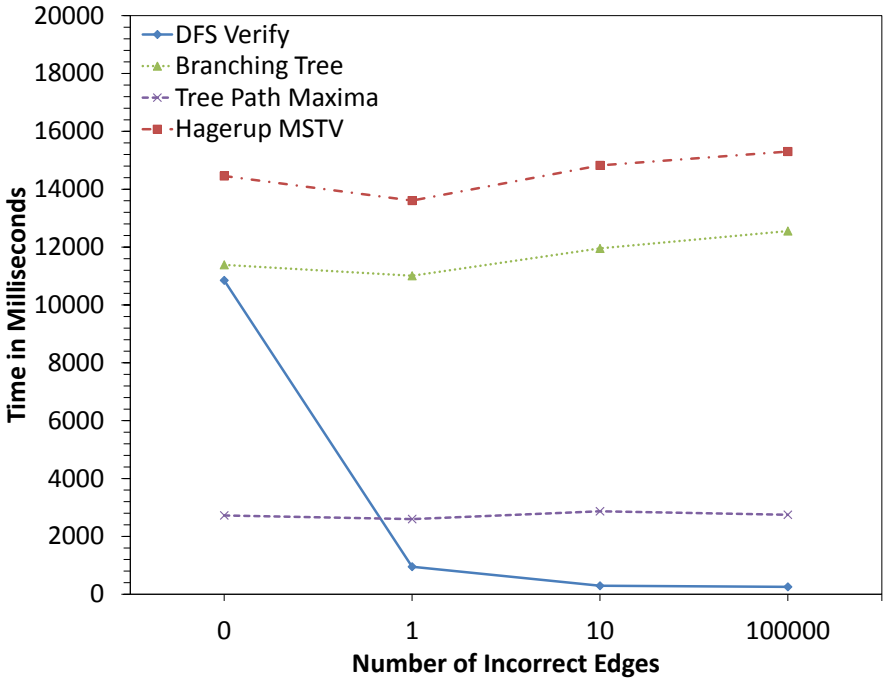


Figure 4.22: MSTV performance for square mesh graphs with 1 million vertices as the number of incorrect edges is varied and $K = 20$.

Part II

The Undirected Negative Cost Cycle Detection Problem

Chapter 5

Introduction

A *negative cost cycle* is defined as a path from a vertex to itself, whose total cost (or weight) is negative. The problem of finding a negative cost cycle in a graph is called the *negative cost cycle detection problem* (NCCD). For directed graphs, the problem has been widely studied [9, 10, 11, 12, 13, 14, 15, 16]. Figure 5.1 provides an example of a negative cost cycle. In this example, the path $b \rightarrow d \rightarrow c \rightarrow b$ is a negative cost cycle since the total cost of the path is $-3 + -5 + -2 = -10$.

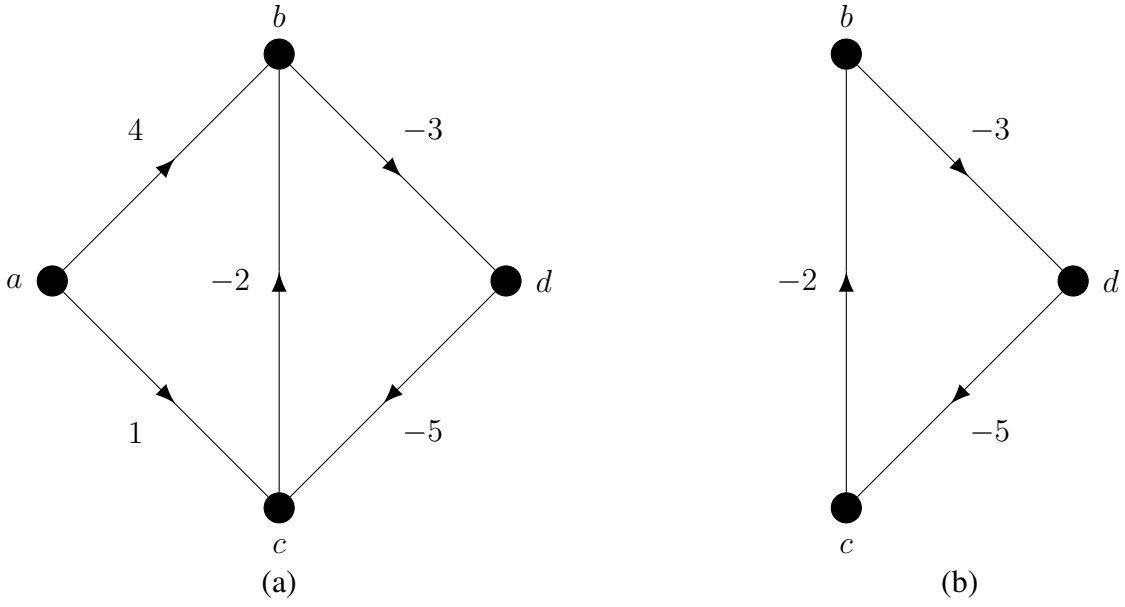


Figure 5.1: An example of a negative cost cycle. (a) is the graph, and (b) is a negative cost cycle of the graph.

In Part II of this thesis, we focus on undirected graphs. Hence, the problem we are interested in is the *undirected negative cost cycle detection problem* (UNCCD). Unlike directed graphs, detecting a negative cost cycle for undirected graphs is significantly difficult. One idea is to transform an undirected graph into a directed graph by replacing each undirected edge with two directed edges going in opposite directions. However, if an undirected edge has a negative cost, then the two directed edges also have negative costs. This results in a negative cost cycle in the directed graph that did not exist in the undirected graph.

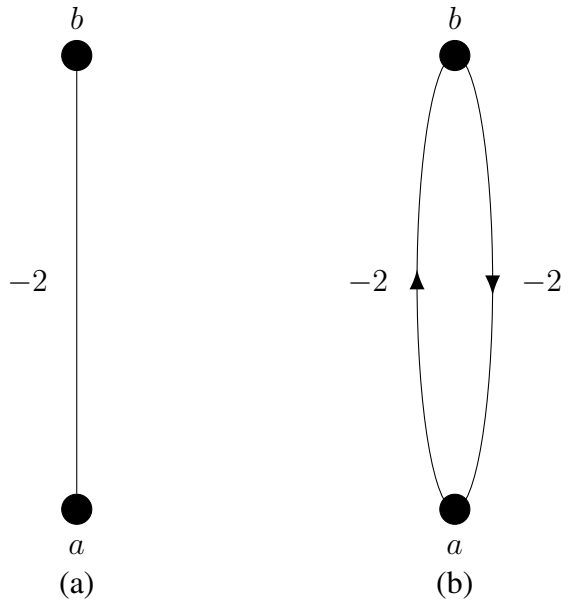


Figure 5.2: The problem with transforming an undirected edge into two directed edges. (a) is the undirected edge, and (b) is the transformed directed edge.

We illustrate this issue in Figure 5.2. Consider edge e_{ab} in Figure 5.2(a). If we transform it into two directed edges that are oriented in opposite directions, we get Figure 5.2(b). Since the cost of the undirected edge is -2, both directed edges must also be -2. However, we now have the path $a \to b \to a$, whose total cost is -4. This results in a negative cost cycle that was not present prior to the transformation.

The next two chapters present our results related to the UNCCD problem. We first discuss two known approaches for the UNCCD problem. The *b-matching approach* transforms a graph G into a new graph G' and finds the minimum weight perfect matching (MWPM) in G' [17, 18] to detect the presence of a negative cost cycle. On a graph with n vertices and m edges, we show

that the algorithm runs in $O((m+n)^2 \cdot \log(m+n))$ time. The *T-join approach* is an $O(n^3)$ time algorithm [23] that utilizes properties of T-joins [19, 20] and contains algorithms for the all pairs shortest paths (APSP) and MWPM problems as subroutines.

We next describe how to improve both UNCCD algorithms when the edge costs are integers in the range $\{-K \cdot \cdot K\}$, where K is a positive constant. By using efficient MWPM and APSP algorithms in graphs with integral positive edge costs, we show that the b-matching approach runs in $O((m+n)^{1.5} \cdot (\log(m+n))^{1.5} \cdot \sqrt{\alpha(m+n, m+n)})$ time, and the T-join approach runs in $O(n^{2.5} \cdot (\log n)^{1.5} \cdot \sqrt{\alpha(n^2, n)})$ time, where $\alpha(x, y)$ represents the inverse Ackermann function [21, 22]. Both running times are the best known bounds.

We then present the first extensive empirical study for the UNCCD problem. Although there exists a previous empirical study for the UNCCD problem [23], there are several limitations with the study. First, the study examines only the T-join approach and mentions the b-matching approach only in passing. The implementation used in the study is only a proof of concept to illustrate the algorithmic techniques used. Only families of sparse graphs are included in the study. In the empirical study in this thesis, we examine comprehensive implementations for both the b-matching and T-join approaches for various graph families and sizes. We note there already exist several empirical studies that analyze negative cost cycle detection algorithms for directed graphs [11, 13, 53, 16]. However, we provide the first empirical study that analyzes negative cost cycle detection algorithms for *undirected* graphs.

5.1 Preliminaries and Notation

We are given a weighted, undirected graph $G = (V, E, c)$, where V is the set of n vertices, E is the set of m edges, and $\mathbf{c} : E \rightarrow \mathbb{R}$ is a cost function. For each edge $e_{ij} \in E$, we let c_{ij} be the cost of that edge. Similarly, for any set of edges $E' \subseteq E$, we let $c(E')$ denote the total cost of E'. Since G is undirected, $e_{ij} = e_{ji}, \forall e_{ij} \in E$. We represent G as an adjacency list Adj, where for each vertex $v \in V$, $Adj(v)$ is the set of outgoing edges from v in G. Although $|Adj| = 2 \cdot m$ for undirected graphs, this does not negatively affect our UNCCD algorithms.

A *subgraph* of $G = (V, E)$ is a graph $H = (V_H, E_H)$, where $V_H \subseteq V$ and $E_H \subseteq E$. A graph $H = (V_H, E_H)$ is an *induced subgraph* of G if H is a subgraph of G, and $E_H = \{e_{ij} : e_{ij} \in E$

and $v_i, v_j \in V_H\}$. In other words, H is an induced subgraph is G if H has exactly the edges that appear in G over the same vertex set. For ease of exposition, we say that H is the subgraph of G *induced by* V_H, and we denote this as $H = G[V_H]$.

Figure 5.3 provides an example of both a subgraph and induced subgraph. Figure 5.3(a) is the graph G. H (Figure 5.3(b)) is a subgraph of G. However, it is not an induced subgraph since $b, d \in V_H$, but $e_{bd} \notin E_H$. H' (Figure 5.3(c)) is an induced subgraph of G since $a, b, d \in V_{H'}$ and all edges in G that connect these vertices are in $E_{H'}$.

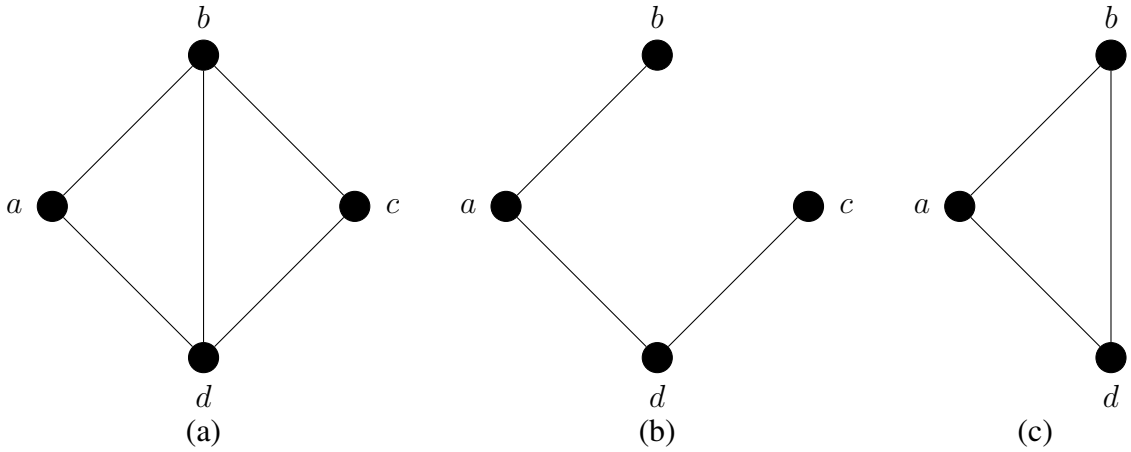


Figure 5.3: An example of subgraphs. (a) is the graph G. (b) is a subgraph H. (c) is an induced subgraph $H' = G[V_{H'}]$.

A *path* P is a sequence of vertices such that there exists a unique edge e_{ij} connecting one vertex to the next vertex in the sequence. A *cycle* C is similar to a path except the first and last vertices in the sequence are the same. Note that C is a subgraph, where V_C is the sequence of vertices, and E_C is the set of edges connecting the vertices in the sequence. Hence, a *negative cost cycle* is a cycle $C = (V_C, E_C, c)$, where $c(E_C) < 0$.

Finally, the *metric closure* of an undirected, weighted graph is defined as follows [20]:

Definition 5.1.1 *Given an undirected, weighted graph $G = (V, E, c)$ with no negative cost cycles, the metric closure of G is a graph $\bar{G} = (V_{\bar{G}}, E_{\bar{G}}, \bar{c})$ such that $V_{\bar{G}} = V$, and $\forall i, j \in V_{\bar{G}}$, e_{ij} is an edge in $E_{\bar{G}}$ with weight $\bar{c}(e)$, where $\bar{c}(e)$ is the cost of the shortest path from i to j in G.*

In other words, if we have a graph that does not have any negative cost cycles, we can find the metric closure by solving the APSP problem.

Figure 5.4 gives an example of the metric closure. We let G (Figure 5.4(a)) be the initial graph. We let G' (Figure 5.4(b)) be the metric closure of G, where for each edge e_{ij} in G', $c(e_{ij})$ is the cost of the shortest path from i to j in G.

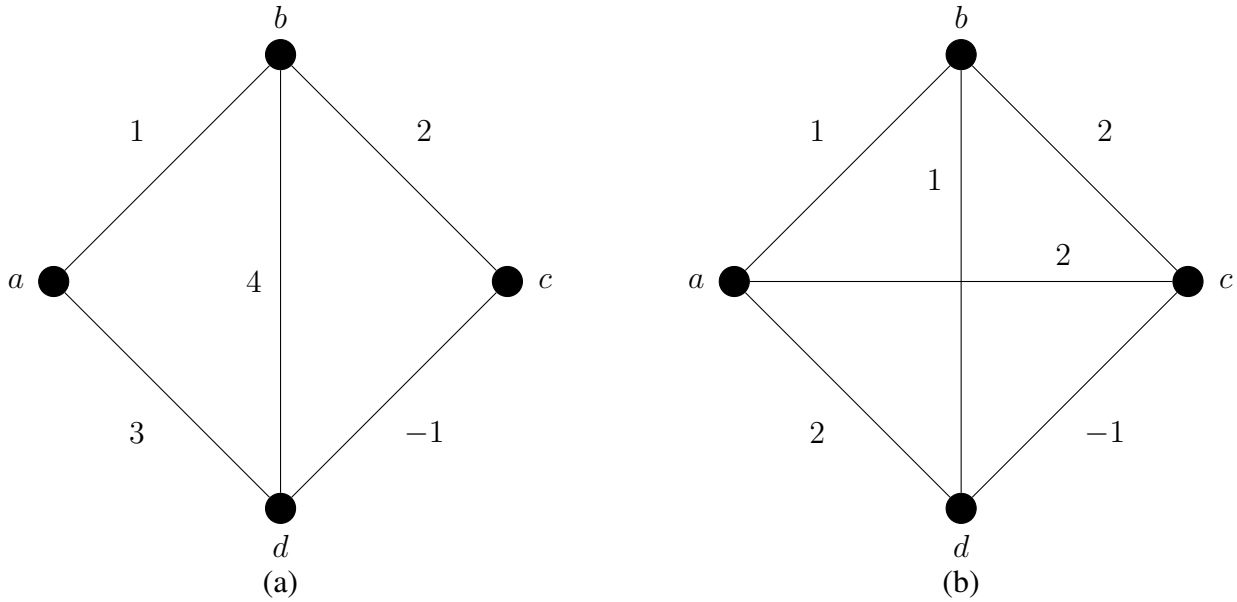


Figure 5.4: An example of the metric closure. (a) is the graph G, and (b) is a the metric closure G'.

Chapter 6

UNCCD Algorithms

In this chapter, we discuss the currently known approaches for the UNCCD problem and how to improve them when the edge costs are restricted to integers. The b-matching approach has been widely studied [54, 55, 18, 20, 56] and involves transforming the graph into a new graph and finding the minimum weight perfect matching (MWPM) in the new graph. On a graph with n vertices and m edges, the b-matching approach runs in $O((m+n)^2 \cdot \log(m+n))$ time. The T-join approach has also been extensively studied [57, 58, 20] and involves algorithms for solving the APSP and MWPM problems as subroutines. This algorithm runs in $O(n^3)$ time.

6.1 The b-matching Approach

In this section, we describe the algorithm that solves the UNCCD problem by utilizing b-matchings. Briefly, the algorithm is a modification of Edmonds' algorithm [17] that solves the shortest path problem in undirected graphs by transforming it into the non-bipartite weighted matching problem. Since the UNCCD problem can be reduced to an instance of the shortest path problem [18], we can use shortest path algorithms to detect negative cost cycles in undirected graphs. The details of the algorithm are explained below.

6.1.1 Preliminaries

We are given a graph $G = (V, E, c)$. For ease of exposition, we let G be uncapacitated (i.e., each edge $e_{ij} \in E$ has capacity equal to 1). The number of edges adjacent to a vertex $v \in V$ is

called the *degree* of v. Matching problems involve choosing a subset of edges in G, subject to specific degree constraints on the vertices. Hence, a 1-matching, or simply *matching*, in G is a set of edges $M \subseteq E$ such that in the subgraph $G_M = (V, M)$ of G, each vertex $v \in V$ is adjacent to at most one edge. In other words, a matching M in G is a subset of E such that no two edges in M are adjacent ot the same vertex.

The definition of a matching can be extended to b-*matchings* [55].

Definition 6.1.1 *H is a b-matching in G if each vertex $v \in V$ is adjacent to no more than b_v edges in the subgraph $G_H = (V, H)$, where b_v is a positive integer.*

Note that $H \subseteq E$, while b_v represents an upper bound for the degree of each vertex $v \in V$.

A b-matching is called *perfect* if the degree constraint imposed on each vertex is satisfied as an equality. In other words, H is a perfect b-matching in G if each vertex $v \in V$ has exactly b_v adjacent edges in G_H. Let $adj_G(v)$ denote the set of edges adjacent to vertex v in a graph G. Then H is a perfect b-matching in G if $|adj_{G_H}(v)| = b_v$ for each $v \in V$. Note that if $b_v = 2$ for all $v \in V$, and H is a perfect b-matching of G, we call H a perfect 2-matching. Accordingly, M is a perfect matching in G if $|adj_{G_M}(v)| = 1$ for each vertex $v \in V$.

In this thesis, we focus on perfect matchings and perfect b-matchings. Figure 6.1 provides such an example. We are given an undirected, uncapacitated graph G in Figure 6.1(a). Figure 6.1(b) contains the edges of a perfect b-matching in G, where $b_a = 1$, $b_b = 2$, $b_c = 2$, and $b_d = 3$. A perfect 2-matching in G is given in Figure 6.1(c) since the degree of all vertices in this subgraph is 2. Note that a perfect 2-matching in G implies a Hamilton cycle in G or a set of vertex-disjoin cycles that cover all the vertices in V. If $b_v = 1$, for all $v \in V$, then M is a perfect matching of G. Figures 6.1(d) and 6.1(e) represent perfect matchings in G.

Given a matching M in a weighted graph G, $c(M)$ denotes the total weight (or cost) of M. In other words, $c(M) = \sum_{e_{ij} \in M} c(e_{ij})$. One key problem in this domain is finding the MWPM. The first known algorithm for the MWPM problem is provided by Edmonds [54, 59]. Since then, several advancements have been made that improve the running time [60, 61, 62, 63, 64]. The current fastest algorithm for solving the MWPM problem is provided by Gabow and runs in $O(n \cdot (m + n \cdot \log n))$ time [65].

Along with designing efficient algorithms for the MWPM problem, there has been an increasing

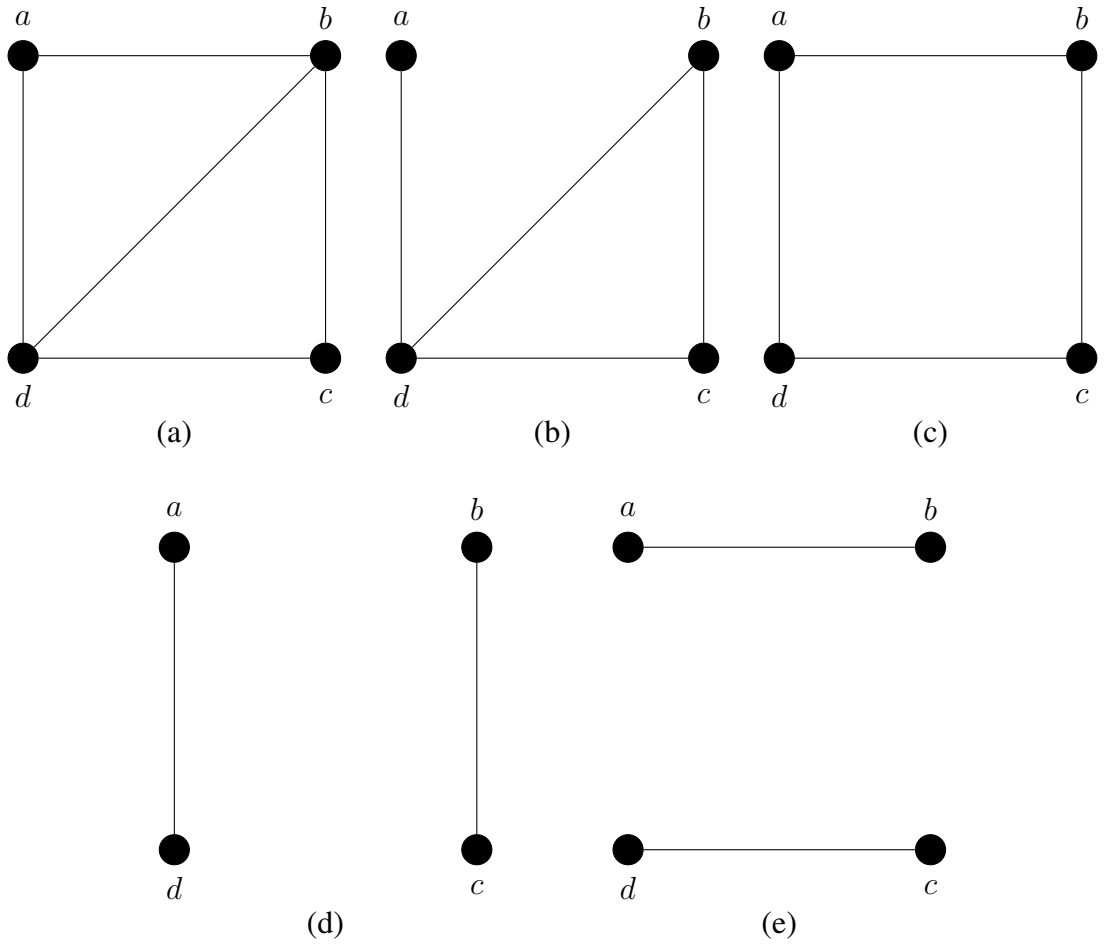


Figure 6.1: Examples of b-matchings and perfect matchings. (a) is an undirected graph G. (b) is a perfect b-matching in G, where $b_a = 1, b_b = 2, b_c = 2$, and $b_d = 3$. (c) is a perfect 2-matching in G. (d) and (e) are perfect matchings in G.

interest in developing robust implementations of Edmonds' algorithm [66, 67, 68, 69, 70, 71, 72, 73, 74, 75]. The most recent implementation is known as the Blossom V algorithm [56]. This implementation has outperformed previous implementations [74, 75].

6.1.2 UNCCD Algorithm based on b-matching

The b-matching approach performs three consecutive transformations on G to obtain a new graph $G' = (V', E')$. We also modify b_v, for each vertex $v \in V$, to obtain the correct perfect matching. The algorithm starts by initializing G' to G. The first transformation adds self-loops to each vertex in G, and we denote the resulting graph as $G_1 = (V_1, E_1)$. We also set b_v to 2, for

each vertex $v \in V'$. We illustrate this transformation in Figure 6.2. In this example, Figure 6.2(a) is our intitial graph G, and Figure 6.2(b) is the transformed graph G_1.

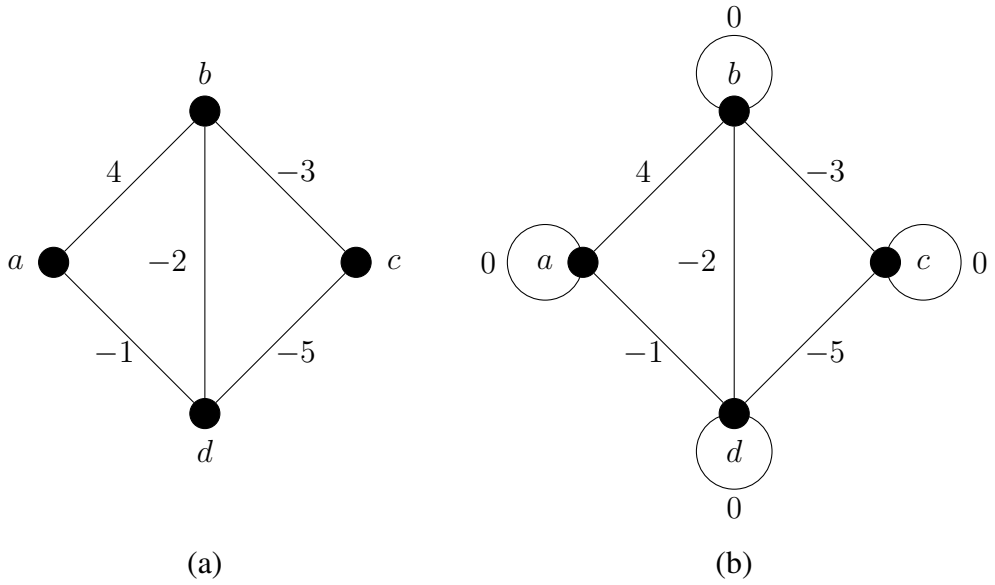


Figure 6.2: An example of the b-matching approach. (a) is the initial graph G. (b) is the resulting graph G_1 after the first transformation.

We then transform G' by adding two vertices, denoted as k and l, for each edge $e_{ij} \in E'$, where $i \neq j$. The resulting graph is denoted as $G_2 = (V_2, E_2)$. We also set each b_k and b_j to 1. We demonstrate this transformation in Figure 6.3, which is the result of transforming G_1 from Figure 6.2(b). As an example, when we split edge e_{ab}, we create edges $e_{ak_1}, e_{k_1l_1}$, and e_{l_1b}. We also set $c_{ak_1} = -0.5, c_{k_1l_1} = 0$, and $c_{l_1b} = -0.5$.

The final transformation splits each vertex $v \in V'$ that has a self-loop into two vertices i' and i'', and the self-loop e_{ii} becomes edge $e_{i'i''}$. For each vertex i' and i'' that are created by this transformation, we set $b_{i'}$ and $b_{i''}$ to 1. An illustration of this transformation is given in Figure 6.4. Using a as an example, we create vertices a' and a''. We replace edge e_{ak_1} with edges $e_{a'k_1}$ and $e_{a''k_1}$, where the costs of these edges are the same as the cost of edge e_{ak_1}. We do the same thing to replace edge e_{ak_5}. For the self-loop, we replace it with edge $e_{a'a''}$ and set the cost to zero.

Once the transformation is complete, we find the MWPM in G'. We let M be this matching. We then get the total cost of the matching which is denoted as $c(M)$. If $c(M)$ is negative, then a negative cost cycle exists in G. If $c(M) \geq 0$, then G does not contain a negative cost cycle. We

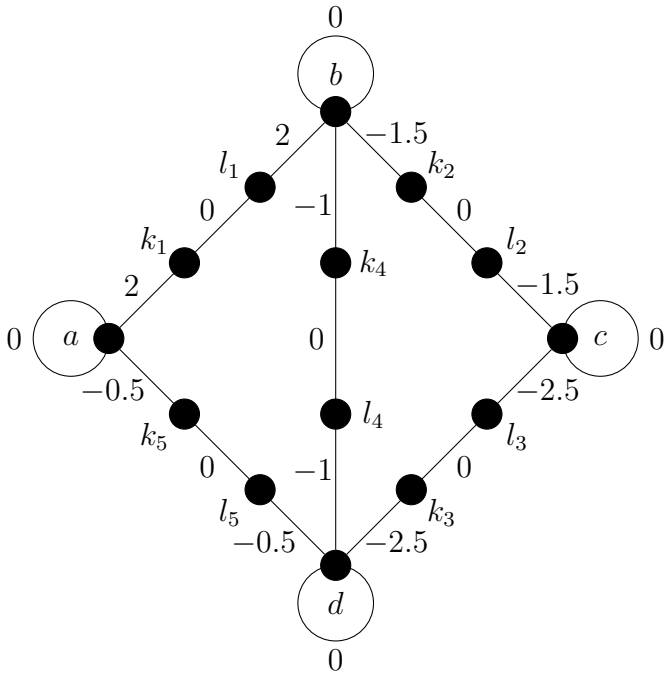


Figure 6.3: An example of the b-matching approach. The resulting graph G_2 after the second transformation.

provide an example of this step in Figure 6.5 using Gabow's algorithm [65]. Note that in this example, total cost of the matching is -10. Therefore, there must be a negative cost cycle in G.

The procedure is shown in Algorithm 6.1.

Resource Analysis

The first transformation (lines 2-6) adds an edge for each vertex in V. This means there are $|V| = n$ iterations, and each iteration takes constant time. Therefore, the first transformation takes $O(n)$ time.

The second transformation (lines 7-13) adds two vertices and one edge for each edge in E_1 that is not a self-loop. This means there are $|\{e_{ij} \in E_1 : i \neq j\}| = m$ iterations, and each iteration takes constant time. Hence, the second transformation takes $O(m)$ time.

In the third transformation (lines 14-24), for each vertex $i \in V_2$, where $b_i = 2$, we modify each edge outgoing of i except the self-loop e_{ii}. Since the only vertices that have $b_i = 2$ are the vertices in V, the number of outgoing edges from i in G_2 is equal to the number of outgoing edges from i in G. Recall that each undirected edge is connected to two vertices in V. This means there are

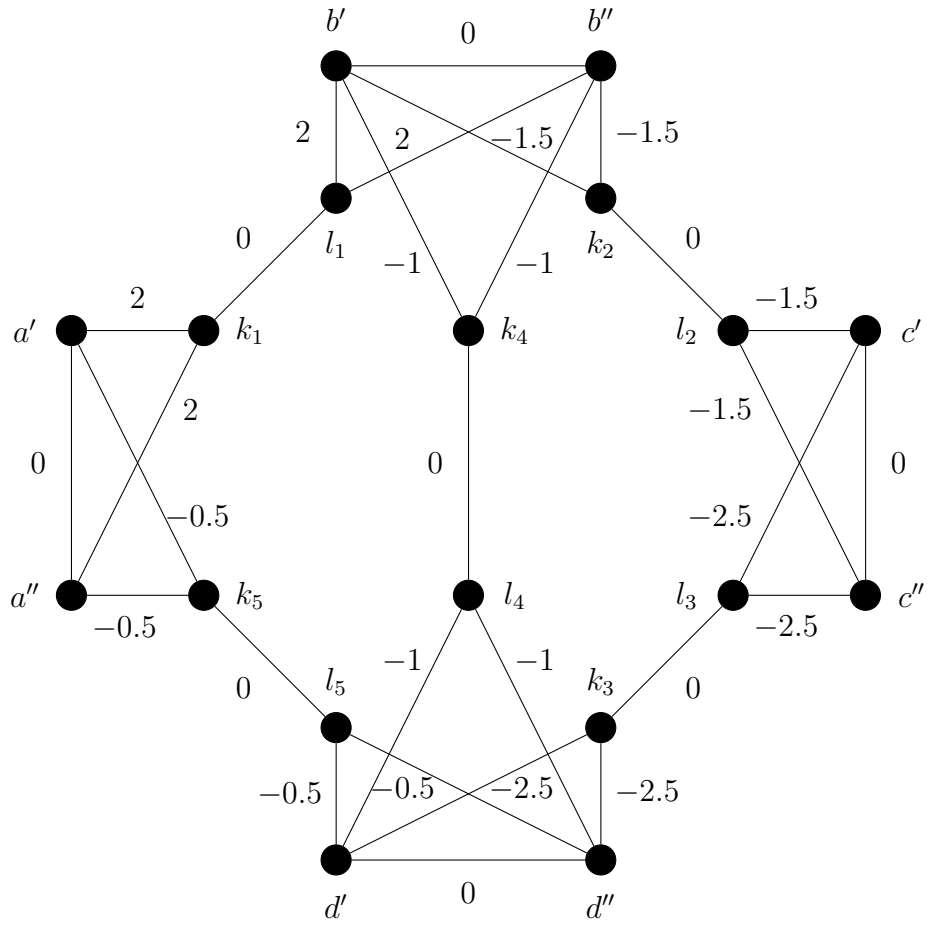


Figure 6.4: An example of the b-matching approach. The resulting graph G' after the third transformation.

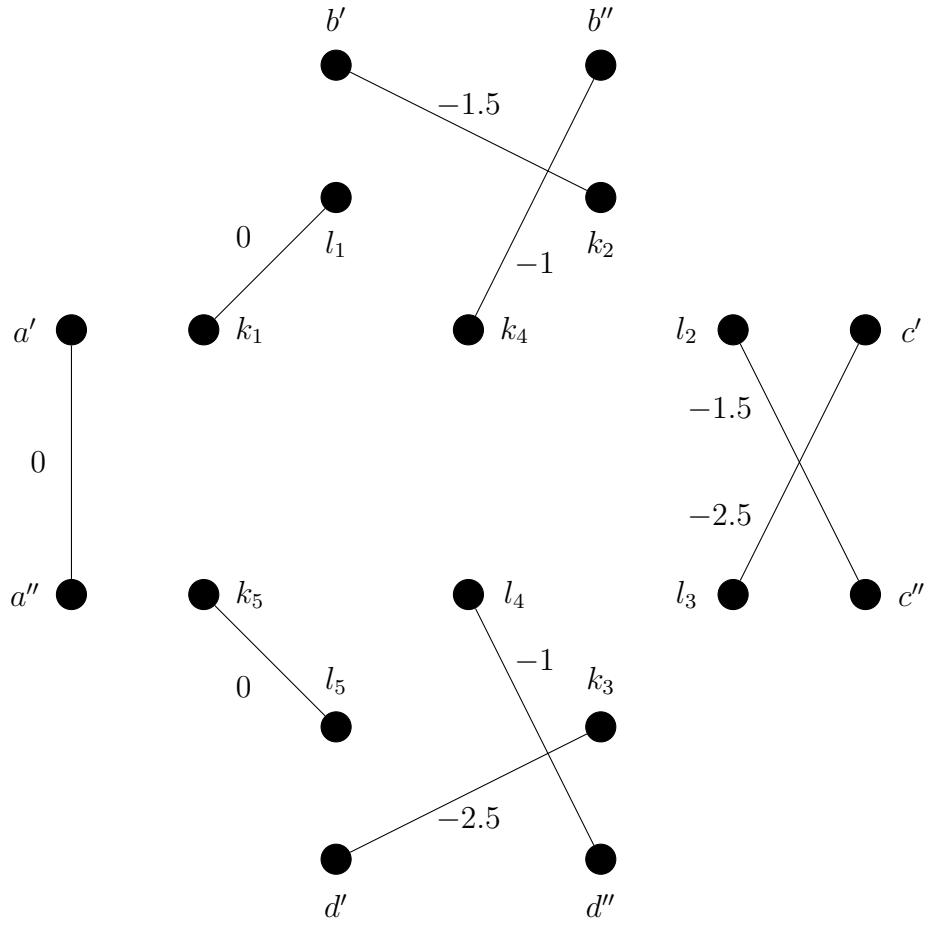


Figure 6.5: An example of the b-matching approach. The minimum weight perfect matching M.

Function b-MATCHING$(G = (V, E, c))$

Create graph $G' = (V', E')$, where $V' = V$ and $E' = E$.
for (each $v \in V'$) **do**
  Add edge e_{vv} to E'.
  $c(e_{vv}) := 0$.
  $b_v := 2$.
end for
for (each $e_{ij} \in E'$ and $i \neq j$) **do**
  Add vertices k and l to V'.
  Add edges e_{ik}, e_{kl}, and e_{lj} to E'.
  $c(e_{ik}) = c(e_{ij})/2$, $c(e_{lj}) = c(e_{ij})/2$, and $c(e_{kl}) = 0$.
  Remove e_{ij} from E'.
  $b_k := 1$; $b_l := 1$.
end for
for (each $i \in V'$ and $b_i = 2$) **do**
  Add vertices i' and i'' to V'.
  $b_{i'} := 1$; $b_{i''} := 1$.
  for (each $e_{ik} \in adj_{G'}(i)$ and $i \neq k$) **do**
    Remove edge e_{ik}, and add edges $e_{i'k}$ and $e_{i''k}$ to E'.
    $c(e_{i'k}) := c(e_{ik})$, and $c(e_{i''k}) := c(e_{ik})$.
  end for
  Remove edge e_{ii}, and add edge $e_{i'i''}$ to E'.
  $c(e_{i'i''}) := 0$.
  Remove vertex i from V'.
end for
Let M be an MWPM in G'.
if $(c(M) < 0)$ **then**
  return "G contains a negative cost cycle."
else
  return "G does not contain a negative cost cycle."
end if

Algorithm 6.1: UNCCD Algorithm: b-MATCHING

$2 \cdot m$ iterations, and each iteration takes constant time. Therefore, the third transformation takes $O(m)$ time.

The bottleneck of the algorithm is finding an MWPM in G' (line 25). From [65], we know that the MWPM takes $O(|V'| \cdot (|E'| + |V'| \cdot \log |V'|))$ time. This means we need to determine exactly what are $|V'|$ and $|E'|$.

Let us first determine $|V'|$. The first transformation does not add any new vertices since we add only self-loops. This means $|V_1| = n$. In the second transformation, we add two vertices for each edge that is not a self-loop. Since $|E| = m$, we add $2 \cdot m$ vertices to V_2, and $|V_2|$ is now $2 \cdot m + n$. In the third transformation, we split each vertex $i \in V_2$ with $b_i = 2$. Since these vertices are the original vertices in G, the number of vertices affected is n. Hence, we are adding n more vertices to V' during the split. Therefore, $|V'| = 2 \cdot m + n + n = 2 \cdot (m + n)$.

We now determine $|E'|$. The first transformation adds an edge for each vertex in V. This means we add $|V| = n$ edges, and $|E_1| = m + n$. In the second transformation, we replace each edge $e_{ij} \in E_1$, where $i \neq j$, with three new edges. This means we add $2 \cdot m$ edges to E_2, and $|E_2| = 3 \cdot m + n$. In the third transformation, we replace every edge $e_{ik} \in E_2$, such that $b_i = 2$, with two new edges. Since there are two such edges in E_2 for each edge $e_{ij} \in E$, the third transformation adds $2 \cdot m$ edges to E'. This means $|E'| = 5 \cdot m + n$.

From the analysis above, $|V'| = |E'| = O(m + n)$. Since the MWPM can be found in $O(|V'| \cdot (|E'| + |V'| \cdot \log |V'|))$ time, the total running time of the b-matching approach is $O((m + n) \cdot ((m + n) + (m + n) \cdot \log(m + n))) = O((m + n)^2 \cdot \log(m + n))$. Note that when $m = O(n^2)$, the b-matching algorithms runs in $O(n^4 \cdot \log n)$ time. This improves the $O(n^6)$ time analysis described by Gu et al. [23].

Correctness

To prove the correctness of the algorithm, we show the following:

1. After the first transformation, there exists a minimum weight perfect 2-matching H in G_1 with $c(H) < 0$ if and only if there exists a negative cost cycle in G.

2. After the second transformation, there exists a minimum weight perfect 2-matching H' in G_2 with $c(H') < 0$ if and only if there exists a minimum weight perfect 2-matching H in

G_1, where $c(H) < 0$ and $c(H') = c(H)$.

3. After the third transformation, there exists a perfect matching M in G' with $c(M) < 0$ if and only if there exists a minimum weight perfect b-matching H' in G_2, where $c(H') < 0$ and $c(H') = c(M)$.

