Applications of Three Common Graph Theoretical Algorithms

Upasana Chatterjee
upcha1997@gmail.com

under the direction of
Dr. Marlena Nowaczyk
Department of Mathematics
Stockholm University

Research Academy for Young Scientists
July 10, 2014

Abstract

This paper explored three common graph theoretic algorithms: the Ford-Fulkerson algorithm, Dijkstra's algorithm, an extension of Dijkstra's algorithm for weighted graphs and the Critical Path method. For each of these algorithms, large scale applications are explained through the use of multiple examples. This paper concludes by discussing the applications of these algorithms on a type of digital network topology called the mesh network.

Contents

1 Introduction

Graphs have been used to serve many purposes, primarily that of showing relationships between multiple sets of information. In 1736, Leonhard Euler revolutionized the field by publishing a paper on the *Königsberg Bridge Problem* [1]. His paper used the relationship between vertices and edges to solve a problem rather than simply illustrate a relationship. This new way of using a graph developed into graph theory. Due to its applicability, graph theory is not only limited to the theoretical sciences where graphs have been used to represent molecules and their relationships within chemistry, and the same for atoms within physics. It has also been applied in the social sciences to show interpersonal relationships, information systems, transport systems, and more.

This paper will explore three common graph theoretic algorithms−the Ford-Fulkerson algorithm, Dijkstra's algorithm and the Critical Path method−and solve common problems related to them. An extension of Dijkstra's algorithm, and a large scale application of the critical path method will also be discussed. The paper will conclude with a section on mesh networks which is an area of burgeoning interest to mathematicians.

1.1 Famous Graph Theoretic Problems

There are two fundamental problems within graph theory, the Chinese postman problem and the travelling salesman problem, which have fascinated mathematicians for centuries. These problems are based on Hamiltonian and Eulerian cycles, which form the fundamental base of graph theory, and are used to represent some of the basic problems which graph theoretic algorithms are used to solve.

The *Chinese postman problem* is based on the premise of a postman who has a particular neighbourhood, the graph G, where he must deliver letters to all the houses, while travelling through all the roads, i.e. edges, connecting the houses and returning to the initial starting point. The postman's objective is to find the shortest path through all of the edges. A graphical representation of a Chinese postman problem is seen in Figure 1.1.

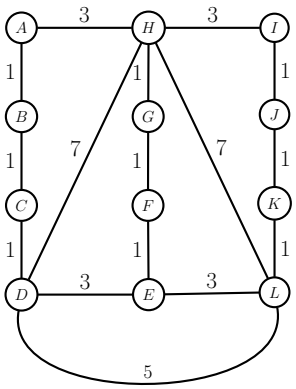


Figure 1.1: A simple example of the Chinese postman problem.

In this graph, the distances between the houses may be denoted as weights on the edges. The problem may also be more simply stated as finding the shortest cycle through G where each edge is used at least once. Given an Eulerian graph, any Eulerian cycle will give the optimal answer. In this graph, there is an Eulerian path so the postman will have start and end his journey at either E or H. Eulerian Paths are further explained in section 2.1. Given a non-Eulerian graph, the problem may be solved using Dijkstra's algorithm which will be introduced in section 2.2.

The *travelling salesman problem* is based on the premise of a salesman who wants to visit a given number of cities, or vertices, and return to his starting point in such a way that he covers the least possible total distance, but visits each vertex at least once. This problem can be restated in the form of a weighted graph, where the weights represent distances and the task is to find a Hamiltonian path of least weight. There is no direct algorithm which solves this problem as its complexity increases with the number of vertices.

2 Foundations of Graph Theory

To begin with, some basic concepts of graph theory such as simple graphs, Eulerian and Hamiltonian paths and cycles, flow networks and task graphs will be introduced.

Within graph theory, the *graph*, G, is defined as a collection of points called *vertices*

and denoted as V, connected by lines called *edges* and denoted as E, so that $G = (V, E)$. There are also special graphs, which differ based on the relationship between their nodes and edges. This paper will be based on the simple graph, the flow network and the task graph.

The *simple graph* has only single edges connecting any two vertices together. We can see that that the graph in Figure 2.1 is a simple graph because there is only one edge connecting any two vertices.

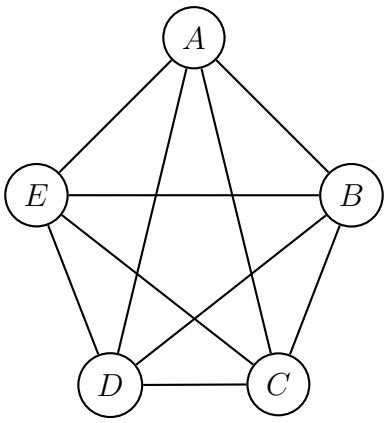


Figure 2.1: A simple, complete graph, with 5 vertices and 10 edges.

Graphs also contain paths and cycles, which can be used to represent several different situations. Thus most graph theoretic algorithms today derive from problems related to paths and cycles. A *path* is a sequence of neighbouring vertices $V_0, V_1, \ldots, V_n$, while a *cycle* is a path where the initial vertex V_0 and the final vertex V_n are the same.

2.1 Hamiltonian and Eulerian Paths and Cycles

Hamiltonian and Eulerian paths and cycles solidify the idea that the relationship between vertices and edges can be used to indicate a variety of situations, as seen in the Chinese postman problem and travelling salesman problem. We can build a graph where the vertices represent bus depots and the edges being the routes between them. Using that graph we may find methods to e.g. optimize the time it takes to get from one bus depot to another. Hamiltonian and Eulerian paths and cycles are also some of the earliest

applications of graph theory, and many algorithms build on them.

A *Hamiltonian path* is defined as one where each vertex is visited exactly once. The graph presented in Figure 2.1 is an example of a complete graph i.e. there is an edge between each pair of vertices. This means that when calculating a path there are 5 choices for V_0 and 4 for V_1 and so on until there is only one choice left for V_4. Thus, the total number of Hamiltonian paths must be $5 \cdot 4 \cdot 3 \cdot 2 \cdot 1 = 5! = 120$. Given the definition of a cycle, we can also say that there are 120 Hamiltonian cycles in the graph, as a Hamiltonian cycle can be truncated to form a Hamiltonian path. For example, in the cycle A, B, C, D, E, A, the last vertex can be removed to give the path A, B, C, D, E. Thus, considering an arbitrary complete graph of n vertices, the number of Hamiltonian paths present is $n!$, and this is also the number of Hamiltonian cycles present.

In Figure 2.1 we can also see an *Eulerian cycle* $A, B, C, D, E, B, D, A, C, E, A$ where each edge is used exactly once and which ends at the initial vertex. If the last vertex, A is truncated, it will be an *Eulerian path* instead. In the graph presented in Figure 2.1 there are 2640 possible Eulerian cycles. An explanation of how this result was calculated is shown in Appendix A.

Determining if a graph has Eulerian paths or cycles can be done by finding the degrees of the vertices in the graph. A degree of a vertex is defined as the number of edges incident to the vertex.

Theorem 1. *A connected graph is Eulerian (i.e. has an Eulerian cycle) if and only if the degree of every vertex is even.*

Proof. Assume that there is an Eulerian path p which travels through a graph. Every time an edge passes through a vertex, it both enters and leaves it. Thus we can say that $deg(V) = n$, where $n = 2m$ for some m. Thus, as in p each edge is traversed exactly once, it can be seen that each vertex must have an even degree. This condition can only be fulfilled if the initial vertex v_1 and the final vertex v_n are connected, giving us an Eulerian cycle.

Similarly, we can prove that if the degree of every vertex is even, we must have an Eulerian cycle. This can be done using a inductive method where we have a graph with 2 edges and 2 vertices. It is clear that this graph must be Eulerian. Given a graph with V vertices and $n > 2$ edges, suppose we start at an arbitrary vertex v and travel along arbitrary edges until we return to v. This is possible as when we reach a particular vertex, we have accounted for one edge incident to it and as there is an even number of edges we will eventually return to the starting point, v. The cycle that has been traversed can be denoted as M, where the set of edges on M are denoted as E. Along the graph there are also sub-graphs $C_1,\ C_2, \ldots,\ C_k$ which are connected, have less than n edges, and vertices with even degrees. This is true, because if we were to remove M, then we would remove an even number of edges, leaving an even number of edges on the remaining vertices in the cycle. Assume, also, that each cycle has an Eulerian cycle, called $E_1,\ E_2,\ E_k$. As the graph is connected, there is a vertex a_i on each sub-graph C_i which is also on M and E_i. Assume that the vertices $a_1,\ a_2, \ldots,\ a_k$ are encountered in that order.

