Development of a UAV placement algorithm to establish redundant communication between nodes

I Venter

21744467

Dissertation submitted in fulfilment of the requirements for the degree *Master of Engineering* in Computer and Electronic Engineering at the Potchefstroom Campus of the North-West University

Supervisor: Prof ASJ Helberg

November 2016

Declaration

I, Ivan Venter hereby declare that the dissertation entitled "Development of a UAV placement algorithm to establish redundant communication between nodes" is my own original work and has not already been submitted to any other university or institution for examination.

Acknowledgements

I want to thank my Lord and Heavenly Father for the opportunity and strength to complete this dissertation.

Prof. Albert Helberg my supervisor. I would like to thank you for the time and energy you invested in me to make this dissertation possible. Thanks for the guidance and feedback throughout the years.

I want to thank the Technology and Human Resources for Industry Programme (THRIP) of the National Research Foundation (NRF) at the Potchefstroom Campus of North-West University for the financial support.

I would like to thank my family, Johannes Venter, Annelise Breedt and Jurie Venter for all the prayers and support; I love you very much. Special thank you to my father Johannes for his efficient motivation and interest.

Lastly, and definitely not least, Jani Van Der Westhuizen. Thank you for all your love, support and patience throughout this dissertation. Thank you for being there through the good times and all the tough times of this dissertation. I love you.

Abstract

The poaching of rhinoceroses in South Africa has increased to such an extent that it may become extinct in the wild. The horn of the rhinoceros is used in traditional medicine to cure various ailments. The price for rhinoceros horns has increased and, as a result, poachers use more advanced technology to obtain the horns. Consequently, the conservation of the rhinoceros becomes more challenging and expensive.

In attempts to reduce rhinoceros poaching, conservationists implemented various tracking systems. The problem with some of these systems is that the communication range is limited and consequently, once out of range, the rhinoceros behaviour is no longer monitored. In which case, a live alarm cannot be sent in the case of an emergency.

The current rhinoceros tracker comprises a collar around the foot of the animal. Signal repeaters are mounted to poles to extend the communication range. These repeaters disrupt the natural environment and are expensive. The proposed solution is to replace the signal repeaters on poles with solar charged Unmanned Aerial Vehicles (UAVs). Dedicated landing positions are predefined, and the UAV can move between these positions while tracking a pair of rhinoceroses. For landing sites outside the communication range of the base station, UAV repeaters are placed at landing sites to propagate the signal to the base station. The developed algorithm has two objectives: Firstly, it should ensure a redundant path between the rhinoceros and the base station exists. This will ensure reliable communication of the status of the rhinoceroses, even if a node in a path should fail. Secondly, the number of UAVs needed to create a path between a rhinoceros and the base station should be minimised.

Two shortest path algorithms were identified, implemented and the results compared to determine which landing sites to use between two communication nodes. The Dijkstra and Bellman-Ford algorithms were modified to use less landing sites when multiple source nodes are used. This is achieved when the source nodes share paths to the base station. The modified algorithms were simulated to compare them in terms of the Individual Path Node Count (IPNC), the Cumulative Paths Node Count (CPNC),

spread dependency and the running time. Twenty different scenarios were simulated for each number of source nodes (2-10).

The IPNC for a single scenario was determined by simulating the shortest path from each individual source node to the base station and then adding up the number of landing sites used with each individual simulation. The IPNC value for a network of a given size, is calculated using the average IPNC value of twenty different scenarios. From the results, was found that the modified Bellman-Ford's algorithm uses fewer landing sites for individual source nodes than the modified Dijkstra's algorithm.

The CPNC was determined by simulating all the source nodes simultaneously for each of the scenarios. The CPNC value for each number of source nodes is the average value of the different scenarios. The average number of landing sites used for each number of source nodes was compared, and the results indicated that the CPNC was less for the modified Bellman-Ford algorithm.

The spread dependency of the algorithms was determined by evenly spreading six source nodes confined to different sector sizes. The IPNC values were kept constant to remove the effects that distance has on the number of landing sites used. The number of landing sites used for each sector size was compared, and the results indicated that both algorithms are dependent on the spread of the source nodes.

The run time for each algorithm was compared for each number of source nodes. The modified Bellman-Ford algorithm took significantly longer to complete than the modified Dijkstra's algorithm. The run time increased as the number of source nodes increased.

In conclusion, two shortest path algorithms were selected and modified to create a redundant communication path between the UAV following a rhinoceros and the base station. Fewer landing sites were used with the modified algorithms to connect all the source nodes to the base station.

Keywords: *Algorithm, Rhinoceros tracking, UAV*

Opsomming

Renosterstropery in Suid Afrika het tot so 'n mate toegeneem, dat hul die gevaar in die gesig staar om uitgewis te word in die wild. Die horing van die renoster word in tradisionele medisyne gebruik vir die behandeling van verskeie siektes. Die prys van renoster horings het toegeneem en gevolglik gebruik die stropers meer gevorderde tegnologie om die horings in die hande te kry. Daarom het die bewaring van renosters moeiliker en duurder geword.

Verskillende opspoorstelsels is al geïmplimenteer in 'n poging om renosterstroping te verminder. 'n Probleem met van hierdie opspoorstelsels is dat dit 'n beperke kommunikasie reikwydte het en daarom word die gedrag van die renoster slegs gemonitor terwyl dit binne die reikwydte is. Dit verhoed lewendige alarm deteksie in die geval van nood.

Die huidige manier om renosters te volg is 'n band om die voet van die renoster. Pale met sein herhalers word geplant om die kommunikasie reikwydte te vergroot. Die pale steur die natuurlike omgewing en is boonop duur. Die voorgestelde oplossing is om die sein herhalers op pale te vervang met songelaaide Onbemande Lugtuie (OLT). Spesifieke landingsposisies word vooraf geïdentifiseer en die OLT'e kan tussen die posisies beweeg terwyl dit die renoster volg. Vir landingsposisies buite die kommunikasie reikwydte na die basisstasie word OLT herhalers geplaas op beskikbare landingsposisies om die sein oor te dra na die basisstasie. Om volgehoue kommunikasie te verseker tydens die faling van 'n node, word oortollige paaie geskep. Die plasing van die OLT'e moet bepaal word ten einde oortollige paaie te verseker, vanaf elke OLT wat 'n renoster volg tot by die basisstasie, deur die minste hoeveelheid landingsposisies te gebruik.

Twee algoritmes vir kortste paaie is geïdentifiseer om die landinsposisies tussen twee punte te bepaal. Die gewigte van Dijkstra en Bellman-Ford se algoritmes is aangepas om minder landingsposisies te gebruik wanneer veelvoudige oorsprong nodes gebruik word. Hierdie doel word bereik wanneer oorsprong nodes paaie na die basisstasie

deel. Die aangepaste algoritmes is gesimuleer om dit te vergelyk in terme van die Individuele Pad Node Telling (IPNT), die Kummulatiewe Pad Node Telling (KPNT), verspreiding afhanklikheid en die simulasietyd. Twintig verskillende scenarios is gesimuleer vir elke hoeveelheid oorsprong nodes (2-10).

Die IPNT vir 'n enkele scenario is bepaal deur die kortste pad vanaf elke oorsprong node na die basisstasie te bepaal en die hoeveelheid landingsposisies wat gebruik is met elke individuele simulasie bymekaar te tel. Die IPNT waarde vir elke hoeveelheid oorsprong nodes is die gemiddeld van die IPNT waarde van die twintig verskillende scenarios. Uit die resultate kan gesien word dat die aangepaste Bellman-Ford algoritme minder landingsposisies gebruik vir individuele oorsprong nodes in vergelyking met die aangepaste Dijkstra algoritme.

Die KPNT is bepaal deur al die oorsprong nodes gelyktydig te simuleer vir elke scenario. Die KPNT waarde vir elke hoeveelheid oorsprong nodes is die gemiddeld van die verskillende scenarios. Die gemiddelde hoeveelheid landingsposisies gebruik vir elke hoeveelheid oorsprong nodes is vergelyk en die resultate het aangedui dat die KPNT minder was vir Bellman-Ford se algoritme.

Die verspreiding afhanklikheid van die algoritmes is bepaal deur ses oorsprong nodes eweredig te versprei in verskillende sektor groottes. Die IPNT waardes is konstant gehou om die invloed van afstand op die hoeveelheid landingsposisies gebruik te elimineer. Die hoeveelheid landingsposisies wat vir elke sektor gebruik is, is vergelyk en die resultate toon dat beide algoritmes afhanklik is van die verspreiding van die oorsprong nodes.

Die simulasietyd van elke algoritme is vergelyk vir elke hoeveelheid van oorsprong nodes. Die aangepaste Bellman-Ford algoritme se simulasietyd was drasties langer as die van die aangepaste Dijkstra algoritme. Die simulasietyd het toegeneem soos wat die hoeveelheid oorsprong nodes verhoog het.

Ten slotte, twee kortste pad algoritmes is gekies en aangepas om 'n oortollige kommunikasie pad tussen die OLT wat 'n renoster volg en die basisstasie te skep. Minder

landingsposisies is gebruik met die aangepaste algoritmes om al die oorsprong nodes met die basisstasie te konnekteer.

Sleutelwoorde: *Algoritme, OLT, Renostersporing*

Contents

List of Figures

List of Tables

List of Algorithms

List of Acronyms

CITES Convention on the Trade of Endangered Species

CPNC Cumulative Paths Node Count

GPS Global Positioning Service

GUI Graphical User Interface

IPNC Individual Path Node Count

IUCN International Union for Conservation of Nature

L and L Lift and Land

UAV Unmanned Aerial Vehicle

List of Symbols

List of Symbols

D	Distance between two antennas
h	Height of antenna above ground
V	Number of vertices
E	Number of edges

Chapter 1

Introduction

This chapter serves as an introduction to the research. It starts with some background to why this problem exists, which is then followed by the problem that will be addressed in this dissertation. The goals of the research are then described followed by a brief overview of the dissertation.

1.1 Background

The International Union for Conservation of Nature (IUCN) declared the western black rhinoceros (Diceros bicornis longipes) as extinct in 2011. The IUCN stated that the other populations of rhinoceros (often abbreviated as rhino) in Africa are also in danger. More than 90% of the total white rhino population is found in South Africa as well as 40% of the remaining black rhino population. Over the past five years, poaching has more than doubled in South Africa per year, which may cause the remaining rhino population to become extinct in the wild [1]. Since there is a ban on the trading of rhino horns, as established by the Convention on the Trade of Endangered Species (CITES), the demand for rhino horns is met by illegal markets that rely on poachers for supplies [2]. The rhino horn ingredient in traditional medicine is assumed to cure a wide

spectrum of ailments. Rhino horns were first used for traditional medicine in China, before spreading to Japan, Korea and Vietnam [3]. The increase in the rhino horn retail prices results in escalating poaching levels. The financial rewards for rhino horns have led poachers to use advanced technology, increasing the difficulty and expenses to protect the rhinos [1]. According to the authors in [4] legalising the trade of rhino horns will benefit the owners and the rhino population. The illegal trade can be replaced with legal horns and the income can be used to protect the rhino [1].

Animal tracking using Global Positioning Service (GPS) has been implemented on multiple occasions for various reasons, for instance, migration pattern observation by ecologists and natural resource managers [5]. Tracking rhino behaviour and gathering additional data is one of many ways to reduce rhino poaching. Disturbances in the behaviour can be monitored and alarms can be incorporated to inform the owner of unusual activity. By collecting the tracking and sensor data, rhino behaviour can be studied and analysed. The analysis can be used to identify the reaction of rhinos toward people, loud noise, cars and a gun shot.

1.2 Problem statement

The problem with existing tracking systems for rhinos is the limited communication range. Rhino tracking collars can't be placed around the neck, because its neck is just as thick as its head. The collar is placed around the rhino's foot resulting in communication range limitations. The behaviour data can only be monitored while the rhino is within the wireless communication range. When a rhino moves outside the communication range, the communication link is lost, the data will be buffered and can only be sent once a new connection is established. Should an alarm be triggered in this buffering period, the alert will not be sent immediately. To extend the communication range of the network, signal repeaters mounted on poles are used. More signal repeaters on poles are planted to improve the communication coverage. This, however, is an expensive exercise and furthermore disrupts the natural environment.

A proposed solution to eliminate the need for increasing numbers of signal repeaters on poles, while maintaining communication with the rhinos, is to replace the signal repeaters on poles with solar charged Unmanned Aerial Vehicles (UAVs). The UAVs will have specific landing positions and will be able to move between the various landing positions to keep track of the rhinos. For the purposes of this dissertation, a UAV monitoring a rhino will be referred to as the source UAV/node. The communication range between the UAV and the base station is exceeded for landing sites that are located further than a specified distance from the base station. When the UAV follows a rhino to these positions, repeater UAVs are used as signal repeaters to maintain communication. UAV repeaters are placed on available landing sites to establish communication between UAVs monitoring the rhinos and the base station. A redundant path is created to ensure that communication is not lost if one node should fail. This redundant path is a second alternative path between the same source node and base station.

The challenge lies in determining on which landing sites UAV repeaters should be placed to create two communication paths between each UAV monitoring a rhino and the base station using the least number of nodes. This algorithm will execute daily, rearranging the placement of the UAVs to adapt the network topology, based on the location of the rhinos.

1.3 UAV's

The UAVs mentioned in the proposed solution will be used to replace the repeaters. They will not constantly hover above the rhino but make use of a Lift and Land (L and L) functionality. When the UAV makes use of its L and L function to transmit a message, it will lift high enough to extend its communication range but low enough to conserve energy. The UAV will not follow the rhino as it moves out of communication range with the UAV. The UAVs will be rearranged daily to form the required communication paths. Even though the term UAV is used throughout this dissertation, it is aimed at a quadcopter application. According to the authors of [6], approval from the

director of Civil Aviation is required for a UAV to fly higher than 150 feet (45.7 meters).

Recent regulations on UAVs have imposed severe restrictions on their use. These regulations are under review. This study assumes that the final results of this review will permit reasonable use.

1.4 Research goal

To arrange UAVs in such a way that redundant paths can coexist, shortest path algorithms will be used. The research goal is to select, modify and test appropriate shortest path algorithms to satisfy the network planning and design constraints of the proposed solution. The algorithms are modified to adhere to the predefined requirements and used to determine the placement of UAV repeaters to connect source nodes with the base station. The source nodes have to be connected to the base station via two paths, to increase the survivability of the network. The modified shortest path algorithms are compared in terms of the number of UAVs used, simulation time and its ability to save landing sites with different distributions of the UAVs monitoring the rhinos.

1.5 Dissertation overview

The remainder of this dissertation is structured as follows: In Chapter 2, a literature review is presented, covering the candidate shortest path algorithms considered in this study. In Chapter 3, the experimental setup and assumptions are described. Chapter 4 is a description of how the implemented algorithms were modified and verified to stay within the requirements described in Chapter 3.

The results of the simulations are presented in Chapter 5, including the validation of the research using a Monte Carlo analysis based on random sampling. Finally, Chapter 6 contains the conclusion and recommendations for future work.

Chapter 2

Literature study

This chapter provides the background information relevant to the research. The technical definitions relevant to this study are provided to avoid any confusion. A number of shortest path algorithms, which can be used to determine the placement of the UAVs are investigate. Since redundant paths are to be formed, we investigated algorithms designed to find more than one shortest path. The k-path shortest path algorithms also make use of single shortest path algorithms. Single shortest path algorithms are also investigated to investigate the option of manipulating these algorithms for this specific application.

2.1 Technical definitions

Full-duplex communication

When two devices have the ability to send and receive data to and from each other simultaneously, it is referred to as full-duplex communication. This can be achieved by using two separate transmission mediums, one to send and another to receive the data. Full-duplex communication is also possible in a single medium, where the capacity of the single channel is divided between the signals flowing in both directions [7] [8].

Repeater

Signals travelling long distances experience attenuation that endangers the integrity of the data. An electronic circuit that receives a partially degraded signal regenerates it and sends it on, is known as a repeater. To transmit data over long distances, several repeaters may be needed to counter the effects of attenuation [7] [9].

Redundant path

According to the authors in [10], redundancy involves the use of duplicate devices to perform in case of an error or malfunction on the primary device. For the same reason, redundant paths are used to maintain communication with the base station in the event of a node failure on the primary path [7].

Node

In computer networks, each computer in the network is sometimes referred to as a node. Any peripheral device connected to a network can also be called a node [9]. Since the source is the origin where something comes from, the source node is the peripheral device where the communication is initiated [11].

Network

A combination of devices connected to each other and capable of communicating is called a network. These devices can be connected via air or cable as the transmission medium, using wireless radio frequencies or propagating a signal over a wire [7] [12].

Wireless communication

Wireless communication does not make use of physical conductors. Instead, it broadcasts electromagnetic waves to communicate. Anyone with a suitable device can receive the signals [7]. Wireless technology usually has limitations in terms of bandwidth, power supply and variation in availability [13]. Radio waves can propagate through space with three different paths, using ground waves, sky waves and space waves.

Ground waves stay close to the earth and follow the curvature of the earth. Ground waves are only possible with vertical polarization when propagated through an antenna and falls within the 30 kHz to 3 MHz frequencies.

Sky waves propagate signals up into the atmosphere and makes use of the ionosphere that refracts the signal back to earth. The travelling distance of this signal depends on the angle they enter the ionosphere and is only possible for signals with frequency between 3 MHz and 30 MHz.

When the transmitting antenna transmits the signal in a straight line to the receiving antenna, it is known as space waves. The space wave is also referred to as line of sight communication because the wave travels in a straight line the antennas have to be in line of sight. Line of sight communication is possible for frequencies above 30 MHz, which is where most open band communication frequencies are. Figure 2.1 indicates that space waves are extremely dependent on the height of the transmitter and the receiver. To calculate the operating distance around the curvature of the earth equation 2.1 can be used [9]. This emphasizes how the communication distance decreases when the collar is placed around the foot of the rhino.

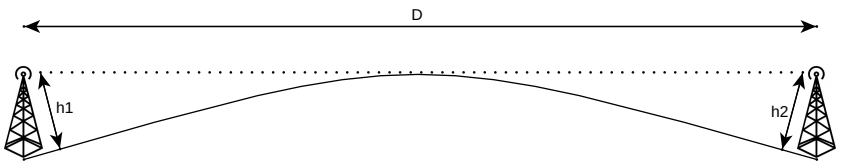


Figure 2.1: Space waves: Line of sight

$$D = 4.12 \times \sqrt{h1} + 4.12 \times \sqrt{h2} \tag{2.1}$$

where D is the direct distance between the two antennas in kilometres. The height of the antennas above ground is given by h1 and h2 in meters.

