Algorithms for Directed Graphs

180005418
- University of St Andrews -
School of Computer Science

13th February 2023

Abstract

TODO: Outline of the project using at most 250 words.:

Declaration

Contents

List of Figures

Chapter 1

Introduction

History of Graphs

Definition 1.0.1 (Node)**.** A **node** is a point. This is also called a **vertex**.

Definition 1.0.2 (Edge)**.** An **edge** is a line that connects nodes together.

Definition 1.0.3 (Graph)**.** A **graph** consists of 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. [4, Chapter 5.1]

Definition 1.0.4 (Walk)**.** A **walk** in a digraph is a sequence $v_1, e_1, v_2, e_2, ..., v_k - 1, e_k - 1, v_k$. If $v_1 = v_k$, it is a closed walk or a circuit. [4, Chapter 5.11].

Definition 1.0.5 (Path)**.** A **path** in a graph is a walk in which all vertices are distinct. If there is a walk from v to w then there is a path from v to w. [4, Chapter 5.11].

Definition 1.0.6 (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 $u = v$.

Definition 1.0.7 (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 $v \to u$.

Definition 1.0.8 (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. [4, Chapter 4.4].

Definition 1.0.9 (Loop)**.** A **loop** is a cycle with one vertex where the single vertex serves as both end points. [4, Chapter 4.4].

Definition 1.0.10 (Edge Weight)**.** An **edge weight** is a weight (also cost or value) associated with an edge in a graph. Such a graph is called a **weighted graph**.

Definition 1.0.11 (Neighbour)**.** A **neighbour** is a vertex adjacent to another vertex (if a path of length 1 exists between them).

Definition 1.0.12 (Out Neighbour)**.** In a directed graph D, all the neighbours outbound from a vertex are considered its **out neighbour**.

Definition 1.0.13 (In Neighbour)**.** In a directed graph D, a vertex is an **in neighbour** if other vertices are directed towards it.

Definition 1.0.14 (Degree)**.** The **degree** of a vertex v is the number of edges incident with v.

Definition 1.0.15 (In-degree)**.**

Definition 1.0.16 (Out-degree)**.**

Chapter 2

Context Survey

I will be working to implement various algorithms in GAP [5] which is a 'system for computational discrete algebra, with a particular emphasis on Computational Group Theory'. GAP provides a programming language, as well as a library of thousands of functions implementing algebraic algorithms which are also written in gap. I will be focusing on a specific library in GAP; mainly the digraphs package. The algorithms I will implement will go into this package. As of now, there are no edge weighted graphs in GAP, so all code I will be writing will be new to GAP.

Literature Review As I have already started writing the report, I have discovered some literature that are useful for describing the type of algorithms and graph theory I am implementing.

Work Plan This is a rough outline of my plan for this project. As of writing this (13th February 2023), it is the start of week 5 and I have implemented 6 of the 7 algorithms outlined in the primary goals. This includes, 2 Minimum Spanning Tree algorithms, 3 shorted path algorithms and 1 out of the 2 required maximum flow algorithms.

1. Week 5 (wc. 13th Feb 2023) - In this week I plan to finish all the algorithms I originally set out to implement. As of now, this means implementing the final maximum flow algorithm.

2. Week 6 (wc. 20th Feb 2023) - In this week, I plan to begin the analysis and comparison of the groups of algorithms; finding strengths and weakness and any optimisations possible. I have already realised some changes and optimisations with prior algorithms implemented. This is why I left this analysis till the end, as to allow myself to learn GAP better and change previous implementations as I discovered other graph properties and implementation details.

3. Week 7 (wc. 6th March 2023) - This week I plan to hopefully, start integrating my code into the digraph packages with the proper documentation.

4. Week 8. (wc. 13th March 2023) - This week, I will hopefully move onto my secondary goals.

Chapter 3

Requirements Specification

TODO: Capturing the properties the software solution must have in the form of requirements specification. You may wish to specify different types of requirements and given them priorities if applicable. :

Chapter 4

Software Engineering Process

TODO: The development approach taken and justification for its adoption.:

Chapter 5

Ethics

TODO: Any ethical considerations for the project. You should scan the signed ethical approval document, and include it as an appendix.:

Chapter 6

Design and Implementation

In this section, as per the primary goals, I will give an introduction to each category of algorithm, provide some context and then talk about the algorithms and their design and implementation.

6.1 Representation of Graphs

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

Definition 6.1.1 (Adjacency Matrix)**.** [2, Chapter 2] An **adjacency matrix** of τ is the n x n matrix $A = A(\tau)$ whose entries a_{ij} are given below by

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

Definition 6.1.2 (Adjacency List)**.**

Comparison Although, both ways are valid ways of representing graphs, they are better in different scenarios. The 'adjacency-list representation provides a compact way to represent **sparse** graphs' [3] - which are graphs for which $|E|$ is much less than $|V^2|$. Adjacency matrices may be better suited for **dense** graphs in which $|E|$ is close to $|V^2|$ and one needs to know quickly, if an edge connects two given vertices or not.