Thus, when describing the Eulerian cycle in the graph, we begin on v and travel along M until reaching a_1, travel along E_1 and return to a_1 and continue until we reach a_2 and repeat the process. This goes on until we reach a_k, travel along E_k, return to a_k and finish M by ending at v, whereby we have an Eulerian cycle as each edge has been traversed either along M or along $C_1,\ C_2, \ldots,\ C_k$.

□

Theorem 2. *A connected graph is semi-Eulerian (i.e. has an Eulerian path) if and only if there are no more than two vertices of odd degree.*

Proof. Assume that we are given a path, K, from vertex v_1 to v_n. Adding an edge, e connecting v_1 and v_n would give us an Eulerian cycle, K', where, as stated in theorem 1, each vertex has an even degree. Thus, we can see that removing e would give v_1 and v_n odd degrees while the vertices $v_2 \ldots v_{n-1}$ have even degrees, thus giving us an Eulerian path. Thus, the path K is Eulerian if and only if there are a maximum of two vertices of an odd degree which must be the initial and final vertices.

Similarly, suppose that we have a path from v_1 to v_n. Except for the first time v_1 is encountered and the last time that v_n is encountered, each time a vertex is listed there are two edges neighbouring it which are accounted for. Thus the degree of each vertex apart from v_1 and v_n must be even and v_1 and v_n must be the start and end vertices. □

The Eulerian and Hamiltonian paths and cycles can be seen in graphs other than the simple graph, such as the weighted graph and the directed graph.

2.2 Weighted and Directed Graphs

When the edges on graphs are given a numerical value, the graphs are called *weighted graphs*. In practical application, this value can be used to denote many things, including distances and measurements of time.

The *directed graph* (or a digraph) is used to more clearly specify the relationship between two neighbouring vertices. It is a graph where each edge has a particular direction and is called an *arc*. Thus as the arc from a vertex x to a vertex y is not the same as the arc from y to x, an arc which is directed from x to y will be denoted in this paper as *(x,y)*. The set of all arcs will be denoted as $\mathcal{A}$.

In a directed graph, an *out-degree* of a vertex is defined as the number of arcs with directions pointing away from the vertex, and an *in-degree* of a vertex as the number of arcs pointing towards a vertex. A *source* is a vertex with an in-degree of zero and a *sink* is a vertex with an out-degree of zero. This definition is implemented in the flow network, which will be introduced in section 1.4.1.

Furthermore, in directed graphs there is an important distinction between paths and directed chains. A directed chain does not allow the possibility of travelling on an arc in a direction opposite from its orientation when travelling from x to y, while a path does. In Figure 2.2 there is a sequence x, A, B, y, which is a path as it contains both forward and reverse arcs, and a sequence x, C, y, which is a directed chain as it only contains forward arcs.

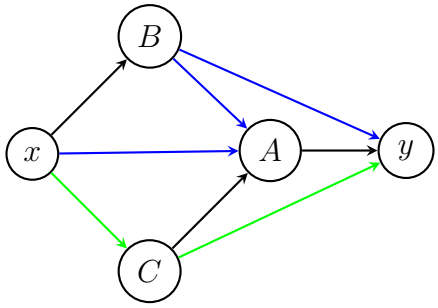


Figure 2.2: A directed graph showing a path in blue and a chain in green.

2.3 The Flow Network

A flow network is a type of directed graph where the Ford-Fulkerson algorithm is used to find the maximal flow through a system. The system can represent a pipeline, traffic flow, electrical transportation systems and more.

In a flow network there is at least one source, s, for flow entering the system, and at least one sink, t from where the flow leaves the system. The vertices (called *nodes* in flow networks) will be referred to as x or y as arcs between those two vertices are denoted as (x, y), and the set of all vertices will be referred to as $\mathcal{N}$.

Given that $x \in \mathcal{N}$, we can say that

$$A(x) = \{y \in \mathcal{N} | (x, y) \in \mathcal{A}\}, \tag{2.1}$$

where $A(x)$ represents all the vertices after the vertex x. Similarly, we can let $B(x)$ be stated as

$$B(x) = \{y \in \mathcal{N} | (y, x) \in \mathcal{A}\}, \tag{2.2}$$

where $B(x)$ represents all the vertices before x.

Thus, the different functions of the nodes in the flow network can be found using the

following formula where $f(x, y)$ and $f(y, x)$ denote the flow through the arc,

$$\sum_{y \in A(x)} f(x, y) - \sum_{y \in B(x)} f(y, x) = \begin{cases} v, & x = s \\ 0, & x \neq s, t \\ -v, & x = t \end{cases} . \tag{2.3}$$

In this formula it can be understood that if the sum of the flow over the arcs after x is greater than the sum of the flow over the arcs before it, and the net flow is v, the node x must be the source as there is more flow leaving the node than entering it. Similarly if the opposite is true, the node x must be the sink and the net flow must be $-v$ as all the flow that has entered the network must leave it. If, however, the net flow through the node is 0, it is clear that the node x is an intermediate node as all the flow entering it also leaves it. This is also true for flow networks with multiple sources and sinks, which will be discussed in section 2.1.3.

In flow networks, each arc has a certain weight associated with it called a *capacity*. This is the maximum flow through the arc (x, y). Thus, we can state the relationship between the flow through, and capacity of, an arc as follows

$$f(x, y) \leq c(x, y). \tag{2.4}$$

Within a flow network, there are certain subsets of arcs called *cuts* and denoted as $\mathcal{C}$ which separate s from t. For example in the set of arcs $(X, \overline{X})$, $s \in X$ and $t \in \overline{X}$. Here the set of X represents the set which contains at least s and any amount of intermediary vertices, while the set of $\overline{X}$ represents all the intermediary vertices not in X and t. The arcs connecting the nodes in X and $\overline{X}$ form the cut. Since the cuts separate the source from the sink it is clear that the flow through the network cannot exceed the capacity of any cut. Furthermore, as there can be any number of cuts separating s from t, the maximum flow through the network must be equal to the capacity of the minimum cut separating the two sets. The capacity of the cut will be denoted by $c(X, \overline{X})$.

2.4 The Task Graph

Graphs can be used to illustrate situations where tasks need to be carried out in a particular order, such as in manufacturing industries and project planning. However, to do this we need to define a new type of graph where the order of the vertices is taken into account. This type of graph is called a *task graph*. As this graph is used to represent situations different from those in the flow network, it may be inferred that the vertices and edges in this graph represent other things too. In a task graph, the vertices represent tasks and the weights on the edges represent a cost in some form, for example time or money. The weights are denoted as $w(x_i, x_j)$ where (x_i, x_j) represents an arc between the two vertices x_i and x_j. An example of a task graph with weights is seen in Figure 2.3.

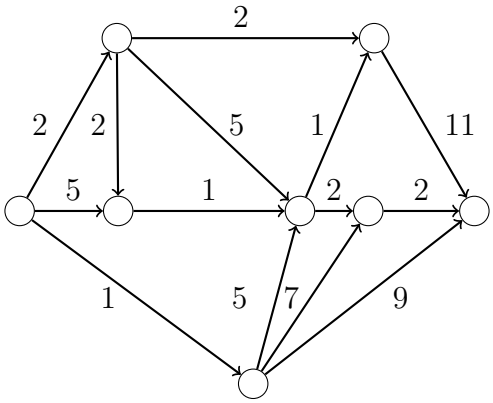


Figure 2.3: An example of a task graph.

3 Graph algorithms

In this section, the *Ford-Fulkerson algorithm*, *Dijkstra's algorithm* and the *critical path algorithm* will be explored. These algorithms can be used and modified to solve some of the most intrinsic graph theoretic problems.

3.1 Ford-Fulkerson Algorithm

The Ford-Fulkerson Algorithm was developed by L.R. Ford Jr. and D.R Fulkerson and published in 1969 [2]. The algorithm is used to calculate the maximal flow through a flow network and is applicable in, as mentioned earlier, several areas including pipelines, and transport networks.

3.1.1 Outline of the Ford-Fulkerson Algorithm

The *Ford-Fulkerson algorithm* aims to determine the maximum flow through a network by finding the minimal cut separating the source from the sink, as the flow through the network cannot exceed the capacity of minimal cut.

