University of St Andrews

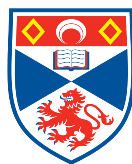
CS5199

Algorithms for Weighted Directed Graphs

Author:
Raiyan Chowdhury

Supervisor:
Dr Michael Young

9th May 2023



Abstract

A graph is a structure representing pairwise relations between objects. The objects may be represented visually using abstractions known as vertices (also known as nodes or points) and are related to other vertices via an edge. Directed graphs are graphs but with a directed edge between pairs of vertices which may have zero, one or many edges between the vertices with a direction in either direction. Edge-weighted digraphs (directed graphs) are graphs with a numerical value associated with each edge. These values may represent different kinds of relations between two vertices such as distance, time and cost. This gives rise to various problems including Minimum Spanning Trees, Shortest Path and Maximal Flow problems. There will also be some algorithms implemented for non-weighted graphs such as the Minimum Cut problem.

This dissertation focuses on the implementations of various graph algorithms on these problems using the GAP programming language. The algorithms are tested, compared and analysed within each group of algorithm as well as benchmarked against the Scipy library. There is also additional work to integrate this into the Digraphs package within GAP and some further work on visualising the digraphs paths and trees. The algorithms, as well as appropriate GAP constructors are integrated into the Digraphs package within GAP.

Declaration

I declare that the material submitted for assessment is my own work except where credit is explicitly given to others by citation or acknowledgement. This work was performed during the current academic year except where otherwise stated. The main text of this project report is 20866 words long, including project specification and plan. In submitting this project report to the University of St Andrews, I give permission for it to be made available for use in accordance with the regulations of the University Library. I also give permission for the title and abstract to be published and for copies of the report to be made and supplied at cost to any bona fide library or research worker, and to be made available on the World Wide Web. I retain the copyright in this work.

Acknowledgements

I would like to greatly thank Michael Young for his useful and helpful support throughout the dissertation. He went beyond what I expected from a supervisor and I thoroughly enjoyed working under his supervision. He was able to provide direction, and clear, understandable advice when I was stuck as well as extremely detailed feedback.

I would also like to thank Alice Buckley, Cliona Kennedy, Luca Bennett and Suyog Gurung for their extremely valuable support throughout the dissertation and throughout University. These people have proven to be invaluable and very important to me.

Finally, this goes without saying but I would like to thank my family for their continued support throughout University and my life.

Effect of Strikes

I was affected by the strikes by not being able to meet with my supervisor when I didn't know how to proceed. This was a recurring event and I often found myself unsure how to progress and having to wait for them to return, losing out on valuable time throughout the semester. The strikes also took place on a majority of Wednesdays throughout the semester which is when the weekly GAP meeting took place and was a valuable time to meet with my supervisor, as well as the other members of the GAP community in St Andrews where I was able to discuss ideas and ask for help but due to the strikes, this was also not possible for several weeks. I tried to work around this but due to the high frequency of strikes during the middle of semester when work was most crucial, was sometimes not possible.

Contents

1 Introduction **11**
1.1 Project Aims . . . 12
1.1.1 Primary . . . 12
1.1.2 Secondary . . . 12

2 Preliminaries **14**
2.1 Representation of Graphs . . . 18
2.2 Binary Heap . . . 19

3 Software Engineering Process **23**

4 Ethics **25**

5 Minimum Spanning Tree Algorithms **26**
5.1 Spanning Trees . . . 26
5.2 Minimum Spanning Trees . . . 26
5.3 Prim's Algorithm . . . 28
5.3.1 Implementation . . . 29
5.4 Kruskal's Algorithm . . . 31
5.4.1 Implementation . . . 31
5.5 Borůvka's Algorithm . . . 36
5.6 Analysis . . . 39
5.6.1 Comparison of Prim's and Kruskal's algorithm . . . 39
5.6.2 Comparison of all MST algorithms . . . 44
5.6.3 Comparison with Scipy's Implementation . . . 45

6 Shortest Path Algorithms **48**
6.1 Shortest Paths . . . 48
6.1.1 Single Source Shortest Path algorithms . . . 48
6.1.2 All Pair Shortest Path algorithms . . . 48
6.1.3 Path Construction . . . 49
6.2 Dijkstra's Algorithm . . . 50
6.2.1 Implementation . . . 52
6.3 Bellman–Ford Algorithm . . . 54
6.3.1 Implementation . . . 55
6.4 Floyd–Warshall Algorithm . . . 57
6.4.1 Implementation . . . 57
6.5 Johnson's Algorithm . . . 58
6.5.1 Implementation . . . 59

6.6 Analysis . . . 60
6.6.1 Single Source Shortest Path Algorithm Comparison . . . 60
6.6.2 All Pairs Shortest Path Algorithm Comparison . . . 61
6.6.3 Comparison with Scipy's Implementation . . . 64

7 Maximal Flow Algorithms 71
7.1 Ford–Fulkerson Method . . . 72
7.2 Edmonds–Karp Algorithm . . . 74
7.2.1 Implementation . . . 74
7.3 Dinic's Algorithm . . . 76
7.4 Push–Relabel Algorithm . . . 78
7.5 Max-Flow Min-Cut Theorem . . . 83
7.5.1 Karger's Algorithm . . . 83
7.5.2 Karger–Stein Algorithm . . . 85
7.6 Analysis . . . 86
7.6.1 Maximal Flow Algorithms . . . 86
7.6.2 Minimum Cut Algorithms . . . 86

8 Working with GAP 95
8.1 GAP Objects . . . 95
8.1.1 Attributes . . . 96
8.1.2 Properties . . . 96
8.1.3 Operations . . . 96

9 Visualisation 99
9.1 Highlighting MSTs . . . 100
9.2 Highlighting Shortest Paths . . . 101
9.3 Implementation . . . 104

10 Testing 105
10.1 Testing . . . 105
10.2 Automated Analysis . . . 106
10.2.1 Pandas and Seaborn . . . 107

11 Evaluation and Appraisal 108
11.1 Primary . . . 108
11.2 Secondary . . . 108

12 Conclusion 111

Index 113

Appendices 117

A Ethics Form 118

B User Manual 120

C Minimum Spanning Tree Algorithm Plots 126

D Shortest Path Algorithm Plots **128**

E Maximal Flow Algorithm Plots **130**

List of Figures

1.1 Example of Bidirectional edges . . . 11

2.1 Example of a Vertex . . . 14
2.2 Example of an Edge . . . 14
2.3 Example Undirected Graph G . . . 15
2.4 Example Directed Graph D . . . 15
2.5 Example Cycle in Directed Graph (G) . . . 15
2.6 Example of Edge Weight . . . 16
2.7 Example of Outneighbours . . . 16
2.8 Example of In-degree . . . 17
2.9 Example of Connected and Disconnected Graph . . . 17
2.10 Example of Strongly Connected Graph . . . 17
2.11 Example of Strongly Connected Components . . . 18
2.12 Example of Symmetric Digraph . . . 18
2.13 Examples of different graph representations . . . 19
2.14 Pushing to Binary Heap Part 1 . . . 21
2.15 Pushing to Binary Heap Part 2 . . . 21
2.16 Pushing to Binary Heap Part 3 . . . 21
2.17 Popping from Binary Heap . . . 22

5.1 Examples of spanning tree subgraphs on the complete graph with 4 vertices . . . 26
5.2 Example of a Disconnected Graph . . . 27
5.3 Examples of Minimum Spanning Trees . . . 27
5.4 Example of Prim's algorithm . . . 29
5.5 Example of Parallel Edges . . . 30
5.6 Example of Kruskal's algorithm . . . 32
5.7 Example of unoptimised DSU Algorithm . . . 33
5.8 Example of DSU Path Compression . . . 34
5.9 Example of DSU in Kruskal's algorithm . . . 36
5.10 Kruskal's algorithm with and without Path Compression . . . 37
5.11 Example of Borůvka's Algorithm . . . 39
5.12 Overall comparison between Prim's and Kruskal's Algorithm . . . 40
5.13 Overall comparison between Prim's and Kruskal's Algorithm with Path Compression . . . 40
5.14 Overall comparison between Prim's and Kruskal's Algorithms (with and without path compression) . . . 41
5.15 Prim's vs Kruskal's Algorithm with 1.0 edge probability . . . 42
5.16 Prim's vs Kruskal's Algorithm with 0.01 edge probability . . . 42

5.17 Case where Prim's algorithm fails with digraph 43
5.18 Prim's Algorithm with and without hash map size specified 44
5.19 Prim's Algorithm Profiled 44
5.20 Prim's vs Kruskal's vs Borůvka's algorithm 45
5.21 Prim's vs Kruskal's vs Borůvka's algorithm for fixed 1000 vertices with varying number of edges 46
5.22 Scipy vs Kruskal's vs Prim's for 1.0 edge probability 46

6.1 Example of Single Source Shortest Path algorithm 48
6.2 APSP Example Output 49
6.3 Example of Dijkstra's algorithm 51
6.4 Incorrect Dijkstra's Algorithm on negative Edge Weighted Digraph . . 52
6.5 Example of Dijkstra's GAP output 52
6.6 Example of Dijkstra's GAP output with no path 53
6.7 Shortest Path has $|V| - 1$ edges 55
6.8 Bellman–Ford with and without optimisation on 1.0 edge probability graph 56
6.9 Adding source s to Graph for Johnson's Algorithm Example 59
6.10 Transforming edge weights to become non negative 60
6.11 Overall comparison of Bellman–Ford vs Dijkstra's Algorithm 62
6.12 Overall comparison of Bellman–Ford (optimised) vs Dijkstra's Algorithm 62
6.13 Overall comparison of Bellman–Ford (unoptimised and optimised) vs Dijkstra's Algorithm 65
6.14 Floyd–Warshall algorithm on various graph densities 66
6.15 Overall comparison of Johnson's vs Floyd–Warshall algorithm 67
6.16 Johnson's vs Floyd–Warshall algorithm on Complete Graph (1.0 edge probability) 67
6.17 Johnson's vs Floyd–Warshall algorithm on sparse graph (0.01 edge probability) 68
6.18 Johnson's vs Floyd–Warshall algorithm threshold probability (0.125 edge probability) 68
6.19 Dijkstra's Algorithm (GAP vs Scipy) 69
6.20 Bellman Ford Algorithm (GAP vs Scipy) 69
6.21 Floyd–Warshall (GAP vs Scipy) 70
6.22 Johnson's Algorithm (GAP vs Scipy) 70

7.1 Example of Ford–Fulkerson Method 73
7.2 Example of output for Maximum Flow Algorithms 75
7.3 Example Graph when Ford–Fulkerson is slow 75
7.4 Undirected Level Graph 76
7.5 Level graph with only relevant edges permitted 76
7.6 Example of Dinic's Algorithm 77
7.7 Example of Preflow 79
7.8 Example of Push–Relabel Algorithm Part 1 81
7.8 Example of Push–Relabel Algorithm Part 2 82
7.9 Example of Minimum Cut 83
7.10 Example of Edge Contraction in Karger's Algorithm 88
7.11 Example of Edge Contraction in Karger's Algorithm with directed graph 88
7.12 Overall comparison of Edmonds–Karp vs Dinic's Algorithm 88

7.13 Comparison of Edmonds–Karp vs Dinic's Algorithm with 0.01 edge probability . . . 89
7.14 Comparison of Edmonds–Karp vs Dinic's Algorithm with 0.125 edge probability . . . 89
7.15 Comparison of Edmonds–Karp vs Dinic's Algorithm with 0.25 edge probability . . . 90
7.16 Comparison of Edmonds–Karp vs Dinic's Algorithm with 0.5 edge probability . . . 90
7.17 Comparison of Edmonds–Karp vs Dinic's Algorithm with 1 edge probability . . . 91
7.18 Overall comparison of Edmonds–Karp, Dinic's and Push–Relabel Algorithm . . . 91
7.19 Comparison of Dinic's vs Push–Relabel Algorithm with 1 edge probability 92
7.20 Comparison of Dinic's vs Push–Relabel Algorithm with 0.5 edge probability . . . 92
7.21 Comparison of Dinic's vs Push–Relabel Algorithm with 0.25 edge probability . . . 93
7.22 Comparison of Dinic's Algorithm vs Push–Relabel with 0.125 edge probability . . . 93
7.23 Comparison of Dinic's vs Push–Relabel Algorithm with 0.01 edge probability . . . 94
7.24 Overall comparison of Karger's vs Karger–Stein's Algorithm . . . 94

8.1 Example of Method Installation in GAP . . . 97
8.2 Example of Method Install with local functions . . . 98

9.1 Example of DotDigraph . . . 99
9.2 Example Visualisation for MST . . . 100
9.3 Example Visualisation for Shortest Paths . . . 101
9.4 Creating Shortest Path subgraph from Algorithm output . . . 102
9.5 Examples of changing visualisation options . . . 103
9.6 Example of all visualisation options on one graph . . . 103
9.7 Example of highlighting a single path . . . 103
9.8 Examples of changing visualisation options . . . 104

C.1 Prim's Algorithm on various graph densities . . . 126
C.2 Borůvka's Algorithm on various graph densities . . . 127
C.3 Kruskal's Algorithm (with path compression) on various graph densities 127

D.1 Dijkstra's Algorithm on various graph densities . . . 128
D.2 Bellman–Ford Algorithm (optimised) on various graph densities . . . 129
D.3 Johnson's Algorithm on various graph densities . . . 129

E.1 Edmonds–Karp Algorithm on various graph densities . . . 130
E.2 Dinic's Algorithm on various graph densities . . . 131

List of Algorithms

1 Prim's Algorithm [11] . . . 28
2 Kruskal's Algorithm [11] . . . 31
3 Find and Union with no optimisations . . . 33
4 Find with Path Compression . . . 34
5 Union with Ranking . . . 35
6 Borůvka's Algorithm . . . 38
7 Dijkstra's Algorithm . . . 50
8 Bellman–Ford Algorithm . . . 54
9 Floyd–Warshall Algorithm . . . 57
10 Johnson's Algorithm . . . 58
11 Simple-Ford–Fulkerson Method [21] . . . 72
12 Ford–Fulkerson Method . . . 73
13 Breadth–First–Search . . . 74
14 Push–Relabel Algorithm [38] . . . 80
15 Karger's Algorithm [16] . . . 84
16 Karger–Stein Algorithm [16] . . . 85

Chapter 1

Introduction

A graph is a structure consisting of sets of objects in which the objects may be related. The objects may be represented visually using abstractions known as vertices, also known as nodes or points (see Figure 2.1), and are related to other vertices via an edge (see Figure 2.2). Directed graphs are graphs but with a direction from one vertex to another and possibly vice versa. For example, if two people meet at a party and shake hands then this can be represented as an undirected graph as A shakes B's hand, B also shakes A's hand. This could also be represented with two bidirectional directed edges, one from A to B and vice versa (see Figure 1.1). Contrary, if A owes money to B, then this may be depicted as a directed graph where an edge from A is directed towards B as the owing of money is not true in the reverse direction. Edge-weighted digraphs (directed graphs) are graphs with a numerical value associated with each edge. These values may represent different kinds of relations between two vertices such as distance, time, cost, as well. This allows different types of problems to be solved including **Minimum Spanning Trees**, **Shortest Path** and **Maximal Flow** problems. There are various algorithms for each of these problems, each with their own properties, strengths and weaknesses fit for different use-cases. I will implement algorithms for these categories, then test, benchmark and analyse the performance and properties of the algorithms to determine which one is preferred. Once analysed, I with then integrate the algorithms in to the Digraphs package and document the functions for the GAP community to use.

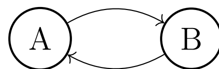


Figure 1.1: Example of Bidirectional edges

GAP [22] is a system for computational discrete algebra, and Digraphs [6] is a package within GAP that contains various functions for directed graphs. It was started at Lehrstuhl D für Mathematik, RWTH Aachen in 1986. After 1987 the development of GAP was coordinated in St Andrews. In the current Digraphs package as of 9th May 2023 (Version 1.6.2), some of these algorithms exist already but there is no notion of edge weights. Therefore, a shortest path algorithm, such as Dijkstra's algorithm, that is already implemented, is equivalent to a Breadth First Search as it finds a path with the least number of edges between a source and destination vertex. Thus, adding edge weights adds the opportunity to open the Digraphs package to

a whole new set of algorithms that may be implemented and take advantage of this edge weight property to model real world and theoretical problems. I can then analyse and benchmark these algorithms to find when they are best utilised and use the most optimal algorithm using properties of the graphs to select an algorithm dynamically.

Throughout this report in a digital version, there are many 'clickable' components such as the 'RETURN TO CONTENTS' at the centre of the footer, the last page on the right of the footer, as well as all references to figures and tables and also references to different sections in the report.

1.1 Project Aims

The main overarching goals of the project are to implement the following categories of algorithms and compare them. The various goals of the project are as follows:

1.1.1 Primary

1. Implement in GAP and compare
   (a) Three Shortest Path Algorithms.
   (b) Two Maximal Flow Algorithms.
   (c) Two Minimum Spanning Tree (MST) Algorithms.
2. Implement a new object type for edge-weighted digraphs in GAP, compatible with the Digraphs package, and include appropriate constructors.

1.1.2 Secondary

1. Implement a further algorithm for each group and compare to the other relevant algorithms (implemented in the primary objectives).
2. Write GAP code to allow for the shortest path and minimum spanning trees to be highlighted during visualisation of graphs.
3. Implement and compare (similarly to primary objectives) a new group of algorithms related to Maximal Flow, called Minimum Cut.
4. Implement a working algorithm for The Travelling Salesman problem.
5. Integrate a selection of the implemented algorithms into the Digraphs package for GAP including appropriate documentation and tests in the standard style used by the GAP community.
6. Explore an industry standard implementation of these algorithms (in Java, Python or C) and compare the performance to that of GAP.

Item 6. in the secondary goals previously involved writing my own algorithms in one of the languages listed but I decided that it might be better to compare it to an existing library as these already exist, are being used and are heavily optimised so will lend to a more interesting comparison.

The evaluation of how these both the primary and the majority of the secondary goals are achieved can be found in Section 11.

Chapter 2

Preliminaries

This section provides an overview of some graph theory and definitions that will be used throughout the document. This will give a general understanding for the terminology used that can can be applied to all the graph problems and more specific definitions will be given as they are introduced in the relevant sections.

Definition 2.1 (Vertex)**.** A **vertex** is a point. This is also called a **node**. See Figure 2.1.

Figure 2.1: Example of a Vertex

Definition 2.2 (Edge)**.** An **edge** is a line that connects vertices together. See Figure 2.2.

Figure 2.2: Example of an Edge

Definition 2.3 (Graph)**.** A **graph** is a pair (V, E), where V is the set of vertices and E the set of edges. We write $V(G)$ for the vertices of G and $E(G)$ for the edges of G. [12, Chapter 5.1]. This is also called an **undirected graph**. See Figure 2.3.

Definition 2.4 (Walk)**.** A **walk** in a graph is a sequence $v_1, e_1, v_2, e_2, ..., v_k-1, e_k-1, v_k$ where k is an integer and the sequence alternates between V and E. If $v_1 = v_k$, it is a closed walk or a circuit. [12, Chapter 5.11].

Definition 2.5 (Path)**.** A **path** in a graph is a walk in which all vertices are distinct. If there is a walk from u to v then there is a path from u to v. [12, Chapter 5.11].

Definition 2.6 (Ordered Pair)**.** An ordered pair (u, v) is a pair of objects in which the order of objects is significant. The ordered pair (u, v) is different from the unordered pair (v, u) unless $u = v$.

$$Let\ V(G) = \{A, B, C, D\},\ Let\ E(G) = \{(A, B), (A, C), (A, D), (B, D)\}$$

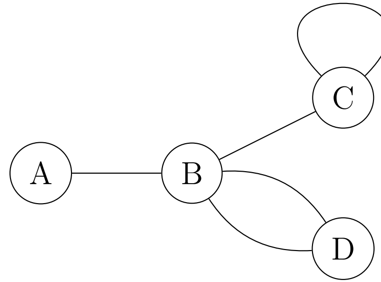


Figure 2.3: Example Undirected Graph G

$$Let\ V(D) = \{A, B, C, D\},\ Let\ E(D) = \{(A, B), (B, C), (B, D), (C, C), (D, B)\}$$

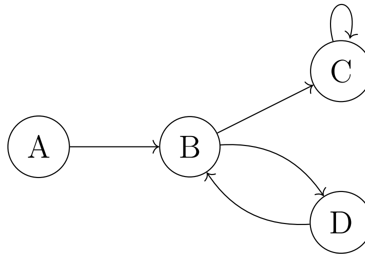


Figure 2.4: Example Directed Graph D

Definition 2.7 (Digraph)**.** A **directed graph** is an ordered pair of vertices and edges. The edge (u, v) signifies the walk is only possible in the direction $u \to v$ and not in the direction $v \to u$. See Figure 2.4.

In a graph, the edge (u, v) represents both the direction (u, v) and (v, u) as the graph is undirected. In a directed graph, this is not the case and every edge needs to be specified. Therefore, the bidirectional edge between vertices B and D is represented twice, one for each direction.

Definition 2.8 (Cycle)**.** A **cycle** is a graph C_n on vertices $v_1, v_2, ..., v_n$ with edges $(v_i, v_{1+(i\ mod\ n)})$ for $1 \leq i \leq n$, and no other edges; this is a path in which the first and last vertices have been joined by an edge. Generally, a cycle has at least three vertices. [12, Chapter 4.4]. See Figure 2.5. A graph without cycles is known as **acyclic**.

$$Let\ V(G) = \{A, B, C\},\ Let\ E(G) = \{(A, B), (B, C), (C, A)\}$$

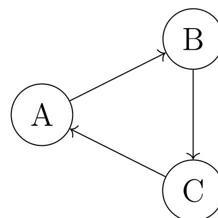


Figure 2.5: Example Cycle in Directed Graph (G)

Definition 2.9 (Loop)**.** A **loop** is a cycle with one vertex where the single vertex serves as both end points. [12, Chapter 4.4]. Vertex C in Figure 2.4 serves an example of a loop.

Definition 2.10 (Edge Weight)**.** An **edge weight** is a cost or value associated with an edge in a graph. Such a graph is called a **weighted graph**. This edge weight may be an integer, float or a rational and in Figure 2.6, the edge weight of (A, B) is 5. This can be written as $w(A, B) = 5$, where w represents the edge weight of the edge between vertices A and B if and only if there is only a single edge from A to B.

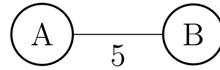


Figure 2.6: Example of Edge Weight

Definition 2.11 (Neighbour)**.** A **neighbour** is a vertex adjacent to another vertex (if an edge exists between them).

Definition 2.12 (Out Neighbour)**.** In a directed graph D, all the neighbours outbound from a vertex are considered its **out neighbours**. See Figure 2.7.

$$outneighbours(A) = \{B, C, D\}$$

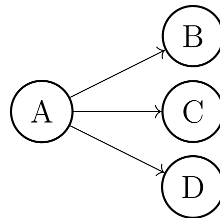


Figure 2.7: Example of Outneighbours

Definition 2.13 (In Neighbour)**.** In a directed graph D, a vertex is an **in neighbour** if it is the out neighbour of another vertex.

Definition 2.14 (Degree)**.** The **degree** of a vertex v is the number of edges incident with v.

Definition 2.15 (In-degree)**.** The **in degree** of a vertex is the number of inbound edges towards the vertex.

Definition 2.16 (Out-degree)**.** The **out degree** of a vertex is the number of outbound edges from the vertex.

Definition 2.17 (Connected graph)**.** Let $G = (V, E)$ be a graph. If for any two distinct elements u and v of V there is a path P from u to v then G is a **connected** graph. If $|V| = 1$, then G is connected. Otherwise, it is **disconnected**. See Figure 2.9.

$$indegree(A) = 3$$

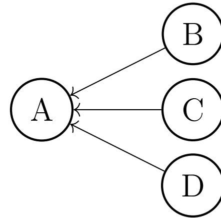


Figure 2.8: Example of In-degree

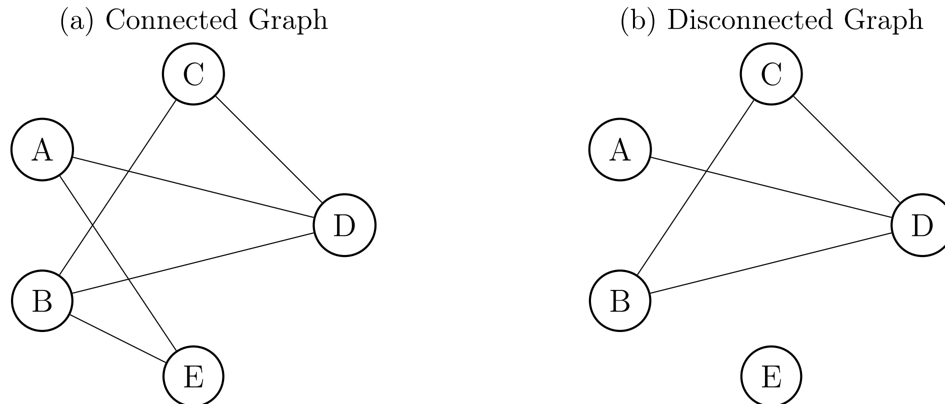


Figure 2.9: Example of Connected and Disconnected Graph

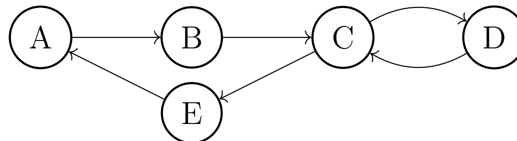


Figure 2.10: Example of Strongly Connected Graph

Definition 2.18 (Strongly Connected)**.** A **strongly connected** graph is a directed graph that has a path from each vertex to every other vertex. A graph may be connected but not strongly connected.

Definition 2.19 (Subgraph)**.** A subgraph S is a graph contained within a graph G where the $(V(S) \cap E(S)) \subseteq G)$.

Definition 2.20 (Strongly Connected Component)**.** A **strongly connected component** (scc) of a directed graph G is a subgraph that is themselves strongly connected and maximal. By maximal, this means there can be no additional edges or vertices from G being included in the subgraph without being strongly connected. See Figure 2.11.

Definition 2.21 (Symmetric)**.** A **symmetric** digraph is a graph where every pair of vertices u and v, if there exists a directed edge from u to v with weight $w(u, v)$, then there must also exist a directed edge from v to u with the same edge weight $w(v, u)$. See Figure 2.12.

$scc_1 = \{A, B, E\}, \ scc_2 = \{C, D, H\}, \ scc_3 = \{F, G\}$

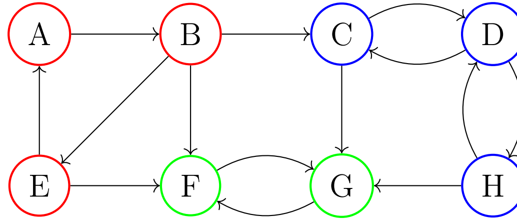


Figure 2.11: Example of Strongly Connected Components

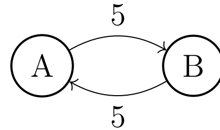


Figure 2.12: Example of Symmetric Digraph

Definition 2.22 (Simple)**.** A **simple** graph, also called a strict graph, is an unweighted, undirected graph containing no loops or parallel edges. A simple graph may be either connected or disconnected. [24].

Definition 2.23 (Simple path)**.** A path that repeats no vertex, except that the first and last in the path may be the same vertex.

Definition 2.24 (Complete graph)**.** A complete graph is a simple undirected graph in which every pair of distinct vertices is connected by a unique edge [3].

2.1 Representation of Graphs

There are a few ways of representing graphs, most notably adjacency matrices and adjacency lists.

Definition 2.25 (Adjacency Matrix)**.** [7, Chapter 2] An **adjacency matrix** of a directed graph τ is the $n \times n$ matrix $A = A(\tau)$ whose entries a_{ij} where v_i and v_j represent two vertices in a_{ij} are given below by

$$a_{ij} = \begin{cases} 1, & \text{if } v_i \text{ and } v_j \text{ are adjacent;} \\ 0, & \text{otherwise.} \end{cases}$$

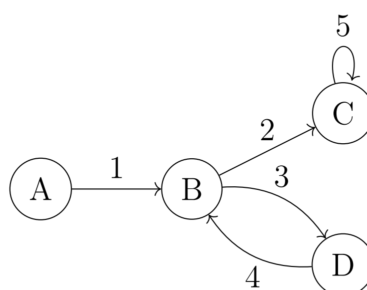
However, as our graphs have edge weights, the entry at a_{ij} represent the edge weight.

Figure 2.13b shows the adjacency matrix representation of the graph in Figure 2.13a.

Definition 2.26 (Adjacency List)**.** An **adjacency list** of τ is a list of the out neighbours associated with each vertex.

The advantages and disadvantages of these representations are shown by Table 2.1.

(a) Example Directed Weighted Graph



(b) Adjacency Matrix Representation

$$\begin{array}{c} \\ A \\ B \\ C \\ D \end{array} \begin{array}{c} \begin{array}{cccc} A & B & C & D \end{array} \\ \begin{pmatrix} 0 & 1 & 0 & 0 \\ 0 & 0 & 2 & 3 \\ 0 & 0 & 5 & 0 \\ 0 & 4 & 0 & 0 \end{pmatrix} \end{array}$$

Figure 2.13: Examples of different graph representations

	Advantages	Disadvantages
Adj. Matrix	• $O(1)$ look up for a specific edge between any two vertices (u, v)	• Uses $O(\|V\|)$ memory. • Inefficient for iterating out neighbours of a vertex u as it goes through vertices that are not out neighbours.
Adj. List	• More memory efficient if graph is sparse. • Faster at iterating out neighbours for a vertex u as we can access the list of neighbours directly.	• Slower at finding a specific edge between any two vertices u and v as it has to iterate the list of edges.

Table 2.1: Advantages and Disadvantages of different graph representations

2.2 Binary Heap

Here I describe a binary heap, how it works and the utility it provides.

Definition 2.27 (complete binary tree)**.** A **complete binary tree** is where every level of the tree at least twice the number of nodes as the level above, except the last one but all the nodes have to be as far leftmost as possible. The tree may have between 1 and 2^h nodes at the last level h where h is the height of the tree.

We need to obtain the least weighted edge each iteration and scanning an adjacency matrix each time is inefficient. We can use a binary heap (also called min/max heap in Python or priority queue in Java). The binary heap takes the form of a tree where the root of the tree contains the lowest or highest value in the tree depending on what type of heap, or comparator is used. Each node is smaller than or equal to its children if a min-heap (and vice versa for a max-heap)- this is known as the heap

property. A binary heap is a complete binary that respects the heap property.

In our use case, a min heap where the tree contains all the edges and is compared using the edge weight. The root of this min heap, is the lowest weighted edge in our graph. Using this data structure we can pop the root node from the tree with $O(\log n)$ complexity [23] instead of $O(n)$ with a list which is because all the elements after the popped element need to be shifted which has an upper bound of $O(n)$ complexity where n is the number of elements in the list. We will only focus on the min heap as this is the most relevant in this project but the max heap is the same, except for the root contains the maximum value and the heap property is reversed.

The main functions to describe a binary heap in GAP are explained below

Binary Heap Methods

- **Insert (push)** - push an element onto a heap.
- **Get min (peek)** - remove the root of the heap and return it.
- **Extract min (pop)** - return the value of the root of the heap.

Within a Binary Heap, when a node is to be re positioned within the tree, we call this ‘sifting up’ and ‘sifting down’ depending on which direction the node will move within the tree.

Sift up and Sift down

- **sift up**

  While the node is smaller than its parent, we swap the node with its parent. Repeat this until the node is no longer smaller.

- **sift down**

  For this operation, while the node has smaller children, we swap it with the smallest child (in case where multiple children have a smaller value). We repeat this until the heap property is once again guaranteed.

The time complexity of sift up and down is $O(\log n)$ as each iteration in the sift, the node moves one level in the tree so the complexity dependent on the height of the tree. As the tree is a complete binary tree which is always balanced, this is $O(\log n)$.

There are two other functions, update and build, but they are not utilised in the implementation of this algorithm so they will not be described in this paper. Let us go through some examples.

Now, what if we push an element that is not the smallest but has a value between 5 and 7.

In Figure 2.15, the 6 is added to the left most leaf position and then sift up is performed on this node. In the example, the value 6 is added as the leftmost child of

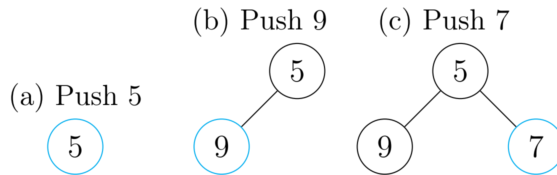


Figure 2.14: Pushing to Binary Heap Part 1

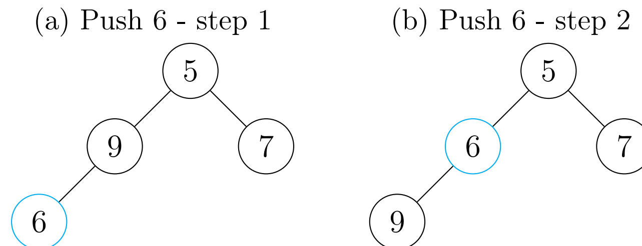


Figure 2.15: Pushing to Binary Heap Part 2

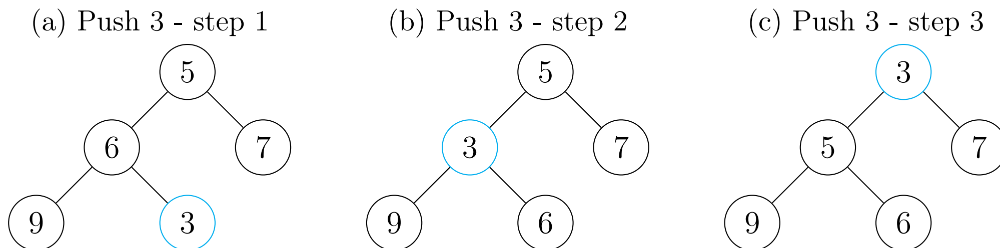


Figure 2.16: Pushing to Binary Heap Part 3

9 and then sifted up.

Like before, the 3 is added to the leftmost leaf position and sifted up. As it is the smallest value in the heap, it is now the root of the heap. Now, we will explore how the root is replaced when a node is popped.

When popping, we swap the root with the last element added, in the example, the root with value 3 and last added element 6 are swapped. We can then remove the previous root node. The new root violates the heap property and therefore is sifted down to a valid position.

Binary Heap Methods Complexities

- Insert (push) - $O(\log n)$

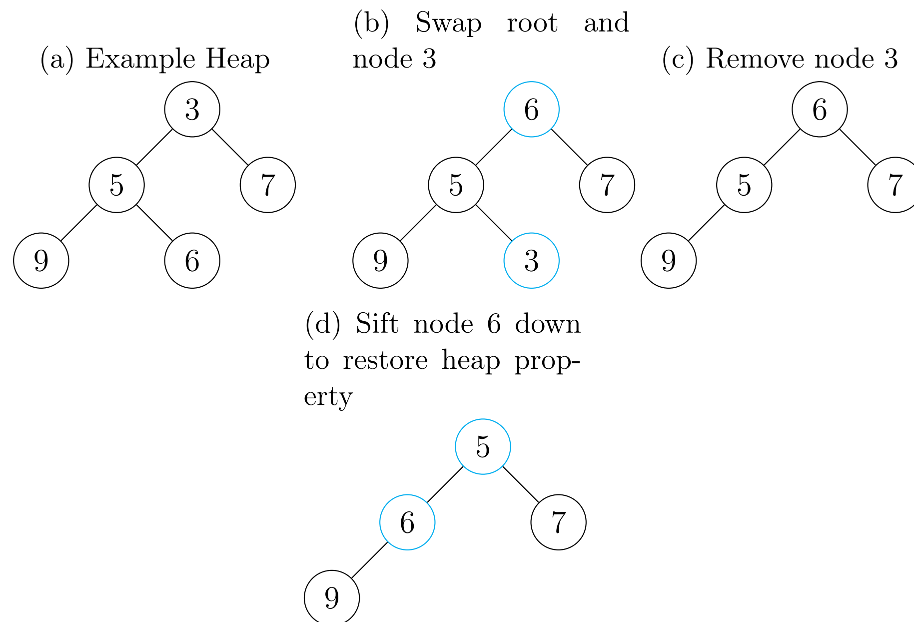
  This is because we insert the node at the bottom of the tree and then perform a sift up operation.

- Get min (peek) - $O(1)$

  This simply returns the root of the node so is a constant operation.

- Extract min (pop) - $O(\log n))$

  This is because when the root is swapped with the last node, the new root node may not adhere to the heap property and so must be sifted down.



(a) Example Heap

(b) Swap root and node 3

(c) Remove node 3

(d) Sift node 6 down to restore heap property

Figure 2.17: Popping from Binary Heap

To read more about binary heaps, refer to [11, Chapter 6.1].

Chapter 3

Software Engineering Process

This chapter focuses on how the code was developed and integrated into the Digraphs repository on Github [25].

To be able to add appropriate constructors to the Digraphs package, I forked the repository locally and started adding changes to that. As this clone was nested in my directory which was also version controlled, this made it a submodule - effectively a nested git repository which I hadn't encountered before. Any changes made outside inside this submodule was separate to the outer repository and pushes were not added. The same was true for the outer directory as any changes there could not be pushed into this submodule.

As for the development, I worked individually with the code written being version controlled (see [8]). I worked methodically through each group of algorithms, testing each part of code as I wrote them until the complete algorithm was written. I then tested the algorithms on small examples that tested the code for normal function cases, edges cases, special inputs (see Section 10) before creating larger examples to test the algorithms on. Once I had written at least two algorithms for each group, I was then able to test them with the output from each other and automate this to be able to run on any size example. If any errors were encountered I would try to debug the code using small examples or if there were performance issues I could profile the code using the profiling package [42] in GAP and check the number of executions as well as the timing for each process to help narrow down the source of the problem. This was useful a few times during the development of the algorithms as I was able to realise bottlenecks and unnecessary work such as looping parallel edges instead of keeping the minimum edge for $E(u, v)$ in a minimum spanning tree.

I followed an agile-like methodology by which, each week I set my self goals to complete which I cleared with my supervisor and worked on them until our next meeting. This was agile-like as I didn't formally keep track of tasks but kept and informal list of tasks. As this was an individual project, I was acting effectively acting as the scrum master each week and any tasks that weren't completed where either put in the backlog or added to next weeks set of tasks. This was repeated each week until the code, testing and analysis was completed.

When the code was ready to be merged into the Digraphs library, I made a pull

request which once reviewed will be merged into the Digraphs library. There are pipeline tests which are already passed. The pipeline tests for factors that could affect the code such as code testing coverage, tests passing and linting to ensure the code is of the same format as the rest of the library.