Attenuation or degradation of signals is inevitable for all mediums of transport. Signals will attenuate proportionally to the square of the traveling distance. While the original signal travels at the speed of light, all mediums will slow down the signal. Noise will also have an impact on the quality of the signal. Every external signal broadcast will be picked up by the receiver and have an impact on the integrity of the signal [9].

2.2 K-path shortest path algorithms

In this section algorithms that can calculate more than one path between the source and sink node are described. In Graph Theory, a node is referred to as a vertex, and the links connecting the nodes are referred to as edges. When calculating paths, k-shortest path algorithms will define two paths as ”different” when the two paths differ even as little as a single edge. When two paths from the source node to the sink node is formed without sharing an edge, the paths are referred to as edge-disjoint paths. Paths formed from source to sink node that has no intermediate vertices in common, is referred to as vertex-disjoint paths [14]. Edge- and vertex-disjoint paths ensures robustness in the network in the case of edge or node failure, since an alternative functioning path remains available [15] [16].

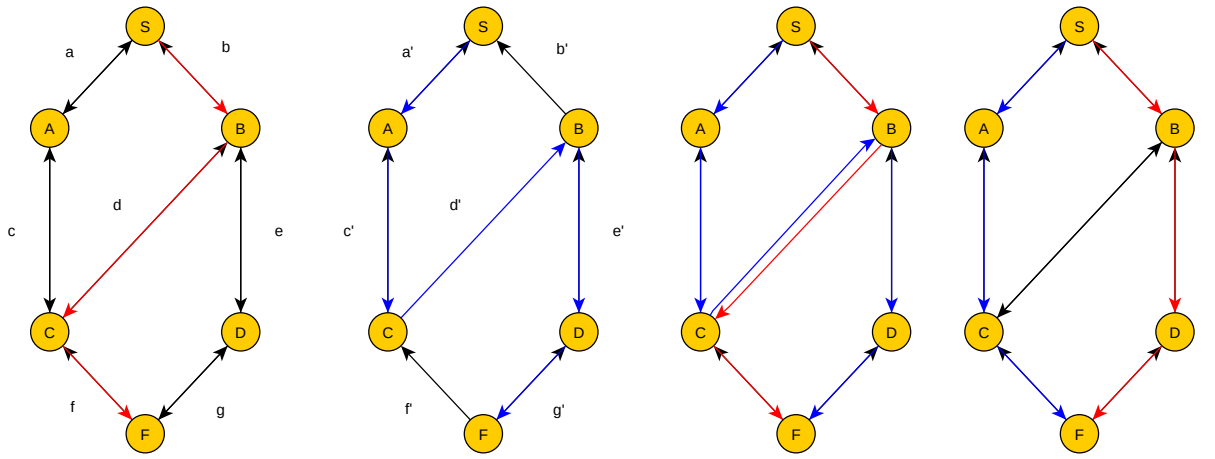
2.2.1 Suurballe and Tarjan’s algorithm

The Suurballe and Tarjan’s algorithm is an algorithm used to find a pair of edge-disjoint paths of minimal total cost [17]. One of the advantages of Suurballe and Tarjan’s algorithm is that it evades the ”trap topology” problem. The problem occurs when a method fails to find two edge-disjoint paths, while two edge-disjoint paths exist [14], [18]. To describe the mechanics of Suurballe and Tarjan’s algorithm, figure 2.2 will be used. Figure 2.2(a) introduces the graph G. The red lines in this figure indicates the shortest path, as calculated with Dijkstra’s shortest path algorithm. More infor-

mation on Dijkstra's algorithm will be provided later in this chapter. All the edges in graph G is then transformed to graph G' as shown in figure 2.2(b), using equation 2.2.

$$c'(v,w) = c(v,w) - d(s,w) + d(s,v) \tag{2.2}$$

All the edges that form part of the shortest path are removed from graph G', to ensure the two paths will be edge-disjoint. Dijkstra's algorithm is implemented again on the transformed graph, to find the shortest path to the sink node with the transformed edge-weights as indicated in figure 2.2(b). The edges crossed in both directions after the second iteration of Dijkstra's algorithm (figure 2.2(c)), are removed from the graph, and the remaining edges are then used to find the shortest pair of edge-disjoint paths between two vertices (figure 2.2(d)) [18]. Graph G can also be solved when negative edge weights exist. In the case of negative edge weights with no negative cycles, the single shortest paths can be determined with Bellman Ford's algorithm (explained later in this chapter) [17].



(a) Graph G (b) Transformed graph G' (c) Two paths (d) Two shortest paths

Figure 2.2: Images to explain the steps used for two edge-disjoint shortest paths determined with Suurballe and Tarjan's algorithm.

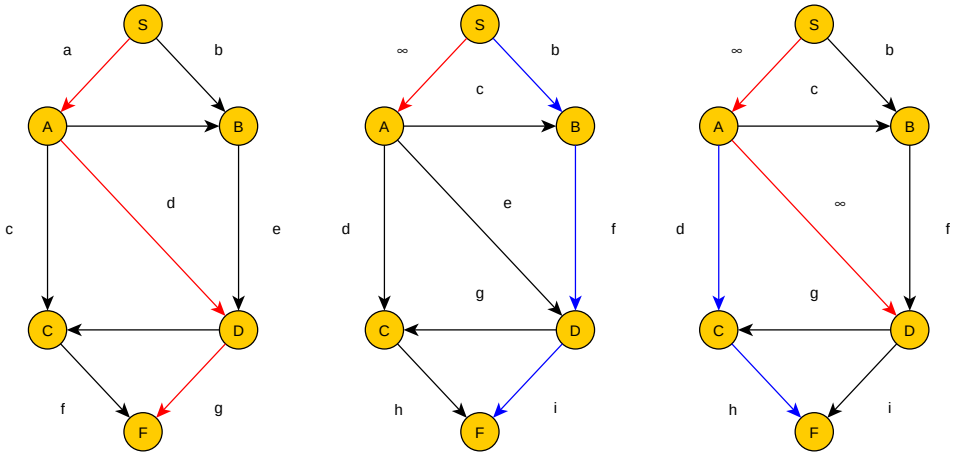
2.2.2 Yen's algorithm

Yen's algorithm is used to find K paths with a minimum length/cost between two nodes, in which no loops are allowed. One of the advantages of Yen's algorithm is that it's computation time increases linearly with the number of shortest paths K [19]. To explain the mechanics of this algorithm, figure 2.3 is used. Let $A^1, A^2, ..., A^K$ represent the first, second and Kth shortest paths respectively. Any efficient single shortest path algorithm (such as Dijkstra's algorithm) can be used to determine A^1, which is shown in figure 2.3(a). If two paths with an equal cost are found, any one of these paths can be selected as A^1. The path A^1 is then added to List A, which contains paths $A^1, A^2, ..., A^K$ [20].

The next step involves a couple of iterations using the following definitions. Path A^K is a deviation of path A^{K-1} at (i), where (i) is some node on the path from the source node (S) to the sink node (F). The paths of A^K and A^{K-1} coincide from node (S) to node (i), which is also the root (R_i^K) of the path A_i^K. The spur node S_i^K is the last part of path A_i^K, which has one node coinciding with A^{K-1}.

Node (S) becomes the spur node that has a root path to itself. The first check is for edges on the root path that coincide with the paths in List A. The costs of these edges are changed to ∞ for the new path (A^k) to deviate from the previous A^{k-1}. An efficient single shortest path algorithm is used again to determine the shortest path from the spur node (S) to the sink node (F), which is also the spur path $S_1^2 = (S) \rightarrow (B) \rightarrow (D) \rightarrow (F)$. The first potential k-shortest path is derived by joining the root path with the spur path, which is $A_1^2 = R_1^2 + S_1^2 = (S) \rightarrow (B) \rightarrow (D) \rightarrow (F)$ with a cost of $b + f + i$ (as can be seen in figure 2.3(b)). The potential k-shortest path S_1^2 is then inserted into List B, which contains the potential k-shortest paths. The spur node is shifted to (A) and the above process is repeated to arrive at the second potential k-shortest path A_2^2 shown in figure 2.3(c). A third and final iteration is required to arrive at A_3^2 visible in figure 2.3(d), which is the third potential k-shortest path inserted into List B. No more than $K - k + 1$ potential k-shortest paths needs to be stored in to List B, where K is the number of paths and k the current iteration.

The path in List B with minimum cost is moved to List A as the second shortest path (A^2). If two paths have the same cost, any one of the two can be selected as A^2. The algorithm requires another iteration of the process described in the previous paragraph to add potential k-shortest paths to the List B, after which the path with minimum cost is selected as A^3 and inserted into List A. Finally figure 2.3(e) shows three loopless shortest paths found using the K-shortest path algorithm of Yen [20] [21].



(a) Graph example (b) First path for List B (c) Second path for List B

(d) Third path for List B (e) Three shortest paths

Figure 2.3: Steps to explain the method of Yen's algorithm to determine three loopless shortest paths.

2.2.3 Critical evaluation

In Chapter 1, it was suggested that redundant paths are created from source node to sink node to ensure a communication path exists to the base station in the case of a node failure. To maintain a communication path from source to sink node in the case of a node failure the paths need to be vertex-disjoint. The above literature showed how Suurballe and Tarjan's algorithm can be used to determine a pair of edge-disjoint paths in a graph. The authors of [17] state that Suurballe and Tarjan's algorithm can be adapted to compute vertex-disjoint paths. In Chapter 3 certain restrictions are placed on the paths for this application. To apply the restrictions, it is necessary to use an algorithm which is simple to modify in order to remain functioning, even with the restrictions. Suurballe and Tarjan transform the graph in order to determine the second shortest path. To apply modifications to a graph that is transformed will not be simple to implement. Yen's algorithm is also not an option for this application, since the two paths for this application should be vertex-disjoint and the k path coincides with the $k-1$ path from the source node to node i (R_i^K).

Both algorithms start by using a single shortest path algorithm to determine the first path. The difference between the algorithms is their approach to determine the second path. One could argue that the approaches taken are for the algorithm to comply with a certain predefined application. It was decided that the same approach will be followed in this research: using a single shortest path algorithm to determine the first path. The second path will then be determined by manipulation of a single shortest path algorithm to satisfy the restrictions in Chapter 3.

2.3 Single shortest path algorithms

This section covers different shortest path algorithms, which is one of the fundamental problems in graph theory. The shortest path problem involves finding the path between two nodes in a graph with weighted edges, where the weights may repre-

sent some cost associated with using that edge that result in the lowest possible cost. Minimizing the total cost leads to the shortest path [22]. Shortest path algorithms can be used to determine the shortest route between two locations on a map. A critical path can also be determined in a PERT chart, where the jobs that need to be performed are the edges and the time for each job is the weights [23]. With many available applications for the shortest path problem, four different algorithms are discussed as candidate algorithms for implementation in this study.

2.3.1 Dijkstra's algorithm

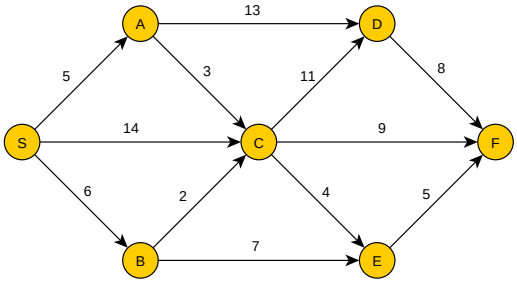
Dijkstra's algorithm is probably the most popular shortest path algorithm. All the nodes are labelled as either tentative or permanent, which helps to identify the next node to work from. Except for the source node that has a permanent value of zero; all the nodes are labelled tentative at the start of the algorithm. The status of a node changes to permanent when the shortest possible path from source node to that node is found [24]. When a node is labelled as permanent, it is removed from the set Q shown in Algorithm 1. This algorithm can only use positive distances (edge weights) between nodes. Weaker communication between nodes results in higher allocated edge weights. The initial path cost to each node is infinite and changes as shorter paths are found. When the path with lowest cost from the source to a node is found this cost is then assigned to the node and the node's status is changed to permanent.

Algorithm 1 Dijkstra

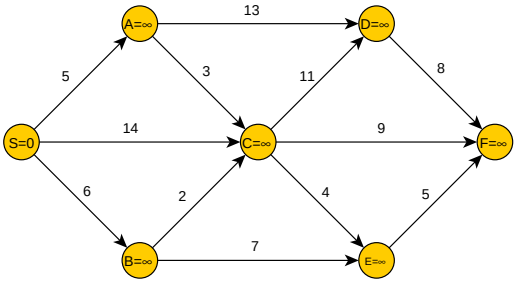
Given a network N, with a set of vertices V and a source vertex S. The distance/cost between vertices is given by D in the form D(u,v), which is the distance from vertex u to v.

1. **For** each vεV **set** Distance(v) = ∞ **and** Previous(v) = β
2. **Set** Distance(S) = 0
3. **Let** Q = The set of all nodes in N
4. **While** Q not empty
   Let u = node in Q with smallest Distance[u]
   Remove u from Q
   For each neighbour v of u
   Temp = Distance[u] + D(u,v)
   If Temp $<$ Distance(v)
   Then set Distance(v) = Temp **and** Previous(v) = u

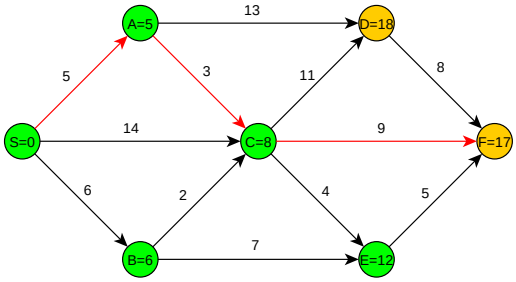
To further demonstrate Dijkstra's algorithm, the shortest path will be determined in a worked example using the network in figure 2.4(a). According to the first three steps in algorithm 1, the shortest paths to each of the nodes is temporarily infinity long except for the source node that is zero, and all the tentative nodes are loaded into the set Q. These first steps are the initial steps of the Dijkstra algorithm that is calculated only once when starting with the computations and is demonstrated in figure 2.4(b). The next steps will execute in a while loop and repeated until the shortest path is found; this is where the node F is the next tentative node with the lowest value. If it is desired to obtain the shortest path from a node to all other nodes, the iterative step will execute N-1 times within a network with N nodes [25] [26]. The steps to determine the shortest path using Dijkstra for this example is as follows.



(a) Dijkstra: Example



(b) Dijkstra: Initialization



(c) Dijkstra: Shortest path found

Figure 2.4: The network, initialization and final shortest path for the Dijkstra worked example.

1. The source node has the smallest distance: u = S. S is removed from the set Q, and the values for the neighbour nodes are A = 5, B = 6 and C = 14. All three these values are smaller than infinity and are assigned as the new distance to each of the nodes with prior node S.

2. In this step node A has the smallest tentative distance: u = A. A is removed from the set Q and thus the value of A is permanent. The distance to the neighbour nodes still in the set Q is C = 8 and D = 18. Since $8 < 14$ and $18 < \infty$ both nodes are assigned the new distance values with their prior node set as A.

3. The node with the smallest tentative distance is node B: u = B. Node B is then removed from the set Q, and the distance to B is permanent. Distances to the neighbour nodes in the set Q are C = 8 and E = 13. The distance to node C is already 8 which is not smaller and thus not changed, although the distance to

node E is changed since $13 < \infty$. The prior node for C is still node A, and the prior node for E is set to node B.

4. The next node with the smallest tentative distance is node C: u = C. This means that C is removed from the set Q, and the distance to node C is permanent. The distances to the neighbour nodes of C in set Q are D = 19, F = 17 and E = 12. Given that $19 > 18$, $12 < 13$ and $17 < \infty$, node D stays the same while new distances are assigned to nodes E and F. The prior nodes for E and F are changed to C while the prior node for D stays node A.

5. The node with the smallest tentative distance is node E: u = E. Node E is removed from the set Q, and the distance to node E is now permanent. Node E has one neighbour node in the set Q, and the distance to this node is F = 17. Since the distance to node F is already 17, it is not changed, and the prior node of node F is still node C.

6. The final step is illustrated in figure 2.4(c). Two tentative nodes are remaining in set Q, and our final node (node F) has the smallest distance. This means that the distance to node F will be permanent, and the shortest path has been found. The shortest path for this network determined using Dijkstra's algorithm is shown in figure 2.4(c) with red arrows.

2.3.2 Bellman-Ford algorithm

In cases where negative edge weights are assigned, Dijkstra's algorithm can't be used, and the Bellman-Ford algorithm should be considered. The Bellman-Ford algorithm can handle networks with some negative edge weights, but should not have a negative length cycle [27]. Dynamic programming is the basis of this method, which calculates the shortest path in n-1 iterations with n being the number of nodes [28]. The Bellman-Ford algorithm starts at the node closest to the source and continues through its neighbour nodes to the destination in the same order each iteration. The cost to each node, except the source that is zero, starts with infinity and is changed when a

shorter path to that node is found. The costs from the source to each node are added up, and if a shorter path is found, the new value is allocated to that node. These costs and the node's predecessor are stored and used for the next iteration. These values are updated for each of the n-1 iterations if a shorter path is found. After the iterations, the total cost of the destination node will be the shortest path [29] [30].

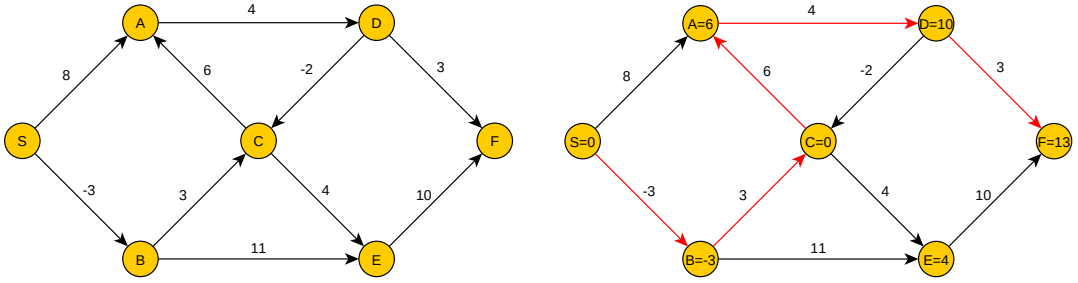
Algorithm 2 Bellman-Ford

Given a set of vertices V with source vertex S. The distance/cost between vertices is given by D and the number of vertices is given by n.