Theorem 3 (Max-flow min-cut theorem)**.** *In any network, the value of any maximal flow is equal to the minimal cut capacity of all cuts separating s and t.*

The algorithm starts by assuming that the flow throughout the network is 0. Then, it looks for a path from source to sink for which the residual capacity, c_f is strictly greater than 0. The residual capacity along the given path, p is determined by:

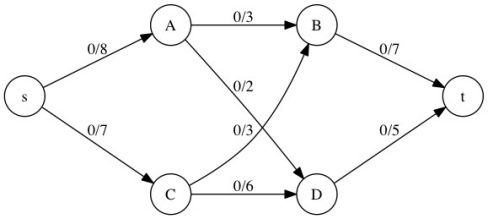
$$c_f = \min\{c(x,y) - f(x,y) \text{ if } (x,y) \in \ p, \text{ or, } f(y,x) \text{ if } (y,x) \ \in p\}, \tag{3.1}$$

where the minimum is taken over all arcs belonging to the given path. If c_f is positive we increase the flow along the forward arcs by c_f and decrease the flow along the reverse arcs by c_f, as shown in the following algorithm.

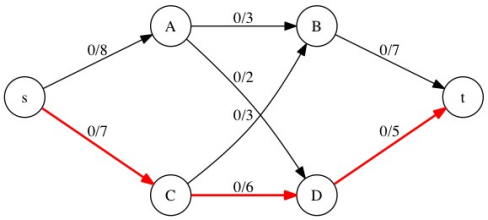
$$\begin{aligned} f(x,y) &\rightsquigarrow f(x,y) + c_f \\ f(y,x) &\rightsquigarrow f(y,x) - c_f \end{aligned}$$

The algorithm reaches completion when there is no path from source to sink where c_f is greater than 0. This algorithm will be illustrated in the following example.

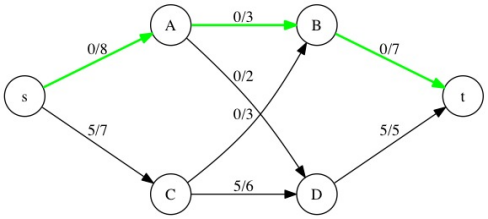
3.1.2 Example of Ford-Fulkerson Algorithm



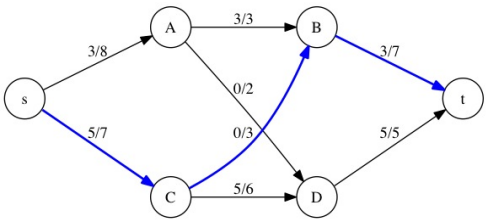
Here we can see a flow network with no flow. This means that flow can only be sent through forward arcs.



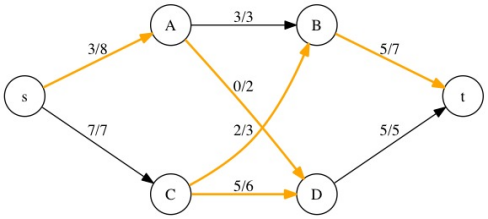
This graph shows one possible path, marked in red, from source to sink. The residual capacity of this path is given as $\min\{7-0, 6-0, 5-0\} = 5$.



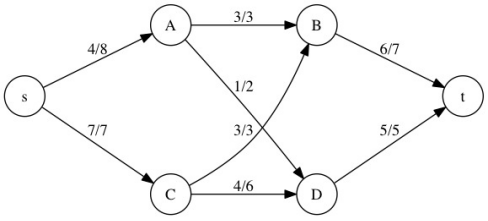
Along the path marked in red, the flow was increased by 5 units. On the green path the residual capacity is given as $\min\{8-0, 3-0, 7-0\} = 3$.



Along the path marked in green, the flow was increased by 3 units. On the blue path the residual capacity is given as $\min\{7-5, 3-0, 7-3\} = 2$.



Along the path marked in blue, the flow was increased by 2 units. On the yellow path the residual capacity is given as $\min\{8-3, 2-0, \mathbf{5}, 3-2, 7-5\} = 1$. As the arc (C, D) is traversed in the opposite direction, the maximum amount that can be transported through it is equal to the non-zero flow in the direction of the arc.



Along the path marked in yellow, the flow was increased by 1 unit on the forward arcs and decreased by one on the arc (C, D). As there are no more paths with a non-zero residual capacity, the algorithm terminates. We can determine the maximum flow by noting the flow entering the sink or leaving the source as both are equal. In this graph we have a maximum flow of 11 units.

After terminating the algorithm we can find a minimum cut by investigating the vertices,

beginning with the source, connected by arcs with non-zero residual capacity. These vertices are in the set of X while the other vertices are in the set of $\overline{X}$. The capacity of the arcs formed between X and $\overline{X}$ is the minimum cut and thus the maximum flow. This can be seen in Figure 3.1.

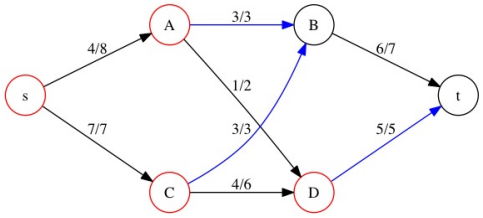


Figure 3.1: The arcs between $X = \{s, A, C, D\}$ and $\overline{X} = \{B, D, t\}$ are (A, B), (C, B) and (D, t)

Another method to find the maximum flow is to find the capacity of the minimal cut. To do this, we need to consider all the cuts in the graph. This method is shown in table 1.

Table 1: The capacities of the different cuts in the graph

X	$\overline{X}$	arcs between X and $\overline{X}$	value of cuts
s	A, B, C, D, t	$(s, A),\ (s, C)$	15
s, A	B, C, D, t	$(A, D),\ (s, C),\ (A, B)$	12
s, C	A, C, D, t	$(C, D),\ (s, C),\ (C, B)$	17
s, A, B	C, D, t	$(s, C),\ (A, D),\ (B, t)$	16
s, C, D	A, B, t	$(s, A),\ (C, B),\ (D, t)$	16
s, A, D	C, B, t	$(s, C),\ (A, B),\ (D, t)$	15
s, C, B	A, D, t	$(s, A),\ (C, D),\ (B, t)$	21
s, A, C	B, D, t	$(A, B),\ (A, D),\ (C, B)\ (C, D)$	14
s, A, B, C	D, t	$(C, D),\ (A, D),\ (B, t)$	15
s, A, C, D	B, t	$(C, B),\ (A, B),\ (D, t)$	**11**
s, A, C, D, B	t	$(B, t),\ (D, t)$	12

As we can see, this method is longer, especially as the number of nodes increases, thus the Ford-Fulkerson algorithm for finding maximum flow is preferable.

3.1.3 Applications of the Ford-Fulkerson Algorithm

Some common applications of the Ford-Fulkerson algorithm are within transportation problems, where flow networks could be expanded to have multiple sources and sinks, and weights associated with both arcs and nodes. These transportation problems can be illustrated by a company, C, which has three factories, S_1, S_2 and S_3 from where it wants to transport the maximum possible number of items to the warehouses T_1, and T_2. This is illustrated in Figure 3.2 To solve this we can create the super-source S', i.e. a source which has an infinite capacity and arcs that lead only to the sources, and super-sink T' which has an infinite capacity as well and only incoming arcs from the other sinks. Now, a normal maximum flow problem has been created which can be solved using the simple Ford-Fulkerson algorithm.

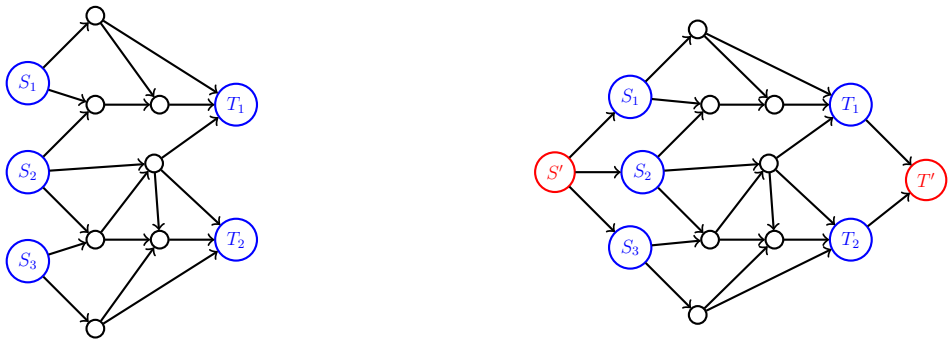


Figure 3.2: To the left we can see a graph with multiple sources and sinks and to the right we can see an extension of the same graph with a super-source and super-sink.

Assume that C has to send a truck from s to t and wants to find the minimum time taken for the trip. During the trip, the truck must pass some traffic lights, shown as any node x, and given an associated waiting time, w, shown as weights on the nodes. In this situation we can see x as a miniature flow network where w is the weight associated with an arc flowing from x', representing the start of the node, to x'', representing the end of the node. When this is done with all the weighted nodes we are given a normal flow network, and the problem is easily solvable. This is seen in Figure 3.3.