Chapter 4

Ethics

There were no ethical considerations for this project. All questions in the initial self assessment questionnaire were answered with 'No'. See Appendix A.

Chapter 5

Minimum Spanning Tree Algorithms

This chapter focuses on three minimum spanning tree algorithms; Prim's, Kruskal's and Borůvka's algorithm. First we introduce the theory for spanning and minimum spanning trees before delving into the design, implementation and analysis of these algorithms.

5.1 Spanning Trees

A spanning tree is an acyclic graph that connects all the vertices of a (undirected) graph together, forming a tree with the least number of edges possible. A graph must be connected, otherwise, it cannot contain a spanning tree. This is because, in a disconnected graph, there may be two vertices in which no path in the graph has these vertices as endpoints and therefore it is impossible to create a spanning tree that connects these two vertices. Figure 5.1 shows three similar possible spanning trees on the same graph, shown by the coloured paths. The spanning tree is a subgraph of the graph.

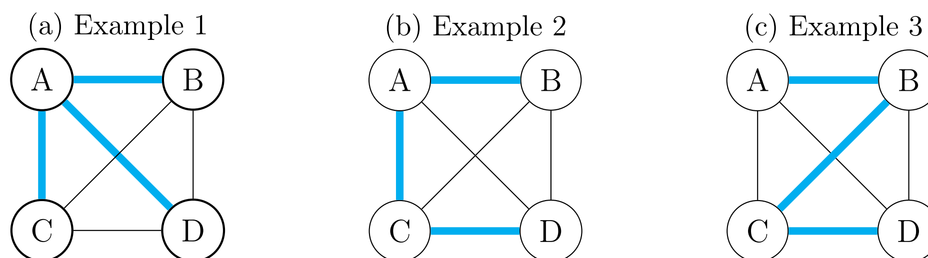


Figure 5.1: Examples of spanning tree subgraphs on the complete graph with 4 vertices

5.2 Minimum Spanning Trees

A Minimum Spanning Tree (MST) is a spanning tree in which all the edges have a weight. The problem shifts from creating a spanning tree with the least number of edges to, instead, the minimum sum of all the edge weights. This is more formally shown by Equation 5.1 [11, Chapter 23], where we 'wish to find an acyclic subset $T \subseteq E$ (where E is the set of possible edges) that connects all of the vertices and whose total weight w is minimized'.

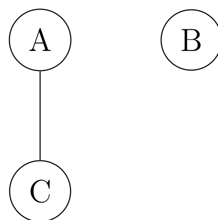


Figure 5.2: Example of a Disconnected Graph

$$w(T) = \sum_{(u,v) \in T} w(u, v) \tag{5.1}$$

There are some properties of graphs that must hold if it is possible to find a minimum spanning tree.

MST Graph Properties [45, Chapter 4.3]

- **Connected graph**

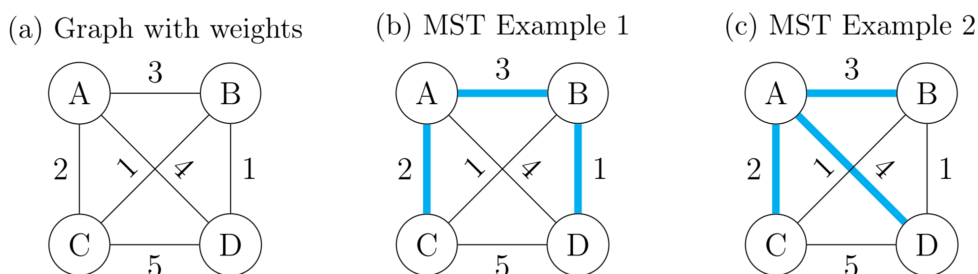
  The graph needs to be connected, otherwise it is impossible to find the minimum spanning tree of the graph (as demonstrated by Figure 5.2). There is no path from either A or C to B, so a spanning tree and therefore a minimum spanning tree is impossible to obtain.

- **Edge Weight Polarity**

  The edge weights may be positive or negative and a minimum spanning tree can still be obtained.

- **Edge Weight Uniqueness**

  If all the edges have unique weights, then there will be only one, unique minimum spanning tree. Figure 5.3 demonstrates that with multiple edges with the same edge weights, there may be multiple minimum spanning trees.



$w(T) = 2 + 3 + 1 = 6$ $\quad$ $w(T) = 2 + 3 + 1 = 6$

Figure 5.3: Examples of Minimum Spanning Trees

5.3 Prim's Algorithm

While this algorithm is known as 'Prim's Algorithm', and is what it will be referred to as in this paper, it was originally developed by Vojtěch Jarník [2]. Robert Prim later published this algorithm in his 1957 paper [41]. This algorithm is sometimes referred to as Jarník's Algorithm, Jarník-Prim Algorithm or even Prim-Dijkstra Algorithm as Edsger Dijkstra also published this algorithm in 1959, 2 years after Prim.

Prim's algorithm [11] is a greedy algorithm which means the algorithm always attempts to select the best edge - in this case the lowest weighted edge outwards from some initially arbitrarily selected vertex u. The endpoint of the edge is vertex v, from which we then find the next lowest edge outwards from this vertex. We only consider edges that when added to our growing minimum spanning tree doesn't create a cycle. Only the minimum edge is considered between u and v in the case of parallel edges. We repeat this process until all the vertices have been visited, meaning they are explored at least once. The pseudo-code for Prim's algorithm is shown more formally by Algorithm 1.

Algorithm 1 Prim's Algorithm [11]

```
1:  procedure PRIM(G)
2:      p ← ∅
3:      root ← some arbitrary vertex ∈ V(G)
4:      for all v ∈ out neighbours of root do
5:          p ← p ∪ {(root, v)}
6:      t ← 0
7:      visited ← ∅
8:      mst ← ∅
9:      while p ≠ ∅ do
10:         {(u, v)} ← minimum weighted edge in p
11:         p ← p \ {(u, v)}
12:         if v ∉ visited then
13:             visited ← visited ∪ {v}
14:             mst ← mst ∪ {(u, v)}
15:             t ← t + w(u, v)
16:             for all v ∈ out neighbours of u do
17:                 p ← p ∪ (v, u)
18:     return mst, t
```

For this algorithm I implemented two versions, one using an adjacency matrix representation of the graph and the other, an adjacency list and a binary heap. The second implementation follows the pseudo code more closely as that is a more optimised version. The two algorithms only differ in the data structures used to represent the problem but the nature of the code is the same. Figure 5.4 shows how the algorithm works. In Figure 5.4d, the edge (A, B) (highlighted in red) would create a cycle as the vertex B is already part of the minimum Spanning Tree and thus is ignored. The next least weighted edge in the graph is (A, C) which doesn't create a cycle and

this can be added to our growing tree. After this, all the vertices are connected by an edge and the algorithm is finished.

5.3.1 Implementation

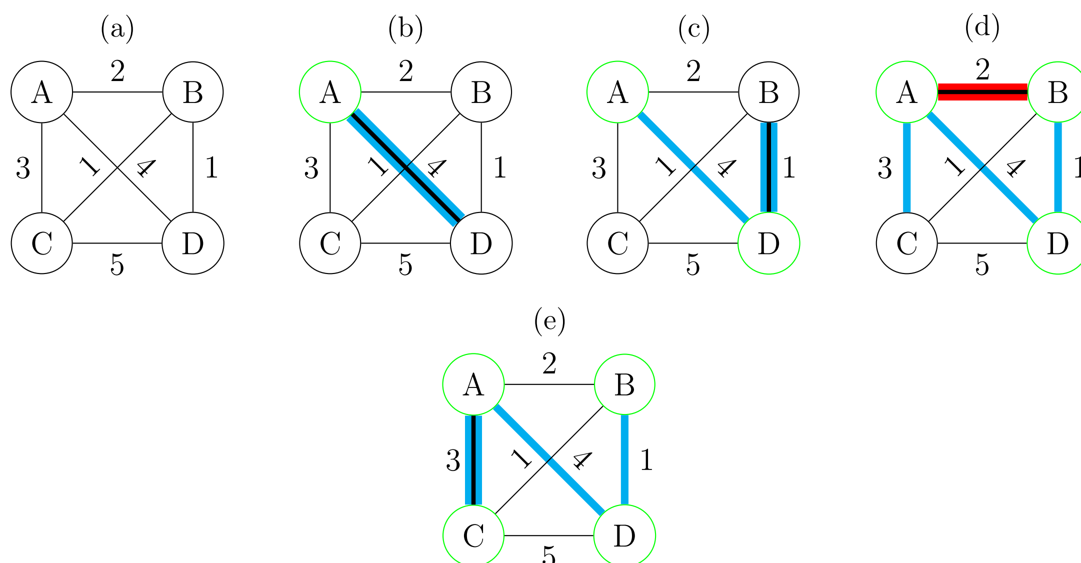


Figure 5.4: Example of Prim's algorithm

Adjacency Matrix Implementation

Prim's algorithm was the first one I implemented and it was simpler to use an adjacency matrix so I could get comfortable with the GAP language. Therefore, I implemented this variation of the algorithm first although it is not the most optimised version. The data structure to represent the graph is a 2D matrix where the element at (i, j) is the least cost weighted edge that exists for any two adjacent u and v as there may be parallel edges (see Figure 5.5). The premise of this algorithm is to perform a linear scan through the matrix and for all u that have previously been selected, and for all v that have not been previously selected and find the least weighted edge connecting an already seen vertex u with a new vertex v.

Adjacency List Implementation

In sub section 5.3.1, to find the minimum edge, all the edges where vertex v of that edge is not already part of the minimum spanning tree are scanned. This is a lot of repeated work to find the next minimum edge that doesn't create a cycle. We can optimise the first version of this algorithm in two ways – using an Adjacency List and Binary Heap (see Section 2.1). There is also a minor optimisation which breaks out the main loop when $|V| - 1$ edges have been added to the minimum spanning tree as this is the maximum number of edges possible for a minimum spanning tree with V vertices.

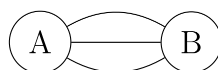


Figure 5.5: Example of Parallel Edges

By using an adjacency list, we only search the out neighbours of a vertex u, instead of every vertex v, which may not be adjacent to u whilst also using less space to represent the graph. The advantages and disadvantages of this are outlined in Section 2.1.

Similarly to the adjacency matrix implementation, this algorithm can be seen as two parts - building the graph and running the algorithm. When I first implemented this version, I realised two things which I hadn't considered prior. The first is that, the graph must be undirected. In GAP, the user may use a graph that is directed, so when the representation of the graph is being built, we must convert the directed graph into an undirected one. This is done by adding another edge in the opposite direction with the same weight if there is not already one. This results in a symmetric graph. The second consideration, which I found to be often ignored in examples of this algorithm and (pseudo) code implementations of graph algorithms is the case when a vertex u has multiple edges to a vertex v - also known as parallel edges as shown by Figure 5.5.

The adjacency list representation typically uses a hash map where the key is vertex u and its value is a list of edges. For a weighted graph, we can instead use a second hash map, as the value to the key u in the first hash map and the value of the second hash map is the weight of the edge. However, this isn't possible for parallel edges, as the second key (vertex v) will be overwritten every time an edge is added between u and v. Prior to only including the lowest weighted edge where there were parallel edges, I was iterating through every edge which unsurprisingly decreased the performance of the algorithm.

The primitive solution to parallel edges issue is where the value of the second hash map may be a list containing all the weights of the edges between vertices u and v. This means when we traverse the graph, we must also traverse all the parallel edges of the edge (u, v) too. However, this is a very naive solution which I hadn't realised until I did the analysis. Instead of storing every edge when building the graph representation and iterating them during the algorithm phase, we only need to store the minimum edge. Therefore, we can revert back to the original hash map implementation where the key of the first map is vertex u, the value is another hash map where the key is vertex v and its value is the least weighted edge between vertices u and v instead of a list of all the possible edges.

With these two additional considerations to the implementation, the first phase becomes slightly more complicated and requires some safety checks to ensure we don't get key access errors and build the graph correctly. From some analysis, when looping through all the edges, the algorithm, on a graph with 1000 nodes would take in the region of 30 seconds. By reducing the edges to one, the time taken was reduced to around 7 seconds.

5.4 Kruskal's Algorithm

The history of this algorithm is far simpler than that of Prim's Algorithm (see Section 5.2). Kruskal's algorithm was developed by Joseph Kruskal and is described in his 1956 paper [35].

Kruskal's algorithm solves the minimum spanning tree problem, similarly to Prim's but in a different way. The idea of this algorithm is to 'find a safe edge to add to the growing forest by finding, of all the edges that connect any two trees in the forest, an edge (u, v) of least weight' [11, Chapter 23.2]. Algorithm 2 shows the pseudo-code for Kruskal's Algorithm.

Algorithm 2 Kruskal's Algorithm [11]

```
1:  procedure KRUSKAL(G)
2:      parent, rank = []
3:      t ← 0
4:      mst ← ∅
5:      for all u ∈ V(G) do
6:          parent[u] ← u
7:          rank[u] ← 1
8:      q ← sort edges of G(V,E) by ascending weight
9:      for all (u, v) ∈ q do
10:         if Find(u) ≠ Find(v) then
11:             mst ← mst ∪ {(u, v)}
12:             t ← t + w(u, v)
13:             Union(u, v)
14:     return mst, t
```

A visual example of this algorithm is shown by Figure 5.6. The thicker lines represent the edges being considered where cyan highlighted edge shows a valid edge and red highlighted edge shows an invalid edge that will create a cycle.

5.4.1 Implementation

The implementation is fairly simple, with the idea simply being sort the edges by their weights and keep adding cycles to the forests that do not create a cycle until we have a tree. As we are adding the edges in increasing order, it is guaranteed that the spanning tree achieved is minimal. We don't need to use any specific graph representation for this and simply just need to create a list holding all the edges as (w, u, v). We then sort this list according to the edge weight, w. In GAP, I used the `StableSort` function, which maintains the order the edges are sorted which isn't strictly required but does ensure, we can get the same output each time and removes any uncertainty as the same order of elements is retained. The `StableSort` function also uses merge sort which has $O(n \log n)$ complexity in the best, average and worst case [37] where n is the number of elements. Although the mainstream languages such as Java and Python use Timsort which is a combination of merge and insertion sort and is similar in terms of performance as merge sort but has $O(n)$ complexity performance in the best case [46].

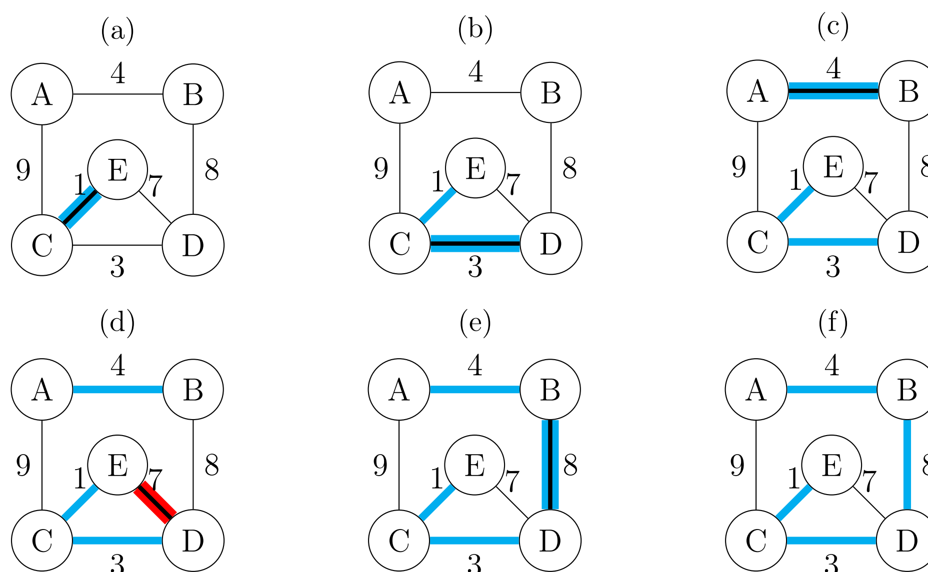


Figure 5.6: Example of Kruskal's algorithm

Disjoint Set Union (DSU)

A disjoint set [15] is a data structure that stores a collection of non-overlapping (disjoint) sets. Union allows the merging of these sets. This was a key data structure in Kruskal's algorithm, Borůvka's algorithm, as well as Karger's algorithm later on.

Definition 5.1 (disjoint sets)**.** The sets A and B are called disjoint if $A \cap B = \varnothing$. [30, Section 4].

Definition 5.2 (forest)**.** A forest is a graph, in which all of whose connected components are trees. In particular, a forest with one component is a tree. [5, Section 3]. Connected components were defined in Definition 2.17.

Kruskal's algorithm makes use of a technique called Disjoint Set Union [15] which includes two methods, Find and Union. These allows us to, given multiple disjoint sets, to combine them and check which set an element belongs. In the context of this algorithm, it will allow us to detect cycles when adding an edge to our forest.

Find and Union

- **find**

  Find finds which set a particular set belongs to by finding the root of that component by following each nodes parents nodes until the parent of the node is it self.

- **union**

  To unify to sets, find the root nodes of each set and if the root nodes are different, let one of the roots be the parent of the other.

The most basic, unoptimised implementation of this algorithm operates as follows:

- 5.7a - Let us represent the sets as trees.
- 5.7b - Union of sets A and B where A is arbitrarily selected to be the parent.
- 5.7c - Union of sets C and D where C is arbitrarily selected to be the parent.
- 5.7d - Union of sets A and C where A is arbitrarily selected to be the parent.

Pseudo-code for these unoptimised functions are shown below by Algorithm 3.

Algorithm 3 Find and Union with no optimisations

```
1: procedure FIND(parent, node)
2:     if parent[node] is node then
3:         return node
4:     return Find(parent, parent[node])
5: procedure UNION(parent, x, y)
6:     a ← Find(parent, x)
7:     b ← Find(parent, y)
8:     if a ≠ b then
9:         parent[b] ← a
```

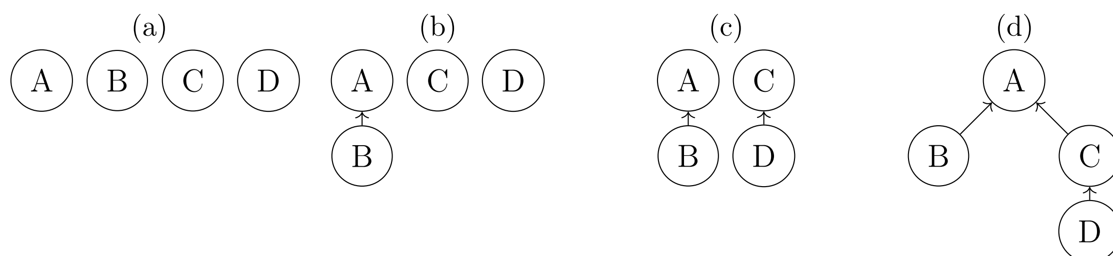


Figure 5.7: Example of unoptimised DSU Algorithm

This, however is naive and inefficient as 'the trees [may] degenerate into long chains' [15] which can take $O(n)$ time to traverse when calling `find` as it may search the entire tree to find the parent. In Figure 5.7 if set A was one long chain and we were trying to find the parent of node n which was at a depth n from A, finding A would have $O(n)$ complexity. We can optimise using two methods that will allow us to perform `find` and `union` much faster.

Path Compression

Path compression can speed up `find` by instead building long chains of nodes, we set all the nodes to point to the same parent. The steps for path compression are as follows:

- 5.8a - This is a tree where A is the parent of all the sets and finding the parent of F would take n iterations, where n is the height of the tree. Note: this kind of tree would not be possible when using path compression.

- 5.8b - During a `find` operation, path compression will take place which would transform a tree such as in Figure 5.8a into Figure 5.8b where all the nodes have a much shorter path length to their parent A. We 'compress' the path and explains the derivation for the name of this operation. Visually and computationally, the performance increase is clear as now the parent can be found with amortized $O(\log v)$ complexity [34]. This is because the height of the tree grows at a much slower rate and although the official complexity is $O(\log|V|)$, in practice is closer to $O(1)$ [47].

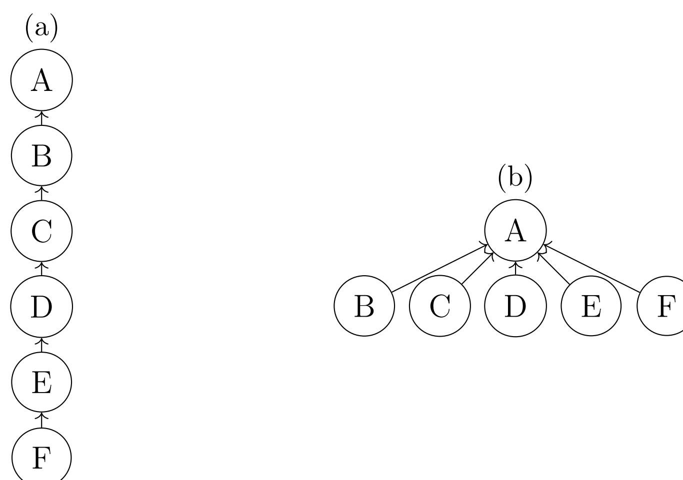


Figure 5.8: Example of DSU Path Compression

With path compression, the pseudo-code for Find is shown by Algorithm 4.

Algorithm 4 Find with Path Compression

```
1: procedure Find(parent, node)
2:     if parent[node] is node then
3:         return node
4:     parent[node] ← Find(parent, parent[node])
5:     return parent[node]
```

Union by Rank

This is the second optimisation which helps decide which set to set as the parent, effectively, in which way to attach the two trees. Previously in the unoptimised version, the second tree was always attached to the first one. This can lead to trees containing chains of length $O(n)$. There are various heuristics that we can use such as the size of the trees - number of nodes in the tree, or the depth of the tree which is different to the size of the tree. Both achieve the same goal of attaching the smaller tree, the one with the lower rank, to the tree with the higher rank. In the case of a tie, we have two similar sized trees so we simply attach one tree to the other - the orientation of this doesn't matter. Both of these methods have the same time complexity [15].

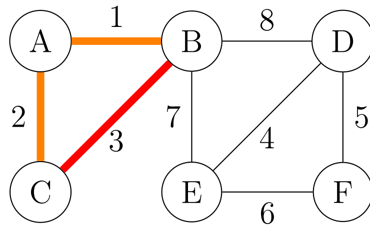
In Kruskal's algorithm, the DSU method can be used to detect cycles as when we add an edge, we can check if the two vertices already belong to the same set. This

can be best demonstrated by an example as shown by Figure 5.9. We add the edges, checking that u and v, the two vertices on either side of the edge don't belong to the same group. If they don't, add the edge to the tree and perform a union on the two trees, otherwise discard it. With ranking, the pseudo-code for Union is shown by Algorithm 5.

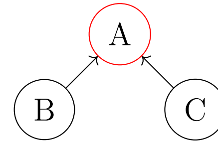
Algorithm 5 Union with Ranking

```
1:  procedure UNION(parent, rank, x, y)
2:      a ← Find(parent, x)
3:      b ← Find(parent, y)
4:      if rank[a] < rank[b] then
5:          parent[A] ← b
6:      else if rank[b] > rank[b] then
7:          parent[b] ← a
8:      else
9:          parent[b] ← a
10:         rank[a] ← rank[a] + 1
```

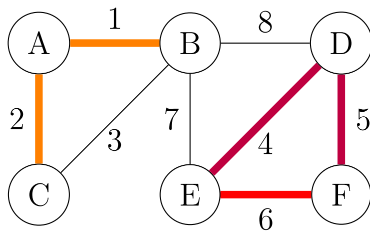
These optimisations make a significance difference to the performance of Kruskal's algorithm as shown by Figure 5.10 which shows a similar slope of the curve but with much faster execution time. The usual algorithm is already using union by ranking so this increase in performance is solely due to path compression.



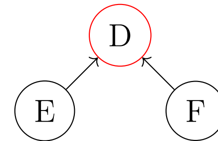
(a) The edge (B, C) would create a cycle



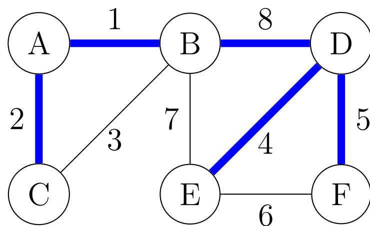
(b) Vertices B and C both belong to to the same set (orange set)



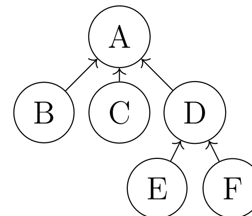
(c) The edge (E, F) would create a cycle



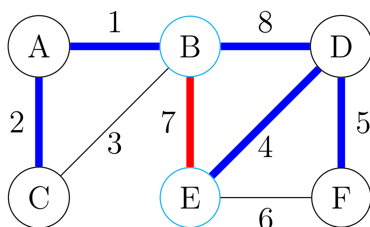
(d) Vertices E and F both belong to to the same set (purple set)



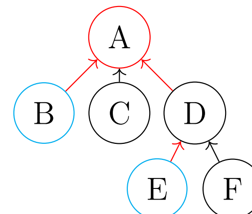
(e) Adding the edge (B, D)



(f) Union of the orange and purple sets to create blue set



(g) Adding the edge (B, D)



(h) Both vertices B and E have A as a parent

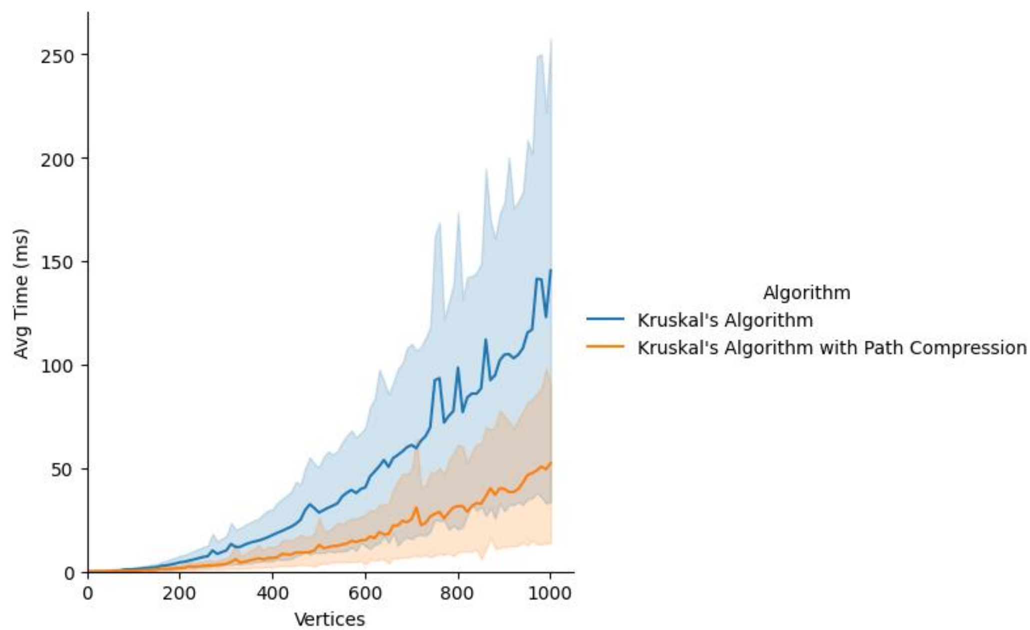
Figure 5.9: Example of DSU in Kruskal's algorithm

5.5 Borůvka's Algorithm

This is another greedy algorithm first published by Otakar Borůvka in his 1926 paper [40]. This is the first solution of the minimum spanning tree problem and the algorithm provides the basis for some of the other algorithms and shares similarities with both Prim's and Kruskal's algorithm.

The idea of this algorithm is to first find the minimum weighted edge incident for each u - a procedure similar to Prim's algorithm where we greedily pick the lowest weighted edge for the vertex we have selected. We then add this vertex to the tree and repeat a similar process for finding the minimum weighted edge from each constructed tree and adding all those edges the forest. When there is only one tree left, the min-

Figure 5.10: Kruskal's algorithm with and without Path Compression



imum spanning tree is found. This makes use of the Find and Union functions first used in Kruskal's algorithm and thus its similarity to Borůvka's algorithm is realised. The idea behind this algorithm, shown as a list of steps as follows:

Borůvka's algorithm Idea

1. Initialise all vertices as their own individual set.
2. While there is more than one tree we find the minimum weighted edge that connects one tree to another.
3. Add this edge to the tree

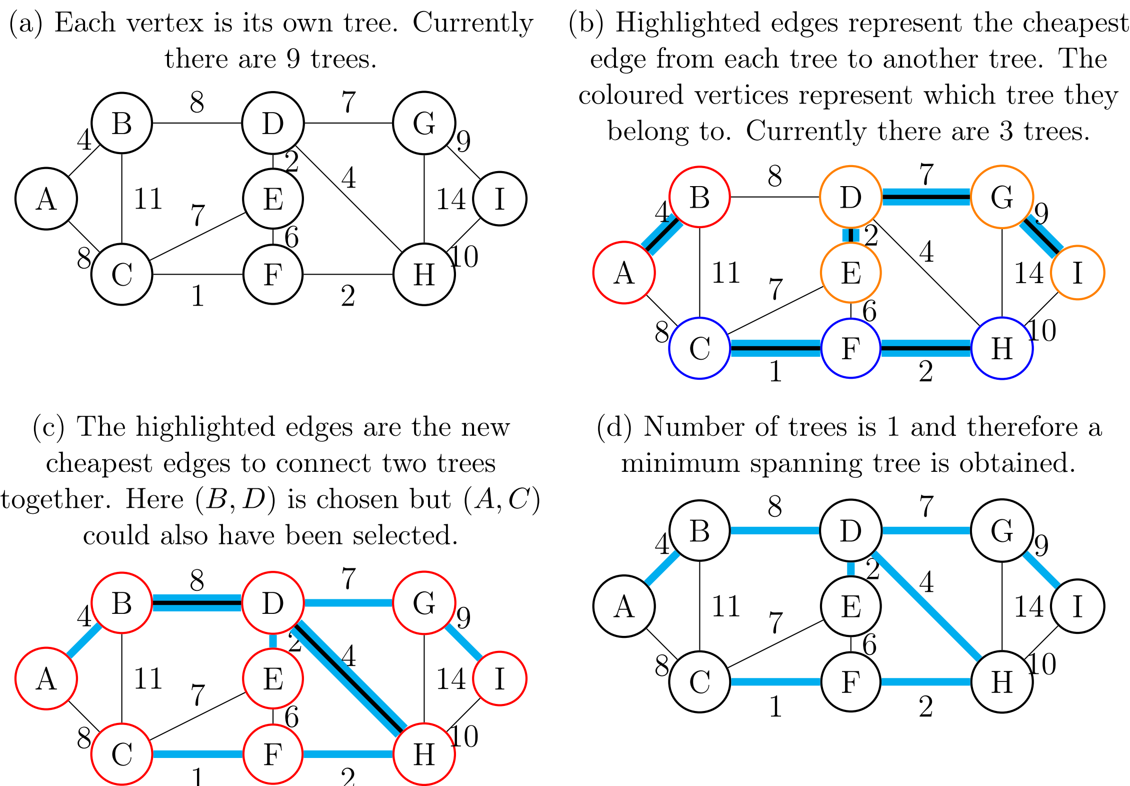
A visual example of this algorithm as shown by Figure 5.11 may make the algorithm more clear.

The pseudo-code for this algorithm is shown by Algorithm 6.

Lines 7, 8, 16 and 17 can be found using the Find operation discussed in Disjoint Set Union section.

Algorithm 6 Borůvka's Algorithm

```
1: procedure BORŮVKA(G)
2:     trees ← |V|
3:     mst ← ∅
4:     t ← 0
5:     while trees > 1 do
6:         for all (u, v) ∈ E do
7:             x ← parent of tree containing u
8:             y ← parent of tree containing v
9:             if x ≠ y then
10:                if edge to another tree ∉ x or w(u, v) < existing edge from x then
11:                    set edge of x to (u, v)
12:                if edge to another tree ∉ y or w(u, v) < existing edge from y then
13:                    set edge of y to (u, v)
14:        for all u ∈ V do
15:            if (u, v) was found from u then
16:                x ← parent of tree containing u
17:                y ← parent of tree containing v
18:                if x ≠ y then
19:                    mst ← mst ∪ {(u, v)}
20:                    t ← t + w(u, v)
21:                    Union(x, y)
22:                    trees ← trees - 1
23:        reset the cheapest edges from each tree
24:    return mst, t
```



(a) Each vertex is its own tree. Currently there are 9 trees.

(b) Highlighted edges represent the cheapest edge from each tree to another tree. The coloured vertices represent which tree they belong to. Currently there are 3 trees.

(c) The highlighted edges are the new cheapest edges to connect two trees together. Here (B, D) is chosen but (A, C) could also have been selected.

(d) Number of trees is 1 and therefore a minimum spanning tree is obtained.

Figure 5.11: Example of Borůvka's Algorithm

5.6 Analysis

To read about the process I used to create and analyse the graphs, see automated analysis section. For the analysis of minimum spanning trees, the graphs need to be connected so when creating the random digraphs I passed 'IsConnectedDigraph' as a filter. Also, to make the testing easier, the graphs had unique edge weights as to guarantee one solution. This made it possible to test the exact output as there was only one solution and therefore I could run multiple Minimum Spanning Tree algorithms and verify the output was the same. There are also graphs created for the individual graph algorithms with the varying edge probabilities but provided no insight into the comparison between the algorithms but does show that the graphs perform worse as the density of the graphs increase. These graphs can be found in Appendix C.

5.6.1 Comparison of Prim's and Kruskal's algorithm

The first plot to look at is the overall comparison of Prim's and Kruskal's algorithm. This plot shows the range of times for each of the densities and the highlighted line is the average of the times taken of all the data.
This chart is shown by Figure 5.12. It is clear that for every density Kruskal's is faster and the time taken is increasing at a lower rate than Prim's algorithm. This graph can be misleading so it would be better to look at the individual comparison for each edge weight probability. We can also see the difference between Prim's algorithm with Kruskal's algorithm with path compression (see Figure 5.13) in which the difference in the range of average times is greater. This is more easily seen with Figure 5.14.

Figure 5.12: Overall comparison between Prim's and Kruskal's Algorithm

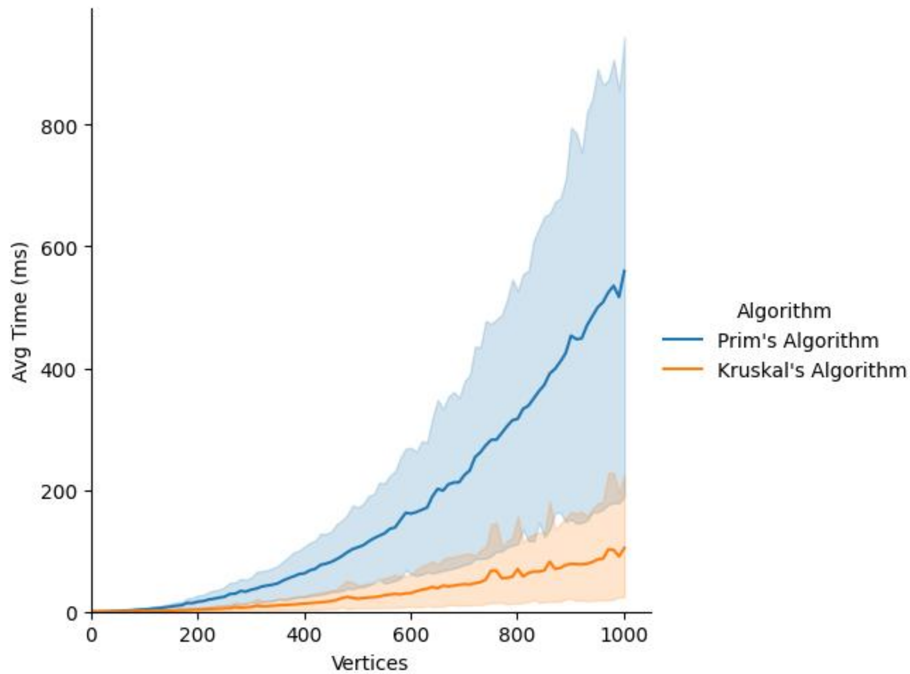
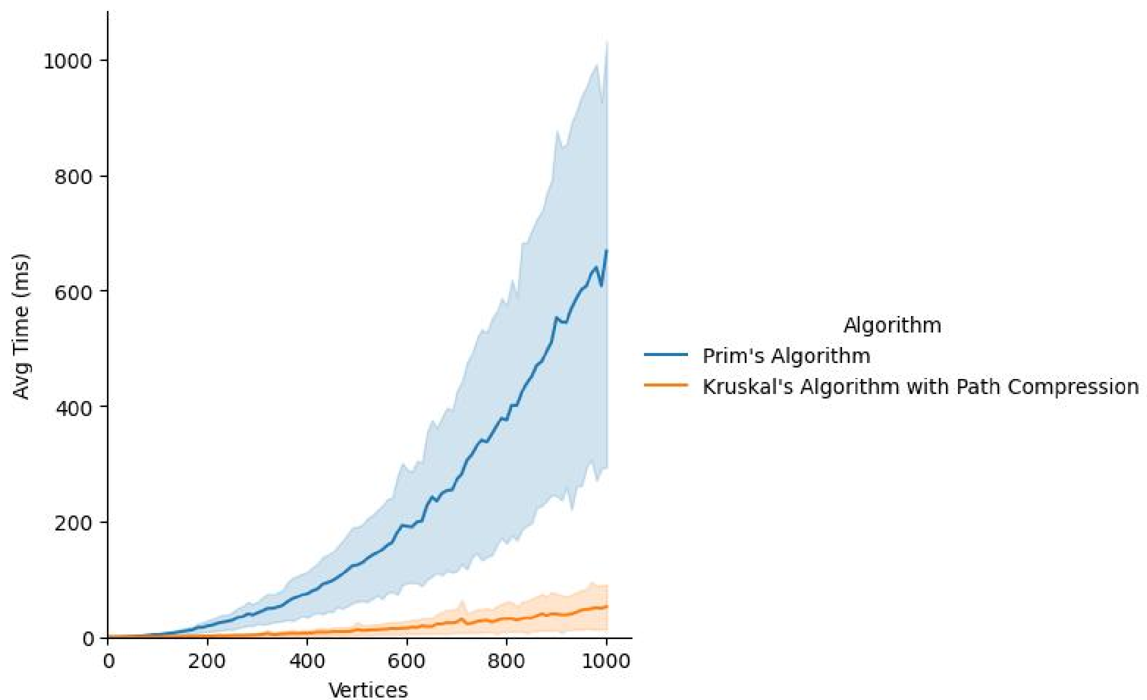
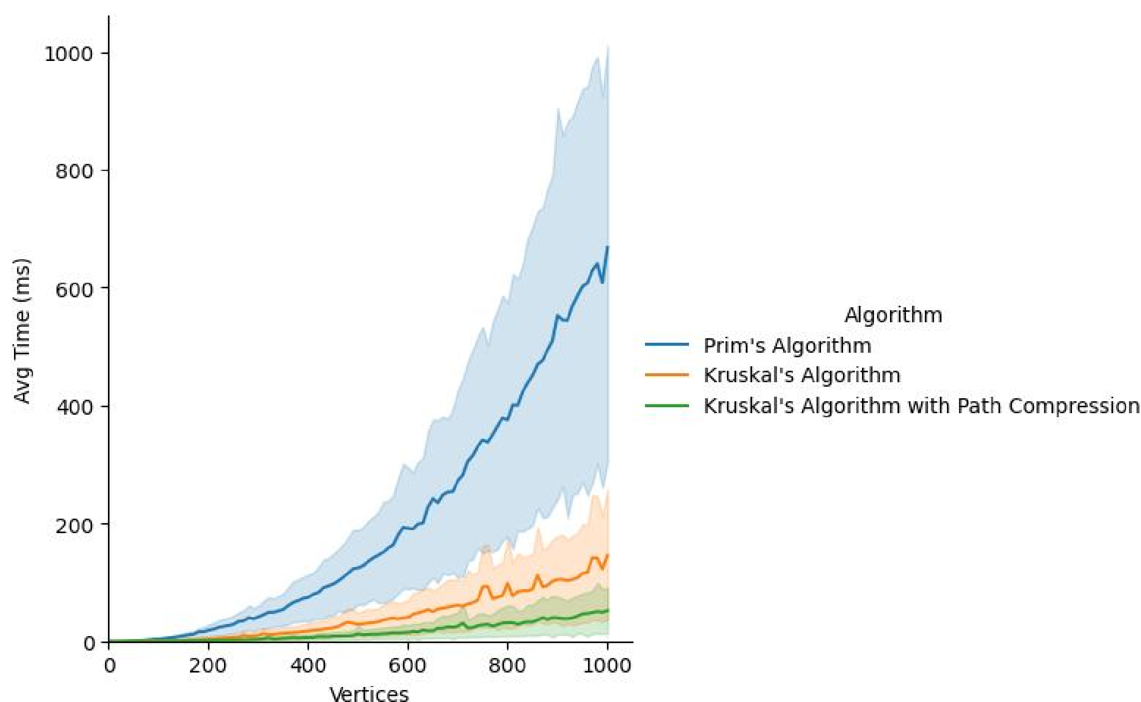


Figure 5.13: Overall comparison between Prim's and Kruskal's Algorithm with Path Compression



Looking at the graph of Kruskal's against Prim's algorithm (Figure 5.15) with edge

Figure 5.14: Overall comparison between Prim's and Kruskal's Algorithms (with and without path compression)



probability 1.0 - a complete graph - we can see the trend in performance is similar.