Lemma 6.1.1 *After the first transformation, there exists a minimum weight perfect* 2*-matching* H *in* G_1 *with* $c(H) < 0$ *if and only if there exists a negative cost cycle in* G*.*

Proof: Let us assume there exists a negative cost cycle in G, denoted as $C = (V_C, E_C, c)$. Since $b_i = 2$ for each vertex i in V_1, and there is a loop e_{ii} with zero cost for each vertex i, there exists a perfect 2-matching with zero cost. Recall that a perfect 2-matching in G_1 implies a Hamilton cycle in G_1 or a set of vertex-disjoint cycles that covers all the vertices in V_1. If there are no negative cost cycles in G, then the 2-matching of zero cost in G_1 is minimum. Even if G contained edges with negative cost, they cannot be included in the 2-matching. Otherwise, the matching would not be perfect.

If there is a negative cost cycle C in G, then there must be a perfect 2-matching H in G_1 with negative cost. This is because the edges in C, denoted as E_C, with negative cost must be included in the matching. Let us assume this is not the case. In order for H to be a minimum perfect matching, the cost of each edge picked from G_1 must be either zero or negative. Suppose there exists an edge $e_{ij} \in E_C$ such that $c_{ij} < 0$ and $e_{ij} \notin H$. This means we can replace an edge in H with zero cost with edge e_{ij} to obtain a perfect matching with a smaller total cost. Therefore, H must contain the edges in the C that have a negative cost. As a result, $c(H) = c(E_C) + \sum_{i \notin V_C} c(e_{ii}) = c(E_C) + 0 = c(E_c)$. Hence, there exists a minimum weight perfect 2-matching H in G_1 with a negative cost (i.e., $c(H) < 0$) if and only if G contains at least one negative cost cycle C. □

Lemma 6.1.2 *After the second transformation, there exists a minimum weight perfect* 2*-matching* H' *in* G_2 *with* $c(H') < 0$ *if and only if there exists a minimum weight perfect* 2*-matching* H *in* G_1*, where* $c(H) < 0$ *and* $c(H') = c(H)$*.*

Proof: We show that the correspondence between H and H' arises from the following:

(i) Edge e_{ij} is in a b-matching H in G_1 if and only if e_{ik} and e_{lj} are in the b-matching H' in G_2, while $c(e_{ij}) = c(e_{ik}) + c(e_{lj})$.

(ii) Edge $e_{ij} \notin H$ if and only if $e_{kl} \in H'$, while $c(e_{kl}) = 0$.

The critical observation is that since $b_i = 2$, for any $i \in V_1$, vertex i either participates in two different edges in H, or it participates through loop $e_{ii} \in H$. Thus, for point (i), since $b_i = b_j = 2$ in G_1, $e_{ij} \in H$ implies that both i and j satisfy their degree constraints through edge e_{ij} and some other edge, say e_{ib} for vertex i and e_{jd} for vertex j, where $b, d \in V_1$. By matching i with k in H' instead of j in H (and with some node a in H' instead of b in H), the degree constraint for i is still satisfied. The same holds for j. Moreover, since $c(e_{ik}) = c(e_{lj}) = c(e_{ij})/2$, replacing e_{ij} in G_1 with e_{ik} and e_{lj} in G_2 sustains the total cost of the b-matching. The reverse can be shown similarly.

For point (ii), since $b_k = b_l = 1$ in G_2, $e_{kl} \in H'$ if only if $e_{ik} \notin H'$ and $e_{lj} \notin H'$. This means $e_{ij} \notin H$ (see point (i)), which proves our claim. Further, since $c(e_{kl}) = 0$, adding edge e_{kl} to G_2 sustains the total cost of the b-matching.

The above reasoning holds for any such transformation of a perfect b-matching in G_1 to a perfect b-matching in G_2. This is because the transformation sustains the cost of the perfect b-matchings. Note that points (i) and (ii) imply a one to one correspondence between the perfect b-matchings in G_1 and those in G_2. Hence, if H is minimum for G_1, H' is minimum for G_2 (and vice-versa), while $c(H) = c(H')$. □

Lemma 6.1.3 *After the third transformation, there exists a perfect matching M in G' with $c(M) < 0$ if and only if there exists a minimum weight perfect b-matching H' in G_2, where $c(H') < 0$ and $c(H') = c(M)$.*

Proof: Recall that the algorithm splits each vertex $i \in V_2$ with $b_i = 2$ into two vertices i' and i'' with $b_{i'} = b_{i''} = 1$. Hence, every $v \in V'$ has $b_v = 1$. Also, in G_2, each edge leaving vertex $i \in V_2$ with $b_i = 2$ is either (i) towards some vertex k with $b_k = 1$ (from the second transformation), or (ii) towards itself (from the first transformation).

We first examine case (i). Let vertex $i \in V_2$ be matched with vertices k, a in H' (i.e., $e_{ik}, e_{ia} \in H'$). The correspondence of H' and M is sustained by matching i' to k and i'' to a in M, or vice versa, by having $e_{i'k}, e_{i''a} \in M$ (or $e_{i''k}, e_{i'a} \in M$). Note that the degree constraints are

satisfied since $b_{i'} = b_{i''} = b_k = b_a = 1$. Moreover, since $c(e_{i'k}) = c(e_{ik})$ and $c(e_{i''a}) = c(e_{ia})$, the transformation sustains the total cost of the matching.

For case (ii), let vertex $i \in V_2$ be matched with itself in H' (i.e., $e_{ii} \in H'$). The correspondence of H' and M is sustained by matching i' to i'' by $e_{i'i''} \in M$. The degree constraints are satisfied since $b_{i'} = b_{i''} = 1$, while the transformation sustains the total cost of the corresponding matching, since $c(e_{i'i''}) = c(e_{ii}) = 0$. The reverse can be proved in the same manner. □

By Lemmas 6.1.1-6.1.3, if there exists a negative cycle C in G, then there exists an MWPM M in G' such that $c(M) < 0$. Hence, the algorithm correctly identifies the existence of a negative cost cycle in G if $c(M) < 0$.

6.2 The T-join Approach

In this section, we describe the algorithm that solves the UNCCD problem for an undirected graph G by utilizing T-joins. Briefly, the algorithm utilizes the reduction [20] from the minimum weight T-join problem with non-negative weights [19] to the UNCCD problem. We perform the reduction by first obtaining a special graph of G, where the cost of each edge is non-negative. We next solve an APSP problem to get the metric closure of the new graph. We then find the MWPM of a specific induced subgraph to get a minimum weight T-join. With this T-join, we can determine if there exists a negative cost cycle in G. The details of the algorithm are explained below.

6.2.1 Preliminaries

We are given a graph $G = (V, E)$. For the ease of exposition, we let G be uncapacitated (i.e., each edge $e_{ij} \in E$ has capacity equal to 1). Let $adj_G(v)$ denote the set of edges adjacent to vertex v in a graph G. We define a T-join [20] as follows:

Definition 6.2.1 *Given an undirected graph $G = (V, E)$ and a set $T \subseteq V$ of even cardinality, a set $J \subseteq E$ is a T-join if $|J \cap adj_G(v)|$ is odd if and only if $v \in T$.*

If $T = \emptyset$, then we refer the T-join as an $\emptyset$-*join*. Also, if $T = \emptyset$, then every vertex $v \in V$ has to be incident to an even number of edges (or no edges) in J [58]. Therefore, an $\emptyset$-join is either an

edge set of one or more cycles in G or the empty set $\emptyset$.

Figure 6.6 provides several examples of a T-join. We are given an undirected graph G, as shown in Figure 6.6(a). Let $T_1 = \{c, d\}$. Recall that $adj_G(c) = \{e_{bc}, e_{cd}\}$ and $adj_G(d) = \{e_{ad}, e_{bd}, e_{cd}\}$. The set of edges $J_1 = \{e_{cd}\}$ is ia T_1-join in G since $|T_1|$ is even, and both $|J_1 \cap adj_G(c)|$ and $|J_1 \cap adj_G(d)|$ are odd. Likewise, $J_2 = \{e_{bd}, e_{bc}\}$ is also a T_1-join since $|J_2 \cap adj_G(c)|$ and $|J_2 \cap adj_G(d)|$ are odd. Let $T_2 = \emptyset$. Then, $J_3 = \{e_{da}, e_{ab}, e_{bc}, e_{cd}\}$, $J_4 = \{e_{bd}, e_{dc}, e_{db}\}$, and $J_5 = \emptyset$ are three possible T_2-joins or, equivalently, $\emptyset$-joins.

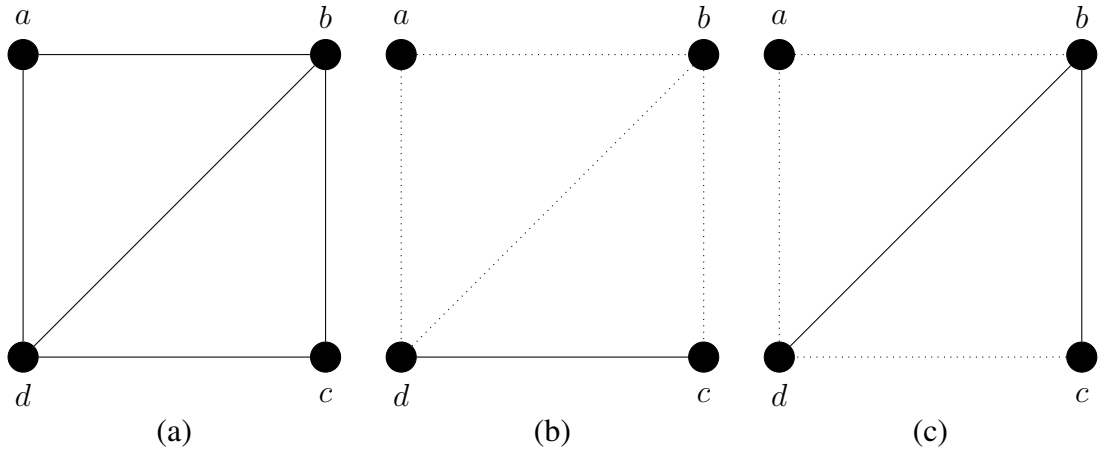


Figure 6.6: Examples of a T-join. (a) is the undirected graph G. (b) is a T_1-join, where $T_1 = \{c, d\}$ and $J_1 = \{e_{cd}\}$. (c) is also a T_1-join, where $T_1 = \{c, d\}$ and $J_2 = \{e_{bd}, e_{bc}\}$.

The problem of finding a *minimum c-weight T-join* in G, where c represents the cost function of graph G, can be reduced to the MWPM problem when G has edges with only positive weights [19], edges with both positive and negative weights but no negative cost cycles [58], and even negative cost cycles [20]. The fastest known algorithm for finding the minimum c-weight T-join runs in $O(n^3)$ time [20].

6.2.2 UNCCD Algorithm based on T-join

The crucial observation for the T-join approach is formalized by the following theorem:

Theorem 6.2.1 *There is a negative cost cycle in $G = (V, E, c)$ if and only if J is the minimum c-weight $\emptyset$-join with $c(J) < 0$.*

Proof: Recall that an $\emptyset$-join is either an edge set of one or more edge-disjoint cycles in G or the empty set $\emptyset$. Also, the weight of an $\emptyset$-join $J_\emptyset = \emptyset$ is $c(J_\emptyset) = 0$. Hence, if there are no negative cost cycles in G, $J_\emptyset$ will always be a minimum c-weight $\emptyset$-join. Otherwise, the existence of at least one negative cost cycle in G implies that there also exists at least one $\emptyset$-join J' with $c(J') < 0$. This means the minimum c-weight $\emptyset$-join J will have $c(J) \leq c(J') < 0$.

The reverse follows from the fact that the minimum c-weight $\emptyset$-join J has $c(J) < 0$ and corresponds to $t \geq 1$ cycles in G. Among those cycles, there must exist at least one with negative cost. Otherwise, $c(J) \geq 0$, which is a contradiction to $c(J) < 0$. □

Therefore, to determine if the graph has a negative cost cycle, we need to find a minimum c-weight $\emptyset$-join J. Thus, a negative cost cycle exists in G if and only if $c(J) < 0$.

The algorithm starts by finding the set of edges with negative cost, which is denoted as $E^- \subseteq E$, and the set of vertices that are incident to an *odd* number of negative edges, which we denote as $T^- \subseteq V$. We construct graph $G_d = (V, E, d)$, with cost function $\mathbf{d} : E \to \mathbb{R}^+$ such that for any edge $e_{ij} \in E$, $d(e_{ij}) = |c(e_{ij})|$. In other words,

$$d(e_{ij}) = \begin{cases} -c(e_{ij}) & e_{ij} \in E^- \\ c(e_{ij}) & e_{ij} \in E \setminus E^-. \end{cases}$$

We now provide an example of this step. We will use the same graph G in Figure 6.2(a), which we show in Figure 6.7(a). We first need to determine sets E^- and T^-. In this case, $E^- = \{e_{bc}, e_{cd}, e_{bd}\}$, and $T^- = \{a, c\}$. The next step is to construct graph G_d by setting the cost of each edge in G_d to the absolute value of the cost of the corresponding edge in G. The resulting graph is provided in Figure 6.7(b).

The next step is to create the *metric closure* $\bar{G}_d = (V, E_{\bar{G}_d}, \bar{d})$ of G_d (Definition 5.1.1) by solving an APSP problem. We then construct the induced subgraph $\bar{G}_d[T^-] = (T^-, E_{\bar{G}_d[T^-]}, \bar{d})$. The algorithm then proceeds with finding an MWPM M in $\bar{G}_d[T^-]$. Using the example from Figure 6.7, the metric closure of G_d is given in Figure 6.8(a). Since a and c are the only vertices in T^-, we get the induced subgraph given in Figure 6.8(b). The next step is to find a minimum weight perfect matching M in $\bar{G}_d[T^-]$. Since $\bar{G}_d[T^-]$ contains only a single edge, in this case, $M = E_{\bar{G}_d[T^-]} = \{e_{ad}\}$.

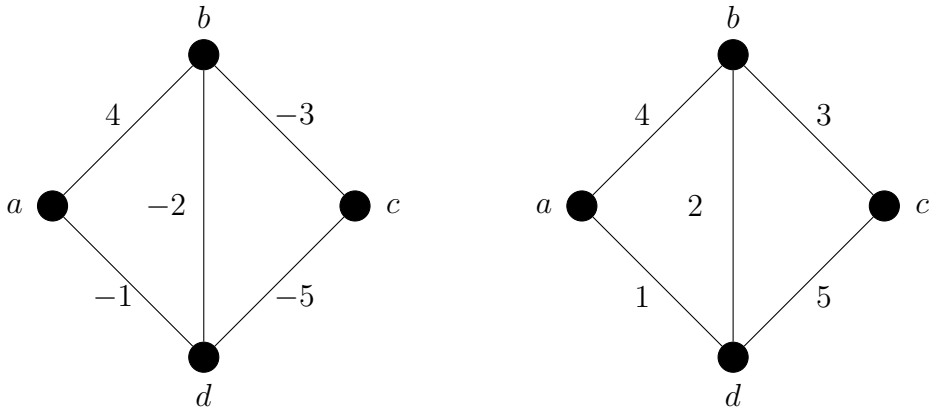


Figure 6.7: An example of the T-join approach. (a) is the initial graph G. (b) is the modified graph G_d.

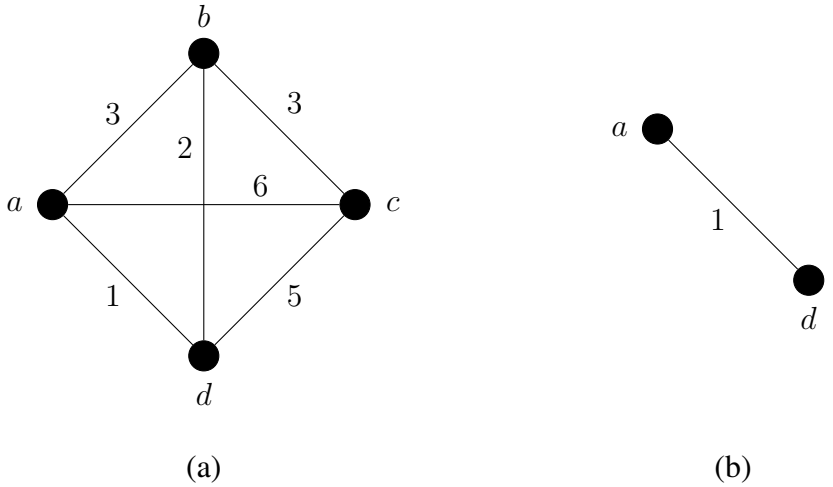


Figure 6.8: An example of the T-join approach. (a) is the metric closure $\bar{G}_d$. (b) is the induced subgraph $\bar{G}_d[T^-]$.

To detect the presence of a negative cost cycle (lines 7-13), let J be the minimum c-weight $\emptyset$-join of G, and let J_d be the minimum d-weight T^--join of G_d. Also, let $\bar{P}_{ij} = (V_{\bar{P}_{ij}}, E_{\bar{P}_{ij}})$ denote the shortest path between i and j in G_d. The algorithm first identifies J_d, which is the symmetric difference of the sets of edges of each shortest path from i to j, where $e_{ij} \in M$ (denoted as $\triangle_{e_{ij} \in M} E_{\bar{P}_{ij}}$). It then calculates the total cost of J as $c(J) = d(J_d) + c(E^-)$, where $d(J_d)$ is the cost of T^--join J_d, and $c(E^-)$ is the cost of all edges in E^-. If $c(J) < 0$, then G contains a negative cost cycle. If $c(J) \geq 0$, then G does not contain any negative cost cycles.

From our example in Figure 6.8, we need to find the symmetric difference of the edges in M and the edges involved in the shortest paths between all pairs of vertices with edges in M. In this example, e_{ad} is the only edge in M, and the shortest path from a to d in G_d contains a single edge, e_{ad}. By definition, this means $J_d = \emptyset$ since e_{ad} is in both sets. We then need to compute $c(J)$, which gives us $c(J_d) + c(E^-) = 0 + (-2 + -3 + -5) = -10$. Since $c(J) < 0$, G must contain a negative cost cycle.

The procedure is shown in Algorithm 6.2.

Function T-JOIN$(G = (V, E, c))$
1: $E^- = \{e_{ij} \in E : c(e_{ij}) < 0\}$.
2: $T^- = \{v \in V : |adj_G(v) \cap E^-|$ is odd $\}$.
3: Create graph $G_d = (V, E, d)$.
4: **for** (each $e_{ij} \in E$) **do**
5: $\quad d(e_{ij}) := |c(e_{ij})|$.
6: **end for**
7: $\bar{G}_d =$ APSP in G_d.
8: Create induced subgraph $\bar{G}_d[T^-] = (T^-, E_{\bar{G}_d[T^-]}, \bar{d})$.
9: $M =$ MWPM in $\bar{G}_d[T^-]$.
10: $J_d = \triangle_{e_{ij} \in M} E_{\bar{P}_{ij}}$.
11: $c(J) = d(J_d) + c(E^-)$.
12: **if** $c(J) < 0$ **then**
13: $\quad$ **return** "G contains a negative cost cycle".
14: **else**
15: $\quad$ **return** "G does not contain a negative cost cycle".
16: **end if**

Algorithm 6.2: UNCCD Algorithm: T-JOIN

Resource Analysis

Determining E^- takes $O(m)$ time since we need to check all $|E| = m$ edges. To find T^-, we need to scan all $|V| = n$ vertices and their adjacent edges, which means m edges total, to determine which vertices have an odd number of adjacent edges with negative cost. This means we need $O(m + n)$ time. Constructing graph G_d also takes $O(m + n)$ time since each vertex $v \in V$ and edge $e_{ij} \in E$, where $d(e_{ij}) = |c(e_{ij}|$, is added to G_d. Note that all edges in G_d are non-negative. This means we can solve the APSP problem in G_d using $O(n)$ iterations of Dijkstra's algorithm [38], which takes $O(m + n \cdot \log n)$ time per iteration. Therefore, the APSP component takes $O(n \cdot (m + n \cdot \log n))$ time [76, 30].

As a result of the APSP computation, the metric closure of G_d, $\bar{G}_d$, has $|V_{\bar{G}_d}| = n$ vertices but $|E_{\bar{G}_d}| = m' \geq m$ edges. This is because each edge in $\bar{G}_d$ is a path from one vertex to another in G_d. Since each pair of vertices in G_d may have at most one path between them, $m' = O(n^2)$. Therefore, constructing $\bar{G}_d[T^-]$ takes $O(m' + n) = O(n^2 + n) = O(n^2)$ time since we need to scan each vertex and edge in G_d. Using Gabow's algorithm [65], finding a minimum weight perfect matching in $\bar{G}_d[T^-]$ takes $O(n \cdot (m' + n \cdot \log n)) = O(n \cdot (n^2 + n \cdot \log n)) = O(n^3)$ time.

To find the symmetric difference J_d, we need to examine $n/2$ paths. This is because G_d is undirected, which means the path from i to j is the same path from j to i, so we do not need to examine the latter path for each i and j. Since each path has at most n edges, we need $O(n^2)$ time. Computing $c(J)$ takes $O(n^2)$ time since computing $d(J_d)$ takes $O(n^2)$ time, and computing $c(E^-)$ takes $O(m)$ time.

Therefore, the total running time of the algorithm is $O(m+n+n \cdot (m+n \cdot \log n)+n^2+n^3) = O(n^3)$.

Correctness

To prove correctness of Algorithm 6.2, we first need to use a key theorem from [20]. Note that for any two sets of edges E_1 and E_2, $E_1 \triangle E_2$ denotes their symmetric difference.

Theorem 6.2.2 *Consider an undirected, weighted graph G with cost function $c : E \to \mathbb{R}$, and $T \subseteq V$ with $|T|$ even. Moreover, consider sets E^- and T^- (as defined above) and a cost function $c^+ : E \to \mathbb{R}^+$ such that $c^+(e_{ij}) = |c(e_{ij})|$, for each $e_{ij} \in E$.*

(i) For any subset J of E, $c(J) = c^+(J \triangle E^-) + c(E^-)$.

(ii) J is a minimum c-weight T-join if and only if $J \triangle E^-$ is a minimum c^+-weight $(T \triangle T^-)$-join.

By Theorem 6.2.2(ii), J is a minimum c-weight T-join if and only if $J \triangle E^-$ is a minimum c^+-weight $(T \triangle T^-)$-join. In order to find a minimum c-weight $\emptyset$-join of G, we need to find a minimum d-weight $(\emptyset \triangle T^-)$-join or, equivalently, a minimum d-weight T^--join of G_d.

The algorithm starts by constructing sets E^- and T^-. We then construct the graph $G_d = (V, E, d)$ with cost function $\mathbf{d}$ such that $d(e_{ij}) = |c(e_{ij})|$, for each $e_{ij} \in E$. Since G_d is an undirected graph with only non-negative weights, we can use Edmonds' algorithm [19] to correctly identify a minimum d-weight T^--join of G_d.

The algorithm then creates the metric closure $\bar{G}_d$ of G_d. Since G_d is undirected, creating the metric closure $\bar{G}_d = (V, E_{\bar{G}_d}, \bar{d})$ is equivalent to replacing each edge $e_{ij} \in E_{G_d}$ with a pair of oppositely directed edges and then solving the APSP problem on the new graph [20]. Since the APSP algorithm is correct, we know constructing the metric closure is also correct.

The algorithm then constructs $\bar{G}_d[T^-]$ by including all vertices in T^- and all edges of $\bar{G}_d$ that connect any two vertices $i, j \in T^-$. We then find the MWPM M in $\bar{G}_d[T^-]$ using Gabow's algorithm [65], which we know correctly finds an MWPM. Finally, it identifies a minimum d-weight T^--join of G_d, which we denote as J_d, by obtaining the symmetric difference of the edge sets of all shortest paths between i and j such that $e_{ij} \in M$.

Since J_d is a minimum d-weight T^--join of G_d, by Theorem 6.2.2(ii), $J_d = J \triangle E^-$, where J is a minimum c-weight $\emptyset$-join of G. Hence, by Theorem 6.2.2(i), the weight of J can be calculated as $c(J) = d(J_d) + c(E^-)$. By Theorem 6.2.1, the algorithm correctly reports that there exists a negative cost cycle if and only if $c(J) < 0$.

6.3 Improved UNCCD Algorithms for Integer Edge Costs

In this section, we are concerned with the UNCCD problem, where the edge costs are integers in the range $\{-K \cdots K\}$, and K is a fixed constant. Algorithms 6.1 and 6.2 utilize known procedures for solving the MWPM and the minimum weight T-join problems. As a result, the b-matching approach runs in $O((m+n)^2 \cdot \log(m+n))$ time, and the T-join approach runs in $O(n^3)$ time.

However, there exist other algorithms that are more efficient for graphs with integer edge costs. By using these algorithms, we can design new UNNCD algorithms with improved running times. We discuss the improvements for both approaches below.

6.3.1 The Improved b-matching Approach

From Chapter 6.1.2, we know that the b-matching approach detects a negative cost cycle, assuming one exists, in a graph G with n vertices and m edges in $O((m+n)^2 \cdot \log(m+n))$ time. This is achieved by using Gabow's algorithm [65] to solve the corresponding MWPM problem. Recall that Gabow's algorithm runs in $O(n \cdot (m + n \cdot \log n))$ time. For integer edge costs, we can use Gabow and Tarjan's [77] MWPM algorithm instead of Gabow's original algorithm. This algorithm runs in $O(m \cdot \log(N \cdot n) \cdot \sqrt{n \cdot \alpha(m,n) \cdot \log n})$ time, where N is the largest absolute value of the edge costs, and $\alpha(m,n)$ is the inverse Ackermann function [21, 22].

To use a more efficient algorithm for a graph G with integer edge costs, we first need to perform a simple preprocessing step for the b-matching approach. Recall that during the transformation from G_1 to G_2, for any edge $e_{ij} \in V$, we create edges e_{ik} and e_{lj} with cost $c(e_{ik}) = c(e_{lj}) = c(e_{ij})/2$. Although $c(e_{ij})$ is integral, $c(e_{ik})$ and $c(e_{lj})$ may not. This means G' may contain edges that do not have integer edge costs, and thus, we cannot use Gabow and Tarjan's algorithm. To resolve the issue, we can transform G by multiplying the cost of each edge by 2. With this change, the edge costs of the new graph, say $G^* = (V^*, E^*)$, will be even integers in the range $\{-2 \cdot K \cdot \cdot 2 \cdot K\}$. Modifying the edges takes $O(m)$ time since we examine each edge in G. Although we double the edge costs, all negative cost cycles are preserved. Moreover, the edge costs of G' are now integers in the range $\{-K \cdot \cdot K\}$, since the cost of each edge $e_{ij} \in E^*$ is divided by 2.

After the preprocessing step, we transform G into G', as stated in the algorithm, with $2 \cdot (m+n)$ vertices and $5 \cdot m + n$ edges. Recall that $|V'|$ and $|E'|$ are both $O(m+n)$. We then find an MWPM M in G'. Since the running time of the b-matching algorithm is dominated by the MWPM procedure, we can improve the overall running time of the algorithm by using a more efficient MWPM algorithm. Using Gabow and Tarjan's [77] algorithm, the new running time of the b-matching algorithm is $O((m+n)^{1.5} \cdot \log(K \cdot (m+n)) \cdot \sqrt{\alpha(m+n, m+n) \cdot \log(m+n)})$,

where $K = N$, and $\alpha(m+n, m+n)$ is the inverse Ackermann function. Since K is a fixed constant (i.e., $K = O(1)$), the b-matching approach for a graph with integer edge costs runs in $O((m+n)^{1.5} \cdot (\log(m+n))^{1.5} \cdot \sqrt{\alpha(m+n, m+n)})$ time.

Note that when $m = O(n^2)$, the b-matching algorithm solves the UNCCD problem with fixed integer edge costs in $O(n^3 \cdot (\log n)^{1.5} \cdot \sqrt{\alpha(n^2, n^2)})$ time. When the graph is sparse (i.e., $m = O(n)$), the running time is $O(n^{1.5} \cdot (\log n)^{1.5} \cdot \sqrt{\alpha(n, n)})$. Thus, we improve the previous fastest running time in [23], which was $O(n^{2.75} \cdot \log n)$ time using the T-join approach.

6.3.2 The Improved T-join Approach

From Chapter 6.2.2, we know that the T-join approach detects a negative cost cycle, assuming one exists, in a graph G with n vertices and m edges in $O(n^3)$ time. This is accomplished by using an APSP procedure that takes $O(n \cdot (m + n \cdot \log n))$ time [38, 76, 30] and Gabow's MWPM algorithm that takes $O(n \cdot (m' + n \cdot \log n)) = O(n \cdot (n^2 + n \cdot \log n)) = O(n^3)$ time, where $m' = O(n^2)$ is the number of edges in $\bar{G}_d[T^-]$.

To improve the running time of computing the APSP for a graph with integer edge costs, we can use Shoshan and Zwick's algorithm [78] instead of running $O(n)$ Dijkstra computations. This algorithm runs in $\tilde{O}(N \cdot n^\omega) = O(n \cdot n^\omega \cdot (\log n)^t)$ time, where N is the largest absolute value of the edge costs, $\omega < 2.376$ is the exponent for the fastest known matrix multiplication algorithm [79], and t is some constant. For computing the MWPM, we can also use Gabow and Tarjan's algorithm [77] from Chapter 6.3.1 that runs in $O(m \cdot \log(N \cdot n) \cdot \sqrt{n \cdot \alpha(m, n) \cdot \log n})$ time, where N is the largest absolute value of the edge costs, and $\alpha(m, n)$ is the inverse Ackermann function.

When we apply Shoshan and Zwick's algorithm, recall that we are finding the metric closure of G_d which is denoted as $\bar{G}_d$. Since $|V_{G_d}| = n$, the APSP computation takes $O(K \cdot n^{2.376} \cdot (\log n)^t)$ time, where $K = N$, and t is some constant. When we apply Gabow and Tarjan's algorithm, recall that we are finding an MWPM in $\bar{G}_d[T^-]$. Since $|V_{\bar{G}_d[T^-]}| = n$, $|E_{\bar{G}_d[T^-]}| = O(n^2)$, and the maximum edge cost is $n \cdot K$, the MWPM procedure runs in $O(n^2 \cdot \log(n \cdot K \cdot n) \cdot \sqrt{n \cdot \alpha(n^2, n) \cdot \log n}) = O(n^{2.5} \cdot \log(K \cdot n^2) \cdot \sqrt{\alpha(n^2, n) \cdot \log n})$ time.

Both algorithms dominate the running time of the T-join approach. Therefore, the new

running time is $O(K \cdot n^{2.376} \cdot (\log n)^t + n^{2.5} \cdot \log(K \cdot n^2) \cdot \sqrt{\alpha(n^2, n) \cdot \log n})$. Since K is a fixed constant (i.e., $K = O(1)$), the T-join approach for a graph with integer edge costs runs in $O(n^{2.5} \cdot (\log n)^{1.5} \cdot \sqrt{\alpha(n^2, n)})$ time. Thus, we improve the previously known bound for the UNCCD problem for graphs with fixed integer edge costs which was $O(n^{2.75} \cdot \log n)$ time [23] using the T-join approach.

Chapter 7

Implementation Profile for the UNCCD Problem

In this chapter, we profile the b-matching and T-join approaches for the UNCCD problem. In this study, we focus on graphs where the edge costs are integers in the range $\{-K \cdot \cdot K\}$, and K is a fixed constant. Recall from Chapter 6.3 that the b-matching approach runs in $O((m+n)^{1.5} \cdot \log(K \cdot (m+n)) \cdot \sqrt{\alpha(m+n, m+n) \cdot \log(m+n)})$ time, while the T-join approach runs in $O(K \cdot n^{2.376} \cdot (\log n)^t + n^{2.5} \cdot \log(K \cdot n^2) \cdot \sqrt{\alpha(n^2, n) \cdot \log n})$ time, where $\alpha(x, y)$ is the inverse Ackermann function [21, 22], and t is some constant.

7.1 Implemented Algorithms

We describe the implementation details of our study by first discussing the algorithms that were implemented. Both UNCCD algorithms require the use of an MWPM algorithm. In fact, both approaches in this study reduce the UNCCD problem to the MWPM problem by making modifications to the graph. Specifically, the b-matching approach does this by increasing the number of vertices and edges, while the T-join approach accomplishes this by solving the APSP problem in order to obtain the metric closure of graph G_d. To solve the MWPM problem, we utilize an implementation of the Blossom V algorithm [56] (the source code is available online at [80]). We chose this implementation because it is currently one of the most efficient MWPM implementations available.

All Pairs Shortest Paths for the T-join Approach

Recall that the T-join approach first solves the APSP problem as part of the graph modification required for the reduction. For this problem, we utilize an efficient single source shortest path implementation of Goldberg's algorithm [81] and run it n times, one for each source. This algorithm runs in linear time in the average case when the edge costs are uniformly distributed. In the worst case, the algorithm runs in $O(m + n \cdot \log N)$, where N is the largest edge cost. Although there exist other shortest path algorithms that are asymptotically faster, such as [82] which runs in $O(m + n \cdot \sqrt{\log N})$ time, Goldberg's implementation is one of the currently known fastest implementations available and has been used as the reference solver for the 9th DIMACS Shortest Path Implementation Challenge [50].

Even though Shoshan and Zwick's algorithm [78] has the best known time bound for solving the APSP problem for a graph with integer edge costs, we chose to omit the algorithm from our empirical study. This is because several issues arise when implementing the algorithm that impact its efficiency. Recall that the running time of the algorithm depends on the matrix multiplication algorithm used. Specifically, it is assumed that the algorithm of [79] is used (hence, the $O(n^{2.376})$ time bound). However, although theoretically efficient, this matrix multiplication algorithm is not practical to implement; it provides an advantage only for matrices that are too large for modern hardware to handle [83]. Note that since then, [84] designed a faster matrix multiplication algorithm that runs in $O(n^{2.3727})$. However, we encounter the same problem, where the algorithm is efficient for matrices that are too large for modern hardware.

We then explored Strassen's matrix multiplication algorithm [85], which runs in $O(n^{2.8074})$ time. Although this is faster than the $O(n^3)$ matrix multiplication algorithm, we cannot directly use it in Shoshan and Zwick's algorithm. This is because the algorithm actually uses matrix multiplication over the closed semi-ring $\{\min, +\}$, which is known as "funny matrix multiplication" or the "distance product" in the literature. Note that the sum operation is equivalent to the $\min$ operation in "funny matrix multiplication." However, unlike regular matrix multiplication, there is no inverse for the $\min$ operation, which is required to make Strassen's algorithm work.

[86] provides an approach for encoding a distance matrix such that regular matrix multiplication works. This involves using the values in the distance matrix as exponents to the values in the new

matrix. The problem with this conversion is that even if the exponents are small, the values are too large to store in modern hardware, as we have discovered when implementing the algorithm. This means the $O(n^3)$ matrix multiplication algorithm is currently the only practical algorithm to use for implementing Shoshan and Zwick's algorithm. Since this results in the algorithm running in $O(n^3)$ time, compared to $O(m \cdot n + n^2 \cdot \log N)$ time when running Goldberg's algorithm n times, including it in our study would not provide sufficient results.

7.2 Graph Families

Our experiments study the performance of these algorithms on graphs with varying parameters. We study three graph families that are produced by two generators. These families are chosen because they are natural and have been used in previous empirical studies [11, 48]. The graph generators used are part of the 9th DIMACS Shortest Path Implementation Challenge benchmark package [50].

The first generator (SPRAND [51]) creates random graphs with n vertices and $m \geq n$ edges. The generator first constructs a Hamiltonian cycle to ensure the graph is connected. The remaining $m - n$ edges are added by randomly selecting a pair of distinct vertices. Note that the generator can produce parallel edges and/or self-loops. The costs of all edges are selected uniformly and at random among a predetermined interval.

The second generator is called TOR, which originates from the SPGRID generator [51]. These graphs are similar to grid (mesh) graphs, in that they produce two dimensional $x \times y$ grids. The difference is that torus graphs are embedded on a plane, so they "wrap" around. With this generator, we create two types of families: layered torus graphs and square torus graphs. For both families, the costs of all edges are selected uniformly and at random from a predetermined interval.

Layered torus graphs consist of x layers, where each layer contains y vertices. The vertices of a single layer form a simple cycle and additional edges are included by randomly selecting pairs of vertices within the layer. Neighboring layers are connected by edges such that one vertex is in the first layer, and the second vertex is in the other layer. Note that the last layer and the first layer are also connected by edges whose endpoints come from these two layers. This is how we get the graph to "wrap" around. In our experiments, we set $x = \frac{n}{25}$ and $y = 25$. Square torus graphs are

Table 7.1: Negative Cost Cycle Categories

Category	Number of Cycles	Size
No Negative Cost Cycles	0	0
One Small Negative Cost Cycle	1	3
Many Small Negative Cost Cycles	x	3
Few Medium Negative Cost Cycles	10	x
One Hamiltonian Negative Cost Cycle	1	n

very similar to layered torus graphs, except $x = y = \sqrt{n}$. Note that square torus graphs are also known as grid torus graphs in the literature [11].

The TOR generator is also capable of controlling the number and size of negative cost cycles by including additional parameters. The SPRAND generator is not capable of this feature, but it can still produce negative cost cycles based on the predetermined edge cost interval used. When adding a negative cost cycle to a torus graph, all edges except for one have cost zero. The remaining edge has cost -1. In our study, we use the five categories that were used in Goldberg's study [11]. These categories are provided in Table 7.1.

7.3 Experimental Setup

All implemented UNCCD algorithms are written in C/C++, and they are compiled and run in identical experimental settings. We use the adjacency matrix to represent the graph and to compute the MWPM. This is because we compute the all pairs shortest paths for the T-join approach, and both the costs and the shortest paths are required for computing the symmetric difference. We use the array data structure to store the resulting MWPM. For random graphs, the range of the edge costs is between $-K$ and K, where K is the largest edge cost. Although simple heuristics can be used for detecting negative cost cycles with this range, our empirical study assumes only the value of K and does not assume the distribution of the edge costs. For torus graphs, the range of the edge costs is between 1 and K, except for the negative cost cycles added. Our testing platform is a 3.00 GHz 64-bit AMD Phenom II $X4$ 940 quad core machine with 8 GB RAM and a 64 MB cache which runs Ubuntu (version 12.04). The implementations are compiled with the Intel C++ compiler (icpc) version 12.0, and the optimization flag is set to -O3. We report the average

execution time of ten independent trials for each test.

7.4 Results and Analysis

We compare the performance of both the b-matching and T-join algorithms based on the following domains:

(i) Suite A: Number of vertices (Figures 7.1-7.3 and Table 7.2).

(ii) Suite B: Number of edges (Figures 7.4-7.6 and Table 7.3).

(iii) Suite C: Size of K (Figures 7.7-7.9 and Table 7.4).

(iv) Suite D: Negative cost cycles (Figures 7.10-7.12 and Tables 7.5 and 7.6).

Each figure mentioned above plots the execution time for each algorithm and graph. Each algorithm displays three sets of data. For the T-join approach, we provide the time to compute the all pairs shortest paths (T-APSP), the time to compute the minimum weight perfect matching (T-MWPM), and the total time of the algorithm (T-Total). For the b-matching approach, we provide the time to transform the graph (B-Trans), the time to compute the minimum weight perfect matching (B-MWPM), and the total time of the algorithm (B-Total). For all figures, the execution time on the Y-axis is depicted in the logarithmic scale due to the variation of the data collected.

For each experiment, we provide the complete data in its respective table. Each entry in the table consists of the average execution time (the top number) and the standard deviation (the bottom number). Note that an entry with the value 0.000 means the value is less than 0.001.

7.4.1 Number of Vertices

We first compare the performance of the b-matching and T-join algorithms as we vary the number of vertices. In this study, we set the number of vertices to $1000, 4000, 8000, 12000, 16000,$ and 20000. We set the number of edges to $3 \cdot n$. This means the number of edges in the graphs studied are $3000, 12000, 24000, 36000, 48000$, and 60000. For random graphs, the range of edge

costs is between -512 and 512. For torus graphs, the range of edge costs is between 1 and 512, except for the negative cost cycles. Finally, for torus graphs, we have 10 negative cost cycles, each containing x edges, where x is defined in Chapter 7.2.