1. **For** each vεV **set** State(V) = Temp, Distance(v) = ∞ **and** Previous(v) = ϕ
2. **Set** State(S) = Perm **and** Distance(S) = 0
3. **Set** Condition = True **and** Counter = 0
4. **While** Condition = True
   Counter = Counter + 1
   If Counter = n + 1
   Then Output "Negative circuit exists" **and** Stop
   Condition = False
   For each uεV
   For each vεV with (u $\neq$ v)
   if Distance(v) $>$ Distance(u) + D(u,v)
   Then Set Distance(v) = Distance(u) + D(u,v) **and** Previous(v) = u
   Set Condition = True

The first three steps of Algorithm 2 are the initializing steps. Within the initializing steps, it is important to note that the cost to all nodes is ∞ except for the source node that is zero. The value of the source node is also permanent where the ∞ values for other nodes are temporary. Within the while loop, the path to each node is found and assigned to the nodes. This is repeated until the shortest path (lowest cost) from the source node to all other nodes has been found. The while loop is terminated in one of two ways: the first is where a negative cycle is found and the second is when no

shorter path can be found to any of the nodes. With a negative cycle, no shortest path can be found since with each repetition a shorter path exists, resulting in the shortest path of cost -∞. If all the cost values for the nodes stay the same, the shortest path is determined, and the algorithm exits the while loop. A worked example will be used to demonstrate the Bellman-Ford algorithm using the network in figure 2.5(a). The following is the steps followed to find the shortest path from the source to each of the nodes using the Bellman-Ford algorithm.



(a) Bellman-Ford: Example (b) Bellman-Ford: Final shortest path

Figure 2.5: The network and final shortest path for the Bellman-Ford work example.

1. After initialization, the algorithm starts from the source node S. Starting from S there is a path to node A of cost 8 and a path to node B with cost -3. From node A there is one path to node D with cost 4, leaving node D with the cost of 12 with prior node A. Node B has two paths, one to node C with cost 3 giving node C a cost of 0 and one to node E with cost 11 giving node E the cost of 8. Node C has a path to node A of cost 6, changing the cost to node A from 8 to 6 with prior node C and a path to node E of cost 4 changing the cost to node E to 4 with prior node C. Node D has one path to node F with cost 3, giving node F the cost of 15. Lastly node E has a path to node F of cost 10, giving node F the cost of 14 with prior node E. That concludes the first iteration and the costs with the prior nodes are noted in table 2.1.

2. The second iteration is completed in the same order as in the first iteration. With the second iteration, the first change is from node A with cost 6 to node D, giving

node D a cost of 10. This change also causes the cost to node F to change from 14 to 13 with prior node D. These changes are indicated in table 2.1.

3. The third and last iteration for this network has no changes in any of the costs. The algorithm exits from the while loop and the shortest path have been found. The shortest path has been indicated in figure 2.5(b).

Table 2.1 shows the results of the distances and prior nodes for each iteration of the worked example. The left column is the iteration numbers starting at iteration 0, which is the initializing step. The two columns D(v) and P(v), is the distance/cost to node v with the prior node of node v respectively. After the third iteration, no distance to any of the nodes has changed, causing the algorithm to exit the while loop.

Table 2.1: Bellman-Ford iterations example

	A		B		C		D		E		F	
Iter	D(v)	P(v)	D(v)	P(v)	D(v)	P(v)	D(v)	P(v)	D(v)	P(v)	D(v)	P(v)
0	∞	ϕ	∞	ϕ	∞	ϕ	∞	ϕ	∞	ϕ	∞	ϕ
1	6	C	-3	S	0	B	12	A	4	C	14	E
2	6	C	-3	S	0	B	10	A	4	C	13	D
3	6	C	-3	S	0	B	10	A	4	C	13	D

2.3.3 Floyd-Warshall algorithm

Several different shortest path problems exist. The first is trying to find the shortest path between one source node and one destination node, called the single-pair shortest path problem. Then there is the shortest path between a single source node and all other nodes, referred to as the single-source shortest path problem. Then the single-destination shortest path problem is finding the shortest path from all nodes to the destination node, which is a reversal of the single-source shortest path problem. The Floyd-Warshall algorithm can be used to solve the all-pairs shortest path problem, which is when shortest paths are found from all nodes to all other nodes [31] [23]. Independently the Floyd-Warshall algorithm was developed by Floyd and Warshall. This algorithm can also handle negative edge weights, but not a negative weight cycle [32].

Algorithm 3 Floyd-Warshall

Given a set of vertices V. The edge distance between nodes is given by d and the number of nodes contained in V is given by n.

1. **For** each iεV **do**
   For each jεV **do**
   If edge from i to j exists
   Then set Distance[i,j] = d(i,j)
   Else Distance[i,j] = ∞

2. **For** each iεV **do**
   For each jεV **do**
   For each kεV **do**
   If Distance[j,i] + Distance[i,k] $<$ Distance[j,k]
   Then set Distance[j,k] = Distance[j,i] + Distance[i,k]

The initial step for the Floyd-Warshall algorithm is to determine the distances between all adjacent nodes. If there exists an edge between adjacent nodes the cost is used, and if the edge does not exist, the cost is set as ∞. These values are entered into a table containing the minimum distance between nodes. Minimum distances are then determined when passing through intermediate nodes and compared to the values in the table. When the paths through intermediate nodes deliver shorter paths, the table is updated with the new shortest paths between the selected nodes [33]. The diagonal weights in the table will be zero since this is the distance from a node to itself. If the diagonal weights contain a negative value, a negative cycle has been detected, and the shortest paths cannot be found [34]. Figure 2.6 shows a network that will be used as a worked example for the Floyd-Warshall algorithm.

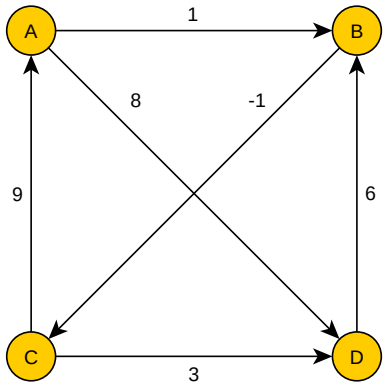


Figure 2.6: Floyd-Warshall: Example

The following steps were used to determine the all-pairs shortest paths problem. The final table shows the shortest path from each node in the network to every other node in the network.

1. The initial step finds the cost to its adjacent nodes. If the edge does not have a distance, a value of infinity is assigned as shown in table 2.2(a). Distances from the node to itself are zero, and this is the zeros found diagonally in the table.

2. In the first iteration, the first column and row are highlighted. If the highlighted row contains an infinite value, as the value from node A to node C, no values in that column can change. The same applies for the highlighted column containing infinity values. Therefore, in the first iteration the only values that can change are the path from node C to B and from node C to D. The highlighted column value is added to the highlighted row value, and if this added value is smaller than the existing value, a shorter path is found. After the first iteration, the only value changed is the path from node C to node B.

3. In the second iteration, the column and row B is highlighted, as shown in table 2.2(b). The same rules apply in terms of when the highlighted column contains an infinity value, the values in that row cannot change. In this table, there are no infinity values in the highlighted column however the highlighted row contains two infinity values. The only values that can change in this table are, therefore,

the path from node A to C and from node D to C. A shorter path for both these paths is found and is changed in the table.

4. The same steps are repeated for each iteration until the last column and row is highlighted as in table 2.2(d). In this iteration, it is unlikely to find an infinity value in the highlighted column or row. This means that all values can change if a shorter path is found, when adding the highlighted column value to the highlighted row value. After the fourth iteration in this example, the final shortest paths from each node to all other nodes are shown in table 2.2(e).

Table 2.2: Floyd-Warshall example

(a) First iteration

	A	B	C	D
A	**0**	**1**	∞	**8**
B	∞	0	-1	∞
C	**9**	∞	0	3
D	∞	6	∞	0

(b) Second iteration

	A	**B**	C	D
A	0	**1**	∞	8
B	∞	**0**	**-1**	∞
C	9	**10**	0	3
D	∞	**6**	∞	0

(c) Third iteration

	A	B	**C**	D
A	0	1	**0**	8
B	∞	0	**-1**	∞
C	**9**	**10**	**0**	**3**
D	∞	6	**5**	0

(d) Fourth iteration

	A	B	C	**D**
A	0	1	0	**3**
B	8	0	-1	**2**
C	9	10	0	**3**
D	**14**	**6**	**5**	**0**

(e) Final shortest paths

	A	B	C	D
A	0	1	0	3
B	8	0	-1	2
C	9	9	0	3
D	14	6	5	0

2.3.4 A-star algorithm

The A-star or A* algorithm is similar to Dijkstra's algorithm regarding adding edge weights to neighbour nodes to find the shortest path to the destination. However, the A-star algorithm makes use of heuristics, which implements a directional search and improves the performance in terms of computational speed. The A-star algorithm has three variables, as shown in equation 2.3. The value of f(x) is used to determine the order in which nodes are visited. The edge weight allocated to nodes, dependent on their distances, is given by g(x), and h(x) is an estimation of the cost for an optimal path from node x to the destination node [35].

$$f(x) = g(x) + h(x) \tag{2.3}$$

Algorithm 4 A-star

Given a network with source node S, destination node D and current node CN. Each node has an F, G and H property, where G is the weight gain to that node, H is the distance from the node to the destination node D, and F is (G + H).

1. **Set** closed-list = ϕ **and insert** S into open-list
2. **Set** G(S) = 0, H(S) = H_calculate(S,D) **and** F(S) = H(S)
3. **While** open-list != empty
   do CN = node with minimum(F) in open-list
   If (CN = D)
   Then return ShortestPath
   For each neighbour node N of CN
   If (N is in closed-list)
   Then nothing
   Else if (N is in open-list)
   calculate N's G, H and F
   If (G(previous) $>$ G(current))
   Set CN's G = G(current)
   N's parent = CN **and** re-add N to open-list
   Else calculate N's G, H and F
   N's parent = CN **and** add N to open-list

The A-star algorithm makes use of an open list and a closed list. The open list contains the current node and the nodes that have been neighbour nodes to previous current nodes. The closed list is simply all the nodes that have been current nodes before [36]. When using the A-star algorithm, the source node and each current node will be assigned an F, G and H value. The source node will have a weight of zero, which means that its F value will be equal to its H value. When the loop is started, the current node will be the node within the open list with the smallest value for F. The F, G and H values are determined for each neighbour node of the current node that is not in the open or

closed list. If the neighbour node is in the open list, a check is performed to determine if the new calculated G value is smaller than the value previously determined. If the new G value is smaller than the previous G value, it is updated and the parent node is changed. The process is repeated until the current node is the destination node. The shortest path is then found by looking at the parent of each node, starting at the destination node [37]. For a further understanding of the A-star algorithm, a worked example is done using the network in figure 2.7.

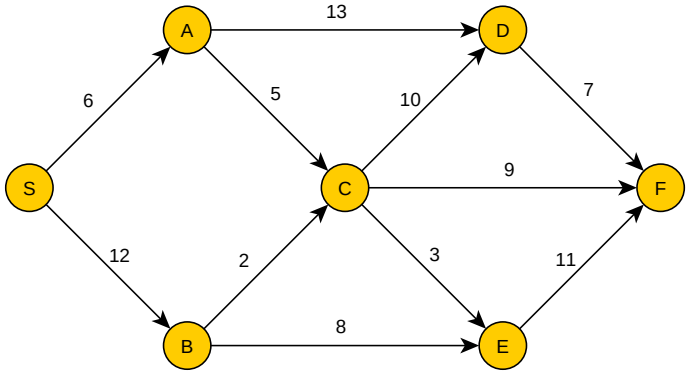


Figure 2.7: A-star algorithm example

The steps followed to determine the shortest path for the network in figure 2.7, will now be discussed. The nodes contained in the open list for each iteration are illustrated using table 2.3

1. In the initial step, the G, H and F values of the source nodes were determined. Table 2.3(a) shows the values obtained for the source node. As discussed earlier, the H value is an estimation of the cost for the optimal path and was selected as 20 for the source node. By coincidence the shortest path had a weight of 20; by choosing a higher value for H will not influence the shortest path found.

2. With the first current node set as the source node, table 2.3(b) shows the two path options available. These two options, node A and B are inserted into the open list, and their G, H and F values determined. These values are inserted into the table with both nodes having the source node as its parent node.

3. In table 2.3(b), node A has the smallest F value, therefore, node A is the new current node. Node A has two neighbour nodes it can communicate with, nodes C and D. These nodes are added to the open list, and their respective G, H and F values determined, as presented in table 2.3(c); both nodes have node A as their parent node.

4. The table illustrates that node C has the smallest F value and is added to the closed list. Node C has three possible path options to node D, E and F. Path D is already in the open list and the new weight value is compared with its previous weight value. The previous weight value to node D is smaller, which causes the parent node of node D to stay node A. The other two nodes are added to the open list, and their respective G, H and F values calculated.

5. In table 2.3(d), node F has the smallest F value. Node F is the destination node, and the shortest path was determined. The parent nodes are used to determine the shortest path, which is S-A-C-F.

Table 2.3: A-star example

(a) Open list iteration 1

CN	G	H	F
S	0	20	20

(b) Open list iteration 2

CN	G	H	F
A_S	6	14	20
B_S	12	11	23

(c) Open list iteration 3

CN	G	H	F
C_A	11	9	20
D_A	19	7	26

(d) Open list iteration 4

CN	G	H	F
E_C	14	11	25
F_C	20	0	20

2.3.5 Critical evaluation

To evaluate the above algorithms, we examined the type of algorithm to see whether it could be amended for our application. The Floyd-Warshall algorithm determines the shortest path from all nodes to all other nodes, since it was developed for an all pairs

shortest path problem. To determine the shortest path from all nodes to all other nodes in the proposed solution is excessive, therefore, the Floyd-Warshall algorithm will not be implemented in this study. The A-star algorithm is a single-pair shortest path algorithm. This algorithm cannot take other source nodes, or the landing sites already used, into consideration; it is only dependent on its own source node and destination node, therefore it is also not suitable for our application. Both Dijkstra and Bellman-Ford's algorithms are single-source algorithms, which determine the shortest path from the source node to all other nodes. To modify these paths is possible without making the implementation overly complex and therefore, Dijkstra and Bellman-Ford's algorithms were selected for implementation in this study.

2.4 Summary

At the beginning of this chapter, a number of relevant definitions were provided. The literature indicate that line of sight communication will be necessary in this study and that the transmitting distance will be heavily dependent on the height of the transmitter and receiver. With the collar around the rhino's foot and the curvature of the earth, it is clear why communication distance will be a problem.

Shortest path problems occur regularly in everyday life. One of the most common instances are determining a suitable route between two locations on a map, with endless possible paths between the two. Hence the uptake in GPS devices and applications by the general population.

Since this problem calls for redundancy in terms of communication routes between a source and a destination, k-path shortest path algorithms were investigated. The first algorithm of Suurballe and Tarjan forms a pair of edge-disjoint shortest paths. Edge-disjoint paths are formed when the paths do not share a single edge. Although Suurballe and Tarjan's algorithm can be modified to be vertex-disjoint, the process involves transformation of the graph, which increases the complexity of modifying the algorithm significantly. Yen's algorithm calculates k-shortest paths in a graph, where

each path k has the same root as path $k-1$ from source node to node i (R_i^K). Since this problem requires the two paths to be vertex-disjoint, Yen's algorithms cannot be used in this application. Both these algorithms make use of single shortest path algorithms as initial steps. This lead to the idea of using single shortest path algorithms and modifying them to address the research problem.

The literature showed that a number of shortest path algorithms exist, including single-pair shortest path, single-source shortest path, single-destination shortest path and the all-pairs shortest path. The A-star algorithm was the only algorithm to address the single-pair shortest path problem, while the Dijkstra and Bellman-Ford algorithms both address the single-source algorithms problem. The Floyd-Warshall algorithm is used to solve the all-pairs shortest path problem, determining the shortest path from all nodes to all other nodes. Since we regard the single-destination shortest path problem as simply being a reverse single-source shortest path problem, an algorithm for all these scenarios was discussed in this chapter.

Another property of the candidate algorithms is the ability to handle negative edge weights. Negative edge weights can be assigned to certain nodes for energy saving purposes, but will be discussed in Chapter 4. Both the Bellman-Ford and Floyd-Warshall algorithms can accommodate negative edge weights within its network. However, it must not contain a negative weight cycle. With a negative weight cycle, a shorter path will always exist and the algorithm will not be able to determine the shortest path. Both the Dijkstra and A-star algorithms require positive edge weights.

According to the authors in [38], the most popular algorithms to find the shortest path between vertices is the algorithms of Dijkstra, Bellman-Ford and Floyd-Warshall. The Dijkstra and Bellman-Ford algorithms are implemented in this study. The Floyd-Warshall algorithm was not used, since determining the shortest path from all nodes to all other nodes will be superfluous. The A-star algorithm was not implemented since it can only be used for single-pair shortest path applications. This algorithm cannot take other source nodes or the landing sites that are already used into consideration; it is only dependent on its own source node and destination node.

Chapter 3

Experimental setup

In this chapter, the experimental setup, as used to implement and evaluate the proposed algorithms, are presented. This include specifications, available data, assumptions as well as limitations associated with the problem that was presented in Chapter 1.

3.1 Assumptions and exclusions

Since all aspects of the real world cannot necessarily be modeled, some assumptions are made in order to simplify the model. Assumptions in this study include:

- There will only be one base station, hence, all paths will terminate at the same destination node.
- The maximum communication range of the collar mounted on the foot of the rhino and a UAV is 800m (this was specified by the client).
- The UAV that monitors the rhino will be labeled as the source UAV.