At the opposite end of the spectrum, comparing the algorithms with sparse graphs (see Figure 5.16), the result is the same.

The average time complexity of Kruskal's Algorithm is $O(|E|\log|E|)$ where $|E|$ is the number of edges. This is due to the sorting of the edge weights at the start of the algorithm. The complexity for Prim's algorithm is $O(|E|\log|V|)$ [11, Chapter 4.3]. These complexities suggests that as the logarithmic part of the complexity depends on number of edges and vertices respectively and that Prim's algorithm would perform faster on denser graphs than Kruskal's - whose performance is more dependant on the number of edges. However, from the plots and data collected it is clear that Kruskal's algorithm is more performant in every case which led me to profile Prim's code to try to deduce why as the implementation didn't have any obvious bottlenecks and was well optimised as far as I could observe.

There is another implementation that is even further optimised than the adjacency list and binary heap implementation I did which makes use of a different type of heap called Fibonacci Heap. With a Fibonacci heap, a time complexity of $O(|E| + |V|\log|V|)$ [11, Chapter 21] can be achieved. This bound is better than $O(|E|\log|V|)$ achieved using the procedure I implemented. However, a Fibonacci Heap isn't implemented in GAP so I didn't use it in my implementation but I would have written my own if I had more time.

I profiled the code for various sizes of digraphs and found that for the most part

Figure 5.15: Prim's vs Kruskal's Algorithm with 1.0 edge probability

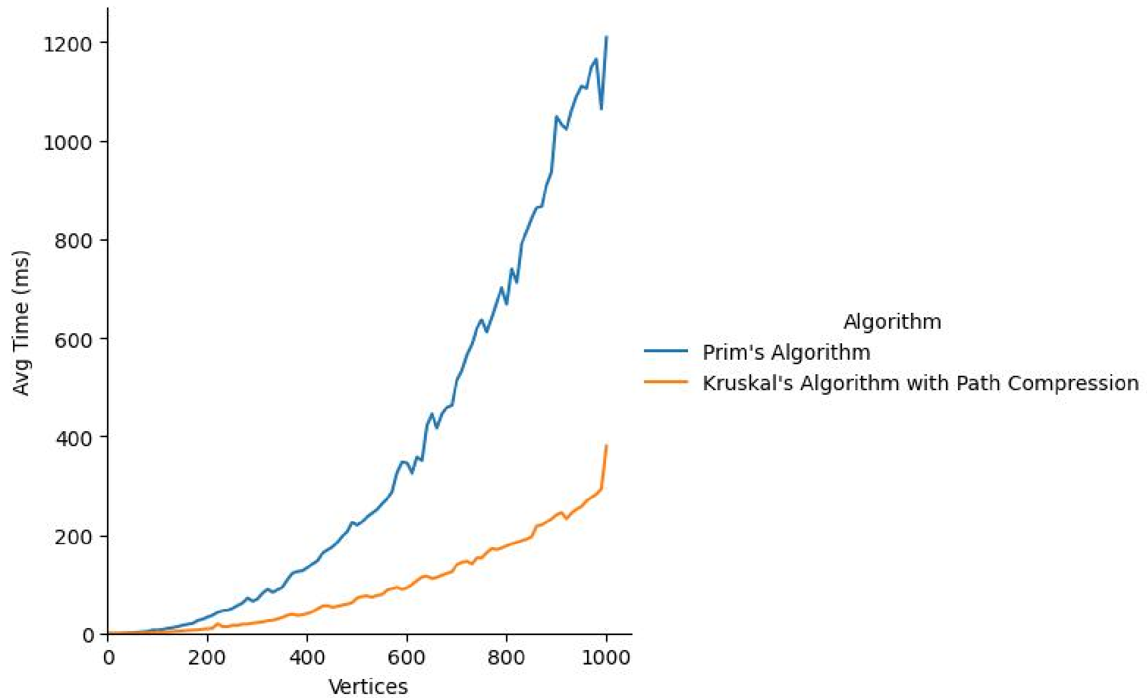


Figure 5.16: Prim's vs Kruskal's Algorithm with 0.01 edge probability

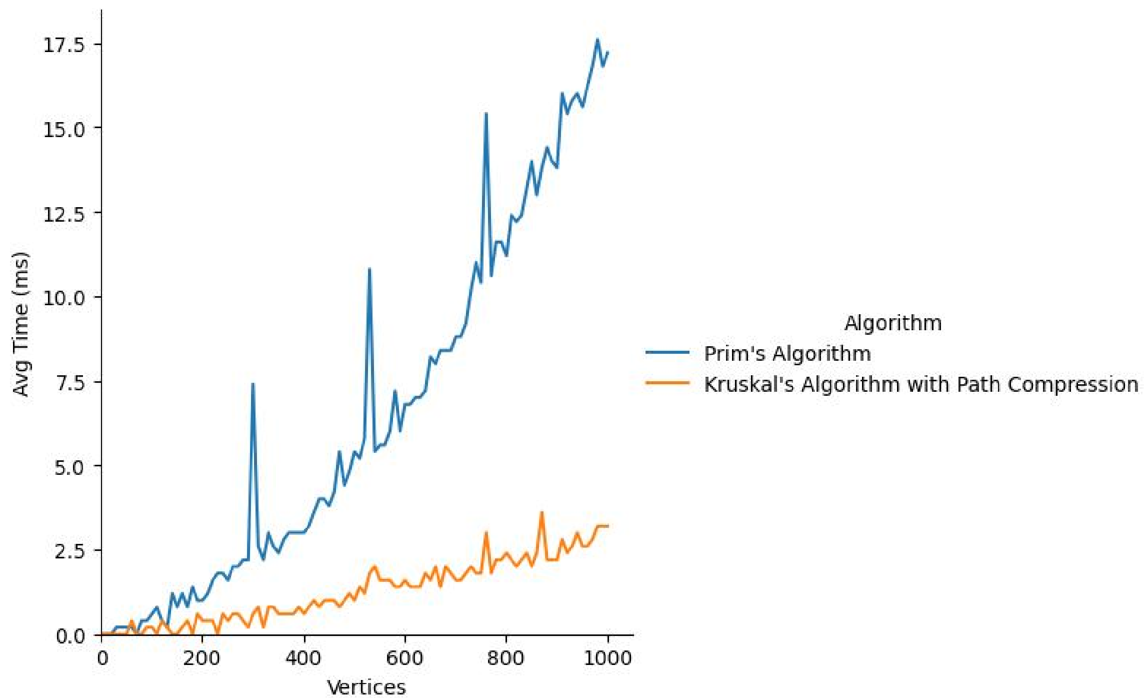
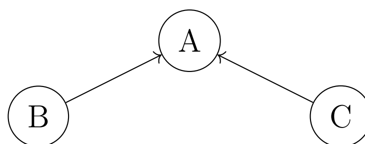


Figure 5.17: Case where Prim's algorithm fails with digraph



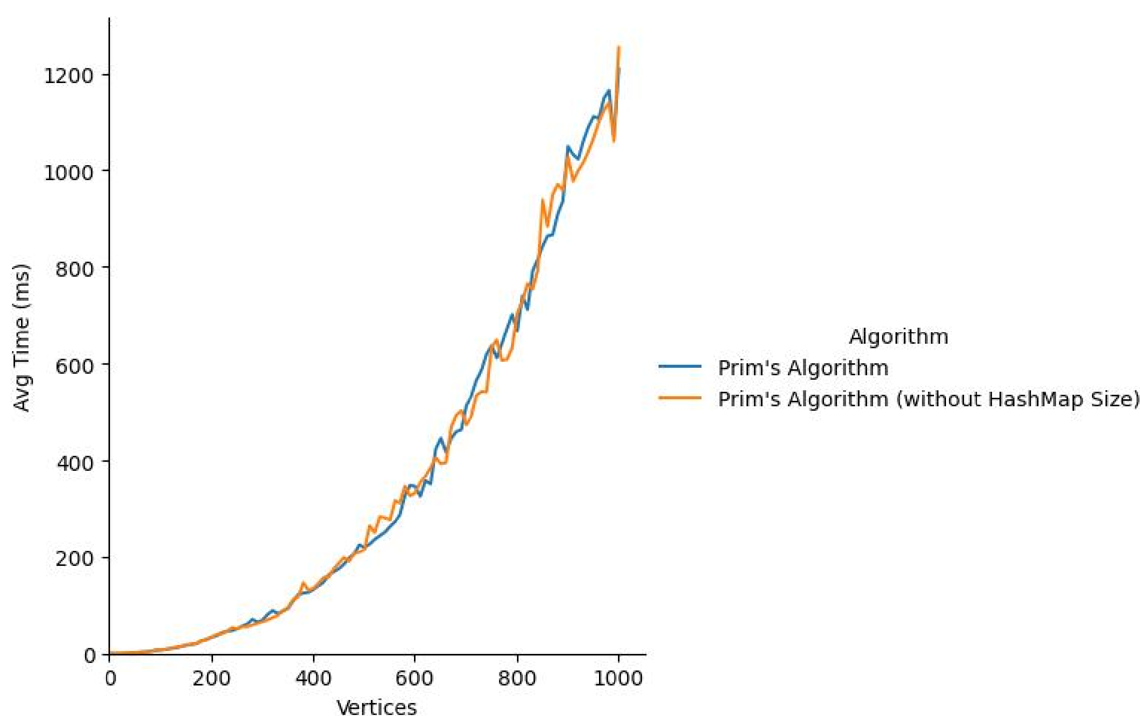
the implementation was working as intended with the expected number executions for certain lines of code but I discovered that the KeyValueIterator in GAP - a function to loop through the entries of keys and values of a hash map at the same time, which although executed the correct amount of times, was where most of the time of the algorithm was spent. Pushing to the Binary Heap should also take a reasonable amount of time, so to see high execution times for method calls to push to the binary heap is not unsurprising.

Another reason that Prim's algorithm is slower but doesn't explain the performance represented by the graph is the fact that my implementation has to convert a digraph in to an undirected one by adding a reverse edge if it doesn't exist. This could potentially double the number of edges that needs to be iterated, which doesn't affect the big O complexity but would result in a worse performance. This is because Prim's algorithm attempts to find a path from each vertex and in a case where there is not path to the other node, the algorithm will fail, as shown by Figure 5.17 where if we started on vertex A, then we can never reach the other vertices.

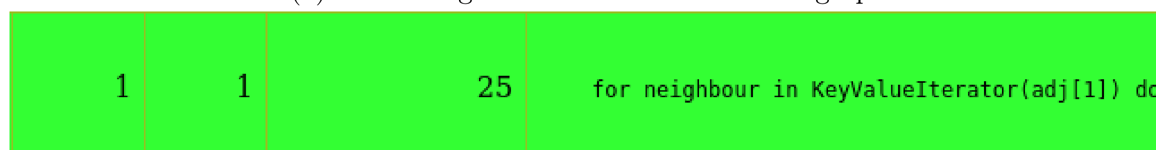
GAP provides a profiling package [42] which allows for line-by-line profiling of GAP code. Using the example below where I profiled Prim's algorithm on a graph with 3 vertices with edge probability of 1.0 so it is a complete graph - the algorithm should be almost instant, taking a few milliseconds at most for any method call but I found that the KeyValueIterator taking a significant amount of time for a given number of executions. In Figure 5.19a, the line is executed once but takes 25ms. The same trend can be noticed in Figure 5.19b in which the algorithm is ran on a digraph with 100 vertices and has 99 accesses (due to a small optimisation added, mentioned in the Adjacency List and Binary Heap Implementation section. Even with just 99 accesses, this took 2551 ms which seems slow for a simple key access. Therefore, I think that the implementation of the KeyValueIterator may not be optimised in C and is therefore the potential cause of the performance issues with my Prim's Algorithm implementation.

Kruskal's algorithm does not make use of this data structure and therefore is void of these issues. A later algorithm I implement (see Section 6.2) makes use of a KeyValueIterator also but performs as expected. With Prim's algorithm, I tried to optimise the hash map by defining an original size (equal to 4x the number of vertices as to give the hash map sufficient buckets to reduce collision but also the total size of it) which I didn't do for Dijkstra's later on as I ran the same algorithm but one with a pre-defined hash map size and one without. The results for this are shown by Figure 5.18. As we can see, the run times are very similar and this had a negligible effect on the performance, if any.

Figure 5.18: Prim's Algorithm with and without hash map size specified

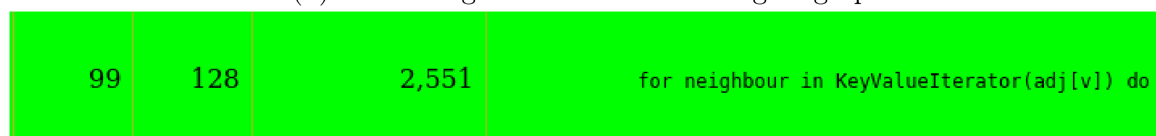


(a) Prim's Algorithm Profiled on small digraph



1	1	25	`for neighbour in KeyValueIterator(adj[1]) do`

(b) Prim's Algorithm Profiled on large digraph



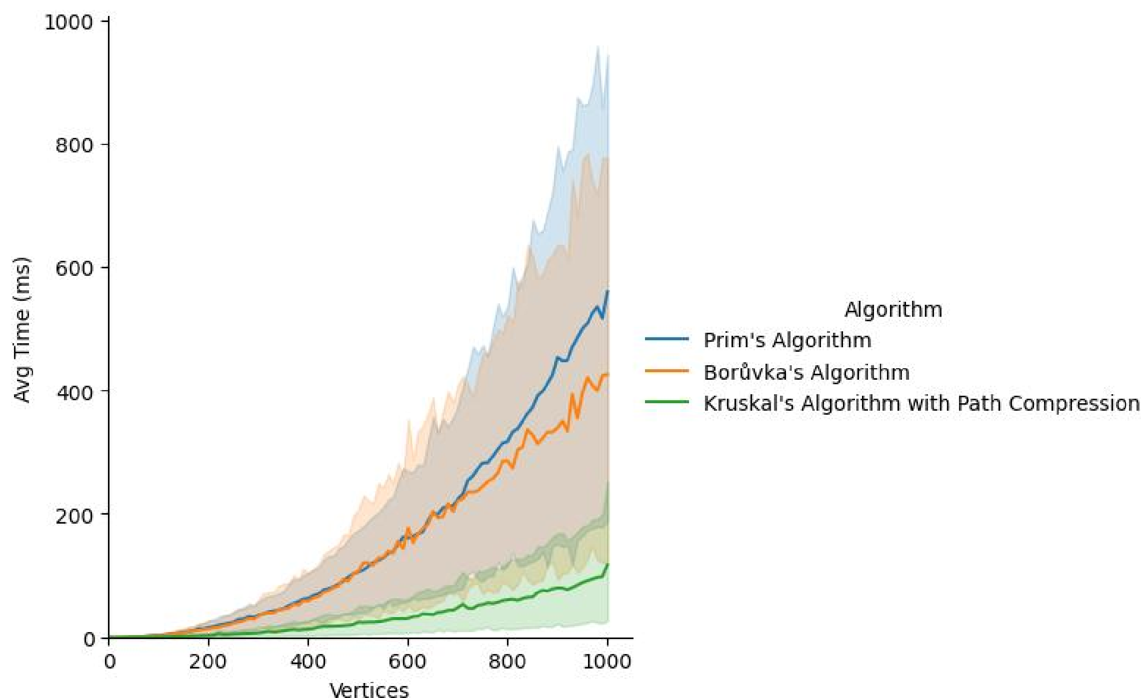
99	128	2,551	`for neighbour in KeyValueIterator(adj[v]) do`

Figure 5.19: Prim's Algorithm Profiled

5.6.2 Comparison of all MST algorithms

Borůvka's algorithm has a time complexity of $O(|E|\log|V|)$, which is the same as Prim's algorithm. The calculation for the complexity comes from the fact the algorithm takes $O(\log|V|)$ iterations in the outer loop as the number of trees at least halves each iteration. We do this $|E|$ times and therefore the overall complexity is $O(|E|\log|V|)$. The performances for all three algorithms are shown by Figure 5.20 in which we can see that Prim's and Borůvka's algorithm have a very similar performance up to 700 vertices until they start to diverge. Even though both algorithms are slower than Kruskal's this leads me to think that there may not be any issues with the Prim's algorithm or the implementation of the hash map.

Figure 5.20: Prim's vs Kruskal's vs Borůvka's algorithm

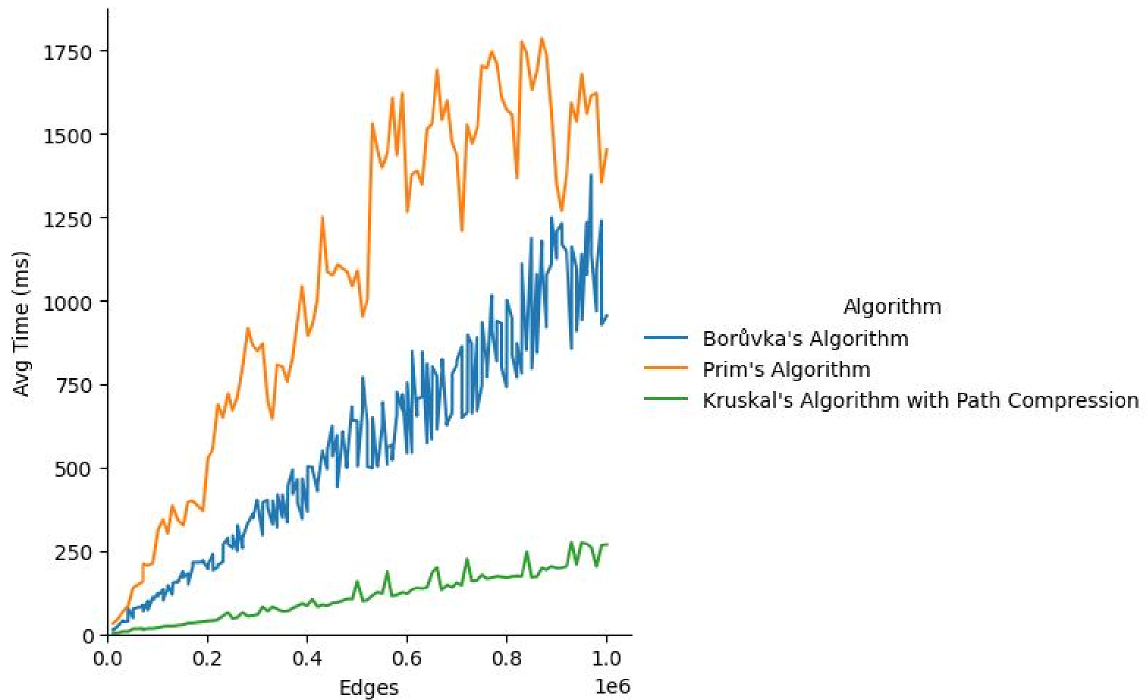


I therefore ran an experiment with a fixed number of vertices (1000) for various number of edges as shown by Figure 5.21 which shows that even for the same number of vertices with varying number of edges that it is still the worst performing algorithm even though the complexities are the same for Prim's and Borůvka's algorithm. Kruskal's algorithm is still much faster, and increasing at a slower rate than the other two algorithm, solidifying my original conclusions that this was the correct algorithm of choice for minimum spanning tree problems.

5.6.3 Comparison with Scipy's Implementation

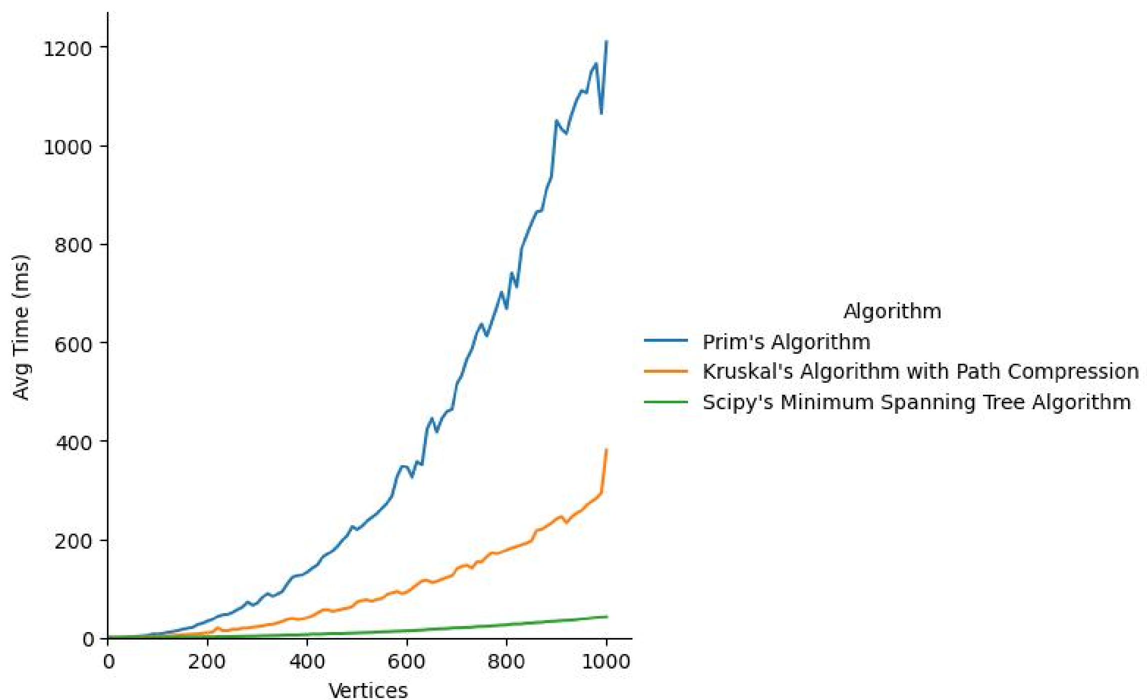
I decided to compare my implementations with an industry standard so I chose Scipy [10] which has a minimum spanning tree function that works on edge weighted graphs. The function uses Kruskal's algorithm as stated on the API documentation. I created random edge weighted graphs using Scipy for the same probabilities as I did for the GAP algorithms. The performance of scipy is significantly better than that even of my fastest implementation in GAP. It must be stated that the comparison is entirely equivalent as the graphs that were used with both these algorithms were different. I expected scipy to be significantly faster so these experiments were run to gauge an idea of the difference. Given more time, I could have automated it so that I could write my GAP graphs to file and then read the file in a separate python script that would pass that graph into Scipy. Regardless, I do not think it would have added any significance detail to the comparison. The graph is shown by Figure 5.22 and the performance of the implementations is immediately obvious. Prim's algorithm performs the worst, with an exponential curve so the performance is deviating much faster than

Figure 5.21: Prim's vs Kruskal's vs Borůvka's algorithm for fixed 1000 vertices with varying number of edges



the other 2. Kruskal's implementation is far quicker but still slower than that of Scipy.

Figure 5.22: Scipy vs Kruskal's vs Prim's for 1.0 edge probability



In this chapter, I described spanning trees, minimum spanning trees giving examples for each of these leading to algorithms that find the minimum spanning trees within a graph. I then analysed the graphs and their performance on various graph densities and found that Kruskal's algorithm is optimal in every case. This is also the algorithm used by the `minimum_spanning_tree` function in the Scipy library.

Chapter 6

Shortest Path Algorithms

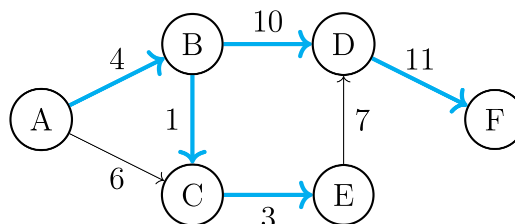
This chapter focuses on four shortest path algorithms; Dijkstra's, Bellman-Ford, Floyd-Warshall and Johnson's Algorithm. The first two algorithms are known as Single Source Shortest Path (SSSP) Algorithms in which we find the shortest path from a single source vertex to every other vertex. The two latter algorithms are All Pairs Shortest Path Algorithms (APSP) in which we find the shortest distance from every pair of vertices. SSSP algorithms are a subset of ASPS.

6.1 Shortest Paths

There are two types of shortest path algorithms that I will explore. This section aims to give examples of these types of algorithms without a specific algorithm attached to it as to give a overview of what the algorithms are achieving.

6.1.1 Single Source Shortest Path algorithms

These algorithms require a source vertex s and will calculate the shortest distance to every other $v \in V$. An example is shown by Figure 6.1 where the source vertex is A. From the coloured edges we can see the shortest path from A to every other vertex.



Shortest Distances from $A \to B : 4, C : 5, D : 14, E : 8, F : 25$

Figure 6.1: Example of Single Source Shortest Path algorithm

6.1.2 All Pair Shortest Path algorithms

An example for this is difficult to show via a graph and will be easier to demonstrate using a matrix. Using the same graph as before (see Figure 6.1), the output for an

$$
\begin{array}{c c}
 & \begin{array}{cccccc} A & B & C & D & E & F \end{array} \\
\begin{array}{c} A \\ B \\ C \\ D \\ E \\ F \end{array} &
\begin{pmatrix}
0 & 4 & 5 & 14 & 8 & 25 \\
- & 0 & 1 & 10 & 4 & 21 \\
- & - & 0 & 10 & 3 & 21 \\
- & - & - & 0 & - & 11 \\
- & - & - & 7 & 0 & 18 \\
- & - & - & - & - & 0
\end{pmatrix}
\end{array}
$$

Figure 6.2: APSP Example Output

APSP algorithm would be as shown by Figure 6.2. The entry at uv where U are the rows and V are the columns. Looking at (A, C) we can see the value 5, signifying the shortest distance from A to C is 5 which is indeed true if we took the path $A \rightarrow B \rightarrow C$. Entries with '-' signify there is no shortest distance from u to v, also meaning there is no path from u to v. Looking at the last row for vertex F, we can see all entries but the distance to itself is a '-' as we cannot reach another vertex from F and the distance to itself is 0.

6.1.3 Path Construction

The two APSP algorithms implemented in sections 6.4 and 6.5 construct the path and produce a similar output to that of the two SSSP algorithms seen in sections 6.2 and 6.3. However, as they find the path from every u to every v, the output is effectively is a list of outputs from the SSSP algorithms computed for every u.

The two APSP algorithms, construct the path in different ways.The Floyd–Warshall algorithm maintains a 2D list of parents from u to v is stored as u initially and a 2D list of which edges which holds which edge taken between u and v. As the Floyd–Warshall algorithm attempts to find the shortest distance from u to v through a vertex k and therefore each time a new path is found with a shortest distance, we also update the parents and edges lists. This allows us to construct and update the path as new shortest distances are found.

Constructing the path for Johnson's algorithm is far simpler as this algorithm utilises both the SSSP algorithms and taking advantage of Dijkstra's algorithm to compute the shortest distance, from which we can also obtain the parents and edges and return them as part of our solution to Johnson's algorithm.

6.2 Dijkstra's Algorithm

Dijkstra's algorithm was developed by Edsger W. Dijkstra where he published the algorithm in his 1959 paper [13].

This is a well known algorithm in graph theory and is often used to find the shortest path between two vertices. This algorithm works in a similar way to Prim's algorithm by making use of a binary heap which maintains the shortest distance to reach a vertex v. The use of a binary heap is part of the implementation and not specified in the original paper. In the graph, if parallel edges exist, only the minimum weighted edge will be kept. In the case of similar edge weighted parallel edges, the first will persist and the others removed from the data structure. This means the algorithm doesn't consider parallel edges when running as they will have been stripped from the graph. The idea for the overall algorithm is as follows:

1. Pick closest unknown vertex
2. Add it to known vertices
3. Update distances

The pseudo code for Dijkstra's algorithm is shown by Algorithm 7.

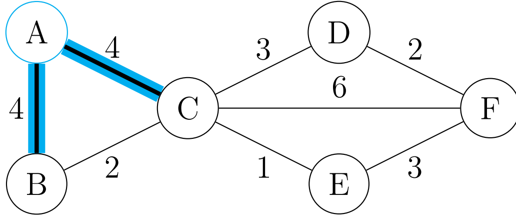
Algorithm 7 Dijkstra's Algorithm

1: **procedure** DIJKSTRA(G, source)
2: $q \leftarrow \varnothing$
3: **for all** $u \in G(V)$ **do**
4: $dist_u \leftarrow \infty$
5: $dist_{source} \leftarrow 0$
6: $q \leftarrow q \cup \{(v, dist_{source}\})$
7: **while** $q \neq \varnothing$ **do**
8: u, $d \leftarrow$ vertex with shortest distance from source in q
9: $p \leftarrow q \setminus u$
10: **if** $u \in$ visited **then** continue
11: **for all** v in out neighbours of u **do**
12: $alt \leftarrow dist_u + (u, v)$
13: **if** $alt < dist_v$ **then**
14: $dist_v \leftarrow alt$
15: $prev_v \leftarrow u$
16: **if** $v \notin$ visited **then**
17: $p \leftarrow q \cup \{(v, \text{alt})\}$
18: **return** $dist, prev$

Figure 6.3 shows a run through of Dijkstra's algorithm. The cyan highlighted edges show which edges are being considered, and the non highlighted cyan edges show the current shortest path to the vertex. The edges highlighted in red are not possible as this edge leads to a vertex that has already been visited. Visited vertices are outlined in cyan also. On the table on the right, the upper row is each node and the lower lower is the distances to each vertex, which is initialised to ∞ to begin with. As the

nodes are updated, the node row in the table is updated and the updated nodes are coloured blue.

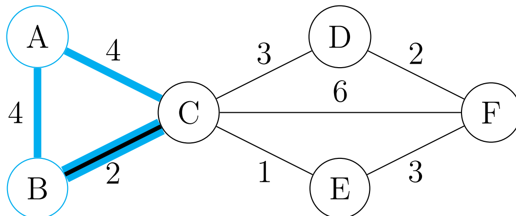
(a) Considering edges from Vertex A.As the edges (A, B) and (A, C) have the same weight, we arbitrarily select one.



(b) Updated distances from A to A,B,C is 0,4,4 respectively.

Node	A	B	C	D	E	F
Dist	0	4	4	∞	∞	∞

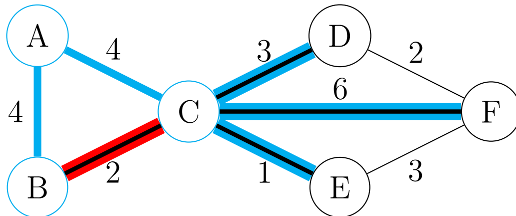
(c) B only has one out neighbour C so only consider this edge.



(d) The edge $(A, C) = 4 + 2 < 4$ as the new distance is smaller, don't update the table.

Node	A	B	C	D	E	F
Dist	0	4	4	∞	∞	∞

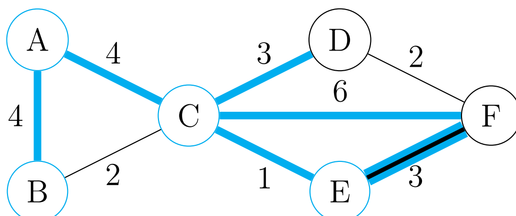
(e) Edges out bound from C considered. As vertex A is already visited, this edge is not considered and is highlighted in red.



(f) Update the distances to vertices D,E,F by adding the current distance to get to C + the respective distance between C and valid out neighbours.

Node	A	B	C	D	E	F
Dist	0	4	4	7	5	10

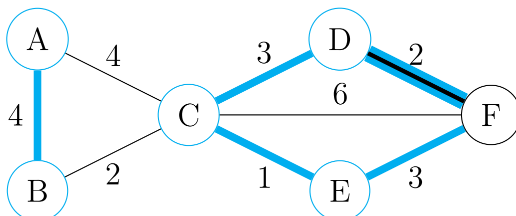
(g) As vertex E in unvisited and has the lowest distance, it is the next node to be considered



(h) $w(E, F) <$ current distance to F so update distance from 10 to 8 $(5 + 3)$.

Node	A	B	C	D	E	F
Dist	0	4	4	7	5	8

(i) Only unvisited out neighbour from D is F but the distance isn't updated.



(j) No vertices are updated this iteration.

Node	A	B	C	D	E	F
Dist	0	4	4	7	5	8

Figure 6.3: Example of Dijkstra's algorithm

In the digraphs package in GAP, there already exists an implementation of Dijkstra's algorithm. However, without the notion of weighted edges, the algorithm only finds the least number of edges connecting a source to every other vertex. Dijkstra's algorithm will fail on graphs with negative edge weights due to its greedy nature. In Figure 6.4, starting at vertex A, the algorithm, will chose the edge (A, B) as it the least weighted edge. From there it will select the only edge possible to get to D from B. And finally, the next lowest weighted edge (A, C) and thus the algorithm will return 5, 10, 10 for the distances from A to B,C,D respectively but it is clear, that due to the negative edge weighted edge (C, D), that the shortest path from A to D is -990. Therefore, demonstrating that Dijkstra's algorithm doesn't work on graphs with negative edge weights.

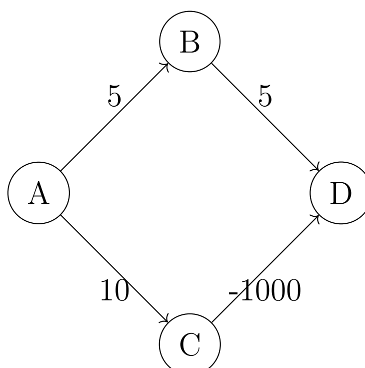
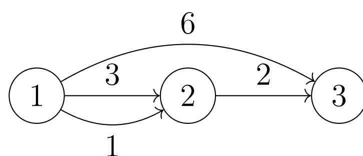


Figure 6.4: Incorrect Dijkstra's Algorithm on negative Edge Weighted Digraph

6.2.1 Implementation

The implementation follows the pseudo code very closely except it is modified to also output which edges are taken in the case where there are parallel edges between two nodes. Therefore Dijkstra's algorithm outputs a record with the follow keys; distances, parents and edges. I will use an example to showcase this, as shown by Figure 6.5, starting from vertex 1. The 'rec' in the caption is short for 'record' which is a type of GAP data structure where the name in the record is an identifier distinguishing this component.

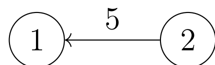


$$rec(distances := [0, 1, 3], edges := [fail, 2, 1], parents := [fail, 1, 2])$$

Figure 6.5: Example of Dijkstra's GAP output

The distances key returns a list of the minimum distance to get to each vertex. The distance to the source (vertex 1) is 0, the distance to vertex 2 is 1 and to 3, it is 3. The edges key returns a list of which edge was taken from the parent. So we will look at these together. In the parents list, index 2 is a 1 which means the parent of

vertex 2 is 1 and the edge taken can be found by looking at the index 2 of the edge list which is 2. The first edge from (A, B) is 3 and the second edge is 1 so the second edge was taken to get to vertex B. Looking at index 3 of the parents list (the last element), it is 2 which means the previous vertex was a 2 and then looking at the edges list at index 3, the value is 1 which means the first edge was taken. This is correct as it is the only edge from vertex 2 to 3. In the case, there are no paths possible, fail will be returned as shown by Figure 6.6 as there is no path from vertex 1 to 2.



$$rec(distances := [0, fail], edges := [fail, fail], parents := [fail, fail])$$

Figure 6.6: Example of Dijkstra's GAP output with no path

6.3 Bellman–Ford Algorithm

This algorithm was developed and proposed by Richard Bellman in his 1958 paper [4], Lester R. Ford Jr in 1956 [20] and Edward F. Moore in 1959 [39]. Therefore, the algorithm is sometimes known as the Bellman–Ford–Moore Algorithm.

Definition 6.1 (negative cycle)**.** A negative cycle is a cycle whose edges are such that the summation of their weights is a negative value. [9].