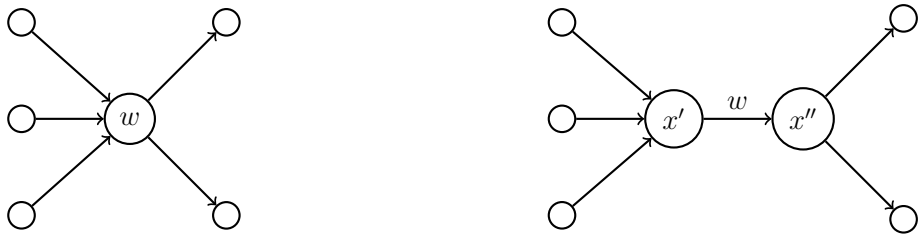


Figure 3.3: The weighted vertex gets split into two vertices with the weight on the edge between them.

3.2 Dijkstra's Algorithm

Dijkstra's algorithm can be used in a variety of practical and theoretical situations to find the shortest path. Thus, we can see that the algorithm only applies in the context of weighted graphs. We may denote the weight on edge (x, y) as $w(x, y)$. Similarly we may call the minimum weight of a path from x to y the distance $d(x, y)$. Dijkstra's algorithm finds not only the shortest distance from the initial vertex, x_0, to the final vertex, y_0, but also for all the vertices along the shortest path.

As this algorithm finds an optimal path for any given situation, we can see that it is particularly important for society today. Variations of Dijkstra's algorithm are used in many different areas. They are used by, e.g. transport systems, such as *Lufthansa*, to find the shortest travel times.

3.2.1 Outline of Dijkstra's Algorithm

Dijkstra's algorithm begins with the set $S = \{x_0\}$ and assumes that the distances $d(x_0, x_i) = d(x_0, y_0) = \infty$ given that $x_0 \neq x_n$. The next step begins by considering all the vertices connected to S and finds the distances $d(x_0, x_i)$ by running the procedure

$$\min\{d(x_0, x_i), d(x_0, x_0) + w(x_0, x_i)\} \rightsquigarrow d(x_0, x_i), \tag{3.2}$$

this means that the right hand side represents the new minimum distance between x_0 and x_i. It should be noted that in the first iteration the minimum distances are equal to the weights of the connecting edges. For the following iterations we choose the vertex

with the smallest minimum distance from x_0, and in case of a tie we may pick of those vertices.

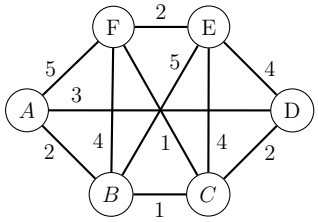
Without loss of generality, we may assume that the shortest distance is between x_0 and x_1 for the first iteration. In the next step we add to the set S the new vertex x_1 so that we may assume that $S = \{x_0, x_1\}$.

We proceed in the same manner to find the distance using a general formula for the $k+1$th iteration where $S = \{x_0, x_1, x_2, \ldots, x_k\}$ where we only consider the neighbours of x_k

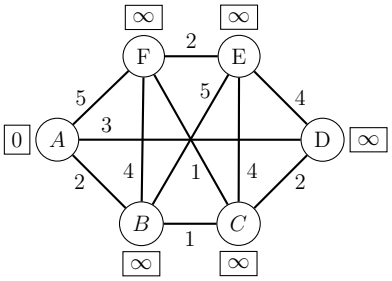
$$\min\{d(x_0, x_i), d(x_0, x_k) + w(x_k, x_i)\} \rightsquigarrow d(x_0, x_i). \tag{3.3}$$

This algorithm terminates when $S = V$. This will be illustrated in the following example.

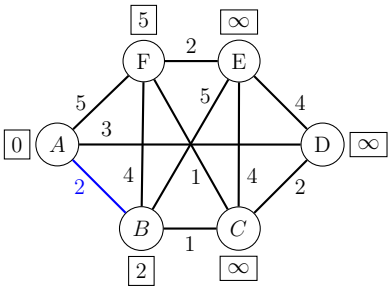
3.2.2 Example of Dijkstra's Algorithm for Simple Graphs



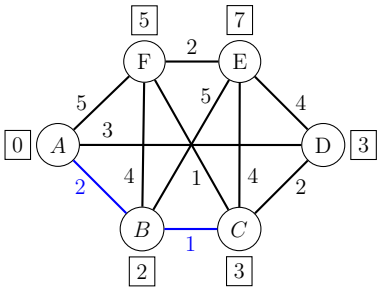
Example of a simple graph with given weights on the edges. For this graph we will calculate the shortest distance from A to E.



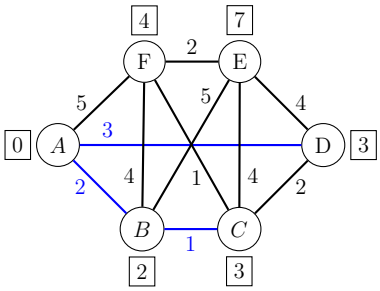
$d(A, A) = 0$ and $d(A, F) = d(A, B) = d(A, D) = d(A, C) = d(A, E) = \infty$.



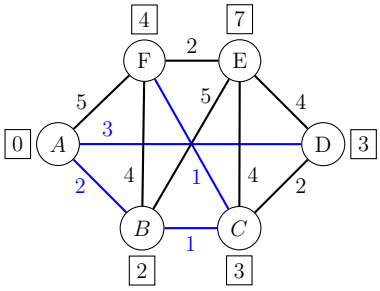
$S = \{A\}$
$d(A, B) = 2,\ d(A, D) = 3,$
$d(A, F) = 5$. As the shortest distance is 2, the vertex B will be added to the set S and the edge (A, B) is on the shortest path so far.



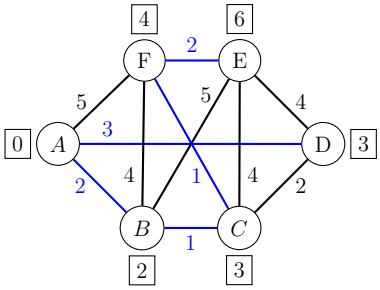
$S = \{A, B\}$
$d(A, F) = 6,\ d(A, C) = 3,$
$d(A, E) = 7$. As the shortest distance between A and any untravelled vertex is 3, the vertex C will be added to the set S and the edge (B, C) is on the shortest path so far. The distance at (A, F) remains unchanged as the path through B is greater than the prior path calculated. As both C and D have the same distance from A, it does not matter whether you choose one over the other.



$S = \{A, B, C\}$
$d(A, F) = 4,\ d(A, E) = 7,$
$d(A, D) = 5$. As the shortest distance between A and any untravelled vertex is 3, the vertex D will be added to the set S and the edge (A, D) is on the shortest path so far. The distance at (A, F) changes as a shorter path to it has been calculated.



$S = \{A, B, C, D\}$
$d(A, E) = 7,\ d(A, C) = 5$. As the shortest distance between A and any untravelled vertex is 4, the vertex F will be added to the set S and the edge (C, F) is on the shortest path so far.



$S = \{A, B, C, D, F\}$
$d(A, E) = 6$, . As $S = V$, the algorithm can terminate and we can see that the shortest path from A to D has 6 units.

Given a situation where we need to calculate the shortest path for an undirected graph with weighted vertices (introduced for a directed graph in Figure 3.3) we may either change the algorithm used to the one presented in Equation 3.4, where the time at the vertices is given as $t(x_k)$

$$\min\{d(x_0, x_i), d(x_0, x_k) + w(x_k, x_i) + t(x_k)\} \rightsquigarrow d(x_0, x_i), \tag{3.4}$$

or keep the initial formula and change the graph as shown in Figure 3.4.

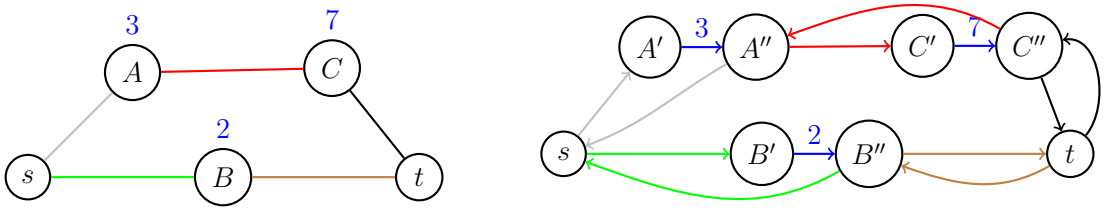
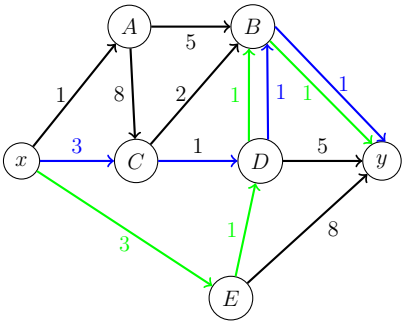


Figure 3.4: An undirected graph with weighted vertices to a directed graph with weighted arcs.