6.2 Spanning Trees

Before describing the implemented algorithms, it will be useful to describe what spanning trees are, and therefore minimum spanning trees. A spanning tree is an acyclic graph that connects all vertices together forming a tree with the minimal number of edges. Figure 6.1 shows three similar graphs (with no edge weights), in which the red paths highlight possible different spanning trees.

6.3 Minimum Spanning Trees (MST's)

Minimum spanning trees are similar to spanning trees, except the edges have weights and we try to minimize the sum of the cost / weight of the edges. If we have an

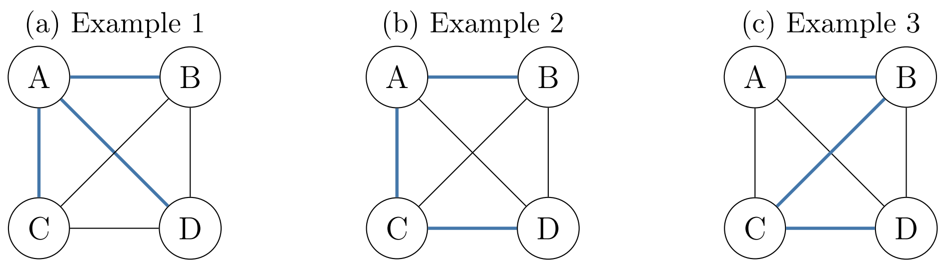
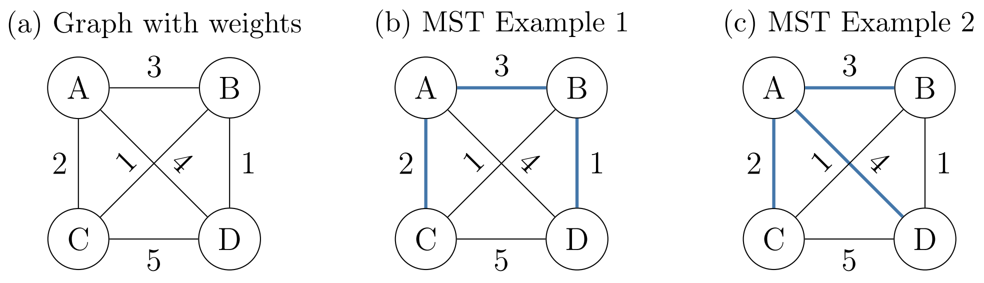


Figure 6.1: Examples of Spanning Trees

undirected graph *G = (V,E)*, where V are the vertices and E and are the edges; each with a weight *w(u, v)*, we want the minimum spanning tree T to be minimised [3, p 585]. Using the same graph as above, except this time with edge weights, we can see the possible MST's that are possible.



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

Figure 6.2: Examples of Minimum Spanning Trees

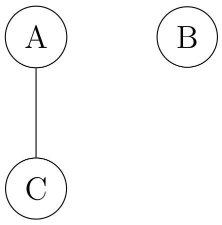
$$w(T) = min(\sum_{(u,v)\in T} w(u, v)) \tag{6.1}$$

There are some properties of the graph are required and others which do not matter for finding a minimum spanning tree which I describe below.

MST Graph Properties

- **Connected graph** - the graph needs to be connected (defined. for each pair of vertices, there exists at least one single path that connects them), otherwise it is impossible to find the minimum spanning tree of the graph (as shown by Figure 6.3). There is no path from either A or C to B, so a spanning tree and therefore a minimum spanning tree cannot be obtained.
- **Edge Weight Polarity** - the edge weights may be positive or negative and the MST can still be obtained.
- **Edge Weight Uniqueness** - If all the edges have unique weights, then there will be only one, unique MST. Figure 6.2 demonstrates that with multiple same edge weights, there may be multiple minimum spanning trees.

Figure 6.3: Example of a Disconnected Graph



6.3.1 Prim's Algorithm

Prim's algorithm [3] is the first algorithm that we will explore. It is a greedy algorithm, by which we mean it always selects the shortest edge first of the vertex it is currently looking at that doesn't create a cycle and repeats this until all vertices are connected. Using the example from above Figure 6.4 shows the steps in Prim's algorithm. We choose an arbitrary node, in this example and in my implementation we start from the first vertex (Vertex 1 / Vertex A) and choose the lowest weighted edge from that node. In our example, Figure 6.4b shows the edge highlighted in green (with weight 1) being chosen as it is the lowest weighted edge from A. All vertices, that edges are considered from are highlighted in cyan. Once we add the edge (A,D) to our MST and A to visited set, we then consider all edges from A and D and again choose the lowest weighted edge; in Figure 6.4c this is the edge (D,B); we repeat this until all the vertices have been visited. We only consider edges that don't form a cycle which is achieved by checking the out neighbour is not in the visited set. In Figure 6.4d, the edge (A,B) is checked but as A is already in our visited set, the edge (A,B) is ignored. The final MST achieved by Prim's algorithm is shown be Figure 6.4g.

Below is a more formal pseudo-code version of Prim's algorithm [3].