The Bellman–Ford algorithm serves the same purpose as that of Dijkstra's algorithm, except it can handle negative edge weighted graphs. However, it cannot handle negative edge weighted cycles. The intuition for this is simple as if a graph contained a negative cycle then we can indefinitely decrease our distance by following the cycle repeatedly, reducing our total distance each cycle. The algorithm is also able to detect negative cycles. This makes it a more versatile algorithm algorithm as it is able to tackle negative edge weights, and also detect negative cycles and won't return an incorrect solution under these conditions unlike Dijkstra's algorithm. However, this is at the cost of a higher run time complexity which will be discussed further in the Analysis section. Negative edge weights can be useful in modelling cash flow, entropy, heat or energy loss in reactions and more. The idea behind the Bellman–Ford algorithm is as follows:

1. Initialise all distances for every vertex to ∞ except for the source which is 0.
2. For every edge (u, v), make the distance to $v = \min(\text{dist}[u] + w(u, v), \text{dist}[v])$. For any vertex u, check that we can reach any of its neighbours in a shorter distance than currently possible. This is called relaxation.
3. Check if there are negative cycles.

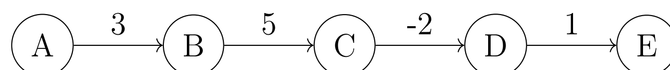
The pseudo-code for Bellman–Ford Algorithm is shown by Algorithm 8

Algorithm 8 Bellman–Ford Algorithm

```
1:  procedure Bellman-Ford(G, source)
2:      for all v ∈ G do
3:          dist_v ← ∞
4:      dist_source ← 0
5:      for all v ∈ V(G) do
6:          for all (u, v) ∈ E(G) do
7:              t ← dist_u + w(u, v)
8:              if t < dist_v then
9:                  dist_v ← t
10:                 prev_v ← u
11:     for all (u, v) ∈ E(G) do
12:         if dist_u + w(u, v) < dist_v then
13:             return Error
14:     return dist, prev
```

The fundamental idea behind Bellman–Ford algorithm is that for any connected graph, the shortest path from any u to v has at most $|V|-1$ edges. Using Figure 6.7, we can see that for 5 vertices, the maximum number of edges between any two vertices is 4 (between A and E). If the path had more than $|V|-1$ edges, then a cycle would have to exist. Therefore, if we iterate $|V|-1$ times, we are guaranteed to find the shortest path. As this algorithm is not greedy, like Dijkstra's Algorithm, all the edges are examined from each vertex and if the distance is shorter than its current distance, it is updated.

Figure 6.7: Shortest Path has $|V|-1$ edges



6.3.1 Implementation

The implementation is very similar to Dijkstra's algorithm in terms of the format of the output. This is intentional as it meant the output was consistent between the algorithms and they both return the same types of information even if they paths returned were different. The Bellman–Ford Algorithm operates much like a dynamic programming problem, more specifically a bottom up solution. A bottom-up dynamic programming (often abbreviated DP) solution aims to build a final solution by breaking the problem into smaller sub problems and solving those first. By solving these sub problems first, it can find the solution to the main problem. A classic example of this is finding the nth member of a Fibonacci series as described below:

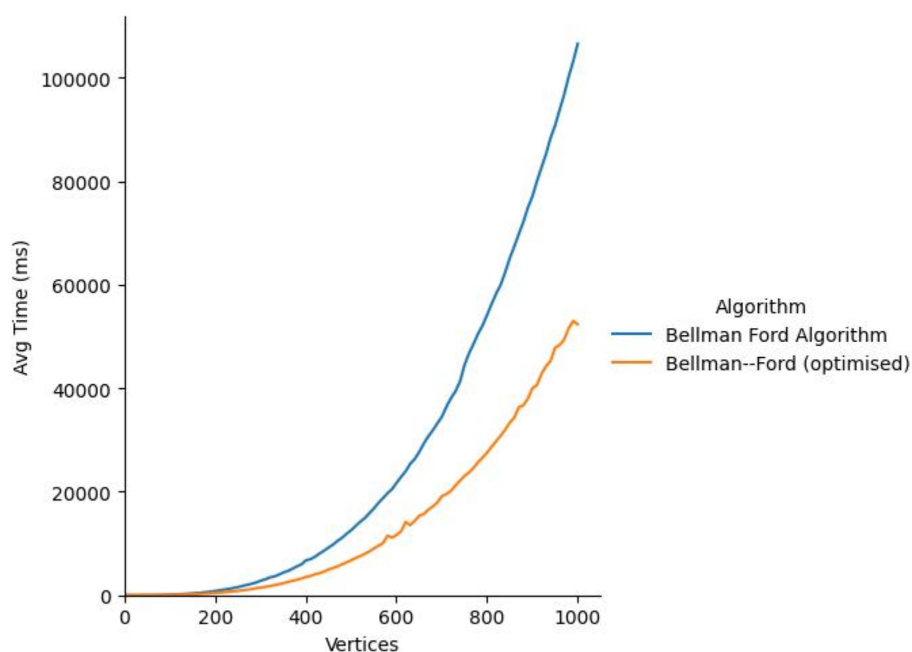
Problem: Find the n^{th}th Fibonacci number.
Input: n (a positive integer)
Output: F_n (the nth Fibonacci number)
Algorithm: Bottom-Up Dynamic Programming

Initialize $F[0]$ to 0 and $F[1]$ to 1
for $i = 2$ to n do
  $F[i] = F[i-1] + F[i-2]$
endfor
Output $F[n]$

In each iteration, it calculates the ith Fibonacci number by summing the $(i-1)^{th}$ and $(i-2)^{th}$ Fibonacci numbers and storing the result in $F[i]$. This is done iteratively, building up the Fibonacci series from the bottom up, by avoiding redundant calculations and reusing previously computed values. This is relevant to Bellman–Ford algorithm as it first calculates the shortest distances for paths that have at most 1 edge; then 2 edges, and so on. After the i_{th} iteration, the shortest paths with at most i edges. As there can be a maximum of $|V|-1$ edges in any simple path, the other loop runs this many times. Therefore, the complexity of this algorithm is $O(|V||E|)$ as for $|V|$ vertices, we iterate $|E|$ times.

There is a small optimisation possible with this algorithm. When we are in the inner loop (see Line 6 of Algorithm 8) checking every edge; if no edges are relaxed then we know that there will be no further changes to the distances and we can break out the outer loop (see Line 5 of Algorithm 8) early. Figure 6.8 shows that the optimised algorithm has a similar trend and complexity as the non optimised version, but executing faster as it doesn't do redundant iterations when there are no more shortest paths to be found. Implementing this is as simple as a boolean flag within the 'if' statement on Line 8 of Algorithm 8 which if not set, we break out after this loop finishes executing.

Figure 6.8: Bellman–Ford with and without optimisation on 1.0 edge probability graph



6.4 Floyd–Warshall Algorithm

The Floyd–Warshall algorithm is another example of dynamic programming and originates from the papers [18] by Robert Floyd and Stephen Warshall [48] in 1962 except this algorithm is a type of All Pairs Shortest Path (APSP) Algorithm in which, as the name suggests, finds the shortest distance between all pairs of vertices.

The pseudo-code for Floyd–Warshall is shown by Algorithm 9.

Algorithm 9 Floyd–Warshall Algorithm

```
1: procedure FLOYD-WARSHALL(G)
2:     D ← |V| × |V| matrix initialised with ∞
3:     fill all D_uv from E(G) with w(u, v)
4:     for all k ∈ V do
5:         for all u ∈ V do
6:             for all v ∈ V do
7:                 D_uv ← min(D_uv, D_uk + D_kv)
       return D
```

6.4.1 Implementation

Like Bellman–Ford, this algorithm can handle negative weighted edges and also detect negative cycles. Let G be a weighted directed graph with V vertices represented by an $|V| \times |V|$ matrix D such that at iteration k, D_{uv} is the shortest path form u to v with nodes, $1, 2, ...k$ [31]. The algorithm uses three nested loops to update the matrix D. Once the algorithm terminates, assuming there are no negative cycles, the shortest distance from u to v is D_{uv}.

The outer loop iterates over all vertices in the graph, while the two inner loops iterate over all pairs of vertices. The algorithm computes the shortest path from vertex u to vertex v through vertex k, and updates the matrix D with the minimum of the current shortest path and the path going through k. If there are negative values in the graph, then special care needs to be taken. As the graph contains ∞, the matrix may end up containing values such as $\infty - 1, \infty - 2$, which is not a valid mathematical calculation indicating that a path between u and v doesn't exist. Therefore, in the most inner loop (see Line 9) when setting the value of D_{uv}, we need to check that D_{uk} and D_{kv} are less than ∞ so we do not change the values for paths that do not exist.

After the algorithm completes, the matrix D will contain the shortest path distances between all pairs of vertices in the graph. The time complexity of the Floyd-Warshall algorithm is $O(|V|^3)$, making it efficient for small to medium-sized graphs.

6.5 Johnson's Algorithm

Johnson's Algorithm is similar to Floyd–Warshall as it is a APSP algorithm. The technique was first published by Donald B. Johnson in his 1977 paper [32]. This algorithm, like the rest, except for Dijkstra's works on negative edge weighted graphs but not on graphs with negative cycles.

The algorithm takes advantage of the idea that if all the edge weights were non-negative, we can find all the shortest paths by using Dijkstra's algorithm. After the transformation to a non negative graph, we can run Dijkstra's algorithm to solve the SSSP problem n times to obtain a APSP solution [31]. The idea is outlined in more detail below:

Johnson's Algorithm Outline [31]

1. Add a vertex s to graph with $w_{sv} = 0$ for all vertices $v \in V$.
2. Use Bellman–Ford algorithm to find the shortest paths from s to every other node. If a negative cost cycle is detected by the algorithm, then terminate at this step. Otherwise, let $distance_v$ be the shortest distance from s to vertex v.
3. Remove s from the graph, and convert the costs in the original graph to non negative costs by setting $w'_{uv} = w_{uv} + h_u - h_v$ where the h is shortest distance.
4. Apply Dijkstra's algorithm n times to graph with updated weights w'.

Algorithm 10 Johnson's Algorithm

```
 1: procedure JOHNSON(G)
 2:     V ← V ∪ {some arbitrary vertex s}
 3:     for all v ∈ V do
 4:         w(s, v) ← 0
        h ← Bellman–Ford(G, s)
 5:     if h has negative cycle then
 6:         return Error
 7:     for all u ∈ V do
 8:         for all v ∈ V do
 9:             w(u, v) ← w(u, v) + h_u + h_v
10:     d ← empty list
11:     for all u ∈ V do
12:         d_u ← Dijkstra(G, u)
13:         for all v ∈ V do
14:             d_uv ← d_uv + h_v − h_u
15:     return d
```

6.5.1 Implementation

This algorithm achieves the same functionality as the Floyd–Warshall algorithm where it finds all shortest paths between every pair of vertices. This algorithm uses Dijkstra's algorithm and Bellman's algorithm so I could reuse my implementations from before exactly without changing any of the implementation as all this algorithm required was the list of distances that the other implementations returned.

This algorithm, like the others, gets the `EdgeWeights` attribute from the digraph passed into the function. However, where the other algorithms just make use of the values, this algorithm modifies the list. Therefore, I needed to add a `EdgeWeightsMutableCopy` operation to return a mutable list. This was done, by Installing a method that returns a shallow copy of the original list.

Following the outline for the algorithm above, we first need to add a new source vertex s and add a path from it to all the other vertices edge weight 0. This is because the graph may not be strongly connected and therefore we can overcome this issue by adding an arbitrary vertex with an edge to every other vertex. We will be using Figure 6.9 to demonstrate this algorithm and as we can see, there is no starting source vertex, from which we can reach the other vertices from. As s is unreachable from any of the other vertices, adding this vertex does not add any paths from u to v $\in G$. Since we have a mutable `EdgeWeights` and `OutNeighbours`, we prepend a 0 to every out neighbour edge weight for s as well as adding an edge to each out neighbour.

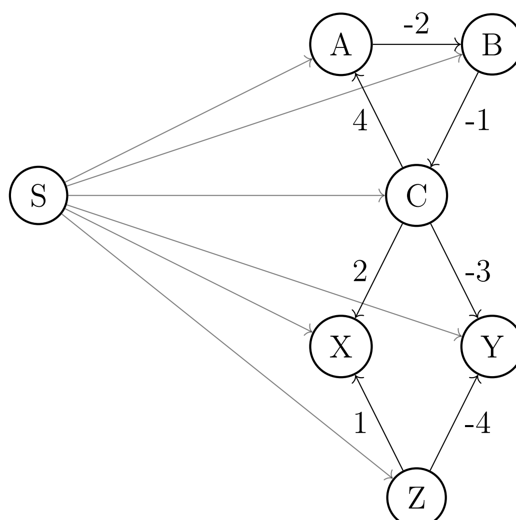


Figure 6.9: Adding source s to Graph for Johnson's Algorithm Example

Now if we compute the shortest distances from s to all the other nodes, we get: *A:0, B:-2, C:-3, X:-1, Y:-6, Z:0.* Now we need to transform the edge weights to become non negative. Using the equation 6.1, which translates to, given the weight of the edge, we add the distance to the head and subtract the distance to the tail.

$$w'_{uv} = w_{uv} + h_u - h_v \tag{6.1}$$

Therefore, for vertex A, the weight of the edge (u, v) is -2 added to the distance to the head A which is 0 subtract the distance to the tail B which is -2 gives us 0 for the edge (A, B). Computing this for all the edges gives us Figure 6.10. Once the edge weights are re weighted, the shortest paths are still preserved as all the edges were re weighted by the same amount, $w_{uv} + h_u - h_v$. Now that the weights are non negative, we can run Dijkstra's algorithm which is faster than the Bellman–Ford algorithm. This will return the incorrect distances so once we have the shortest paths, we can correct the distances by running the reverse of Equation 6.1, as shown by Equation 6.2 where $h(u)$ and $h(v)$ are swapped to return to the original distances.

$$w'_{uv} = w_{uv} + h_v - h_u \tag{6.2}$$

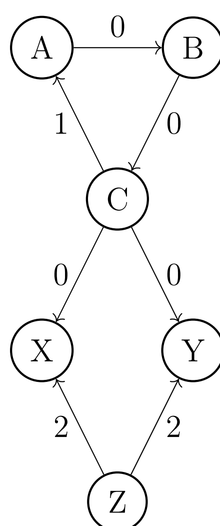


Figure 6.10: Transforming edge weights to become non negative

6.6 Analysis

In this section, it makes sense to compare the two separate groups of algorithms together as they both serve a different purpose and will have different run time complexities. Similarly to the minimum spanning tree problem, all the other plots for the individual graphs can be found in Appendix D.

6.6.1 Single Source Shortest Path Algorithm Comparison

The first two algorithms we compare are Dijkstra's and Bellman–Ford. As with the minimum spanning tree section, I will show the overall comparison for different density of graphs (0.01, 0.125, 0.25, 0.5, 1.0) and then compare individual graphs with specific densities. First we have Figure 6.11 and Figure 6.12 which shows the unoptimised and optimised version of Bellman–Ford against Dijkstra's algorithm. The performance increase is hard to see but if we look at the peak of each graph, we can see at 1000 vertices, the optimised algorithm takes around 20 seconds, compared to that of th unoptimised version which takes just over 30 seconds.

The difference between the three algorithms will be more easily noticed in Figure 6.13. Dijkstra's algorithm is clear faster but as aforementioned, the compromise arises from the Bellman–Ford algorithm being able to handle negative edge weights, where Dijkstra's cannot. The difference in all 3 all algorithms is more easily visible in this graph. A similar trend appears when comparing the algorithms for individual edge weight probabilities, rather than all of them at once.

The complexity for Dijkstra's algorithm is $O(|E| \log |V|)$ when using an adjacency list and binary heap as we do. Using an adjacency list, means we can traverse the edges for each vertex so the complexity for this is $O(|E|)$. Iterating the queue has complexity of $O(|V|)$ as we pop one vertex each iteration. Using the binary heap allows us to extract the minimum edge in $O(\log V)$ time. Therefore, the calculation to achieve the described complexity is shown below by Equation 6.3.

$$\begin{gathered} O(|E| \log |V|) + O(|V| \log |V|) \\ O((|E| + |V|) \log |V|) \\ \text{as } |E| \geq |V| \text{, complexity becomes} \\ O(|E| \log |V|) \end{gathered} \tag{6.3}$$

As for Bellman–Ford, the algorithm starts by initializing the shortest distances from the source vertex to all other vertices to infinity, except for the source vertex itself, which is set to 0. Then, it iteratively relaxes the edges of the graph $|V| - 1$ times.

During each iteration, the algorithm examines each edge in the graph and updates the distance to the neighbouring vertices if a shorter path is found. This process is repeated for $|V| - 1$ iterations, as it is guaranteed that the shortest path in a graph with $|V|$ vertices can have at most $|V| - 1$ edges. After the $|V| - 1$ iterations, the algorithm has found the shortest paths from the source vertex to all other vertices, assuming there are no negative cycles in the graph. Therefore we can calculate the complexity as follows

1. Iterating through vertices: The algorithm iterates through all $|V|$ vertices in the graph to initialize the distances, and then performs $|V|$ iterations to relax the edges. This takes $O(|V|)$ time.

2. Relaxing edges: During each iteration, the algorithm may relax the distances of all $|E|$ edges in the graph. This takes $O(|E|)$ time.

Therefore, the overall complexity is $O(|V||E|)$ as each vertex is examined in $O(|V|)$ iterations and in each iteration, $|E|$ edges may be relaxed.

6.6.2 All Pairs Shortest Path Algorithm Comparison

This is the next group of algorithms that we will analyse which includes Floyd–Warshall algorithm and Johnson's algorithm. Before analysing these, I will calculate the complexity first. The complexity for Floyd–Warshall is trivial. The algorithm consists of three nested loops, iterating through $|V|$ vertices each time and therefore the complexity is $O(|V|^3)$. This is faster than, running Bellman–Ford algorithm which

Figure 6.11: Overall comparison of Bellman–Ford vs Dijkstra's Algorithm

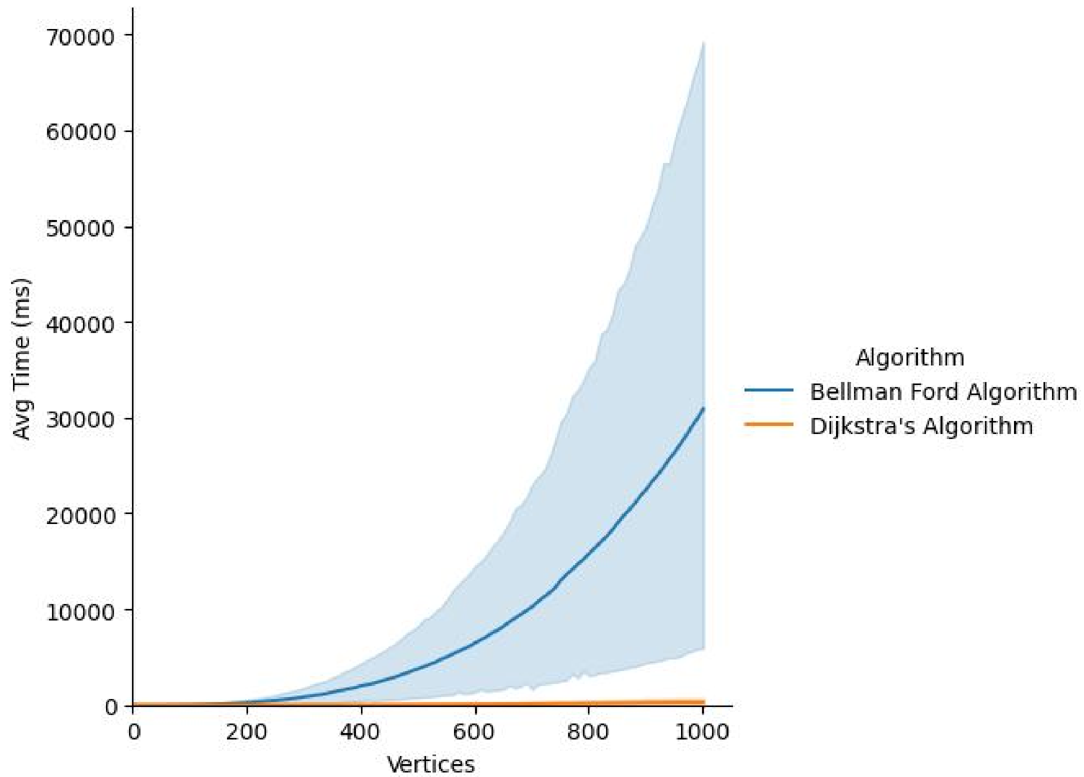
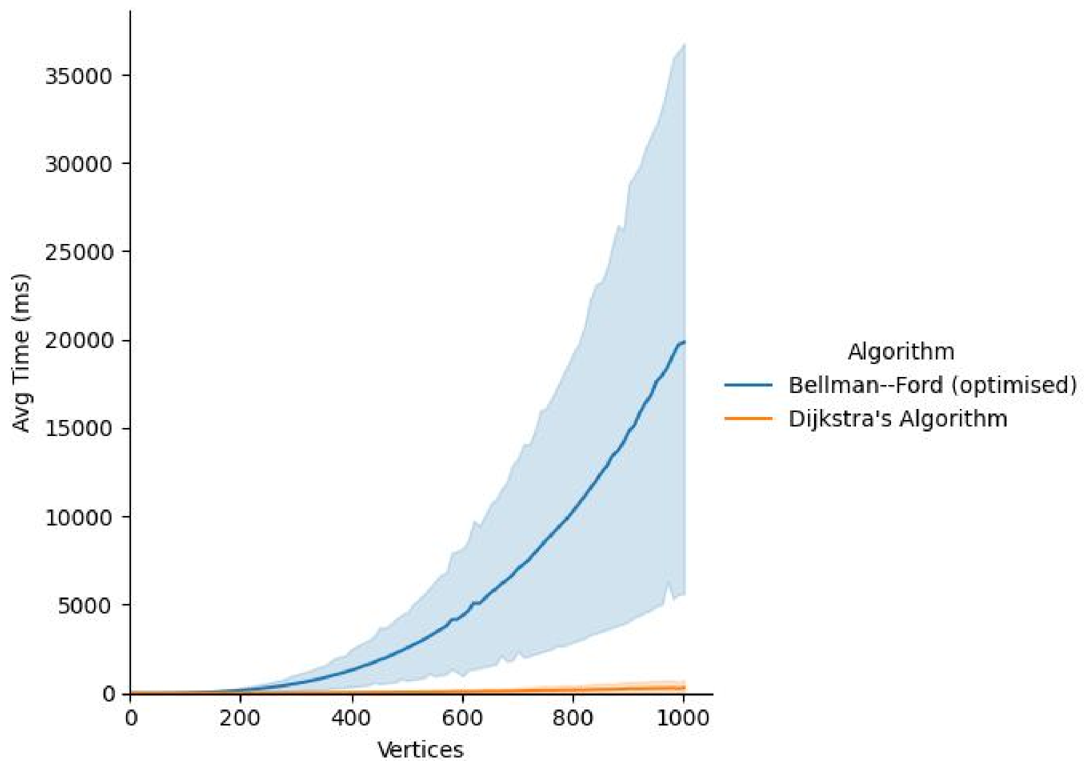


Figure 6.12: Overall comparison of Bellman–Ford (optimised) vs Dijkstra's Algorithm



we would need to execute $|V|$ times which would result in a complexity of $O(|V|^2 \times |E|$ and as $|E| \geq |V|$ for a connected digraph, this would be much slower. The other algorithm, Johnson's algorithm implemented has a complexity of $|E|\log|V| + |V||E|$. The calculation for this is shown below by Equation 6.4. As the complexity for the Floyd–Warshall algorithm is $O(|V|^3)$ and therefore not dependent on the number of edges we can expect the plot of run times to be similar regardless of density. This is indeed found to be the case as shown by Figure 6.14 which shows a similar run time curve for each probability density except for the sparsest graph. However this is still close to the other curves and differs marginally. This is also the only graph algorithm that isn't dependent on the number of edges so this result was interesting to see and further backs up the calculation of the time complexity.

$$\begin{gathered} O(|V|)\text{to add a new edge to all the vertices in the graph} \\ O(|V||E|)\text{for running Bellman–Ford re weighting} \\ O(|E|\log|V|)\text{for running Dijkstra's algorithm} \\ \text{so final complexity is} O(|V||E|\log|V|) \end{gathered} \tag{6.4}$$

This makes for an interesting comparison between Floyd–Warshall and Johnson's algorithm as in the complexity analysis for Floyd–Warshall, we can see that it is not dependent on E so this would suggest that as the graphs become more dense (graphs with higher edge weight probability), Floyd–Warshall should run faster and thus for sparser graphs, Johnson's algorithm will be more performant. From my analysis, I found this to be true. First, from the overall graph with plots for each tested edge weight probability as shown by Figure 6.15, we can see that this shows Floyd–Warshall to be dominant across all density of graphs or at least equivalent when run on lower vertices (about 200 vertices). However, if we run the algorithms for the lowest and highest edge weight probabilities (0.01 vs 1.0) to compare the different extremes of graph density, we can see some interesting results. Figure 6.16 shows the result we expect where for higher number of V, Floyd–Warshall increases at a much lower rate to that of Johnson's algorithm. However, if we compare the algorithms on sparser graphs as shown by Figure 6.17, we can see that Johnson's algorithm does indeed run faster. Therefore we know that, the algorithms are accurate to the theory but we still need to find the threshold for when Floyd–Warshall algorithm bypasses Johnson's algorithm.

Running the algorithms on a graph with 0.125 edge weight probability (see Figure 6.18) we can see that for 0.125 edge probability, the performance of the graphs are similar. This gives a good indication of the density threshold for when Floyd–Warshall overtakes Johnson in performance and therefore when implementing a function to find APSP's in GAP we can do some checks to select the best algorithm. This is achieved by calculating the maximum number of edges for the number of V in the graph which is calculated as $|V| \times (|V| - 1)$. Therefore given the maximum possible edges possible (assuming no parallel edges), if the $|E|$ is less than an $1/8^{th}$ of the maximum possible edges, we run Johnson's algorithm, otherwise Floyd–Warshall. This is due to 0.125 edge probability being equivalent to 1/8.

6.6.3 Comparison with Scipy's Implementation

The Scipy.sparse.csgraph [10] also contains implementations for all 4 of the shortest path algorithms that I implemented. These graphs were created, and computed using Scipy in a similar way to the Minimum Spanning Tree section and similar results were obtained. My implementations for each and every algorithm in GAP were severely out performed by Scipy's implementation in every regard. I will show the graphs for the performance difference for each of the algorithms on all edge probability values. The trends in performance are similar for the individual density of graphs. Figures 6.19, 6.20, 6.21, 6.22 shows the comparison between each algorithm and Scipy in the order that I discuss them in this report.

I believe the significant difference in performance to be due two reasons. Firstly, Scipy uses Cython, which is a 'superset of Python that compiles to C' [49] which makes the performance blisteringly fast. Code written in Cython have a `.pyx` file extension [36]. Looking at the Scipy.sparse.csgraph source code [44], I can see that the Scipy implementations for minimum spanning trees, shortest path and maximal flow algorithms all have a `.pyx` extension and use Cython in the source code. The second reason is closer to an assumption than fact but as the source code is public, the implementations may have been optimised to a very granular level with multiple people working on it to make it as optimal and fast as possible. The disparity of my own and Scipy's implementations may be explained by these two reasons.

In this chapter, I describe four shortest path algorithms; two of which belong to the single source shortest path algorithms and the other two within the all pair shortest path algorithms. I compare the two algorithms in each group, describing their properties and use cases. Afterwards, I analyse the algorithms and plot the run times to be able to compare them with each other.

Figure 6.13: Overall comparison of Bellman–Ford (unoptimised and optimised) vs Dijkstra's Algorithm

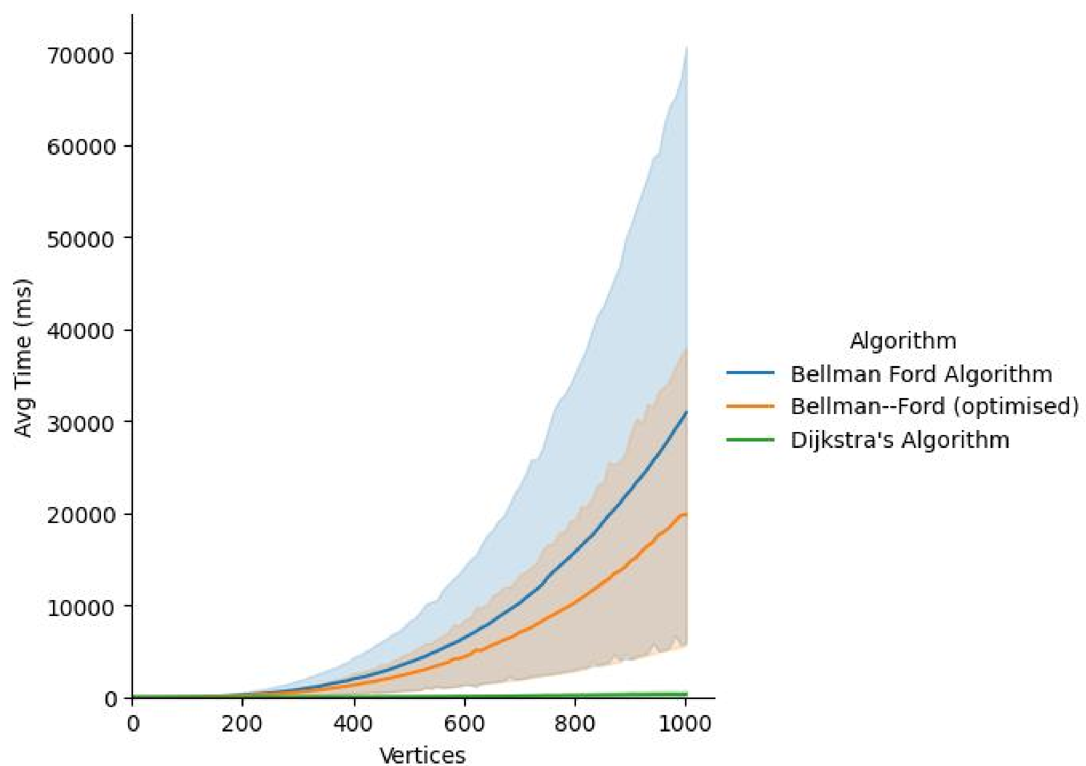


Figure 6.14: Floyd–Warshall algorithm on various graph densities

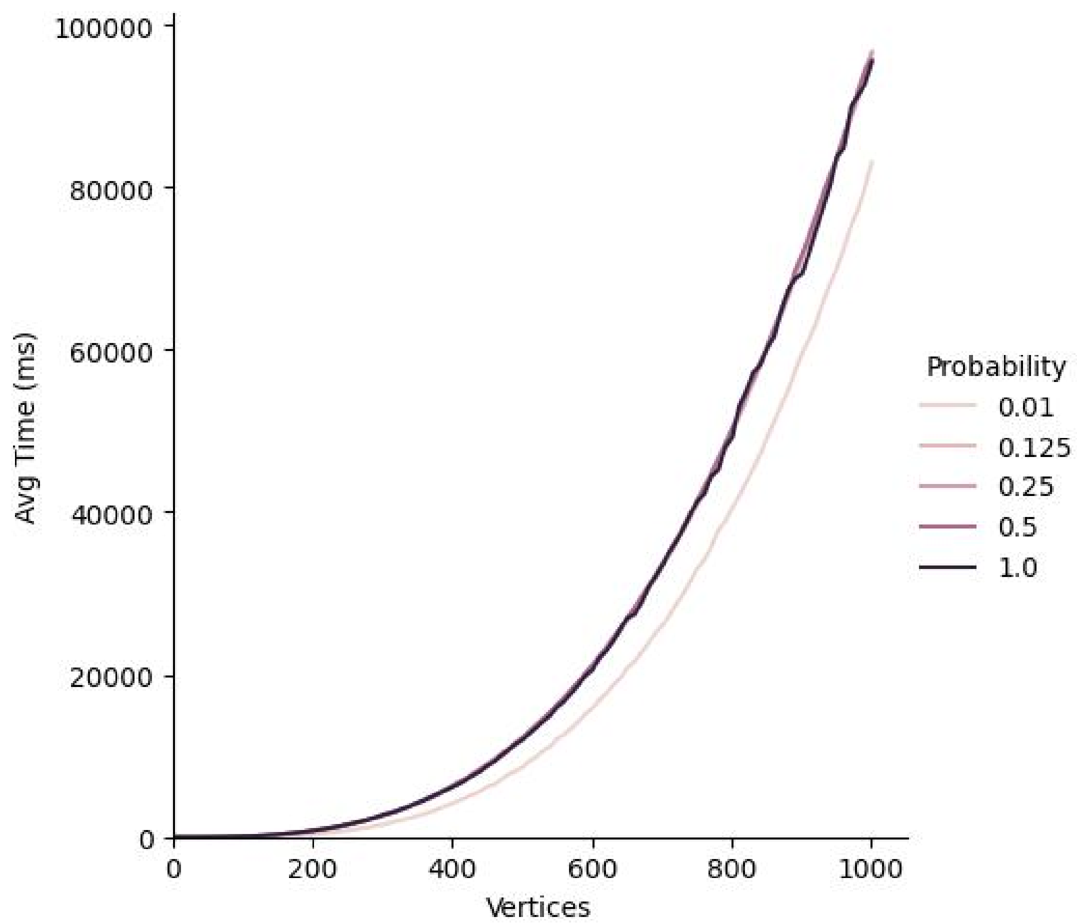


Figure 6.15: Overall comparison of Johnson's vs Floyd–Warshall algorithm

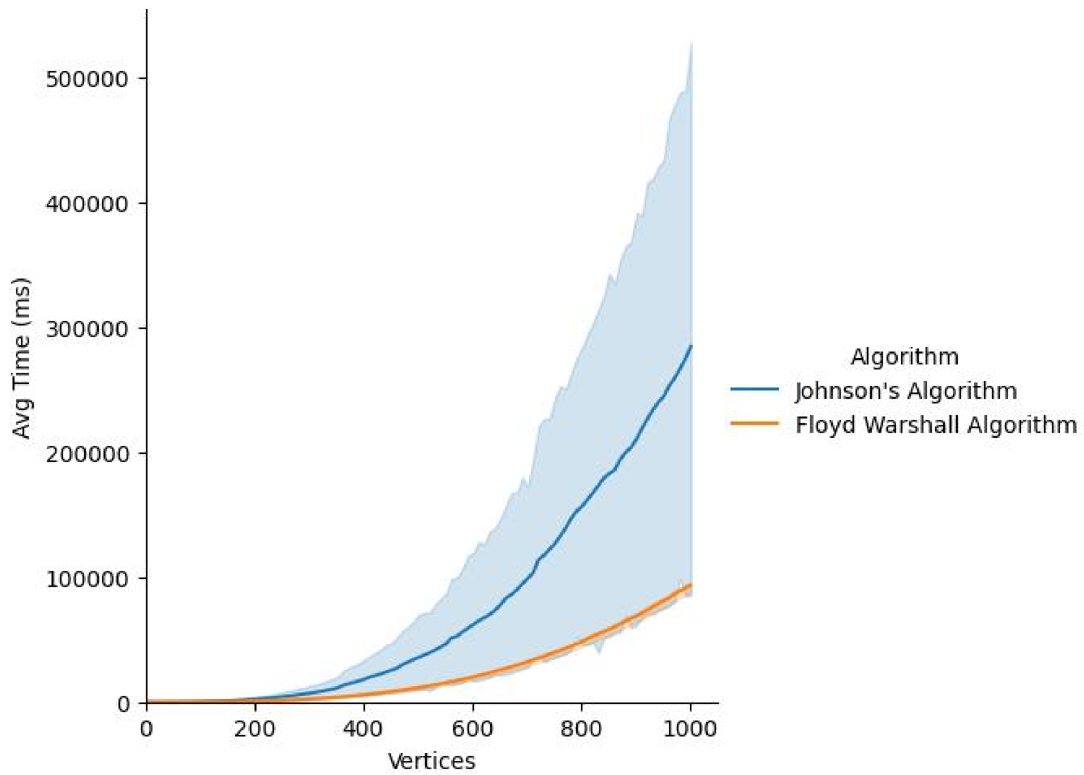


Figure 6.16: Johnson's vs Floyd–Warshall algorithm on Complete Graph (1.0 edge probability)

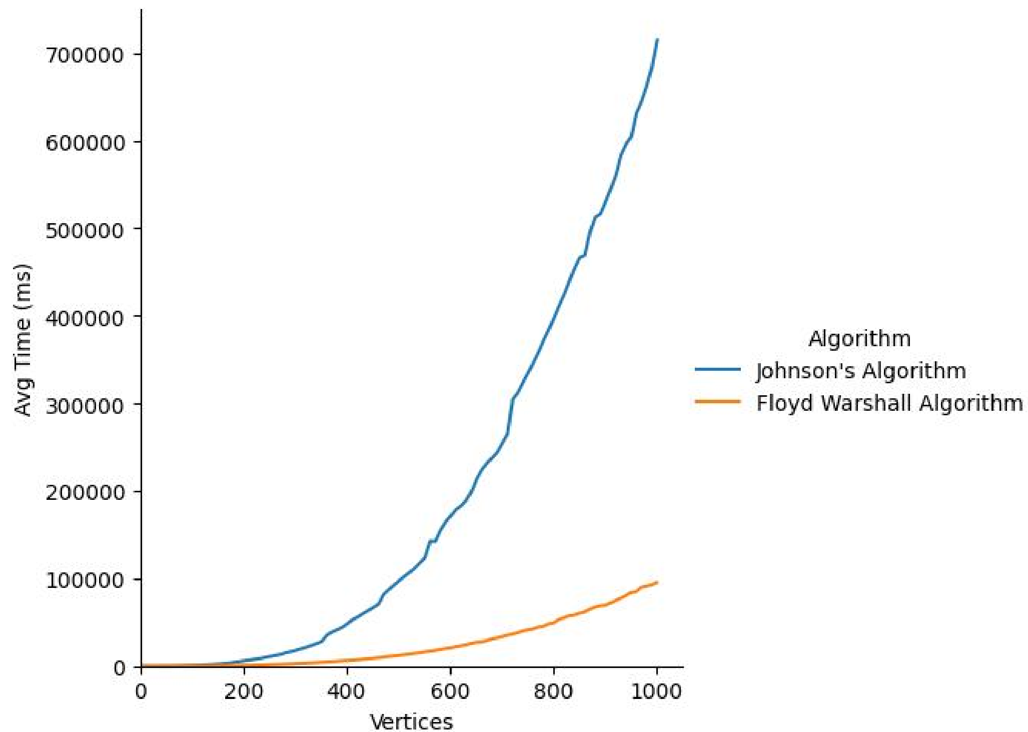


Figure 6.17: Johnson's vs Floyd–Warshall algorithm on sparse graph (0.01 edge probability)