- All rhinos are monitored by a source UAV, and the position of this UAV is known beforehand. It is anticipated that an initial rhino discovery sweep will determine this position.
- Each source UAV is assigned to follow a specific rhino.
- The source UAVs will have a search and locate function to follow the rhinos.
- Landing positions are arranged as indicated in figure 3.1. Thus, neighbour landing positions are 1.6km from each other. This configuration allows for the largest area to be covered.
- UAVs are not limited to specific landing positions.
- While the UAV is in the air, it can communicate with other UAVs up to 3.2km away.
- In the case of an emergency, the UAV will hover higher than normal and try to send the emergency signal directly to the base station.
- If the UAV is not able to reach the base station directly, it will send the signal to the next UAV, which will do the same. This is done until the base station is reached using the predefined routing path depicted in figure 3.2.
- UAVs are airborne when a rhino moves outside the communication range of the source UAV and will then perform the search and locate function, when relocating to another landing spot, or when it sends data to the next UAV.
- The simulations were performed using a minimum area of 2500km^2. Given that the spacing between nodes is 1.6km, a simulation setup with 32 nodes in length and width, meant that the area was 51.2km x 51.2km in size.
- To prevent reverse communication in case of a node failure, each node in the primary and secondary path of a source node should be able to communicate with a node on another path. This will ensure that a directed acyclic graph is formed. The results are used by the communication protocol for this system to create routing tables for the nodes [39].

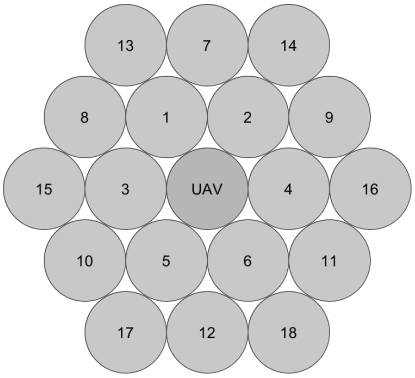
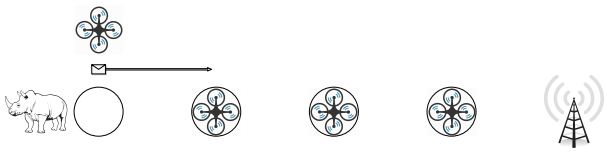


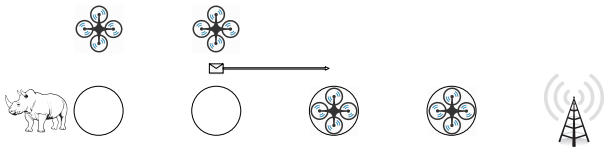
Figure 3.1: Demonstration of landing site spacing

The images in figure 3.2 demonstrate how a message is sent to the base station. UAVs ascend in order to forward the message to the next UAV until the base station is reached. Details of when each UAV ascend and descend are determined by the protocol developed in [39].

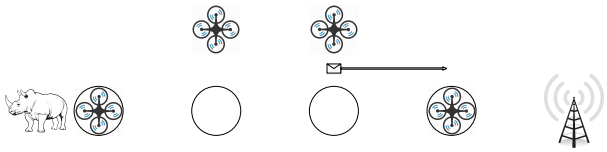
(a) Data message: Collar to UAV 1



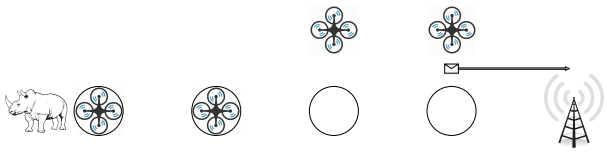
(b) Data message: UAV 1 to UAV 2



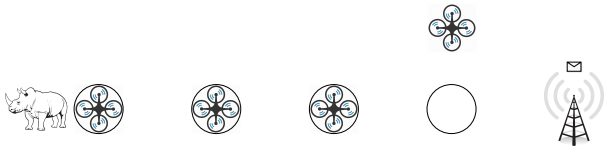
(c) Data message: UAV 2 to UAV 3



(d) Data message: UAV 3 to UAV 4



(e) Data message: UAV 4 to base station



(f) Data message: Delivered

Figure 3.2: Diagram describing the message forwarding scheme.

Communicating via an edge that is already used would waste a significant amount of energy, since it would imply that the UAV would need to ascend again for a single message. A directed acyclic graph needs to be formed, which ensures that a single UAV is never visited twice. This is in the case of a node failure and can be demonstrated

with figure 3.3. In figure 3.3, S and F nodes represent the source and destination nodes, whereas P1 - P3 and S1 - S3 represent the primary and secondary paths respectively. The primary paths (solid lines) and secondary paths (dashed lines) of certain nodes are indicated in the figure. When a message is sent along the primary path and node P3 experiences a node failure, node P2 has the option to send the message to node S3 and continue from there. If the primary and secondary paths were independent, node P2 would be forced to send the message back to the source node, which can then attempt to send the message via its secondary path. Node P2 sending its message back to the source node is referred to as reverse communication.

Node P3 does not have a secondary path, since it is assumed that the base station cannot experience failure. If the base station were to experience problems, sending the message to node S3 will result in an infinite communication loop between nodes P3 and S3.

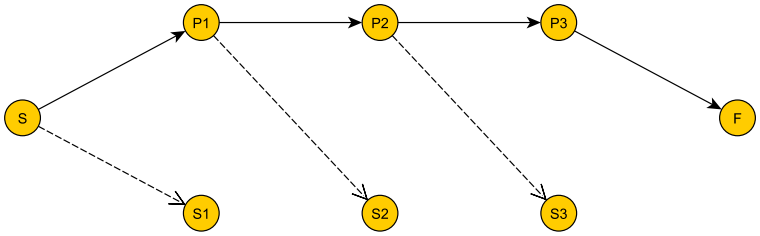


Figure 3.3: Reverse communication

3.2 Edge weights

Edge weights are assigned to edges for both Dijkstra's and the Bellman-Ford algorithm. These edge weights are assigned to represent the distances between nodes. Greater distances between two nodes result in higher edge weights. Figure 3.1 that was described earlier, shows a UAV with eighteen possible landing sites surrounding it. The edge weights to each landing site in the figure is shown using table 3.1.

Table 3.1: Allocated edge weights for nodes

Node Nr. (figure 3.1)	**Weight**
1-6	1
7-12	1.3
13-18	1.64

Table 3.1 illustrates that nodes 1-6 have an edge weight of 1; this is to communicate with the nodes around the UAV, which fall well within the communication range. The nodes 7-12 have an edge weight of 1.3, which is greater than nodes 1-6 because they are further from the UAV. Nodes 13-18 lie at the edge of the communication range and have an edge weight of 1.64. Though the physical distance to these last nodes is double those of nodes 1-6, fewer hops/UAVs are required. Figure 3.4 shows four possibilities for the source and sink node to communicate using only one intermediate node. The numbered nodes are the four possible scenarios for communication between the sink and source node using only one intermediate node. The number of nodes routing via nodes 1 and 2, as well as routing via nodes 3 and 4, will be the same. The edge weights of the nodes are allocated in such a way that scenarios 1 and 2 are preferred over scenarios 3 and 4. This prevents a communication link at the far edge of the communication range being followed by a short distance connection. Communication with nodes on the edge of the communication range is less reliable than with those nodes well within the communication range. For this reason, scenarios 1 and 2 have a higher reliability than scenarios 3 and 4 as the intermediate node, and this has been reflected in the allocated edge weights. No edge weights are assigned to nodes further than the eighteen nodes in the figure, which means that an infinitely large weight was assigned to them.

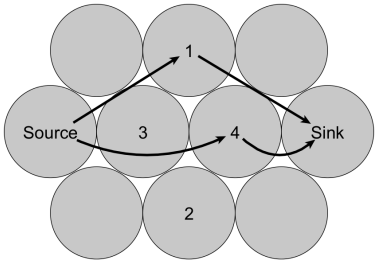


Figure 3.4: Hop possibilities

3.3 Summary

To simplify the model, certain assumptions and exclusions were made. Only one base station is used to receive all the data from the collars on the rhinos. Each of these rhinos will have a UAV nearby receiving its data and sending it to the base station or UAV repeater at certain time intervals. When UAVs transfer a message, they will elevate to increase the communication distance. UAVs will elevate higher than usual when attempting to send an emergency signal to the base station. If this fails, the signal is sent to the next UAV, which repeats the process until the base station is reached. Flight times are limited to when the UAV is searching for the rhino, relocating to another landing site or data is transferred. An important requirement is that each node of the primary and secondary paths must be able to communicate with a node on another path. This is mainly to prevent reverse communication in the case of a node failure, since communicating with a UAV already used would waste significant energy.

The edge weights allocated to the nodes in both the Dijkstra and Bellman-Ford algorithms are the same. The edge weights, chosen according to the distances between nodes, ensure that the communication path does not select a node at the extreme of the communication range followed up with a short distance connection. With the edge weights selected, the next chapter will discuss how the selected algorithms were manipulated with certain variables, in order to satisfy the implementation constraints.

Chapter 4

Implementation and verification

In this chapter, Dijkstra and Bellman-Ford's algorithms are implemented under the assumptions and constraints in Chapter 3. The algorithms were implemented on a similar basis, with different ways to determine the path between two points. The most significant difference between the two algorithms is that the Bellman-Ford algorithm can deal with negative edge weights, whereas with Dijkstra's algorithm, negative edge weights are not allowed. For the algorithms to comply with the specifications of our application as described in Chapter 3, some modifications were made. These modifications ensured the paths stay within communication range of each other, as well as some improvement modifications. QT Creator was used for the simulations and the Graphical User Interface (GUI) is shown in Appendix A.

4.1 Dijkstra's algorithm

This section contains the implementation of Dijkstra's algorithm. The original process is discussed to indicate how deficiencies were identified, and the modifications were implemented to improve the algorithm for this specific application. This is followed by the modifications section, which indicates the changes made as well as why these

changes were necessary to arrive at the final process. The final process indicates the process followed to arrive at the results found in Chapter 5.

4.1.1 Original process

Dijkstra's algorithm is a single-source shortest path algorithm used to determine the shortest path between two nodes. The distance between two nodes is the smallest value of the sum of the edge weights between these two nodes. For this application, redundant shortest paths are determined between two source nodes, or a source node and a base station. Dijkstra's algorithm is used to determine the shortest path between two nodes, then the primary path nodes are removed from the equation, and Dijkstra's algorithm is used again to determine the secondary shortest path. The nodes are connected when the primary and secondary paths between two nodes are found. All the source nodes should be connected to the base station, either directly or via another source node. The original process is shown, and discussed, using the flow chart in figure 4.1.

1. Determine the distances from each source node to the base station.
2. The closest source node to the base station is then connected to the base station with a redundant communication path.
3. The next source node is the source node closest to the base station, which has not yet been connected to the base station.
4. Determine the distance from the above source node to all the connected source nodes.
5. If the distance to the base station is smaller or equal to the distance to any other source node, the node is connected to the base station. If the distance to an already connected source node is shorter, the two nodes are connected to each other.

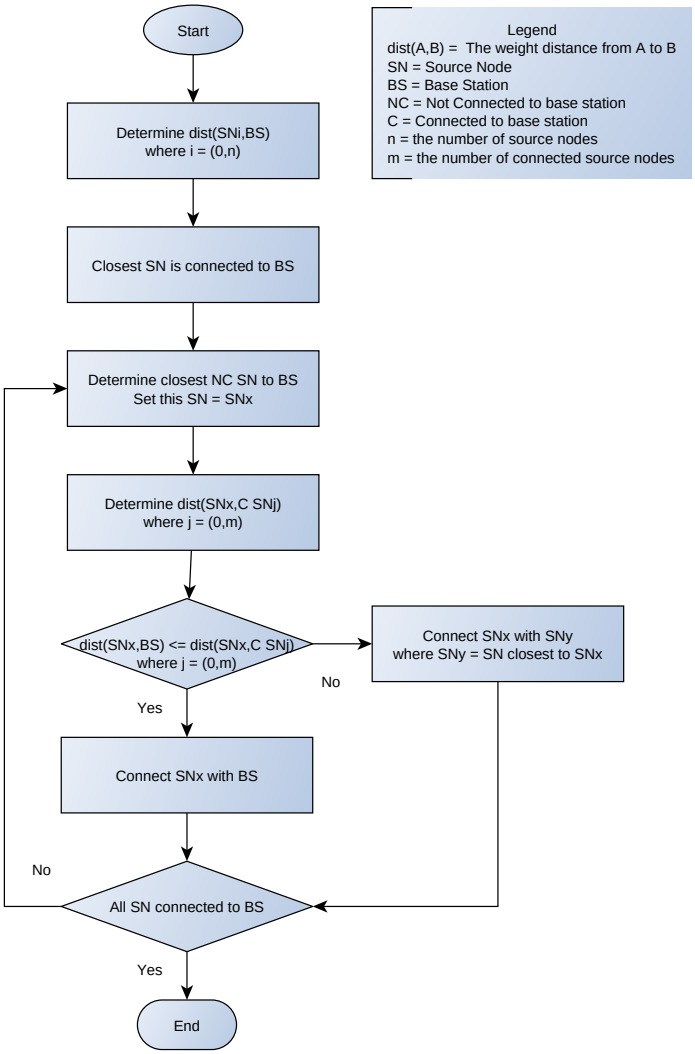


Figure 4.1: Original Dijkstra process flow

6. If all source nodes have a connection to the base station, the algorithm is completed. If there are still source nodes that need a connection to the base station, the process is repeated from step 3.

4.1.2 Modifications

Some deficiencies in Dijkstra's original algorithm were identified during the original process implementation. A modification made to use less landing sites is implemented using a weight benefit, which forces the source nodes to share routes. Another modi-

fication included preventing the primary and secondary path from separating. In the context of this dissertation, when the primary and secondary path separates, it implies that the nodes on these paths do not have a node on another path within its communication range. If the primary and secondary paths separate, node failure will cause the paths not to form a directed acyclic graph, thus wasting energy. It is one of the requirements for each node to have a node on another path within its communication range.

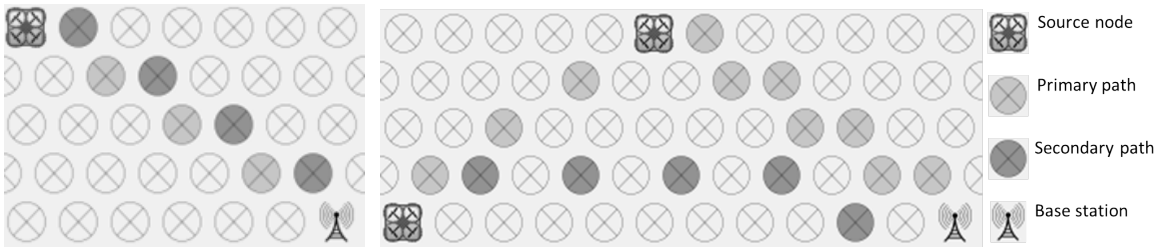
Using existing paths

The first problem with the above process is that a great number of landing sites are used around the base station. If the paths of source nodes close to each other share landing sites, it would reduce the number of UAVs needed to create the communication path. Reducing the number of UAVs further reduces cost. To manipulate new paths to connect to existing paths, a weight benefit is provided. The weight benefit is provided to nodes that utilise landing sites already in the path of other source nodes. The weight benefit is subtracted for these nodes, which reduces the total cost of the path using existing landing sites. The value for this reduction is explained in figure 4.3 and equation 4.1.

It is important that the reduction should not result in a negative edge weight. Firstly, all the edge weights should have a non-negative value when using Dijkstra's algorithm [27]. Secondly, when two consecutive existing path nodes are used and both have negative edges, an infinite cycle will be formed. An infinite cycle is formed since the addition of another node reduces the total edge weight of the path and a shorter path is found. To avoid a negative edge, the weight benefit cannot be higher than the smallest assigned edge weight, which is 1.

Each source node with an existing path has a primary and secondary route, as shown in figure 4.2(a). If the weight benefit used to manipulate the edge weights is too high, the primary route of the new path will tend to connect to all primary and secondary

nodes of the existing path (see figure 4.2(b)). This prohibits the secondary route from using existing path nodes and creates its own path occupying additional landing sites. The weight benefit of using existing nodes should be selected in such a way that the primary path does not use all the possible existing path nodes by taking big hops between existing path nodes.



(a) Scenario 1 (b) Scenario 2

Figure 4.2: An example where the weight benefit is too high, with the keys used in simulations.

In figure 4.3, nodes 1-3 represent landing sites of an existing path. The edge weight of path A is the sum of lengths A_1 and A_2, both of which have an initial edge weight of 1, while path B has an initial edge weight of 1.34, as discussed in Chapter 3. To use fewer nodes, path B is preferred over path A, which means the weight of path B should be less than that of path A.

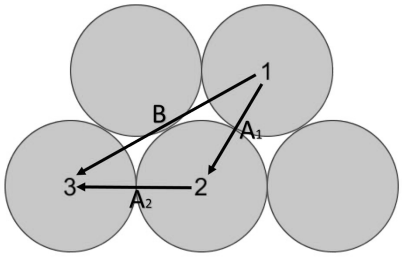


Figure 4.3: Existing path landing sites

Path B is the preferred path to use fewer nodes. For the algorithm to choose path B over path A, a weight benefit value has to be determined. A certain reduction will result in

the same weights for path A and B, called the threshold value. In equation 4.1, the left side represents path A, where the weight benefit is subtracted twice and the right side is path B. The weight benefit is subtracted twice in path A, once for each existing path node used and two existing nodes are used in path A. Path B uses only one existing path node, leaving another existing path node for the secondary path. Equation 4.1 shows the calculation of the threshold value, where weights will be equal.

$$1 - x + 1 - x = 1.34 - x \tag{4.1}$$
$$x = 0.66$$

The weight benefit value plays a significant role for new paths to choose existing path landing sites. The larger this value, the more existing path nodes will be used, however, using a value higher than 0.66, will cause the primary path to use all the nodes on the existing path. Leaving the secondary path with no nodes on the existing path. With a reduction of 0.65 the maximum number of existing nodes is used with both the primary and secondary path able to use landing sites on the existing path. For the secondary path to follow a similar path than the primary path, the weight benefit to use existing path landing sites is also 0.65.

Preventing separation

Simulations revealed the scenario in figure 4.4, which does not comply with the requirements. One of the requirements is that each node of the two paths should be able to communicate with a node on another path. The primary and secondary path become separated from each other to make use of existing landing sites and in doing so, not complying with one of the requirements documented in Chapter 3, which requires each node on the primary and secondary path to have a node on another route within communication range. This scenario is a result of the secondary path having no ties to the primary path. The deficiency was addressed by making a change to the algorithm of the secondary path. The solution is to create a preferred list containing all the landing sites within the communication range of the primary path. When calculating

the edge weights to landing sites, a weight benefit was subtracted for the landing sites within the preferred list.