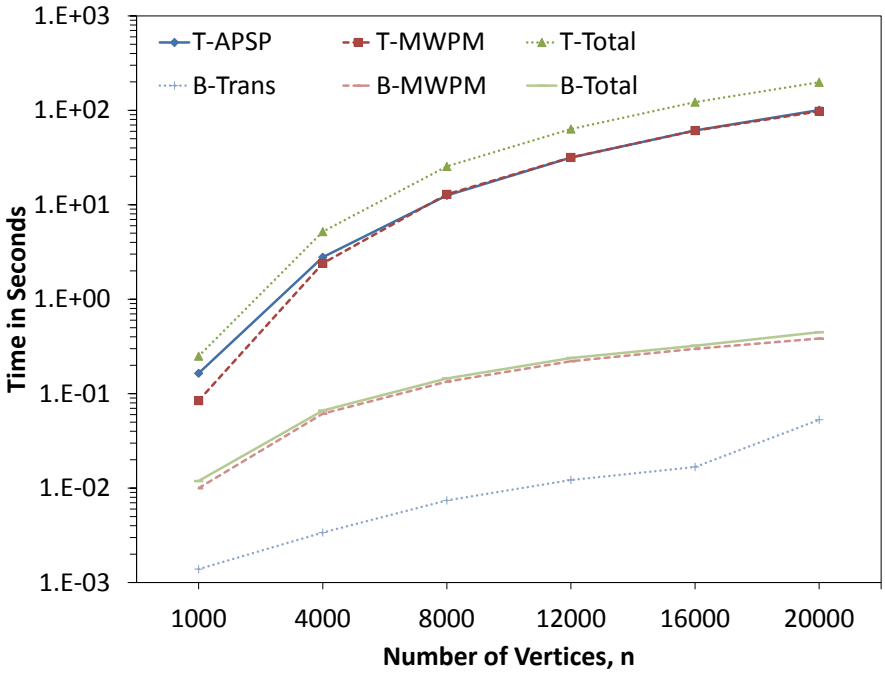


Figure 7.1: Performance of UNCCD algorithms for random graphs as the number of vertices is varied.

For random graphs, we find that the b-matching approach is substantially faster than the T-join approach. As the number of vertices increase, both the APSP and MWPM algorithms for the T-join approach have almost identical execution times. For the b-matching approach, the MWPM computation serves as the bottleneck of the algorithm. What is interesting is how quickly the b-matching approach runs. However, since the input graph is sparse, even when the graph is transformed as specified in the b-matching algorithm, this is expected.

For both torus graphs, we find that the b-matching approach is superior to the T-join approach for all n. We also observe that the MWPM computation dominates the execution time of the b-matching algorithm. However, what is initially surprising is that computing the MWPM is substantially faster than computing the APSP for the T-join algorithm. As a result, the data representing T-APSP and T-Total overlaps in Figures 7.2 and 7.3. This is because of the structure

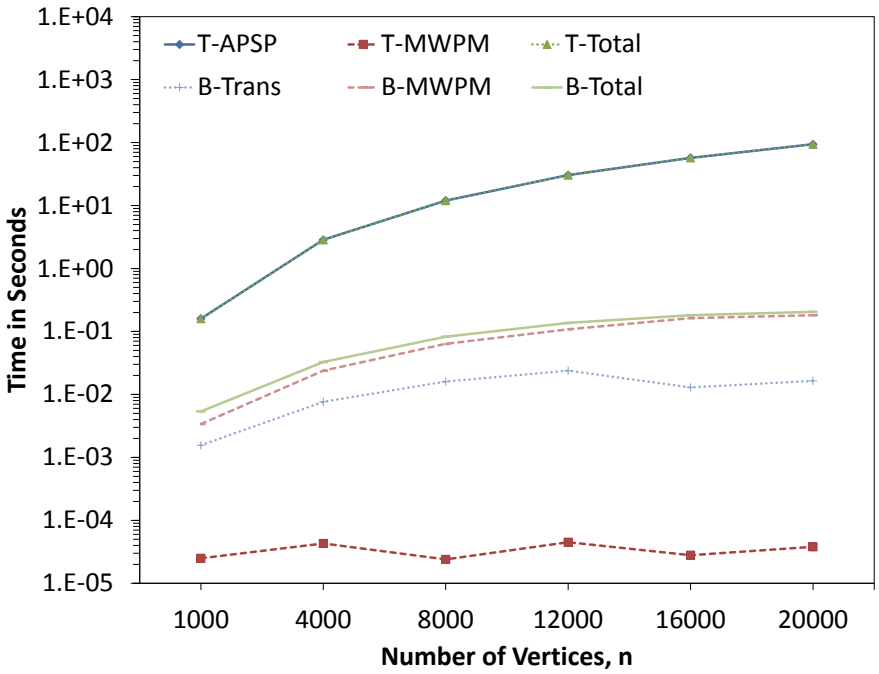


Figure 7.2: Performance of UNCCD algorithms for layered torus graphs as the number of vertices is varied.

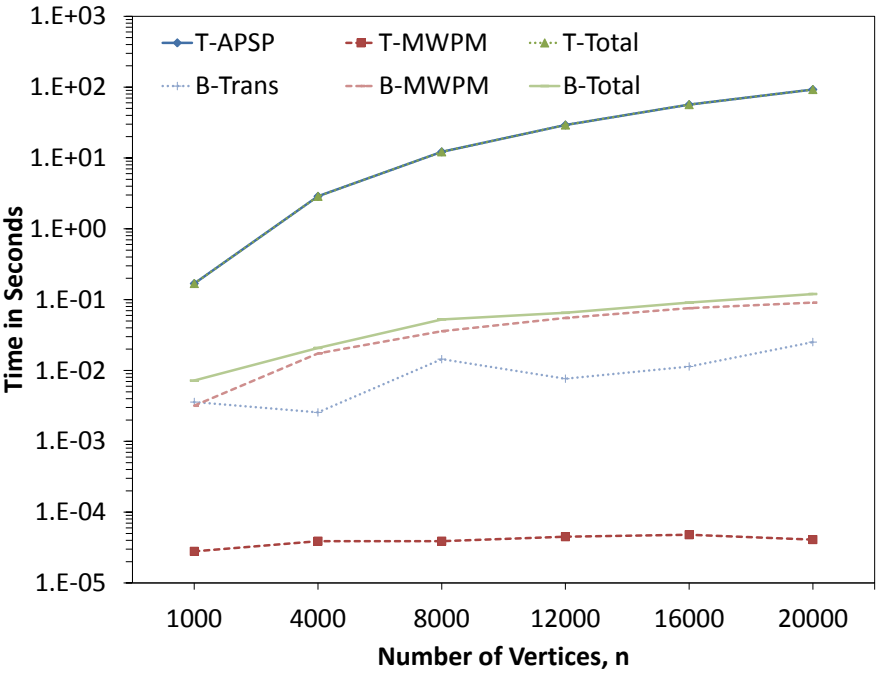


Figure 7.3: Performance of UNCCD algorithms for square torus graphs as the number of vertices is varied.

of the negative cost cycles in our study.

Recall that the T-join approach finds the MWPM of $\bar{G}_d[T^-]$, where G_d is G except the cost of all edges are non-negative, $\bar{G}_d$ is the graph representing the all pairs shortest paths in G_d, and $\bar{G}_d[T^-]$ is the subgraph of $\bar{G}_d$ induced by the set of vertices in G that have an odd number of adjacent negative edges. Since we use similar testing parameters as previous negative cycle detection studies, where the input graph contains a small number of negative cost cycles and each cycle has a single negative edge, $|\bar{G}_d[T^-]|$ is small. Therefore, it makes sense that computing the MWPM takes very little time.

Finally, we observe that both approaches run approximately twice as fast when using either torus graph compared to random graphs. Further, the b-matching approach runs faster for square torus graphs than layered torus graphs. We attribute these results to the structure of the torus graphs themselves. For the T-join approach, this goes along with the fact that the MWPM computation is extremely fast for the torus graphs studied. According to the data in Table 7.2, we can see that the execution time of the APSP computation was not significantly affected by the graph type.

Based on these results, we conclude that the b-matching approach is superior to the T-join approach for all graph types as we increase the number of vertices.

7.4.2 Number of Edges

We next compare the performance of the b-matching and T-join algorithms as we vary the number of edges. In this study, we set the number of vertices to 2500. We set the number of edges to $3 \cdot n, 10 \cdot n, 20 \cdot n, n^{1.5}, 2 \cdot n^{1.5}$, and $4 \cdot n^{1.5}$. Since $n = 2500$, this means we set the number of edges to $7500, 25000, 50000, 125000, 250000$, and 500000. We note that sparse graphs are represented by $c \cdot n$ edges, and dense graphs are represented by $c \cdot n^{1.5}$ edges, where c is some constant. For random graphs, the range of edge costs is between -512 and 512. For torus graphs, the range of edge costs is between 1 and 512, except for the negative cost cycles. Finally, for torus graphs, we have 10 negative cost cycles, each containing x edges, where x is defined in Chapter 7.2.

For random graphs, we find that the T-join approach is substantially superior than the b-matching approach as we increase the number of edges. Recall from Chapter 6.1.2 that the b-

Table 7.2: Experiment Results for Number of Vertices (in Seconds)

		T-join			b-Matching		
Graph Family	n	APSP	MWPM	Total	Transformation	MWPM	Total
Random	1000	0.164	0.084	0.250	0.001	0.010	0.012
		0.008	0.002	0.007	0.001	0.000	0.001
	4000	2.786	2.402	5.195	0.003	0.061	0.066
		0.023	0.167	0.178	0.000	0.001	0.001
	8000	12.568	12.886	25.470	0.007	0.134	0.145
		0.085	1.016	1.039	0.000	0.004	0.004
	12000	31.649	31.698	63.368	0.012	0.221	0.239
		0.437	2.640	2.794	0.000	0.005	0.005
	16000	61.200	61.038	122.195	0.017	0.298	0.322
		0.102	4.855	4.838	0.000	0.006	0.007
	20000	100.472	97.667	198.274	0.053	0.384	0.447
		0.161	5.635	5.739	0.000	0.009	0.008
Layered Torus	1000	0.160	0.000	0.160	0.002	0.003	0.005
		0.023	0.000	0.023	0.001	0.000	0.001
	4000	2.854	0.000	2.854	0.008	0.024	0.033
		0.014	0.000	0.014	0.000	0.000	0.001
	8000	11.969	0.000	11.970	0.016	0.064	0.083
		0.099	0.000	0.099	0.000	0.002	0.002
	12000	30.432	0.000	30.433	0.024	0.108	0.137
		0.091	0.000	0.091	0.000	0.003	0.002
	16000	57.161	0.000	57.162	0.013	0.163	0.181
		0.446	0.000	0.446	0.000	0.001	0.002
	20000	93.941	0.000	93.943	0.016	0.181	0.205
		0.574	0.000	0.574	0.000	0.006	0.007
Square Torus	1000	0.169	0.000	0.169	0.004	0.003	0.007
		0.013	0.000	0.013	0.001	0.000	0.001
	4000	2.874	0.000	2.874	0.003	0.017	0.021
		0.027	0.000	0.027	0.000	0.001	0.001
	8000	12.199	0.000	12.200	0.014	0.036	0.052
		0.084	0.000	0.085	0.000	0.001	0.001
	12000	29.211	0.000	29.212	0.008	0.055	0.065
		0.086	0.000	0.086	0.000	0.001	0.001
	16000	56.662	0.000	56.663	0.011	0.076	0.091
		0.132	0.000	0.132	0.000	0.000	0.001
	20000	92.524	0.000	92.525	0.025	0.091	0.120
		0.079	0.000	0.079	0.000	0.001	0.001

Note: For each entry, the top number is the average execution time, and the bottom number is the standard deviation. Any entry with 0.000 means it is less than 0.001.

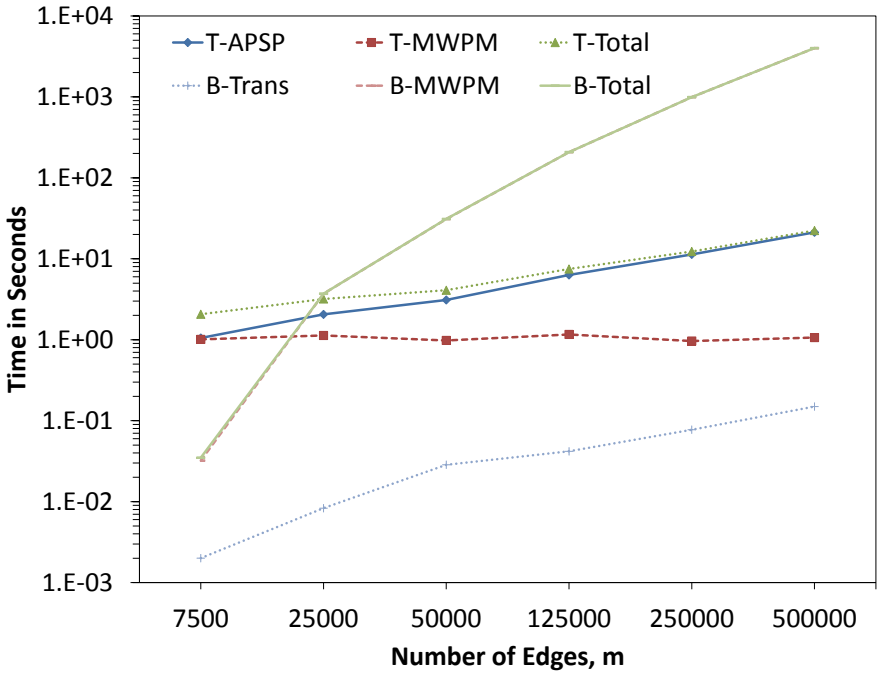


Figure 7.4: Performance of UNCCD algorithms for random graphs as the number of edges is varied.

matching algorithm transforms G into G', such that $|V(G')| = 2 \cdot (n+m)$ and $|E(G')| = n+5 \cdot m$. Since $m = O(n^{1.5})$ in our study, $|V(G')| = O(n^{1.5})$ and $|E(G')| = O(n^{1.5})$. Therefore, the b-matching algorithm technically runs in $O(n^3 \cdot \log n)$ time.

For the T-join approach, recall that we first find the metric closure of G before computing the MWPM. This means $m' = O(n^2)$ regardless of m which explains why the MWPM computation for the T-join approach was unaffected as we increase the number of edges. However, we observe an increase in the execution time of the APSP computation. This is because the APSP algorithm runs in $O(m \cdot n + n^2 \cdot \log N)$ time. Since $m = O(n^{1.5})$ in our study, the APSP algorithm runs in $O(n^{2.5} + n^2 \cdot \log N)$ time, and the MWPM algorithm runs in $O(n \cdot m' + n^2 \cdot \log n))$ time, where $|E(\bar{G}_d)| = m'$. Although m' can be as large as $n^{1.5}$ in this study, the actual value of m' depends on the input graph. Therefore, it makes sense that the T-join approach is much faster than the b-matching approach as we increase the number of edges.

Another interesting observation is that the MWPM algorithm runs significantly faster than the APSP algorithm in the T-join approach. In fact, the data for T-APSP and T-Total are almost

identical in Table 7.3. This is because the test graphs contain a small number of vertices with an odd number of adjacent negative edges. This results in m' being much smaller than $n^{1.5}$, so the running time of the APSP computation is actually greater than the running time of the MWPM computation.

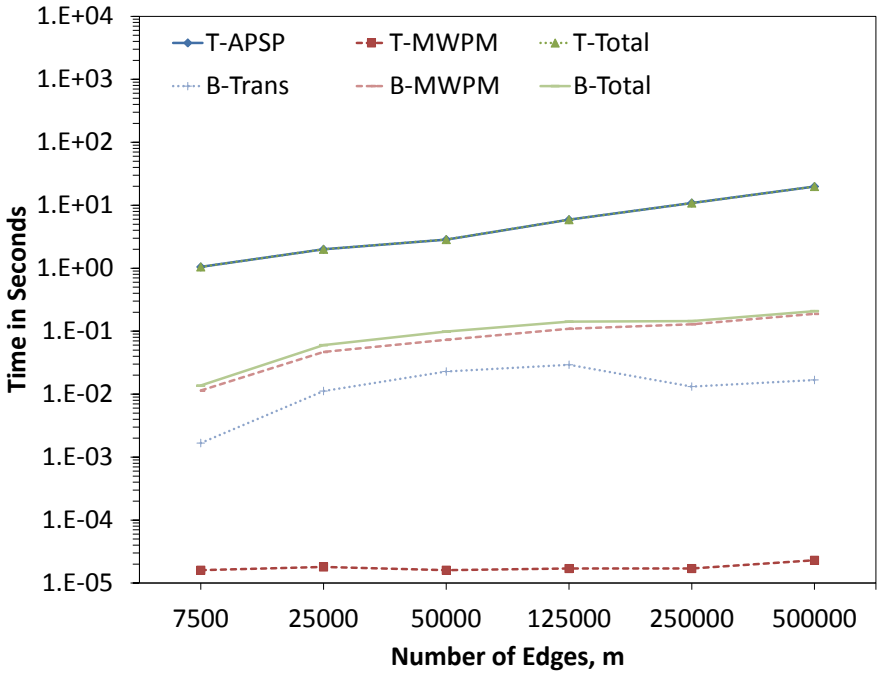


Figure 7.5: Performance of UNCCD algorithms for layered torus graphs as the number of edges is varied.

For both torus graphs, we observe that the b-matching approach is superior to the T-join approach for all m. This is quite surprising considering that the opposite is true for random graphs. Based on the data in Table 7.3, there is an increase in the execution time of the b-matching approach as we increase the number of edges. However, the reason the MWPM computation runs extremely quick is unknown. Our conjecture is that the structure of the negative cost cycle is a contribution for the fast execution time. This is worth investigating in future work.

Similar to the results in Chapter 7.4.1, the APSP algorithm serves as the bottleneck of the T-join approach, while the MWPM algorithm runs extremely fast. We attribute this to the structure of the negative cost cycle, where each negative cost cycle consists of a single negative edge. This results in the input graph for the MWPM algorithm having a small number of vertices for the

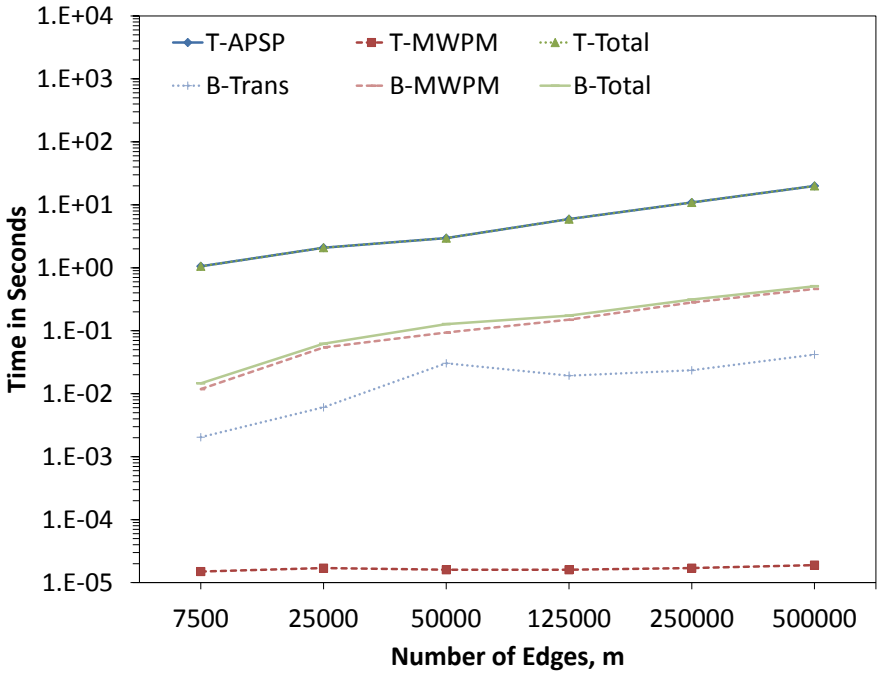


Figure 7.6: Performance of UNCCD algorithms for square torus graphs as the number of edges is varied.

reasons explained in the previous subsection.

Finally, we find that there is no substantial difference between the running time of the T-join approach among all three graph families. The data shows that the difference in the execution times is less than three seconds. Although part of the difference is attributed to the fact that the MWPM algorithm is very fast for both layered and square torus graphs, this shows that the execution time of the APSP algorithm is not affected by the graphs in our study.

We conclude from the results that the T-join approach is superior to the b-matching approach for random graphs as we increase the number of edges. Within the context of the torus graphs in this study, we conclude that the b-matching approach is superior to the T-join approach.

7.4.3 Size of K

We now study the performance of the b-matching and T-join algorithms as we vary K, the magnitude of the largest edge cost. In this study, we set the number of vertices to 20000. We set the number of edges to $3 \cdot n = 60000$. For random graphs, the intervals of edge costs

Table 7.3: Experiment Results for Number of Edges (in Seconds)

		T-join			b-Matching		
Graph Family	m	APSP	MWPM	Total	Transformation	MWPM	Total
Random	7500	1.052	1.011	2.066	0.002	0.032	0.035
		0.019	0.065	0.071	0.000	0.000	0.000
	25000	2.057	1.129	3.191	0.008	3.703	3.715
		0.066	0.058	0.106	0.003	0.005	0.003
	50000	3.101	0.981	4.088	0.029	30.982	31.014
		0.074	0.060	0.077	0.000	2.383	2.383
	125000	6.307	1.163	7.480	0.042	207.876	207.929
		0.029	0.047	0.065	0.000	19.894	11.944
	250000	11.334	0.965	12.311	0.077	992.972	993.071
		0.043	0.001	0.044	0.001	3.053	3.054
	500000	21.249	1.064	22.331	0.150	3998.119	3998.312
		0.066	0.053	0.067	0.002	143.189	143.188
Layered Torus	7500	1.048	0.000	1.049	0.002	0.011	0.014
		0.010	0.000	0.010	0.000	0.000	0.000
	25000	1.997	0.000	2.000	0.011	0.047	0.060
		0.037	0.000	0.037	0.003	0.001	0.003
	50000	2.852	0.000	2.853	0.023	0.073	0.099
		0.043	0.000	0.043	0.000	0.001	0.001
	125000	5.896	0.000	5.899	0.029	0.109	0.142
		0.115	0.000	0.115	0.000	0.002	0.002
	250000	10.851	0.000	10.853	0.013	0.128	0.144
		0.044	0.000	0.044	0.000	0.004	0.004
	500000	19.854	0.000	19.860	0.017	0.189	0.208
		0.046	0.000	0.046	0.000	0.003	0.004
Square Torus	7500	1.058	0.000	1.058	0.002	0.012	0.015
		0.013	0.000	0.013	0.001	0.000	0.001
	25000	2.087	0.000	2.087	0.006	0.054	0.062
		0.052	0.000	0.052	0.000	0.001	0.001
	50000	2.959	0.000	2.959	0.030	0.094	0.128
		0.056	0.000	0.056	0.000	0.002	0.002
	125000	5.926	0.000	5.927	0.019	0.150	0.174
		0.057	0.000	0.058	0.000	0.003	0.004
	250000	10.906	0.000	10.909	0.024	0.282	0.313
		0.090	0.000	0.089	0.000	0.005	0.005
	500000	19.964	0.000	19.973	0.042	0.463	0.510
		0.045	0.000	0.045	0.000	0.008	0.009

Note: For each entry, the top number is the average execution time, and the bottom number is the standard deviation. Any entry with 0.000 means it is less than 0.001.

are set to $[-32, 32]$, $[-64, 64]$, $[-128, 128]$, $[-256, 256]$, and $[-512, 512]$. For both torus graphs, the intervals of edge costs are set to $[1, 32]$, $[1, 64]$, $[1, 128]$, $[1, 256]$, and $[1, 512]$, except for the negative cost cycles. Finally, for both torus graphs, we have 10 negative cost cycles, each containing x edges, where x is defined in Chapter 6.1.2.

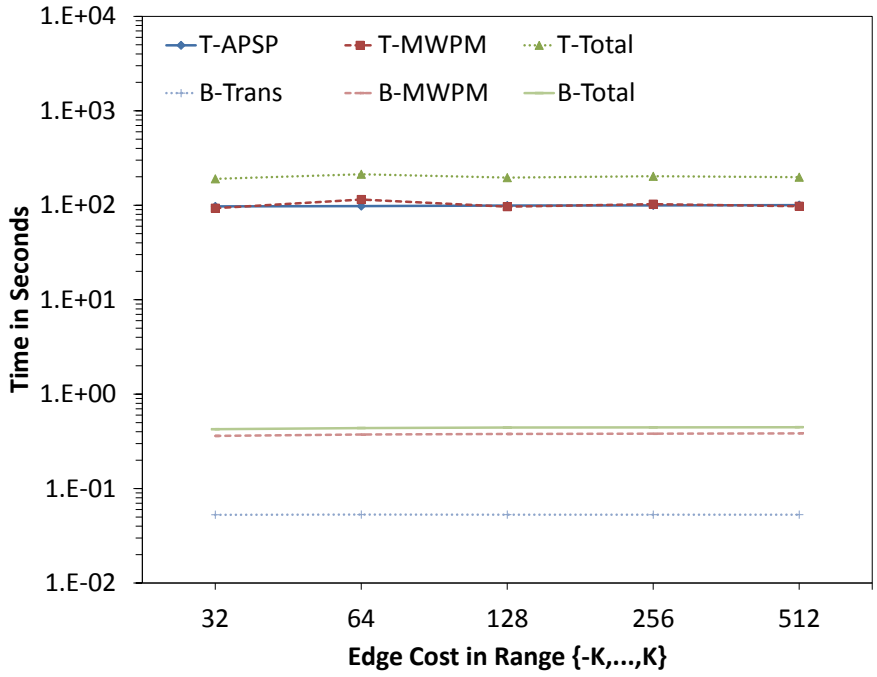


Figure 7.7: Performance of UNCCD algorithms for random graphs as the size of K is varied.

For random graphs, since the graph is sparse, it is expected that the b-matching approach is superior to the T-join approach based on the discussion in Chapter 7.4.1. For the b-matching approach, the execution time of the MWPM algorithm slightly increases as $|K|$ increases. However, it does not appear to be significant enough of an increase to clearly state that the algorithm runs faster when $|K|$ is small.

For the T-join approach, the execution time of the APSP algorithm slowly increases as $|K|$ increases as well. However, there does not appear to be any correlation for the MWPM algorithm since the results are sporadic.

For both torus graphs, we observe similar results to random graphs. For the T-join approach, the execution time of the APSP algorithm slightly increases as we increase $|K|$. Since the MWPM algorithm runs extremely fast, this implies that the execution time of the T-join approach slowly

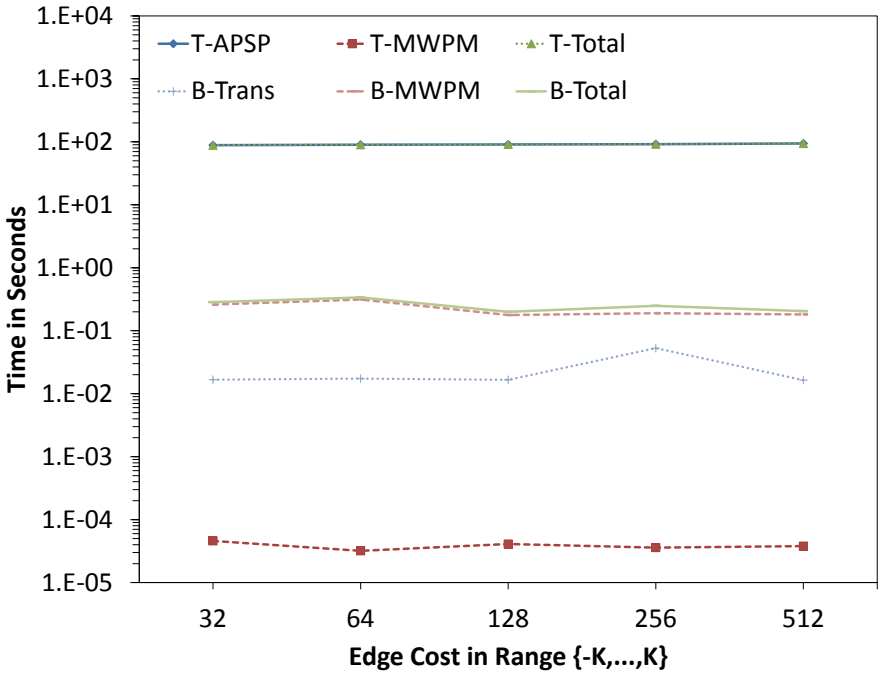


Figure 7.8: Performance of UNCCD algorithms for layered torus graphs as the size of K is varied.

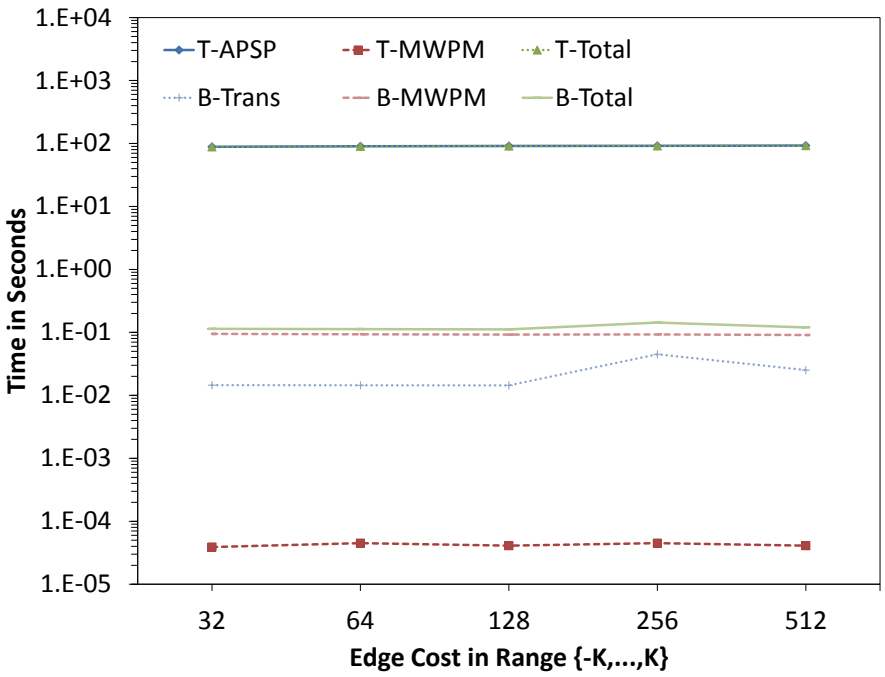


Figure 7.9: Performance of UNCCD algorithms for square torus graphs as the size of K is varied.

increases as well. For the b-matching approach, we find that MWPM computation does not have any correlation. This results in the total execution time being all over the place, even though the times are small.

One reason this occurs for all three graph families is because the running time of the Blossom V implementation does not depend on K. Although there exists a MWPM algorithm where K is a parameter [77], we use the Blossom V implementation because it is efficient regardless of K. Another reason is because we use Goldberg's $O(m + n \cdot \log N)$ time APSP algorithm, where $N = K$, rather than Shoshan and Zwick's algorithm. The data shows that there is a small increase in the execution time of the APSP algorithm as $|K|$ increases. However, if a practical implementation for Shoshan and Zwick's algorithm existed, then we would obtain more significant results. However, as explained in Chapter 7.1, this is not possible with the current hardware.

Based on our observations within the context of our study, we cannot conclude that there is a correlation between the size of K and the execution time of either UNCCD algorithms.

7.4.4 Negative Cost Cycles

Our final study observes the performance of the b-matching and T-join algorithms as we modify the negative cost cycles. In this study, we set the number of vertices to 20000. We set the number of edges to $3 \cdot n = 60000$. For random graphs, recall that the generator cannot directly control the number or size of negative cost cycles. Therefore, we use a similar approach used in [11], where the largest edge cost is fixed at 256, and the smallest edge cost is set to $0, -32, -64, -128, -256$, and -512. For both torus graphs, we use the categories defined in Table 7.1. Note that $x = 800$ for layered torus graphs, and $x = 142$ for square torus graphs in this study.

We should also note that for all figures in this subsection, there is no entry for the MWPM computation in the T-join approach when the graph has zero negative cost cycles. Recall that the T-join algorithm computes the MWPM for the subgraph containing all vertices that have an odd number of adjacent negative edges. Since the graphs in this specific case have no negative edges, meaning no negative cost cycles, the MWPM algorithm does not run.

For random graphs, we observe that the b-matching approach is superior to the T-join approach

Table 7.4: Experiment Results for Size K (in Seconds)

		T-join			b-Matching		
Graph Family	$\|K\|$	APSP	MWPM	Total	Transformation	MWPM	Total
Random	32	97.257	93.116	190.513	0.053	0.361	0.425
		0.239	7.101	7.250	0.000	0.013	0.013
	64	97.991	115.055	213.193	0.053	0.374	0.438
		0.329	3.416	3.212	0.000	0.008	0.007
	128	99.239	96.616	196.032	0.053	0.379	0.443
		0.053	6.715	6.797	0.000	0.008	0.009
	256	99.663	102.728	202.684	0.053	0.381	0.445
		0.144	5.773	5.800	0.000	0.006	0.006
	512	100.472	97.667	198.274	0.053	0.384	0.447
		0.161	5.635	5.739	0.000	0.009	0.008
Layered Torus	32	88.351	0.000	88.352	0.017	0.261	0.284
		0.094	0.000	0.094	0.000	0.008	0.009
	64	89.829	0.000	89.831	0.017	0.314	0.337
		0.100	0.000	0.100	0.000	0.007	0.007
	128	91.062	0.000	91.064	0.017	0.177	0.200
		0.099	0.000	0.099	0.000	0.006	0.001
	256	91.766	0.000	91.767	0.053	0.190	0.249
		0.046	0.000	0.046	0.000	0.008	0.008
	512	93.941	0.000	93.943	0.016	0.181	0.205
		0.574	0.000	0.574	0.000	0.006	0.007
Square Torus	32	88.439	0.000	88.435	0.015	0.095	0.114
		0.095	0.000	0.095	0.000	0.000	0.002
	64	90.264	0.000	90.265	0.014	0.093	0.113
		0.269	0.000	0.269	0.000	0.000	0.002
	128	91.176	0.000	91.178	0.014	0.092	0.112
		0.224	0.000	0.224	0.000	0.000	0.002
	256	91.675	0.000	91.676	0.045	0.093	0.143
		0.065	0.000	0.065	0.000	0.000	0.002
	512	92.524	0.000	92.525	0.025	0.091	0.120
		0.079	0.000	0.079	0.000	0.001	0.001

Note: For each entry, the top number is the average execution time, and the bottom number is the standard deviation. Any entry with 0.000 means it is less than 0.001.

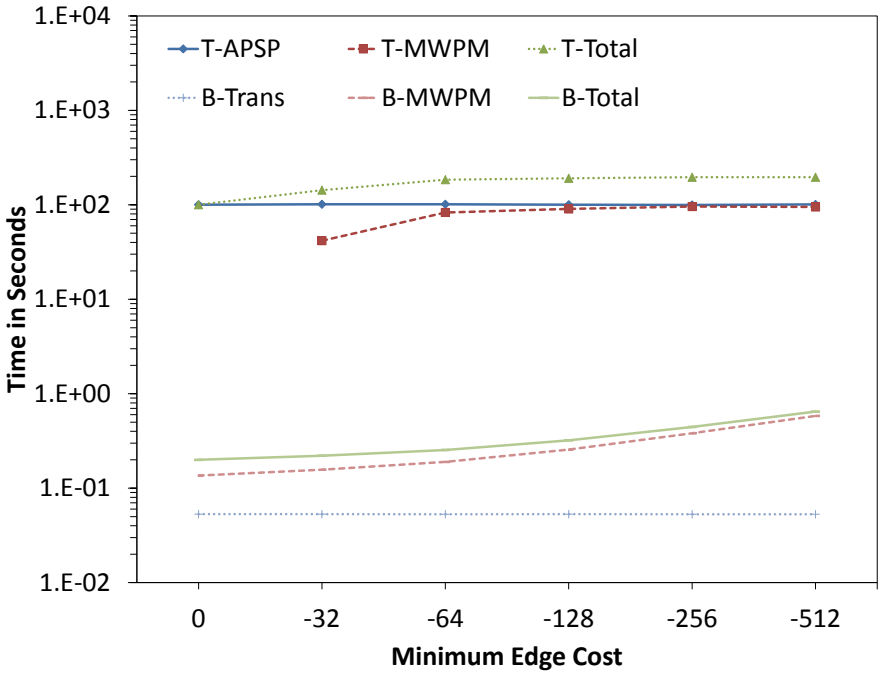


Figure 7.10: Performance of UNCCD algorithms for random graphs as minimum edge cost is varied.

since all input graphs in this experiment are sparse. Also, for both UNCCD algorithms, we find that the total execution times increase as we decrease the minimum edge cost. For the b-matching approach, we see that the rate of increase grows more as we decrease the minimum edge cost. For the T-join approach, even though the execution time increases as we decrease the minimum edge cost, we observe mixed results when the minimum edge cost goes from -256 to -512. In this case, we find that the execution time of the APSP algorithm increases, while the execution time of the MWPM algorithm decreases.

In previous studies, such as [11], it was observed that decreasing the minimum edge cost results in a substantially faster execution time for random graphs. This was because more negative cost cycles implied a higher chance of detecting one. Since previous negative cost cycle detection algorithms terminate as soon as a negative cost cycle is detected, these algorithms result in running much faster when more negative cost cycles are present. However, in our study, this is not the case. This is because the UNCCD algorithms studied in this paper do not detect the presence of a negative cost cycle until the end of the algorithm. In other words, these algorithms must complete

all computations (i.e., APSP and MWPM for the T-join approach and MWPM for the b-matching approach) before determining whether or not a negative cost cycle is present.

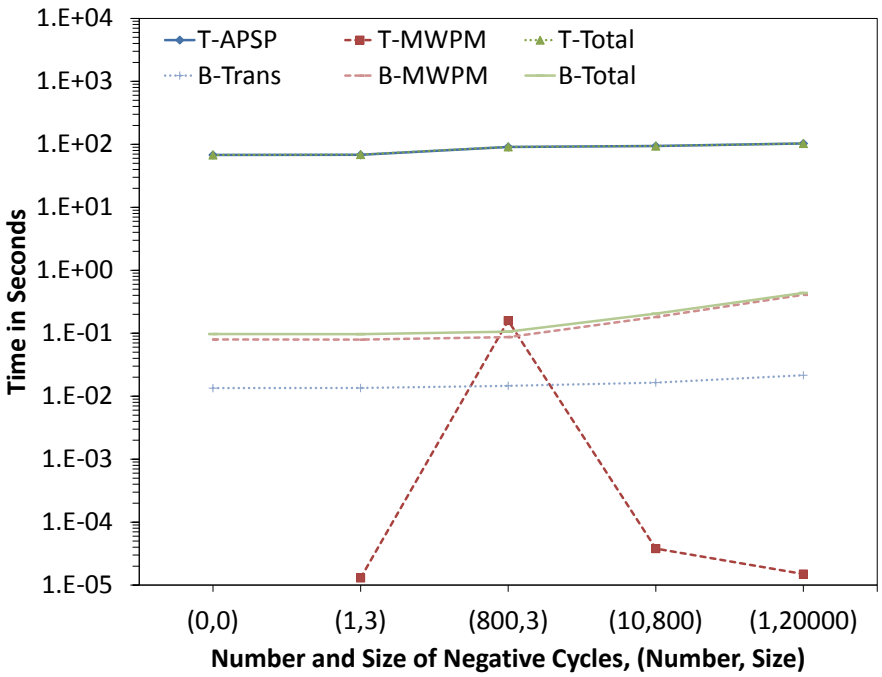


Figure 7.11: Performance of UNCCD algorithms for layered torus graphs as the number and size of negative cost cycles are varied.

For both torus graphs, the results are rather interesting. First, both UNCCD algorithms run faster when the graph did not contain any negative cost cycles. Second, both UNCCD algorithms ran the slowest when the graph consists of a single Hamiltonian negative cost cycle. Next, the algorithms ran faster when the graph contains a single negative cost cycle with size 3 than when the graph contains many negative cost cycles with size 3. This is surprising since it would be expected that having more negative cost cycles would mean the algorithms detect a negative cost cycle faster. Finally, both algorithms were slower when the graph has a few number of medium sized negative cost cycles than when the graph has many negative cost cycles.