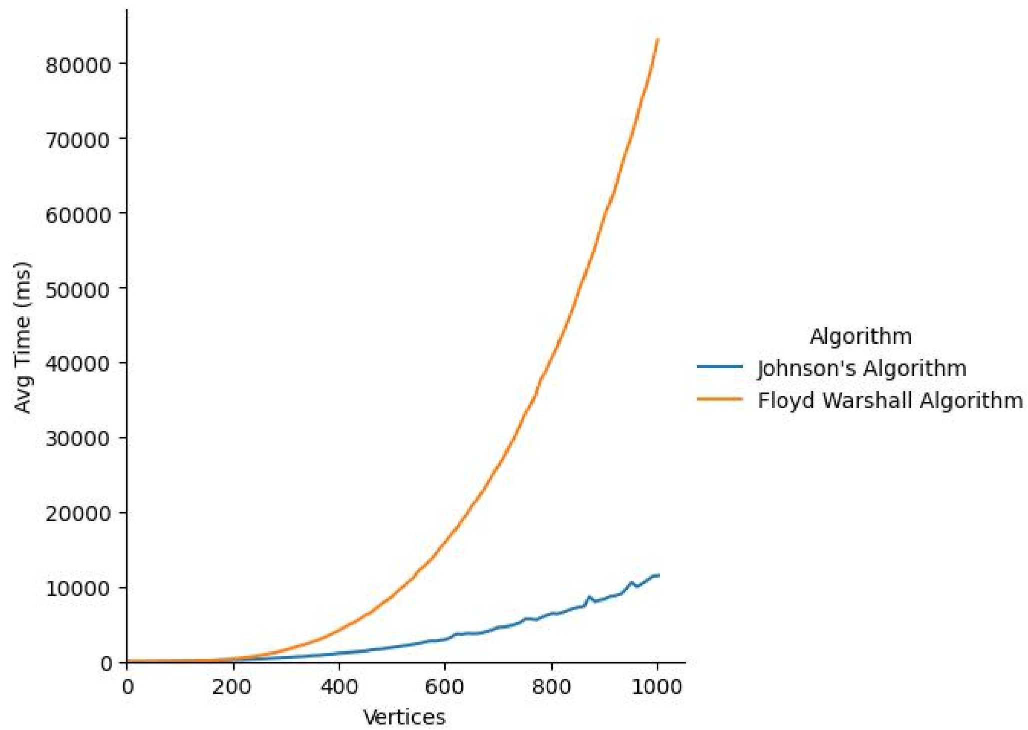


Figure 6.18: Johnson's vs Floyd–Warshall algorithm threshold probability (0.125 edge probability)

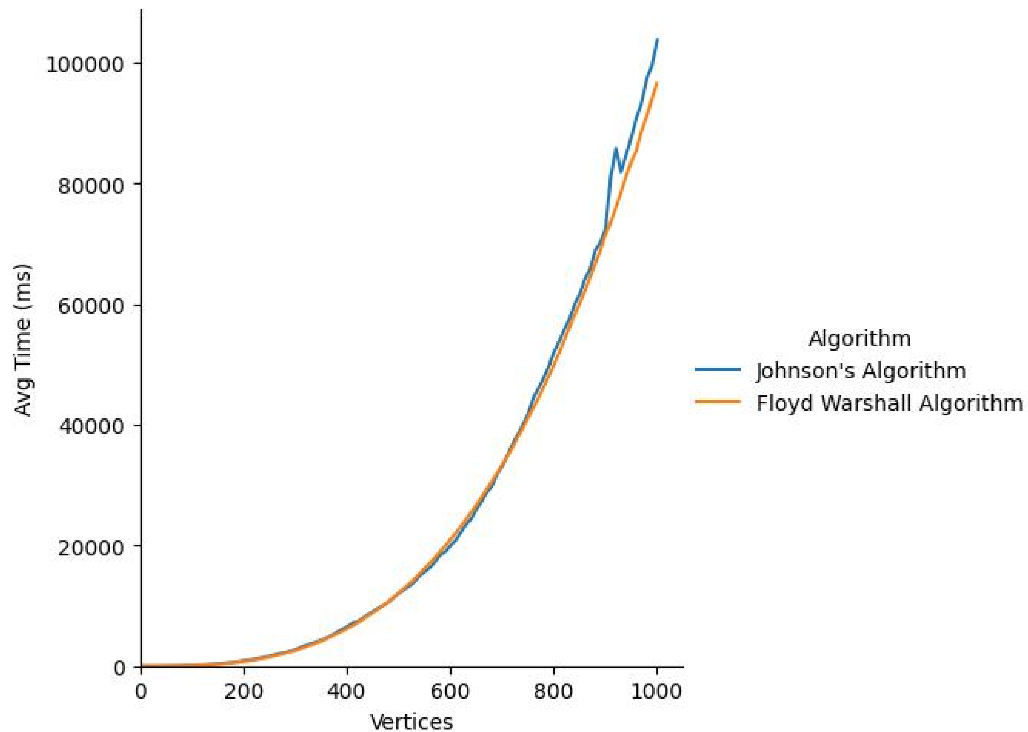


Figure 6.19: Dijkstra's Algorithm (GAP vs Scipy)

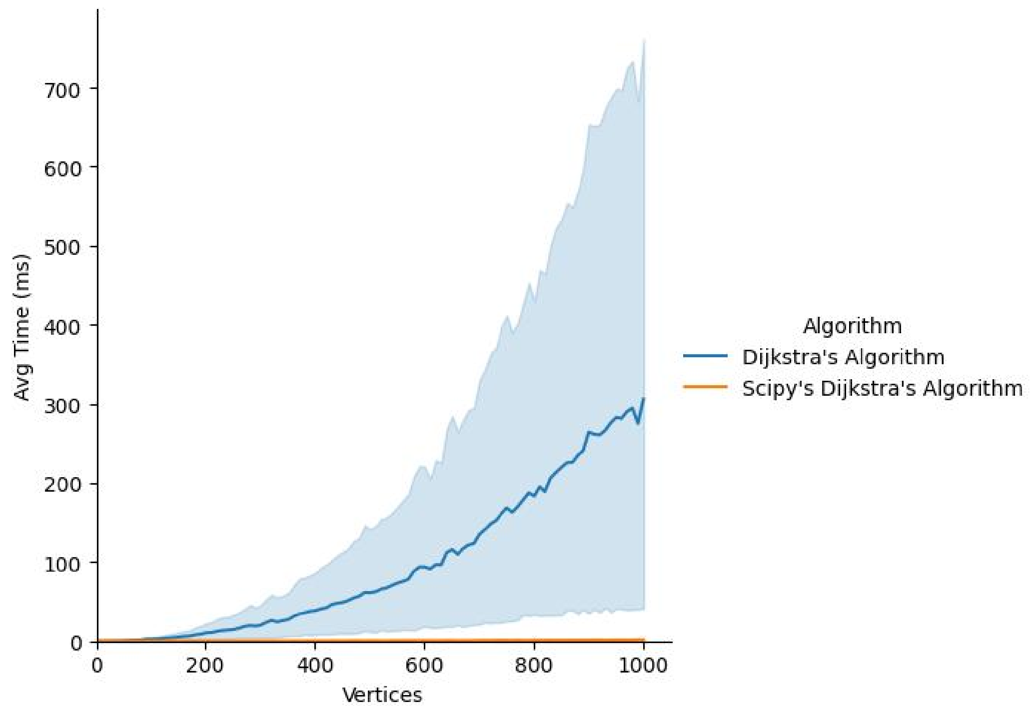


Figure 6.20: Bellman Ford Algorithm (GAP vs Scipy)

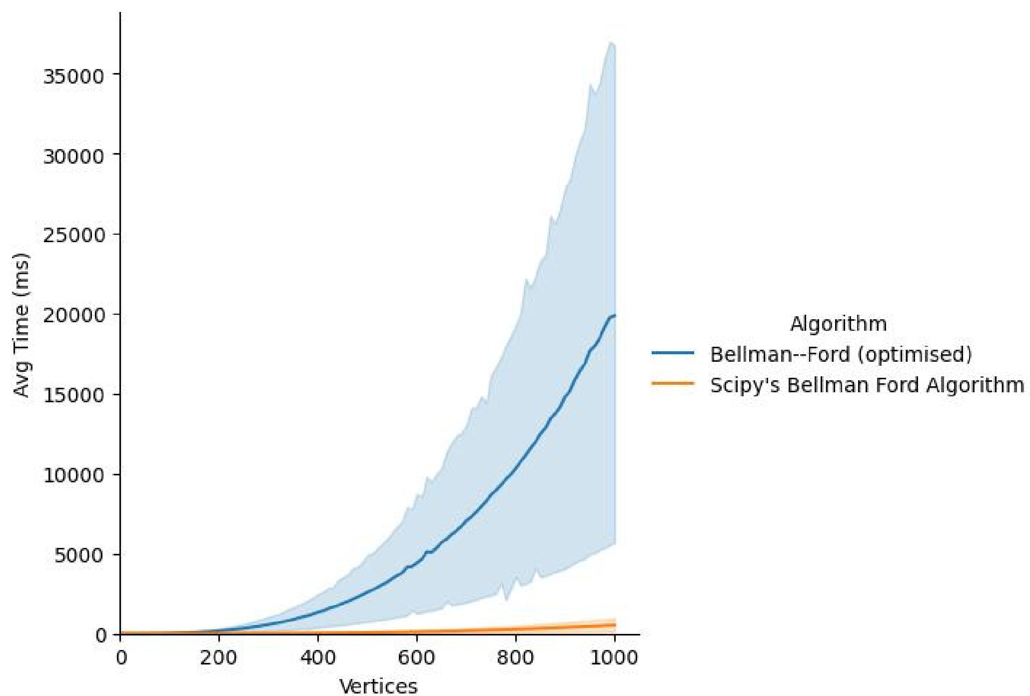


Figure 6.21: Floyd–Warshall (GAP vs Scipy)

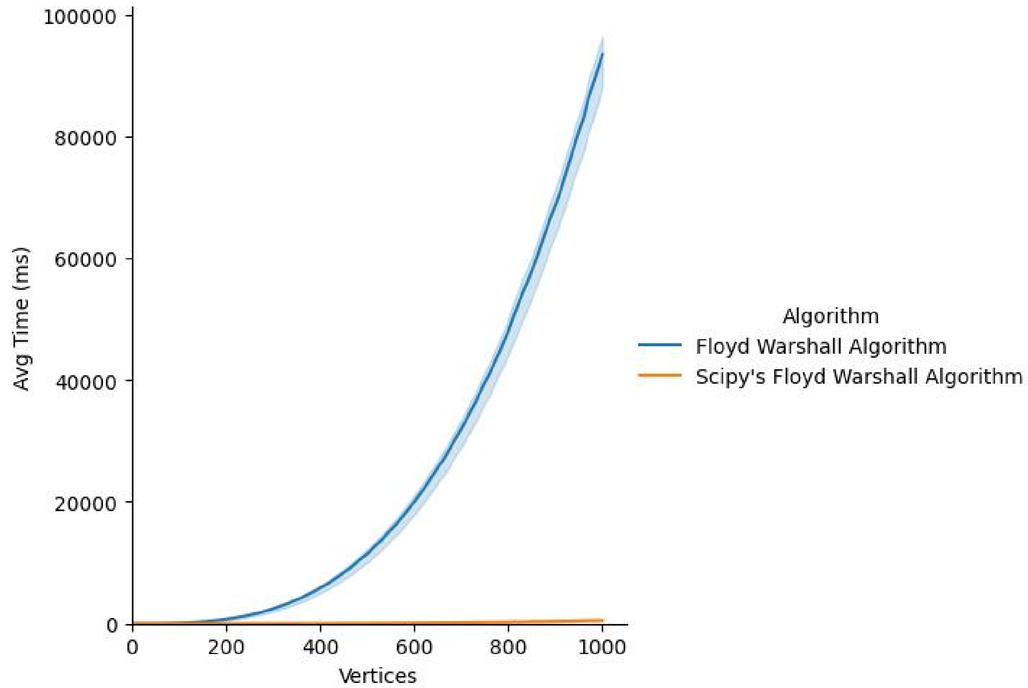
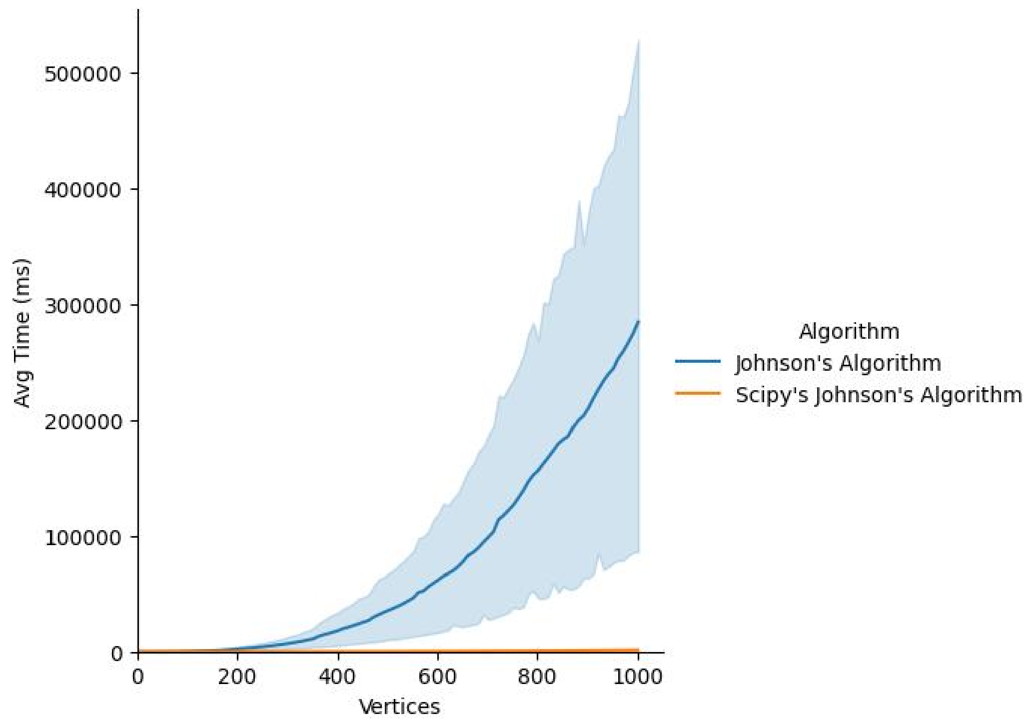


Figure 6.22: Johnson's Algorithm (GAP vs Scipy)



Chapter 7

Maximal Flow Algorithms

This chapter focuses on two maximal flow algorithms on edge weighted digraphs; Edmonds–Karp and Dinic's Algorithm, as well a related minimum cut algorithm called Karger's algorithm but this works as expected on non edge weighted graphs. There is a max-flow-min-cut theory which I will also cover. Afterwards I will perform some analysis on the algorithms to compare them. Maximum flow, specifically can represent many things such as data through a network, water through pipes, traffic on roads so it is a very useful domain of algorithms to explore.

Definition 7.1 (flow network)**.** A **flow network** is a directed graph $G = (V, E)$ with a source $s \in V$, a sink $t \in V$, and capacities along each edge (described by a function $c : E \to \mathbb{R}$ where $c(u, v)$ is the capacity of edge (u, v)) [16].

The amount of **flow** between two vertices is described by a function $f : V \times V \to \mathbb{R}$ [16]. Effectively, it is the transfer of some quantity from u to v in the direction $u \to v$ which has a finite capacity. The function f has the following properties:

Maximal Flow Conditions [16]

1. **Capacity**: $f(u, v) \leq c(u, v)\ \forall u, v \in V$
2. **Anti-symmetry**: $f(u, v) = -f(v, u)\ \forall u, v \in V$
3. **Conservation**: $\sum_{w \in V} f(u, v) = 0\ \forall u \in V - \{s, t\}$

Both the algorithms I explore build upon the Ford–Fulkerson algorithm which is also sometimes referred to as a method as there is no specific implementation details. This method was discovered by Lester R. Ford Jr and Delbert R. Fulkerson in their 1956 paper [19]. This is the same L. R. Ford Jr who developed the Bellman–Ford algorithm in the same year.

The algorithms described are both constructive so a path can be found from source to sink from the output of the algorithms, as well as the total maximum flow value.

7.1 Ford–Fulkerson Method

Before looking at this method, there are some definitions and terminology that may be useful to define as described below:

Definition 7.2 (residual capacity)**. Residual capacity** is the total capacity of any edge (u, v). The residual capacity c_f can be defined as $c_f(u, v) = c(u, v) - f(u, v)$. If there is a flow from a directed edge $u \to v$, then the reversed edge has a capacity of 0 and we can say $f(u, v) = -f(v, u)$.

Definition 7.3 (residual graph)**.** Given graph $G = (V, E)$, a **residual graph** is equivalent to G except we use the residual capacities as capacities.

Definition 7.4 (augmenting path)**.** An **augmenting path** is defined as a simple path from source to sink in a residual graph along the edges whose capacity is 0.

The Ford–Fulkerson method is outlined below:

- Initialise the flow of each edge to 0.
- Check if an augmenting path exists between the source and sink vertices. If a path exists, increase the flow along those edges.
- Repeat this process until there are no more augmenting paths. This means that maximal flow is achieved.

This can be written as pseudo-code as follows (see Algorithm 11) in which the flow augmenting is left intentionally vague.

Algorithm 11 Simple-Ford–Fulkerson Method [21]

```
1: procedure SIMPLE-FORD–FULKERSON(G, source, sink)
2:     flow ← 0
3:     path ← arbitrary augmenting path in G from source to sink
4:     while path exists do
5:         augment the flow along the path
6:         G′ ← residual graph of G
7:         path ← augmenting path in G′ from source to sink
8:     return flow
```

The algorithm in more detail is as follows (see Algorithm 12).

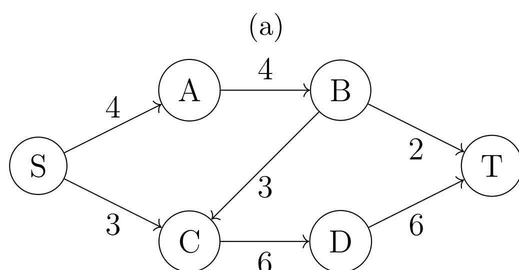
Edmonds–Karp and Dinic's Algorithm are different implementations of Ford–Fulkerson method as certain techniques such as getting the augmenting path or the minimum residual capacity aren't defined specifically. Figure 7.1 shows a visual example of this method where S and T represent the source and sink vertices respectively.

In these examples, the source has no in neighbours and the sink has no out neighbours and in my implementations of the algorithms, if a source is provided which has in neighbours, the in bound flows are ignored and starts at 0 as usual and when the sink has out neighbours, the outbound flows are not considered either.

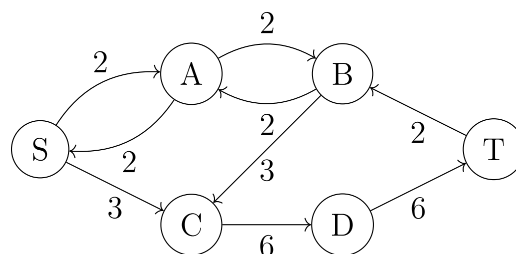
Algorithm 12 Ford–Fulkerson Method

1: **procedure** FORD–FULKERSON(G, source, sink)
2:     **for all** $(u, v) \in G$ **do**
3:         $F_{uv} \leftarrow 0$
4:     $f \leftarrow 0$
5:     path $\leftarrow$ augmenting path in G from source to sink
6:     **while** path exists **do**
7:         min-flow $\leftarrow$ minimum residual capacity in path
8:         $f \leftarrow f$ + min-flow
9:         **for all** $(u, v) \in$ path **do**
10:             $F_{uv} \leftarrow F_{uv}$ + min-flow
11:             $F_{uv} \leftarrow F_{uv}$ - min-flow
12:     **return** flow

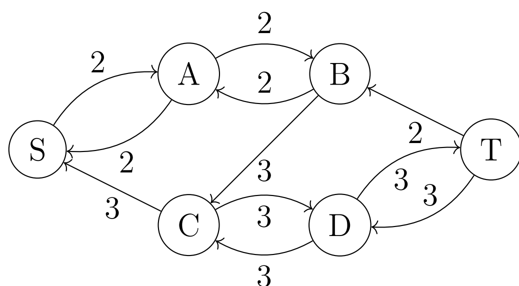
(a)



(b) Taking the path $S \to A \to B \to T$, the minimum flow along this path is 2 from (B, T) so we add a flow of 2 in the reverse direction of the path so the residual graph is as follows.



(c) We can no longer take the path from Figure 7.1b as we cannot traverse (B, T). Therefore, we now search for a different augmenting path giving us $S \to C \to D \to T$ The minimum flow is 3 (S, C) so the residual graph is as follow.



(d) Only one augmenting path remains, $S \to A \to B \to C \to D \to T$. The minimum flow possible is 2 $(A, B)or(S, A)$. Updating the flow along this augmenting path gives us the following residual graph. The total flow is therefore 7 (total flow into S)

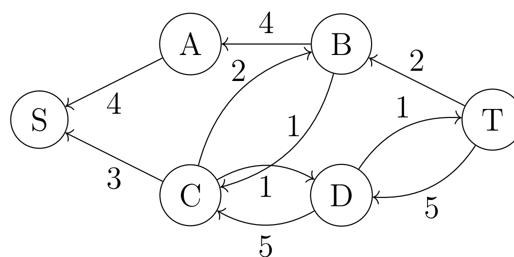


Figure 7.1: Example of Ford–Fulkerson Method

The complexity for Ford–Fulkerson algorithm is $O(|E||f*|)$ where $f*$ is an integer representing the maximum flow. If all flows are integers, the while loop is executed at most $|f*|$ times as in the worst case, the flow is increased by 1 in each iteration. Finding the augmenting path takes $O(|V|+|E|)$ time but as $|E| \geq |V|$ for a connected digraph, this can be simplified to $O(|E|)$ giving us the overall complexity $O(E|f*|)$.

7.2 Edmonds–Karp Algorithm

This algorithm was developed and published by Jack R. Edmonds and Richard M. Karp in their 1972 paper [17] . This, unlike the Ford–Fulkerson method, has a fully specified details of the algorithm. The difference is this algorithm specifies the use of breadth-first Search (BFS) (see Algorithm 13) to find the augmenting paths and therefore the search order of the augmenting paths is well defined.

7.2.1 Implementation

The implementation has two hash maps; one for the graph and storing the capacity and another for the residual flow. We could have used one hash map here but I thought the implementation is simpler to understand this way, although it does use more memory. There are three helper functions in the implementation. The first one is a function to carry out breadth-first search to find an augmenting path. The pseudo code for this helper function is described below by Algorithm 13.

Algorithm 13 Breadth–First–Search

1: **procedure** BREADTH–FIRST–SEARCH(G, source, sink)
2: paths[source] ← empty list
3: **if** source is sink **then return** paths
4: Q ← $\{source\}$
5: **while** Q $\neq \varnothing$ **do**
6: u ← least recently inserted element in Q
7: $Q \leftarrow Q \setminus \{u\}$
8: **for all** (u, v) in neighbours of u **do**
9: paths ← paths + v
10: **if** $c_{(u,v)} - f_{(u,v)} > 0$ and $(u, v) \notin paths$ **then**
11: **if** v is sink **then return** paths
12: $Q \leftarrow Q \cup \{v\}$
return no path found

The second helper function is a function to get the minimum flow from the path as this is the maximum flow we can push through the entire path. The third and final helper function loops through the updated and final flow matrix to find the total flow as well as the flows and parents. By flows, this is the flow inbound into vertex v which is a list of lists. If there are multiple edges, the algorithm will fill up the edges sequentially so if there are 3 edges outbound from u to v with capacities 5,10,15 and there is a flow of 15, it will fill the first two edges with capacities 5 and 10. If there is a flow of 9, then the flow will contain a list with flows of 5 and 4 (see Figure 7.2). This can be coupled with the second component, the parents, which is a list of list of the vertices that each flow comes from. Using this, allows the path of the flow and the flow to be obtained using the flows.

This is shown by the two examples below in Figure 7.2. In Figure 7.2a, the maximum flow is 15 from vertex 1 to 3. In the flows list, the list at index 1 is empty as there is no flow into the source node. The flow at index 2 is [15] as vertex 1 pushes

15 units of flow into 2. Looking at index 1 for the parents list, we can see a list containing 1 as we get a flow of 15 and the parent for vertex 2 is 1. The final flow element is [5,10,0] and the parents are [2,2,2] as we push flow of 5 and 15 from its parent vertex 2 and the first two edges (with capacities 5 and 10 are filled first). In Figure 7.2b, the only difference is the flow outbound from vertex 1 which is now 9 instead of 15 and we can see the difference in the last element of the flows list which is now [5,4,0] as we fill the flow for the first edge and can only push 4 into the second edge which has a capacity of 10.

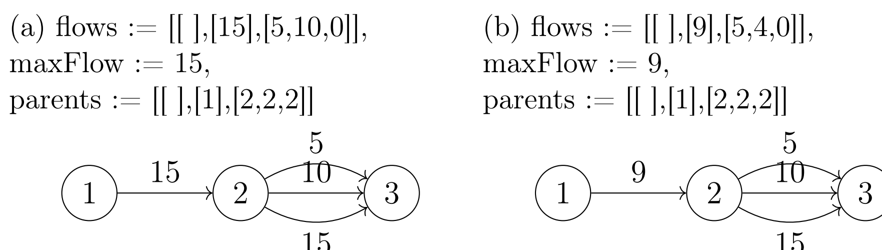


Figure 7.2: Example of output for Maximum Flow Algorithms

This algorithm improves upon the Ford–Fulkerson method which has a complexity of $O(E|f*|)$. Each iteration in this algorithm takes $O(|E|)$ time. Edmond–Karp removes the dependency on the maximal flow making it better for graphs with a large maxium flow such as the one in Figure 7.3. As Edmonds–Karp algorithm runs each iteration in $O(|E|)$ time and there are at most $|V||E|$ iterations, the overall time complexity for Edmonds–Karp is $O(|V||E|^2)$.

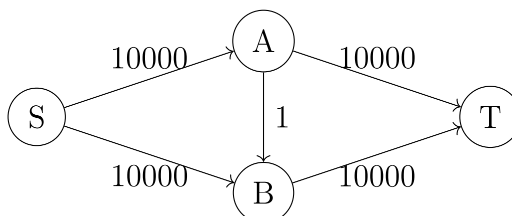


Figure 7.3: Example Graph when Ford–Fulkerson is slow

7.3 Dinic's Algorithm

Dinic's algorithm (also referred to as Dinitz's algorithm) was developed by Yefim Dinitz in 1969 and published in his 1970 paper [14]. The algorithm was later modified slightly and popularized by Shimon Even [43] who mispronounced the name as 'Dinic's Algorithm'.

This intuition behind this algorithm is best described using this analogy. Person A (the source) would like to go to coffee shop C (the sink) which is 'somewhere' east of person A's current position. We would like to make positive progress towards C so it would be preferred to travel in the directions North-East, East or South-East as travelling in one of the other directions would take us further away from the sink. This form of heuristic ensures we continuously make progress. We can apply this to the maximum flow problem by guiding augmenting paths via a level graph. A level graph is a graph is obtained by doing a BFS from the source vertex where the level is the least number of edges connecting the source to a vertex v. The graph shown by Figure 7.4 shows a level graph where the $L = x$ above shows the level of the vertex from source S. An edge is only part of the level graph, if the edge goes from a vertex at level l to a vertex at level $l + 1$. With this condition, the new level graph is shown by Figure 7.5 where the reverse edges (v, u) are removed and edges (D, E) and (H, I) are removed as these edges went from an vertex to another one with a same or lower level.

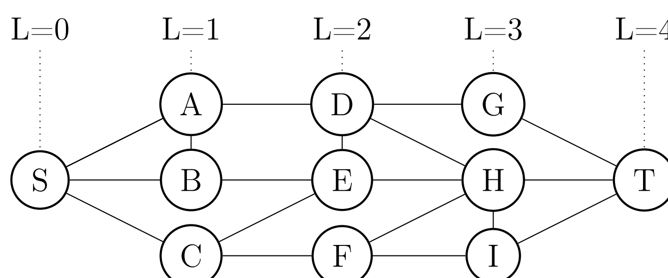


Figure 7.4: Undirected Level Graph

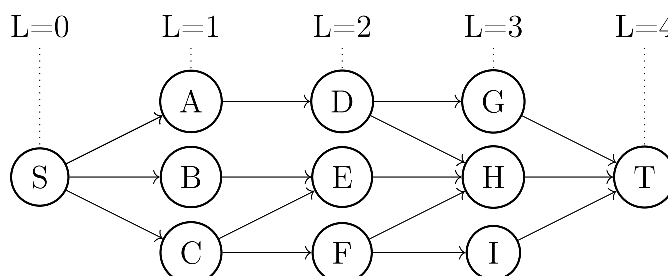


Figure 7.5: Level graph with only relevant edges permitted

The steps for Dinic's algorithm are as follows:

Dinic's Algorithm Steps

1. Construct the level graph using a breadth-first search from the source to every other vertex.

2. If the sink was never reached whilst building the level graph, stop and return the maximum flow.

3. Using only valid edges in the level graph, keep executing depth-first search (DFS) from the source to sink until a blocking flow (no more paths through the level graph) is reached. Calculate the bottleneck values of all augmenting paths to calculate the maximum flow.

4. Repeat steps 1 to 3.

An example of this algorithm is shown by Figure. 7.6.

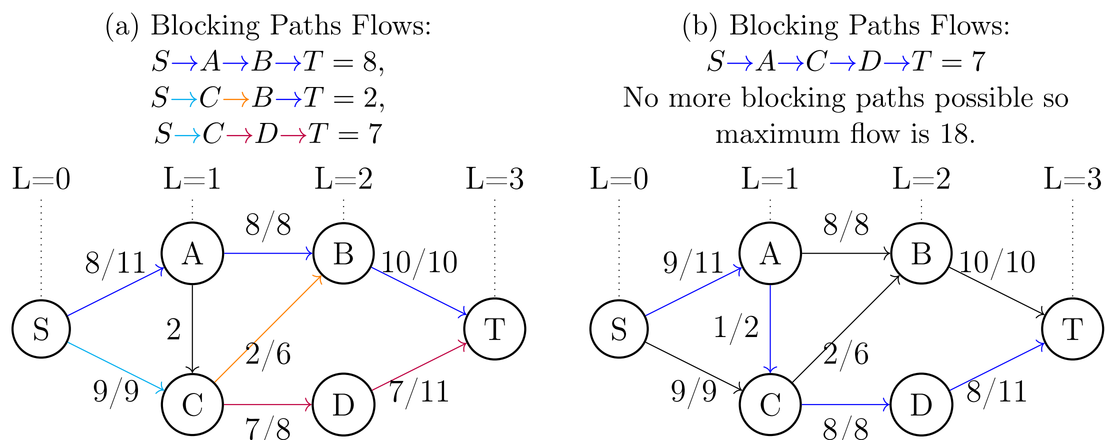


Figure 7.6: Example of Dinic's Algorithm

To calculate the complexity for this algorithm, the level graph is constructed using BFS in $O(|E|)$ time and a blocking flow is found in $O(|V||E|)$ time with a total running time of $O(|E| + |V||E|)$. As there is at most $|V| - 1$ blocking flows in the algorithm, the overall time complexity is $O(|V|^2|E|)$.

7.4 Push–Relabel Algorithm

The Push–relabel algorithm, also known as preflow-push algorithm; just like Edmonds–Karp and Dinic's algorithm, is a maximum flow algorithm. Unlike the other algorithms, this name stems from the two operations used in the algorithm: 'push' and 'relabel'. The algorithm was designed by Andrew V. Goldberg and Robert Tarjan in their 1988 paper [27].

Unlike the Ford–Fulkerson method, which tries to find augmenting paths from source to sink, examining the entire residual graph this algorithm works in a more localized fashion, by attempting to push a flow one vertex at a time. Another difference can be found in the net difference between the flow into and out of the vertices as a net flow of zero is maintained for every vertex in the Ford–Fulkerson method (except for the source and sink vertex). However, in the Push–Relabel algorithm, the value of the flow into a vertex u can exceed the flow out of u except for the source and sink.

The intuition behind this algorithm can be gained by picturing a set of pipes, the *edges*, and joints, the *vertices*. The source is the highest level joint and sends water "downstream" to all its adjacent vertices at a lower *height*, its *out neighbours*. Once a join has too much water, an *excess*, it pushes the water to a joint through a piper lower down and the water is only allowed to go to a joint at a lower height - it can never go up. If water can no longer flow and is trapped at a vertex, the vertex is then *relabelled* meaning its height is increased.

There are some important terms that will be needed to understand this algorithm which are described below:

Important terms for Push–Relabel Algorithm

- **Preflow**

  Preflow occurs when there is no longer a flow conservation (see 3rd condition of the maximum flow conditions) at the non source, s and sink, t, vertices. In a preflow, the flow entering the vertex is allowed to be greater than the flow leaving a vertex but not less. This is more formally described by Equation 7.1[1] and visually by Figure 7.7 where the flow from $S \rightarrow A$ exceeds $A \rightarrow T$. The difference of these values is called an excess.

$$v \in V \setminus s, t, \sum_{uv \in E} f(uv) \geq \sum_{vw \in E} f(vw) \tag{7.1}$$

- **Excess**

  Given a preflow f, a vertex v has an excess value $e(v)$ which is the difference between the flow entering the vertex and the flow leaving it. More formally described by Equation 7.2.

$$e(v) = \sum_{uv \in E} f(u, v) - \sum_{vw \in E} f(v, w) \tag{7.2}$$

- **Active**

  A node v is active if $v \notin s, t$ and has some excess value $e(v) > 0$.

Figure 7.7: Example of Preflow

Two operations needed for this algorithm are the push and relabel operations which I will describe below.

Push and Relabel

- **Push**

  Pushing is when an active vertex has to put its excess flow so that the total incoming flow is equal to the total outgoing flow. There are two types of push:

  - **Saturating push**

    A push operation that causes $f(u, v)$ to fill up $c(u, v)$ as the push uses the total capacity of the edge.

  - **Unsaturating push**

    Otherwise, if the total capacity is not used, this is known as a unsaturating push.

- **Relabel** The relabeling operation effects non source and sink active nodes. If a flow cannot proceed through a vertex, the height of that vertex is changed (relabeled). The labelling function $h : h(u) = h(v) + 1$, often also called the height function is when a height of a vertex u is reassigned. The label of a vertex is valid if

  1. $h(s) = |V|$
  2. $h(t) = 0$
  3. $h(u) \leq h(v) + 1$

If there exists a case where a valid relabel can occur, this means there is no augmenting path from source to sink as such a path will have at most $|V| - 1$ edges and each edge reduces the height of the flow by at most one and therefore this is impossible if conditions 1 and 2 of a valid relabel is true.

Using these two operations, we can push an excess of flow between vertices, relabelling the vertices as we go along ensuring that after each step the preflow and labelling is still valid. Once we obtain a valid preflow and label, meaning there exists no path between s and s and therefore the flow is maximal.

We can perform this algorithm like the Ford–Fulkerson method, with an empty preflow where all the flows of each edge is set to zero. However this is not possible as

this will produce an augmenting flow implying a valid relabelling doesn't exist.

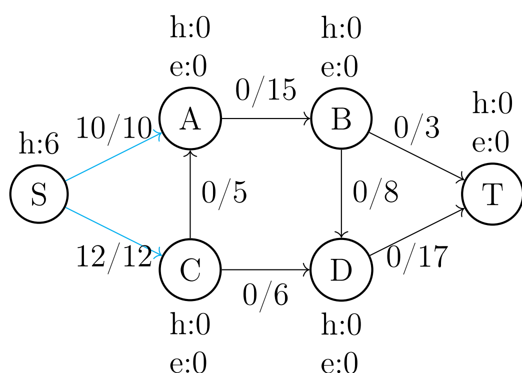
The pseudo-code for this algorithm is shown by Algorithm 14 and an example of this algorithm is shown by Figure 7.8.

Algorithm 14 Push–Relabel Algorithm [38]

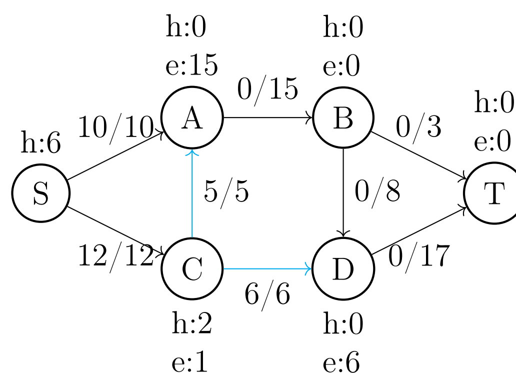
1: **procedure** PUSH–RELABEL(G, source, sink)
2: $h(source) \leftarrow |V(G)|$
3: **for all** $u \in V$ **do**
4: **if** $u \neq source$ **then**
5: $h(u) \leftarrow 0$
6: **for all** $(u, v) \in E$ **do**
7: $f(u, v) \leftarrow c(u, v)$
8: **while** $\exists\ u \neq t$ with $e(u) > 0$ **do**
9: **if** $\exists\ (u, v)$ residual edge in G and $h(v) < h(u)$ **then**
10: **if** (u, v) is forward **then**
11: increase $f(u, v)$ by $\min(e(u), c(u, v) - f(u, v))$
12: **if** (u, v) is backward **then**
13: decrease $f(v, u)$ by $\min(e(u), f(v, u))$
14: **else**
15: $h(u) \leftarrow h(u)$
16: **return** $\sum_{source, v \in E} f(source, \text{outneighbours}(v))$

The complexity of this algorithm is $O(|V|^2|E|)$ as there will be at most $O(|V||E|$ saturating and $O(|V|^2|E|)$ non saturating pushes. Choosing a vertex that allows us to find the next vertex in $O(1)$ time, which we do (as a queue is used in the implementation) gives the overall time complexity.

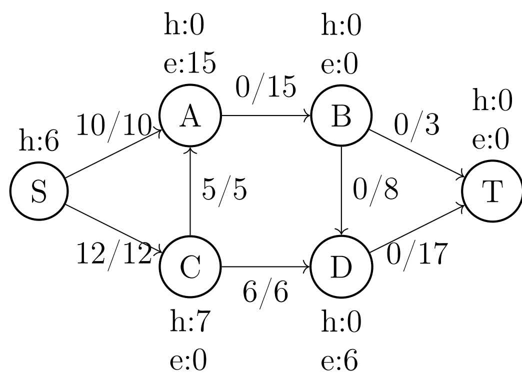
(a) The maximum capacity is pushed from
$S \to A, C$
A and C are now active.