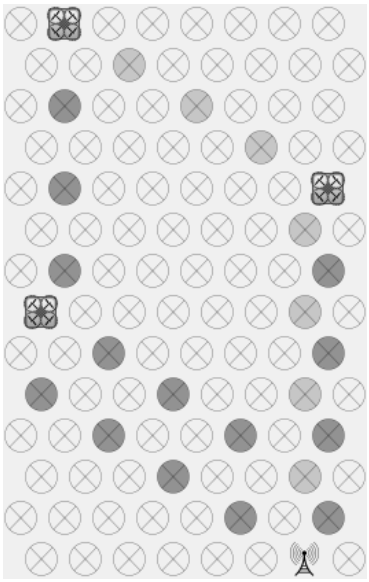


Figure 4.4: Redundant paths separated

The value for the weight benefit has an upper and lower limit. The upper limit will prevent a negative cycle from occurring and cannot be higher than the smallest positive edge weight between nodes, which is 1. It is possible that certain secondary path nodes can benefit from this value as well as the existing path value and therefore, these two values together should not exceed the positive edge weight of 1 (see equation 4.2). With the existing path nodes subtracting a benefit of 0.65, a negative cycle is prevented by setting the upper limit of this value to 0.35.

$$\begin{aligned} x + y &\leq 1 \\ 0.65 + y &\leq 1 \\ y &\leq 0.35 \end{aligned} \tag{4.2}$$

where x is the weight benefit for using existing paths and y is the upper limit to prevent disjointed paths.

The lower limit of the weight benefit value is used to keep the primary and secondary paths within communication range of each other. The lower limit of the weight benefit is the smallest possible value that will ensure this. To demonstrate the secondary path stays within the communication range of the primary path under all circumstances, it is important to look at the cases where the paths are most likely to become separated. The first step is to find the smallest θ in figure 4.5, where source node number 1 will join the existing path of source node number 2 rather than connect to the base station. In the next step, a source node (source node 3) is placed at the bottom with the same size for θ and the same distance to the base station as source node 2. The total cost from source node 1 to both source nodes 2 and 3 is the same because they are symmetric. The case where the primary and secondary path are most likely to become separated will only occur in the horizontal scenario, as shown in figure 4.5, and not vertically where the base station is at the bottom or top. This is because moving horizontally has a bigger edge weight than moving vertically due to the landing site placements.

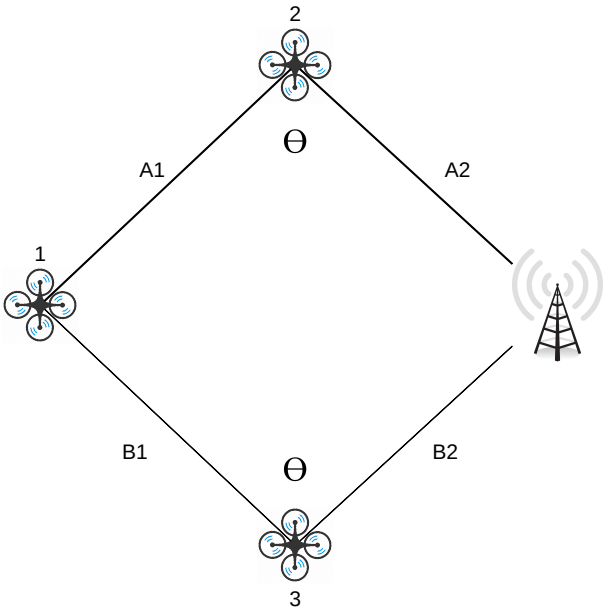


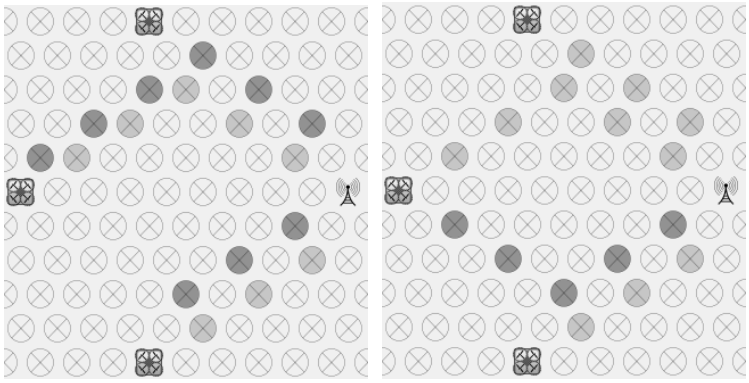
Figure 4.5: Variables effecting separating paths

The next step in finding the case where the primary and secondary paths are most likely to become separated is to determine the distance these source nodes are located from the base station. It was found that with shorter distances for the A and B components in figure 4.5, the tendency for the paths to become separated increased. There is

no difference in total cost for the primary path to form along route A (which is $A1 + A2$) or route B (which is $B1 + B2$). Since the primary path is formed using the shortest path along route A or B, the next shortest path will be equal to that of the primary path, however, on the other route (A or B). The current value being calculated is thus important to prohibit separating paths. The two scenarios in figure 4.6 show firstly in 4.6(a), the case where the value is above the lower limit and in 4.6(b), where the value is too low. The difference between these is that in scenario 1, the current value is subtracted six times, whereas in scenario 2, this value is subtracted twice. This implies that with a bigger scenario the impact of the two values subtracted from scenario 2 will be less. A smaller scenario is used because it is more likely for the secondary path to separate from the primary path. There are 18 landing sites within communication range of a single node, as shown in figure 3.1, with only three different edge weights assigned to these nodes. The value calculated is for the secondary path nodes that surround the primary path node. The nodes further away from the primary path nodes, have a weight benefit value of 0.01 less per node when the distance increases. In figure 4.6, not all the secondary path nodes in scenario 2 are within communication range of a primary path node and do not make use of the weight benefit. The minimum limit of this value is now calculated with equation 4.3.

$$1 + (5 * 1.3) + 1.64 - (3 * x) - 6y < (6 * 1.3) - (3 * x) - 2(y - 0.01) \tag{4.3}$$
$$1 + (5 * 1.3) + 1.64 - (3 * 0.65) - 6y < (6 * 1.3) - (3 * 0.65) - 2(y - 0.01)$$
$$7.19 - 6y < 5.87 - 2y$$
$$y > 0.33$$

where x is the weight benefit for using existing path nodes and y is the lower limit to prevent path separation.



(a) Scenario 1 (b) Scenario 2

Figure 4.6: Showing two scenarios where weight benefit to prevent separating paths is correct (a), and in (b) where this benefit is too small.

It is mentioned above that with smaller scenarios, the secondary path has a greater tendency to become separated from the primary path, however, the scenario in figure 4.6 is not the smallest possible scenario. The paths of a smaller scenario are separated, but all the nodes of the primary and secondary path are within the communication range of two other nodes, as shown in figure 4.7.

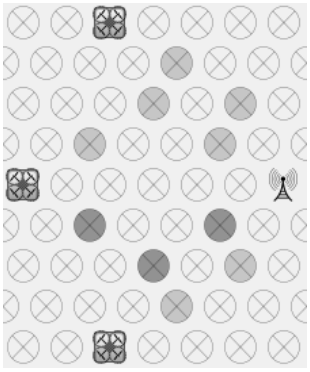


Figure 4.7: Small separated paths scenario

All weight benefits subtracted from the original edge weights have an effect on the shortest path between two points. The existing path value adjusts the shortest path of the primary and secondary route to use existing path landing sites, which may result in a longer route but occupies less landing sites. The value in this section changes the

shortest path of the secondary route to remain within the communication range of its primary route. Ultimately, a higher value causes more changes to the shortest path, therefore, the smallest value for the paths to remain within communication range is selected. Equation 4.3 shows that the value should be higher than 0.33 and is set at 0.34 for this algorithm.

4.1.3 Final process

The modified Dijkstra's algorithm ensures that source nodes prefer to share landing sites and prevents the primary and secondary paths from separating. The source nodes are connected with the base station, using the modified Dijkstra's algorithm to determine the paths. The algorithm was implemented, and discussed, following the process illustrated in figure 4.8.

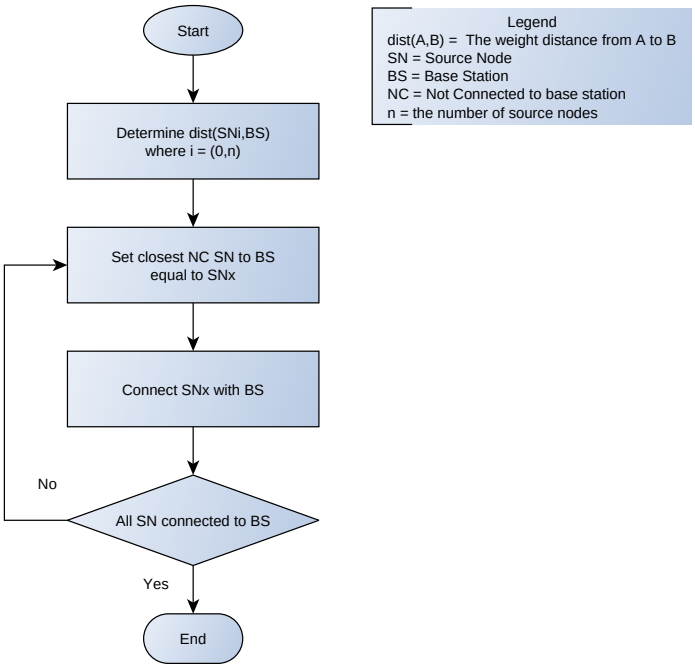


Figure 4.8: Final Dijkstra process flow

1. Determine the distances from each source node to the base station.

2. The closest source node that is not connected to the base station is used.

3. The above source node is connected to the base station.

4. If all source nodes are connected with the base station, the algorithm is completed. If there are still source nodes that need to be connected to the base station, the process continues from step 2.

With the original process, a source node was connected to either the base station or the closest non-connected source node. The closer one of the two was selected to use less landing sites, but it also increased the complexity of the algorithm. The modifications made to the original Dijkstra's algorithm ensures that the paths make use of occupied landing sites on its route to the base station, by manipulating the edge weights of nodes on existing paths.

In a network with a large number of source nodes, Dijkstra's original algorithm uses excess landing sites that could have been shared. The modified algorithm eliminates the use of the excess landing sites by forcing the source nodes to share paths. This is the main difference between implementing the original and the modified Dijkstra's algorithms, as discussed in this chapter.

Although the modified Dijkstra's algorithm delivers usable results, it lacks a certain property. The Bellman-Ford algorithm can assign negative edge weights to nodes. These negative edge weights can be assigned to certain landing sites, which have advantages that are discussed in the next section.

4.2 Bellman-Ford algorithm

This section is dedicated to the implementation of the Bellman-Ford Algorithm. The distinctive difference between the Dijkstra and Bellman-Ford algorithms, is negative edge weights are implemented in the latter. Other modifications were made to utilise less landing sites and to comply with the specified requirements. The process implemented for the Bellman-Ford algorithm is described after the modifications section.

Before continuing with the implementation of the Bellman-Ford algorithm, it is necessary to discuss the importance of this algorithm. The Bellman-Ford algorithm has the following application advantages over Dijkstra's algorithm for this system:

- Landing sites around the base station are regularly used, therefore, repeater towers can be placed around the base station, and negative edge weights can be assigned to these towers.
- Negative edge weights can be assigned to nodes with more energy. Certain nodes can have more energy due to short flight times, or better charging capabilities.
- An advantage can be given to nodes with a longer communication reach due to the positioning of the landing site and the geometric surroundings.
- High energy repeaters can be placed on existing infrastructure, such as windmills, water towers or storage spaces. Assigning negative edge weights to these repeaters could also be advantages.

4.2.1 Modifications

To utilise the advantages discussed above requires the implementation of negative edge weights to the Bellman-Ford algorithm. The first topic discussed is how negative edge weights were implemented for the algorithm. The purpose of the second modification is to use less landing sites, by sharing paths.

Negative edge weights

The first distinctive attribute of the Bellman-Ford algorithm is the ability to handle negative edge weights. Negative edge weights were assigned to selective nodes contained in landing sites of existing paths. Negative edge weights cannot be assigned to two existing landing sites within communication range of each other since this will result in a negative cycle and no shortest path will be calculated.

The landing sites with assigned negative edge weights are not defined prior to each simulation. Instead, a process is used to identify the negative nodes selected from already used landing sites. The process of selecting the nodes with assigned negative edge weights is shown in figure 4.9. The process makes use of an existing path landing site list as well as a negative list. The existing path landing site list contains the landing sites that are used in the existing paths while the negative list contains the nodes to which negative edge weights are assigned. The closest node to the source node is identified, removed from the existing path landing site list and inserted into the negative list. Since the nodes within the negative list will have a negative edge weight, the weight benefit for existing path landing sites does not have to be subtracted from them. The existing path landing site nodes within three hops (A hop is a jump from one node to one of its adjacent nodes) of the first negative node are removed. The nodes within two hops are removed since it is in the communication range and will result in a negative cycle. The reason for removing nodes within three hops is further illustrated in figure 4.10. The path will prefer to use all the nodes within the negative list. When removing the nodes within two hops, route A (A1 + A2) will be followed, and when nodes within three hops are removed, route B (B1 + B2) is used. Route B is preferred over route A to utilise less landing sites, therefore, the nodes within three hops are removed from the negative list. Negative list nodes can only be assigned to nodes still contained in the existing path landing site list. The next negative list node is the closest node to the previous node added. The process is repeated until the existing landing sites list is empty.

An edge weight of -0.1 was assigned to the nodes in the negative list. The smaller this value, the more paths of source nodes are forced to connect to the negative edged nodes. The negative edged nodes are not always in the direction of the base station, therefore, it is not preferable that source nodes connect to all the nodes in the negative list. The negative value is selected to have the smallest possible impact as a negative edge weight, since it already has an impact on the total path cost.

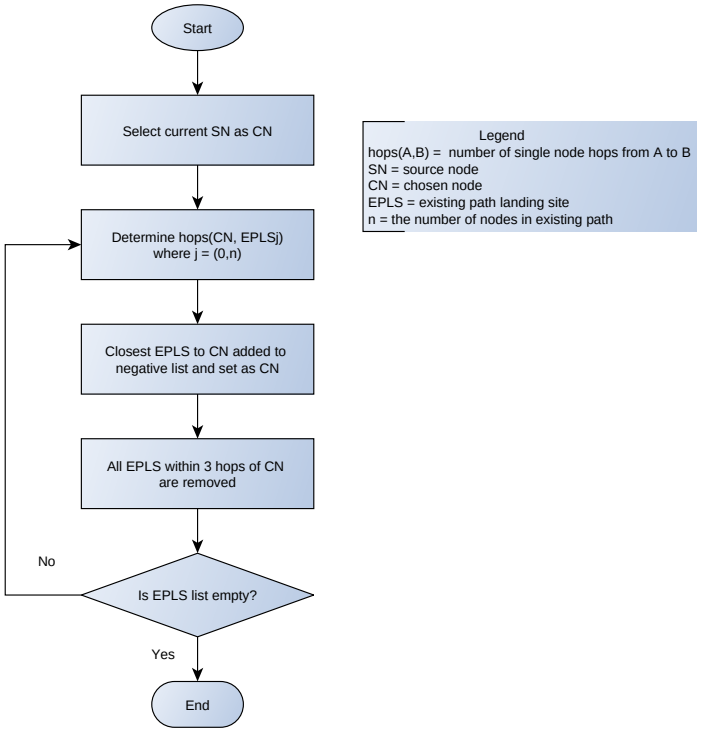


Figure 4.9: Selecting nodes for negative edge weight

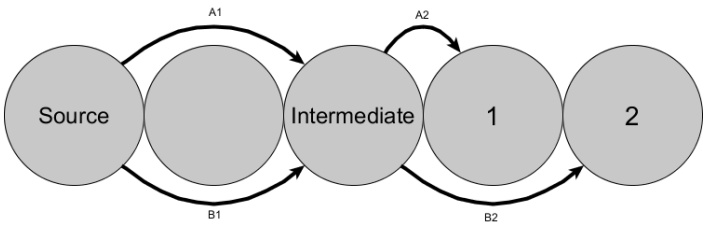


Figure 4.10: Removing three hops for negative list

Using existing paths

A weight benefit is used to ensure that source nodes share paths in order to use less landing sites. The weight benefit is subtracted for nodes using landing sites of an existing path. The value of the weight benefit was determined the same way as for Dijkstra (see section 4.1.2). The value was selected as high as possible for the algorithm to use the highest possible number of the already utilised landing sites. Having a value of 0.65, it is small enough to prevent the primary route from using all the existing path nodes.

Preventing separation

As with the Dijkstra's algorithm, the secondary routes of the modified Bellman-Ford's algorithm require ties to the primary route to prevent the paths from separating. This occurs when the primary and secondary paths make use of occupied landing sites in different directions. In doing so, some nodes on the primary path cannot communicate with those on the secondary path and vice versa. The main reason to prevent the primary and secondary paths from separating, is to ensure that each node on both these paths has a node on another path with which to communicate. To do this, the secondary path nodes within the communication range of the primary path nodes were placed into a list. A benefit weight was subtracted from the edge weights of all the nodes in this list, reducing its cost.

To determine the weight benefit that is subtracted, the upper and lower limit needs to be determined. The upper limit is the highest possible value that does not cause a negative cycle while the lower limit is the minimum value that can be used to stay within the constraints. The upper limit is calculated similarly to that of Dijkstra's algorithm, except that more variables are affecting the modified Bellman-Ford algorithm. To calculate the upper limit of the current value, the negative edge weight, existing path weight and the smallest assigned edge weight (that is 1) are used. To prevent negative cycles, the weights between two nodes should stay positive. Since the negative edge weight node already has a negative cost of -0.1, the upper limit weight and existing path weights added together should not be higher than 0.9. The upper limit for the current value to prevent separated paths is therefore 0.25, as shown in equation 4.4. As the upper limit of this value is smaller than the value used for Dijkstra's algorithm (0.34), it is an indication that this method might not be sufficient to prevent separated paths. Within one simulation, it was visible that using this method with a value of 0.25, is insufficient, and the paths separated from each other. To use this method, the value should be increased, which cannot be done without causing a negative cycle.