When comparing layered torus graphs to square torus graphs, we find that both UNCCD algorithms run faster for layered torus graphs when the graph contains no negative cost cycles, one small negative cost cycle, or one Hamiltonian negative cost cycle. Both UNCCD algorithms run faster for square torus graphs when the graph consists of many small negative cost cycles or

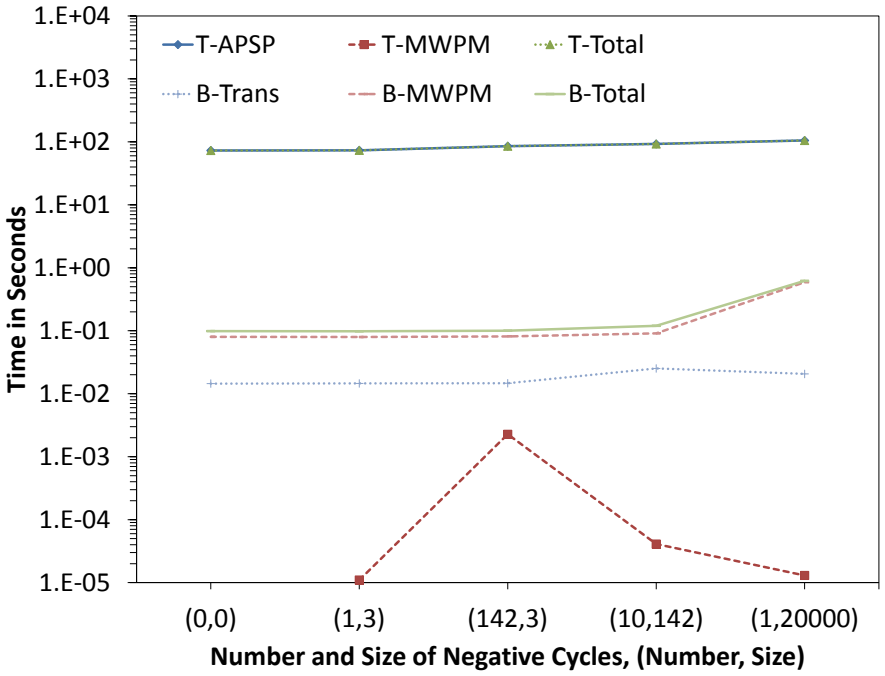


Figure 7.12: Performance of UNCCD algorithms for square torus graphs as the number and size of negative cost cycles are varied.

few medium sized negative cost cycles.

We attribute these results to the nature of APSP and MWPM algorithms. Recall that the TOR generator is designed such that it creates the graph using the input parameters first and then creates negative cost cycles as specified. This means additional edges are included on top of the current number of edges. For instance, when we include a Hamiltonian negative cost cycle, the number of edges increases from $3 \cdot n$ to $4 \cdot n$ since we add n new edges to create the negative cost cycle. This means m increases with each category from Table 7.1. Since the running times of both the APSP and MWPM algorithms depend on m, increasing m naturally increases the execution times of both algorithms.

Based on the above results, we conclude that both UNCCD algorithms perform the best when the graphs do not contain any negative cost cycles. However, if negative cost cycles are present, both algorithms perform better when there are fewer and smaller negative cost cycles.

Table 7.5: Experiment Results for Negative Cost Cycles in Random Graphs (in Seconds)

		T-join			b-Matching		
Graph Family	Min. Cost	APSP	MWPM	Total	Transformation	MWPM	Total
Random	0	100.253	0.000	100.253	0.053	0.136	0.200
		0.611	0.000	0.611	0.000	0.003	0.004
	-32	101.340	41.720	143.104	0.053	0.157	0.221
		0.608	0.026	0.609	0.000	0.003	0.003
	-64	101.502	83.033	184.602	0.053	0.190	0.254
		0.242	1.333	1.441	0.000	0.006	0.005
	-128	100.280	90.449	190.916	0.053	0.256	0.321
		0.139	5.530	5.471	0.000	0.008	0.007
	-256	99.753	95.881	195.969	0.053	0.381	0.445
		0.164	8.089	7.964	0.000	0.006	0.006
	-512	101.006	94.852	196.087	0.053	0.583	0.647
		0.435	3.388	3.693	0.000	0.019	0.020

Note: For each entry, the top number is the average execution time, and the bottom number is the standard deviation. Any entry with 0.000 means it is less than 0.001.

Table 7.6: Experiment Results for Negative Cost Cycles in Torus Graphs (in Seconds)

			T-join			b-Matching		
Graph Family	Number of Cycles	Size	APSP	MWPM	Total	Transformation	MWPM	Total
Layered Torus	0	0	67.796	0.000	67.796	0.013	0.080	0.097
			0.074	0.000	0.074	0.000	0.001	0.002
	1	3	68.091	0.000	68.916	0.013	0.079	0.097
			0.087	0.000	0.087	0.000	0.001	0.002
	800	3	90.988	0.159	91.165	0.015	0.087	0.106
			0.399	0.003	0.223	0.000	0.001	0.002
	10	800	93.941	0.000	93.943	0.016	0.181	0.205
			0.574	0.000	0.574	0.000	0.006	0.007
	1	20000	103.128	0.000	103.129	0.022	0.408	0.437
			0.091	0.000	0.091	0.000	0.017	0.017
Square Torus	0	0	72.893	0.000	72.893	0.014	0.080	0.099
			0.058	0.000	0.058	0.000	0.001	0.002
	1	3	73.367	0.000	73.368	0.015	0.080	0.098
			0.093	0.000	0.093	0.000	0.000	0.001
	142	3	85.373	0.002	85.380	0.015	0.081	0.100
			0.111	0.000	0.110	0.000	0.001	0.002
	10	142	92.524	0.000	92.525	0.025	0.091	0.120
			0.079	0.000	0.079	0.000	0.001	0.001
	1	20000	105.105	0.000	105.106	0.021	0.591	0.620
			0.122	0.000	0.122	0.000	0.034	0.033

Note: For each entry, the top number is the average execution time, and the bottom number is the standard deviation. Any entry with 0.000 means it is less than 0.001.

Part III

The Negative Cost Girth Problem

Chapter 8

Introduction

Consider a network (or directed graph) $G = (V, E)$, where V is the vertex set $|V| = n$, and E is the edge set with $|E| = m$. We now introduce new problem related to negative cycle detection called the *negative cost girth* (NCG) problem. The *girth* of an unweighted network is defined as the length (i.e., number of edges) of the shortest cycle. If the network is acyclic, then the girth is infinity. In this paper, we extend the notion of girth to weighted networks and introduce the notion of *negative cost girth* in a weighted network. Briefly, the negative cost girth of a weighted network is the negative cost cycle with the fewest number of edges. The NCG problem finds applications in several domains, such as constraint-solving, program verification and real-time scheduling.

The NCG problem was introduced in [15], wherein the first polynomial time algorithm was proposed. The algorithm, known as the matrix multiplication (MM), is a dynamic programming approach that finds paths of increasing cost from each vertex to itself. The first path of negative cost from a vertex to itself is the NCG. The MM approach runs in $O(n^3 \cdot \log k)$ time, where k is the size of the NCG. In [15], the NCG problem was referred to as the *Optimal Length Resolution Refutation* (OLRR) problem. We discuss the MM approach in Appendix B.

The next three chapters present several algorithms related to finding the NCG. We first discuss two strongly polynomial NCG algorithms [24], namely Edge-Progress (EP) and Edge-Relax (ER). As in [15], both algorithms find paths of increasing cost from each vertex to itself to determine if any cycle has a negative cost. However, both algorithms apply different techniques for extending paths. In the EP algorithm, for each pair of vertices u and x, we scan all vertices y that are *incoming* to x to find the shortest paths from u to y and y to x. For the ER algorithm, we relax

each edge to find the shortest path of *non-positive* cost between each pair of vertices. The EP algorithm runs in $O(n^2 \cdot k + m \cdot n \cdot k)$ time, and the ER algorithm runs in $O(m \cdot n \cdot k)$ time, where k is the size of the NCG.

We also provide an empirical study comparing the performance of the above NCG algorithms, including the MM algorithm. The experiments indicate that the EP and ER algorithms are superior to the MM algorithm in sparse networks. For dense networks, the MM algorithm proves to be superior to both the EP and ER algorithms. Finally, the empirical study concludes that the ER algorithm is superior to the EP algorithm in all cases.

We note that the aforementioned NCG algorithms terminate when the first negative cost cycle is detected. These algorithms use various shortest path methods to determine the NCG. As with all shortest path algorithms, once a negative cost cycle is detected, we cannot make correct inferences about the costs of any simple paths for each iteration. This is because the correctness of the costs of paths is contingent on the existence of simple paths. Once a negative cost cycle has been detected, any shortest path computed after the detection may consist of a combination of negative cost cycles and paths which, by definition, is not a simple path.

We next describe a work-efficient parallel implementation of the MM approach that runs in $O(\log k \cdot \log n)$ parallel time using $O(n^3)$ processors. We also conduct an empirical analysis for both the parallel implementation, using MPI, and the corresponding sequential NCG implementation. Our experiments indicate that as the number of processors doubles, the total execution time reduces by approximately half for sparse networks.

We then present a new NCG algorithm for planar, directed networks. We can apply the above NCG algorithms for general networks to find the NCG in planar networks. However, the extant algorithms are topology-oblivious. In other words, the algorithms do not consider the topology of the network. The planar NCG algorithm is based on the well-known Lipton-Tarjan planar separator theorem [87]. This results in an algorithm that runs in $O(n^{1.5} \cdot k)$ time, which is superior to all previously known NCG algorithms when restricted to planar networks. Our algorithm also works correctly for classes of general networks that have separators. On a network having separator size n^a, number of edges n^b, and in which the separator can be found in $O(n^d)$ time, our NCG algorithm runs in $O(n^{a+b} \cdot k + n^d \cdot \log n)$ time.

8.1 Preliminaries and Notation

We are given a weighted network $G = (V, E, c)$, where V is the set of n vertices, E is the set of m edges, and we have a cost function $\mathbf{c} : E \to \mathbb{R}$. For each edge $e_{ij} \in E$, we let c_{ij} be the cost of that edge.

The *girth* of an unweighted, undirected graph is defined as the length, or number of edges, of the shortest simple cycle contained in the graph. If the graph is acyclic, then the girth is infinity. We provide examples of the girth of a graph in Figure 8.1. In Figure 8.1(a), the graph has a single cycle $C = (V_C, E_C, c)$, where $E_C = \{e_{ab}, e_{bc}, e_{cd}, e_{da}\}$. Since $|E_C| = 4$, the girth of the graph is 4. In Figure 8.1(b), there are two simple cycles, which we will denote as C_1 and C_2. In this example, $E_{C_1} = \{e_{ab}, e_{bc}, e_{cd}, e_{da}\}$, while $E_{C_2} = \{e_{bc}, e_{cd}, e_{db}\}$. Since $|E_{C_2}| = 3$, $|E_{C_1}| = 4$, and $|E_{C_2}| < |E_{C_1}|$, the girth of the graph is 3.

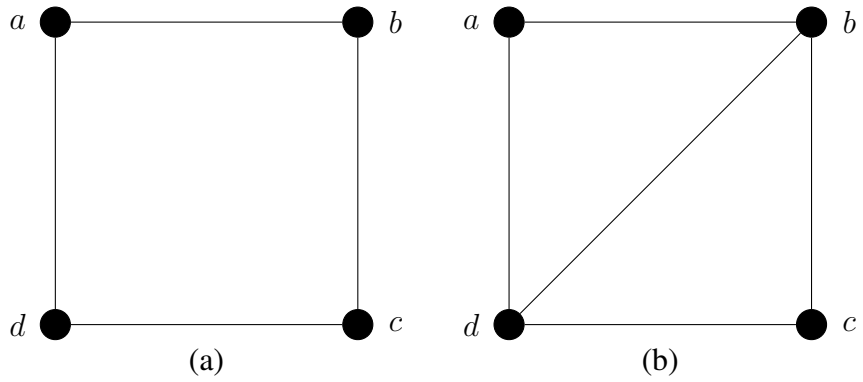


Figure 8.1: Girth examples. The network in (a) has girth 4, while the network in (b) has girth 3.

In this thesis, we apply the notion of girth to networks (or directed graphs in the literature), where the edges are weighted. We define the *negative cost girth* (NCG) as the length, or number of edges, of the negative cost cycle with the fewest number of edges. Recall from Chapter 5 that a negative cost cycle is a path from a vertex to itself, whose total cost is negative. Similar to the girth in unweighted, undirected graphs, if the network is acyclic, then the NCG is infinity.

We provide examples of the girth of a graph in Figure 8.2. In Figure 8.2(a), the graph has a single negative cost cycle $C = (V_C, E_C, c)$, where $E_C = \{e_{ab}, e_{bc}, e_{cd}, e_{da}\}$. Since $|E_C| = 4$, the NCG of the graph is 4. In Figure 8.2(b), the girth of the graph is 3 since there is a cycle $E_{C_1} = \{e_{bc}, e_{cd}, e_{db}\}$ with 3 edges. However, the total cost of C_1 is not negative. The only negative

cost cycle in the graph is C_2, where $E_{C_2} = \{e_{ab}, e_{bc}, e_{cd}, e_{da}\}$ and has 4 edges. Therefore, the NCG of the graph is 4.

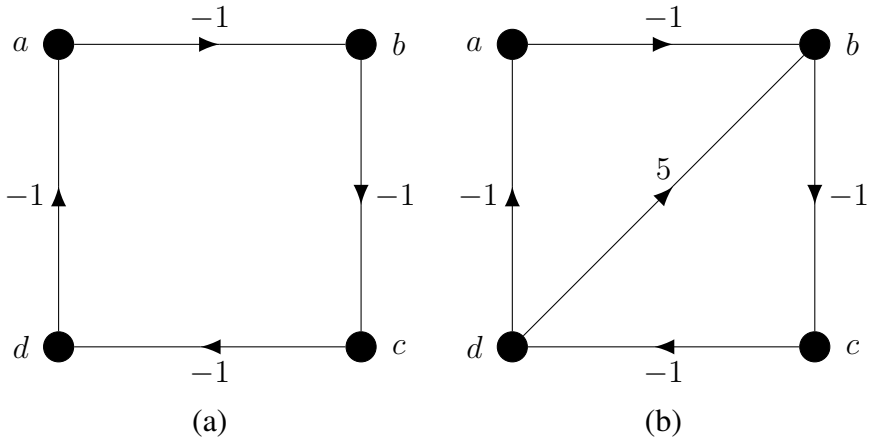


Figure 8.2: Negative cost girth examples. Both networks have a negative cost girth 4.

Using the terminology above, we define the *negative cost girth problem* as follows:

Given a network $G = (V, E)$ with arbitrarily weighted edges, find the negative cost girth of the network.

Chapter 9

Improved NCG Algorithms for General Networks

In this chapter, we present two strongly polynomial algorithms for the NCG problem in general networks. The Edge-Progress (EP) approach involves finding the shortest paths for all pairs of vertices by examining the vertices that are *incoming* to each vertex. On a network with n vertices, m edges, and k is the NCG, the EP algorithm runs in $O(n^2 \cdot k + m \cdot n \cdot k)$ time. The Edge-Relax (ER) approach includes relaxing each edge to find the shortest paths of *non-positive* cost between all pairs of vertices. This algorithm runs in $O(m \cdot n \cdot k)$ time. These algorithms are also discussed in [24].

9.1 The Edge-Progress Algorithm

In this section, we describe the EP approach for solving the NCG problem. The notation used in the algorithm is described in Chapter 8.1. We also represent G as an adjacency list Adj. However, Adj works differently from the adjacency list structure described in [22]. Previously, for each vertex $v \in V$, $Adj(v)$ is the set of outgoing edges from v in G. In the EP algorithm, $u \in Adj(v)$ implies there exists an edge e_{uv} that is *incoming* to vertex v in G. An example of Adj is provided in Figure 9.1.

We let $\mathbf{F}_k$ be an $n \times n$ matrix that stores the shortest paths between each pair of vertices using at most k edges, where $1 \leq k \leq n$. We initialize the values of $\mathbf{F}_1$ as follows. For every pair of

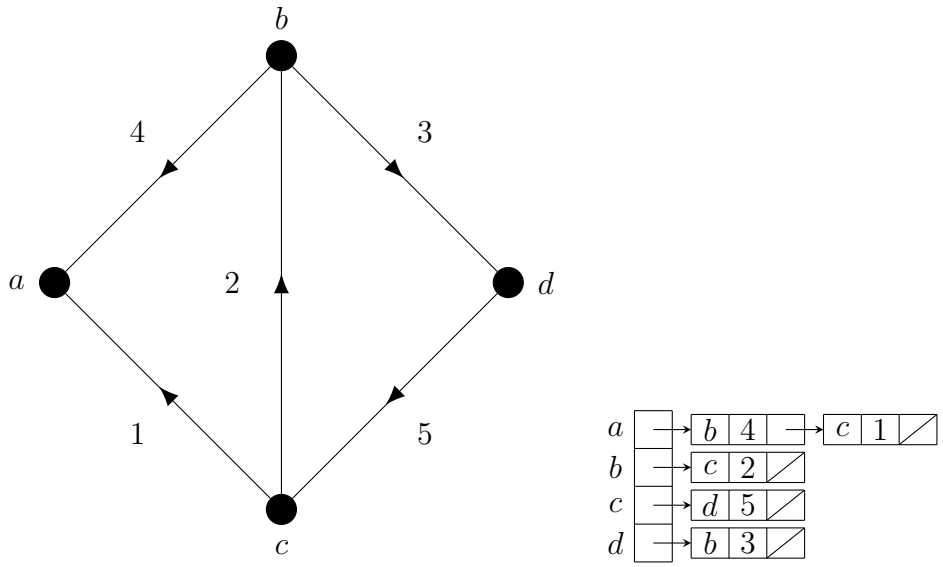


Figure 9.1: Example of the Adjacency List using Incoming Edges

vertices u and x, if $u = x$, then $F_1(u, x) = F_1(u, u) = 0$. If there exists an edge $e_{ux} \in \mathbf{E}$, then $F_1(u, x) = c_{ux}$. Otherwise, $F_1(u, x) = \infty$.

We use a dynamic programming approach for the EP algorithm. Observe that a negative cost cycle in a network must have at least two edges. For each pair of vertices u and x, we scan all the vertices $y \in Adj(x)$. As defined above, $F_k(u, x)$ is the length of the shortest path from u to x, using at most k edges. Assume that the values $F_{k-1}(u, y)$, where $y \in Adj(x)$, have been computed. Now, $F_k(u, x)$ can easily be computed as: $F_k(u, x) = \min_{y \mid y \in Adj(x)}(F_{k-1}(u, y) + c_{yx})$.

After computing all the shortest paths starting from u, we determine if there are any negative cost cycles by checking if $F_k(u, u) < 0$. If this is true, then we have found the NCG, which is k. Otherwise, we repeat this process for the next vertex $v \in V$, where $v \neq u$. The algorithm repeats the above steps for increasing values of k until either a negative cost cycle is found or it is determined that none exists.

These observations are summarized in Algorithm 9.1 and Algorithm 9.2. Note that Algorithm 9.2 returns the NCG, if a negative cost cycle exists. The actual cycle can be obtained by using a predecessor subgraph.

For the purpose of simplifying the expositions in Chapters 9.1.1 and 9.1.2, we define the following:

```
Function INITIALIZE()
 1: for (u = 1 to n) do
 2:    for (x = 1 to n) do
 3:       if (u = x) then
 4:          F_1(u, u) := 0.
 5:       else
 6:          if (e_ux ∈ E) then
 7:             F_1(u, x) := c_ux.
 8:          else
 9:             F_1(u, x) := ∞.
10:          end if
11:       end if
12:    end for
13: end for
14: return
```

Algorithm 9.1: NCG Edge-Progress Algorithm: Initialization

```
Function EDGE-PROGRESS(G)
 1: for (k = 2 to n) do
 2:    for (each vertex u ∈ V) do
 3:       for (each vertex x ∈ V) do
 4:          if (u = x) then
 5:             F_k(u, x) := 0.
 6:          else
 7:             F_k(u, x) := ∞.
 8:          end if
 9:          for (all y ∈ Adj(x)) do
10:             if (F_{k-1}(u, y) + c_yx < F_k(u, x)) then
11:                F_k(u, x) := F_{k-1}(u, y) + c_yx.
12:             end if
13:          end for
14:       end for
15:       if (F_k(u, u) < 0) then
16:          return ("The negative cost girth is k.")
17:       end if
18:    end for
19: end for
20: return ("G does not contain any negative cost cycles.")
```

Algorithm 9.2: NCG Edge-Progress Algorithm: EDGE-PROGRESS

1. Let f_1 be the **for** loop from lines 1 to 19 in Algorithm 9.2.

2. Let f_2 be the **for** loop from lines 2 to 18 in Algorithm 9.2.

3. Let f_3 be the **for** loop from lines 3 to 14 in Algorithm 9.2.

9.1.1 Resource Analysis

The function INITIALIZE (Algorithm 9.1) takes $O(n^2)$ time since we initialize $F_1(u, x)$ for each pair of vertices u and x.

For each iteration of f_3, we scan all the vertices y that are adjacent to x by using $Adj(x)$. Since Adj is structured as an adjacency list, the scanning process ensures that each edge that is adjacent to x is scanned exactly once. With $\mathbf{F}_k$ set up as an $n \times n$ matrix, any modifications made for a given pair of vertices u and x would be constant time operations. Because there are n vertices, f_3 runs in $O(m + n)$ time.

It is clear that f_2 runs $O(n)$ times; once for each of the n vertices. This means a single iteration of f_2 runs in $O(n \cdot (m + n))$ time. However, lines 15 and 16 force the algorithm to stop when we find the negative cost girth, which is k. This means that f_1 executes $O(k)$ times. It follows that Algorithm 9.2 runs in $O(k \cdot n \cdot (m + n)) = O(n^2 \cdot k + m \cdot n \cdot k)$ time.

As far as space is concerned, we store Adj as an adjacency list of size $O(m + n)$. It would appear that we need $O(n^3)$ space since each matrix $\mathbf{F}_k$ is size $n \times n$, and we have n matrices. However, note that at iteration l of the algorithm, where $l \leq n$, we need only the data from $\mathbf{F}_{l-1}$ and Adj. In other words, matrices $\mathbf{F}_1$ to $\mathbf{F}_{l-2}$ are no longer needed. Since the remaining matrices are no longer used, we can safely remove them from storage. As a result, at any given iteration l, we use exactly two matrices: $\mathbf{F}_{l-1}$ and $\mathbf{F}_l$. This means we only need $O(n^2)$ space for storing the matrices. Therefore, the algorithm requires $O((m + n) + n^2) = O(n^2)$ space.

9.1.2 Correctness

We have already established that the EP algorithm terminates, in that it runs in $O(n^2 \cdot k + m \cdot n \cdot k)$ time, where k is the NCG. In the event that the network does not have a negative cost cycle, the

algorithm runs in $O(n^3 + m \cdot n^2)$ time. If we assume that $m = \Omega(n)$, then the running time of the algorithm is $O(m \cdot n^2)$, in the absence of negative cost cycles.

In order to establish the correctness of Algorithm 9.2, we observe that the algorithm implements the following dynamic program:

$$F_k(u,x) = \begin{cases} c_{ux} & k = 1 \\ \min_{y\,:\,(y,x)\in E}\{F_{k-1}(u,y) + c_{yx}\} & k > 1. \end{cases}$$

The correctness of the above dynamic program follows through an inductive application of the Principle of Optimality [88], which states the following:

An optimal policy has the property that whatever the initial state and initial decision are, the remaining decisions must constitute an optimal policy with regard to the state resulting from the first decision.

9.1.3 Example of the Edge-Progress Algorithm

We now provide an example of the EP algorithm. Suppose we are given the network in Figure 9.2. We start the algorithm by initializing all the values for $F_1(u,x)$, $\forall u, x \in V$. This is shown in Table 9.1. Let $k = 2$. For each pair of vertices u and x, we scan all the vertices adjacent to x to find the shortest paths using at most two edges. These values are given in Table 9.2. We now let $k = 3$. Note that when we determine the path from a to a, we find that $F_2(a,b) + c_{ba} = 5 + -6 = -1 < F_2(a,a) = 0$. Since $F_3(a,a) < 0$, the algorithm immediately stops and declares that the NCG is 3. The resulting negative cost cycle is shown in Figure 9.3 and Table 9.3.

Table 9.1: Example of the Edge-Progress algorithm. Initialize $\mathbf{F}_1$.

$\mathbf{F}_1$	a	b	c	d
a	0	∞	3	∞
b	-6	0	∞	∞
c	∞	2	0	4
d	∞	-5	∞	0

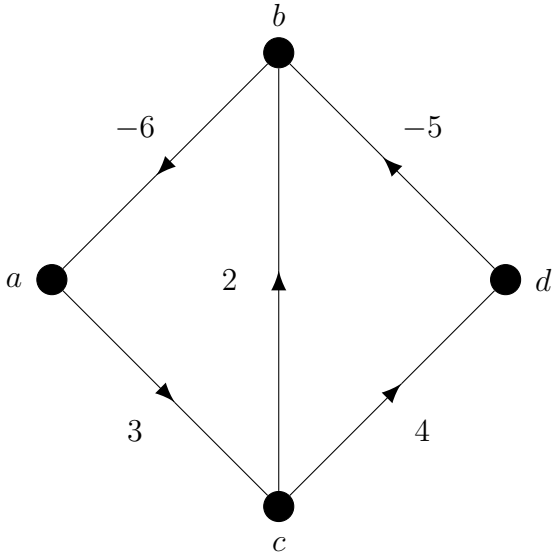


Figure 9.2: Example of the Edge-Progress algorithm. Initial network G.

Table 9.2: Example of the Edge-Progress algorithm. Matrix $\mathbf{F}_2$.

$\mathbf{F}_2$	a	b	c	d
a	0	5	3	7
b	−6	0	−3	∞
c	−4	−1	0	4
d	−11	−5	∞	0

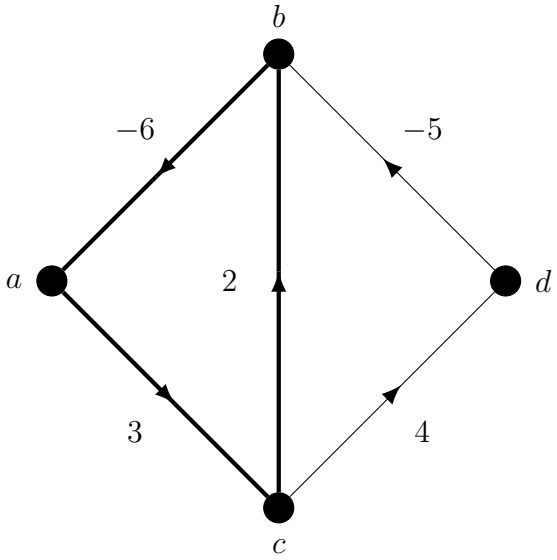


Figure 9.3: Example of the Edge-Progress algorithm. NCG is 3.

Table 9.3: Example of the Edge-Progress algorithm. Matrix $\mathbf{F}_3$.

$\mathbf{F}_3$	a	b	c	d
a	-1	2	3	4
b	-6	-1	-3	1
c	-7	-1	-1	4
d	-11	-5	-8	0

9.2 The Edge-Relax Algorithm

In this section, we discuss the ER approach for solving the NCG problem. The notation used in the algorithm is described in Chapter 8.1. We also store the edges in an additional set of arrays $StartV$ and $EndV$. $StartV[i]$ contains the tail of edge e_i, and $EndV[i]$ contains the head of edge e_i, where $1 \leq i \leq m$. For instance, consider the edges to be sorted in some order $\{e_1, e_2, \ldots, e_m\}$. Let $e_{ij} = e_1$. Accordingly, $StartV[1] = i$, and $EndV[1] = j$.

We let $\mathbf{D}^{(k)-}$ be an $n \times n$ matrix that monitors the shortest path of non-positive cost between each pair of vertices using at most k edges, where $1 \leq k \leq n$. For each pair of vertices i and j, $d_{ij}^{(k)-}$ is the cost of the shortest path of non-positive cost from i to j using at most k edges. We initialize the values in $\mathbf{D}^{(1)-}$ as follows. For every pair of vertices i and j, if $i = j$, then $d_{ij}^{(1)-} = d_{ii}^{(1)-} = 0$. If there exists an edge $e_{ij} \in E$, where $c_{ij} < 0$, then $d_{ij}^{(1)-} = c_{ij}$. Otherwise, $d_{ij}^{(1)-} = \infty$.

Our approach involves focusing only on the shortest paths with a *non-positive cost*. We let k be the number of edges in a given path, where $2 \leq k \leq n$. For each k, we relax all the edges in E. Given an edge $e_{ij} \in E$ and $i = Start[t]$ and $j = EndV[t]$, where $1 \leq t \leq m$, for the relaxation step, we scan each vertex $r \in V$. Assume that $d_{ri}^{(k-1)-}$, where $r \in V$, has been computed. We determine if both $d_{ri}^{(k-1)-}$ and $d_{ri}^{(k-1)-} + c_{ij}$ are negative. Because we are only concerned with paths with non-positive costs, we need to check the following:

1. The shortest path from r to i using at most $k - 1$ edges is negative, and

2. Adding edge e_{ij} to the path from r to i makes the cost of the path from r to i to j, using at most k edges, negative.

Thus, we compute $d_{rj}^{(k)-}$ as,

$$d_{rj}^{(k)-} = \min\{d_{ri}^{(k-1)-} + c_{ij}, d_{rj}^{(k-1)-}\}.$$

After computing $d_{rj}^{(k)-}$, we determine if we have a negative cost cycle by checking if $r = j$. If this is true, then $d_{rj}^{(k)-} = d_{rr}^{(k)-} < 0$. This means we have found the negative cost girth, which is k. Otherwise, we repeat this process for the next vertex $r \in V$. The algorithm repeats the above steps for increasing values of k until either a negative cost cycle is found, or we conclude that the network does not contain a negative cost cycle. The above observations are summarized in Algorithm 9.3 and Algorithm 9.4. Observe that Algorithm 9.4 gives us only the NCG. The actual negative cost cycle can be obtained by using a predecessor subgraph.

Function INITIALIZE()
1: **for** $(i = 1$ to $n)$ **do**
2: **for** $(j = 1$ to $n)$ **do**
3: **if** $(i = j)$ **then**
4: $d_{ij}^{(1)-} := 0.$
5: **else**
6: **if** $(c_{ij} < 0)$ **then**
7: $d_{ij}^{(1)-} := c_{ij}.$
8: **else**
9: $d_{ij}^{(1)-} := \infty.$
10: **end if**
11: **end if**
12: **end for**
13: **end for**
14: **return**

Algorithm 9.3: NCG Edge-Relax Algorithm: Initialization

For the purpose of simplifying the composition of Chapters 9.2.1 and 9.2.2, we define the following:

1. Let f_1 be the **for** loop from lines 1 to 18 in Algorithm 9.4.

2. Let f_2 be the **for** loop from lines 7 to 17 in Algorithm 9.4.

3. Let f_3 be the **for** loop from lines 9 to 16 in Algorithm 9.4.

Function EDGE-RELAX($G, StartV, EndV$)

1: **for** ($k = 2$ to n) **do**
2: **for** ($u = 1$ to n) **do**
3: **for** ($v = 1$ to n) **do**
4: $d_{uv}^{(k)-} := \infty$.
5: **end for**
6: **end for**
7: **for** ($t = 1$ to m) **do**
8: $i := StartV[t]; j := EndV[t]$.
9: **for** ($r = 1$ to n) **do**
10: **if** ($d_{ri}^{(k-1)-} < 0$ and $d_{ri}^{(k-1)-} + c_{ij} < 0$) **then**
11: $d_{rj}^{(k)-} := \min\{d_{ri}^{(k-1)-} + c_{ij}, d_{rj}^{(k-1)-}, d_{rj}^{(k)-}\}$.
12: **if** ($r = j$) **then**
13: **return** ("The negative cost girth is k.")
14: **end if**
15: **end if**
16: **end for**
17: **end for**
18: **end for**
19: **return** ("G does not contain any negative cost cycles.")

Algorithm 9.4: NCG Edge-Relax Algorithm: EDGE-RELAX

9.2.1 Resource Analysis

In the function INITIALIZE (Algorithm 9.3), we initialize all the values in matrix $\mathbf{D}^{(1)-}$. Since this is an $n \times n$ matrix, this process takes $O(n^2)$ time.

For each iteration, we relax all the edges and check the costs from each vertex to the relaxed edge. Since we use the edges stored in the lists $StartV$ and $EndV$, recall that the sizes of these lists are both m. This means f_2 runs $O(m)$ times. For each edge e_{ij}, we scan all the edges incoming to vertex i to determine if $d_{ri}^{(k-1)-} < 0$ and $d_{ri}^{(k-1)-} + c_{ij} < 0$. Consequently, the operations in lines 10 and 11 take constant time, and f_3 runs $O(n)$ times. Thus, the total running time of f_2 is $O(m \cdot n)$ time.

We now need to address f_1. For each iteration, we first initialize the values in $\mathbf{D}^{(k)-}$. Since $\mathbf{D}^{(k)-}$ is an $n \times n$ matrix, initializing $\mathbf{D}^{(k)-}$ takes $O(n^2)$ time. Then, we find all the shortest paths using at most k edges, where $2 \leq k \leq n$. It would appear that f_1 runs $O(n)$ times. However, note that in lines 12 and 13, we halt the algorithm when we find the first negative cycle and return k. This means the algorithm can end before we reach the n^{th} iteration. In fact, since we have

previously defined k as the length of the shortest refutation, we can say that f_1 runs $O(k)$ times. Therefore, the total time of the algorithm is $O(k \cdot (n^2 + m \cdot n)) = O(n^2 \cdot k + m \cdot n \cdot k)$.

We should note that the running time can be improved to $O(m \cdot n \cdot k)$ by modifying how we initialize $\mathbf{D}^{(k)-}$. Instead of using the initialization process in lines 2 to 6 in Algorithm 9.4, we makes two modifications. First, in the INITIALIZE procedure, we add the statement $d_{ij}^{(2)-} = \infty$ after line 12. The first iteration of Algorithm 9.4 requires that $\mathbf{D}^{(2)-}$ is initialized. Otherwise, $d_{rj}^{(2)-}$ could be incorrect in line 11. Second, we add the statement **if** $(k + 1 \leq n)$, **then** $d_{rj}^{(k+1)-} = \infty$ after line 9 in Algorithm 9.4. This allows us to initialize all the values in $\mathbf{D}^{(k+1)-}$ if $k + 1 \leq n$. Both of these modifications are constant time operations which means the running time is now $O(m \cdot n \cdot k)$ since we removed the **for** loop in lines 2 to 6.

For the space analysis, note that $\mathbf{A}$ is stored as an adjacency matrix of size $O(n^2)$, and both $StartV$ and $EndV$ as arrays of size $O(m)$. It would appear that we need $O(n^3)$ space since each matrix $\mathbf{D}^{(k)-}$ is size $n \times n$ and we have n matrices. However, note that at iteration l in the algorithm, where $l \leq n$, we only need the values from $\mathbf{D}^{(l-1)-}$. In other words, matrices $\mathbf{D}^{(1)-}$ to $\mathbf{D}^{(l-2)-}$ are no longer needed. This means we can safely remove them from storage. As a result, at any given iteration l, we use exactly two matrices: $\mathbf{D}^{(k-1)-}$ and $\mathbf{D}^{(k)-}$. Therefore, the total space required is $O(n^2 + m) = O(n^2)$.

9.2.2 Correctness

We first need to show that we can focus only on shortest paths with non-positive costs.

The basis of our approach revolves around the following theorem:

Theorem 9.2.1 *Suppose we are given a directed cycle C of negative cost in network G. There exists a vertex $u \in C$ such that the cost from u to v is negative, $\forall v \in C$, using the unique path from u to v in C.*

Proof: Assume this is not the case. In other words, for all vertices $u \in C$, there exists some vertex $v \in C$, where the cost from u to v is non-negative. In fact, for each vertex $u \in C$, let v be the first vertex in the path starting from u such that the cost from u to v is non-negative. For expositional convenience, we say that u *connects* to v, if v meets the non-negative property stated above and is the first such vertex to do so.

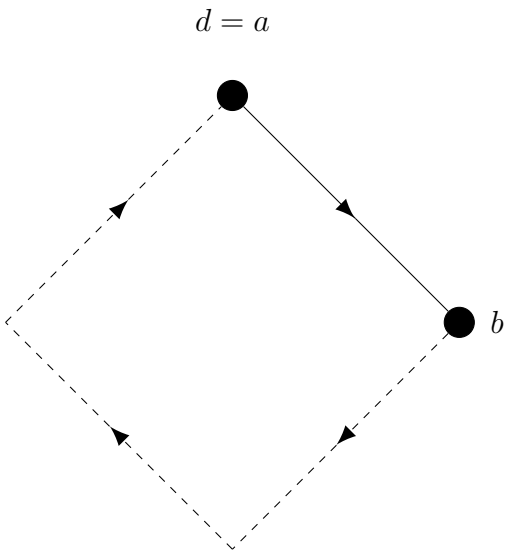


Figure 9.4: Proof of Theorem 9.2.1. Cycle C, where $d = a$.

Suppose we start with some vertex $a \in C$, and let a connect to $b \in C$. Likewise, let b connect to $d \in C$. There are three possible cases for d: d precedes a in the path in C, $d = a$, or d succeeds a in the path in C.

1. Suppose $d = a$, as indicated in Figure 9.4. This means we can add the costs of the paths from a to b and b to $d = a$. Since both of these paths are non-negative, the total cost from a to itself must be non-negative. This contradicts the fact that C is a negative cost cycle.

2. Suppose d succeeds a in the path from b, using the edges in C. Three possibilities arise, as discussed below.

   (a) If $d = b$, then b connects to itself, as we can see in Figure 9.5. This implies that the cost of the path from b to itself is non-negative, thereby contradicting the fact that C is a negative cost cycle.

   (b) Assume that d succeeds a but precedes b (see Figure 9.6). In this case, observe that the cost of the edge incoming to d must be large enough to make the cost of the path from b to d non-negative. Otherwise, b could not connect to d. Further, note that the cost of the path from b to a is negative, since b does not connect to a. Finally, note that the cost of the path from a to d must be negative, since a does not connect to d. It follows that

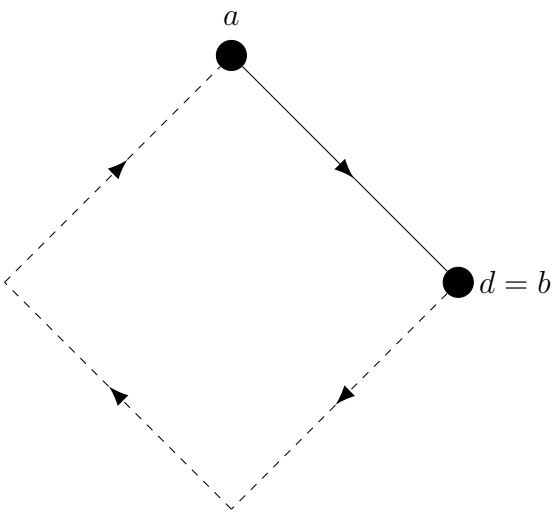


Figure 9.5: Proof of Theorem 9.2.1. Cycle C, where $d = b$.

the cost of the path from b to d, which is the sum of the costs of the paths from b to a and from a to d, must also be negative, which contradicts the choice of d, as the vertex which connects to b, along C.

(c) Suppose that d succeeds b, as shown in Figure 9.7. Note that this is not possible because C is a negative cost cycle, and the cost of the path directly from b to d must be larger than the cost of the path from b to d going once through C.

3. Suppose d precedes a in the path from b, using the edges in C, as shown in Figure 9.8. This means that the cost of the path from a to d is non-negative. We proceed along C from d to identify the first vertex e, such that the vertex that e connects to is either a or a vertex between a and b. Both of these cases have already been covered previously. Note that the existence of e is guaranteed. Otherwise, the cost of the cycle C is non-negative.

□

From Theorem 9.2.1, we only need to find the paths with non-positive costs. This is because if a negative cost cycle exists in the network, we will find one edge of the negative cost cycle in the first iteration of Algorithm 9.4. For each iteration after, we keep finding adjacent edges of the negative cost cycle until the cycle is found in the k^{th} iteration. Therefore, Theorem 9.2.1 holds for the negative cost girth.

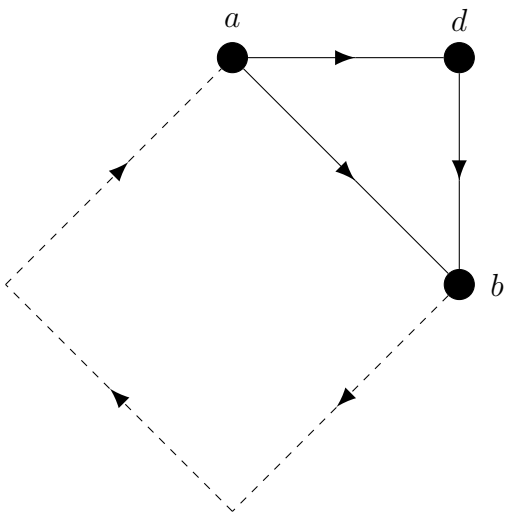


Figure 9.6: Proof of Theorem 9.2.1. Cycle C, where d comes after a but before b.