(b) choose C
$h(C) = h(C) + 1 = 1$
push 6 from $C \to D$, $e(D) = 6$
$h(C) = h(C) + 1 = 2$
push 5 from $C \to A$, $e(A) = 15$
$e(C) = 1$
A, C and D are now active.



(c) choose C
$h(C) = h(S) + 1 = 7$
push 1 from $C \to S$, $e(C) = 0$
A and D are active.



(d) choose A
$h(A) = h(A) + 1 = 1$
push 15 from $A \to B$, $e(B) = 15$
$e(A) = 0$
B and D are active.

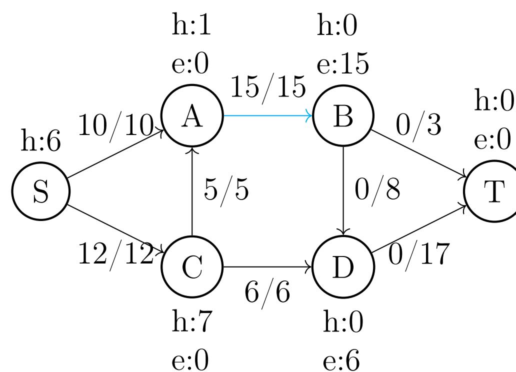


Figure 7.8: Example of Push–Relabel Algorithm Part 1

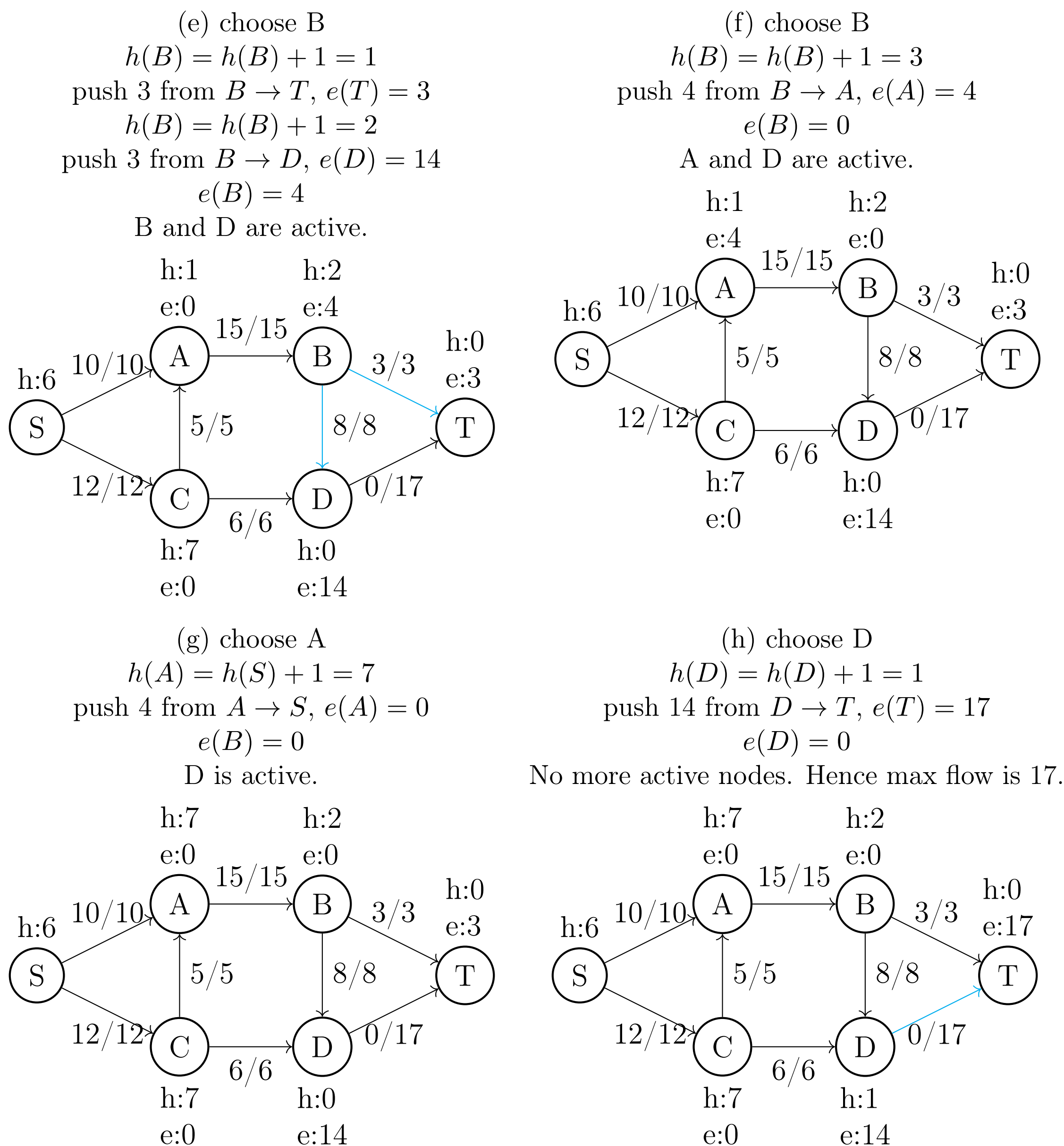


Figure 7.8: Example of Push–Relabel Algorithm Part 2

7.5 Max-Flow Min-Cut Theorem

This is a theorem stating that in a flow network, the maximum amount of flow passing from source to sink is equal to the total weight of the edges in a minimum cut.

Definition 7.5 (Cut [26])**.** Given a graph $G = (V, E)$ and a subset S of V, the cut $\delta(S)$ induced by S is the subset of edges $(u, v) \in E$ such that $|\{u, v\} \cap S| = 1$. That is, $\delta(S)$ consists of all those edges with exactly one endpoint in S. Given an undirected graph $G = (V, E)$ and for each edge $e \in E$ a non negative weight c_e, the cost of a cut $\delta(S)$, is the sum of the costs of the edges in the cut (see Equation 7.3).

$$c(\delta(S)) = \sum_{e \in \delta(S)} c_e \tag{7.3}$$

Definition 7.6 (Minimum-cut [26])**.** In a weighted graph $G = (V, E)$, a **minimum-cut** is a cut with the minimum cost. If all costs are 1 then the problem becomes the problem of finding a cut with as few edges as possible.

A minimum cut of graph is shown by Figure 7.9 where the dotted red lines represent a minimum cut (2 edges are removed) to disconnect the graph into two disjoint components.

Figure 7.9: Example of Minimum Cut

7.5.1 Karger's Algorithm

This is a randomized algorithm that can be used to find the minimum cut of a connected graph, which was developed by David Karger and first published in his 1993 paper [33].

Definition 7.7 (edge contraction)**. Edge contraction** is an operation on an an edge (u, v) where the edge is removed and its two incident vertices u and v are merged into a new vertex x, where the edges incident to x each correspond to an edge incident to either u or v. [28, Chapter 4].

If there are parallel edges, these are not removed as the contraction makes use of the disjoint union algorithm, when find is operated on two incident vertices u and v, these will belong to the same parent.

This algorithm works on edge weighted symmetric digraphs but the Digraphs package allows for bidirectional edges with different weights, therefore this algorithm does

not work on unsymmetrical edge weighted graphs but will still correctly return the correct solution to the unweighted minimum cut problem. The algorithm is a randomized algorithm in which we select a random edge (u, v) and contract the vertices u and v. Effectively, all the vertices combine into one super nodes, whilst the size of the graph decreases until two vertices remain. These two vertices may have any number of parallel edges. The number of edges left on this edge contracted graph where two vertices remain, is the minimum cut. The idea behind the contract in Karger's algorithm is listed below and visually demonstrated via graph $G = (V, E)$ in Figure 7.10.

Karger's Algorithm - Edge Contraction [16]

1. Replace u and v by a new vertex x.
2. Delete all edges (u, v).
3. For every edge (y, u), replace it by (y, x) where y is an incident vertex to u.
4. Similarly for every edge (y, v), replace it by (y, x).

The pseudo-code is described by Algorithm 15.

Algorithm 15 Karger's Algorithm [16]

```
1:  procedure KARGER(G)
2:      n ← |V(G)|
3:      C ← E
4:      for all i ← 1 to (n choose 2) do
5:          G′ ← G
6:          repeat
7:              choose arbitrary edge e ∈ E(G′)
8:              E(G′) ← E(G′) \ e
9:          until |V(G′)| = 2
10:         if |C| ≥ |E(G′)| then
11:             C ← E(G′)
        return C
```

In Figure 7.10, the graph does not have directed edges but the algorithm works the same with and without direction but will return a different number of cuts depending on how the edges are represented. In Figure 7.11, which is how undirected edges are represented in the Digraphs GAP package, the number of cuts will be 2, instead of 1 for each edge.

Figure 7.10d shows two remaining vertices $ABCD$ and E with two edges between them which means the minimum cut is 2. We can reuse the Find and Union algorithms that we first saw in the Disjoint Set Union section to find the vertices incident of the edge we wish to remove and then call union to merge them together. This procedure was seen in Kruskal's algorithm so this is interesting to see similar techniques being used across different groups of algorithms.

As this algorithm is randomized, it may not reach the correct solution on the first run making this a type of 'Monte Carlo' algorithm in which the output of such an algorithm has a small probability of being incorrect.

A single edge contraction is implemented with a linear number of updates and as the 'running time for contracting any given graph to two vertices is $O(n^2)$ as we contract n - 2 edges. Therefore the total running time for the algorithm is $O(n^4)$' [16]. As this is a Monte Carlo algorithm, the success probability is $1/\binom{n}{2}$. See [16] for the proof. The algorithm also implies that for any graph G, there are at most $\binom{n}{2}$ cuts [16]. It is unlikely, that the solution will be found on the first run and therefore we need to run the algorithm multiple times, keeping the smallest number of edges each run. Running the algorithm $|V|^2$ times gives a probability of failure of $1/e$ which doesn't depend on the number of vertices. However, the optimal number of iterations is $|V|^2 \ln|V|$ where the probability of failure is less than $1/|V|$ and probability of success is greater or equal to $1 - (1/|V|)$ [29].

7.5.2 Karger–Stein Algorithm

There is an optimisation we can make on this algorithm called the Karger–Stein algorithm. The idea behind this is that during the sequence of edge contractions, the chance of choosing the right edges to remove is high at the start but decreases as the number of vertices reduces. So for low vertices we run the normal Karger's algorithm and for larger number of vertices we run it more times. The pseudo-code for this optimisation is shown by Algorithm 16 in which the value 6 is chosen as $(6/\sqrt{2}) + 1 \geq 2$ which means for $|V|$ less than or equal to 6, we use Karger's algorithm and Karger–Stein algorithm otherwise. This is due to the fact that the original Karger's algorithm runs faster than Karger–Stein for graphs with less than or equal to 6 vertices.

Algorithm 16 Karger–Stein Algorithm [16]

1: **procedure** KARGER–STEIN(G)
2:     **if** $n \geq 6$ **then**
3:         cuts $\leftarrow$ perform 2 edge contractions and return the run with lower number of cuts
4:         **while** $|V| \geq |V|/\sqrt{2} + 1$ **do**
5:             contract edges
6:         Karger–Stein(G')
7:     **return** cuts

The total running time of this optimised algorithm is $O(|V|^2 \log^3|V|)$ [16].

7.6 Analysis

Here we will compare and analyse the maximal flow algorithms and then the two versions of the minimum cut algorithm. As with the previous chapters, extra plots for the individual algorithms can be found in Appendix E.

7.6.1 Maximal Flow Algorithms

Starting with the plot which takes into the average of all the graph density results gives us Figure 7.12. As we can see Dinic's algorithm out performs Edmonds–Karp on average but as the complexities are different with Edmond–Karp grows slower with respect to the number vertices, it might perform better on sparser graphs.

Comparing the algorithms on sparse graphs shows us that the two algorithms perform similarly as shown by Figure 7.13 but as we increase the density of the graphs, a clear trend emerges. Figures 7.14, 7.15, 7.16 shows that the gap between the two algorithms performance grows as we increase the density until Figure 7.17 which is a complete graph highlighting the biggest performance difference. As Dinic's algorithm performs better than Edmonds–Karp in every instance and Edmonds–Karp can only match the performance of Dinic's algorithm, when integrating the max flow algorithm into GAP I selected Dinic's for every situation.

This was until I implemented the third maximal flow algorithm - the Push–Relabel algorithm which if we compare to the original two algorithms, we can see by Figure 7.18 that it outperforms them both on average. But as we know this can give a good sense of the algorithms performance but there may be cases when Dinic's algorithm out performs it. We we will only compare Dinic's as this algorithm performs better than Edmonds–Karp algorithm in every case. Again, if we compare the algorithms for each density of graph, we can see that for denser graphs (see Figure 7.19), the performance is similar but the Push–Relabel algorithm is slightly faster, with this gap increasing for sparser graphs as shown by Figure 7.20, 7.21, 7.22 and 7.23 which is the reverse trend to the comparison between Edmonds–Karp and Dinic's algorithm. Nonetheless, there is no case where the Push–Relabel algorithm is improved upon and therefore will be the one I implement solely into the Digraphs package.

7.6.2 Minimum Cut Algorithms

This section will compare Karger's and the Karger–Stein algorithms. From the respective sections, we saw that there is a big disparity in the two algorithms with time complexity of $O(n^4)$ for Karger's algorithm and $O(n^2 \log^3 n)$ for Karger–Stein.

The plots for these algorithms align with the predicted complexities with the performance for Karger's algorithm executing much faster in every scenario. Figure 7.24 shows the extreme performance difference that the optimised Karger–Stein algorithm makes. This is prevalent across all the graph densities tested. Furthermore, as the complexity of the base Karger's algorithm is of order 4, I decided to run the algorithms up to graphs of size 500 vertices as this was computable in a reasonable amount of time

and also was enough to show the trend and performance difference of the algorithms.

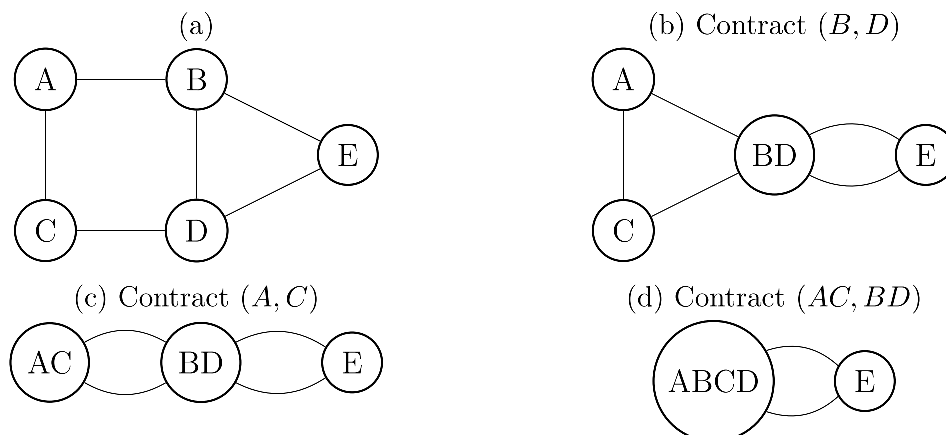


Figure 7.10: Example of Edge Contraction in Karger's Algorithm

Figure 7.11: Example of Edge Contraction in Karger's Algorithm with directed graph

Figure 7.12: Overall comparison of Edmonds–Karp vs Dinic's Algorithm

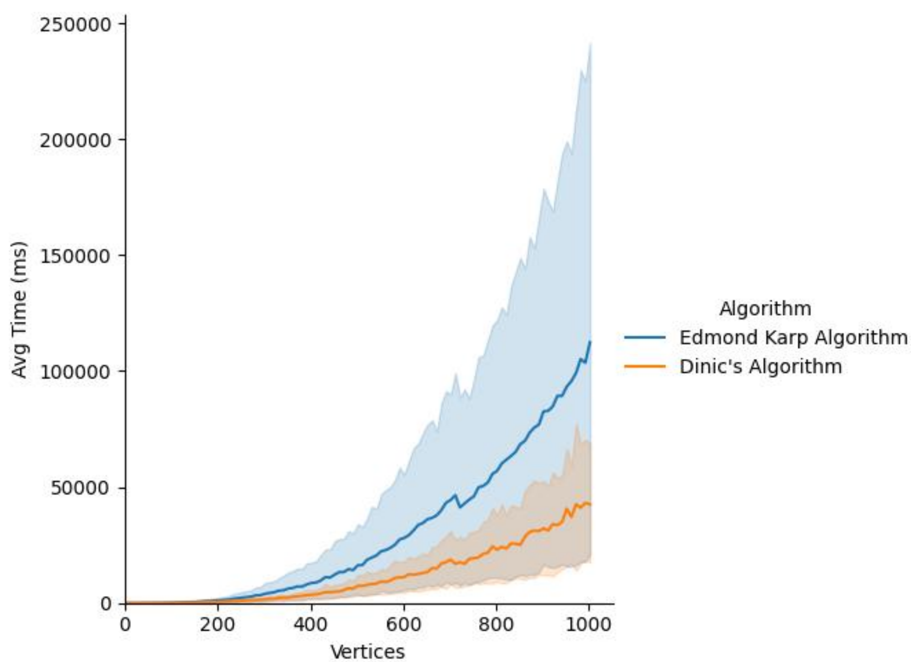


Figure 7.13: Comparison of Edmonds–Karp vs Dinic's Algorithm with 0.01 edge probability

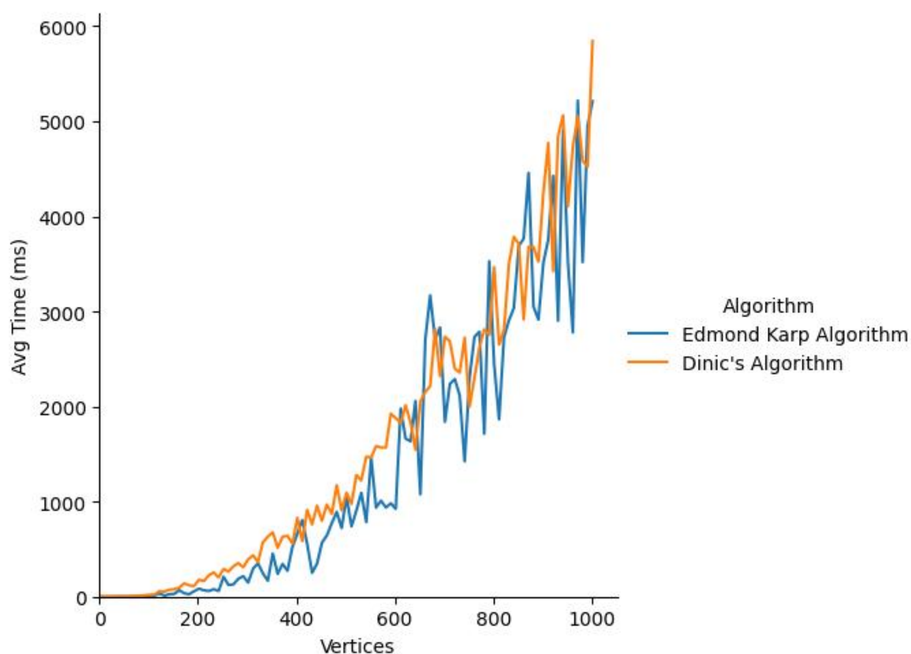


Figure 7.14: Comparison of Edmonds–Karp vs Dinic's Algorithm with 0.125 edge probability

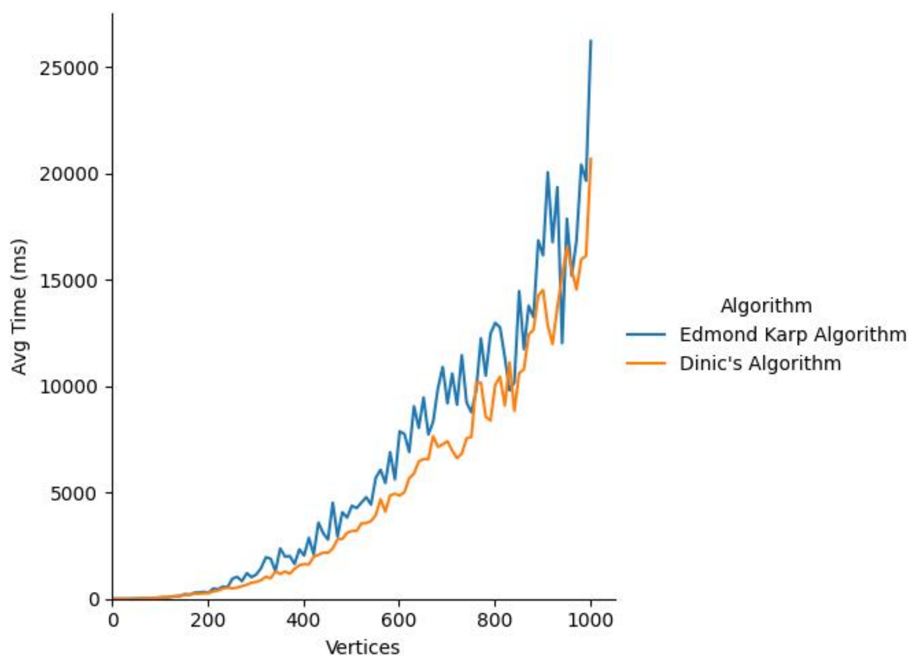


Figure 7.15: Comparison of Edmonds–Karp vs Dinic's Algorithm with 0.25 edge probability

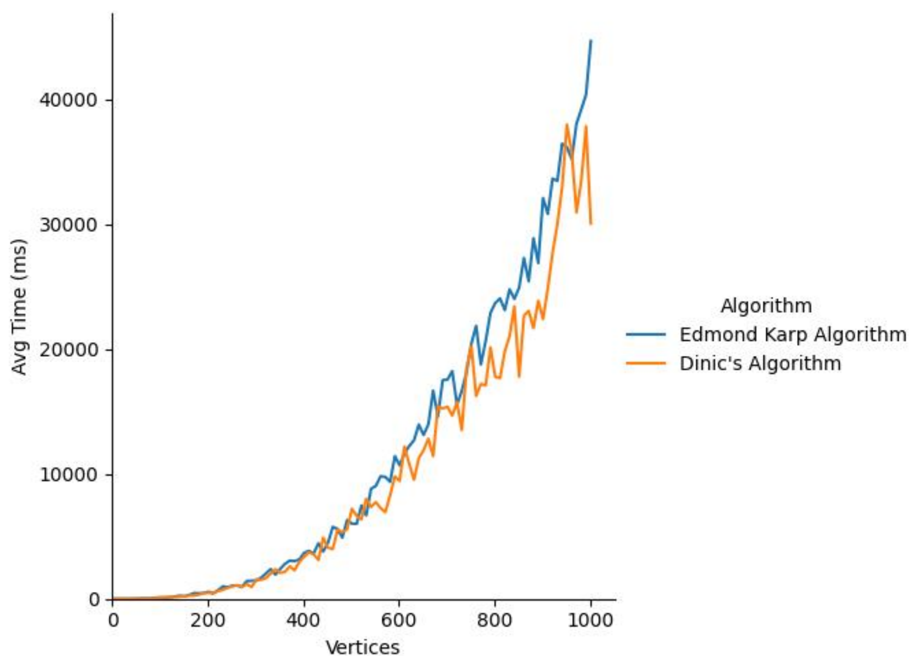


Figure 7.16: Comparison of Edmonds–Karp vs Dinic's Algorithm with 0.5 edge probability

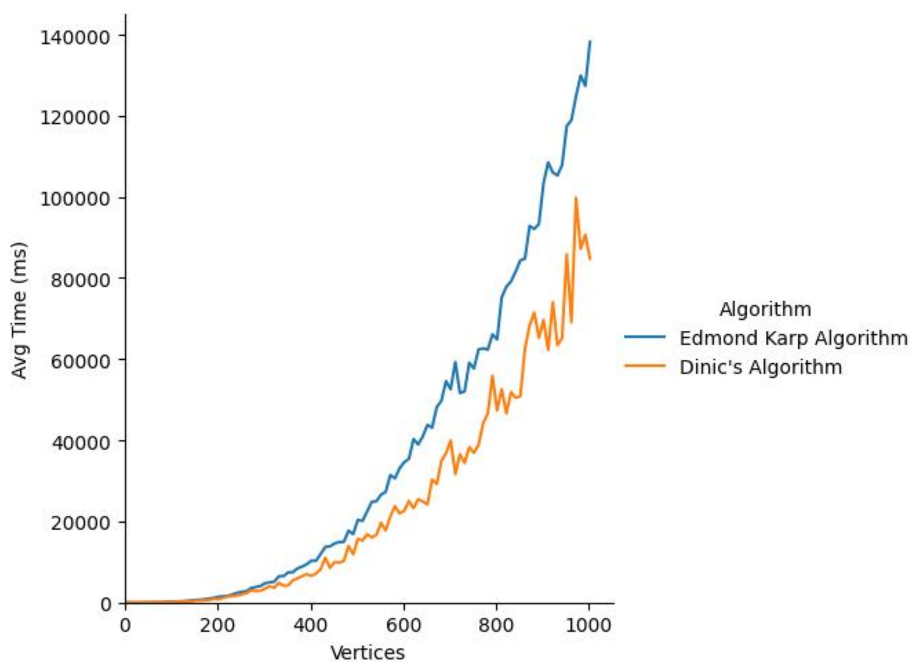


Figure 7.17: Comparison of Edmonds–Karp vs Dinic's Algorithm with 1 edge probability

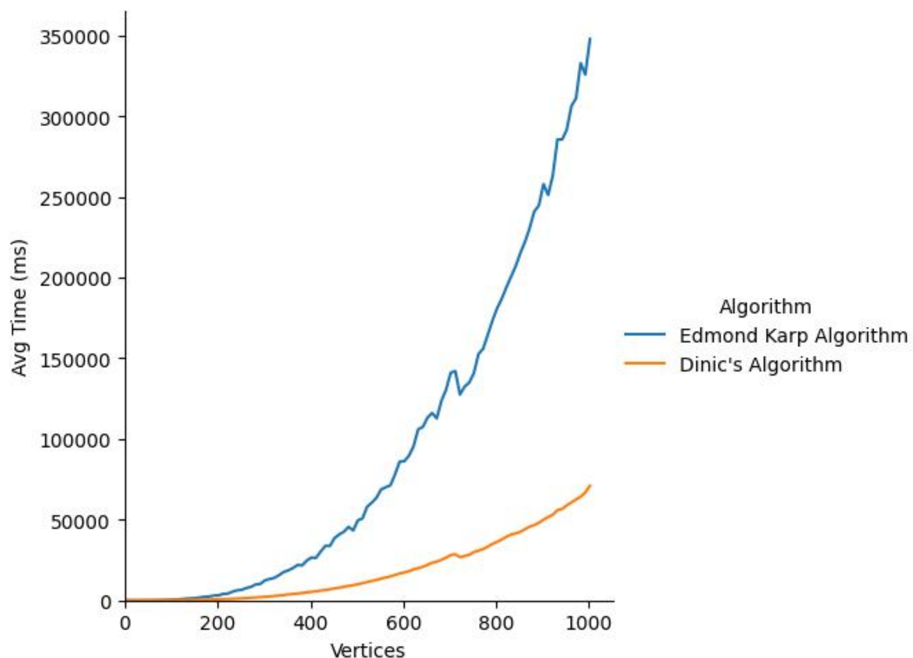


Figure 7.18: Overall comparison of Edmonds–Karp, Dinic's and Push–Relabel Algorithm

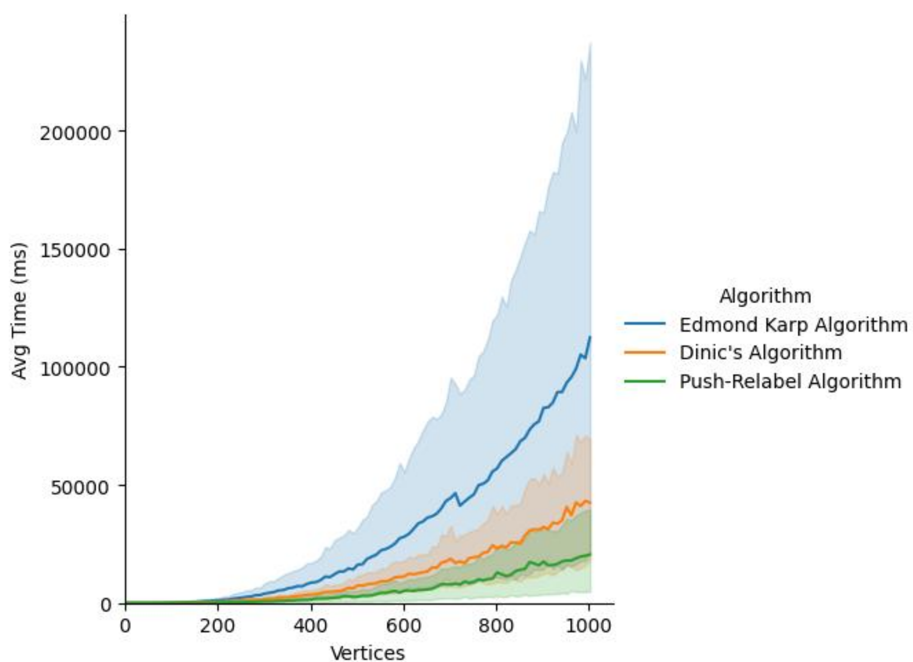


Figure 7.19: Comparison of Dinic's vs Push–Relabel Algorithm with 1 edge probability

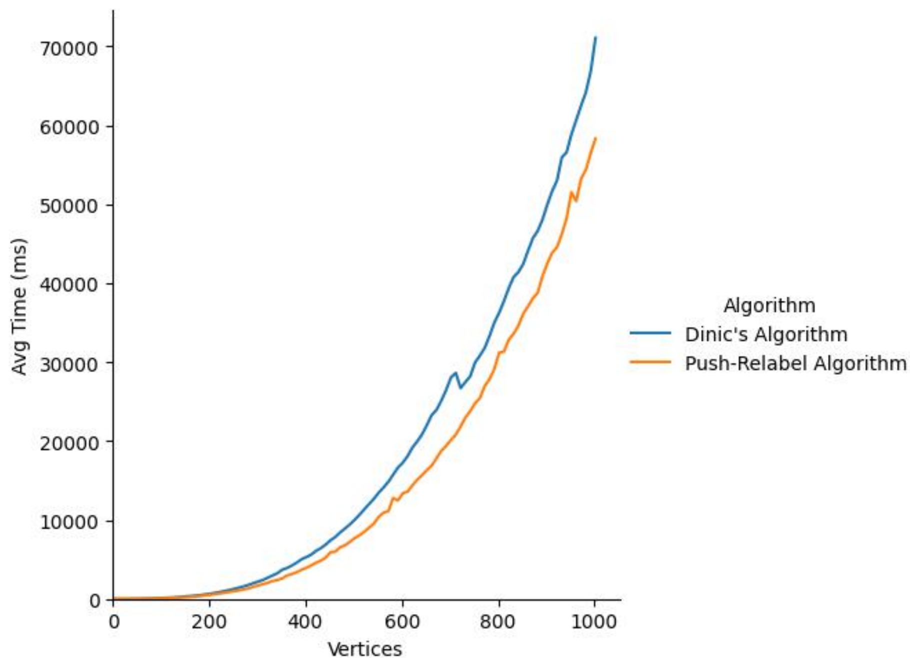


Figure 7.20: Comparison of Dinic's vs Push–Relabel Algorithm with 0.5 edge probability

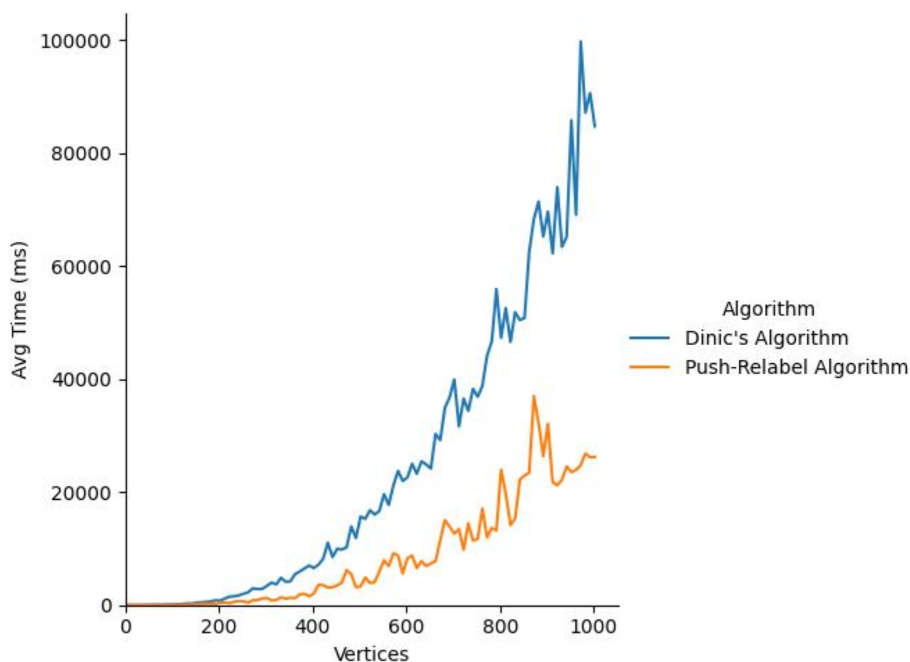


Figure 7.21: Comparison of Dinic's vs Push–Relabel Algorithm with 0.25 edge probability

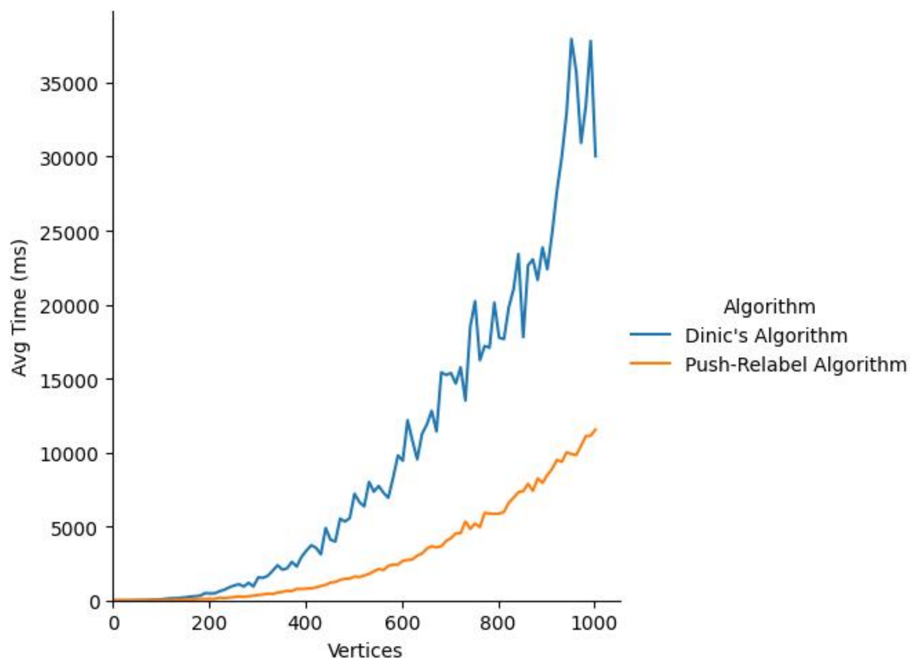


Figure 7.22: Comparison of Dinic's Algorithm vs Push–Relabel with 0.125 edge probability

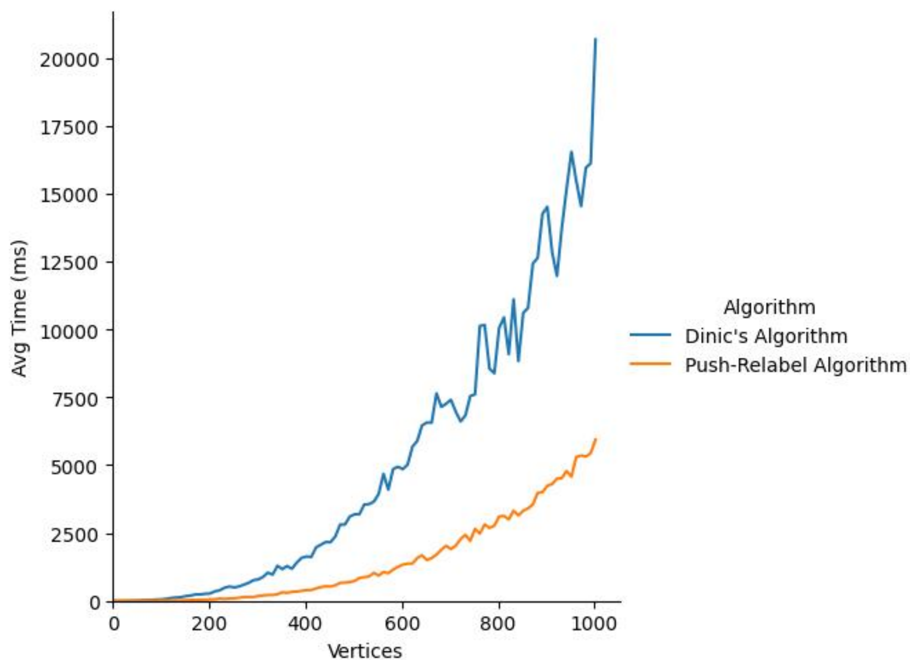


Figure 7.23: Comparison of Dinic's vs Push–Relabel Algorithm with 0.01 edge probability

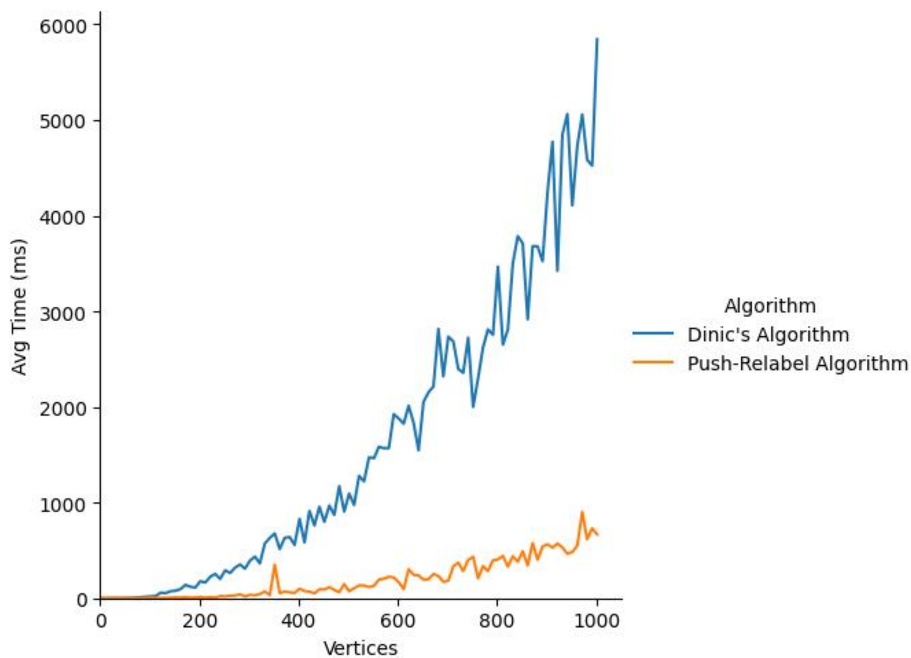
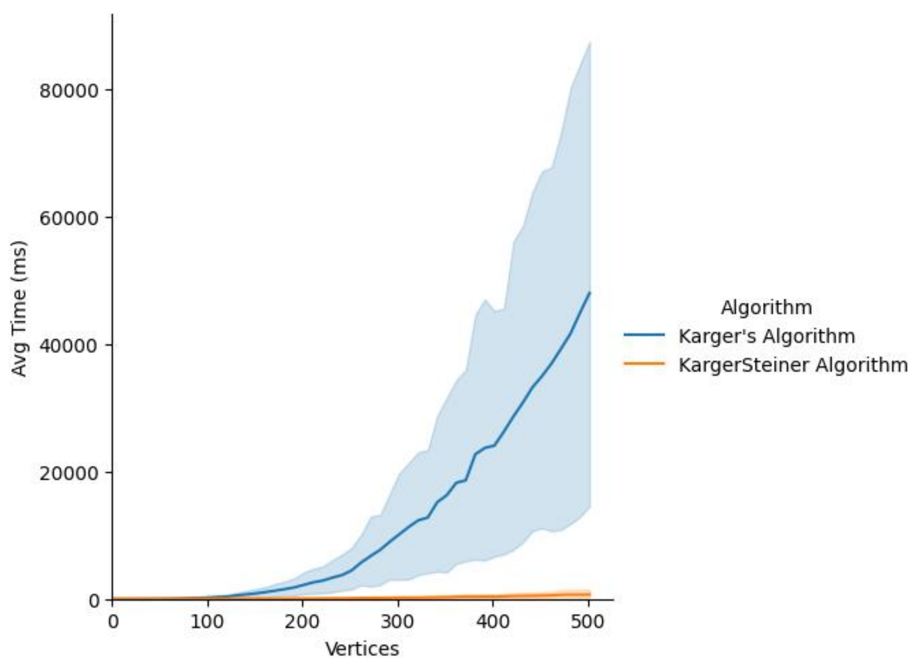


Figure 7.24: Overall comparison of Karger's vs Karger–Stein's Algorithm



Chapter 8

Working with GAP

GAP is a non-object oriented language which is very dissimilar to programming paradigm to languages I have worked with such as Java and Python which both have objected oriented features or even Haskell, as functional programming language. This was not an issue and GAP was easy to pick up.