There may only be forward arcs from any x' to an x'' as x' represents the entrance to the vertex x where the associated weight must be traversed in order to reach x'', or the exit of the node. Thus, we can see that arcs leaving a node x'' may only reach a node y'' or y as if it reaches y' it would have to travel to y'' anyway.

3.2.3 Example of Dijkstra's Algorithm for Directed Graphs

This example is based on a question proposed by Roberts and Tesman in [3]. The problem is based on a traveller who wants to fly via aeroplane from a point x to a point y. However, there are no direct flights and the traveller must make at least one stop. The problem is to find the shortest route. This is illustrated in Figure 3.5. In order to apply Dijkstra's algorithm to a directed graph, such as in this example, we need only to modify it so that only the neighbours with arcs directed from x_0 to x_i can be considered and not arcs in the opposite direction.



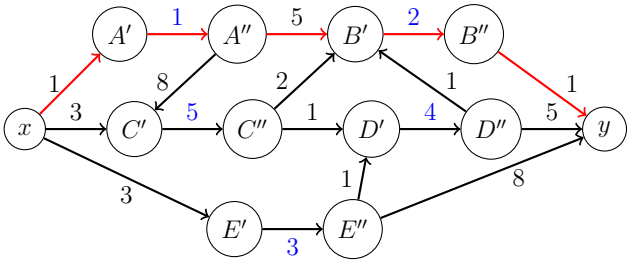
$d(x, A) = 1$
$d(x, B) = 5$
$d(x, C) = 3$
$d(x, D) = 4$
$d(x, E) = 3$
$d(x, y) = 6$

Figure 3.5: The aeroplane problem with the shortest paths marked in blue and green.

The shortest path from x to y is through x, C, D, B, y or x, E, D, B, y. This is seen in

Figure 3.5 and are marked respectively in blue and green.

Note that in the aeroplane problem discussed, there are no waiting times associated with the vertices. However, this problem can be expanded to include waiting times (weights), in which case, the shortest path changes. This is illustrated in Figure 3.6.



$d(x, A') = 1$
$d(x, B') = 7$
$d(x, C') = 3$
$d(x, D') = 7$
$d(x, E') = 3$
$d(x, y) = 10$

Figure 3.6: The aeroplane problem with weights at the vertices.

The shortest path when the aeroplane problem is reformulated to take into account weights at the vertices is x, A, B, y, which has a total distance of 10.

3.2.4 Applications of Dijkstra's Algorithm

Dijkstra's algorithm can also be used to solve common logical problems such as the Cabbage, Goat, Wolf, Ferryman (CGWF) problem. In this situation we have a ferryman who wants to ferry over all three things to the other side of the river but who can only carry one at a time in his boat. He cannot leave the goat with the cabbage or the wolf with the goat. We must find the optimal route for the ferryman.

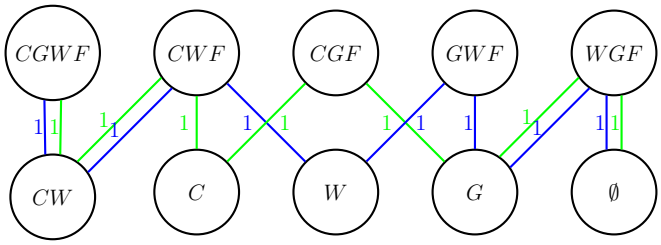


Figure 3.7: A graphical representation of the CGWF problem, with the participants on the initial side, I, of the river, so that $I = \{G, C, W, F\}$. The goal is to reach $I = \emptyset$.

In Figure 3.7, the solution was determined by finding the shortest path from $I = \{C, G, W, F\}$ to $I = \emptyset$. In this graph there were 2 shortest paths determined, both with

a weight of 7 units. The two different paths are shown respectively in green and blue.

This problem was also presented graphically in an introductory book about graph theory by Gondran and Minoux [4]. However their graph had an error as it included an edge between GWF and W which is impossible.

3.3 Critical Path Method

The critical path method is commonly used within project management in many fields such as construction, product development and maintenance. This method finds the path that takes the longest time to complete as all the tasks on those vertices have the same earliest and latest start time. This means optimizing the process would require allocating more resources to the vertices on the critical path or running more activities in parallel. Using the critical path method to study and optimize a project is such a popular choice that there are dozens of websites dedicated to providing this service.

3.3.1 Outline of the Critical Path Method

When considering the critical path problem with a given task graph, G, we begin solving the problem by first topologically ordering the vertices. This is because some activities simply cannot be done without others preceding it.

The *topological order* is found by first finding a vertex, x_0, with no edges preceding it (a source). This vertex is given the label 0 and represents the starting point of the set of activities. Once we have found and labelled x_0, we delete it and all of its outgoing arcs from G, and are thus left with one or more vertices with no incoming arcs. We choose one of the new sources and label it as 1. We continue to delete the vertices that are labelled until we reach a final vertex, x_n with no outgoing edges which will have the final label. This can be seen in Figure 3.8.

Now, we may find the so-called *earliest times* for the activities to start. The earliest time for each vertex represents the moment that the dependant activities have finished and thus, the earliest time the activity at that vertex may begin. Calculating this is done

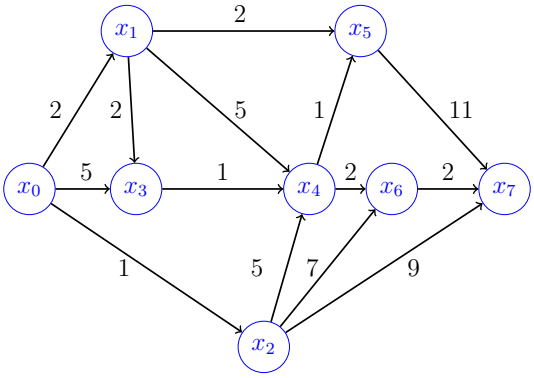


Figure 3.8: A task graph with ordered vertices.

by the following formula given that $t_0 = 0$, then for $j > 0$

$$t_j = \max\{t_i + w(x_i, x_j)\}, \text{ where } x_i \text{ precedes } x_j. \tag{3.5}$$

The final vertex x_n (the sink) will have the greatest earliest time, which also indicates the minimal time required to complete the task.

For the graph presented in Figure 3.8, the calculations for the earliest times are shown below.

t_0	$=$	0	0
t_1	$=$	$\max\{t_0 + w(x_0, x_1)\} = \max\{2\}$	2
t_2	$=$	$\max\{t_0 + w(x_0, x_2)\} = \max\{2\}$	2
t_3	$=$	$\max\{t_0 + w(x_0, x_3), t_1 + w(x_1, x_3)\} = \max\{5, 4\}$	5
t_4	$=$	$\max\{t_2 + w(x_2, x_4), t_3 + w(x_3, x_4), t_1 + w(x_1, x_4)\} = \max\{7, 6, 7\}$	7
t_5	$=$	$\max\{t_1 + w(x_1, x_5), t_4 + w(x_4, x_5)\} = \max\{4, 8\}$	8
t_6	$=$	$\max\{t_4 + w(x_4, x_6), t_2 + w(x_2, x_6)\} = \max\{9, 9\}$	9
t_7	$=$	$\max\{t_6 + w(x_6, x_7), t_2 + w(x_2, x_7), t_5 + w(x_5, x_7)\} = \max\{11, 11, 19\}$	19

Given this information, we may calculate the *latest times*. The latest time represents the last time at which the activity at the given vertex may commence without slowing the process down. For example, if there is a activity, A_1, that takes 2 hours and a parallel activity A_2 that takes 4, both of which need to be completed before the final activity A_3

can commence, the latest time that A_1 can commence is when A_2 has been carried out for 2 hours. If it starts any later, then the entire project would be delayed as A_3 will start later.