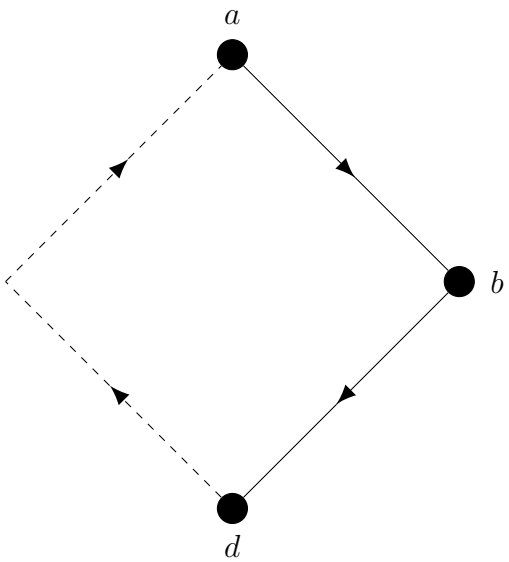


Figure 9.7: Proof of Theorem 9.2.1. Cycle C, where d comes after b.

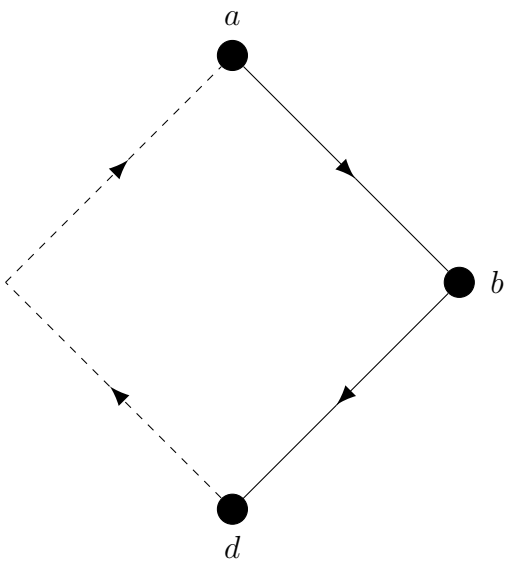


Figure 9.8: Proof of Theorem 9.2.1. Cycle C, where d comes before a.

We have already shown that the ER algorithm terminates since it runs in $O(m \cdot n \cdot k)$ time, where k is the NCG. If the network does not contain a negative cost cycle, the algorithm runs in $O(m \cdot n^2)$ time.

In order to establish the correctness of Algorithm 9.4, we apply Theorem 9.2.1 and observe that the algorithm implements the following dynamic program:

$$d_{rj}^{(k)-} = \begin{cases} c_{rj}, & k = 1 \\ \min_{i \in \mathbf{V},\, d_{ri}^{(k-1)-} < 0,\, d_{ri}^{(k-1)-} + c_{ij} < 0} (d_{ri}^{(k-1)-} + c_{ij}, d_{rj}^{(k-1)-}), & k > 1 \end{cases}$$

The correctness of the above dynamic program follows through an inductive application of the principle of optimality [88], which is stated in Chapter 9.1.2.

9.2.3 Example of the Edge-Relax Algorithm

We now give an example of how the ER algorithm works. Suppose we are given the network G and matrix $\mathbf{A}$ in Figure 9.9 and Table 9.4, respectively. We start the algorithm by initializing all the values in $\mathbf{D}^{(1)-}$. As we can see in Table 9.5, this turns out to be only the edges from $\mathbf{A}$ that have a non-positive cost. Let $k = 2$. We find all the paths with a non-positive cost using at most two edges. For each edge e_{ij}, we determine if there is a path with non-positive cost from

a vertex r to i and if the path from r to i to j is also non-positive. These values are shown in Table 9.6. We now let $k = 3$. Note that when we observe edge e_{cb}, we find that $d_{bc}^{(2)-} = -3$ and $d_{bc}^{(2)-} + c_{cb} = -3 + 2 = -1 < 0$. Since $d_{bb}^{(3)-} < 0$, the algorithm immediately stops and declares that the NCG is 3. The resulting negative cost cycle is shown in Figure 9.10 and Table 9.7.

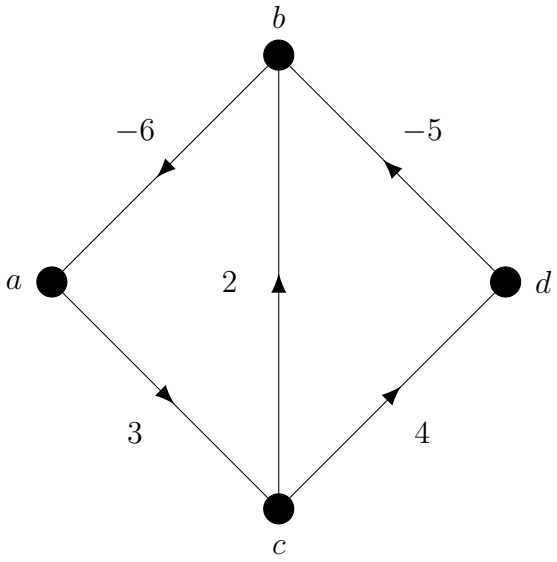


Figure 9.9: Example of the Edge-Relax algorithm. Initial network G.

Table 9.4: Example of the Edge-Relax algorithm. Matrix $\mathbf{A}$.

A	a	b	c	d
a	0	∞	3	∞
b	-6	0	∞	∞
c	∞	2	0	4
d	∞	-5	∞	0

9.3 Empirical Study

In this section, we profile the NCG algorithms discussed in this paper. For the ease of exposition, we refer to the EP and ER algorithms collectively as the new algorithms. We will continue to refer to the algorithm in [15] and Appendix B as the matrix multiplication algorithm.

Table 9.5: Example of the Edge-Relax algorithm. Matrix $\mathbf{D}^{(1)-}$.

$\mathbf{D}^{(1)-}$	a	b	c	d
a	0	∞	∞	∞
b	-6	0	∞	∞
c	∞	∞	0	∞
d	∞	-5	∞	0

Table 9.6: Example of the Edge-Relax algorithm. Matrix $\mathbf{D}^{(2)-}$.

$\mathbf{D}^{(2)-}$	a	b	c	d
a	0	∞	∞	∞
b	-6	0	-3	∞
c	∞	∞	0	∞
d	-11	-5	∞	0

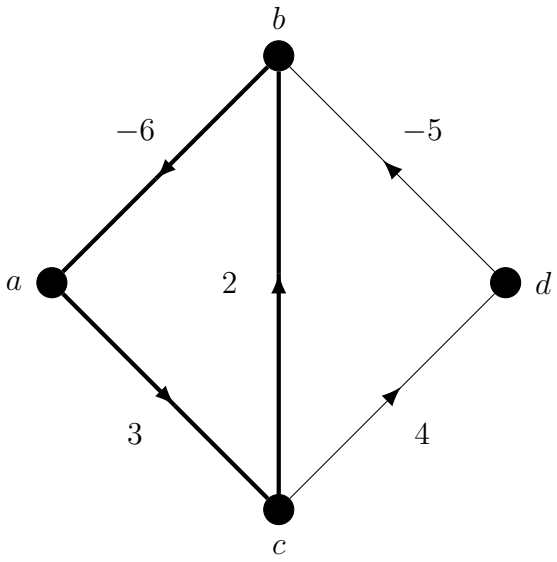


Figure 9.10: Example of the Edge-Relax algorithm. NCG is 3.

Table 9.7: Example of the Edge-Relax algorithm. Matrix $\mathbf{D}^{(3)-}$.

$\mathbf{D}^{(3)-}$	a	b	c	d
a	0	∞	∞	∞
b	-6	-1	-3	∞
c	∞	∞	0	∞
d	-11	-5	-8	0

9.3.1 Experimental Setup

Our experiments study the performance of these algorithms on graphs with varying parameters. We study two graph families that are produced by two generators. The graph generators used are part of the 9th DIMACS Shortest Path Implementation Challenge benchmark package [50].

The generator (SPRAND [51]) creates random graphs with n vertices and $m \geq n$ edges. The generator first constructs a Hamiltonian cycle to ensure the graph is connected. The remaining $m - n$ edges are added by randomly selecting a pair of distinct vertices. Note that the generator can produce parallel edges and/or self-loops. Our experiments include both sparse and dense random graphs of varying sizes. For sparse graphs, we set $m = 4 \cdot n$, where 4 is an arbitrary constant to represent sparse graphs since $m = O(n)$. For dense graphs, we set $m = 0.9 \cdot n^2$.

Since our algorithms are designed for finding negative cost cycles, we choose the edge costs from a fixed set of random integers such that each graph contains at least one negative cycle. We also allow the sizes of the graphs to be $100, 250, 500$, and 750 vertices, and we let k vary from 0 to the number of vertices for each graph, where k is the NCG.

All three algorithms are written in C/C++, and they are compiled and run in identical experimental settings. For the matrix multiplication algorithm, we store $2 \cdot \log n$ adjacency matrices, where n is the number of vertices in the graph, since the algorithm requires the results from the previous iterations for the binary search. For the edge progression algorithm, we also include the adjacency list data structure for storing $Adj(x)$, which contains the edges that are incoming to each vertex x. For the edge relaxation algorithm, we use array data structures for storing $StartV$ and $EndV$, which hold the starting and ending vertices respectively for all edges. Since the OLRR problem is only concerned with the integral domain, we use integer edge costs in our experiments.

Our testing platform is a 2.0 GHz 32-bit Intel Core 2 Duo machine with a 4 GB RAM and a 2 MB cache which runs Ubuntu (version 9.10). The implementations are compiled with the Intel C compiler (icc) version 11.0, and the optimization flag is set to -O3. We report the average execution time of ten independent trials for each test which is common is previously known experiments [51, 11, 48].

9.3.2 Results and Analysis

We compare the performance of the matrix multiplication (MM), Edge-Progress (EP), and Edge-Relax (ER) algorithms based on the type of graph, the size of the graph, and the negative cost girth k. For all figures, the execution time on the Y-axis is depicted in the logarithmic scale due to the variation of the data collected. We note that $k = 0$ implies that the graph does not contain any negative cost cycles. Finally, the data corresponding to our study can be found in Tables 9.8 and 9.9.

Sparse Random Graphs

We first evaluate the performance of all three algorithms on sparse graphs for increasing values of n. We vary the sizes of the graph from 100 to 750 vertices and let $k = 100$. Figure 9.11 plots the execution times for each algorithm and graph. We can see that as the number of vertices increases, both the EP and ER algorithms run faster than the MM algorithm. Also, we see that as n increases, the ER algorithm outperforms the EP algorithm.

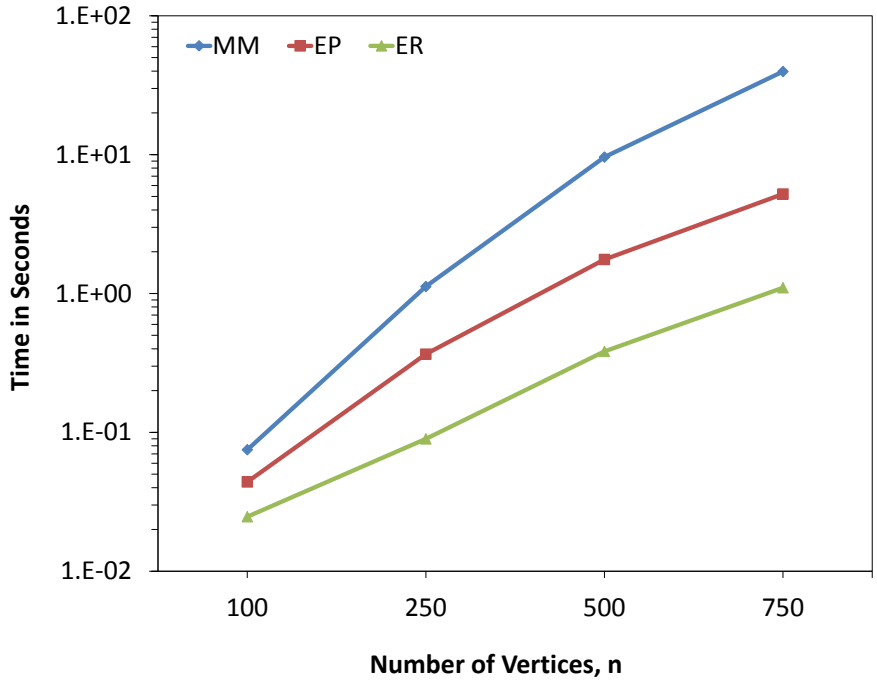


Figure 9.11: NCG performance for a sparse random graph as the size of the graph is varied and $k = 100$.

We next study the performance of all three algorithms for sparse graphs with increasing values of k. Figure 9.12 provides the execution times for a graph with 100 vertices and 400 edges. Although the size of the graph is relatively small, we can see that when $k > 0$, the new algorithms are superior to the MM algorithm, especially for smaller values of k. Further, the ER algorithm runs faster than the EP algorithm in all cases. Finally, we observe that the new algorithms scale at a faster rate as the value of k increases, while the MM algorithm appears to scale at a linear rate.

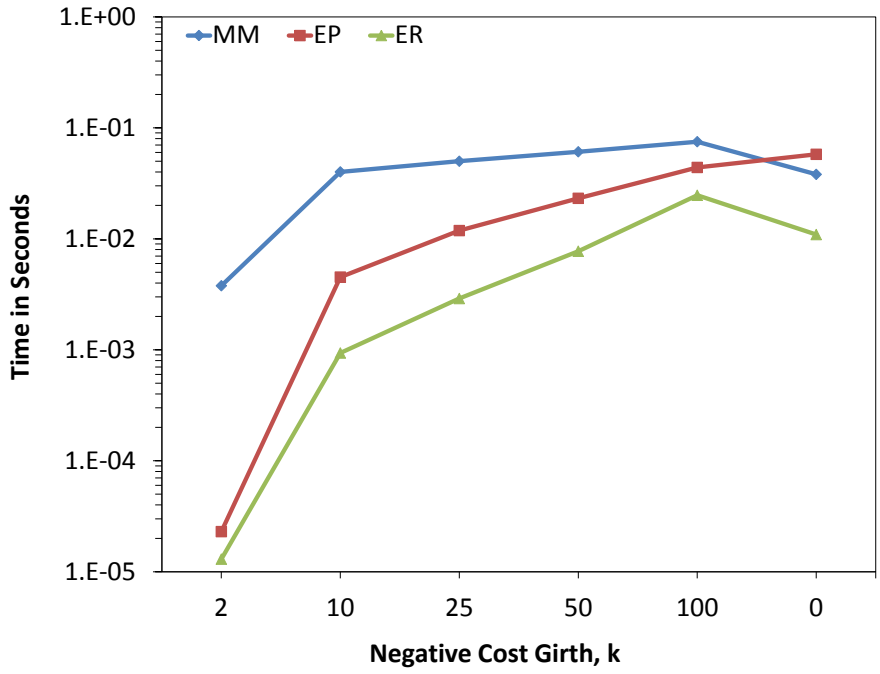


Figure 9.12: NCG performance for a sparse random graph (100 vertices, 400 edges) as the value of k is varied.

Figure 9.13 plots the execution times for a graph with 250 vertices and 1000 edges. Our findings are similar to those indicated in Figure 9.12, in that the new algorithms are faster than the MM algorithm for all values of $k > 0$, and the ER algorithm outperforms the EP algorithm. We encounter the same results for graphs with 500 vertices and 2000 edges (Figure 9.14) and graphs with 750 and 3000 edges (Figure 9.15).

Although the new algorithms perform better when $k > 0$, we note that the EP algorithm is slower than the MM algorithm when $k = 0$. This is because the MM algorithm doubles the value of k and then multiplies the corresponding matrices to find a negative cost cycle. Once a negative

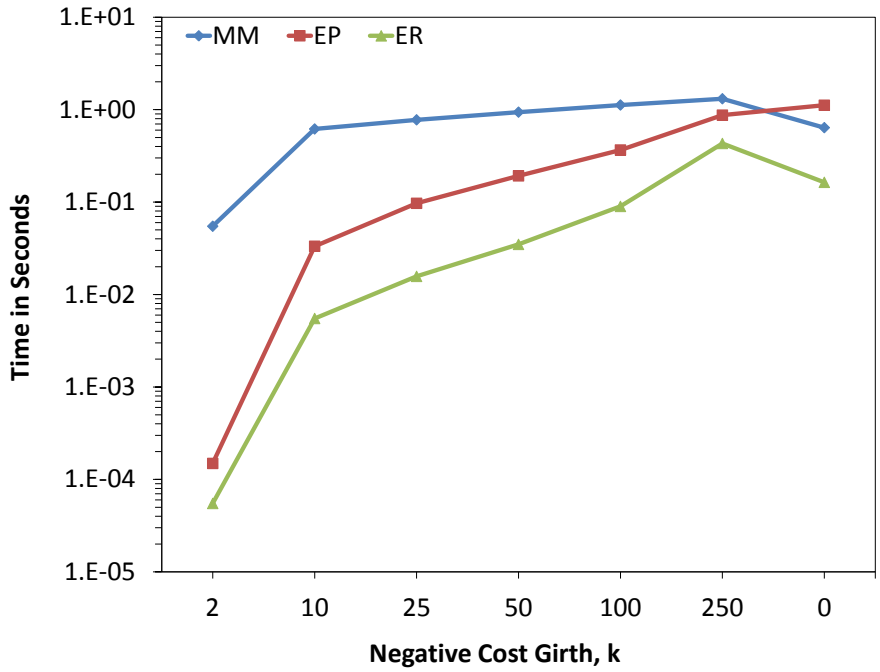


Figure 9.13: NCG performance for a sparse random graph (250 vertices, 1000 edges) as the value of k is varied.

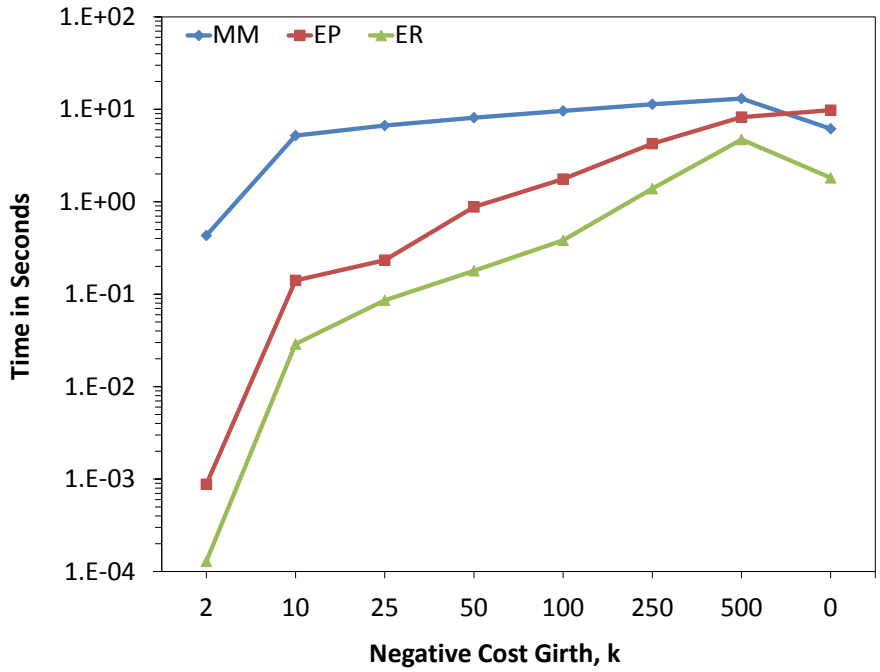


Figure 9.14: NCG performance for a sparse random graph (500 vertices, 2000 edges) as the value of k is varied.

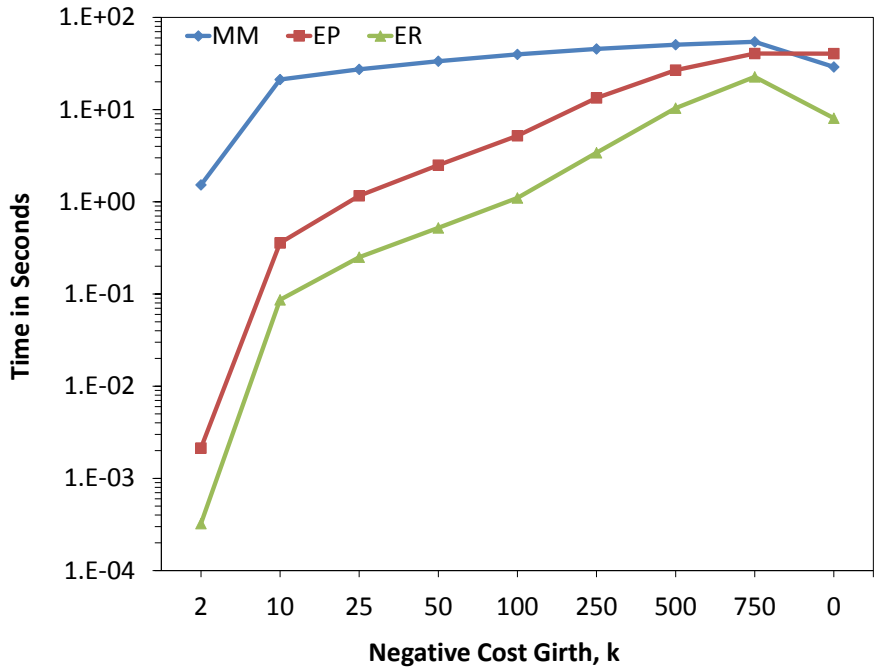


Figure 9.15: NCG performance for a sparse random graph (750 vertices, 3000 edges) as the value of k is varied.

cost cycle has been found, we use a binary search to determine the exact value of k. If the graph does not contain a negative cost cycle, we skip the binary search phase. This means the matrix multiplication algorithm does not have to complete all the steps to conclude that a negative cost cycle is not present. However, the EP algorithm has to run to completion for all values of k, before determining that there are no negative cost cycles.

Based on our observations for sparse graphs, we can conclude that both the EP and ER algorithms have smaller execution times than the MM algorithm. Also, the ER algorithm outperforms the EP algorithm in terms of the execution time for all graph sizes and for all $k \geq 0$.

Dense Random Graphs

We now evaluate the performance of all three algorithms on dense graphs for increasing values of n. We vary the sizes of the graph from 100 to 750 vertices and let $k = 100$. Figure 9.16 plots the execution times for each algorithm and graph. Unlike the cases involving sparse graphs, we can see that as the value of k increases, the new algorithms run slower than the matrix multiplication

Table 9.8: Experiment Results for Sparse Networks (in Seconds)

n	m	k	Matrix Multiplication	Edge-Progress	Edge-Relax
100	400	2	0.004	0.000	0.000
		10	0.040	0.005	0.001
		25	0.050	0.012	0.003
		50	0.061	0.023	0.008
		100	0.075	0.044	0.025
		0	0.038	0.058	0.019
250	1000	2	0.055	0.000	0.000
		10	0.618	0.033	0.006
		25	0.776	0.097	0.016
		50	0.940	0.192	0.035
		100	1.123	0.366	0.090
		250	1.312	0.872	0.431
		0	0.0639	1.117	0.164
500	2000	2	0.432	0.001	0.000
		10	5.200	0.141	0.029
		25	6.676	0.233	0.086
		50	8.120	0.880	0.179
		100	9.608	1.759	0.383
		250	11.349	4.253	1.387
		500	13.059	8.235	4.717
		0	6.161	9.774	1.816
750	3000	2	1.525	0.002	0.000
		10	21.202	0.358	0.086
		25	27.349	1.163	0.251
		50	33.491	2.497	0.521
		100	39.708	5.206	1.101
		250	45.498	13.390	3.404
		500	50.580	26.821	10.370
		750	54.361	40.597	22.654
		0	28.998	40.511	8.073

Note: Any entry with 0.000 means it is less than 0.001.

algorithm. Further, the EP algorithm scales at a significantly faster rate (i.e., degrades quicker) compared to the ER algorithm. Both observations are expected since we are dealing with dense graphs, where the number of edges is $O(n^2)$. This means the EP algorithm and ER algorithm take $O(n^2 \cdot k + n^3 \cdot k)$ time and $O(n^3 \cdot k)$ time respectively, while the MM algorithm takes $O(n^3 \cdot \log k)$ time.

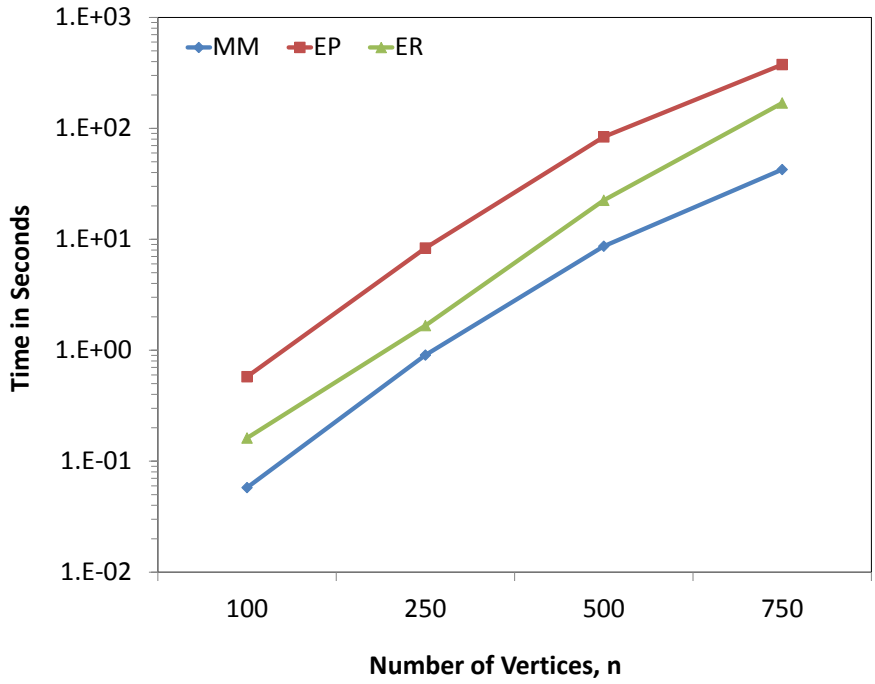


Figure 9.16: NCG performance for a dense random graphs as the size of the graph is varied and $k = 100$.

We next study the performance of all three algorithms for dense graphs with increasing values of k. Figure 9.17 provides the execution times for a graph with 100 vertices and 9000 edges. For the case where $k = 2$, the new algorithms outperform the MM algorithm. However, as k increases, the EP algorithm becomes substantially slower compared to the other two algorithms. Furthermore, the EP algorithm eventually becomes slower than the MM algorithm once the value of k reaches a certain threshold. However, the ER algorithm is still faster than the EP algorithm in all cases.

Figure 9.18 shows the execution times for a graph with 250 vertices and 56250 edges. We observe similar findings to Figure 9.17 in that the Edge-Progress algorithm degrades rapidly as

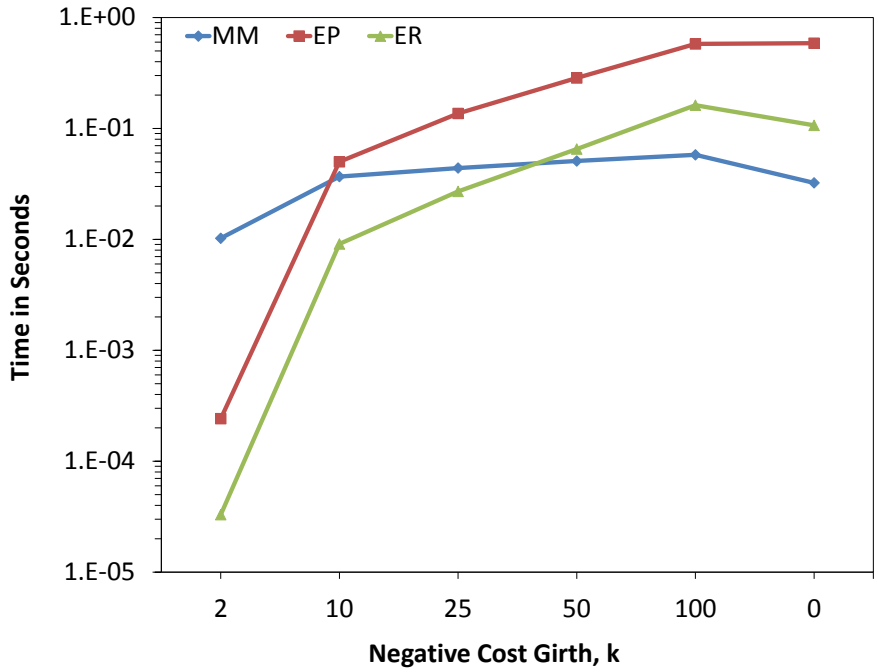


Figure 9.17: NCG performance for a dense random graph (100 vertices, 9000 edges) as the value of k is varied.

the value of k increases, and the ER algorithm becomes worse than the MM algorithm once k surpasses a specific threshold. We encounter the same results for graphs with 500 vertices and 225000 edges (Figure 9.19) and graphs with 750 and 50650 edges (Figure 9.20). Finally, we observe that the new algorithms scale significantly faster (i.e., degrade quicker) than the MM algorithm, which appears to scale at a linear rate with the number of vertices.

For the case where $k = 0$ for all dense graphs, the new algorithms run much slower compared to the MM algorithm. As mentioned in the case for sparse graphs, the matrix multiplication algorithm terminates sooner because it does not have to perform any binary searches due to the absence of negative cost cycles. Further, in the case of dense graphs, both new algorithms must scan through all the edges several times before coming to the conclusion that there are no negative cost cycles. Since we are working with dense graphs, it is no surprise that the new algorithms have very large execution times.

From our experiments in the case of dense graphs, we can conclude that the MM algorithm has a faster execution time than both the ER and the EP algorithms. However, even for dense

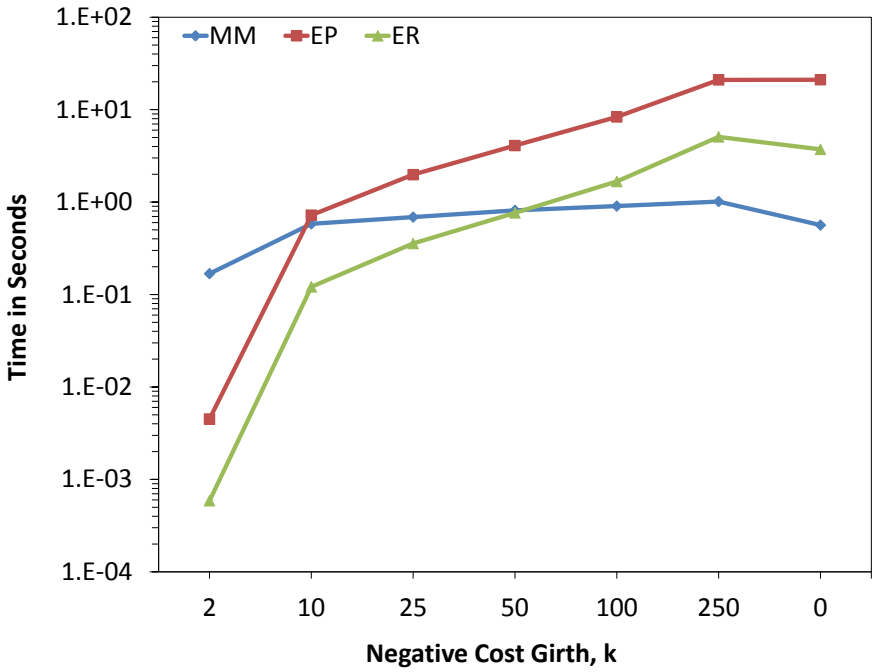


Figure 9.18: NCG performance for a dense random graph (250 vertices, 56250 edges) as the value of k is varied.

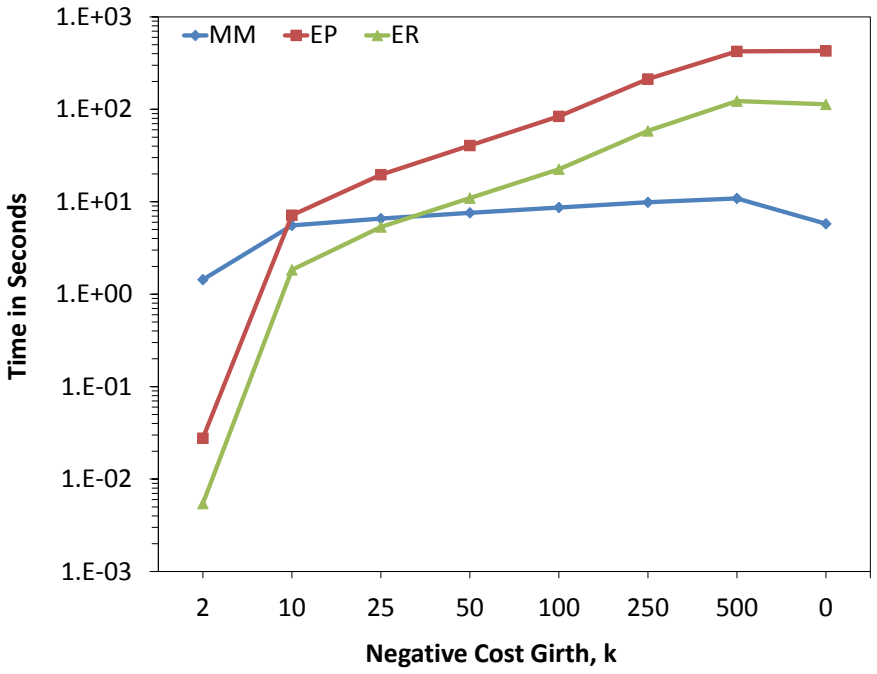


Figure 9.19: NCG performance for a dense random graph (500 vertices, 225000 edges) as the value of k is varied.

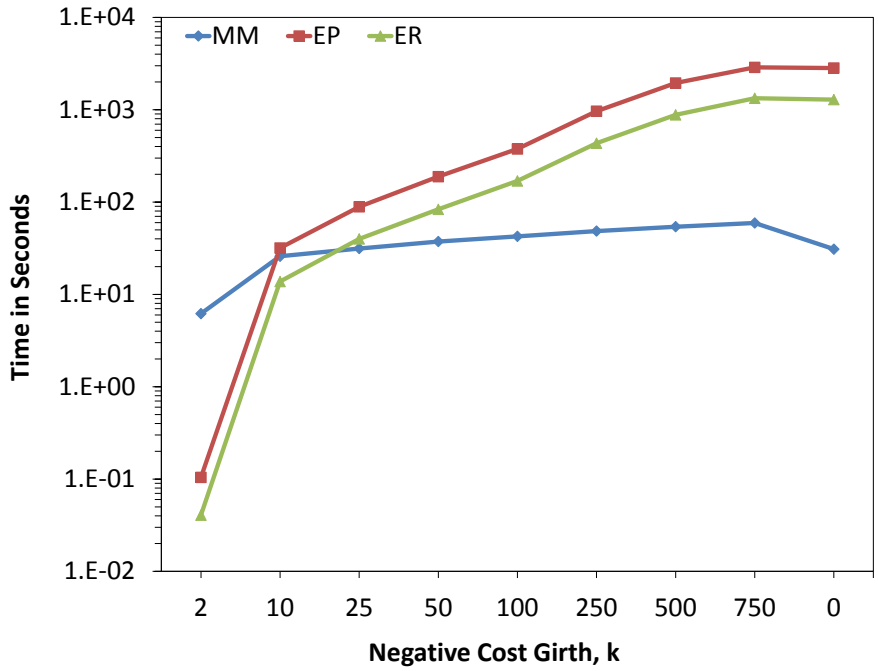


Figure 9.20: NCG performance for a dense random graph (750 vertices, 506250 edges) as the value of k is varied.

graphs, the ER algorithm runs faster than the EP algorithm for all graph sizes and for all $k \geq 0$.

Table 9.9: Experiment Results for Dense Networks (in Seconds)

n	m	k	Matrix Multiplication	Edge-Progress	Edge-Relax
100	9000	2	0.010	0.000	0.000
		10	0.037	0.050	0.009
		25	0.044	0.136	0.027
		50	0.051	0.285	0.065
		100	0.058	0.578	0.162
		0	0.032	0.586	0.107
250	56250	2	0.168	0.005	0.001
		10	0.581	0.722	0.121
		25	0.689	1.980	0.356
		50	0.814	4.089	0.765
		100	0.905	8.344	1.670
		250	1.014	20.989	5.080
		0	0.564	21.080	3.719
500	225000	2	1.436	0.028	0.005
		10	5.547	7.149	1.834
		25	6.569	19.627	5.339
		50	7.587	40.556	10.967
		100	8.672	83.907	22.495
		250	9.867	212.630	58.422
		500	10.868	423.447	122.713
		0	5.776	427.997	113.124
750	506250	2	6.191	0.104	0.041
		10	25.920	31.819	13.789
		25	31.443	88.965	39.969
		50	37.430	188.598	83.468
		100	42.581	376.321	169.568
		250	48.577	965.446	434.232
		500	54.164	1950.350	881.341
		750	59.412	2879.649	1333.006
		0	30.920	2834.284	1289.332

Note: Any entry with 0.000 means it is less than 0.001.

Chapter 10

A Parallel Implementation for the NCG Problem

In this chapter, we discuss the implementation of a parallel algorithm for the NCG problem. Currently, there is an increasing trend in favor of parallel programming. This is because we have a greater need for developing faster solutions and solving problems that are significantly larger in size [89]. There are three potential factors that have contributed to the rise in popularity for parallel processing. First, the cost for hardware has decreased which means it is possible to build systems that contain multiple processors for a reasonable cost. Second, very large scale integration (VLSI) circuit technology [90] has advanced such that we can design complex systems where a single chip can contain millions of transistors. Finally, we are approaching the physical limitations of the von Neumann processor, with respect to the cycle time. We are also finding that there exists a substantially higher resource cost as we attempt increase the performance of a sequential processor.

10.1 Preliminaries

In this section, we discuss several preliminaries prior to presenting the parallel implementation. We first describe the model of computation that is used for the algorithm. We then provide additional definitions and notations pertaining to parallel algorithms and implementations.

10.1.1 Model of Computation

We use the PRAM (Parallel Random Access Machine) model for our parallel implementation. In this model, we have a set of identical processors, although we assume the number of processors is unlimited. The memory is globally shared among all processors, and accessing the shared memory takes constant time. Each processor also has its own local memory (of unlimited size in theory) and can request data from other processors via synchronized communication. Finally, in each unit of time, each processor is allowed to run an instruction or remain idle. In other words, instructions are executed synchronously under the control of a common clock.

To handle shared memory conflicts, we use the CREW (Concurrent Read Exclusive Write) strategy. With this approach, all processors are allowed to read the shared data simultaneously, but only one processor is allowed to write at any given time. For instance, if we consider a parallel approach for matrix multiplication, all processors can retrieve data regarding different elements of the matrices at the same time. However, only one processor can write back to a specific element in a matrix at a time.

However, there are concerns with the PRAM model. The model does not consider the time complexity for synchronizing instructions or communicating between processors. Also, implementing algorithms using the PRAM model are nearly impossible because modern hardware is not capable of some PRAM concepts, such as globally sharing memory. These algorithms are examples of fine-grained parallelism, which occurs when each processor handles a small amount of data to compute, but there exists a large number of processors and a high amount of communication between all processors. The problem is that the number of processors is usually unrealistic. When implementing these algorithms, a more practical approach is to use coarse-grained parallelism, where each processor handles a much larger amount of the data for computation, but the number of processors and the frequency of communication among the processors are more realistic.

10.1.2 Definitions

Suppose we are given problem Q of size n. Let us assume there exists a parallel algorithm $\mathcal{A}$ that solves the input problem using $P(n)$ processors. The running time of $\mathcal{A}$, denoted as $T(n)$, is the maximum time spent among all $P(n)$ processors. The total *work* done for A is the product of

the running time of $\mathcal{A}$ and the number of processors, denoted as $W(n) = T(n) \cdot P(n)$.

Let $T^*(n)$ be the running time of the fastest known sequential algorithm for solving Q. $\mathcal{A}$ is *work-efficient* if $T(n) = O(\log^k n)$ and $P(n) = O(n^c)$, where k and c are both fixed constants. In other words, a parallel algorithm is work-efficient if it runs in polylogarithmic time using a polynomial number of processors. $\mathcal{A}$ is *work-optimal* if $W(n) = T(n) \cdot P(n) = T^*(n)$. In other words, a parallel algorithm is work-optimal if the total running time among all $P(n)$ processors is the same as the running time of the fastest known sequential algorithm.

For instance, if we use the parallel matrix multiplication algorithm from [91] and [22], we can solve the all pairs shortest paths problem in $O(\log^2 n)$ parallel time using $O(n^3)$ processors. While this algorithm is work-efficient, it is not work-optimal since the Floyd-Warshall algorithm [10] runs in $O(n^3)$ time.

10.2 Related Work

The extant NCG algorithms involve finding shortest paths from all vertices, and this chapter presents a parallel implementation for solving the NCG problem. Therefore, it is appropriate to discuss parallel algorithms for solving the all pairs shortest path (APSP) problem.