GAP consists of a core system and a number of packages - including Digraphs, the package I wrote all the algorithms for as well as other useful ones such as datastructures, GAPDoc, profiling. Using the PackageManager library, it was easy to install the relevant packages required for the project. If there was a function I was looking for, the documentation was easy to use by typing a question mark followed by some characters describing what you are looking for and GAP would return a list of functions that matches what was typed. Typing a double question mark followed by the function name returned the GAP documentation.

As of version 1.6.2, in `Digraphs/gap/`, all the GAP code for Digraphs can be found here but as Edge Weights weren't implemented yet, I added my own `weights.gd` and `weights.gi` files. The file with the `.gd` extension can be seen as a header file in C containing all the declarations for the implementations in `weights.gi` file. To be able to compile and use these files, the `weights.gd` needed to be added to `./Digraphs/init.g` file and likewise for `weights.gi` file within `./digraphs/read.g` so GAP knew where these files were located. A similar procedure for the documentation was needed as I added a `weights.xml` file containing all the documentation for the relevant functions and the name of this file was added to `./Digraphs/gap/doc.g`. In order for the documentation to added to GAP, a new section also needed to be added to a chapter of the documentation. I added my documentation as an 'Edge Weights' section in Chapter 6 of the documentation.

8.1 Relevant Gap Object Types

In GAP, there are families of objects which include *attributes*, *properties* and a way to select methods using *operations*. The following sections will describe the importance of these families and how they were used.

8.1.1 Attributes

Attributes are knowledge that can be computed about an object. The GAP documentation states 'the attributes of an object describe knowledge about it. It is an unary operation' [23]. 'Once these values of the objects are computed, they are cached so that it is very cheap to get the value when the attribute is called again' [23].

Example attributes declared in project

- *EdgeWeights*
- *DigraphEdgeWeightedMinimumSpanningTree*
- *DigraphEdgeWeightedShortestDistances*
- *DigraphMinimumCuts*

8.1.2 Properties

Properties of an object are those whose values are only true or false. These are also cached the first time they are accessed for future calls, similarly to attributes.

Examples of properties declared in this project

- *IsNegativeEdgeWeightedDigraph*

8.1.3 Operations

Operations correspond to a set of GAP functions, called *methods* of the operation which are installed in `weights.gi`. 'Each call of an operation causes a suitable method to be selected and called' [23]. This selection is dependent on the type of arguments.

Examples of operations declared in this project

- *EdgeWeights*
- *DigraphEdgeWeightedMinimumSpanningTree*
- *DigraphEdgeWeightedShortestDistances*
- *DigraphMinimumCuts*

An example of operations being 'overrided' is best shown when we try to create a random digraph using *RandomUniqueEdgeWeightedDigraph* which can take in multiple types of arguments and is therefore multiple operations are declared as below:

- DeclareOperation("RandomUniqueEdgeWeightedDigraph",[IsPosInt]);
- DeclareOperation("RandomUniqueEdgeWeightedDigraph", [IsPosInt, IsFloat]);

- DeclareOperation("RandomUniqueEdgeWeightedDigraph", [IsPosInt, IsRat]);
- DeclareOperation("RandomUniqueEdgeWeightedDigraph", [IsFunction, IsPosInt, IsFloat]);
- DeclareOperation("RandomUniqueEdgeWeightedDigraph", [IsFunction, IsPosInt, IsRat]);

The second argument of the declaration represents the types required for that method. If `RandomUniqueEdgeWeightedDigraph(5, 0.1)` is called, then the relevant operation is called- the one with `[IsPosInt, IsFloat]` and the matching method is selected.

Now that all the necessary objects are declared, an installation of the methods is required. The simplest example of this in the project is creating a mutable edge weight copy with the declaration `DeclareOperation("EdgeWeightsMutableCopy", [IsDigraph and HasEdgeWeights]);` in which the list in the second arguments specify the type required for the object passed into the method; it has to be a digraph object and also have edge weights. The matching method is as below (see Figure 8.1) which simply takes the object and returns a mutable copy of it. The third parameter, the list, must match the declaration.

```
 InstallMethod(EdgeWeightsMutableCopy, "for a digraph by edge
weights", [IsDigraph and HasEdgeWeights], D -> List(EdgeWeights(D),
ShallowCopy));
```

Figure 8.1: Example of Method Installation in GAP

There maybe cases where we need to declare functions that are private, such as helper functions for the methods that are documented and intended for the user. There is no notion of access modifiers in GAP and all functions declared in `.gi` files can be accessed. However the syntax and convention in GAP, in particular the Digraphs package, is to write these functions prefixed with 'DIGRAPHS_', followed by the function name. An example of this is the Find and Union helper functions defined as 'DIGRAPHS_Find' and 'DIGRAPHS_Union' outlined in Disjoint Set Union section. These are functions that may be used by other installed methods and are therefore best conceptualised as private methods.

If the function is private but only relevant to one method, they can be defined within the `InstallMethod` operation as seen by the `DigraphMaximumFlow` method which simply calls `Dinic(digraph, source, sink);` where `Dinic` and all helper functions, also including global variables for these methods are declared. An outline of this method is shown below by Figure 8.2 which shows the installed method 'DigraphMaximumFlow' calling `Dinic` which in turn calls the local helper functions only defined within this operation and isn't visible outside of this scope.

```
1    InstallMethod(DigraphMaximumFlow, "for an edge weighted digraph",
2    [IsDigraph and HasEdgeWeights, IsPosInt, IsPosInt],
3    function(digraph, source, sink)
4        function: helperOne();
5
6        function: helperTwo();
7
8        function: Dinic(digraph, source, sink);
9
10       return Dinic(digraph, source, sink);
11   end;
```

Figure 8.2: Example of Method Install with local functions

Chapter 9

Visualisation

This chapter strays from the algorithm implementation, to the visualisation and highlighting of subgraphs within a digraph. This is useful for highlighting trees and paths (in the context of this project- minimum spanning trees and shortest paths) allowing a user to more easily obtain the information they are looking for rather than constructing the subgraph manually from the algorithm output themselves. The highlighting also provides utility for specifying different options for highlighting such as source vertex colours as well as other configurations.

To visualise Digraphs at the moment we can visualise graphs by using the open source Graphviz software, in particular making use of the DOT language. In GAP, we can create a representation of a digraph using the `DotDigraph` function where the vertices are displayed as circles and arrows displayed between vertices, with the arrowhead of each line pointing towards the range of the edge. Figure 9.1 shows an example digraph which is able to be visualised after the DOT language is compiled.

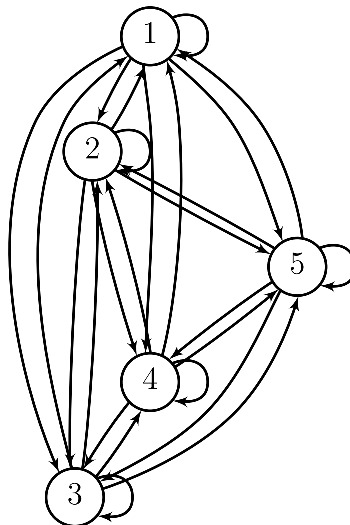


Figure 9.1: Example of DotDigraph

This is clearly a useful tool which can be expanded upon to be able to highlight subgraphs within this digraph such as minimum spanning trees and shortest paths.

9.1 Highlighting Minimum Spanning Trees

For the graph represented by Figure 9.2a shows the original digraph and Figure 9.2b shows the the minimum spanning tree (the subgraph) highlighted. As the DigraphEdge WeightedMinimumSpanningTree function returns a digraph. There is a small detail that must be taken into account, which is that the graph must be undirected and during the implementation we convert a directed graph into an undirected one to allow the algorithms to function. Therefore, when looking at the highlighted graph, we must ignore the edge directions as taking them into account would mean some vertices are impossible to reach such as vertex 4 in Figure 9.2b would be unreachable if the directions are followed.

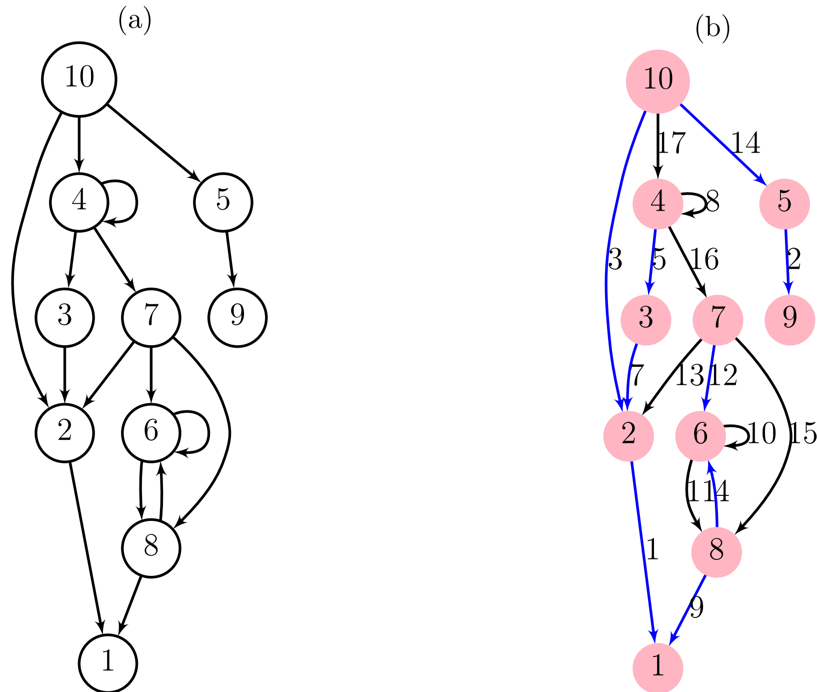


Figure 9.2: Example Visualisation for MST

9.2 Highlighting Shortest Paths

This utility is very similar to visualising the MST's, but can highlight the shortest path from one source vertex to all other vertices. The output for the MST's includes a digraph so to highlight it we can pass the MST, as well as the original digraph. However, in order to highlight shortest paths, we need to convert the output from the relevant algorithms - a record containing a list of parents, edges and shortest distances. We can use the parents and edges list to construct a digraph, which will be the required subgraph that can be highlighted. We can pass the record from the output of one of Dijkstra's or the Bellman–Ford algorithm into one of two function. The two functions are closely related and are described below:

Functions to generate a subgraph from Shortest Path Algorithm output

- **DigraphFromPaths(digraph, record)**

  This function takes the original digraph and a record which is the output from one of the SSSP algorithms and returns a digraph of all the shortest paths.

- **DigraphFromPath(digraph, record, destination)**

  This function is similar to above, except it creates a digraph containing the path from the source to destination vertex. The source is obtainable from the record in the parameter as one of the entries

As in the previous section, the original digraph and digraph with highlighted paths are shown by Figure 9.3. Here, the source vertex is 2 and all the shortest paths from the source vertex are highlighted in blue.

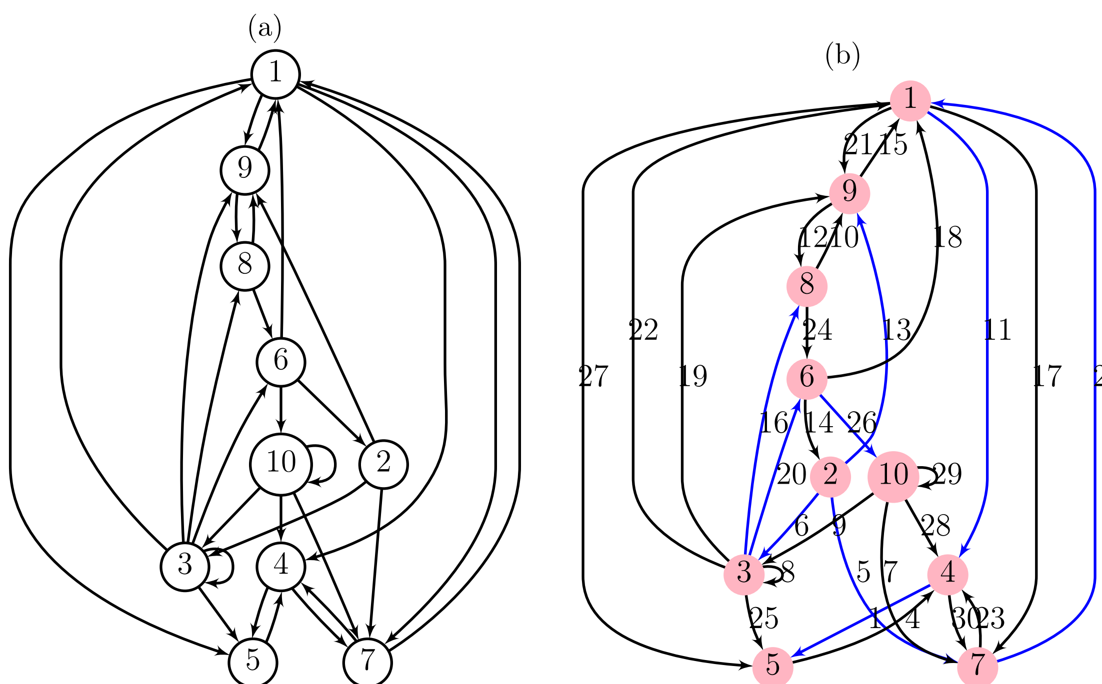


Figure 9.3: Example Visualisation for Shortest Paths

To get from vertex 2 to vertex 1, we take the path $2 \rightarrow 7 \rightarrow 1$. The sequence of commands to create the subgraph and highlight it is shown by Figure 9.4 where we get the shortest path (sp) and then create the subdigraph (sd) and finally we can splash the DOT of both the original and subgraph. In the figure, an empty record is passed through which means the visualisation will use the default parameters which are outlined as follows:

```
gap> d := RandomUniqueEdgeWeightedDigraph(IsStronglyConnectedDigraph, 10,0.1);
<immutable digraph with 10 vertices, 29 edges>
gap> sp := DigraphEdgeWeightedShortestPaths(d, 2);
rec( distances := [ 7, 0, 6, 18, 19, 26, 5, 22, 13, 52 ],
  edges := [ 1, fail, 1, 1, 1, 4, 2, 5, 3, 3 ],
  parents := [ 7, fail, 2, 1, 4, 3, 2, 3, 2, 6 ] )
gap> sd := DigraphFromPaths(d, sp);
<immutable digraph with 10 vertices, 9 edges>
gap> Splash(DotEdgeWeightedDigraph(d, sd, rec()));
```

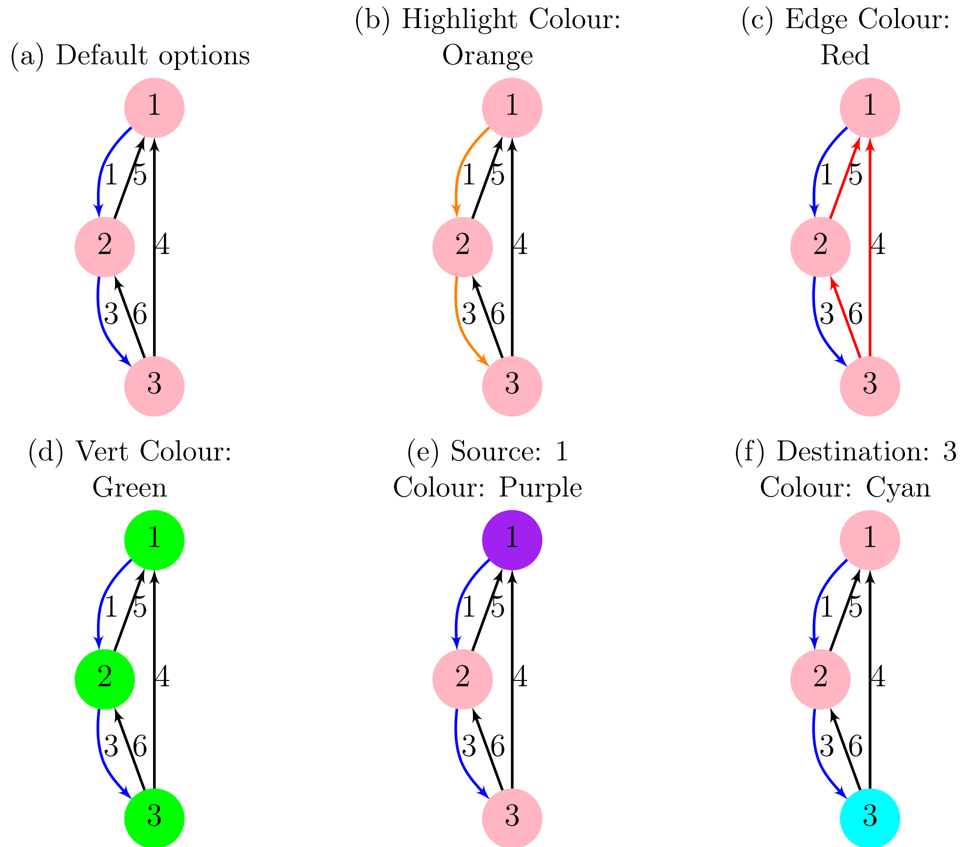
Figure 9.4: Creating Shortest Path subgraph from Algorithm output

Highlighting Graph Options

- **highlightColour** [default: blue]: This is the colour of the highlighted edges.
- **edgeColour** [default: black]: This is the colour of the non highlighted edges.
- **vertColour** [default: light pink]: This is the colour of the vertices.
- **sourceColour** [default: green]: This is the colour of the source vertex.
- **destColour** [default: red]: This is the colour of the destination vertex.

The user may also specify a 'source' and 'vertex'. If no colours are provided, then the default one will be used for the respective vertex. There is no default source or vertex and specifying one is up to the discretion of the user. Some examples of changing the options are shown below by Figure 9.5. Figure 9.6 shows all the options one single graph. The colours chosen are quite vibrant and is mainly to highlight the visualisation aspect.

These graphs were demonstrated using the DigraphFromPaths function. If we wanted to highlight the path from a single source to a single destination, we would use the DigraphFromPath function, specifying a destination. The source is able to be obtained from the shortest path algorithms and therefore we can highlight the a single path from source to destination without highlight all other paths. This is shown by Figure 9.7 shows the path from source vertex 3 to destination vertex 1 without highlighting the path to vertex 2.



(a) Default options

(b) Highlight Colour: Orange

(c) Edge Colour: Red

(d) Vert Colour: Green

(e) Source: 1 Colour: Purple

(f) Destination: 3 Colour: Cyan

Figure 9.5: Examples of changing visualisation options

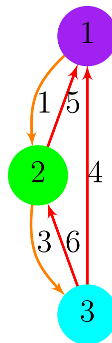


Figure 9.6: Example of all visualisation options on one graph

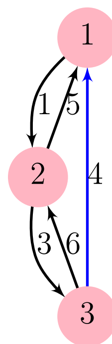


Figure 9.7: Example of highlighting a single path

9.3 Implementation

The implementation will focus on three distinct parts; the first two consisting of describing how the subgraph is constructed from the output from the shortest path algorithms and the third part about converting the digraphs into the DOT language.

The first part is regarding the DigraphFromPath function which will start at the destination provided in the function parameters and work backwards following the list of parents until a fail is reached, indicating this is the source vertex.

The second part is about the DigraphsFromPaths function which will create a digraph of all the paths in the subgraph. This is most easily demonstrated by showing the original digraph and the subgraph that is produced but before it is highlighted as shown by Figure 9.8. Using these two digraphs, we can then traverse the digraph and set the relevant vertices and edges to the default or selected colours.

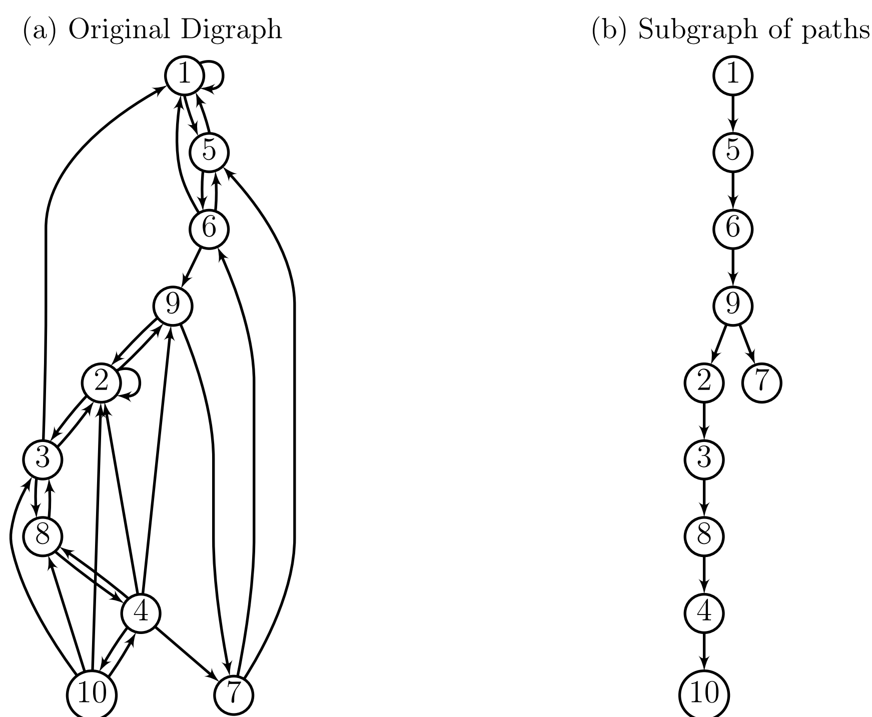


Figure 9.8: Examples of changing visualisation options

Chapter 10

Testing and Analysis Process

This chapter looks at how testing was performed during the implementations of the algorithm and also during the integration with the Digraphs package. The testing and analysis was automated, so I was able to collect data on the desired algorithm and then run a python script (analyser.py) to automatically plot the data depending the requirements, such as plotting individual algorithms or comparing multiple algorithms.

I also profiled the code using the GAP profiling package, as well as the CI/CD pipeline already implemented for the Digraphs package. I was able to make use of the CI/CD features when making a pull requests to merge my fully profiled, tested and linted code to the Digraphs package.

10.1 Testing

There were two forms of testing that I conducted. The first was using the tests that could be executed via the `DigraphsTestStandard` function. This runs a sequence of tests in the `Digraphs/standard/tst/` directory. This involved writing tests, similar to unit tests where we have an expected output and can compare against the output of the code to verify that it is what we expected. Using this form of testing I was able to test normal inputs - where no errors are expected, edge cases and special inputs.

The other form of testing carried out was to check the algorithms were working as expected on bigger input sizes. Therefore, I wanted to be able to check the output of the algorithms were correct and I was able to achieve this by checking the output of one algorithm against each other. When testing, certain algorithms may return different paths to each other so I had to compare the part of the output that would be consistent between them. For minimum spanning trees, the minimum value of the mst returned must be the same between the algorithms. The mst algorithms may return different subgraphs unless the original graph had unique edge weights, in which case there will be an unique solution and thus we can compare both the mst's returned by the minimum spanning trees algorithms.

A similar pattern is followed for the algorithms for shortest path and maximal flow algorithms except the shortest path algorithms are further divided into two categories; the two SSSP and two APSP algorithms are compared with each other as they each return the same output format.

The final group of algorithms are the maximum flow algorithms which could return different flows so these algorithms were tested using the maximum value that they return. This is a consistent answer that is expected to be the same between the two algorithms if implemented correctly.

There is another group of algorithms I wrote code for which was the Minimum Cut algorithms but these algorithms compute the answer in a random way each time and is therefore not guaranteed to return the correct answer every time. This made these algorithms difficult to test as the expected answer may not always be obtained or one of the two algorithms fails to find the lowest value and the other one does and this will result in a failed test. Therefore, when testing these algorithms, I allowed for a buffer of one so if the algorithms were within one cut of each other they would pass.

To ensure that all parts of the code was tested or at the very least, executed, I was able to profile the code. This involved running all the test cases and checking which parts of the code were being executed and if the correct output was returned. The final implementation has a coverage of 100 percent and every single line of code was covered and tested.

10.2 Automated Analysis

To automate the analysis, I had to first generate random graphs that can be passed into each of the algorithms as arguments. These graphs are created with random edge weights with arguments: $filter$, n and p where $filter$ is the type of graph that is to be created such as Connected, Strongly Connected, n is the number of vertices and p is the approximate probability of the vertices having an edge. When creating the random graphs with random weights, the implementation for this algorithm made use of a hash set and while loop to select a random unique weight between 1 and the number of edges. The while loop would keep selecting a random weight and if it wasn't already in the hashset, it would use that weight. This was suitable for lower number of vertices as the number of edges was lower. But it is obvious that as the number of edges grew, the while loop could become extremely slow as the pool of random numbers available shrunk as they became selected. This would result in the random graph creating becoming a bottle neck for the analysis. To remedy this issue, I precomputed a list of unique random numbers of the same range as before and iterated through the list when selecting a random weight.

Another random factor of the testing that was required was the creation of the graphs for the algorithm to operate on.

I wrote a gap file for each group of algorithm with the parameters as defined `(algName, nodes, probability, nrIterations, step)`. The purpose of these are described below.

- algName: Pass the name of the algorithm that you wish to run, that matches the name of the file containing the code for the algorithm. You can also pass 'all' as parameter to run all the algorithms at once. This can be temperamental and sometimes it works when the files are read in manually in the GAP terminal.

- nodes: The maximum number of nodes that you want to run the algorithm to.
- probability: The probability that two vertices u and v will have an edge. This effects how dense or sparse the graph will be. A higher p will result in a more dense graph and the opposite is true for a lower p value.
- nrIterations: How many times is the algorithm repeated for a given number of nodes.
- step: This is the increment between the number of nodes.

The files containing the algorithm also contained some additional lines of code to output the time take for the algorithm to run into a CSV file. The tester file creates the CSV file with the appropriate headers and the files with algorithm appends to it the data which consists of Number of Vertices, Number of Edges, Start time and End time. The start and end time were computed using the `Runtimes()` function in GAP, specifically the 'user_time' element in the record.
I ran the algorithms on a variety of probabilities and number of vertices. This varied the density of the graphs. The probabilities I analysed were 1.0 (which is effectively a complete graph), 0.5, 0.25, 0.125 and 0.01. The number of nodes tested was between 1 and 1000. As I was running 5 repeats to obtain an average I found 1000 nodes was sufficient to see the trend of the performance of the algorithms without taking an unreasonable amount of time.

There was also further analysis I did on the minimum spanning tree algorithms to analyse Prim's, Kruskal's and Borůvka's algorithm as I found the results for these algorithms strange. These plots involved plotting number of edges against time instead of number of number of vertices and I carried out testing for this in a similar fashion to the rest of the automated data generation and analysis. This also involved generating the data with the same column headers but with a fixed number of vertices, varying the edges. As the number of edges were random and not fixed, I wasn't able to run repeats as for each edge probability, the number of edges in that random graph is not guaranteed to be the same very time. Once the data was collected, I was able to plot the data in a similar way, except with different axis for the plots.

10.2.1 Pandas and Seaborn

Once, the data is collected in CSV's for the various dense graphs. I wrote a python program that has the following API:

Python Analysis API

- **-p [required]**: Provide paths to all the CSVs of one single algorithms
- **-c [optional]**: If multiple algorithms' CSVs are to be compared
- **-s [optional, default: false]**: If we wish to save the plots

Given the CSV files for the algorithm, this would plot the graphs with the appropriate title, legend, axes automatically. To do this I used pandas to create a dataframe containing the data and used seaborn to plot the pandas dataframe. All the plots of run times against number of vertices were created using this setup.

Chapter 11

Evaluation and critical appraisal

In this section I will repeat the goals outlined in Project Aims in a similar fashion to how they were laid out and discuss if and how they were achieved as well as reviewing my own work to current similar work done regarding algorithms for directed graphs.

11.1 Primary

Primary Goals

1. Implement in GAP and compare ✓
   (a) Three Shortest Path Algorithms ✓
   This was achieved as Dijkstra's, Bellman–Ford and Floyd–Warshall algorithm were all implemented and compared with each other.
   (b) Two Maximal Flow Algorithms ✓
   This was achieved as Edmonds–Karp and Dinic's algorithm were all implemented and compared with each other.
   (c) Two Minimum Spanning Tree (MST) Algorithms ✓
   This was achieved as Prim's and Kruskal's algorithm were both implemented and compared with each other.
2. Implement a new object type for edge-weighted digraphs in GAP, compatible with the Digraphs package, and include appropriate constructors. ✓

   This was achieved as constructors for edge weighted digraphs were created, with suitable testing, documentation and a pull request was made to merge the code into the Digraphs package for the community to use.

11.2 Secondary

1. Implement a further algorithm for each group and compare to the other relevant algorithms (implemented in the primary objectives). ✓

   For each category I implemented a further algorithm for each category as outlined below:

   (a) Shortest Path Algorithm: Johnson's algorithm
   (b) Maximal Flow Algorithm: Push–Relabel Algorithm
   (c) Minimum Spanning Tree Algorithm: Borůvka's Algorithm

   These were then also compared with the previously implemented algorithms for each relevant group.

2. Write GAP code to allow for the shortest path and minimum spanning trees to be highlighted during visualisation of graphs. ✓

   This was achieved as both minimum spanning tree and shortest path(s) can be visualised in GAP with options to display the output in customizable manner which was not previously possible.

3. Implement and compare (similarly to primary objectives) a new group of algorithms related to Maximal Flow, called Minimum Cut. ✓

   Minimum Cut was explored and Karger's Algorithm was implemented, as well as another optimised variation of this algorithm and was compared with the original algorithm.

4. Implement a working algorithm for The Travelling Salesman problem. ✗

   This was not attempted or implemented. I briefly looked into it but realised I would not have time to implement this. If I had more time, this would have been what I would have tackled next.

5. Integrate a selection of the implemented algorithms into the Digraphs package for GAP including appropriate documentation and tests in the standard style used by the GAP community. ✓

   After comparing the algorithms and finding which one was better depending on the situation, I implemented these into GAP and these have been suitably linted, tested and documented and await integration into the Digraphs package.

6. Explore an industry standard implementation of these algorithms (in Java, Python or C) and compare the performance to that of GAP. ✓

   This was also completed for the minimum spanning tree and shortest path group of problems. I did not do it for maximal flows because from the two previous groups I realised that the industry implementation of these algorithms were far superior to that of mine in GAP and much more heavily optimised.

Although I did not come up with the ideas for the algorithms myself, I am the first to write them in GAP and include constructors for edge weights which does not exist in GAP and opening the doors for a whole new set of digraph algorithms. When comparing to existing implementations (specifically Scipy) - I found that although my implementations in GAP are outperformed by by them that the general use of the algorithms to be similar. Example of this can be that Kruskal's algorithm is the best to use for the minimum spanning tree problems or that a use by case for Dijkstra and Bellman–Ford algorithm depending on edge weight negativity. There was also very little code for visualising graphs and subgraphs so I added some useful functions for this too giving the ability to more easily visualise the minimum spanning tree and

shortest path problems.

Overall, all the primary and all but one of the secondary goals were achieved. A total of 12 algorithms were implemented and this took a lot of work and time to learn about each of them and then implement and analyse them. I believe I have done an excellent job getting used to GAP and then learning about it to be able to implement and integrate the algorithms. I would have liked to attempt the Travelling Salesman problem as it seems very interesting but I did not have time and prioritised some of the other goals which I felt were more realistic to achieve. There are other minor objectives I realised throughout the project that I would have also liked to attempt, such as adding a new Fibonacci data structure to GAP which would have optimised some of the algorithms, such Prim's, Dijkstra's or Push–Relabel and provide a faster implementation.

Chapter 12

Conclusion

In summary, the project achieved all of its goals bar one secondary goal. There was a lot of work put into this project and some key achievements include implementing several algorithms belonging to different groups of problems, performing analysis on these and bench marking them to compare performances, as well as an industry standard implementation of these algorithms and then selecting an algorithm to implement into GAP accordingly. Many algorithms had similar time complexities and thus differed in the their run time which was compared across various types of graphs and reasons for selecting a specific algorithm were justified. This was all done in the GAP language, with the work due to be merged in for others to use. The algorithms were suitable profiled with various forms of testing including unit-test like tests and my own tests to test graphs of larger sizes. In addition, some work towards visualising the algorithms was done to make it easier to visualise the out of the minimum spanning tree and shortest paths algorithms.

The work I have done will allow for a whole new set of edge weighted algorithms to be implemented as well as use the algorithms I have implemented for their intended purpose and build upon them such as adding edge weight capabilities for the minimum cut algorithms. Future work could involve optimising the implementations further and try to match the performance of Scipy, or at least get as close to as possible to it. This could be achieved by implementing the algorithms in C or even implementing more efficient data structures such as the Fibonacci Heap, which could improve the run time complexity of Prim's algorithm from $O(|E|\log|V|)$ to $O(|E|+|V|\log|V|)$.

I have started work on the visualisation aspect in Digraphs, which, as of version 1.6.2 is quite sparse and has the potential for many more features and can build upon work I have started. Future work on the visualisation could involve highlighting other subgraphs such as maximal spanning trees, cycles, as well as the maximum flow algorithms. I have only implemented a small number of algorithms and there are many others that I did not even explore or that I am even aware of or even various properties of graphs such as a minimum spanning tree.

Directed graphs is a huge and broad field with a lot of theory that I have not encountered or am even aware of. This dissertation has only captured a small section of Digraphs, focusing on edge weighted algorithms, for which, there currently exists no implementation or support in the Digraphs package highlighting that there are

more types algorithms that could be implemented. Graphs with edge weights provide a lot of utility, providing the ability to represent various real life scenarios such as cost, distance, weight, entropy, energy.

Index

acyclic
  cycle, 15
adjacency list, 18
adjacency matrix, 18
augmenting path, 72

binary heap, 19
  max heap, 19
  min heap, 19
  priority queue, 19

complete binary tree, 19
complete graph, 18
connected graph
  disconnected, 16
cut, 83
cycle, 15

degree, 16
digraph, 15
disconnected
  connected graph, 16
disjoint sets, 32

edge, 14
edge contraction, 83
edge weight, 16
excess, 78

flow, 71
flow network, 71
forest, 32

graph, 14
greedy, 28

in degree, 16
in neighbour, 16

loop, 16

minimum-cut, 83

negative cycle, 54
neighbour, 16

ordered pair, 14
out degree, 16
out neighbour, 16

parallel edges, 30
path, 14
preflow, 78

residual capacity
  residual graph, 72
residual graph
  residual capacity, 72

simple, 18
  simple path, 18
strongly connected, 17
strongly connected component, 17
subgraph, 17
symmetric, 17

undirected graph, 14

vertex, 14
  node, 14

walk, 14
weighted graph, 16

Bibliography

[1] *15-451/651: Design & Analysis of Algorithms.* [Accessed 05-May-2023]. Oct. 2013. URL: https://www.cs.cmu.edu/~avrim/451f13/lectures/lect1010.pdf.

[2] E. Alfs-Votipka. *Prim :: CC 315 Textbook — textbooks.cs.ksu.edu.* https://textbooks.cs.ksu.edu/cc315/iii-graphs/9-graphs-minimum-spanning-trees/4-prim/. [Accessed 15-Apr-2023].

[3] J. Bang-Jensen and G. Z. Gutin. *Digraphs: theory, algorithms and applications.* Springer Science & Business Media, 2008.

[4] R. Bellman. "On a routing problem". In: *Quarterly of applied mathematics* 16.1 (1958), pp. 87–90.

[5] E. A. Bender and S. G. Williamson. *Lists, decisions and graphs.* S. Gill Williamson, 2010.

[6] J. D. Beule et al. *Digraphs - GAP package, Version 1.6.1.* Dec. 2022. URL: https://digraphs.github.io/Digraphs.

[7] N. Biggs, N. L. Biggs, and B. Norman. *Algebraic graph theory.* 67. Cambridge university press, 1993.

[8] R. Chowdhury. *Algorithms for Directed Graphs.* https://github.com/RaiyanC/CS5199-Proj-Code. 2023.

[9] *Code Studio — codingninjas.com.* https://www.codingninjas.com/codestudio/problem-details/negative-cycle-in-a-directed-graph_1090517. [Accessed 13-Apr-2023].