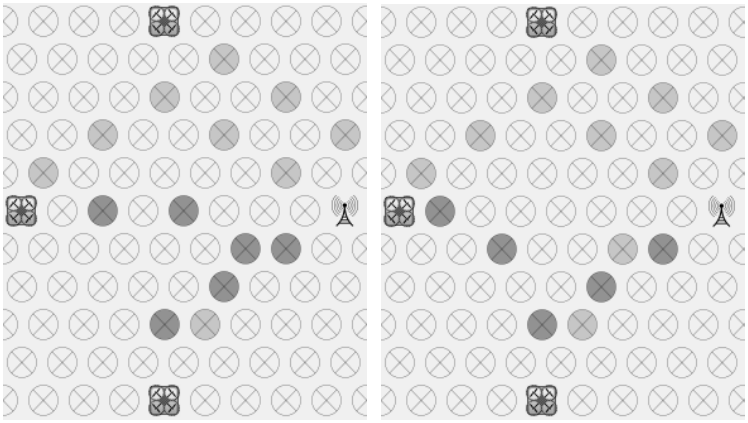
$$\begin{aligned} x + y &\leq 1 - z \\ 0.65 + y &\leq 1 - 0.1 \\ y &\leq 0.25 \end{aligned} \tag{4.4}$$

Where x is the weight benefit for using existing path nodes, y is the upper limit to prevent separated paths and z is the negative edge weight.

The above method subtracted weights to benefit secondary path nodes within the communication range of the primary path nodes. Instead of subtracting a weight, the second method will add a weight to secondary path nodes not within communication range of the primary path nodes. One of the advantages of this method is that there is no upper limit for this value. An upper limit is used to determine a value that ensures that a negative cycle doesn't exist or to stay within the requirements by preventing separated paths. The upper limit value of the second method does not influence either of the two requirements discussed, therefore, it has no upper limit.

The lower limit of the weight benefit value still needs to be calculated before selecting a value. The lower limit will serve the same purpose as in the first method, where the lower limit is the smallest value that ensures the paths do not separate. To ensure this for all circumstances, it is proven that the two paths will not separate for the scenario in which these paths are most likely to do so.

The scenario where the primary and secondary paths are most likely to separate is determined the same way as for Dijkstra's algorithm, using figure 4.5. The first step is to find the smallest θ where source node 1 connects with source node 2, rather than to connect directly to the base station. A source node is placed in the mirror image of source node 2, which has the same size for θ. The next step is to find the distances for A and B where the scenario is most likely to separate. As found in section 4.1.2, shorter lengths for A and B render a bigger tendency for the primary and secondary paths to separate. By using the scenarios in figure 4.11, the lower limit for this value is determined in equation 4.5.



(a) Scenario 1 (b) Scenario 2

Figure 4.11: Showing two scenarios where the weight benefit to prohibit separation is correct (a) and in (b) where this benefit is too small.

$$(2 * 1.3) + (2 * 1.64) - (x + z) < 1 + (3 * 1.3) - x - (2 * z) + 3y \tag{4.5}$$

$$(2 * 1.3) + (2 * 1.64) - (0.65 + 0.1) < 1 + (3 * 1.3) - 0.65 - (2 * 0.1) + 3y$$

$$5.13 < 4.05 + 3y$$

$$y > 0.36$$

Where x is the weight benefit for using existing path nodes, y is the upper limit to prevent separating paths and z is the negative edge weight.

The shortest path is affected with every value added and subtracted from the original edge weights. The effect of the current benefit value is the same for all values higher than the lower limit. The exact value for this weight benefit does not matter, any value greater than 0.36 can be used; the value 5 was selected for this algorithm.

4.2.2 Final Process

The final process for the modified Bellman-Ford algorithm, is a combination of all the modifications to connect source nodes with the base station, occupying as few landing sites as possible. The existing path weight benefit and negative edge weight nodes within the modified Bellman-Ford algorithm forced source nodes to use landing sites on the existing paths. The process flow of the modified Bellman-Ford's algorithm is illustrated, and discussed, in figure 4.12.

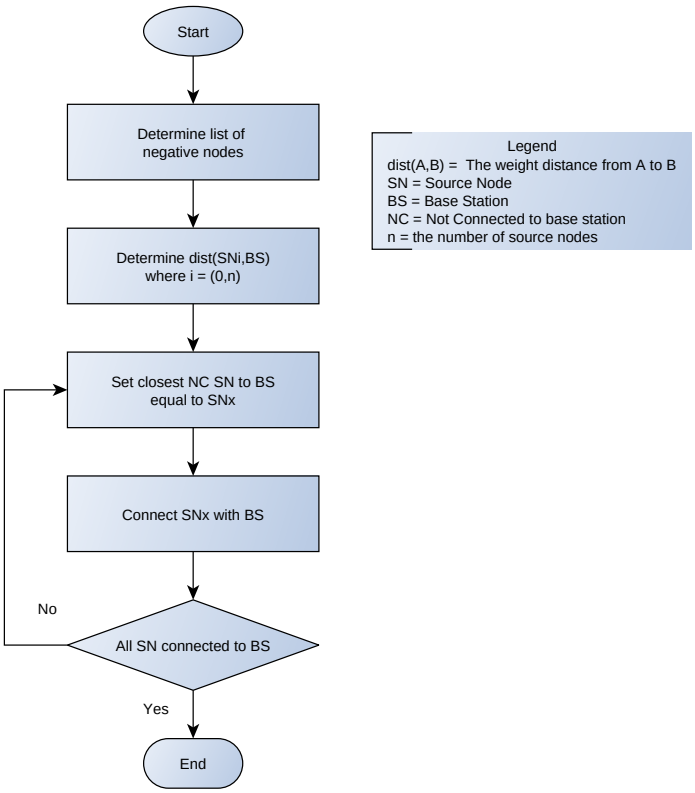


Figure 4.12: Final Bellman-Ford process flow

1. At first, the nodes in the negative list are determined. These nodes are existing path landing sites, not within communication range of each other.

2. The distance from each source node to the base station is determined.

3. The closest source node that is not connected to the base station is selected.

4. The selected source node is then connected to the base station with a redundant path.

5. The algorithm is completed when all source nodes are connected to the base station. If there are source nodes that are not connected to the base station, the process continues from step 3.

The modified Bellman-Ford algorithm was implemented instead of the original Bellman-Ford algorithm to ensure that paths are shared. This will ensure that less landing sites are occupied. The original algorithm will determine the shortest path between each source node and the base station, while the modified algorithm forces source nodes to share paths to the base station. Sharing paths will result in less UAVs occupying landing sites.

4.3 Implementation output

The results obtained in this study are used by the communication protocol to create routing tables [39]. This section will discuss the format of the output from the results obtained with the implementations above. The output is formatted to match the input used for the communication protocol. The format of the output is the same for both the modified Dijkstra and Bellman-Ford algorithms.

The modified Dijkstra and Bellman-Ford algorithms calculates redundant paths from each source node to the base station. This ensures that for each node, there is another on a separate communication path with which it can communicate. When calculating the output, Dijkstra's algorithm was used to determine the shortest path from each node (formed with the modified algorithms) to the base station. Each of these nodes should have two paths with different starting positions. It is important to note that the output secondary path may use the nodes in the output primary path subsequent to the failed node when it is a shorter path.

To understand figure 4.13, it is important to note that the simulation area contains 32 landing sites in length and width (as discussed in Chapter 3). The landing sites were identified by numbering them from 1 to 1024, horizontally and downwards. The figure shows the output of a scenario that is sent to the communication protocol. The first number in each of the two rows is the starting node. The two rows represent the shortest and second shortest path, with different starting positions from the starting node to the base station. The B at the end of the row represents the base station that is reached. The output contains both paths from all the occupied landing sites.

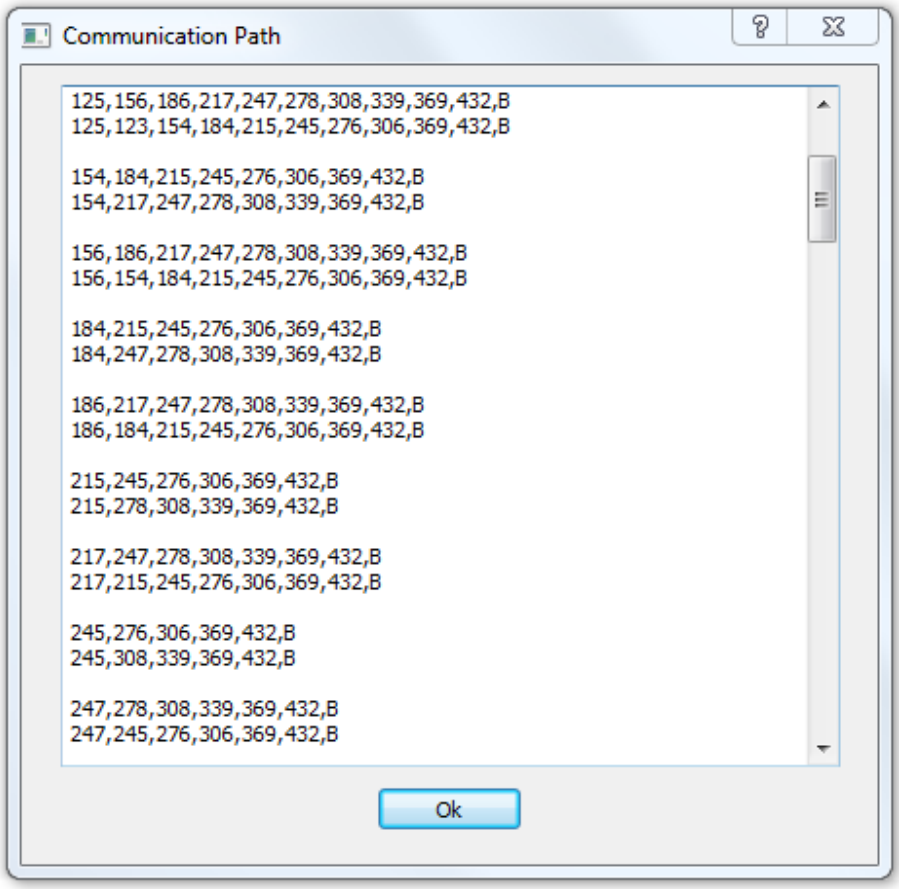


Figure 4.13: Implementation output

To verify that the output data complies with the requirements set in Chapter 3, two checks can be performed. The first check will verify that the first and second path of a node does not start at the same node. After the first and second path start at

different nodes, the second test will confirm that these nodes are not contained in its opposite path. This can be explained using figure 4.14, which is an example for the first two lines in figure 4.13, where a solid arrow represents the primary route and the dashed arrow represents the secondary route. In figure 4.14, the starting node is nr 125, whose primary path starts at node 156, and the secondary path starts at node 123. The first check discussed ensures that the primary and secondary path for node 125 does not start at the same node. In the event of a node failure at node 156, node 125 will send the message to node 123. This is where the second check is important. The second check ensures that node 156 (failed node) is not in the secondary path of node 125. If node 156 was in the secondary path, the communication link would be broken although other options exist.

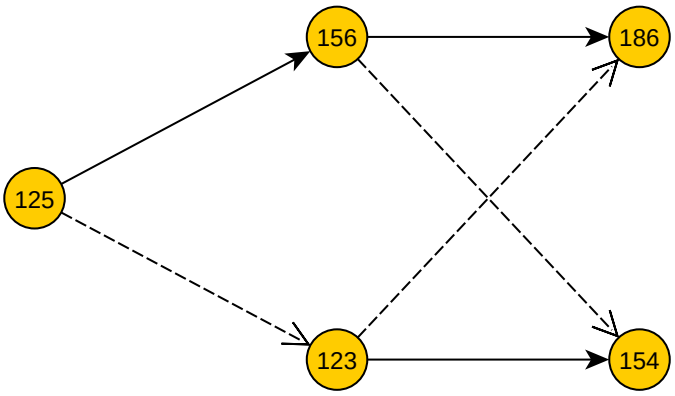


Figure 4.14: Output requirements

4.4 Summary

At the beginning of this chapter, the first implementation of Dijkstra's algorithm was discussed, after which the deficiencies of this implementation were pointed out. The deficiencies were addressed by changing its operation and implementing a weight benefit to landing sites already used. The weight benefit forced source nodes to share landing sites, but also created another deficiency. This required the modification to prevent the primary and secondary paths of a source node from separating. Preventing the

two paths from separating ensures that all the nodes on the primary and secondary paths have a node on another path within its communication range. It was achieved by adding another weight benefit to the secondary path, for staying within communication range to the primary path. After the modifications, the final process that was implemented for Dijkstra's algorithm was discussed.

The main difference between the Dijkstra and Bellman-Ford algorithms is the negative edge weights. It was discussed how negative edge weights can be assigned to nodes with energy advantages caused by different situations. The Bellman-Ford algorithm can make use of these nodes with an energy advantage or a longer communication reach. The advantages of the modifications to Dijkstra's algorithm were enough to skip the original process for the Bellman-Ford algorithm and directly implement the same modifications. The additional modification was the implementation of nodes with negative edge weights. Certain landing sites already in use were assigned negative edge weights, to increase the number of landing sites shared. After the modifications, the final implementation of the Bellman-Ford algorithm was discussed using a flow diagram.

The output of the implementations above was also discussed. The output is used as the input for the communication protocol. The format of the output was shown and two checks verified that the output complied with the requirements. The first check ensured that each node had a primary and secondary path to the base station. The second check ensured the node at the start of the primary path was not in the secondary route and vice versa. With successful checks performed, the output illustrates the primary and secondary communication path for each of the occupied nodes.

This chapter illustrated how the two algorithms were modified for the application specified in Chapter 3. Both algorithms create two paths from each source node to the base station. The algorithms allow new paths to benefit from sharing landing sites with existing paths and ensures each node on all the pairs of paths stays within communication range of a node on another path. The theoretical implementation of this chapter can now be simulated to obtain the results.

Chapter 5

Results and validation

In this chapter, the two algorithms of Dijkstra and Bellman-Ford are implemented in the simulation environment as discussed in Chapter 4. The results of these simulations are used to determine how effective each algorithm operates. The tables of these results are provided in the Appendix B. This chapter forms part of the validation by confirming that the result of each simulation complies with all the specified requirements before it can be used. The data shows the effect of using more source nodes as well as spreading the source nodes.

5.1 Dijkstra's algorithm

This section contains the results of simulations with modified Dijkstra's algorithm. Firstly, the effects of increasing the number of source nodes are shown using data derived from multiple scenarios of simulations. This is also portrayed with figures that clarify the influence the number of source nodes has on the algorithm. This is followed by a short section illustrating the significant impact the spread of the source nodes has and finally, a brief description of the algorithm's running time is included.

A Monte Carlo analysis approach was used for the simulations, which is useful when performing random sampling. The results were obtained by simulating twenty different random scenarios for each number of source nodes. Twenty scenarios were used since the average landing sites used for each number of source nodes converged. Simulation with one source node was left out, as there are no other source nodes with which to share paths. The simulations contain each integer value of source nodes, from two up to ten source nodes. Each simulation was inspected to ensure it complies with all the requirements set out in Chapter 3 before it was used, validating the algorithm. The following three properties of each simulation were saved:

- Individual Path Node Count (IPNC). This is where the number of landing sites used for all the source nodes to connect to the base station individually, are added together. This is a naive representation, which would have been used for these source nodes if they were not sharing landing sites.
- Cumulative Paths Node Count (CPNC). This is the number of landing sites used with the implementation of the modified Dijkstra's algorithm discussed in the previous chapter. All source nodes are selected at the same time to benefit from existing landing sites.
- Running time. This is the processing time from the beginning to the end of the simulation.

5.1.1 Increasing source nodes

Goal: This section attempts to display how the modified Dijkstra's algorithm is affected when the number of source nodes increases. The IPNC and CPNC are plotted together to show that the modified algorithm has a positive difference regarding the number of landing sites used.

Figure 5.1 is the first visual representation, where the number of landing sites used is plotted against the number of source nodes. The x-axis represents the number of

source nodes used in the simulation. The y-axis represents the number of landings sites used, which is the average of the multiple different scenarios with its respective number of source nodes. The top line represents the number of landing sites used when the source nodes are connected to the base station individually and then added together (IPNC). This line is linear because no landing sites are shared and as source nodes are added the number of used landing sites will keep rising. The bottom line is the result when all source nodes are simulated together using the modified Dijkstra implementation (CPNC). This line is not linear and will have an asymptote where the number of landing sites used does not increase with the increase of source nodes. The area between these two lines shows the number of landing sites saved with this implementation of Dijkstra.

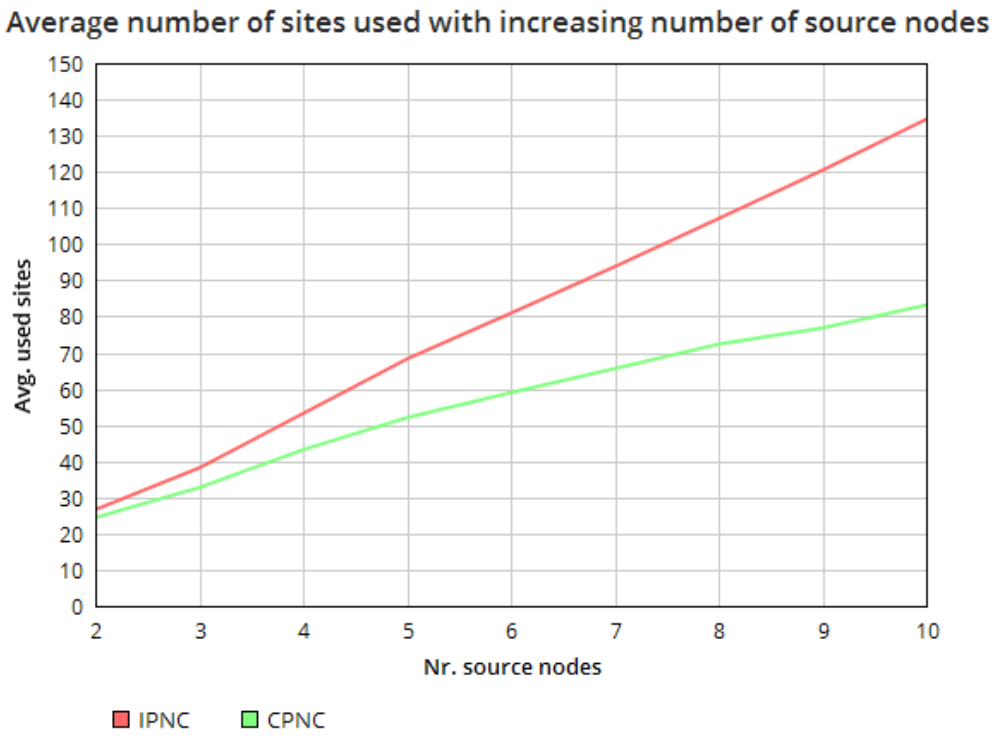


Figure 5.1: Dijkstra: The average sites used for IPNC and CPNC

Goal: The following is used to show the number of landing sites saved using the modified Dijkstra's algorithm. The figure will show the proportion of change in the number of landing sites saved as the number of source nodes increase.