There exist several sequential algorithms for the APSP problem, such as [38], [76], and [10]. However, these algorithms require at least $n - 1$ recursive steps, in the worst case, which means the parallel running time is at least some order of n, regardless of the number of processors. For instance, a straightforward parallel implementation of [10] runs in $O(n)$ parallel time using $O(n^2)$ processors.

For parallel algorithms, there has been much research done to solve the APSP problem and related problems [92, 93, 94, 95]. Kucera [96] provides a parallel algorithm that runs in $O(\log^2 n)$ parallel time using the PRAM model and $O(\log n)$ parallel time using the CRCW (Concurrent Read Concurrent Write) PRAM model [97]. G.H. Chen et al. [93] describes a parallel algorithm for the APSP problem based on the Processor Arrays with Reconfigurable Bus System (PARBS) model. This algorithm runs in $O(\log n)$ parallel time using $n^2 \times n \times n$ processors. Pan and Preparata [98, 99] develop a parallel algorithm that runs in $O(\log^{2.5} n)$ parallel time and $O(n^3)$ operations.

Pan, Han, and Reif [100] improve the previous time bound to $O(f(n)/p + I(n) \cdot \log n)$ parallel time using $O(n^3)$ operations and p processors. In this algorithm, $f(n) = o(n^3)$, and $I(n)$ is the running time to compute the minimum of n elements using n processors. Therefore, $I(n)$ is $O(\log n)$ in the EREW (Exclusive Read Exclusive Write) PRAM model, $O(\log \log n)$ in the CRCW PRAM model, and $O(1)$ in the randomized CRCW PRAM model [101].

10.3 The Parallel Implementation

In this section, we describe the parallel implementation for solving the NCG problem.

The implementation uses the following notation: We are given the network $G = (V, E, c)$, where V is the vertex set with n vertices, E is the edge set with m edges, and $\mathbf{c} : E \to \mathbb{R}$ is the cost function that assigns a real number to each edge in E. We let c_{ij} represent the cost of the edge from vertex i to vertex j. We represent G as an adjacency matrix $A = (a_{ij})$ such that $a_{ij} = c_{ij}$ if there exists an edge e_{ij} with cost c_{ij}, and $a_{ij} = \infty$ otherwise.

We let $\mathbf{D}^{(k)}$ be an $n \times n$ matrix that monitors the shortest path between each pair of vertices using at most k edges, where $1 \leq k \leq n$. For each pair of vertices i and j, $d_{ij}^{(k)}$ is the cost of the shortest path from i to j using at most k edges. We initialize the values in $\mathbf{D}^{(1)}$, as shown in Algorithm 10.1, as follows. For every pair of vertices i and j, if $i = j$, then $d_{ij}^{(1)} = d_{ii}^{(1)} = 0$. If there exists an edge $e_{ij} \in \mathbf{E}$, then $d_{ij}^{(1)} = c_{ij}$. Otherwise, $d_{ij}^{(1)} = \infty$.

Our implementation mimics matrix multiplication in parallel. From [91, 22], we can multiply two $n \times n$ matrices in $O(\log n)$ parallel time using $O(n^3)$ processors by calculating each of the n^3 products in its own processor, and each processor takes $O(\log n)$ time to sum the products.

Algorithm 10.2 provides the main algorithm. We use repeated squaring to compute $\mathbf{D}^{(k)}$ by calling the PRODUCT procedure, which is shown in Algorithm 10.3. This continues until we have a matrix $\mathbf{D}^{(k)}$ such that there exists a negative value on the diagonal of $\mathbf{D}^{(k)}$. If this occurs, we have a negative cost cycle and call the BINARY-SEARCH procedure, provided in Algorithm 10.5, to find the smallest value of k, where k is the NCG. If we do not find a negative cost cycle, the algorithm reports that no negative cost cycles were found.

Algorithm 10.3 mirrors the parallel algorithm for matrix multiplication [91, 22]. To compute the shortest path, we use $O(n^3)$ processors to calculate each cost from i to j using at most l_3 edges,

Function INITIALIZE ()

1: $n = |V|$
2: **for** $(i = 1$ to $n)$ **do in parallel**
3: **for** $(j = 1$ to $n)$ **do in parallel**
4: **if** $(i = j)$ **then**
5: $d_{ij}^{(1)} = 0.$
6: **else**
7: **if** $(e_{ij} \in E)$ **then**
8: $d_{ij}^{(1)} = c_{ij}.$
9: **else**
10: $d_{ij}^{(1)} = \infty.$
11: **end if**
12: **end if**
13: **end for**
14: **end for**
15: **return**

Algorithm 10.1: Parallel NCG Algorithm: Initialization

Function NCG-PARALLEL ()

1: $k = 1$
2: $found =$ **false**.
3: **while** $(found =$ **false and** $k \leq n)$ **do**
4: $k = 2 \cdot k.$
5: $\mathbf{D}^{(k)} =$ MATRIX-MULTIPLICATION$(\mathbf{D}^{(k/2)}, \mathbf{D}^{(k/2)})$.
6: **for** $(i = 1$ to $n)$ **do in parallel**
7: **if** $(d_{ii}^{(k)} < 0)$ **then**
8: $found =$ **true**.
9: **end if**
10: **end for**
11: **end while**
12: **if** $(found =$ **true**$)$ **then**
13: BINARY-SEARCH(k).
14: **return**
15: **end if**
16: **return** ("There are no negative cost cycles.")

Algorithm 10.2: Parallel NCG Algorithm: NCG-PARALLEL

where l_3 is the sum of l_1 and l_2. All calculations are merged into a single matrix $\mathbf{D}^{(l_3)}$ by finding the minimum cost for each pair of vertices i and j.

Function MATRIX-MULTIPLICATION $(\mathbf{D}^{(l_1)}, \mathbf{D}^{(l_2)})$
1: Create $\mathbf{D}^{(l_3)}$.
2: A copy of $\mathbf{D}^{(l_3)}$ exists in each processor.
3: **for** $(i = 1$ to $n)$ **do in parallel**
4: **for** $(j = 1$ to $n)$ **do in parallel**
5: **for** $(r = 1$ to $n)$ **do in parallel**
6: $d_{ij}^{(l_3)} = d_{ir}^{(l_1)} + d_{rj}^{(l_2)}$.
7: **end for**
8: **end for**
9: **end for**
10: **for** $(i = 1$ to $n)$ **do in parallel**
11: **for** $(j = 1$ to $n)$ **do in parallel**
12: Let $\mathbf{S}_{ij}$ contain the n different values for $d_{ij}^{(l_3)}$.
13: $d_{ij}^{(l_3)} =$ MERGE-MIN$(\mathbf{S}_{ij}, 1, n)$.
14: **end for**
15: **end for**
16: **return** $(\mathbf{D}^{(l_3)})$

Algorithm 10.3: Parallel NCG Algorithm: MATRIX-MULTIPLICATION

Algorithm 10.4 gives the MERGE-MIN procedure that finds the minimum $d_{ij}^{(l_3)}$ for each pair of vertices i and j among all processors. We use an approach similar to MERGE-SORT [22] to divide the values into groups of two and find the minimum of each set of two values. We repeat this procedure until we find the minimum among all the values. This procedure is a simple parallel reduction. However we include it in the algorithm because it is a key component in the implementation and part of the reason we get an efficient parallel running time.

Algorithm 10.5 explains how we use binary search to find the correct value of k. Once we find a negative cost cycle with l edges, where l is a power of 2, we know the NCG is of size at most l and at least $l/2$. This means we can use binary search in the interval $[l/2 + 1, l]$ to find the smallest value of k such that $\mathbf{D}^{(k)}$ has a negative value on its diagonal.

10.3.1 Resource Analysis

In the function INITIALIZE (Algorithm 10.1), we initialize all the values in $\mathbf{D}^{(1)}$. Since this is an $n \times n$ matrix, we use $O(n^2)$ processors, where each processor computes one of the values in

Function MERGE-MIN $(\mathbf{S}, p, r)$
1: **if** $(p < r)$ **then**
2: $q = \lfloor (p + r)/2 \rfloor$.
3: $s_1 =$ MERGE-MIN$(\mathbf{S}, p, q)$.
4: $s_2 =$ MERGE-MIN$(\mathbf{S}, q + 1, r)$.
5: **return** $(\min(s_1, s_2))$
6: **else**
7: **return** $(S[p])$
8: **end if**

Algorithm 10.4: Parallel NCG Algorithm: MERGE-MIN

Function BINARY-SEARCH (k)
1: $found =$ **false**; $kfound =$ **false**.
2: $high = k; low = k/2$.
3: $mid = (high + low)/2$.
4: $r = 4$.
5: **while** $(kfound =$ **false**$)$ **do**
6: $\mathbf{D}^{(mid)} =$ MATRIX-MULTIPLICATION$(\mathbf{D}^{(low)}, \mathbf{D}^{(k/r)})$.
7: **for** $(i = 1$ to $n)$ **do in parallel**
8: **if** $(d_{ii}^{(mid)} < 0)$ **then**
9: $found =$ **true**.
10: **end if**
11: **end for**
12: **if** $(found =$ **true**$)$ **then**
13: **if** $(mid$ is even$)$ **then**
14: $high = mid$; $r = 2 \cdot r$.
15: $mid = (high + low)/2$.
16: $found =$ **false**.
17: **else**
18: $k = mid$.
19: $kfound =$ **true**.
20: **end if**
21: **else**
22: **if** $(mid$ is even$)$ **then**
23: $low = mid$; $r = 2 \cdot r$.
24: $mid = (high + low)/2$.
25: **else**
26: $k = mid + 1$.
27: $kfound =$ **true**.
28: **end if**
29: **end if**
30: **end while**
31: **return** ("The NCG is of size k.")

Algorithm 10.5: Parallel NCG Algorithm: BINARY-SEARCH

$\mathbf{D}^{(1)}$. This takes $O(1)$ parallel time using $O(n^2)$ processors.

We divide our analysis into two parts: finding the first negative cost cycle, and finding the smallest k. For finding the first negative cost cycle, we keep doubling the value of k in Algorithm 10.2. This means the **while** loop in lines 3 to 11 has $O(\log k)$ iterations since we stop at the first k where a negative cost cycle is detected. For each iteration, we call Algorithm 10.3, which is similar to matrix multiplication. This is because we compute each value in a separate processor, meaning we get $O(1)$ parallel time with $O(n^3)$ processors. As for merging all calculations, we call Algorithm 10.4, which takes $O(\log n)$ time per processor since it reflects the MERGE-SORT algorithm. This means the total time of a single iteration is $O(\log n)$ parallel time using $O(n^3)$ processors.

Once all calculations are complete, we use $O(n)$ processors to check the diagonal. Since there are $O(n)$ values in the diagonal, we use one processor for each value. This takes $O(1)$ parallel time using $O(n)$ processors to detect the negative cost cycle. Since we have $O(\log k)$ iterations, finding the first negative cycle takes $O(\log k \cdot \log n)$ parallel time with $O(n^3)$ processors.

We now examine the time it takes to find the smallest k. For the **while** loop in lines 5 to 30 in Algorithm 10.5, we are given the interval $[l/2 + 1, l]$, and we compute $\mathbf{D}^{((l/2+l)/2)}$. Computing the matrix takes $O(\log n)$ parallel time with $O(n^3)$ processors since we are calling Algorithms 10.3 and 10.4. Similar to Algorithm 10.2, finding a negative value within the diagonal takes $O(1)$ parallel time with $O(n)$ processors. However, for each iteration of the **while** loop, we find the middle value of the interval $[l/2 + 1, l]$, where $l/2 + 1 \leq k \leq l$. This means we have $O(\log k)$ iterations. Therefore, the total time to find the smallest k is $O(\log k \cdot \log n)$ parallel time using $O(n^3)$ processors.

Consequently, the parallel NCG algorithm takes $O(\log k \cdot \log n)$ parallel time using $O(n^3)$ processors. From our definition in Chapter 10.1.2, our parallel algorithm is work-efficient since $O(\log k \cdot \log n) \leq O(\log^2 n)$.

10.3.2 Correctness

Our parallel algorithm is identical to the sequential NCG algorithm in [15] and Appendix B, which is known as the matrix multiplication algorithm in Chapter 9, with two exceptions:

1. We provide a more detailed pseudocode for the $O(n^3 \cdot \log k)$ algorithm.

2. For all i, j, and k, $d_{ij}^{(k)}$ is computed in parallel.

Since [15] verifies that the sequential algorithm is correct, it must therefore follow that the parallel algorithm is also correct.

10.4 Empirical Study

In this section, we profile the two implementations discussed in this paper. For ease of exposition, we refer to the matrix multiplication algorithm as the sequential implementation, and the parallel algorithm discussed in this chapter as the parallel implementation.

10.4.1 MPI Implementation

Before explaining the empirical study, we first need to discuss the implementation details of the parallel algorithm. We implement our algorithm in C/C++ using MPI.

We recognize that using $O(n^3)$ processors can be unrealistic if n is large enough. To reduce the number of processors needed for computing $\mathbf{D}^{(l_3)} = \mathbf{D}^{(l_1)} \cdot \mathbf{D}^{(l_2)}$, we use the row-wise block striped decomposition [102]. We partition $\mathbf{D}^{(l_1)}$ into n/p rows, where p is the number of processors. We let n be a multiple of p to ensure that $\mathbf{D}^{(l_1)}$ is evenly partitioned. Each processor contains one of the n/p partitions of $\mathbf{D}^{(l_1)}$, all of $\mathbf{D}^{(l_2)}$, and the n/p entries in $\mathbf{D}^{(l_3)}$ that result from multiplying $\mathbf{D}^{(l_1)}$ and $\mathbf{D}^{(l_2)}$. After each processor computes its respective partition of $\mathbf{D}^{(l_3)} = \mathbf{D}^{(l_1)} \cdot \mathbf{D}^{(l_2)}$, we use the command MPI_ALLGATHER to collect and combine the results so that at least one processor has all of $\mathbf{D}^{(l_3)}$. The main drawback of this approach is that the running time of each processor to multiply the matrices is $O(n^3/p)$. However, this does not negatively impact our results.

For checking the diagonal of $\mathbf{D}^{(l_3)}$ to detect a negative cost cycle, we divide the n entries into p partitions, and let each processor check n/p entries. Recall that we let n be a multiple of p, so each processor checks the same number of entries. If any of the processors detect a negative value, then there exists a negative cost cycle.

A single processor is used as the "host" processor. All sequential processes, such as incrementing k or performing the binary search, are performed using the host processor. Whenever we need to run the matrix multiplication procedure, we notify all p processors about the operation, and each processor handles the partitioning and computing as mentioned above. After all computations are made, the results are given to the host processor, and the sequential process continues as intended. This implies that all parallel computations are completed and sent to the host processor before continuing with the implementation.

10.4.2 Experimental Setup

We study the performance of both the sequential and parallel implementations on graphs with varying parameters, such as the number of vertices and the size of the NCG. The graph generator used is part of the 9th DIMACS Shortest Path Implementation Challenge benchmark package [50].

Our experiments only use random graphs. The generator (SPRAND [51]) creates random graphs with n vertices and $m \geq n$ edges. The generator first constructs a Hamiltonian cycle to ensure the graph is connected. The remaining $m - n$ edges are added by randomly selecting a pair of distinct vertices. Note that the generator can produce parallel edges and/or self-loops. Our experiments specifically use sparse graphs of varying sizes. We note that we performed the same experiments for dense graphs, but the results were identical to sparse graphs. Therefore, the results for dense graphs are omitted.

We are aware that there exist other types of graphs that can be used in the experiment. However, testing our implementations for specific values of k, the size of the NCG, requires us to modify the generator such that it produces a negative cost cycle with no fewer than k edges. This means we need to force the graph generator to produce the required negative cost cycle first and then create the remainder of the graph. Accomplishing this for random graphs is quite simple, whereas achieving this for other types, such as long mesh or square mesh graphs, proves to be complex.

Since our algorithms are designed for finding negative cost cycles, we select the edge costs from a fixed set of integers such that each graph contains at least one negative cycle.

Both algorithms are written in C/C++ using MPI, as we detailed in Chapter 10.4.1, are

compiled and run in identical experimental settings, and store $2 \cdot \log n$ adjacency matrices, where n is the number of vertices in the graph, since these algorithms require the results from previous iterations for the binary search.

Since we use sparse graphs, we let the number of edges, m, be $4 \cdot n$, where n is the number of vertices. Since $m = O(n)$ for sparse graphs, we chose 4 as an arbitrary constant to represent sparse graphs. We allow n to be $128, 256, 512$, and 1024, and we let k, the size of the NCG, be $0 \cdot n, 0.25 \cdot n, 0.50 \cdot n, 0.75 \cdot n$, and $1 \cdot n$. We note that $0 \cdot n$ implies that the graph does not contain any negative cost cycles.

For the parallel implementation, we let p be the number of processors. We vary p starting from 1 and keep doubling p up to 1024. This is because the parallel implementation requires n to be divisible by p such that n/p vertices are calculated for each processor.

We also observe the time spent communicating between processors. This is because one of the issues with MPI is the large amount of time spent communicating between processors. Therefore, we also monitor how much of the total execution time is spent communicating between all p processors.

Finally, we examine the speedup of the parallel algorithm as we increase the number of processors. As defined in [89], we let n be the input size, $T^*(n)$ be the sequential complexity of the problem, and $T_p(n)$ be the running time of the parallel implementation using p processors. The speedup is calculated using the equation,

$$S_p(n) = \frac{T^*(n)}{T_p(n)}.$$

If $S_p(n) = p$, then we achieve linear (ideal) speedup.

Our testing platform is the Frost IBM Blue Gene/L supercomputer. This system consists of 8192 processors, where each processor is a PowerPC-440 CPU with 0.7 GHz and 256 MB RAM, and runs SuSE Linux Enterprise Server 9. The implementations are compiled with the MPI C compiler (mpicc), and the optimization flag is set to -O3. We report the average execution time of ten independent trials for each test.

10.5 Results and Analysis

We compare the performance of the parallel implementation to the performance of the sequential implementation based on the type of graph, the size of the graph, the size of the NCG (k), and the number of processors used (p). We note that $k = 0 \cdot n$ implies that the graph does not contain any negative cost cycles.

10.5.1 Performance Results

We first evaluate the performance of both implementations for increasing values of n. We vary the sizes from 128 to 1024 vertices and let $k = 0.5 \cdot n$. Figure 10.1 provides the execution times for each implementation and graph. A logarithmic scale is used because of the quadratic growth of the execution time. We note that $p = 1$ represents the sequential implementation, while $p > 1$ represents the parallel implementation. The data corresponding to our study can be found in Table 10.1.

As the number of processors increases, the parallel implementation runs faster than the sequential implementation. Since the purpose of parallel algorithms is to run multiple concurrent processors to achieve a faster execution time, this result is expected. What is interesting is the growth rate as the number of vertices increases. For our parallel implementation, as the number of vertices doubles, the execution time increases by about a factor of 9. However, for the sequential implementation, as the number of vertices increase from 512 to 1024, the execution time increases by a factor greater than 10. Following this trend, it appears the sequential implementation will continue to grow, with respect to the execution time, at a faster rate than the parallel implementation.

We next study the performance of both implementations with increasing values of p and k. Figure 10.2 gives the execution times for a graph with 128 vertices and 512 edges. As p doubles, where $p > 2$, the execution times of the parallel implementation decrease by approximately half for all k. This indicates an exponential decay in the execution time as we increase the number of processors.

We also observe that the execution times for $k = 0.75 \cdot n$ and $k = 1 \cdot n$ are almost identical. Recall that in both algorithms, we use repeated squaring to detect a negative cost cycle, which is

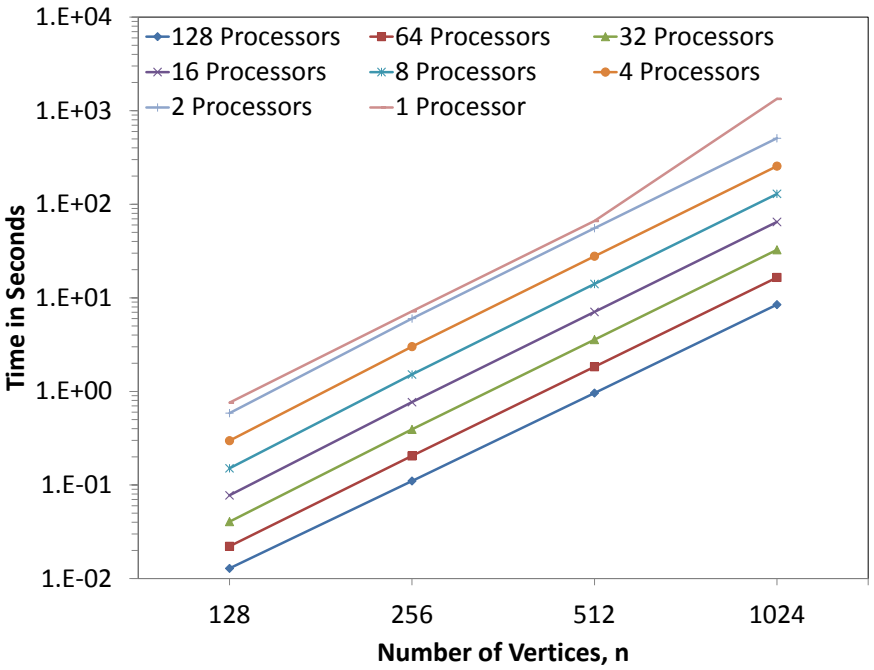


Figure 10.1: NCG performance for sparse random graphs as the size of the graph is varied, $k = 0.50 \cdot n$.

not necessarily the NCG. We then use binary search to determine the exact value of k. Let l be the length, or number of edges, of the negative cost cycle detected using repeated squaring (i.e. before running the binary search). Since we use a doubling technique, l will be the same value when $k = 0.75 \cdot n$ and $k = 1 \cdot n$. We find that the number of binary steps are also identical. The key difference is the number of times a negative cost cycle is detected in the matrix computations. Since our algorithm halts as soon as a negative cycle is detected, in terms of the actual execution time, the case where $k = 0.75 \cdot n$ will run slightly faster than when $k = 1 \cdot n$. Therefore, if $2^r < t < 2^{r+1}$, where t and r are integers and $2^{r+1} \leq n$, then the execution time when $k = t$ will be almost identical to the execution time when $k = 2^{r+1}$.

Finally, we find that the execution time is faster when $k = 0$. In both the sequential and parallel algorithms, we use repeated squaring to detect a negative cost cycle, which is not necessarily the NCG, and then use binary search to determine the exact value of k. If the graph does not have a negative cost cycle, we skip the binary search phase completely. This means neither algorithm has to complete all the steps to conclude that a negative cost cycle is absent.

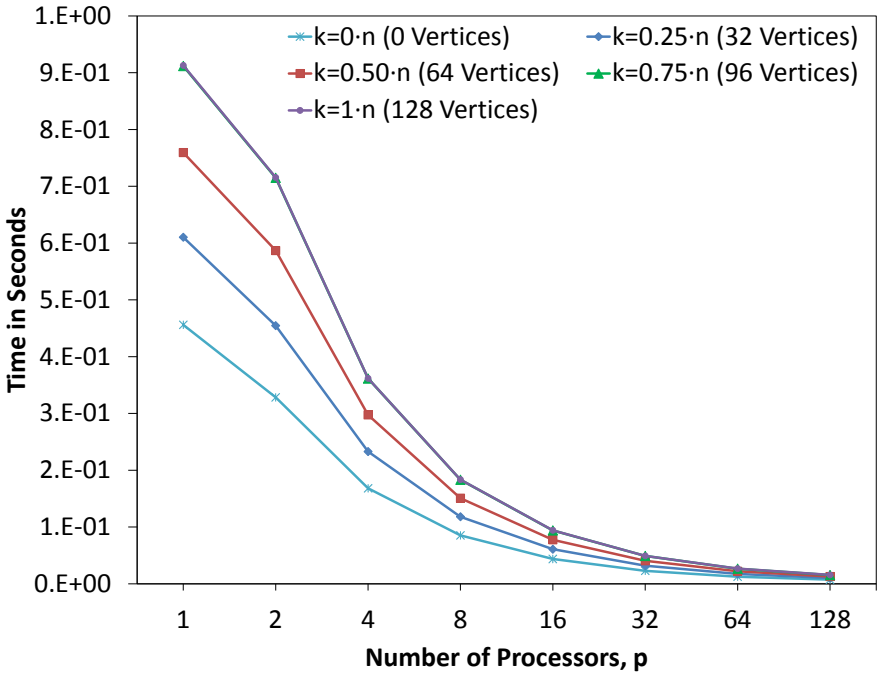


Figure 10.2: NCG performance for sparse random graphs (128 vertices, 512 edges) as the value of k and the number of processors are varied.

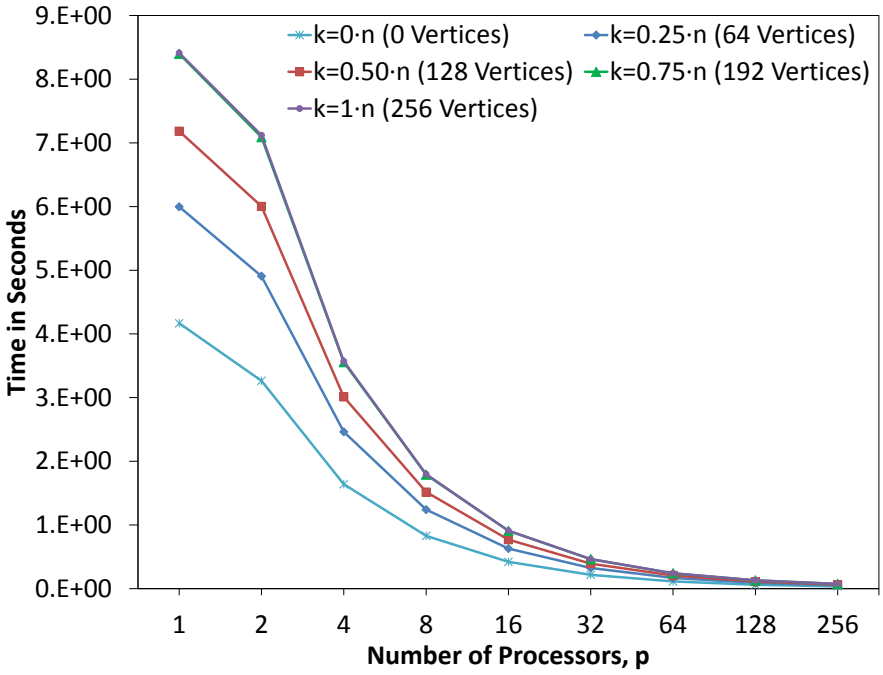


Figure 10.3: NCG performance for sparse random graphs (256 vertices, 1024 edges) as the value of k and the number of processors are varied.

Figure 10.3 plots the execution times for a graph with 256 vertices and 1024 edges. Our findings are similar to those observed in Figure 10.2, in that the execution times of the parallel implementation decrease by about half as p doubles. We observe the same results for graphs with 512 vertices and 2048 edges (Figure 10.4) and graphs with 1024 vertices and 4096 edges (Figure 10.5).

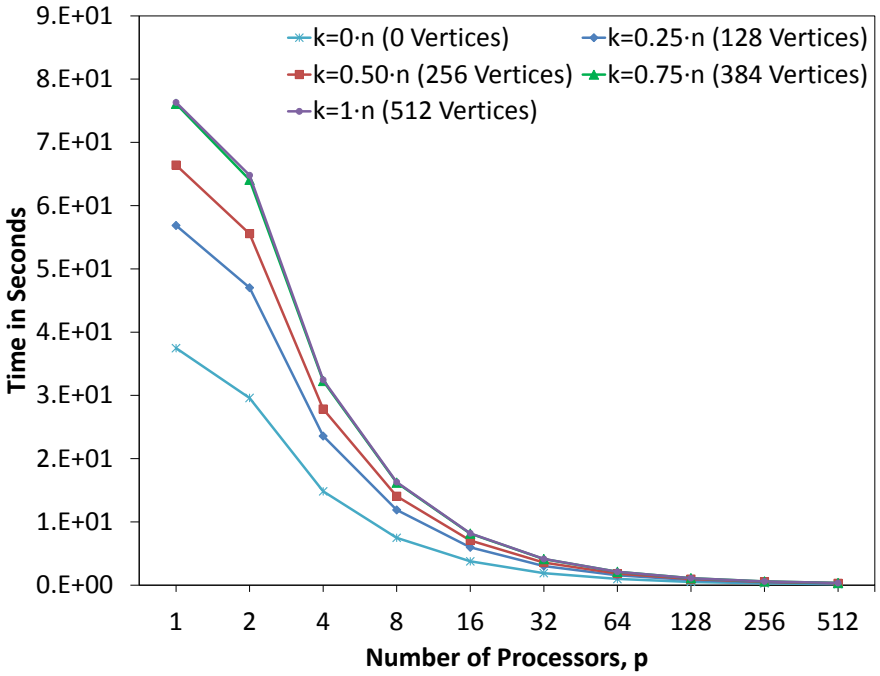


Figure 10.4: NCG performance for sparse random graphs (512 vertices, 2048 edges) as the value of k and the number of processors are varied.

Communication Results

We now examine the communication between the processors in the parallel implementation. This is because a portion of the total execution time is spent sending information among all p processors. Table 10.2 provides the percentage of the time spent communicating between the processors for all values of k. For each k, we examine the number of vertices per processor (i.e., n/p). We observe that the percentage substantially increases as n/p approaches 1, which occurs when the number of vertices equals the number of processors. This means as we reduce the number of vertices per processor (i.e. increase the number of processors), more time is spent

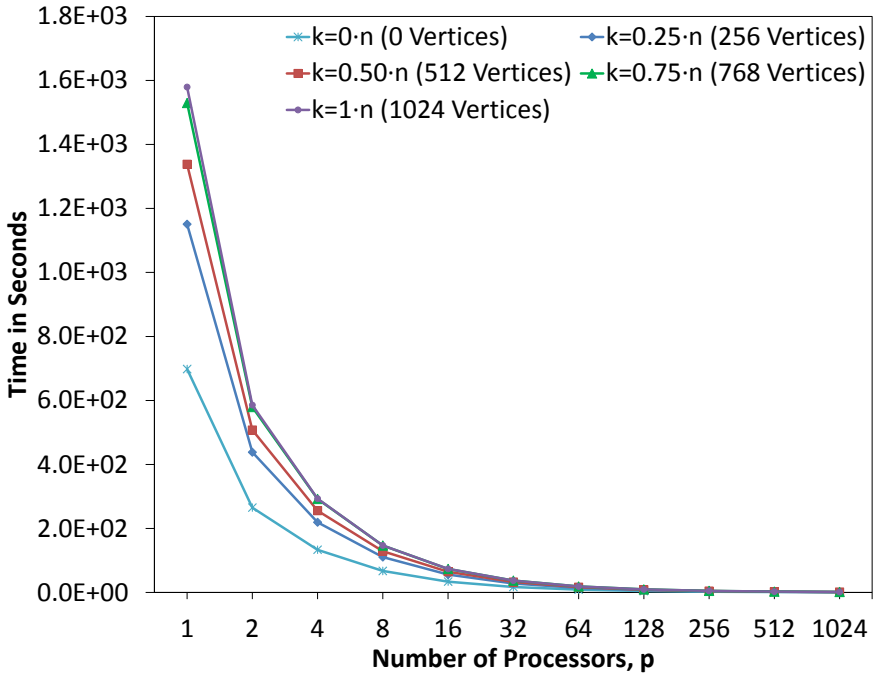


Figure 10.5: NCG performance for sparse random graphs (1024 vertices, 4096 edges) as the value of k and the number of processors are varied.

Table 10.1: Experiment Results for Parallel NCG Implementation (in Seconds)

			Number of Processors, p										
n	m	k	1	2	4	8	16	32	64	128	256	512	1028
128	512	32	0.610	0.455	0.233	0.118	0.061	0.032	0.017	0.010	N/A	N/A	N/A
		64	0.759	0.587	0.297	0.151	0.078	0.041	0.022	0.013	N/A	N/A	N/A
		96	0.912	0.715	0.361	0.183	0.094	0.049	0.027	0.015	N/A	N/A	N/A
		128	0.913	0.716	0.362	0.183	0.094	0.049	0.027	0.016	N/A	N/A	N/A
		0	0.456	0.328	0.168	0.085	0.044	0.023	0.013	0.007	N/A	N/A	N/A
256	1024	64	5.997	4.907	2.462	1.241	0.629	0.323	0.168	0.091	0.052	N/A	N/A
		128	7.184	6.003	3.011	1.516	0.768	0.393	0.205	0.110	0.063	N/A	N/A
		192	8.399	7.091	3.557	1.790	0.907	0.463	0.241	0.129	0.073	N/A	N/A
		256	8.413	7.120	3.570	1.797	0.911	0.464	0.241	0.130	0.074	N/A	N/A
		0	4.167	3.264	1.640	0.828	0.420	0.216	0.112	0.061	0.034	N/A	N/A
512	2048	128	56.867	47.028	23.573	11.892	5.985	3.031	1.557	0.817	0.445	0.257	N/A
		256	66.415	55.573	27.849	14.064	7.072	3.579	1.835	0.961	0.521	0.301	N/A
		384	76.098	64.084	32.284	16.207	8.155	4.125	2.113	1.103	0.597	0.343	N/A
		512	76.317	64.805	32.464	16.294	8.196	4.145	2.123	1.110	0.600	0.346	N/A
		0	37.474	29.601	14.835	7.483	3.770	1.907	0.982	0.516	0.279	0.160	N/A
1024	4096	256	1150.882	438.233	219.403	110.577	55.830	28.173	14.263	7.309	3.827	2.088	1.213
		512	1338.114	507.593	255.618	129.086	64.744	32.626	16.524	8.461	4.428	2.412	1.399
		768	1529.574	579.476	292.205	146.672	73.551	37.063	18.741	9.589	5.001	2.714	1.565
		1024	1579.393	585.636	293.207	146.854	73.714	37.140	18.778	9.615	5.025	2.726	1.570
		0	698.112	265.201	133.441	67.257	33.919	17.134	8.662	4.443	2.317	1.257	0.716

Note: Any entry with N/A means it was not recorded. This is because n cannot be greater than p.

towards communicating with all the processors. Also, we find that the percentage monotonically decreases as k increases. The reasoning for this phenomenon remains unknown.

	k				
n/p	$0 \cdot n$	$0.25 \cdot n$	$0.5 \cdot n$	$0.75 \cdot n$	$1 \cdot n$
1	21.96%	20.38%	19.96%	19.61%	19.47%
2	12.53%	11.44%	11.16%	10.97%	10.96%
4	6.77%	6.13%	5.94%	5.85%	5.84%
8	3.54%	3.18%	3.09%	3.03%	3.02%
16	1.81%	1.63%	1.58%	1.55%	1.54%
32	0.92%	0.83%	0.80%	0.78%	0.78%
64	0.47%	0.42%	0.40%	0.40%	0.39%
128	0.23%	0.20%	0.20%	0.19%	0.19%
256	0.12%	0.10%	0.10%	0.10%	0.10%
512	0.06%	0.05%	0.05%	0.05%	0.05%

Table 10.2: Percentage of execution time communicating with processors

Speedup Results

We now discuss the speedup results of the parallel implementation. Figure 10.6 shows the speedup for a graph with 128 vertices and 512 edges. We observe that as we double the number of processors p, our speedup increases by slightly less than 2. However, the rate of increase actually decreases. This means with larger graph sizes and more processors, we have diminishing returns with more processors. According to Amdahl's Law [103], this is expected.

We also find that for each p, the speedup is slightly less than half of p. While this appears to be a linear scale, since there exists a diminishing return, this is a logarithmic scale if we use more processors. Finally, we do not see any significant change in the speedup as we change the value of k for any graph. This means the size of the NCG does not impact the speedup. This is not surprising since our asymptotic running time contains $\log k$, which means changing k does not substantially affect the execution time. We find similar results for graphs with 256 vertices and 1024 edges (Figure 10.7) and graphs with 512 vertices and 2048 edges (Figure 10.8).

Figure 10.9 plots the speedup for a graph with 1024 vertices and 4096 edges. Similar to previous cases, the speedup nearly doubles as we double the processors, we have diminishing returns as the number of processors increases, and the size of the NCG (k) does not have affect

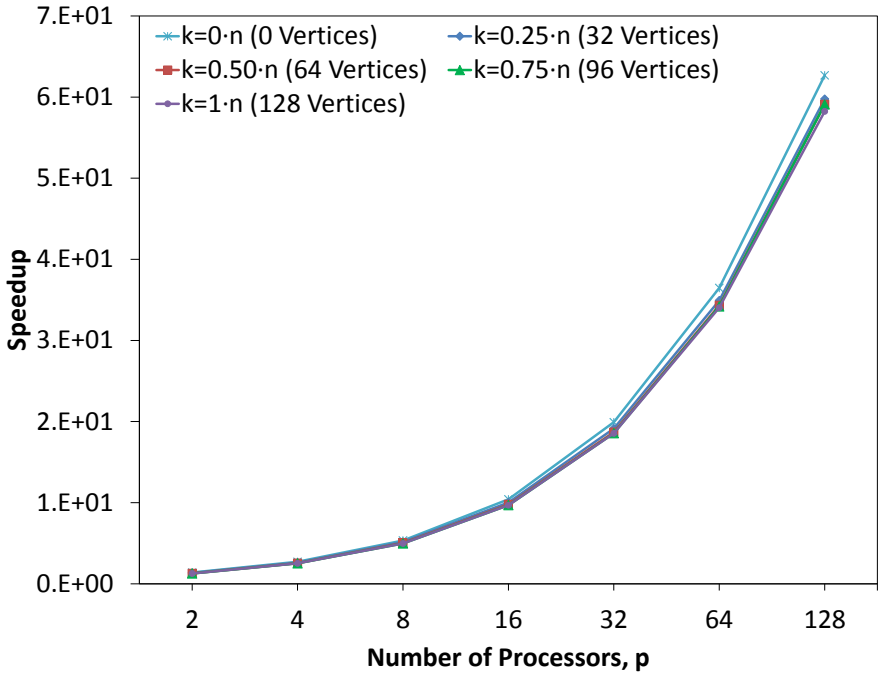


Figure 10.6: Speedup performance for sparse random graphs (128 vertices, 512 edges) as the value of k and the number of processors are varied.

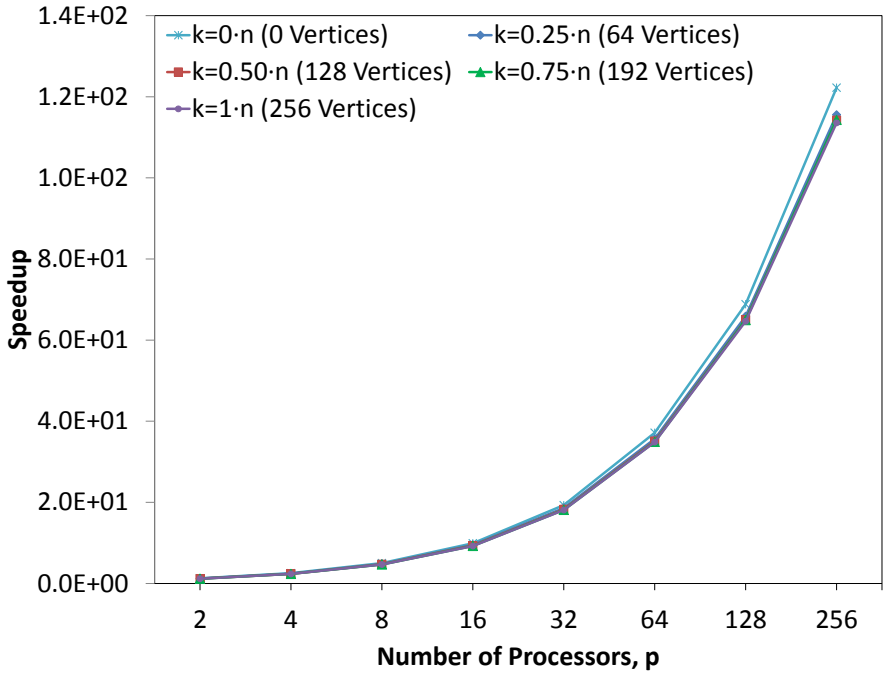


Figure 10.7: Speedup performance for sparse random graphs (256 vertices, 1024 edges) as the value of k and the number of processors are varied.