[10] *Compressed sparse graph routines (scipy.sparse.csgraph) 2014; SciPy v1.10.1 Manual — docs.scipy.org.* https://docs.scipy.org/doc/scipy/reference/sparse.csgraph.html. [Accessed 10-Apr-2023].

[11] T. H. Cormen et al. *Introduction to algorithms.* MIT press, 2022.

[12] G. David. "An introduction to combinatorics and graph theory". In: *Creative Commons* 543 (2013).

[13] E. W. Dijkstra. "A note on two problems in connexion with graphs". In: *Numerische mathematik* 1.1 (1959), pp. 269–271.

[14] Y. A. Dinitz. "An algorithm for the solution of the problem of maximal flow in a network with power estimation". In: *Doklady Akademii nauk.* Vol. 194. 4. Russian Academy of Sciences. 1970, pp. 754–757.

[15] *Disjoint Set Union - Algorithms for Competitive Programming — cp-algorithms.com.* https://cp-algorithms.com/data_structures/disjoint_set_union.html. [Accessed 05-Apr-2023].

[16] A. edgecontraction. *Lecture Notes CS:5350 Karger's Mincut Algorithm*. [Accessed 05-May-2023]. Oct. 2019. URL: https://homepage.cs.uiowa.edu/~sriram/5350/fall19/notes/10.17/10.17.pdf.

[17] J. Edmonds and R. M. Karp. "Theoretical improvements in algorithmic efficiency for network flow problems". In: *Journal of the ACM (JACM)* 19.2 (1972), pp. 248–264.

[18] R. W. Floyd. "Algorithm 97: shortest path". In: *Communications of the ACM* 5.6 (1962), p. 345.

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

[20] L. R. Ford Jr. *Network flow theory*. Tech. rep. Rand Corp Santa Monica Ca, 1956.

[21] *Ford-Fulkerson Algorithm | Brilliant Math Science Wiki — brilliant.org*. https://brilliant.org/wiki/ford-fulkerson-algorithm/. [Accessed 19-Apr-2023].

[22] *GAP – Groups, Algorithms, and Programming, Version 4.12.2*. The GAP Group. 2022. URL: %5Curl%7Bhttps://www.gap-system.org%7D.

[23] *GAP (datastructures) - Chapter 3: Heaps — docs.gap-system.org*. https://docs.gap-system.org/pkg/datastructures/doc/chap3_mj.html. [Accessed 03-Apr-2023].

[24] A. Gibbons. *Algorithmic graph theory*. Cambridge university press, 1985.

[25] *GitHub - digraphs/Digraphs: The GAP package Digraphs — github.com*. https://github.com/digraphs/Digraphs. [Accessed 22-Apr-2023].

[26] M. Goemans. *18.433: Combinatorial Optimization*. May 2007. URL: https://math.mit.edu/~goemans/18433S07/mincut.pdf.

[27] A. V. Goldberg and R. E. Tarjan. "A new approach to the maximum-flow problem". In: *Journal of the ACM (JACM)* 35.4 (1988), pp. 921–940.

[28] J. L. Gross and J. Yellen. *Graph theory and its applications*. [Accessed 21-Apr-2023]. CRC press, 2005.

[29] E. Gulsun. *Beauty of Karger's algorithm — towardsdatascience.com*. https://towardsdatascience.com/beauty-of-kargers-algorithm-6de7e923874a. [Accessed 29-Apr-2023].

[30] P. R. Halmos. *Naive set theory*. van Nostrand, 1960.

[31] D. Hochbaum. *Lecture Notes for IEOR 266: Graph Algorithms and Network Flows*. URL: https://hochbaum.ieor.berkeley.edu/files/ieor266-2014.pdf.

[32] D. B. Johnson. "Efficient algorithms for shortest paths in sparse networks". In: *Journal of the ACM (JACM)* 24.1 (1977), pp. 1–13.

[33] D. R. Karger. "Global Min-cuts in RNC, and Other Ramifications of a Simple Min-Cut Algorithm." In: *Soda*. Vol. 93. 1993, pp. 21–30.

[34] J. Kleinberg and E. Tardos. *Algorithm design*. Pearson Education India, 2006.

[35] J. B. Kruskal. "On the Shortest Spanning Subtree of a Graph and the Traveling Salesman Problem". In: *Proceedings of the American Mathematical Society* 7.1 (1956). [Accessed 15-Apr-2023], pp. 48–50. ISSN: 00029939, 10886826. URL: http://www.jstor.org/stable/2033241.

[36] *Language Basics x2014; Cython 3.0.0b2 documentation — cython.readthedocs.io.* https://cython.readthedocs.io/en/latest/src/userguide/language_basics.html. [Accessed 18-Apr-2023].

[37] *Merge Sort Algorithm - Java, C, and Python Implementation | DigitalOcean — digitalocean.com.* https://www.digitalocean.com/community/tutorials/merge-sort-algorithm-java-c-python. [Accessed 15-Apr-2023].

[38] J. Mestre. *3.1 the push-relabel algorithm - resources.mpi-inf.mpg.de.* URL: https://resources.mpi-inf.mpg.de/departments/d1/teaching/ws09_10/Opt2/handouts/lecture3.pdf.

[39] E. F. Moore. "The shortest path through a maze". In: *Proc. Int. Symp. Switching Theory, 1959.* 1959, pp. 285–292.

[40] J. Nešetřil, E. Milková, and H. Nešetřilová. "Otakar Borůvka on minimum spanning tree problem Translation of both the 1926 papers, comments, history". In: *Discrete mathematics* 233.1-3 (2001), pp. 3–36.

[41] R. C. Prim. "Shortest Connection Networks And Some Generalizations". In: *Bell System Technical Journal* 36.6 (1957), pp. 1389–1401. DOI: https://doi.org/10.1002/j.1538-7305.1957.tb01515.x. eprint: https://onlinelibrary.wiley.com/doi/pdf/10.1002/j.1538-7305.1957.tb01515.x. URL: https://onlinelibrary.wiley.com/doi/abs/10.1002/j.1538-7305.1957.tb01515.x.

[42] *profiling — gap-system.org.* https://www.gap-system.org/Packages/profiling.html. [Accessed 15-Apr-2023].

[43] Rosselliott. *Understanding Dinic's Algorithm — medium.com.* https://medium.com/smucs/understanding-dinics-algorithm-ebf892e66227. [Accessed 21-Apr-2023].

[44] *scipy/scipy/sparse/csgraph at main · scipy/scipy — github.com.* https://github.com/scipy/scipy/tree/main/scipy/sparse/csgraph. [Accessed 18-Apr-2023].

[45] R. Sedgewick and K. Wayne. *Algorithms (Fourth edition deluxe).* Addison-Wesley, 2016. ISBN: 978-0-1343-8468-9.

[46] *Tim Sort - javatpoint — javatpoint.com.* https://www.javatpoint.com/tim-sort. [Accessed 25-Apr-2023].

[47] *Time and Space Complexity of Kruskal's algorithm for MST — iq.opengenus.org.* https://iq.opengenus.org/time-and-space-complexity-of-kruskal-algorithm/. [Accessed 09-Apr-2023].

[48] S. Warshall. "A theorem on boolean matrices". In: *Journal of the ACM (JACM)* 9.1 (1962), pp. 11–12.

[49] S. Yegulalp. *What is Cython? Python at the speed of C — infoworld.com.* https://www.infoworld.com/article/3250299/what-is-cython-python-at-the-speed-of-c.html. [Accessed 18-Apr-2023].

Appendices

Appendix A

Ethics Form

This appendix includes the first page of the Ethics Document which was filled at the beginning of the semester.

UNIVERSITY OF ST ANDREWS
TEACHING AND RESEARCH ETHICS COMMITTEE (UTREC)
SCHOOL OF COMPUTER SCIENCE
PRELIMINARY ETHICS SELF-ASSESSMENT FORM

This Preliminary Ethics Self-Assessment Form is to be conducted by the researcher, and completed in conjunction with the Guidelines for Ethical Research Practice. All staff and students of the School of Computer Science must complete it prior to commencing research.

This Form will act as a formal record of your ethical considerations.
Tick one box

- ☐ **Staff Project**
- ☐ **Postgraduate Project**
- ☒ **Undergraduate Project**

Title of project

Algorithms for Directed Graph

Name of researcher(s)

Raiyan Chowdhury

Name of supervisor (for student research)

Michael Young

OVERALL ASSESSMENT (to be signed after questions, overleaf, have been completed)

Self audit has been conducted **YES** ☒ **NO** ☐

There are no ethical issues raised by this project

Signature Student or Researcher

Raiyan.C

Print Name

RAIYAN CHOWDHURY

Date

21-01-2000

Signature Lead Researcher or Supervisor

M. Young

Print Name

MICHAEL YOUNG

Date

23/01/2023

This form must be date stamped and held in the files of the Lead Researcher or Supervisor. If fieldwork is required, a copy must also be lodged with appropriate Risk Assessment forms. The School Ethics Committee will be responsible for monitoring assessments.

Appendix B

User Manual

This appendix includes the specific documentation written by me, extracted from the Digraphs manual regarding Edge Weights.

```
false
gap> gr2 := Digraph([[1, 2, 3], [3], [3]]);
<immutable digraph with 3 vertices, 5 edges>
gap> IsTransitiveDigraph(gr2);
true
gap> gr2 = DigraphTransitiveClosure(D);
true
gap> gr3 := Digraph([[1, 2, 2, 3], [3, 3], [3]]);
<immutable multidigraph with 3 vertices, 7 edges>
gap> IsTransitiveDigraph(gr3);
true
```

6.3 Edge Weights

6.3.1 EdgeWeights

▷ `EdgeWeights(`*`digraph`*`)` (attribute)
▷ `EdgeWeightsMutableCopy(`*`digraph`*`)` (operation)

Returns: A list of lists of integers, floats or rationals.

`EdgeWeights` returns the list of lists of edge weights of each edge of the digraph *`digraph`*. More specifically, a value `j` appears in `weights[i]` each time there exists the edge `[i, j]` in *`digraph`*.

The function `EdgeWeights` returns an immutable list of lists, whereas the function `EdgeWeightsMutableCopy` returns a copy of `EdgeWeights` which is a mutable list of mutable lists.

Example
```
gap> gr := EdgeWeightedDigraph([[2], [3], [1]], [[5], [10], [15]]);
<immutable digraph with 3 vertices, 3 edges>
gap> EdgeWeights(gr);
[ [ 5 ], [ 10 ], [ 15 ] ]
gap> a := EdgeWeightsMutableCopy(gr);
[ [ 5 ], [ 10 ], [ 15 ] ]
gap> a[1][1] := 100;
100
gap> a;
[ [ 100 ], [ 10 ], [ 15 ] ]
gap> b := EdgeWeights(gr);
[ [ 5 ], [ 10 ], [ 15 ] ]
gap> b[1][1] := 534;
Error, List Assignment: <list> must be a mutable list
```

6.3.2 EdgeWeightedDigraph

▷ `EdgeWeightedDigraph(`*`digraph, weights`*`)` (function)

Returns: A digraph or `fail`

`digraph` may be a Digraph or a list of lists of integers, floats or rationals. *`weights`* must be a list of lists of integers, floats or rationals of an equal size and shape to *`digraph`*, otherwise it will fail. This will create a EdgeWeightedDigraph and set the EdgeWeights to *`weights`*. See `EdgeWeights` (6.3.1).

Example
```
gap> g := EdgeWeightedDigraph(Digraph([[2], [1]]), [[5], [15]]);
<immutable digraph with 2 vertices, 2 edges>
gap> g := EdgeWeightedDigraph([[2], [1]], [[5], [15]]);
<immutable digraph with 2 vertices, 2 edges>
```

6.3.3 DigraphEdgeWeightedMinimumSpanningTree

▷ `DigraphEdgeWeightedMinimumSpanningTree(`*`digraph`*`)` (attribute)

Returns: A record.

This function returns the record with 2 components `total` and `mst`. The first component `total` represents the sum of the edge weights of the digraph that is returns. The second component `mst` is the edge weighted digraph representation of the mst.

This algorithm only works on connected undirected graphs. If it is given a disconnected digraph, it will error. The function will internally convert *`digraph`* representation to an undirected representation. See `EdgeWeights` (6.3.1).

Example
```
gap> g := EdgeWeightedDigraph([[2],[1],[1,2]], [[12],[5],[6,9]]);
<immutable digraph with 3 vertices, 4 edges>
gap> DigraphEdgeWeightedMinimumSpanningTree(g);
rec( mst := <immutable digraph with 3 vertices, 2 edges>, total := 11
 )
```

6.3.4 DigraphEdgeWeightedShortestPath

▷ `DigraphEdgeWeightedShortestPath(`*`digraph, start`*`)` (operation)

Returns: A record.

This operation, given a edge weighted *`digraph`* and a *`start`* vertex will return a record with 3 components. The first component is the distances which is a list of shortest distance to each node from the *`start`* node. The distance from the start node to itself is always 0. The second component is the edges, which signifies which edge was taken to get to that vertex from the parent of that node which is the third component; a list of vertices which are the parents of that vertex. Using both these components together, you can find the shortest edge weighted path to all other vertices from a starting vertex. In In cases, where a path doesn't exist and therefore there are no distances, edges or parents, the lists will contain a fail.

This operation can handle negative edge weights BUT it will error if a negative cycle exists.

See `EdgeWeights` (6.3.1).

Example
```
gap> g := EdgeWeightedDigraph([[2,3],[4],[4],[]],[[5,1],[6],[11],[]]);
<immutable digraph with 4 vertices, 4 edges>
gap> DigraphEdgeWeightedShortestPath(g,1);
rec( distances := [ 0, 5, 1, 11 ], edges := [ fail, 1, 2, 1 ],
  parents := [ fail, 1, 1, 2 ] )
gap> ncg := EdgeWeightedDigraph([[2],[3],[1]],[[-1],[-2],[-3]]);
<immutable digraph with 3 vertices, 3 edges>
gap> DigraphEdgeWeightedShortestPath(ncg,1);
Error, negative cycle exists,
```

6.3.5 DigraphEdgeWeightedShortestPaths

▷ `DigraphEdgeWeightedShortestPaths(digraph)` (attribute)

Returns: A list of lists of integers, floats or rationals.

Given an edge weighted *digraph*, this returns a list of lists of the shortest distance from one vertex to every other vertex. If no paths exist, then fail will be returned in the 2D list. This will return an incorrect answer if negative cycles exists. See `EdgeWeights` (6.3.1).

Example
```
gap> g := EdgeWeightedDigraph([[2],[3],[1]],[[1],[2],[3]]);
<immutable digraph with 3 vertices, 3 edges>
gap> DigraphEdgeWeightedShortestPaths(g);
rec( distances := [ [ 0, 1, 3 ], [ 5, 0, 2 ], [ 3, 4, 0 ] ],
  edges := [ [ fail, 1, 1 ], [ 1, fail, 1 ], [ 1, 1, fail ] ],
  parents := [ [ fail, 1, 1 ], [ 2, fail, 2 ], [ 3, 3, fail ] ] )
```

6.3.6 DigraphMaximumFlow

▷ `DigraphMaximumFlow(digraph, start, destination)` (attribute)

Returns: A record.

Given an edge weighted *digraph*, this returns a record with 3 components. The first component is the flow inbound into vertex v which is a list of lists. If there are multiple edges, the algorithm will fill up the edges sequentially so if there are 3 edges outbound from u to v with capacities, 5,10,15 and there is a flow of 15, it will fill the first two edges 5 and 10. If there is a flow of 9, then the flow will contain a list with flows 5 and 4.

This can be coupled with the second component which is a list of list of the vertices that each flow comes from. Using this, allows the path of the flow and the flow to be obtained using the first component.

The third and last component is the maximum flow value which is the highest flow that we can obtain from start to destination.

See `EdgeWeights` (6.3.1).

Example
```
gap> g := EdgeWeightedDigraph([[2,2],[3],[]],[[3,2],[1],[]]);
<immutable multidigraph with 3 vertices, 3 edges>
gap> DigraphMaximumFlow(g, 1, 3);
rec( flows := [ [  ], [ 1, 0 ], [ 1 ] ], maxFlow := 1,
  parents := [ [  ], [ 1, 1 ], [ 2 ] ] )
```

6.3.7 RandomUniqueEdgeWeightedDigraph

▷ `RandomUniqueEdgeWeightedDigraph([filt, ]n[, p])` (attribute)

Returns: An edge weighted digraph.

If the optional first argument *filt* is present, then this should specify the category or representation the digraph being created will belong to. For example, if *filt* is `IsMutableDigraph` (3.1.2), then the digraph being created will be mutable, if *filt* is `IsImmutableDigraph` (3.1.3), then the digraph will be immutable. If the optional first argument *filt* is not present, then `IsImmutableDigraph` (3.1.3) is used by default.

As well as the filters implemented in `RandomDigraph` (3.4.1), the following filters are implemented: `IsStronglyConnectedDigraph` (6.6.6). For `IsStronglyConnectedDigraph` (6.6.6), first

a random connected tree is created which it self may have numerous strongly connected components (scc) which are then them selves connected. For each sequential pair of strongly connected component , a random u from the first scc and v from the second scc and given a directed edge from u to v. This is then repeated with an edge from a random vertex in the second scc to the first scc. If *n* is a positive integer, then this function returns a random edge weighted digraph with *n* vertices, without multiple edges but with unique edge weights. The result may or may not have loops. If using `IsAcyclicDigraph` (6.6.1), the resulting graph will not have any loops by definition.

If the optional second argument *p* is a float with value $0 \le p \le 1$, then an edge will exist between each pair of vertices with probability approximately *p*. If *p* is not specified, then a random probability will be assumed (chosen with uniform probability).

Example

```
gap> g := RandomUniqueEdgeWeightedDigraph(
>      IsStronglyConnectedDigraph, 5, 1);
<immutable digraph with 5 vertices, 25 edges>
gap> g := RandomUniqueEdgeWeightedDigraph(5, 1);
<immutable digraph with 5 vertices, 25 edges>
```

6.3.8 DigraphFromPaths

▷ `DigraphFromPaths(digraph, record)` (attribute)

Returns: An edge weighted digraph.

Given a *digraph* and a *record* of distances, edges and parents this will compute the start vertex and will build a digraph of the shortest path from the start vertex to all other vertices.

Example

```
gap> g := EdgeWeightedDigraph([[2],[3],[]],[[2],[1],[]]);
<immutable digraph with 3 vertices, 2 edges>
gap> sp := DigraphEdgeWeightedShortestPath(g, 1);
rec( distances := [ 0, 2, 3 ], edges := [ fail, 1, 1 ],
  parents := [ fail, 1, 2 ] )
gap> sd := DigraphFromPaths(g, sp);
<immutable digraph with 3 vertices, 2 edges>
```

6.3.9 DigraphFromPath

▷ `DigraphFromPath(digraph, record, dest)` (attribute)

Returns: An edge weighted digraph.

Given a *digraph* and a *record* of distances, edges and parents this will compute the start vertex and will build a digraph of the shortest path from the start vertex to *dest* vertex.

Example

```
gap> g := EdgeWeightedDigraph([[2],[3],[]],[[2],[1],[]]);
<immutable digraph with 3 vertices, 2 edges>
gap> sp := DigraphEdgeWeightedShortestPath(g, 1);
rec( distances := [ 0, 2, 3 ], edges := [ fail, 1, 1 ],
  parents := [ fail, 1, 2 ] )
gap> sd := DigraphFromPath(g, sp, 3);
<immutable digraph with 3 vertices, 2 edges>
gap> sd := DigraphFromPath(g, sp, 2);
<immutable digraph with 3 vertices, 1 edge>
```

6.3.10 DotEdgeWeightedDigraph

▷ DotEdgeWeightedDigraph(*digraph, subdigraph, record*) (attribute)

Returns: A string.

Given an*digraph*, *subdigraph* and a *record* of a subdigraph within the original digraph, using the record optional parameters, this will return a DOT of the subdigraph within the original digraph.

Optional parameters in the *record* include: - highlightColour (default blue): the colour of the path of the subdigraph - edgeColour (default black): the colour of the non subdigraph path - vertColor (default lightpink): the colour of the vertices - sourceColour (default green): the colour of a source vertex - destColour (default red): the colour of a destination vertex An empty record may be passed as a parameters, in which case the default values will be used.

Example
```
gap> g := EdgeWeightedDigraph([[2],[3],[]],[[2],[1],[]]);
<immutable digraph with 3 vertices, 2 edges>
gap> sp := DigraphEdgeWeightedShortestPath(g, 1);
rec( distances := [ 0, 2, 3 ], edges := [ fail, 1, 1 ],
  parents := [ fail, 1, 2 ] )
gap> sd := DigraphFromPath(g, sp, 3);
<immutable digraph with 3 vertices, 2 edges>
gap> DotEdgeWeightedDigraph(g, sd, rec());
"//dot\ndigraph hgn{\nnode [shape=circle]\n1[color=lightpink, style=fi\
lled]\n2[color=lightpink, style=filled]\n3[color=lightpink, style=fill\
ed]\n1 -> 2[color=blue, label=2]\n2 -> 3[color=blue, label=1]\n}\n"
```

6.4 Orders

6.4.1 IsPreorderDigraph

▷ IsPreorderDigraph(*digraph*) (property)

▷ IsQuasiorderDigraph(*digraph*) (property)

Returns: true or false.

A digraph is a preorder digraph if and only if the digraph satisfies both IsReflexiveDigraph (6.2.13) and IsTransitiveDigraph (6.2.16). A preorder digraph (or quasiorder digraph) *digraph* corresponds to the preorder relation $\leq$ defined by $x \leq y$ if and only if [x, y] is an edge of *digraph*.

If the argument *digraph* is mutable, then the return value of this property is recomputed every time it is called.

Example
```
gap> D := Digraph([[1], [2, 3], [2, 3]]);
<immutable digraph with 3 vertices, 5 edges>
gap> IsPreorderDigraph(D);
true
gap> D := Digraph([[1 .. 4], [1 .. 4], [1 .. 4], [1 .. 4]]);
<immutable digraph with 4 vertices, 16 edges>
gap> IsPreorderDigraph(D);
true
gap> D := Digraph([[2], [3], [4], [5], [1]]);
<immutable digraph with 5 vertices, 5 edges>
gap> IsPreorderDigraph(D);
false
gap> D := Digraph([[1], [1, 2], [2, 3]]);
```

Appendix C

Minimum Spanning Tree Algorithm Plots

This appendix includes all the extra plots for the minimum spanning tree algorithms.

Figure C.1: Prim's Algorithm on various graph densities

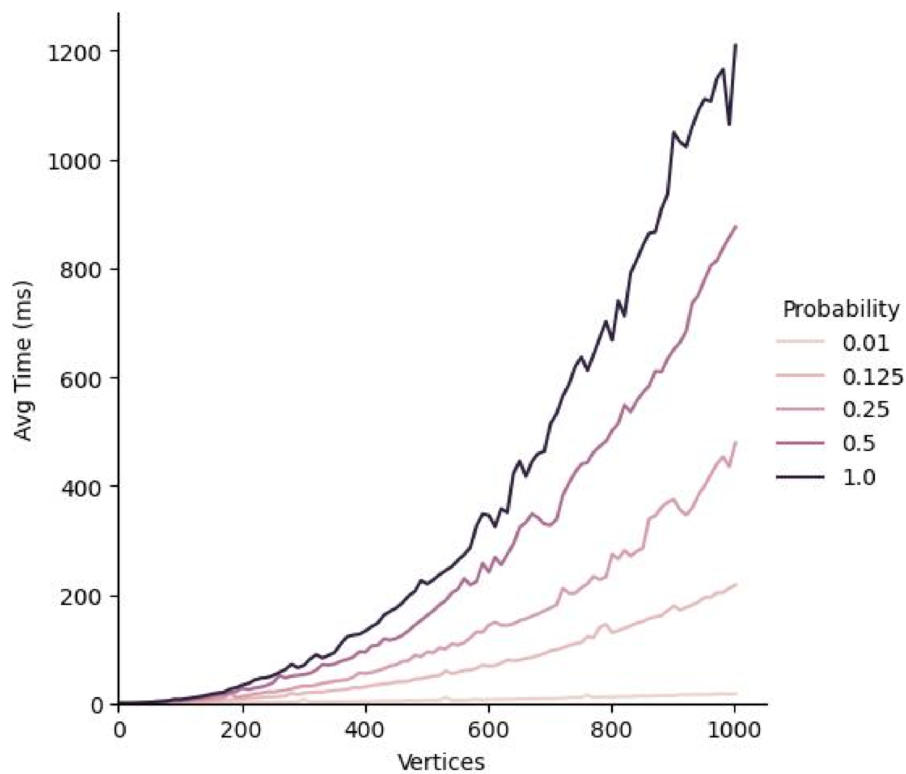


Figure C.2: Borůvka's Algorithm on various graph densities

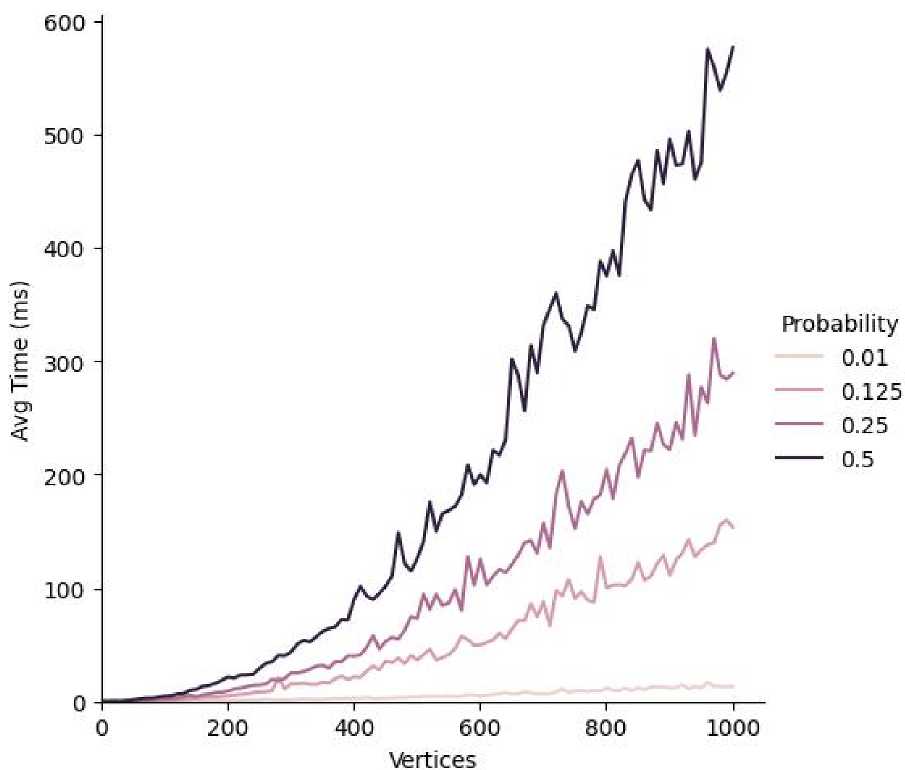
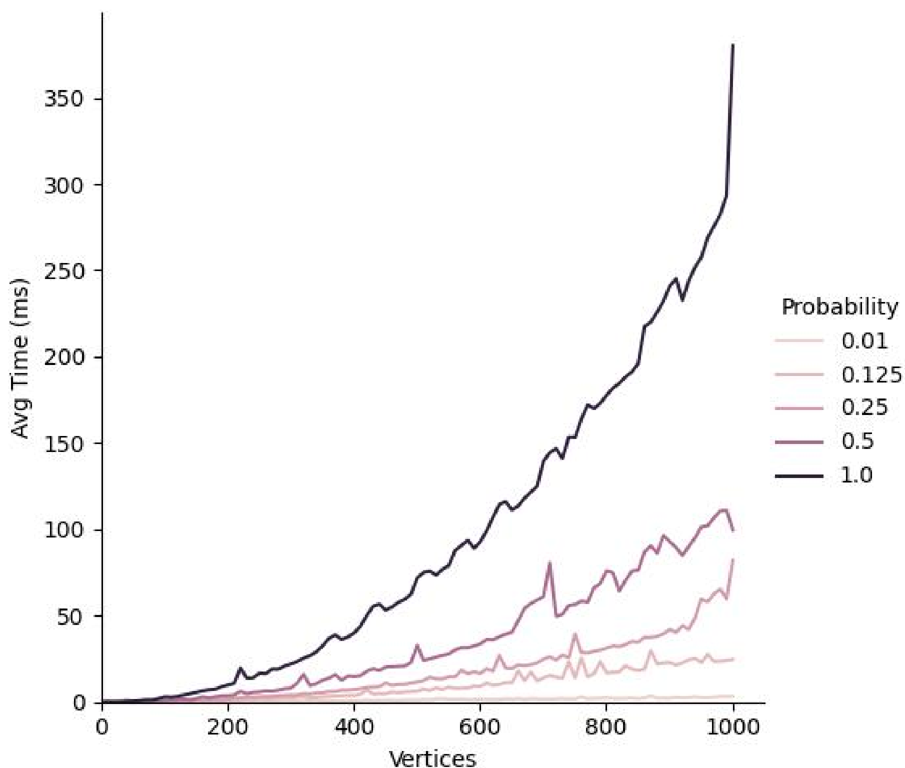


Figure C.3: Kruskal's Algorithm (with path compression) on various graph densities



Appendix D

Shortest Path Algorithm Plots

This appendix includes all the extra plots for the maximal flow algorithms.

Figure D.1: Dijkstra's Algorithm on various graph densities

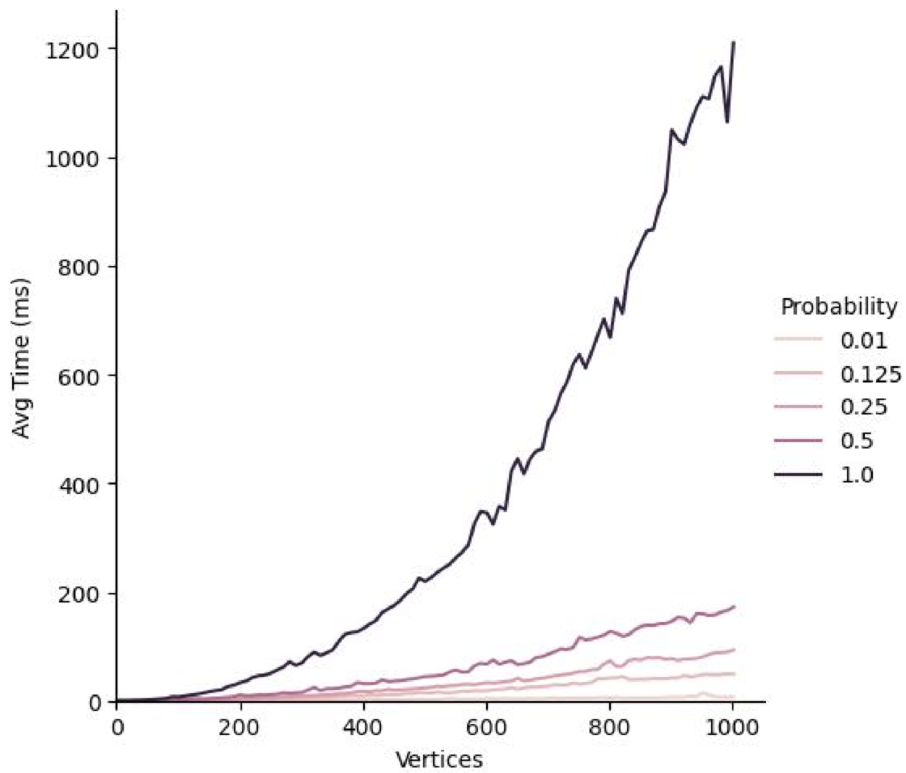


Figure D.2: Bellman–Ford Algorithm (optimised) on various graph densities

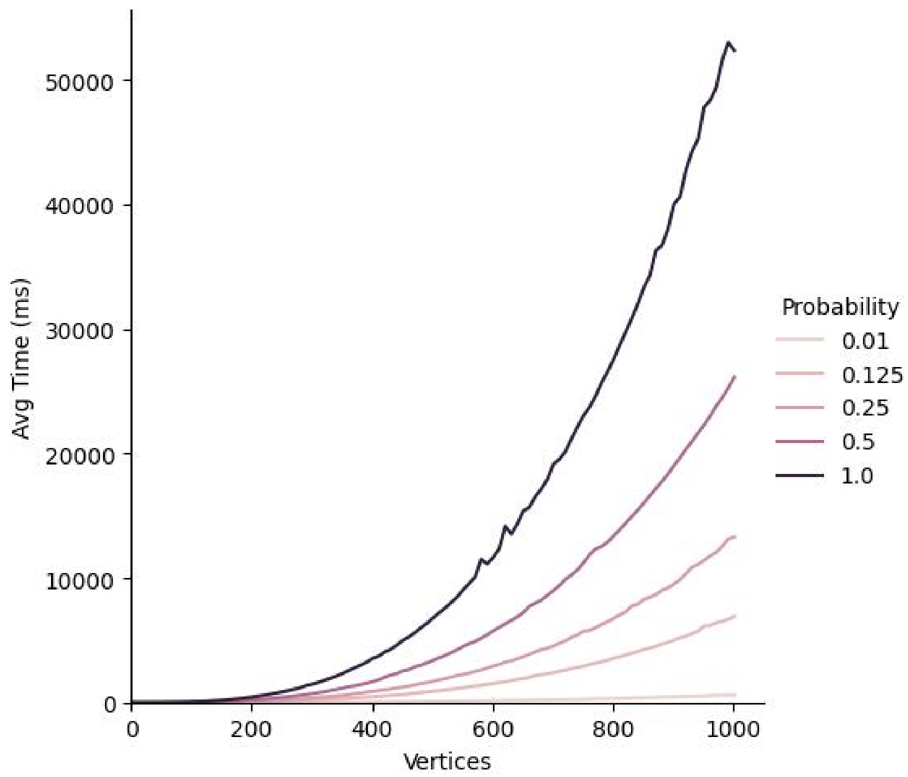
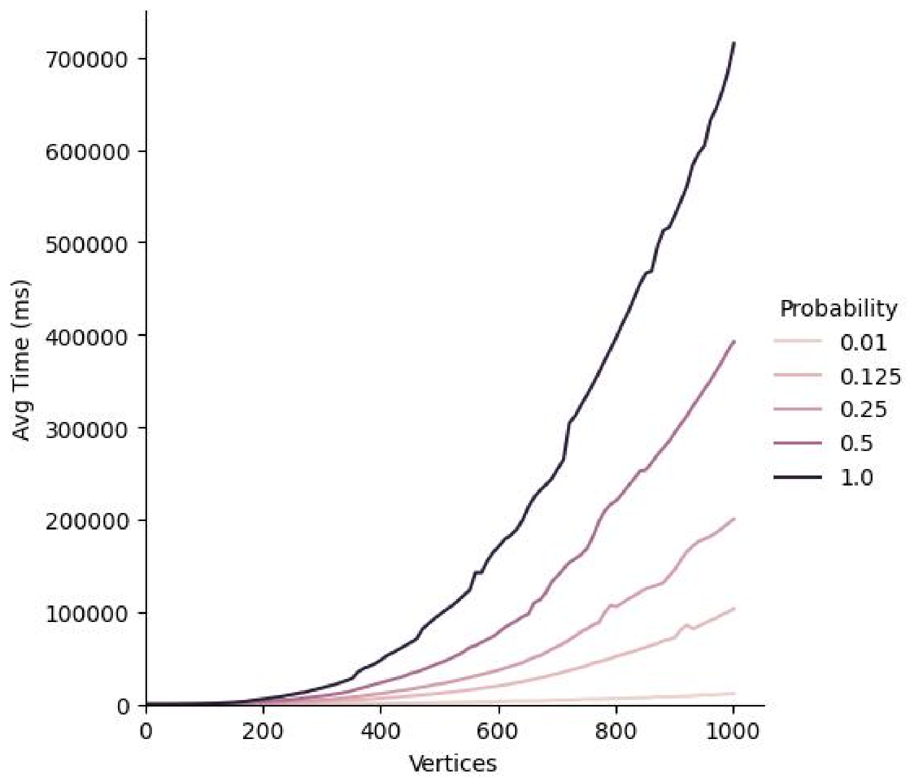


Figure D.3: Johnson's Algorithm on various graph densities



Appendix E

Maximal Flow Algorithm Plots

Figure E.1: Edmonds–Karp Algorithm on various graph densities

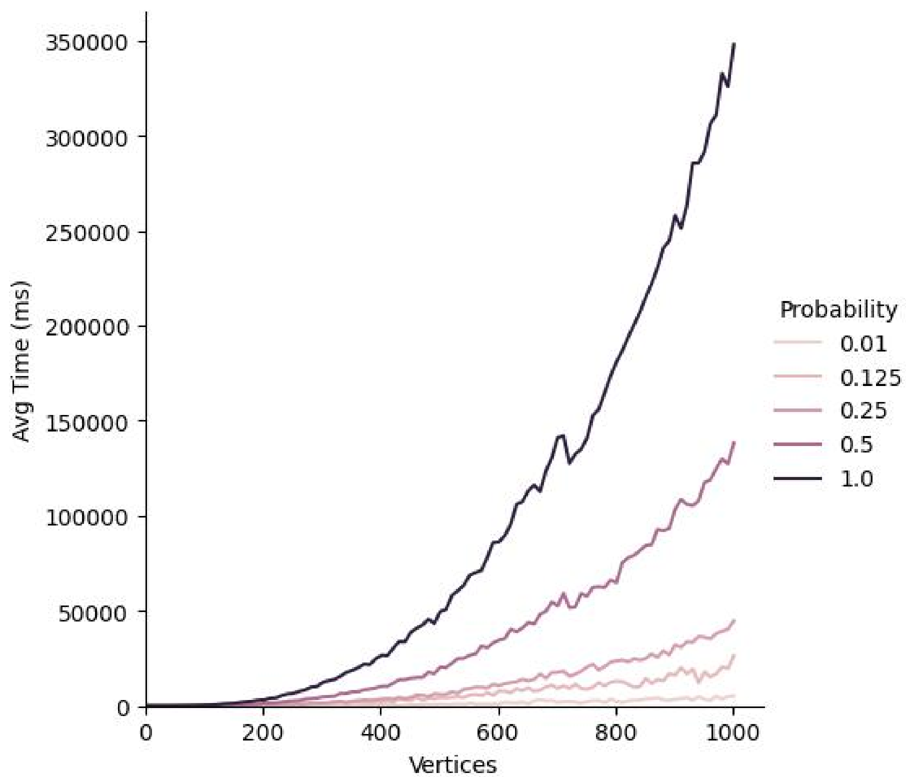


Figure E.2: Dinic's Algorithm on various graph densities