To calculate the latest time we assume that $T_n = t_n$ so that the latest time of the final vertex is equal to the earliest time. Then for $j < n$

$$T_i = \min\{T_j - w(x_i, x_j)\}, \text{ where } x_j \text{ follows } x_i. \tag{3.6}$$

T_7	$=$	t_7	19
T_6	$=$	$\min\{T_7 - w(x_6, x_7)\} = \min\{17\}$	17
T_5	$=$	$\min\{T_7 - w(x_5, x_7)\} = \min\{8\}$	8
T_4	$=$	$\min\{T_5 - w(x_4, x_5), T_6 - w(x_4, x_6)\} = \min\{7, 15\}$	7
T_3	$=$	$\min\{T_4 - w(x_3, x_4)\} = \min\{6\}$	6
T_2	$=$	$\min\{T_4 - w(x_2, x_4), T_7 - w(x_2, x_7), T_6 - w(x_2, x_6)\} = \min\{2, 10, 10\}$	2
T_1	$=$	$\min\{T_3 - w(x_1, x_3), T_4 - w(x_1, x_4), T_5 - w(x_1, x_5)\} = \min\{4, 2, 6\}$	2
T_0	$=$	$\min\{T_1 - w(x_0, x_1), T_2 - w(x_0, x_2), T_3 - w(x_0, x_3)\} = \min\{0, 0, 1\}$	0

We may also calculate the *slack*, which is defined as the difference between the latest time and the earliest time for any vertex x_i. The slack is denoted as $m_i = T_i - t_i$. It may be inferred that any vertices with $m_i = 0$ must be on the *critical path*.

Table 2: The slacks of the vertices in Figure 3.8. We can see that all vertices apart from m_3 and m_6 are on the critical path.

m_0	$=$	$T_0 - t_0 = 0 - 0$	0
m_1	$=$	$T_1 - t_1 = 2 - 2$	0
m_2	$=$	$T_2 - t_2 = 2 - 2$	0
m_3	$=$	$T_3 - t_3 = 6 - 5$	1
m_4	$=$	$T_4 - t_4 = 7 - 7$	0
m_5	$=$	$T_5 - t_5 = 8 - 8$	0
m_6	$=$	$T_6 - t_6 = 17 - 9$	8
m_7	$=$	$T_7 - t_7 = 19 - 19$	0

Thus, it can be seen that the critical paths for Figure 2.3 are $x_0,\ x_1,\ x_4,\ x_5,\ x_7$ and $x_0,\ x_2,\ x_7$.

3.3.2 Example and Application of the Critical Path Method

In this example, a critical path for the recruitment process of IBM India[1] will be identified. A graph of the process can be seen in Figure 3.9. In this graph, the process is taken to end at step 16, which if we refer to Appendix B, is not the end of the process. However, as the process contains no parallel activities from step 16, it is clear that the critical path must include all vertices from step 16, until step 20 when the process concludes.

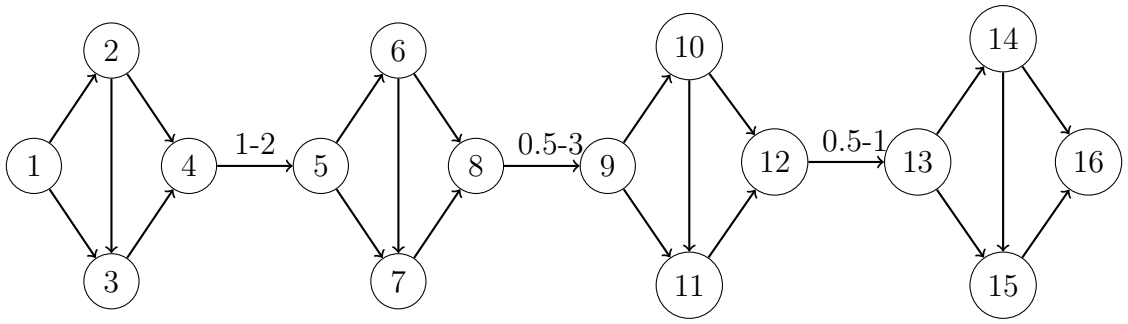


Figure 3.9: a graphical illustration of IBM India's recruitment process.

In this graph there are 3 bottleneck arcs, $(4, 5)$, $(8, 9)$ and $(12, 13)$. This means that if we cut the bottleneck arcs, we will have 4 different sub-graphs. For visibility, only the range of weights associated with the bottleneck arcs are shown in Figure 3.9.

As detailed information about the tasks carried out at the vertices is not directly necessary to calculate the critical path, except in the case of the activities at the vertices 4, 9, and 13, the data is available for the interested reader in Appendix B.

In this graph, a range of time for the completion of each step can be seen. This range represents the difference between a best case run of the graph, and a worst case run and suggests that there could be slack at each vertex.

It can be seen that the critical path in Figure 3.10 consists of the arcs (1,2), (2,3) and (3,4). This may also be checked by calculating the latest start times and slack, which is

[1]The data used was received from Sanchita Dutta, an ex-employee at IBM India.

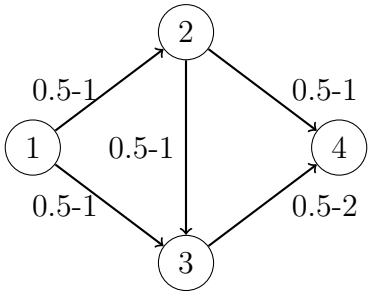


Figure 3.10: The first part of the recruitment process, called *sourcing*.

Table 3: The minimum ranges for the earliest start times for Figure 3.10.

(1)	=	0
(2)	=	$\max\{0 + 0.5\} = 0.5$
(3)	=	$\max\{0 + 0.5, (2) + 0.5\} = 1$
(4)	=	$\max\{(2) + 0.5, (3) + 0.5\} = 1.5$

left to the reader.

When comparing Tables 3 and 4, we can see that there is a great difference between the ranges, and can infer that reducing time on all the vertices would have a larger impact than reducing time at any particular vertex.

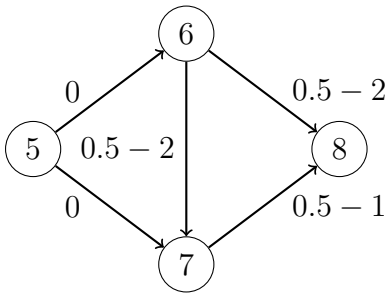


Figure 3.11: The second part of the recruitment process, called *tech select*.

The arcs (5,6) and (5,7) have no associated weight as the activity at vertex 5 represents *CV select* which is not a process but an outcome of Figure 3.10. As it is an outcome there is no time associated as it is completed instantaneously once the activity at vertex 4 is completed. This is also true for the vertices 9 and 13.

It can be seen, from the Tables 5 and 6, that the critical path in the Figure 3.11 consists of the arcs (5, 6), (6, 7) and (7, 8).

It can be seen from the Tables 7 and 8 that the critical path in Figure 3.12 consists

Table 4: The maximum ranges for the earliest start times for Figure 3.10.

(1)	=	0
(2)	=	$\max\{0+1\} = 1$
(3)	=	$\max\{0+1, (2)+1\} = 2$
(4)	=	$\max\{(2)+1, (3)+2\} = 4$

Table 5: The minimum ranges for the earliest start times for Figure 3.11.

(5)	=	$\max\{(4)+1\} = 2.5$
(6)	=	$\max\{(5)+0\} = 2.5$
(7)	=	$\max\{(5)+0, (6)+0.5\} = 3$
(8)	=	$\max\{(6)+0.5, (7)+0.5\} = 3.5$

Table 6: The maximum ranges for the earliest start times for Figure 3.11.

(5)	=	$\max\{(4)+2\} = 3.5$
(6)	=	$\max\{(5)+0\} = 3.5$
(7)	=	$\max\{(5)+0, (6)+2\} = 5.5$
(8)	=	$\max\{(6)+2, (7)+1\} = 6.5$

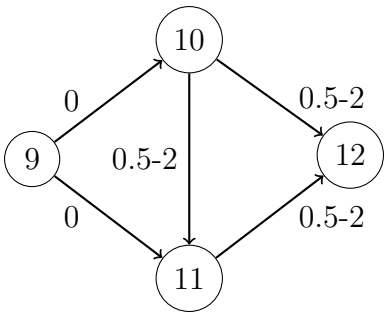


Figure 3.12: The third part of the recruitment process, called *PDM select*.

Table 7: The minimum ranges for the earliest start times for Figure 3.12.

(9)	=	$\max\{(8)+0.5\} = 4$
(10)	=	$\max\{(9)+0\} = 4$
(11)	=	$\max\{(9)+0, (10)+0.5\} = 4.5$
(12)	=	$\max\{(10)+0.5, (11)+0.5\} = 5$

of the arcs (9, 10), (10, 11) and (11,12).