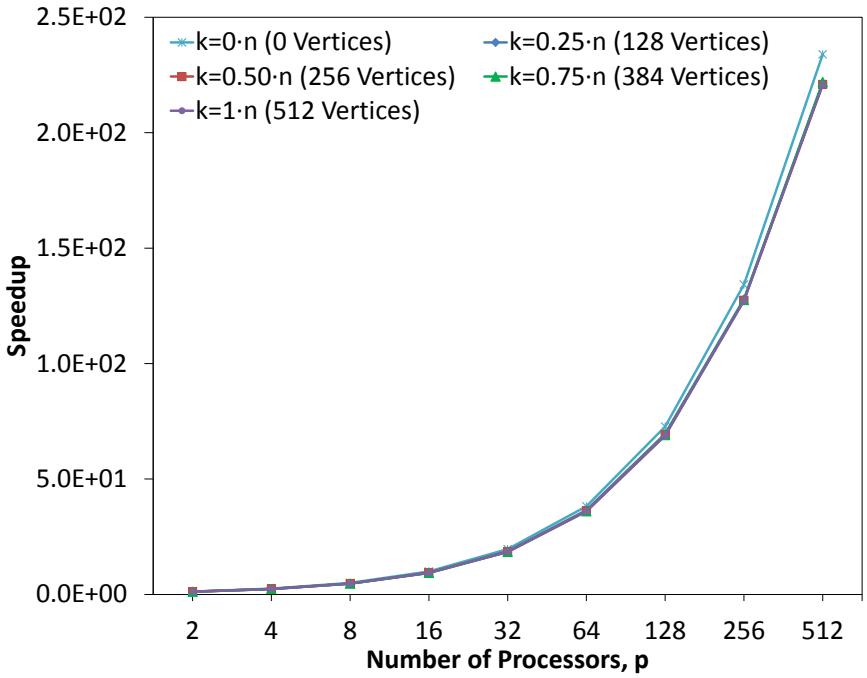


Figure 10.8: Speedup performance for sparse random graphs (512 vertices, 2048 edges) as the value of k and the number of processors are varied.

the speedup. A surprising observation is that the speedup is almost identical to the number of processors. This means we have linear speedup for the case where $n = 1024$. The reasoning behind the improved speedup is unknown.

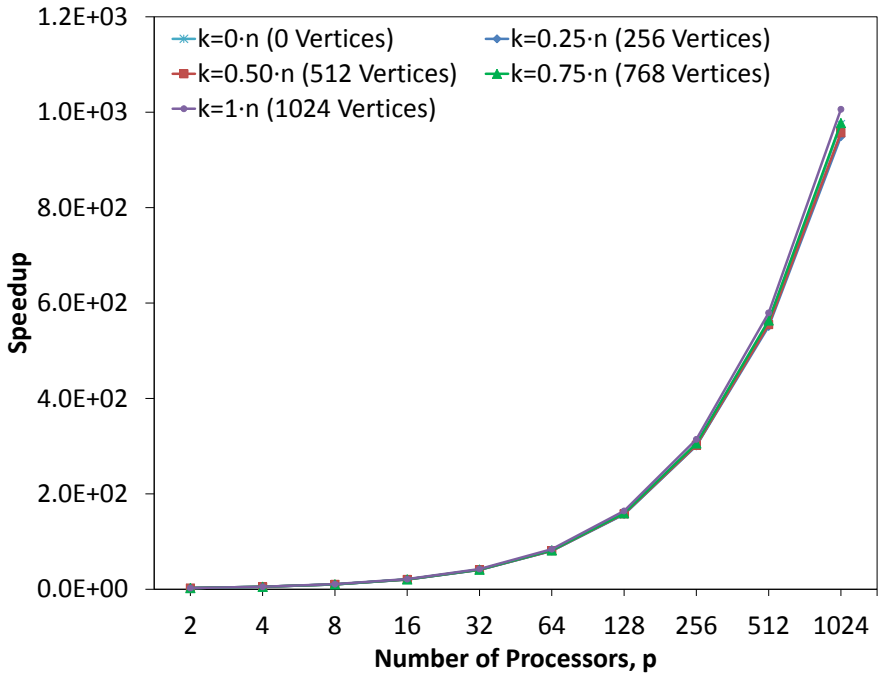


Figure 10.9: Speedup performance for sparse random graphs (1024 vertices, 4096 edges) as the value of k and the number of processors are varied.

Chapter 11

The NCG Algorithm for Planar Networks

In this chapter, we present a new, efficient algorithm for the negative cost girth (NCG) problem in planar, directed networks. We can apply the extant NCG algorithms for general networks to find the NCG in planar networks. However, the extant algorithms are topology-oblivious. The algorithm presented in this chapter exploits the properties of planarity and results in a running time that is superior to any previously known NCG algorithm, even when restricted to planar networks.

Suppose we are given a network G with n vertices, m edges, and negative cost girth k. The current fastest topology-oblivious NCG algorithm is the Edge-Relax algorithm from Chapter 9 that runs in $O(m \cdot n \cdot k)$ time [24]. In a planar network, we must have $m = O(n)$. Thus, for planar networks, the Edge-Relax algorithm takes $O(n^2 \cdot k)$ time. Our NCG algorithm for planar networks runs in $O(n^{1.5} \cdot k)$ time. Further, we can extend our algorithm to general networks that have a separator. In this case, the NCG algorithm runs in $O(n^{a+b} \cdot k + n^d \cdot \log n)$ time, where n^a is the size of the separator, n^b is the number of edges, and we can find the separator in $O(n^d)$ time.

11.1 Related Work

The extant NCG algorithms involve finding shortest paths, and this paper presents an NCG algorithm for planar networks. Therefore, it would be appropriate to first discuss advancements made for solving the shortest paths problem in planar networks.

11.1.1 Shortest Paths in Planar Networks

The first significant contribution for planar networks was made by Lipton and Tarjan with the planar separator theorem in [87]. As an application to the theorem, the first single source shortest path algorithm was proposed by Lipton, Rose, and Tarjan in [104] and ran in $O(n^{3/2})$ time. This algorithm first partitions the network into pieces. Each piece consists of border vertices and internal vertices, where a vertex v is a border vertex of a piece S if $v \in S$ and there exists an edge e_{vx} where $x \notin S$, and a vertex v is an internal vertex of a piece S if $v \in S$ and for all edges e_{vx}, $x \in S$. The algorithm then recursively computes the distances from all border vertices for each piece using multiple iterations of Dijkstra's algorithm [38] to build a dense network. The algorithm then uses the Bellman-Ford algorithm [9] on the resulting dense network to create the solution. This algorithm works for not only planar networks but also any $\sqrt{n}$-separable networks, which are networks that have $O(\sqrt{n})$ size separators.

Frederickson [105] developed the notion of an r-division graph, where the graph is divided into $O(\frac{n}{r})$ regions. Each region consists of r vertices, where $O(\sqrt{r})$ of the r vertices are border vertices. Finding an r-division can be done in $O(n \log n)$ time by using a recursive application of the planar separator theorem [87]. Frederickson then showed how to use a recursive approach for creating r-divisions to compute the shortest paths between all boundary vertices for each region. By combining this approach as preprocessing with a searching method for topology-based heaps, Frederickson has an algorithm for solving the single source shortest path problem that runs in $O(n \cdot \sqrt{\log n})$ time.

Henzinger, Klein, Rao, and Subramanian [106] utilized Frederickson's algorithm [105] to create an algorithm that runs in $O(n^{4/3} \cdot \log^{2/3} D)$ time [106], where D is the sum of the absolute values of the costs. This is accomplished by dividing the network into divisions as specified in [105] and computing the shortest paths from a source s to the border vertices of each region by using Goldberg's $O(\sqrt{n} \cdot m \cdot \log N)$ time algorithm [107], where N is the largest edge cost. The algorithm then computes the shortest paths among all internal vertices for each region.

Fakcharoenphol and Rao [108] presented an algorithm for finding the shortest paths in a planar network with real edge costs in $O(n \cdot \log^3 n)$ time and $O(n \cdot \log n)$ space. The algorithm first recursively decomposes the network by combining the planar separator theorem [104] and

Frederickson's algorithm in [105] to create dense distance networks. The distance matrices corresponding to these networks obey a non-crossing property called the Monge property. The algorithm then combines Dijkstra's algorithm [38] and the Bellman-Ford algorithm [9] with techniques for searching Monge matrices in sublinear time to find the shortest paths among the border vertices of each dense distance network. Using the computed shortest paths, the algorithm finds the shortest paths from a source vertex s to all border vertices.

Klein, Mozes, and Weimann [109] improved the algorithm in [108] to run in $O(n \cdot \log^2 n)$ time and linear space. This algorithm first finds a Jordan curve [110] that passes through $O(\sqrt{n})$ border vertices such that between $\frac{n}{3}$ and $\frac{2 \cdot n}{3}$ vertices are part of the curve. This divides the network into two parts $\mathbf{G}_0$ and $\mathbf{G}_1$. Computing the shortest paths consist of five stages:

1. Recursively compute the distances from an arbitrary border vertex r within each part $\mathbf{G}_i$, for $i = 0, 1$.

2. For each $\mathbf{G}_i$, compute all distances between all boundary vertices in $\mathbf{G}_i$.

3. Use a variant of the Bellman-Ford algorithm and the Monge property to compute distances in $\mathbf{G}$ from r to all boundary vertices.

4. Use the distances from the previous stages to transform the distances in $\mathbf{G}_i$ into non-negative distances and use Dijkstra's algorithm to compute all distances in $\mathbf{G}$ from r to all vertices.

5. Using the distances in $\mathbf{G}$ from r, transform $\mathbf{G}$ such that the distances are non-negative and use Dijkstra's algorithm to compute the distance from the source vertex s to all other vertices.

Although several of the aforementioned shortest path algorithms in planar networks are efficient, they cannot be applied directly to the NCG problem. This is because these algorithms include Dijkstra's algorithm as a subroutine for computing the shortest path distances. To use Dijkstra's algorithm, either the planar network contains edges with strictly non-negative cost, or the network can be transformed such that the edges have non-negative cost. While the shortest path algorithms can be used to detect the presence of a negative cost cycle even when altering the edge costs, the NCG problem requires the edges to remain negative in order to correctly find the NCG.

11.2 Single Vertex Negative Cost Girth

Before describing the NCG algorithm in planar networks, we first need to explain how we can find the NCG for general networks if we are given at least one vertex in the NCG. This is because the algorithm provided in this section serves as a subroutine in the NCG algorithm in planar networks. We assume that if a negative cost cycle exists, then we are also given a vertex s that is in the NCG. If vertex s is not in any negative cost cycles, then our algorithm will not find the NCG. A network may exist such that s is not in the NCG. However, for this algorithm, we are concerned with only the case where s is in the NCG.

Our algorithm uses dynamic programming [22] to find paths of increasing cost from a source vertex s to itself. We let d_k be an array that monitors the shortest path from s, where s is a vertex in the NCG, to all other vertices in V using at most k edges, where $1 \leq k \leq n$. For each vertex v, $d_k(v)$ is the cost of the shortest path from the source vertex to v using at most k edges. We initialize the values in d_1 as follows. Let s be the source vertex that is contained in the NCG. For each vertex $v \in V$, if there exists an edge $e_{sv} \in E$ with cost c_{sv}, then $d_1(v) = c_{sv}$. Otherwise, $d_1(v) = \infty$.

Let k be the number of edges in a path, where $2 \leq k \leq n$. Assume that d_{k-1} has been computed. For each k, we relax all edges in E. For each edge e_{ij} relaxed, we check if $d_k(j) < d_{k-1}(j) + c_{ij}$. Therefore, we compute $d_k(j)$ as,

$$d_k(j) = \min\{d_{k-1}(j), d_{k-1}(i) + c_{ij}\}.$$

Suppose vertex s is known to be in the NCG. After computing d_k, we determine if we have a negative cycle by checking if $d_k(s) < 0$. If this is true, then we have found the negative cost girth, which is k. Otherwise, we repeat the above steps for increasing values of k until a negative cost cycle is found or we conclude that the network does not contain any negative cost cycles containing s.

The above observations are summarized in Algorithm 11.1 and Algorithm 11.2. Observe that Algorithm 11.2 gives us only the NCG. The actual cycle can be obtained by using a predecessor network.

Function INITIALIZE()

1: $n = |V|$.
2: $d_1(s) = 0$.
3: **for** ($v = 2$ to n) **do**
4: **if** ($e_{sv} \in E$) **then**
5: $d_1(v) = c_{sv}$.
6: **else**
7: $d_1(v) = \infty$.
8: **end if**
9: **end for**
10: **return**

Algorithm 11.1: Single Vertex NCG Algorithm: Initialization

Function NCG-SINGLE(G, s)

1: INITIALIZE ()
2: **for** ($k = 2$ to n) **do**
3: **for** ($v = 1$ to n) **do**
4: $d_k(v) = \infty$.
5: **end for**
6: **for** (each edge $e_{ij} \in E$) **do**
7: **if** ($d_k(j) < d_{k-1}(i) + c_{ij}$) **then**
8: $d_k(j) = d_{k-1}(i) + c_{ij}$.
9: **end if**
10: **end for**
11: **if** ($d_k(s) < 0$) **then**
12: **return** ("The negative cost girth is k.")
13: **end if**
14: **end for**
15: **return** ("G does not contain any negative cost cycles.")

Algorithm 11.2: Single Vertex NCG Algorithm: NCG-SINGLE

11.2.1 Resource Analysis

For the purpose of simplifying the composition of the resource analysis, we define the following:

(a) Let f_1 be the **for** loop from lines 2 to 14 in Algorithm 11.2.

(b) Let f_2 be the **for** loop from lines 3 to 5 in Algorithm 11.2.

(c) Let f_3 be the **for** loop from lines 6 to 10 in Algorithm 11.2.

In the function INITIALIZE (Algorithm 11.1) we initialize all the values in the array d_1. Since this array has n items, this process takes $O(n)$ time.

In f_2, we initialize the values in d_k. Since d_k is an array of size n, initializing d_k takes $O(n)$ time. In f_3, we relax all edges and check the costs from each vertex to the relaxed edge. Since we have m edges, relaxing all edges requires $O(m)$ steps. For each edge e_{ij}, we check if $d_k(j) < d_{k-1}(i) + c_{ij}$. If this is true, we change the value of $d_k(j)$. This means the operations in lines 7 to 9 take constant time. Since we have $O(m)$ iterations, f_3 takes $O(m)$ time. The operations in lines 11 to 13 are constant time since we are checking if $d_k(s) < 0$. Thus, the total running time of lines 3 to 13 is $O(m + n)$.

We now need to address f_1. From the discussion above, we know that a single iteration takes $O(m+n)$ time. It would appear that f_1 runs $O(n)$ times. However, note that the algorithm halts at line 12 when we find the first negative cost cycle and return k. This means the algorithm can halt before we reach the n^{th} iteration. Since we previously defined k as the negative cost girth, we can say that f_1 runs $O(k)$ times. Therefore, the total running time of the algorithm is $O(n \cdot k + m \cdot k)$.

We note that the running time can be improved to $O(m \cdot k)$ by modifying how we initialize d_k. Instead of using the initialization process in f_2, we make two modifications.

(1) In the INITIALIZE procedure, we add the statement $d_2(v) = \infty$ after line 8. The first iteration of Algorithm 11.2 requires that d_2 is initialized. Otherwise, $d_2(j)$ could be incorrect in line 8.

(2) We add the statement **if** $(k + 1 \leq n)$, **then** $d_{k+1}(i) = \infty$ and $d_{k+1}(j) = \infty$ after line 6 in Algorithm 11.2. This allows us to initialize all the values in d_{k+1} if $k + 1 \leq n$.

Both of these modifications are constant time operations which means the running time is now $O(m \cdot k)$ since we removed the **for** loop in lines 3 to 5.

For the space analysis, note that G is stored as an adjacency list of size $O(m+n)$ and d is an array of size $O(m)$. It would appear we need $O(m \cdot k)$ space since each list has size $O(m)$ and we have $O(k)$ lists. However, note that in the algorithm at each iteration l, where $l \leq k$, we need only the values from d_{l-1}, and lists d_1 to d_{l-2} are not needed. This means we can safely remove them from storage. At any iteration l, we use exactly two lists: d_{l-1} and d_l. Therefore, the total space required is $O(m+n)$.

11.2.2 Correctness

We have already shown that the algorithm terminates since it runs in $O(m \cdot k)$ time. If the network does not contain a negative cost cycle, then the algorithm runs in $O(m \cdot n)$ time.

In order to establish the correctness of Algorithm 11.2, we observe that the algorithm implements the following dynamic program:

$$d_k(j) = \begin{cases} c_{sj}, & k = 1 \\ \min_{i \in V}\{d_{k-1}(i) + c_{ij}, d_{k-1}(j)\}, & k > 1 \end{cases}$$

The correctness of the above dynamic program follows through an inductive application of the Principle of Optimality [88], which states the following:

An optimal policy has the property that whatever the initial state and initial decision are, the remaining decisions must constitute an optimal policy with regard to the state resulting from the first decision.

11.3 Negative Cost Girth in Planar Networks

In this section, we describe the NCG algorithm in planar networks. Our algorithm consists of two key subroutines. First, we recursively decompose the network into pieces using an approach similar to [108]. We then apply the single vertex NCG algorithm in Chapter 11.2 to all border vertices for each piece. The first negative cost cycle found among all pieces must be the negative

cost girth of the planar network. If k is the number of edges in the first negative cost cycle found, then the size of the NCG is k. The details of our algorithm are explained below.

11.3.1 Planar Network Decomposition

We now explain how to recursively decompose the network. The idea is to divide the network into two non-disjoint pieces, where each piece is stored as a new network. Each piece is then divided into two new non-disjoint pieces, and we continue recursively dividing each piece created until a piece contains a single edge.

Let $G = (V, E, c)$ be the initial network. From the planar separator theorem [87], we can partition V into three sets A, B, and S satisfying the following criteria:

(1) There does not exist any edge $e_{ij} \in E$ such that $i \in A$ and $j \in B$, or $i \in B$ and $j \in A$.

(2) $|A| \leq \frac{2}{3} \cdot n$ and $|B| \leq \frac{2}{3} \cdot n$.

(3) $|S| \leq 2 \cdot \sqrt{2} \cdot \sqrt{n}$.

We use this partition to create two new networks. Let $G_1 = (V_1, E_1, c)$ such that $V_1 = \{v : v \in A \cup S\}$ and $E_1 = \{e_{ij} : i, j \in V_1\}$, and let $G_2 = (V_2, E_2, c)$ such that $V_2 = \{v : v \in B \cup S\}$ and $E_2 = \{e_{ij} : i, j \in V_2\}$. We repeat the recursive decomposition algorithm for both G_1 and G_2 until there exists a network G_i such that $|E_i| = 1$.

Since the decomposition algorithm creates new networks, we need to determine exactly how many networks we have after the algorithm terminates. This is because our NCG algorithm requires us to examine each network created to correctly compute the NCG. Let p be the number of *new* networks created using the planar decomposition algorithm. At the beginning of the algorithm, $p = 0$ since we have only the initial network G.

We let G' be an arbitrary network created using the planar decomposition algorithm. We let p_0 be the total number of networks created after creating G'. If G' consists of exactly one edge, the algorithm returns p_0. Otherwise, we divide the network into two pieces using the approach described above. We first create G_{p_1}, where $p_1 = p_0 + 1$, and run the algorithm recursively with parameters G_{p_1} and p_1. The algorithm returns p_1' as the current number of new networks created after completely decomposing G_{p_1}. We then create G_{p_2}, where $p_1 = p_1' + 1$, and run the

algorithm recursively with parameters G_{p_2} and p_2. The algorithm returns p'_2 as the current number of new networks created after completely decomposing G_{p_2}. Since we do not create any additional networks, the algorithm returns $p = p'_2$ as the total number of networks created.

The above observations are summarized in Algorithm 11.3.

Function PLANAR-DECOMPOSITION (G', p_0)
1: **if** $(|E| > 1)$ **then**
2: Use the planar separator theorem to partition V into sets A, B, and S.
3: $p_1 = p_0 + 1$.
4: Create new network $G_{p_1} = (V_{p_1} = \{v : v \in A \cup S\}, E_{p_1} = \{e_{ij} : i, j \in V_{p_1}\})$.
5: $p'_1 =$ PLANAR-DECOMPOSITION (G_{p_1}, p_1).
6: $p_2 = p'_1 + 1$.
7: Create new network $G_{p_2} = (V_{p_2} = \{v : v \in B \cup S\}, E_{p_2} = \{e_{ij} : i, j \in V_{p_2}\})$.
8: $p'_2 =$ PLANAR-DECOMPOSITION (G_{p_2}, p_2).
9: $p = p'_2$.
10: **return** (p).
11: **else**
12: **return** (p_0).
13: **end if**

Algorithm 11.3: Planar Network Decomposition Algorithm

Resource Analysis

Line 2 of Algorithm 11.3 is the Lipton-Tarjan planar separator theorem [87] and takes $O(n)$ time to separate the network into sets A, B, and S. The rest of the algorithm is similar to the algorithm in [108] except we store each graph created, and we count the number of networks created. Storing the new networks and calculating the number of created networks, denoted as p, are both constant time operations. Therefore, the algorithm runs in $O(n \cdot \log n)$ time.

Correctness

Since the algorithm consists of the algorithms in [87] and [108], we know that the algorithm correctly decomposes the initial network. The remaining concern is whether or not we correctly compute the total number of new networks created, denoted as p. However, note that we increment the total number of new networks by one each time we create a network. Therefore, the algorithm correctly computes p.

11.3.2 Negative Cost Girth Algorithm

We now describe how to find the NCG in planar networks. Suppose we have a network G_i, where $1 \leq i \leq p$, that was created from the planar decomposition algorithm in Chapter 11.3.1, and we want to determine if the NCG exists in G_i. If $|E_i| = 1$, then the NCG clearly cannot be in G_i. Let A_i, B_i, and S_i be the sets formed as a result of the planar separator theorem. For ease of exposition, let G_1 and G_2 denote the new networks created such that $V_1 = \{v : v \in A_i \cup S_i\}$ and $V_2 = \{v : v \in B_i \cup S_i\}$. If the NCG exists in G_i, we have three cases:

(1) The NCG consists of vertices in $A_i \cup S_i$.

(2) The NCG consists of vertices in $B_i \cup S_i$.

(3) The NCG consists of vertices in $A_i \cup B_i \cup S_i$.

If the NCG consists of vertices in $A_i \cup S_i$, then we will find the NCG when we search G_1 in a later iteration. If the NCG consists of vertices in $B_i \cup S_i$, then NCG will be found when we search G_2 in a later iteration. This means the only case we need to address is if the NCG consists of vertices in $A_i \cup B_i \cup S_i$.

If the NCG consists of vertices in all three sets, then the NCG must start with some vertex in A_i (or B_i), pass through a vertex in S_i, reach some vertex in B_i (or A_i), pass through a different vertex in S_i, and return to the original vertex in A_i (or B_i). The key observation is there exist at least two vertices in S_i that must be in the NCG. Otherwise, the NCG can be found in either G_1 or G_2. Therefore, we can use the single vertex NCG algorithm in Chapter 11.2 for all vertices in S_i. Running this algorithm with a vertex in S_i that is in the NCG will return the size of the NCG. Figure 11.1 provides an illustration of the above discussion.

We now need to show how we can search for the NCG in all networks created by the planar decomposition algorithm. If we use a sequential approach and search for the NCG in one network at a time, it is possible that the first negative cost cycle detected is not the one with the fewest number of edges. Instead, for a specific iteration $2 \leq k \leq n$, we will search each network G_i for the shortest paths for all vertices in S_i using at most k edges. For example, we start by finding the shortest paths for all vertices in S_i for all networks G_i using at most two edges. Then, we find the

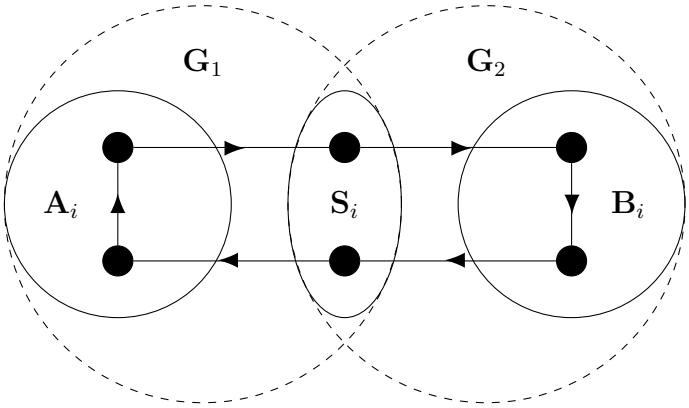


Figure 11.1: Planar NCG Algorithm: At least two vertices in $\mathbf{S}_i$ must be in the NCG.

shortest paths for all vertices in S_i for all networks G_i using at most three edges. We continue this process until we find the first network where the single vertex NCG algorithm detects a negative cycle with k edges. Since all previous iterations produced shortest paths using at most k edges, and no negative cost cycles were found, the size of the NCG must be k.

Observe that the NCG algorithm in planar networks calls the single vertex NCG algorithm for each network G_i, each vertex in S_i, and each iteration k. Therefore, we need to modify the algorithm in Chapter 11.2. First, we change how we initialize d_k for each network by applying the modifications mentioned in Chapter 11.2.1. Since we check each network for each k, the **for** loop where k goes from 2 to n is now outside of the single vertex NCG algorithm. This means the single vertex NCG algorithm must include k as a parameter. Finally, when we run the single vertex NCG algorithm for network G_i with k as a parameter, we have to make sure $k \leq |V_i|$ before running the algorithm. This is because there may exist networks where the number of vertices is smaller than k. Running the single vertex NCG algorithm for these networks would mean we are finding the shortest paths using more edges than a cycle can have in the network. Since this computation does not change the shortest paths, running the algorithm in this case is unnecessary.

The above observations are summarized in Algorithm 11.4, Algorithm 11.5, and Algorithm 11.6. Note that the algorithm gives us only the NCG. We can obtain the cycle representing the NCG by using a predecessor network.

Function NCG-PLANAR (G)

```
1: p = PLANAR-DECOMPOSITION (G, 0).
2: for (i = 1 to p) do
3:    for (each vertex v ∈ S_i) do
4:       INITIALIZE (G_i, v).
5:    end for
6: end for
7: for (k = 2 to n) do
8:    for (i = 1 to p) do
9:       for (each vertex v ∈ S_i) do
10:         NCG-SINGLE (G_i, v, k).
11:         if (NCG-SINGLE returned True) then
12:            return ("The negative cost girth is k.")
13:         end if
14:      end for
15:   end for
16: end for
17: return ("G does not contain any negative cost cycles.")
```

Algorithm 11.4: Planar NCG Algorithm: NCG-PLANAR

Function INITIALIZE (G, s)

```
1: n = |V|.
2: d_1(s) = 0.
3: for (v = 2 to n) do
4:    if (e_sv ∈ E) then
5:       d_1(v) = c_sv.
6:    else
7:       d_1(v) = ∞.
8:    end if
9:    d_2(v) = ∞.
10: end for
11: return
```

Algorithm 11.5: Planar NCG Algorithm: Updated Single Vertex NCG Initialize

Function NCG-SINGLE (G, s, k)
1: $n = |V|$.
2: **if** $(k \leq n)$ **then**
3: **for** (each edge $e_{ij} \in E$) **do**
4: **if** $(k + 1 \leq n)$ **then**
5: $d_{k+1}(i) = \infty$.
6: $d_{k+1}(j) = \infty$.
7: **end if**
8: **if** $(d_k(j) < d_{k-1}(i) + c_{ij})$ **then**
9: $d_k(j) = d_{k-1}(i) + c_{ij}$.
10: **end if**
11: **end for**
12: **if** $(d_k(s) < 0)$ **then**
13: **return** (**True**)
14: **end if**
15: **end if**
16: **return** (**False**)

Algorithm 11.6: Planar NCG Algorithm: Updated Single Vertex NCG

Resource Analysis

Line 1 of Algorithm 11.4 takes $O(n \cdot \log n)$ time since this is the planar network decomposition algorithm, which is Algorithm 11.3.

We next analyze the running time of lines 9 to 14 in Algorithm 11.4. Observe that the NCG-SINGLE procedure (Algorithm 11.6) is similar to Algorithm 11.2 except we run the $O(k)$ iterations outside of the function. Also, since the network is planar, $m = O(n)$. This means Algorithm 11.6 runs in $O(n)$ time rather than $O(m \cdot k)$ time. For the **for** loop in lines 9 to 14 in Algorithm 11.4, observe that $|S_i| = O(\sqrt{n})$ by the planar separator theorem. Since we have $O(\sqrt{n})$ iterations, and each iteration takes $O(n)$ time, the running time of lines 9 to 14 is $O(n^{1.5})$.

We now need to consider the **for** loop in lines 8 to 15 in Algorithm 11.4. Observe we have p iterations, where p is the number of pieces created from the planar network decomposition algorithm from Chapter 11.3.1. However, it is important to note that not all p pieces have the same size (i.e., number of vertices). Let $T(n)$ be the total time to compute the shortest paths for all pieces of a network with n vertices for a single iteration of k. We get the following recurrence relation:

$$T(n) = \max_{n_A, n_B} \{T(n_A \cdot n) + T(n_B \cdot n)\} + c \cdot n^{1.5},$$

where c is some constant, and

$$\begin{aligned} n_A, n_B &\leq 2/3 \\ n_A, n_B &\geq 1/3 \\ n_A + n_B &\leq 1. \end{aligned}$$

We now show that $T(n) = O(n^{1.5})$ by guessing $T(n) \leq d \cdot n^{1.5}$, where $d > 0$ is some constant. Using our guess, we get

$$\begin{aligned} T(n) &\leq \max_{n_A, n_B} \{d \cdot (n_A \cdot n)^{1.5} + d \cdot (n_B \cdot n)^{1.5}\} + c \cdot n^{1.5} \\ &= \max_{n_A, n_B} \{d \cdot n_A^{1.5} \cdot n^{1.5} + d \cdot n_B^{1.5} \cdot n^{1.5}\} + c \cdot n^{1.5} \\ &= \max_{n_A, n_B} \{d \cdot n^{1.5} \cdot (n_A^{1.5} + n_B^{1.5})\} + c \cdot n^{1.5} \\ &= d \cdot n^{1.5} \cdot \max_{n_A, n_B} \{n_A^{1.5} + n_B^{1.5}\} + c \cdot n^{1.5} \end{aligned}.$$

Recall that $n_A, n_B \leq \frac{2}{3}$, meaning $n_A^{1.5} < n_A$ and $n_B^{1.5} < n_B$. Since $n_A + n_B \leq 1$, $n_A^{1.5} + n_B^{1.5} < 1$. Therefore,

$$\begin{aligned} d \cdot n^{1.5} \cdot \max_{n_A, n_B} \{n_A^{1.5} + n_B^{1.5}\} + c \cdot n^{1.5} &\leq d \cdot n^{1.5} \\ c \cdot n^{1.5} &\leq d \cdot n^{1.5} \cdot (1 - \max_{n_A, n_B} \{n_A^{1.5} + n_B^{1.5}\}) \\ c &\leq d \cdot (1 - \max_{n_A, n_B} \{n_A^{1.5} + n_B^{1.5}\}) \\ c/(1 - \max_{n_A, n_B} \{n_A^{1.5} + n_B^{1.5}\}) &\leq d \end{aligned}.$$

This means $T(n) \leq d \cdot n^{1.5}$ when $d \geq c/(1 - \max_{n_A, n_B} \{n_A^{1.5} + n_B^{1.5}\})$. Therefore, $T(n) = O(n^{1.5})$. Since Algorithm 11.5 takes $O(n)$ time, the analysis above can be used to show that lines 2 to 6 take $O(n^{1.5})$ time.

The last part of the analysis is analyzing the running time of lines 7 to 16 of Algorithm 11.4. Note that we have $O(k)$ iterations, and we showed each iteration takes $O(n^{1.5})$ time. This means lines 7 to 16 take $O(n^{1.5} \cdot k)$ time.

When we combine all components of the analysis above, we find that the running time of

the NCG algorithm in planar networks is $O(n \cdot \log n) + O(n^{1.5}) + O(n^{1.5} \cdot k)$. Since $O(n^{1.5} \cdot k)$ dominates all other terms, the total running time is $O(n^{1.5} \cdot k)$.

For the space analysis, note that G is stored as an adjacency list of size $O(n)$ since G is planar. For the ease of exposition, let n_i be the number of vertices of piece G_i, where $1 \leq i \leq p$. Algorithm 11.6 stores the shortest paths in an array of size $O(n_i)$. We also need $O(\sqrt{n_i})$ arrays because we compute the shortest paths from $O(\sqrt{n_i})$ vertices. This means the total space we need for G_i is $O(n_i^{1.5})$. Using the same recurrence relation above for calculating the running time, the total space needed for all pieces of the planar network decomposition is $O(n^{1.5})$.

It would appear we need $O(n^{1.5} \cdot k)$ space since we have $O(k)$ lists and $O(n^{1.5})$ pieces. However, note that in the algorithm at each iteration l, where $l \leq k$, we need only the values from the lists in iteration $l-1$, and the lists from iterations 1 to $l-2$ are not needed. This means we can safely remove them from storage. At any iteration l, we use the lists from iterations $l-1$ and l. Therefore, the total space needed is $O(n^{1.5})$.

We can extend our algorithm for general networks that have a separator. Let $n^a = |S|$, $n^b = m$, and the time to find the separator be $O(n^d)$. Line 1 of Algorithm 11.4 takes $O(n^d \cdot \log n)$ time since this is the network decomposition algorithm, where it takes $O(n^d)$ time to find the separator rather than $O(n)$ time. Algorithm 11.6 takes $O(n^b)$ time, since we scan all the edges in the network. For lines 9 to 14, we have $O(n^a)$ iterations since $n^a = |S_i|$. This means the total time of lines 9 to 14 is $O(n^{a+b})$. For lines 8 to 15 (and lines 2 to 6), we have the following recurrence relation:

$$T(n) = \max_{n_A, n_B} \{T(n_A \cdot n) + T(n_B \cdot n)\} + c \cdot n^{a+b},$$

where c is some constant, $1/3 \leq n_A, n_B \leq 2/3$, and $n_A + n_B = 1$. Using the same approach to solve the first recurrence relation, solving the above equation gives us $T(n) = O(n^{a+b})$. Since we have $O(k)$ iterations in lines 9 to 18, the total time is $O(n^{a+b} \cdot k)$. Thus, combining all components of the analysis gives us a total time of $O(n^{a+b} \cdot k + n^d \cdot \log n)$. We can use the same argument to show that the total space needed is $O(n^{a+b})$.

Correctness

From Chapter 11.3.1, we know line 1 of Algorithm 11.4 correctly decomposes the network. From Chapter 11.2.2, we also know Algorithm 11.6 correctly computes the shortest paths from a given source vertex for each iteration k. Therefore, we only need to prove that the NCG exists in a piece G_i, where $1 \leq i \leq p$, such that at least one vertex of the NCG is in S_i.

Suppose there does not exist a set S_i, where $1 \leq i \leq p$, such that a vertex in the NCG is in S_i. This means for each piece G_i, the NCG is in either A_i or B_i. Without loss of generality, assume all vertices of the NCG are in A_i.

Observe there exists a piece $G_{i'}$ containing the NCG such that $|V_{i'}| \geq k$ and $|A_{i'}| < k$, where k is the size of the NCG. Otherwise, the planar network decomposition algorithm does not create pieces consisting of a single edge (i.e., two vertices). Since $|A_{i'}| < k$, there exists at least one vertex in the NCG, denoted as v, that is not in $A_{i'}$. This means v is in either $S_{i'}$ or $B_{i'}$.

Suppose $v \in S_{i'}$. This contradicts the assumption that there does not exist a set S_i, where $1 \leq i \leq p$, such that a vertex in the NCG is in S_i.

Suppose $v \in B_{i'}$. Since the NCG is a cycle, there must exist a vertex $u \in A_{i'}$ that connects to $v \in B_{i'}$. However, from the planar separator theorem, this cannot occur unless there is some vertex $t \in S_{i'}$ connected to both u and v such that t is also in the NCG. This contradicts the assumption that there is no set S_i such that a vertex in the NCG is in S_i.

We have proven that the algorithm correctly decomposes the planar network into p pieces such that one piece G_i, where $1 \leq i \leq p$, has the set S_i containing at least one vertex in the NCG. Since the NCG-SINGLE procedure correctly detects the NCG if the source vertex is in the NCG, we can conclude that the algorithm correctly computes the NCG.

Chapter 12

Conclusions and Future Work

In this thesis, we presented several algorithms for solving three different problems in network optimization. These problems are known as the minimum spanning tree verification (MSTV) problem, the undirected negative cost cycle detection (UNCCD) problem, and the negative cost girth (NCG) problem. We summarize our conclusions for each problem below.

12.1 The MSTV Problem

We first described how to compute an MST when the graph contains few distinct edge weights. We also discussed a new algorithm for the MSTV problem for the same graph. Both algorithms run in $O(m + n \cdot K)$ time, where K is the number of distinct edge weights. When K is a fixed constant, both algorithms run in linear time.

We also extensively profiled our MSTV algorithm with Hagerup's algorithm. The empirical study indicated that our algorithm is superior to Hagerup's algorithm when K is small, specifically when $K \leq 24$, for random sparse, long mesh, and square mesh graphs. For random dense graphs, our algorithm is superior when $K \leq 32$, but it appears our algorithm remains superior for larger values of K. An interesting observation is that our algorithm is superior to Hagerup's algorithm for all "No" instances regardless of the number of incorrect edges.

12.2 The UNCCD Problem

We next presented the b-matching and the T-join approaches for solving the UNCCD problem. By improving the resource analysis, we have a tighter time bound for the b-matching approach, which runs in $O((n+m)^2 \cdot \log(n+m))$ time. Moreover, when the edge costs are integers in the range $\{-K \cdot \cdot K\}$, we presented efficient algorithms for both approaches along with their parameterized running times in terms of edge costs. When $K = O(1)$, we showed that the b-matching algorithm runs in $O((m+n)^{1.5} \cdot (\log(m+n))^{1.5} \cdot \sqrt{\alpha(m+n, m+n)})$ time, and the T-join algorithm runs in $O(n^{2.5} \cdot (\log n)^{1.5} \cdot \sqrt{\alpha(n^2, n)})$ time, where $\alpha(x, y)$ represents the inverse Ackermann function [21, 22]. Thus, we improved the current time bound for solving the UNCCD problem for both sparse graphs (by using the b-matching approach) and general graphs (by using the T-join approach).

Further, we presented the first extensive empirical study for analyzing negative cycle detection algorithms in undirected graphs. Our results indicated that the b-matching approach is superior to the T-join approach for all sparse graphs. However, for dense graphs, while the b-matching approach is faster than the T-join approach for the torus graphs in our study, the T-join approach is actually superior to the b-matching approach for random graphs. As for the size of K, our study could not conclude a correlation between the size of K and the execution times of either UNCCD algorithm. Finally, our study finds that both UNCCD algorithms run faster when the graph contains either no negative cost cycles or fewer and smaller (i.e., the number of edges) negative cost cycles. The overall results of our empirical study reinforce the asymptotic analysis of both the b-matching and T-join approaches.

12.3 The NCG Problem

We then introduced the NCG problem and presented several algorithms for solving the problem for both general and planar networks. We first proposed and analyzed two strongly polynomial algorithms for the NCG problem. The first of these algorithms, viz., the Edge-Progress algorithm, runs in $O(n^2 \cdot k + m \cdot n \cdot k)$ time, while the second algorithm, viz., the Edge-Relax algorithm, runs in $O(m \cdot n \cdot k)$ time, where k is the NCG. In the case of sparse graphs, both of these algorithms are

asymptotically superior to the matrix multiplication algorithm, which runs in $O(n^3 \cdot \log k)$ time.

Additionally, we extensively profiled the above NCG algorithms, with a view towards determining whether superior asymptotic complexity corresponded to superior empirical performance. Our empirical analysis indicated that the Edge-Progress and Edge-Relax algorithms are superior to the matrix multiplication algorithm in the case of sparse graphs, while the reverse is true in the case of dense graphs. Thus the empirical analysis confirmed the asymptotic analysis. What is surprising though is the superiority of the Edge-Relax algorithm over the Edge-Progress approach, in every single instance.

We then presented a parallel implementation for the NCG problem. This implementation exploits the fact that the algorithm in [15] utilizes a matrix multiplication approach, which can be parallelized. As a result, our implementation runs in $O(\log k \cdot \log n)$ parallel time with $O(n^3)$ processors, where the size of the NCG is k.

We also profiled both the sequential and parallel implementations discussed. Our empirical analysis indicates that the execution time cuts in half as we double the number of processors. We also found that the speedup increases logarithmically as number of processors increases. A surprising observation was the increasing communication time between the processors as we increase the size of the NCG.