The number of landing sites saved using this implementation of Dijkstra was briefly mentioned above. To determine the number of landing sites saved, the CPNC values are subtracted from the IPNC values for each number of source nodes. This is plotted with similar axis as in the previous figure, except the y-axis is changed from the average landing sites used to the average landing sites saved. Figure 5.2 illustrates how the number of landing sites saved increases when the number of source nodes increase. It is visible that this line is not linear and increases with a bigger margin as source nodes are added; this is because more source nodes lead to more landing sites being shared. The number of landing sites saved will increase with each addition of a source node until a source node has been placed at all the landing sites.

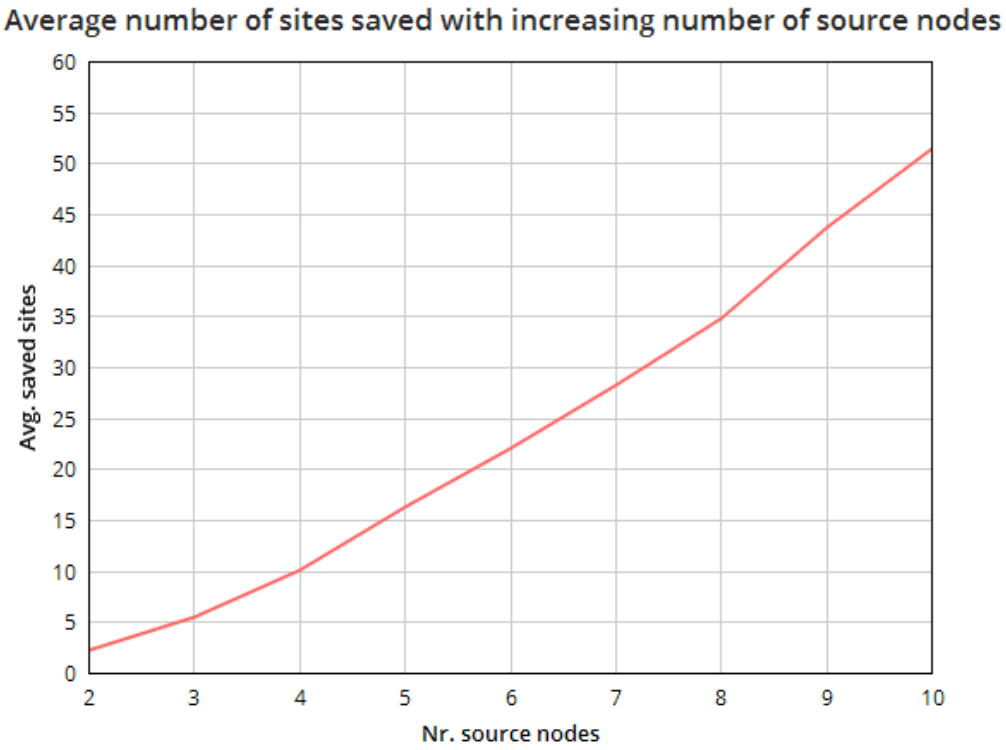


Figure 5.2: Dijkstra: The average sites saved with this implementation

5.1.2 Spread dependent

Goal: This section is used to demonstrate that the number of landing sites saved is dependent on the spread of the source nodes.

Six source nodes were evenly spread, confined to 90°, 180° and then 360° sectors. To eliminate the impact of the distance, the IPNC value was kept constant for all the sectors. Different IPNC values were used to indicate that the algorithm remains spread dependent on varying distances to the base station. Table 5.1 shows the number of landing sites used and sites saved for the three different sector ranges. The table shows the number of landing sites saved increases as the source nodes are confined to smaller sectors. The number of landing sites saved can be doubled if the source nodes are spread in half the sector size.

Table 5.1: Dijkstra: Spread dependent

	90° sector		180° sector		360° sector	
IPNC	CPNC	Saved	CPNC	Saved	CPNC	Saved
100	53	47	79	21	94	6
80	36	44	59	21	73	7
60	30	30	43	17	55	5
40	14	26	28	12	37	3
20	4	16	11	9	16	4

5.1.3 Running times

Goal: To comment on the feasibility and practicality of this system, it is important to show the running times for the modified Dijkstra's algorithm.

Figure 5.3 shows the average time, in seconds, the algorithm runs for each number of source nodes. This is a relatively linear line, as the time increases evenly when a source node is added. Ten source nodes ran for an average of 132 seconds, with a maximum run time of 169 seconds.

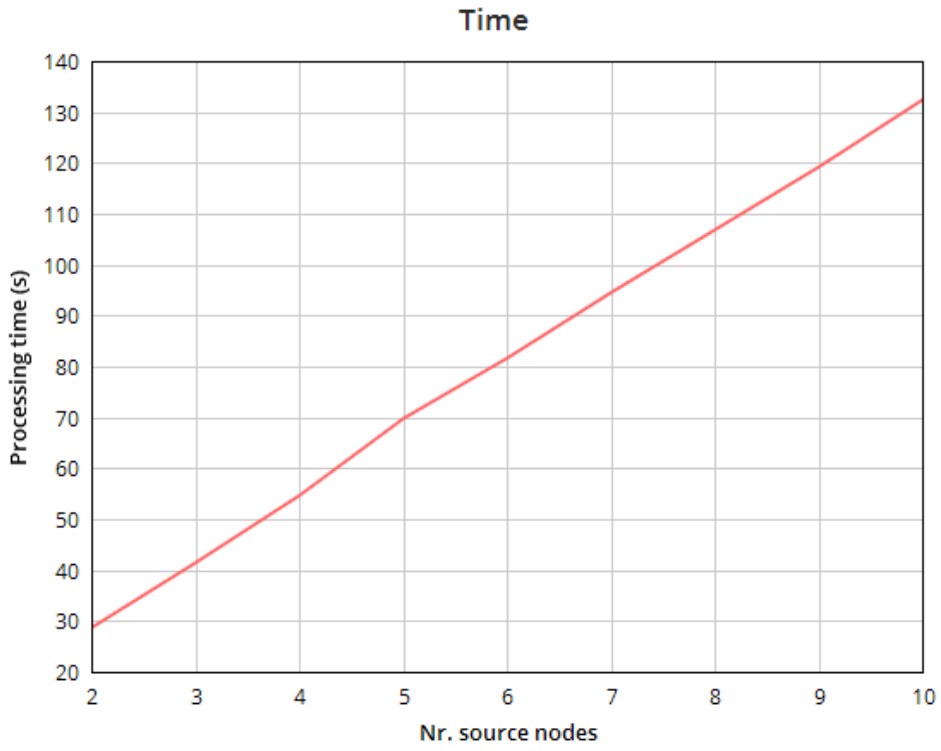


Figure 5.3: Dijkstra: The average algorithm run time

5.2 Bellman-Ford algorithm

The Bellman-Ford algorithm can contain negative edge weight nodes that can be assigned to landing sites and structures with more energy. The modified Bellman-Ford results will contain the number of landing sites that can be used and saved for a different number of source nodes. A table will be used to show how this algorithm is dependent on the spread of the source nodes, followed by its running times. It is important to note that for the simulations of the modified Bellman-Ford algorithm, the source nodes were placed on the same landing sites as with the modified Dijkstra's algorithm above; doing this makes it easier to compare the algorithms to each other. To eliminate the impact of distance when determining spread dependency, it was ensured that both algorithms occupy the same number of landing sites. This meant the source nodes were not necessarily placed on the same landing sites.

5.2.1 Increasing source nodes

Goal: This section will showcase how the modified Bellman-Ford algorithm reacts as the source nodes increase. The number of source nodes is increased and plotted for IPNC and CPNC to display how the number of saved landing sites changes.

In figure 5.4 a graph of the number of landing sites used against the number of source nodes is provided. The y-axis is the average used landing sites for each number of source nodes obtained from multiple simulations. The top line in this figure is represented by the number of landing sites used when source nodes are connected to the base station individually and then adds these values together (IPNC). This is a naive representation of the number of landing sites that will be used when paths are not shared. With the top line, no landing sites are saved and, therefore, the line is linear. The bottom line shows the number of landing sites used when using the modified Bellman-Ford algorithm as discussed in Chapter 4 (CPNC). This line curves away from the linear line as the source nodes increases.

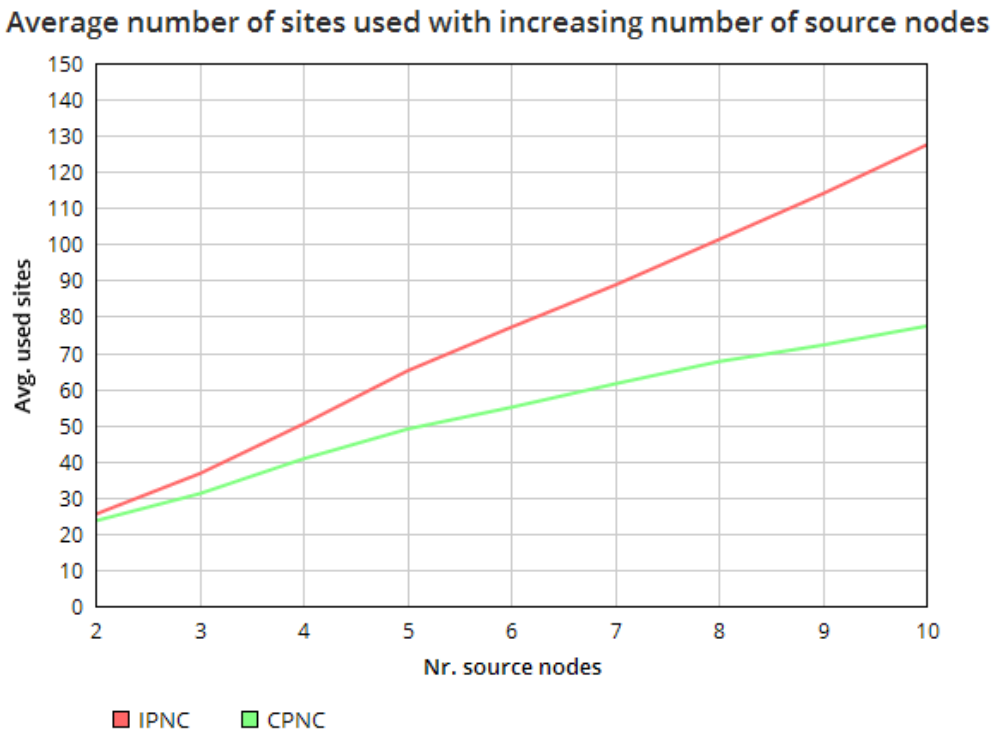


Figure 5.4: Bellman-Ford: The average sites used for IPNC and CPNC

Goal: The following shows the number of landing sites saved using the modified Bellman-Ford algorithm. The number of landing sites saved in proportion to the number of source nodes used is shown in the figure below.

Figure 5.5 shows the average number of landing sites saved as the source nodes increase. This line is also represented by the area between the two lines of the previous figure. The number of saved landing sites does not increase linearly when source nodes are added, therefore, more source nodes result in better effectiveness of the algorithm. More source nodes will use more landing sites, which then increases the chance that landing sites are shared. This figure also shows that the algorithm will be somewhat unproductive when only two source nodes are used, saving an average of only 1.85 landing sites.

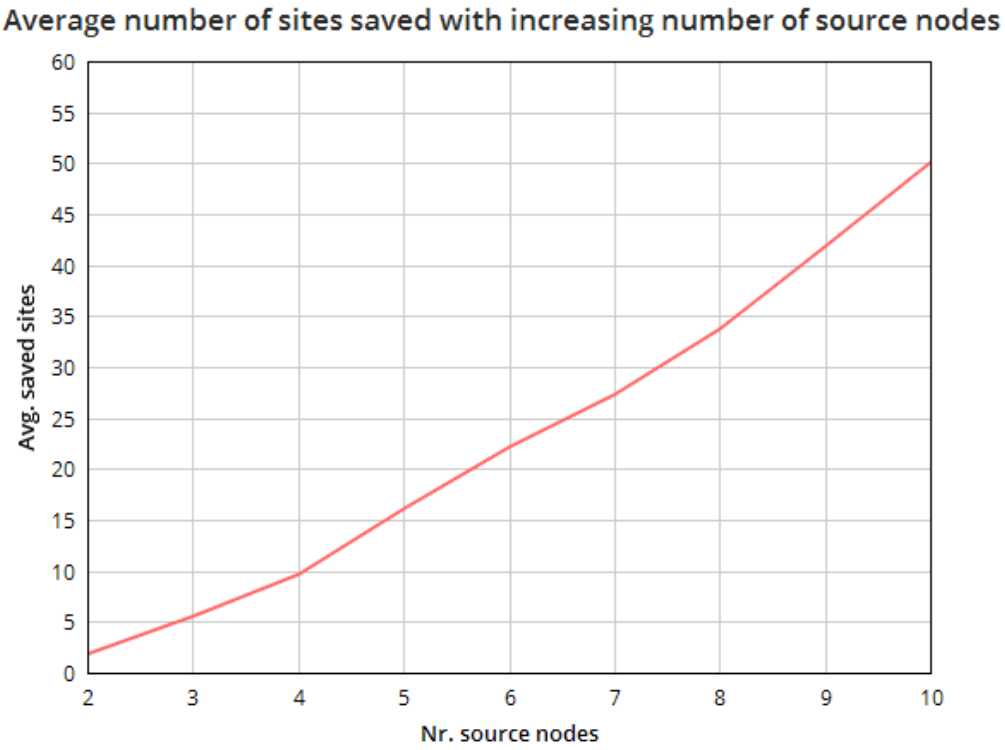


Figure 5.5: Bellman-Ford: The average sites saved with this implementation

5.2.2 Spread dependent

Goal: The results of modified Bellman-Ford's algorithm are also dependent on the spread of the source nodes. This section will show how the number of landing sites saved changes, as the source nodes are confined to different sector sizes.

Table 5.2 shows source nodes that were placed within 90°, 180° and then 360° sectors. Six source nodes were used and evenly distributed within the various sectors. When determining the effect of the spread, it is important to remove the impact that distances have on the result. The impact of distance was eliminated by ensuring the IPNC for all three sectors stayed the same. The table shows how the number of landing sites saved decreases with wider sectors, regardless of the distance the source nodes are from the base station. The table shows an enormous increase in the number of landing sites saved as the sector size decreases. It is clear the number of nodes saved stays dependent on the spread of the source nodes, irrespective of the distance to the base station.

Table 5.2: Bellman-Ford: Spread dependent

	90° sector		180° sector		360° sector	
IPNC	CPNC	Saved	CPNC	Saved	CPNC	Saved
100	50	50	64	36	85	15
80	32	48	56	24	71	9
60	30	30	46	14	56	4
40	14	26	27	13	36	4
20	8	12	12	8	15	5

5.2.3 Running times

Goal: To evaluate the feasibility and practicality of the modified Bellman-Ford algorithm for this system, the running times for each number of source nodes is evaluated.

Figure 5.6 shows the average processing time in minutes for each number of source nodes, using the modified Bellman-Ford algorithm. The processing time for this al-

gorithm increases dramatically with each source node that is added. The processing time for this system does not play a big role, except in this case where six source nodes have an average processing time of just over an hour. This indicates that the modified Bellman-Ford algorithm is slower to process, since the ten source nodes have an average processing time of 237 minutes. The figure shows how the time increases exponentially, as was expected with a theoretical running time of O($|E|$,$|V|$), where E is the number of edges and V, the number of vertices [27].

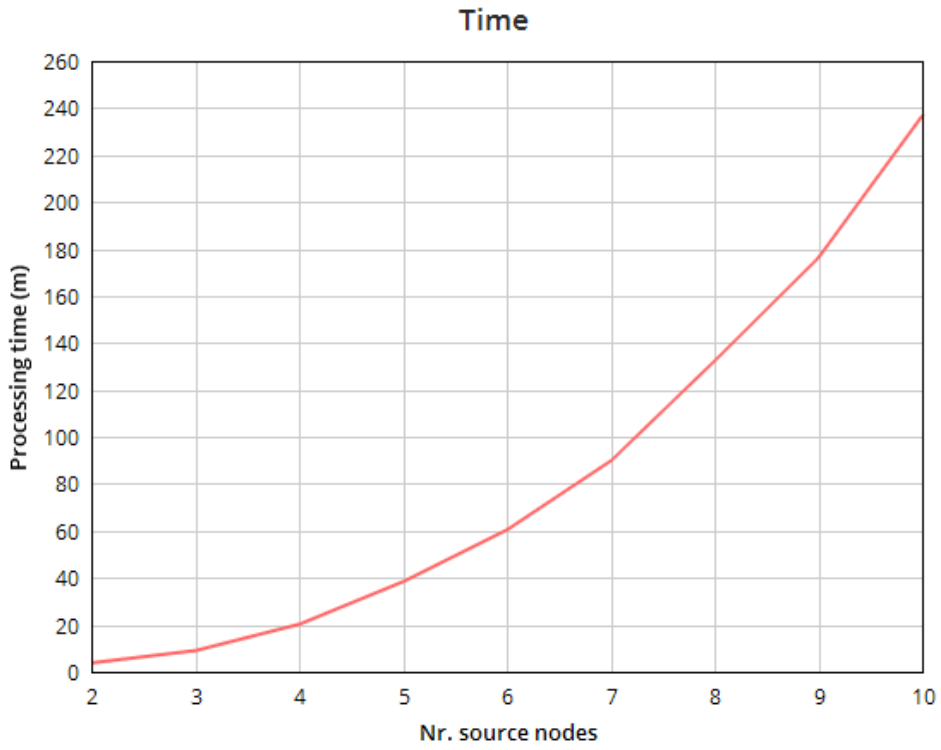


Figure 5.6: Bellman-Ford: The average algorithm run time

5.3 Comparing the modified Dijkstra and -Bellman-Ford

This section is used to demonstrate the results of both modified algorithms against each other. Since the source nodes were placed on the same landing sites, the number of landing sites used and saved can be compared. The modified Dijkstra and Bellman-Ford algorithms are both affected by the increase in the number and the spread of

the source nodes. The running times will be plotted together to show the enormous difference between the running times of both algorithms.