Table 8: The maximum ranges for the earliest start times for Figure 3.12.

(9)	=	$\max\{(8)+3\} = 9.5$
(10)	=	$\max\{(9)+0\} = 9.5$
(11)	=	$\max\{(9)+0, (10)+2\} = 11.5$
(12)	=	$\max\{(10)+2, (11)+2\} = 13.5$

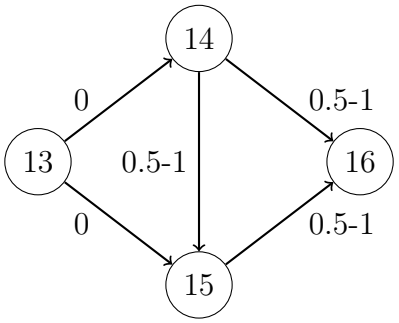


Figure 3.13: The fourth, and last, part of the recruitment process, called *offer roll-out*.

Table 9: The minimum ranges for the earliest start times for Figure 3.13.

(13)	=	$\max\{(12)+0.5\} = 5.5$
(14)	=	$\max\{(12)+0\} = 5.5$
(15)	=	$\max\{(13)+0, (14)+0.5\} = 6$
(16)	=	$\max\{(14)+0.5, (15)+0.5\} = 6.5$

Table 10: The maximum ranges for the earliest start times for Figure 3.13.

(13)	=	$\max\{(12)+1\} = 14.5$
(14)	=	$\max\{(13)+0\} = 14.5$
(15)	=	$\max\{(13)+0, (14)+1\} = 15.5$
(16)	=	$\max\{(14)+1, (15)+1\} = 16.5$

It can be seen that the critical path for Figure 3.13 consists of the arcs (13, 14), (14, 15) and (15, 16). As, the critical paths for all the sub-graphs have been calculated, the critical path of the task graph Figure 3.9 has also been calculated.

As all the vertices are on the critical path, streamlining the process would require reducing the time taken at all vertices. We can see that this would have a large impact on the process as the range at each vertex is relatively small but the difference between

the best and worst run is much greater.

4 Using the Algorithms in Mesh Networks

A *mesh network* is a type of digital network topology, where each node represents a device which can send and receive certain amounts of data to other nodes via the arcs connecting them. Mesh networks include both grounded networks with a defined infrastructure and the increasingly more common *wireless mesh network* which is comprised of mobile devices. These networks can either be centralised or decentralised, where all nodes have an equal importance. The decentralised wireless mesh network, more commonly known as the *wireless ad-hoc network* (WANET) will the main focus of this section.

Mesh networks can be represented by many different types of graphs, and the most common being the complete graph where each node is connected to each other, *star graph* where each node is connected to a central node called a router, and the *mesh graph* where the nodes are interconnected randomly.

As the reliability of a network is becoming an increasingly important question, there are three key aspects of a network, namely its connectivity, the shortest path through it, and the maximum flow through it, which will be discussed in this section.

In the complete graph seen in Figure 2.1 there is a high level of connectivity, since if one of the arcs or nodes were to die, the rest of the graph would remain unaffected. We may also deduce that the paths from v_1 to v_n will have a length of 1. However, because of the network's connectivity, the cost is also higher as each node receives information about all other nodes.

Looking at the star graph in Figure 4.1 we can see that the cost for this graph is low as well as the paths from v_1 to v_n will have a maximum length of 2. However, there is a low level of connectivity as if the router, or central node A, were to die, all of the other nodes would receive no information.

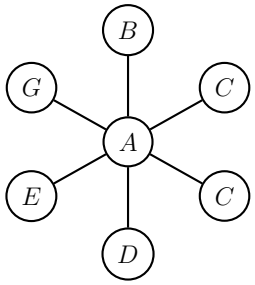


Figure 4.1: A star graph.

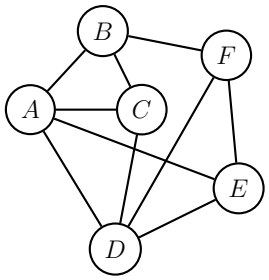


Figure 4.2: A mesh graph.

As a mesh graph can represent a graph with any level of connectivity between that of a complete graph and star graph, there are no conclusive statements which can be made about its connectivity or its paths. An example of a mesh graph can be seen in 4.2.

4.1 Challenges Presented in the WANET

A large scale WANET commonly consists of several smaller *clusters*, or groups of nodes where each node can either be a simple node, a gateway node from where data can enter or exit the cluster, or a cluster head which coordinates the operations in the cluster [5]. As WANET comprises of devices connected wirelessly, the number of nodes in the graph is not static as devices may both join and leave the network at any time.

This dynamic attribute of a WANET makes it impossible to use the traditional Ford-Fulkerson and Dijkstra's algorithms in order to calculate the maximum flow and the shortest path as both algorithms rely on a constant number of nodes. The algorithms can only be applicable for a system at a particular moment in time. Thus, the challenge presented in this type of graph is to find a dynamic set of algorithms which are generally applicable for a WANET system.

5 Future Work

As presented in the introduction, graph theory is applicable to many different areas. In section 3, we discussed one particular application of it within computer science. Expanding upon this area, and finding a dynamic algorithm, is crucial to the development of

faster and more reliable networks which is becoming increasingly important in today's society.

Furthermore, if the parameters of what arcs and vertices can be used to represent are redefined, graphs can be used to illustrate more diverse situations and recognise different kinds of patterns which can be used to further our understanding of other areas; such as the spread of disease, which can be illustrated using a social network.

As graph theory is a young area within mathematics, its applications, both practical and theoretical are increasing by leaps and bounds. This paper simply aims to provide the layman with an introduction from which to further develop their understanding and use of this area.

6 Acknowledgements

I would like to begin by thanking my mentor, Marlena Nowczyk, for her patience and for being one of the best teachers I could ask for. I would also like to thank Rays, The Royal Patriotic Society, The Jacob Wallenberg Foundation and Stockholms Matematik Centrum for the opportunity to research in an absolutely fascinating area of study.

I would like to thank my fellow students for the journey which is certainly more memorable than the final destination, the laughs and the late nights. I would like to extend a special thanks to Mariam Andersson, Agnes Nordqvist, Dennis Alp and Philip Frick for making my summer not only challenging but also an absolutely marvellous experience. I would like to extend special acknowledgements to Dennis Alp for generally being a mountain (one that is not overrated) to lean on through the vicissitudes of Rays, and to his alter-ego BordDennis, or the Compiling King, for the tech support.

Last but not least I would like to thank my mother, Sanchita Dutta; not just for her parental support, but for the information she provided from IBM India which helped further my research greatly.

References

[1] Euler, L., *Solutio Problematis ad Geometriam Situs Pertinentis*, Dartmouth [Internet], 1736, [cited 2014, July 9], P. 128. Available from `http://www.math.dartmouth.edu/~euler/docs/originals/E053.pdf`

[2] Ford, L.R. Jr., Fulkerson, D.R., *Flows in Networks,* Princeton University Press, USA, 1969, P. 9–23.

[3] Roberts, F.S., Tesman, B., *Applied Combinatorics,* 2nd ed., Chapman & Hall, USA, 2005, P. 737–772.

[4] Gondran, M., Minoux, M., Vajda, S. [translator],*Graphs and Algorithms,* John Wiley & Sons, GB, 1984, P. 81.

[5] Gábos, D., Varga, M., *Cluster Formation in Wireless Mesh Networks,* Acta Universitatis Sapientiae: Electrical and Mechanical Engineering, **4**, 2011, P. 5–32.

[6] McKay, B. D., Robinson, R. W., *Asymptotic enumeration of Eulerian circuits in the complete graph*, Combinatorics, Probability and Computing, **7**, 1998, P. 448.

[7] Bondy, J.A., Murty, U.S.R., *Graph Theory with Applications,* Elsever Science Publishing Co. Inc., USA, 1976, P. 15, 62–65.

[8] Wilson, R.J., *Introduction to Graph Theory,* 2nd ed., Longman House, UK, 1979, P. 8–38.

A The number of Eulerian paths in K_5 (or Figure 2.1)

The complete graph, K_5 seen in figure 2.1 can be broken into three unique subgraphs seen in Figure A.1.