Algorithm 1 Prim's Algorithm

```
1: procedure MSTPrim(G)
2:     $p \leftarrow$ priority_queue
3:     for all v in neighbours(root) do
4:         $p \leftarrow p + (w(root, v), root, v)$
5:     $total \leftarrow 0$
6:     $visited \leftarrow SET$
7:     while $pq$ do
8:         $edge \leftarrow$ priority_queue.pop()
9:         if $edge.v \notin visited$ then
10:            $visited \leftarrow visited + edge.v$
11:            $total \leftarrow total + node.w$
12:            for all v in neighbours(edge.u) do
13:                $next\_v \leftarrow neighbours(v)$
14:                $p \leftarrow p + (w(v, next\_v), v, next\_v)$
15:    return total
```

Implementation Now that we know how Prim's algorithm works, we need to be able to implement this in gap. The first step is to model the graph. In GAP, we can

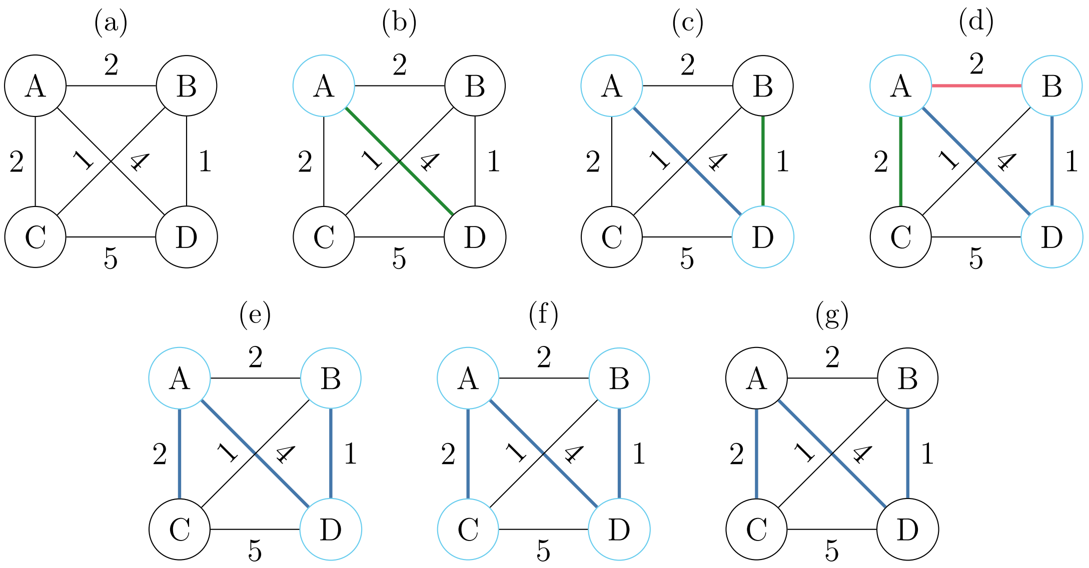


Figure 6.4: Example of Prim's algorithm

```
graph := Digraph([[2,3,4], [1,3,4], [1,2,4], [1,2,3]]);
weights := [[2,2,1], [2,1,1], [2,1,5],[1,1,5]];
function_name(graph, weights);
```

Figure 6.5: Code for creating graph and weights

make a digraph object / constructor, however, there is no implementation for edge weights. As I built my algorithms, the function takes in the digraph, and a nested list of weights. Figure 6.5 shows a snippet of how the graph in Figure 6.2a is created.

The first nested list shows all the nodes that connect to vertex 1 (from the diagrams, this is vertex A); indexing in GAP starts at 1. So vertex 1 connects to nodes 2,3,4. Vertex 2 connects to vertex 1,3,4 and the weights are the direct association of these. So, for the first vertex 1 and its edge to vertex 2 (edge(1, 2), edge(A,B) in the diagrams) have weight 2. Another example is vertex 4 and vertex 3 (edge(D,C)) has weight 5.

6.3.2 Kruskal's algorithm

6.3.3 Analysis

6.3.4 Real World Applications

6.4 Shortest Path Algorithms

6.4.1 Dijkstra's Algorithm

6.4.2 Bellman-Ford's Algorithm

6.4.3 Floyd-Warshall Algorithm

6.4.4 Analysis

6.5 Maximal Flow Algorithms

6.5.1 Analysis

Chapter 7

Evaluation and critical appraisal

TODO: You should evaluate your own work with respect to your original objectives. You should also critically evaluate your work with respect to related work done by others. You should compare and contrast the project to similar work in the public domain, for example as written about in published papers, or as distributed in software available to you. :

Chapter 8

Conclusion

TODO: You should summarise your project, emphasising your key achievements and significant drawbacks to your work, and discuss future directions your work could be taken in. :

Chapter 9

Dummy Chapter

Main book..[1]

9.1 Dummy Section

New dummy section

9.2 Drawing graphs test

New dummy page

Appendix A

Appendix

Index

Bibliography

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

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

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

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

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