Finally, we presented a new algorithm for the NCG problem in planar networks. We first described how to compute the NCG in general networks in $O(m \cdot k)$ time, where k is the size of the NCG, if we are given a single vertex that is in the NCG. We then showed how to compute the NCG in planar graphs in $O(n^{1.5} \cdot k)$ time. This is done by decomposing the planar network into pieces using the planar separator theorem and running the single vertex NCG algorithm for each piece created in the decomposition. Our planar NCG algorithm also works for classes of general networks that have a separator. In this case, the algorithm runs in $O(n^{a+b} \cdot k + n^d \cdot \log n)$ time, where the separator size is n^a, the number of edges is n^b, and the separator can be found in $O(n^d)$ time.

12.4 Future Work

There are several avenues open for future research. We list and detail each open problem below:

1. Analyzing the MSTV algorithm based on randomization - Suppose we are given a graph with K distinct edge weights, where each edge has a weight chosen uniformly and at random among the K distinct edge weights. To show that an input spanning tree is not the MST, for each non-tree edge e_{ij}, we need to find an edge in the path from i to j in the spanning tree whose weight is greater than c_{ij}. Based on how the graph is constructed, each edge in this path has some probability of being greater than c_{ij}. We are interested in seeing if we can utilize this property to show that detecting "No" instances takes expected constant time.

2. Certifying algorithm for the UNCCD problem - Recent work in algorithm design has increasingly emphasized the role the of certifying algorithms [111, 112]. The idea is that the algorithm provides a witness to its output, which is easily certifiable. In the case of the UNCCD problem, a negative cost cycle can serve as a certificate for a "yes" instance. It is not clear what an easily verifiable certificate would be for "no" instances.

3. Scaling algorithm for the NCG problem - For the negative cost cycle detection (NCCD) problem in networks with integer costs, Goldberg [107] designed a scaling algorithm that runs in time $O(\sqrt{n} \cdot m \cdot \log C)$, where n is the number of vertices, m is the number of edges, and C is the largest edge cost (in magnitude). When $C < 2^{\sqrt{n}}$, this algorithm is superior to the Bellman-Ford variants that run in $\Omega(m \cdot n)$ time. We are interested in designing a scaling algorithm for the NCG problem which is superior to the algorithms discussed in this thesis, when the edge costs are small integers.

4. Randomized algorithms for the NCG problem - In Chapter 11, we discussed an NCG algorithm in general networks when we are given a single vertex that is in the NCG. However, it is unrealistic to know in advance even one vertex that is in the NCG without assistance. We would be interested in designing a randomized algorithm. These algorithms have grown in popularity because they are simple to implement, and they are extremely

efficient in the expected case. One area worth exploring is how to select a vertex uniformly and at random that is in the NCG with high probability.

5. Empirical study for the NCG problem in planar networks - In previous studies, there have been situations where algorithms and/or data structures were asymptotically more efficient than other algorithms and/or data structures. However, in practice, these algorithms and/or data structures were actually shown to be less efficient. For instance, Fibonacci heaps are theoretically more efficient than binary heaps. However, it is known that Fibonacci heaps are less efficient than binary heaps in practice due to the constant factors and programming complexity of Fibonacci heaps [22]. Since our NCG algorithm in planar networks is theoretically more efficient than the extant NCG algorithms, we would be interested in implementing our algorithm. This allows us to perform an empirical analysis to compare the performance of our algorithm with other NCG algorithms in planar networks.

Appendix A

A Linear Time Version of Dijkstra's Algorithm

In this appendix, we review the algorithm in [25] for solving the single source shortest path (SSSP) problem when the number of distinct edge costs is bounded by a fixed constant. On a graph $G = (V, E)$ with n vertices, m edges, and K distinct edge costs, the algorithm is a variation of Dijkstra's algorithm that runs in $O(m + n \cdot K)$ time. Note that when $n \cdot K = O(m)$, the algorithm runs in $O(m)$ time.

A.1 Formal Problem Statement

We are given a graph $G = (V, E, c)$, where V is the vertex set with n vertices, E is the edge set with m edges, and $\mathbf{c} : E \to \mathbb{R}$ is the cost function that maps an edge to a real number. For each edge $e_{ij} \in E$, c_{ij} is the cost from vertex i to vertex j. We assume that $c_{ij} > 0$ is a real number.

We represent G as an adjacency list Adj. For each vertex $v \in V$, we let $Adj(v)$ be the set of outgoing edges from v in G. We also let $L = \{l_1, \ldots, l_K\}$ be the set of distinct edge costs, where $l_1 \leq l_2 \leq \cdots \leq l_K$. We assume that $Adj(v)$ is sorted. Otherwise, we can sort L in $O(K \log K)$ time, which is $O(1)$ time if K is a constant. We provide an example of a graph with two distinct edge costs in Figure A.1, where $L = \{3, 5\}$.

We let $s \in V$ be the source vertex of the graph. We also let $\delta(v)$ be the distance of the shortest

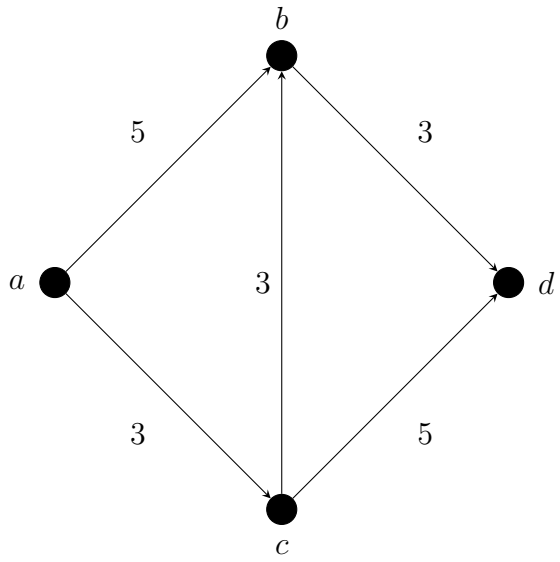


Figure A.1: Example of a graph with 2 distinct edge costs.

path from s to v in the graph. If a path from s to v does not exist, then $\delta(v) = \infty$. The single source shortest path problem is defined as follows:

> *Given a graph $G = (V, E)$ with non-negative edge costs and a source vertex $s \in V$, find the shortest path from s to all other reachable vertices in the graph.*

A.2 Dijkstra's Algorithm in Linear Time

We now describe the single source shortest path algorithm, which is a variation of Dijkstra's algorithm that runs in $O(m + n \cdot K)$ time. This algorithm incorporates additional data structures to achieve the linear running time when K is small. We let S be the set of permanently labeled vertices and $T = V - S$ be the set of temporary labeled vertices. We let $d(j)$ be a distance label for vertex j, where if $j \in S$, then $d(j) = \delta(j)$, which is the cost of the shortest path from a source vertex s to j.

In this algorithm, we let $L = \{l_1, \cdot, l_K\}$ be the set of distinct edge costs. For each distinct edge cost $l_t \in L = \{l_1, \ldots, l_K\}$, we have a linked list $E_t(S) = \{e_{ij} \in E : \ i \in S, c_{ij} = l_t\}$. Each edge e_{ij} is sorted in the order j is added to S. This means if e_{ij} comes before $e_{i'j'}$ in $E_t(S)$, then $d(i) \leq d(i')$.

We let the pointer $CurrentEdge(t)$ be the first edge e_{ij} of $E_t(S)$, where $j \in T$. If $E_t(S)$ has such an edge, then we denote $f(t) = d(i) + l_t$ as the sum of the cost of the shortest path from s to i and the cost of edge e_{ij}. If $E_t(S)$ does not have such an edge, then $CurrentEdge(t) = \emptyset$. We can find the vertex with the smallest distance label in T by finding $argmin\{f(t) : 1 \leq t \leq K\}$, which runs in $O(K)$ time.

We use a subroutine UPDATE(t) to change $CurrentEdge(t)$ such that it either points to the first edge in $E_t(S)$, where the endpoint is in T, or sets $CurrentEdge(t)$ to $\emptyset$. If $CurrentEdge(t)$ points to an edge e_{ij}, then we set $f(t) = d(i) + l_t$. Otherwise, we set $f(t) = \infty$. We denote $CurrentEdge(t).next$ as the operation that moves the pointer $CurrentEdge(t)$ to point to the next edge in $E_t(S)$.

The algorithm runs as follows: We start with the source vertex $s \in V$. We then find $r = argmin\{f(t) : 1 \leq t \leq K\}$ and let edge e_{ij} be $CurrentEdge(r)$. We set $d(j) = d(i) + l_r$ and move j from T to S. For each outgoing edge e_{jk}, we add e_{jk} to the end of $E_t(S)$, where $l_t = c_{jk}$. We then change $CurrentEdge(t)$ if it was initially null. For each distinct edge cost k, we run UPDATE(k), which updates the $CurrentEdge$ pointer for each link list if needed.

The procedure is shown in Algorithms A.1, A.2, and A.3.

A.2.1 Resource Analysis

The initialization process takes $O(n)$ time since it places all adjacent vertices from s into the possible linked lists. Finding the smallest distance label takes $O(K)$ time per iteration. Since we have $O(n)$ iterations, the total time is $O(n \cdot K)$.

As for updating the $CurrentEdge$ pointer, we have two cases: either the pointer changes or it does not change. If the pointer changes, note that the edge at the beginning of the iteration is not scanned again. This is because each edge in $E_t(S)$ is scanned sequentially, meaning each edge is scanned at most once. Therefore, this case runs in $O(m)$ time. If the pointer does not change, nothing changes, and the iteration takes constant time. However, we have $O(n \cdot K)$ iterations. This means the total time is $O(m + n \cdot K)$, which is also the total running time of Algorithm A.2.

Function INITIALIZE()

1: S := $\{s\}$; T := $V - \{s\}$.
2: $d(s) := 0$; $pred(s) := \emptyset$.
3: **for** (each vertex $v \in$ **T**) **do**
4: $d(v) = \infty; pred(v) = \emptyset$.
5: **end for**
6: **for** ($t = 1$ **to** K) **do**
7: $E_t(S) := \emptyset$.
8: CurrentEdge(t) := NIL.
9: **end for**
10: **for** each edge e_{sj} **do**
11: Add e_{sj} to the end of the list $E_t(S)$, where $l_t = c_{sj}$.
12: **if** (CurrentEdge(t) = NIL) **then**
13: CurrentEdge(t) := e_{sj}
14: **end if**
15: **end for**
16: **for** ($t = 1$ to K) **do**
17: UPDATE(t)
18: **end for**

Algorithm A.1: Dijkstra's Linear Time Algorithm: Initialization

Function NEW-DIJKSTRA()

1: INITIALIZE()
2: **while** ($T \neq \emptyset$) **do**
3: let $r = argmin\ \{f(t) :\ 1 \leq t \leq K\}$.
4: let $e_{ij} = CurrentEdge(r)$.
5: $d(j) := d(i) + l_r$; $pred(j) := i$.
6: $S = S \bigcup \{j\}; T := T - \{j\}$.
7: **for** (each edge $e_{jk} \in Adj(j)$) **do**
8: Add the edge to the end of the list $E_t(S)$, where $l_t = c_{jk}$.
9: **if** (CurrentEdge(t) = NIL) **then**
10: CurrentEdge(t) := e_{jk}
11: **end if**
12: **end for**
13: **for** ($t = 1$ to K) **do**
14: UPDATE(t).
15: **end for**
16: **end while**

Algorithm A.2: Dijkstra's Linear Time Algorithm: NEW-DIJKSTRA

Function UPDATE(t)
1: Let $e_{ij} = CurrentEdge(t)$.
2: **if** $(j \in T)$ **then**
3: $f(t) = d(i) + c_{ij}$.
4: **return**
5: **end if**
6: **while** $((j \notin T)$ **and** $(CurrentEdge(t).next \neq NIL))$ **do**
7: Let $e_{ij} = CurrentEdge(t).next$.
8: $CurrentEdge(t) = e_{ij}$.
9: **end while**
10: **if** $(j \in T)$ **then**
11: $f(t) = d(i) + c_{ij}$.
12: **else**
13: Set $CurrentEdge(t)$ to $\emptyset$.
14: $f(t) = \infty$.
15: **end if**

Algorithm A.3: Dijkstra's Linear Time Algorithm: Update

A.2.2 Correctness

Theorem A.2.1 *Algorithm A.2 determines the shortest path from vertex s to all other vertices in $O(m + n \cdot K)$ time.*

Proof: The algorithm is the same as Dijkstra's algorithm. The only difference is that the algorithm uses additional data structures while implementing the FINDMIN() operation. Therefore, Algorithm A.2 correctly finds the shortest paths from vertex s to all other vertices in the graph G.

□

Appendix B

The Matrix Multiplication Approach

In this appendix, we review the matrix multiplication algorithm from [15] that is discussed in Chapters 9 and 10. The algorithm is a dynamic programming approach for finding the negative cost girth of a weighted network.

B.1 Formal Problem Statement

We are given a graph $G = (V, E, c)$, where V is the vertex set with n vertices, E is the edge set with m edges, and $\mathbf{c} : E \to \mathbb{R}$ is the cost function that maps an edge to a real number. For each edge $e_{ij} \in E$, c_{ij} is the cost from vertex i to vertex j.

We represent G as an adjacency matrix $\mathbf{D}$. If there is an edge from vertex i to vertex j, then d_{ij} is set to c_{ij}. Otherwise, d_{ij} is set to ∞. The edge costs are stored in a matrix $\mathbf{W}$, where w_{ij} stores the cost of edge e_{ij} (i.e., c_{ij}). We use the convention $w_{ii} = 0, \forall i = 1, 2, \ldots, n$.

B.2 NCG Algorithm Based on Matrix Multiplication

Let $d_{ij}^{(k)}$ be the cost of the shortest path from vertex i to vertex j using at most k edges. We compute $d_{ij}^{(k)}$ recursively using the already computed value of $d_{ij}^{(k-1)}$. Note that one of the following must be true for $d_{ij}^{(k)}$:

1. The cost of the shortest path from i to j using at most k edges is the same as the cost of the

shortest path from i to j using at most $k-1$ edges. In this case,

$$d_{ij}^{(k)} = d_{ij}^{(k-1)}$$

2. The cost of the shortest path from i to j using at most k edges is less than the cost of the shortest path from i to j using at most $k-1$ edges. This means we can divide the path into two sub-paths: the shortest path from i to r using at most $k-1$ edges, where r is a neighbor of j, and the edge from r to j. This gives us

$$d_{ij}^{(k)} = \min_{1 \leq r \leq n, r \neq j} (d_{ir}^{(k-1)} + w_{rj}).$$

Since the cost of a self-loop is 0, we can combine these two equations which results in the equation,

$$d_{ij}^{(k)} = \min_{1 \leq r \leq n} (d_{ir}^{(k-1)} + w_{rj})$$

Note that $d_{ij}^{(1)} = w_{ij}$, for $i \neq j$, and $d_{ii}^{(1)} = 0$. We are interested in the $d_{ii}^{(k)}$ entries, for $k = 1, 2, \ldots, n$. This is because if $d_{ii}^{(k)} < 0$ for some k, then we know a negative cost cycle exists in G. The smallest k such that $d_{ii}^{(k)} < 0$ is the NCG of G. The above observations are summarized in Algorithm B.1.

B.2.1 Resource Analysis

Computing one $d_{ij}^{(k)}$ takes $O(n)$ time. Let $\mathbf{D}^{(k)}$ be the $n \times n$ matrix that monitors the shortest path between each pair of vertices using at most k edges. Since $\mathbf{D}^{(k)}$ contains $O(n^2)$ entries, calculating all d_{ij} for an arbitrary k takes $O(n^3)$ time. Since $2 \leq k \leq n$, we need to calculate all $O(n)$ matrices, where each matrix takes $O(n^3)$ time. Therefore, the total running time of the original algorithm is $O(n^4)$.

For the space analysis, note that G is stored as an adjacency matrix of size $O(n^2)$. It would appear that we need $O(n^3)$ space since each matrix $\mathbf{D}^{(k)}$ is size $n \times n$, and we have n matrices. However, note that in order to compute $\mathbf{D}^{(k)}$, we only need $\mathbf{D}^{(k-1)}$ and the initial cost matrix $\mathbf{W}$.

Function NCG-MATRIX-MULTIPLICATION ($\mathbf{D}, \mathbf{W}$)
1: $\mathbf{D}^{(1)} := \mathbf{W}$.
2: **for** ($k = 2$ to n) **do**
3: **for** ($i = 1$ to n) **do**
4: **for** ($j = 1$ to n) **do**
5: $d_{ij}^{(k)} = \infty$.
6: **for** ($r = 1$ to n) **do**
7: $d_{ij}^{(k)} = \min\{d_{ij}^{(k)}, d_{ir}^{(k-1)} + w_{rj}\}$.
8: **end for**
9: **end for**
10: **end for**
11: **for** ($i = 1$ to n) **do**
12: **if** ($d_{ii}^{(k)} < 0$) **then**
13: **return** ("The negative cost girth is k.")
14: **end if**
15: **end for**
16: **end for**
17: **return** ("G does not contain any negative cost cycles.")

Algorithm B.1: NCG Matrix Multiplication Algorithm

Therefore, the total space required is $O(n^2)$.

Improving the Running Time

The running time can be reduced to $O(n^3 \cdot \log n)$ by using repeated squaring to compute $\mathbf{D}^{(2)}$, $\mathbf{D}^{(4)}$, and so on instead of computing each matrix sequentially (i.e., $k = 2, 3, \ldots, n$). If we detect a negative cost cycle in matrix $\mathbf{D}^{(l)}$, then the NCG is at most l and greater than $l/2$. We use binary search in the interval $[l/2 + 1, l]$ to find the smallest r such that $\mathbf{D}^{(r)}$ has a negative cost cycle. Finding $\mathbf{D}^{(l)}$ requires $O(\log n)$ matrix multiplications, and the binary search requires $O(\log n)$ additional matrix multiplications. Therefore, the running time is $O(n^3 \cdot \log n)$.

While the algorithm runs in $O(n^3 \cdot \log n)$ time, the repeated squaring halts when we first find a negative cycle. Since this requires $O(k)$ iterations in the binary search, the matrix multiplication algorithm runs in $O(n^3 \cdot \log k)$ time, where the NCG has k edges. Since we have to retain all the product matrices $\mathbf{D}^{(2)}, \mathbf{D}^{(4)}, \ldots, \mathbf{D}^{(2^{\lceil \log k \rceil}}$, the space required is now $O(n^2 \cdot \log k)$.

B.2.2 Correctness

We first observe that there exists a negative cost cycle in G, if and only if $d_{ii}^{(k)} < 0$, for some $i = 1, \ldots, n$ and $k = 2, \ldots, n$. To observe this, note that if one of the diagonal entries, say $d_{rr}^{(k)}$, is negative, then by definition, the shortest path from vertex r to itself has negative cost. Hence, there must be a negative cost cycle around r having at most k edges. Likewise, if there exists a negative cost cycle around vertex r with at most k edges, then the shortest path from r to itself having at most k edges is negative (i.e., $d_{rr}^{(k)} < 0$).

Since Algorithm B.1 examines the network for negative cost cycles in order of increasing size, it follows that the NCG is identified in lines 11 to 14.

References

[1] O. Borůvka, "O jistém problému minimálním," *Práce Moravské Přtrodovědecké Společnosti v Brně*, vol. 3, pp. 37–58, 1926.

[2] Joseph. B. Kruskal, "On the shortest spanning subtree of a graph and the traveling salesman problem," *Proceedings of the American Mathematical Society*, vol. 7, pp. 48–50, 1956.

[3] J. Komlos, "Linear verification for spanning trees," in *SFCS '84: Proceedings of the 25th Annual Symposium onFoundations of Computer Science, 1984*, Washington, DC, USA, 1984, pp. 201–206, IEEE Computer Society.

[4] David R. Karger, Phillip N. Klein, and Robert Endre Tarjan, "A randomized linear-time algorithm to find minimum spanning trees," *Journal of the ACM*, vol. 42, no. 2, pp. 321–328, 1995.

[5] Bernard Chazelle, "A minimum spanning tree algorithm with inverse-ackermann type complexity," *J. ACM*, vol. 47, pp. 1028–1047, November 2000.

[6] R. L. Graham and Pavol Hell, "On the history of the minimum spanning tree problem," *IEEE Ann. Hist. Comput.*, vol. 7, no. 1, pp. 43–57, 1985.

[7] Valerie King, "A simpler minimum spanning tree verification algorithm," *Algorithmica*, vol. 18, no. 2, pp. 263–270, 1997.

[8] Torben Hagerup, "An even simpler linear-time algorithm for verifying minimum spanning trees," in *WG*, 2009, pp. 178–189.

[9] R. E. Bellman, "On a routing problem," *Quarterly of Applied Mathematics*, vol. 16, pp. 87–90, 1958.

[10] Robert W. Floyd, "Algorithm 97: Shortest path," *Commun. ACM*, vol. 5, no. 6, pp. 345, 1962.

[11] Boris V. Cherkassky and Andrew V. Goldberg, "Negative-cycle detection algorithms," in *Algorithms—ESA '96, Fourth Annual European Symposium*, Josep Díaz and Maria Serna, Eds., Barcelona, Spain, 1996, vol. 1136 of *Lecture Notes in Computer Science*, pp. 349–363, Springer.

[12] K. Subramani, "Stressing is better than relaxing for negative cost cycle detection in networks," in *Proceedings of the* 4^{th} *International Conference on Ad-Hoc, Mobile and Wireless Networks (ADHOC-NOW)*, V. R. Syrotiuk and E. Chávez, Eds. October 2005, vol. 3738 of *Lecture Notes in Computer Science*, pp. 320–333, Springer-Verlag.

[13] K. Subramani and D. Desovski, "On the empirical efficiency of the vertex contraction algorithm for detecting negative cost cycles in networks," in *Proceedings of the* 5^{th} *International Conference on Computational Science (ICCS)*, et. al. Peter Sloot, Ed. May 2005, vol. 3514 of *Lecture Notes in Computer Science*, pp. 236–247, Springer-Verlag.

[14] K. Subramani, "A Zero-Space algorithm for negative cost cycle detection in networks," *Journal of Discrete Algorithms*, vol. 5, no. 3, pp. 408–421, 2007.

[15] K. Subramani, "Optimal length resolution refutations of difference constraint systems," *Journal of Automated Reasoning (JAR)*, vol. 43, no. 2, pp. 121–137, 2009.

[16] K. Subramani, C. Tauras, and K. Madduri, "Space-time tradeoffs in negative cycle detection - an empirical analysis of the stressing algorithm," *Applied Mathematics and Computation*, vol. 215, no. 10, pp. 3563–3575, 2010.

[17] Jack Edmonds, "An introduction to matching," in *Mimeographed notes, Engineering Summer Conference*, The University of Michigan, Ann Arbor, MI, 1967.

[18] R. K. Ahuja, T. L. Magnanti, and J. B. Orlin, *Network Flows: Theory, Algorithms and Applications*, Prentice-Hall, 1993.

[19] J. Edmonds and E.L. Johnson, "Matching, euler tours and the chinese postman problem," *Mathematical Programming*, vol. 5, pp. 88–124, 1973.

[20] B. Korte and J. Vygen, *Combinatorial Optimization*, Number 21 in Algorithms and Combinatorics. Springer-Verlag, New York, 4^{th} edition, 2010.

[21] Robert Endre Tarjan, "Efficiency of a good but not linear set union algorithm," *J. ACM*, vol. 22, pp. 215–225, April 1975.

[22] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein, *Introduction to Algorithms*, The MIT Press, Cambridge, MA, 3rd edition, 2009.

[23] Xiaofeng Gu, Kamesh Madduri, K. Subramani, and Hong-Jian Lai, "Improved algorithms for detecting negative cost cycles in undirected graphs," in *Proceedings of the* 3^{rd} *Annual International Frontiers of Algorithmics Workshop*, Xiaotie Deng, John E. Hopcroft, and Jinyn Xe, Eds. June 2009, vol. 5598 of *Lecture Notes in Computer Science*, pp. 40–50, Springer-Verlag.

[24] K. Subramani, Matthew Williamson, and Xiaofeng Gu, "Improved algorithms for optimal length resolution refutation in difference constraint systems," *Formal Aspects of Computing*, vol. 25, no. 2, pp. 319–341, 2013.

[25] James B. Orlin, Kamesh Madduri, K. Subramani, and M. Williamson, "A faster algorthm for the single source shortest path problems with few distinct postive lengths," *Journal of Discrete Algorithms*, vol. 8, no. 2, pp. 189–198, 2010.

[26] R.C. Prim, "Shortest connection networks and some generalizations," *Bell Sys. Tech. Journal*, vol. 36, pp. 1389–1401, 1957.

[27] Robert E Tarjan, "Applications of path compression on balanced trees.," Tech. Rep., Stanford, CA, USA, 1975.

[28] Brandon Dixon, Monika Rauch, and Robert E. Tarjan, "Verification and sensitivity analysis of minimum spanning trees in linear time," *SIAM J. Comput.*, vol. 21, no. 6, pp. 1184–1192, 1992.

[29] V. Jarník, "O jistém problému minimálním," *Práce Moravské Přtrodovědecké Společnosti v Brně*, vol. 6, pp. 57–63, 1930.

[30] Michael L. Fredman and Robert Endre Tarjan, "Fibonacci heaps and their uses in improved network optimization algorithms," *J. ACM*, vol. 34, no. 3, pp. 596–615, 1987.

[31] Andrew Chi-Chih Yao, "An $O(|E| \log\log |V|)$ algorithm for finding minimum spanning trees," *Inf. Process. Lett.*, vol. 4, no. 1, pp. 21–23, 1975.

[32] David R. Cheriton and Robert Endre Tarjan, "Finding minimum spanning trees," *SIAM J. Comput.*, vol. 5, no. 4, pp. 724–742, 1976.

[33] H N Gabow, Z Galil, T Spencer, and R E Tarjan, "Efficient algorithms for finding minimum spanning trees in undirected and directed graphs," *Combinatorica*, vol. 6, no. 2, pp. 109–122, jan 1986.

[34] Bernard Chazelle, "A faster deterministic algorithm for minimum spanning trees," in *FOCS*, 1997, pp. 22–31.

[35] Seth Pettie, "Finding minimum spanning trees in $O(m\alpha(m, n))$ time," Tech. Rep., Austin, TX, USA, 1999.

[36] Michael L. Fredman and Dan E. Willard, "Trans-dichotomous algorithms for minimum spanning trees and shortest paths," *J. Comput. Syst. Sci.*, vol. 48, no. 3, pp. 533–551, 1994.

[37] Seth Pettie and Vijaya Ramachandran, "An optimal minimum spanning tree algorithm," *J. ACM*, vol. 49, no. 1, pp. 16–34, 2002.

[38] E. W. Dijkstra, "A note on two problems in connexion with graphs," *Numerische Mathematik*, vol. 1, pp. 269–271, 1959.

[39] Robert Endre Tarjan, "Applications of path compression on balanced trees," *J. ACM*, vol. 26, no. 4, pp. 690–715, 1979.

[40] Amos Korman and Shay Kutten, "Distributed verification of minimum spanning trees," in *Proceedings of the twenty-fifth annual ACM symposium on Principles of distributed computing*, New York, NY, USA, 2006, PODC '06, pp. 26–34, ACM.

[41] Valerie King, Chung Keung Poon, Vijaya Ramachandran, and Santanu Sinha, "An optimal EREW PRAM algorithm for minimum spanning tree verification," *Inf. Process. Lett.*, vol. 62, pp. 153–159, May 1997.

[42] Seth Pettie, "An inverse-Ackermann type lower bound for online minimum spanning tree verification," *Combinatorica*, vol. 26, no. 2, pp. 207–230, 2006.

[43] Adam L. Buchsbaum, Haim Kaplan, Anne Rogers, and Jeffery R. Westbrook, "Linear-time pointer-machine algorithms for least common ancestors, mst verification, and dominators," in *Proceedings of the thirtieth annual ACM symposium on Theory of computing*, New York, NY, USA, 1998, STOC '98, pp. 279–288, ACM.

[44] Adam L. Buchsbaum, Loukas Georgiadis, Haim Kaplan, Anne Rogers, Robert E. Tarjan, and Jeffery R. Westbrook, "Linear-time algorithms for dominators and other path-evaluation problems," *SIAM J. Comput.*, vol. 38, pp. 1533–1573, November 2008.

[45] Robert Endre Tarjan, "Sensitivity analysis of minimum spanning trees and shortest path trees," *Information Processing Letters*, vol. 14, no. 1, pp. 30–33, 1982.

[46] Heather Booth and Jeffery Westbrook, "A linear algorithm for analysis of minimum spanning and shortest path trees of planar graphs," *Algorithmica*, vol. 11, pp. 341–352, 1992.

[47] John Hopcroft and Robert Tarjan, "Algorithm 447: efficient algorithms for graph manipulation," *Commun. ACM*, vol. 16, pp. 372–378, June 1973.

[48] Andrew V. Goldberg, "An efficient implementation of a scaling minimum-cost flow algorithm," *Journal of Algorithms*, vol. 22, pp. 1–29, 1992.

[49] Cüneyt F. Bazlamaçci and Khalil S. Hindi, "Minimum-weight spanning tree algorithms a survey and empirical study," *Comput. Oper. Res.*, vol. 28, no. 8, pp. 767–785, 2001.

[50] C. Demetrescu, A.V. Goldberg, and D. Johnson, "9th DIMACS implementation challenge – Shortest Paths," 2005, http://www.dis.uniroma1.it/~challenge9/.

[51] B.V. Cherkassky, A.V. Goldberg, and T. Radzik, "Shortest paths algorithms: theory and experimental evaluation," *Mathematical Programming*, vol. 73, pp. 129–174, 1996.

[52] Walter Bright, "D Programming Language," 2001, http://www.dlang.org.

[53] Boris V. Cherkassky, Loukas Georgiadis, Andrew V. Goldberg, Robert Endre Tarjan, and Renato Fonseca F. Werneck, "Shortest path feasibility algorithms: An experimental evaluation," in *ALENEX*, 2008, pp. 118–132.

[54] Jack Edmonds, "Paths, trees and flowers," *Canadian Journal of Mathematics*, vol. 17, pp. 449–467, 1965.

[55] G. L. Nemhauser and L. A. Wolsey, *Integer and Combinatorial Optimization*, John Wiley & Sons, New York, 1999.

[56] Vladimir Kolmogorov, "Blossom V: a new implementation of a minimum cost perfect matching algorithm," *Mathematical Programming Computation*, vol. 1, pp. 43–67, 2009, 10.1007/s12532-009-0002-8.

[57] L. Lovász and M.D. Plummer, *Matching Theory*, North-Holland, Amsterdam, 1986.

[58] A. Frank, "A survey on t-joins, t-cuts, and conservative weightings," in *Combinatorics, Paul Erdös is Eighty*, (D. Miklós, V.T. Sós, and eds.) T. Szönyi, Eds., vol. 2, pp. 213–252. Bolyai Society, Budapest, 1996.

[59] Jack Edmonds, "Maximum matching and a polyhedron with $0, 1$ vertices," *J. of Res. the Nat. Bureau of Standards*, vol. 69 B, pp. 125–130, 1965.

[60] H. Gabow, *Implementation of algorithms for maximum matching on non-bipartite graphs*, Ph.D. thesis, Stanford University, 1974.

[61] E.L. Lawler, *Combinatorial Optimization: Networks and Matroids*, Holt, Rinehart, and Winston, New York, 1976.

[62] H. N. Gabow, "A scaling algorithm for weighted matching on general graphs," in *Proceedings* 26^{th} *Annual Symposium of the Foundations of Computer Science*. 1985, pp. 90–100, IEEE Computer Society.

[63] Z. Galil, S. Micali, and H.N. Gabow, "An $O(EV \log V)$ algorithm for finding a maximal weighted matching in general graphs," *SIAM Journal of Computing*, vol. 15, pp. 120–130, 1986.

[64] H.N. Gabow, Z. Galil, and T.H. Micali, "Efficient implementation of graph algorithms using contraction," *Journal of the ACM*, vol. 36, pp. 540–572, 1989.

[65] H. N. Gabow, "Data structures for weighted matching and nearest common ancestors with linking," in *Proceedings of the First Annual ACM-SIAM Symposium on Discrete Algorithms*. 1990, pp. 434–443, Association for Computing Machinery.

[66] J. Edmonds, E.L. Johnson, and S.C. Lockhart, "Blossom I: A computer code for the matching prolem," Unpublished report, 1969.

[67] W.R. Pulleyblank, *Faces of matching polyhedra*, Ph.D. thesis, University of Waterloo, Waterloo, Ontario, 1973.

[68] W.H. Cunningham and A.B. Marsh, "A primal algorithm for optimum matching," *Mathematical Programming Study*, vol. 8, pp. 50–72, 1978.

[69] T.F. Havel, *The combinatorial distance geometry approach to the calculation of molecular conformation*, Ph.D. thesis, University of California, Berkeley, 1982.

[70] M. Grötschel and O. Holland, "Solving matching problems with linear programming," *Mathematical Programming*, vol. 33, pp. 243–259, 1985.

[71] M. Trick, *Networks with additional structured constraints*, Ph.D. thesis, School of Industrial and Systems Engineering, Georgia Institute of Technology, Atlanta, Georgia, 1987.

[72] U. Derigs and A. Metz, "Solving (large scale) matching problems combinatorially," *Mathematical Programming*, vol. 50, pp. 113–122, 1991.

[73] D.L. Miller and J.F. Pekny, "A staged primal-dual algorithm for perfect b-matching with edge capacities," *ORSA Journal on Computing*, vol. 7, pp. 298–320, 1995.

[74] W. Cook and A. Rohe, "Computing minimum-weight perfect matchings," *INFORMS Journal on Computing*, vol. 11, pp. 138–148, 1999.

[75] K. Mehlhorn and G. Schäfer, "Implementation of $o(nm \log n)$ weighted matchings in general graphs: the powers of data structures," *Journal of Experimental Algorithms*, vol. 7, no. 4, 2002.

[76] D.B. Johnson, "Efficient algorithms for shortest paths in sparse networks," *J. ACM*, vol. 24, no. 1, pp. 1–13, 1977.

[77] H.N. Gabow and R.E. Tarjan, "Faster scaling algorithms for general graph matching problems," *Journal of the ACM*, vol. 38, pp. 815–853, 1991.

[78] Avi Shoshan and Uri Zwick, "All pairs shortest paths in undirected graphs with integer weights," in *FOCS*, 1999, pp. 605–615.

[79] D. Coppersmith and S. Winograd, "Matrix multiplication via arithmetic progressions," in *Proceedings of the nineteenth annual ACM symposium on Theory of computing*, New York, NY, USA, 1987, STOC '87, pp. 1–6, ACM.

[80] Vladimir Kolmogorov, "Implementation of the blossom V algorithm," `http://pub.ist.ac.at/~vnk/software.html`, 2009.

[81] A.V. Goldberg, "A simple shortest path algorithm with linear average time," in *9th Ann. European Symp. on Algorithms (ESA 2001)*, Aachen, Germany, 2001, vol. 2161 of *Lecture Notes in Computer Science*, pp. 230–241, Springer.

[82] Ravindra K. Ahuja, Kurt Mehlhorn, James B. Orlin, and Robert E. Tarjan, "Faster algorithms for the shortest path problem," *Journal of the ACM*, vol. 37, no. 2, pp. 213–223, Apr. 1990.

[83] S. Robinson, "Toward an optimal algorithm for matrix multiplication," *SIAM News*, vol. 38, no. 9, 2005.

[84] Virginia Vassilevska Williams, "Multiplying matrices faster than Coppersmith-Winograd," in *STOC*, 2012, pp. 887–898.

[85] V. Strassen, "Gaussian elimination is not optimal.," *Numerische Mathematik*, vol. 14, no. 3, pp. 354–356, 1969.

[86] Noga Alon, Zvi Galil, and Oded Margalit, "On the exponent of the all pairs shortest path problem," *J. Comput. Syst. Sci.*, vol. 54, no. 2, pp. 255–262, 1997.

[87] R. J. Lipton and R. E. Tarjan, "A separator theorem for planar graphs," *SIAM Journal App. Math.*, vol. 36, pp. 177–189, 1979.

[88] R. E. Bellman, *Dynamic Programming*, Princeton University Press, Princeton, NJ, 1957.

[89] Joseph JaJa, *Introduction to Parallel Algorithms*, Addison Wesley, 1^{st} edition, 1992.

[90] Wai-Kai Chen, *The VLSI Handbook, Second Edition*, CRC Press, Inc., Boca Raton, FL, USA, 2006.

[91] Justin R. Smith, *The design and analysis of parallel algorithms*, Oxford University Press, Inc., New York, NY, USA, 1993.

[92] B. F. Wang and G. H. Chen, "Constant time algorithms for the transitive closure and some related graph problems on processor arrays with reconfigurable bus systems," *IEEE Trans. Parallel Distrib. Syst.*, vol. 1, no. 4, pp. 500–507, oct 1990.

[93] G. H. Chen, B. F. Wang, and C. J. Lu, "On the parallel computation of the algebraic path problem," *IEEE Trans. Parallel Distrib. Syst.*, vol. 3, no. 2, pp. 251–256, mar 1992.

[94] G. H. Chen, Olariu S., Schwing J. L., Wang B. F., and Zhang J., "Constant-time tree algorithms on reconfigurable meshes of size $n \times n$," *Journal of Parallel and Distributed Computing*, vol. 26, no. 2, pp. 187–150, 1995.

[95] Eliezer Dekel, David Nassimi, and Sartaj Sahni, "Parallel matrix and graph algorithms," *SIAM J. Comput.*, vol. 10, no. 4, pp. 657–675, 1981.

[96] Ludek Kucera, "Parallel computation and conflicts in memory access," *Inf. Process. Lett.*, vol. 14, no. 2, pp. 93–96, 1982.

[97] L.G. Valiant, "Parallelism in comparison models," *SIAM J. Comput.*, vol. 4, no. 3, pp. 348–355, 1975.

[98] Victor Y. Pan and Franco P. Preparata, "Supereffective slow-down of parallel computations," in *Proceedings of the fourth annual ACM symposium on Parallel algorithms and architectures*, New York, NY, USA, 1992, SPAA '92, pp. 402–409, ACM.

[99] Victor Y. Pan and Franco P. Preparata, "Work-preserving speed-up of parallel matrix computations," *SIAM J. Comput.*, vol. 24, no. 3, pp. 811–821, jun 1995.

[100] Yijie Han, Victor Y. Pan, and John H. Reif, "Efficient parallel algorithms for computing all pair shortest paths in directed graphs," *Algorithmica*, vol. 17, no. 4, pp. 399–415, 1997.

[101] Rüdiger Reischuk, "Probabilistic parallel algorithms for sorting and selection," *SIAM J. Comput.*, vol. 14, no. 2, pp. 396–409, 1985.

[102] Michael J. Quinn, *Parallel Programming in C with MPI and OpenMP*, McGraw-Hill Education Group, 2003.

[103] Gene M. Amdahl, "Validity of the single processor approach to achieving large scale computing capabilities," in *Proceedings of the April 18-20, 1967, spring joint computer conference*, New York, NY, USA, 1967, AFIPS '67 (Spring), pp. 483–485, ACM.

[104] Richard J. Lipton, Donald J. Rose, and Robert E. Tarjan, "Generalized nested dissection," *SIAM J. Numer. Anal.*, vol. 16, no. 2, pp. 346–358, 1979.

[105] Greg N. Frederickson, "Fast algorithms for shortest paths in planar graphs, with applications," *SIAM J. Comput.*, vol. 16, no. 6, pp. 1004–1022, 1987.

[106] Monika R. Henzinger, Philip Klein, Satish Rao, and Sairam Subramanian, "Faster shortest-path algorithms for planar graphs," *J. Comput. Syst. Sci.*, vol. 55, pp. 3–23, August 1997.

[107] Andrew V. Goldberg, "Scaling algorithms for the shortest paths problem," *SIAM Journal on Computing*, vol. 24, no. 3, pp. 494–504, June 1995.

[108] Jittat Fakcharoenphol and Satish Rao, "Planar graphs, negative weight edges, shortest paths, and near linear time," *J. Comput. Syst. Sci.*, vol. 72, pp. 868–889, August 2006.

[109] Philip N. Klein, Shay Mozes, and Oren Weimann, "Shortest paths in directed planar graphs with negative lengths: A linear-space $O(n \log^2 n)$-time algorithm," *ACM Transactions on Algorithms*, vol. 6, no. 2, pp. 1–18, 2010.

[110] Gary L. Miller, "Finding small simple cycle separators for 2-connected planar graphs," in *Proceedings of the sixteenth annual ACM symposium on Theory of computing*, New York, NY, USA, 1984, STOC '84, pp. 376–382, ACM.

[111] Lan Guo, Supratik Mukhopadhyay, and Bojan Cukic, "Does your result checker really check?," in *Dependable Systems and Networks*, 2004, pp. 399–404.

[112] Dieter Kratsch, Ross M. McConnell, Kurt Mehlhorn, and Jeremy Spinrad, "Certifying algorithms for recognizing interval graphs and permutation graphs SIAM J," *Comput*, vol. 36, no. 2, pp. 326–353, 2006.