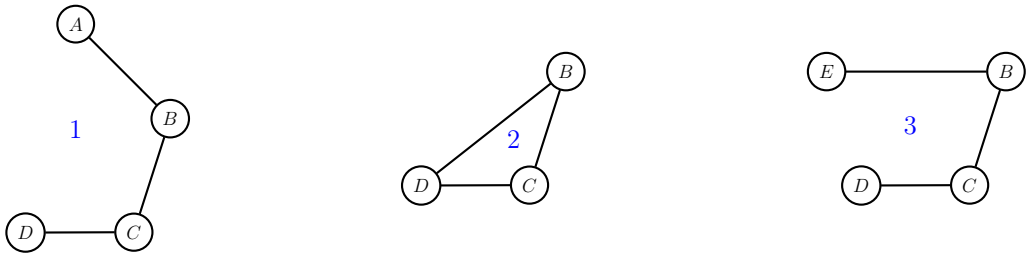


Figure A.1: Three unique subgraphs of K_5 from which all the Eulerian cycles in K_5 may be calculated.

These sub-graphs can be used to construct all the Eulerian cycles for the complete graph K_5. Through observation we can see that the sub-graphs 1 and 3 are similar types of sub-graphs as they have the same number of edges which have the same type of relationship with the other vertices and edges. Thus, as the number of possible edge-connections of 1 and 3 must be the same because the two sub-graphs are essentially the same, the number of Eulerian cycles must also be the same.

For the first sub-graph, 1, calculating the number of Eulerian cycles is done by first adding an edge from D to all possible vertices so that the following sub-graphs are obtained.

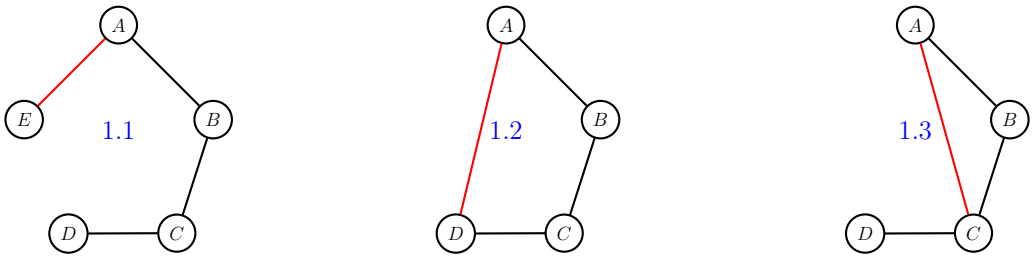


Figure A.2: The three possible continuations of the first sub-graph.

The subgraph 1.1 can be split into three further unique sub-graphs from which Eulerian cycles may be calculated.

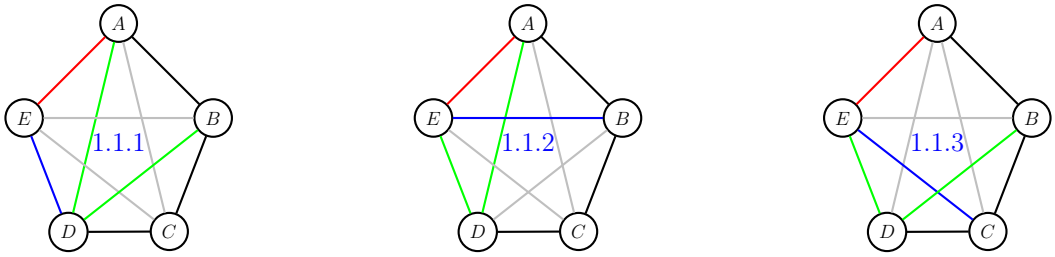


Figure A.3: Three sub-graphs of 1.1 with 2 unique paths each.

The edges in yellow represent the possible divergences for the cycle. This means that there are not just 3 sub-graphs of 1.1 but 6.

Figure A.4: The subgraphs of 1.2, with two possible Eulerian cycles each.

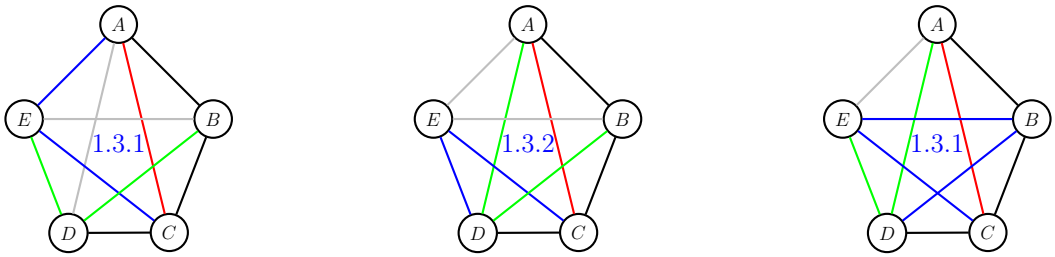


Figure A.5: The sub-graphs of 1.3, showing 2 possible Eulerian cycles each.

Thus, the total number of Eulerian cycles that can be formed by the sub-graph 1 are $5 \cdot 4 \cdot 3 \cdot (6 + 4 + 6) = 960$ cycles. This is because there are 5 choices for the first vertex, 4 for the second, and 3 for the third before the sub-graphs start splitting into unique Eulerian cycles.

Like the sub-graphs 1 and 3, the sub-graphs 2.1 and 2.2 are also similar and must have the same number of Eulerian cycles.

We can see that there are 6 possible sub-graphs of 2.1. Thus, the total number of Eulerian cycles when starting with sub-graph 2 are $5 \cdot 4 \cdot 3 \cdot 2 \cdot 6 = 720$.

Figure A.6: The unique sub-graphs of 2.

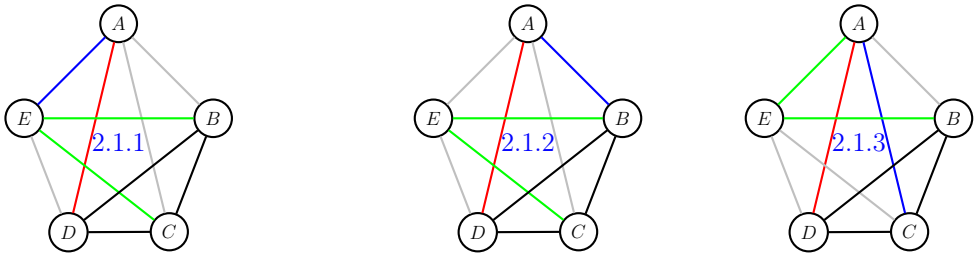


Figure A.7: The subgraphs of 2.1, with 2 paths each.

To find the total number of Eulerian cycles in the complete graph K_5, we need only to add the number of Eulerian cycles in the sub-graphs 1, 2 and 3. This gives us $(2 \cdot 960) + 720 = 2640$ Eulerian cycles.

The calculations for the number of Eulerian cycles in K_5 have also been done by Professor McKay and Dr. Robinson in [6]. However the number of Eulerian cycles calculated by them was only 264. This is because the method used in this paper takes into account the starting points and possible directions for the cycle (the edges shown in green). If we were to disregard this, than we would need to divide our result by $(5 \cdot 2)$. We divide by 5 as there are 5 different starting points for the cycle and by 2 as for all of the different cycles, there are two different choices at the last step. When this is done, the results from the method used in this paper and the McKay-Robinson method are the same.

As the method in this paper relies on breaking down the graph into smaller pieces and manually counting the possible Eulerian cycles, it is not recommended for graphs larger than K_5 as the number of Eulerian cycles for these increases very fast.

B Data from the IBM India Recruitment Process

Step no.	Title	Time Frame (in days)
Step 1	Approved Hiring request	0.5-1
Step 2	Create Hiring Tracker	0.5-1
Step 3	Opening to Sourcing Partners	0.5-2
Step 4	Screening	1-2
Step 5	CV Selected	0
Step 6	Schedule Tech Interview	0.5-2
Step 7	Tech Interview Arrangement	0.5-1
Step 8	Tech Round	0.5-3
Step 9	Tech Selected	0
Step 10	Schedule PDM Interview	0.5-2
Step 11	PDM Interview Arrangement	0.5-2
Step 12	PDM Round	0.5-1
Step 13	PDM Selected	0
Step 14	Offer Documentation	0.5-1.5
Step 15	Offer Validation	0.5-1.5
Step 16	Get Approvals	0.25-3
Step 17	Print Offer	0.25-0.5
Step 18	Quality Check	0.25-2
Step 19	Offer Dispatch	0.25-0.5
Step 20	Offer Follow Up	0.5-1

In the table above, the acronym PDM, refers to *Program Deployment Manager*. The steps that are in bold represent those which can be conducted parallelly and those in italics represent outcomes of steps, not processes, and thus have no associated cost in time as they occur instantaneously.

Using the calculations from section 2.3.1, we can do some simple addition to determine that the range for the total time taken in the IBM India recruitment process lies between 8 and 23.5 days.