5.3.1 Increasing source nodes

Goal: To compare the effectiveness of the modified algorithms, the number of landing sites saved is compared. The algorithms saving more landing sites should be the most effective.

Figure 5.7 shows the average number of landing sites saved when using the modified Dijkstra and Bellman-Ford algorithm. The number of landing sites saved is calculated by subtracting the CPNC from the IPNC. The figure illustrates that the number of landing sites saved is similar with the modified Dijkstra's algorithm having a slight advantage. Although the number of landing sites saved is similar, there is a difference in the number used. Since the simulations for the modified algorithms are done on the same landing sites, the IPNC values should be similar. The IPNC value for two source nodes with the modified Bellman-Ford averages 1.3 landing sites less than using the modified Dijkstra's algorithm. This value increases with the number of source nodes, where ten source nodes will indicate the modified Bellman-Ford averages 7.15 landing sites less than the modified Dijkstra's algorithm. Since the IPNC values are not as similar as expected, and has an impact on the number of saved sites, the effectiveness of the modified algorithm cannot be compared by using the number of landing sites saved.

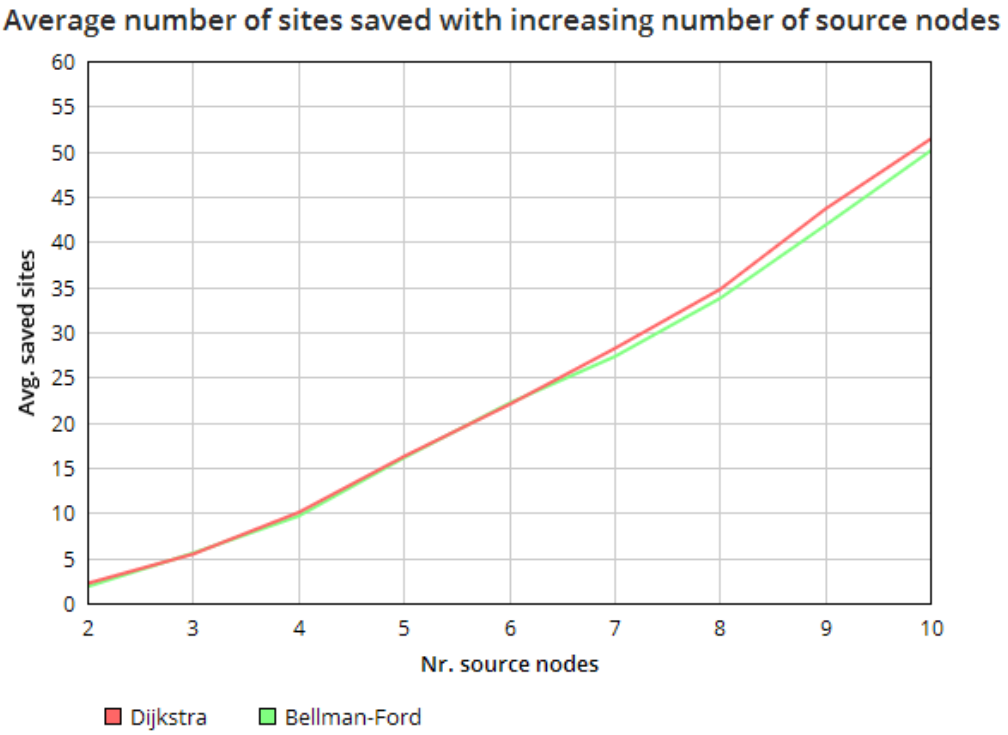


Figure 5.7: The average number of sites saved with the modified algorithms

Goal: The following is used to display the difference between the modified Dijkstra's algorithm and modified Bellman-Ford's algorithm regarding landing sites used. The number of landing sites used is a possible way to compare the effectiveness of the modified algorithms and will be displayed as the number of source nodes increase.

The CPNC shows the average number of landing sites used for the modified Dijkstra and Bellman-Ford algorithms, where the algorithms were implemented with the source nodes at the same positions. Figure 5.8 shows the CPNC for both the modified algorithms. The figure illustrates that the modified Bellman-Ford algorithm consistently uses fewer landing sites than the modified Dijkstra's algorithm, and the difference increases as source nodes are added. When two source nodes are used, the average difference in the CPNC is 0.95; where ten source nodes were used, the modified Bellman-Ford algorithm averages 5.85 landing sites less than the modified Dijkstra's algorithm. It is clear the modified Bellman-Ford algorithm is more effective than the modified Dijkstra's algorithm when the CPNC is used to compare effectiveness.

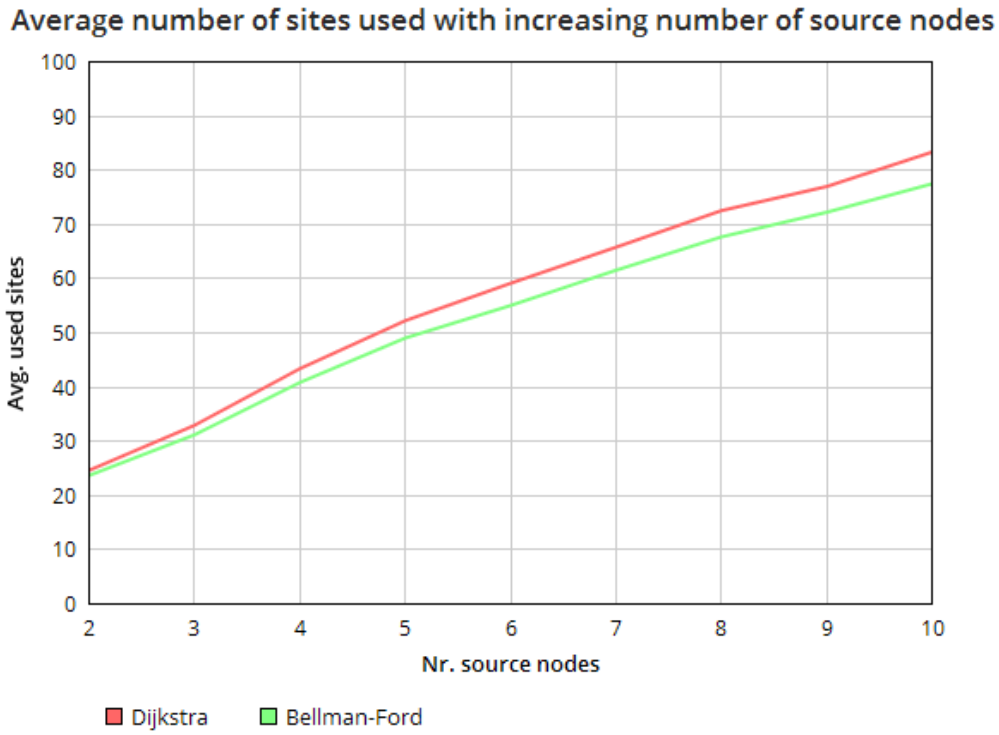


Figure 5.8: The average number of sites used with the modified Dijkstra and Bellman-Ford algorithms (CPNC)

5.3.2 Spread dependent

Goal: This section is used to compare the spread of source nodes for the modified algorithms with each other. The algorithm less dependent on the spread of the source nodes will share more paths.

Figure 5.9 illustrates the average landing sites saved for different sector sizes, which is determined for different IPNC values, using the data from the tables in Appendix B. The bar graph shows the modified Bellman-Ford algorithm saves more landing sites for each sector size, indicating it is less dependent on the spread of source nodes than the modified Dijkstra's algorithm. This is due to negative edge weights assigned to certain landing sites already used, which forces paths to share landing sites.

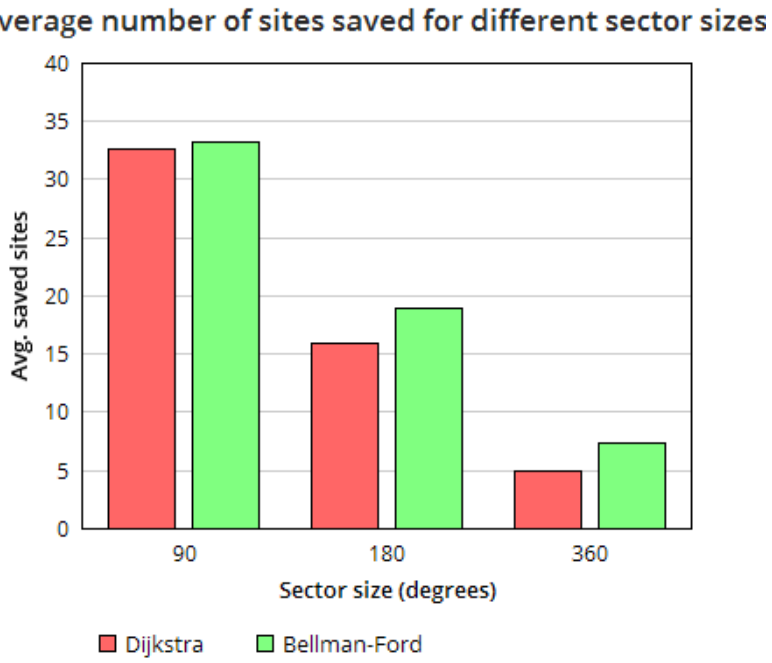


Figure 5.9: The average number of sites saved for the modified Dijkstra and Bellman-Ford algorithms with different sector sizes

5.3.3 Running times

Goal: This section will show the difference in running time for the modified algorithms. The processing times will be compared by plotting the two algorithms on the same scale.

The running times of the modified algorithms are compared using figure 5.10. With the running time of the modified Dijkstra's algorithm plotted in terms of minutes, it cannot be seen what the processing time is for each number of source nodes. The average running time for ten source nodes using the modified Dijkstra's algorithm is less than three minutes, which is why it is not clearly visible in the figure. The figure illustrates the enormous difference between the two modified algorithms regarding running time.

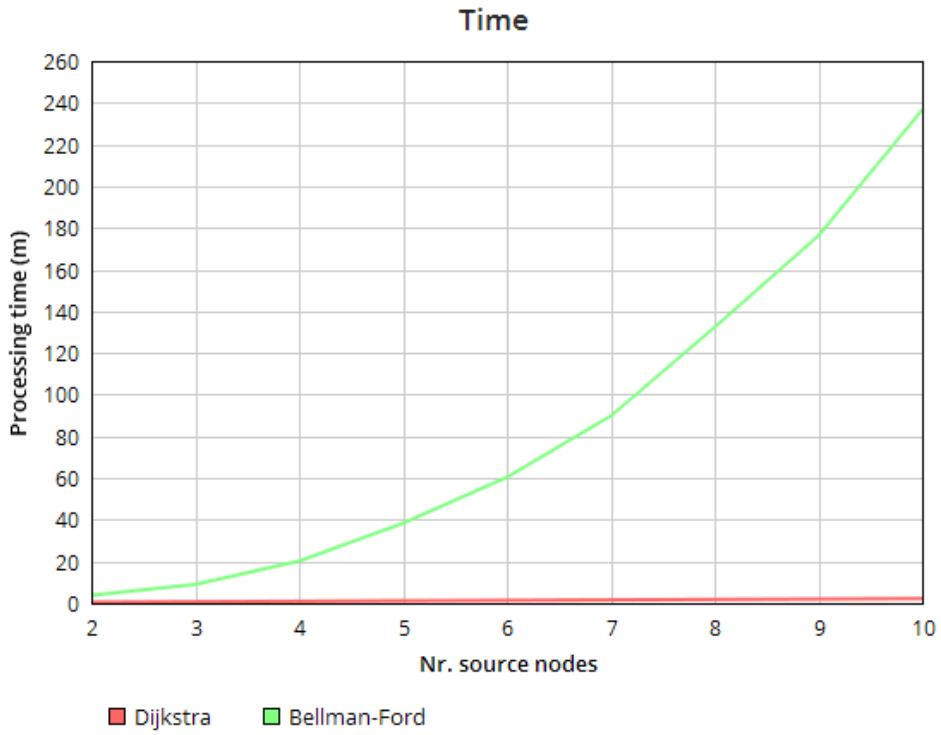


Figure 5.10: The average run time for the modified Dijkstra and Bellman-Ford algorithms

5.4 Summary

The expectation was that the IPNC values for the modified algorithms would be similar because the simulations were made with the source nodes on the same landing sites. This was not the case, as the IPNC of the modified Bellman-Ford algorithm for ten source nodes uses 5.31% less landing sites, on average, than that of the modified Dijkstra's algorithm. Where initial thoughts suggested that the number of saved landing sites would be the only measure of effectiveness, the number of landing sites used with CPNC seems to carry more weight. The CPNC directly indicates the average landing sites both modified algorithms used for each number of source nodes. The modified Bellman-Ford's CPNC values are between 4.31% and 5.53% smaller for the number of source nodes ranging from two to ten, indicating the modified Bellman-Ford algorithm has the advantage regarding effectiveness.

Both algorithms are dependent on the spread of the source nodes. The tables showed that for six source nodes, the modified Dijkstra's algorithm used just under 150% more landing sites, on average, when the source nodes were placed in a 360° sector as to when they were placed within a 90° sector. The modified Bellman-Ford algorithm saw an average increase of over 100% landing sites used when the sector size was changed from 90° to 360°. The comparison showed the modified Bellman-Ford algorithm is less spread dependent, therefore, sharing more paths. This property is one of the reasons the modified Bellman-Ford algorithm is more effective than the modified Dijkstra's algorithm.

The running time graph of the modified Dijkstra's algorithm is relatively linear, showing that adding a source node will add to the workload but not complicate the calculations. This cannot be said for the modified Bellman-Ford algorithm, as adding a source node to it adds to the workload and increases the complexity of the calculations. The increase in complexity causes the running time not to increase linearly, but O($|E|$,$|V|$). At first the running time for these algorithms was not a crucial property in this system. However, time does play a role when the algorithm uses more than an hour to process. This is the case with the modified Bellman-Ford algorithm, where the average running time is more than an hour when six or more source nodes are used. The average running time for the modified Bellman-Ford algorithm when using two source nodes is 96 seconds longer than the average time for ten source nodes using the modified Dijkstra's algorithm. This is one big drawback of the modified Bellman-Ford algorithm, as the running time for a larger number of source nodes is not worth the number of landing sites saved, compared to the modified Dijkstra's algorithm.

The results show that the derived modified algorithms proved to be a heuristic technique, a practical solution, which is not necessarily optimal but proves to be good enough. In the concluding chapter these results are taken in consideration to determine a practical solution.

Chapter 6

Conclusion and recommendations

This chapter includes an overview of all the work done and the process that was followed in compiling this dissertation. This is followed by concluding remarks regarding both modified algorithms. A recommendation on when to use which modified algorithm is given, and is followed by some possible future work on this topic.

6.1 Overview of work

The aim of this study was to select, modify and compare different shortest path algorithms to determine the placement of UAV repeaters to connect source nodes monitoring rhinos with the base station. To form redundant paths from the source nodes to the base station, some k-path shortest path algorithms were investigated. Suurballe and Tarjan's algorithm determine a pair of edge-disjoint paths using a transformed graph, while Yen's algorithm can be used to determine k-shortest paths. To perform modifications on a transformed graph is a complicated task, which is why Suurballe and Tarjan's algorithm was not used. Since the constraints require the primary and secondary paths to be vertex-disjoint, Yen's algorithm is also not an options, since the

k path shares nodes with the k-1 path. Although these algorithms were not used in this study, the algorithms used had a similar approach. The algorithms used in this study were implemented in the same way by determining the first path using a single shortest path algorithm, and to add a second path related to the first path within the specified requirements. This led to the research of four single shortest path algorithms (Dijkstra, Bellman-Ford, Floyd-Warshall and A-star). The Dijkstra and Bellman-Ford algorithms were implemented for this study since the Floyd-Warshall algorithm is an all-pairs shortest path problem and the A-star is a single-pair shortest path problem. The all-pairs shortest path problem is where the shortest path from all nodes to all nodes is determined, which is more than what is required to solve the problem of this study. The single-pair shortest path problem is where the path is found between a single source node and a destination node, and cannot consider other source nodes.

The Dijkstra and Bellman-Ford algorithms were modified to adhere to the predefined requirements of the intended use case. The algorithms were used to form two paths from each source node to the base station. The first modification made to Dijkstra's algorithm, was a weight benefit that was added for using nodes already in use. This forced the source nodes to share paths and reduce the number of occupied landing sites.

After the first modification, the paths were not all formed in a straight line between the source node and the base station, which caused the primary and secondary paths to separate from each other on certain instances. The separated primary and secondary paths resulted in some landing sites not having a node on another path to communicate with, which is against the stated requirements of the system. The next modification was to prevent the primary and secondary path from separating. It was performed by adding a weight/cost benefit to the nodes in the secondary path when within the communication range of the primary path nodes.

The same modification of Dijkstra's algorithm to share landing sites was made on the Bellman-Ford algorithm. The modification on Bellman-Ford's algorithm, to prevent separated primary and secondary paths, was approached differently. Instead of

adding a weight benefit to the secondary path nodes, which is within the communication range of the primary path, a detrimental weight is given to secondary path nodes that are not within the communication range of the primary path nodes.

For the Bellman-Ford algorithm, there is the additional modification of assigning certain nodes with a negative edge weight; these weights can be assigned to landing sites that have an energy advantage over other sites. For this application, negative edge weights were assigned to certain nodes that were already in use. Two nodes with a negative edge weight cannot be within communication range of each other, as this would result in a negative cycle. When a negative cycle exists in a network, no shortest path is available.

The modified algorithms were simulated to test the different properties and compare them to each other. The simulation area was 51.2km by 51.2km and only had one base station. The results were obtained by running multiple simulations with a different number of source nodes. The averages of the multiple simulations for each number of source nodes were used. The number of source nodes varied from 2 to 10 and were placed randomly over the simulation area. This is also the validation using the Monte Carlo method of random sampling. For each simulation, the number of landing sites used and its processing time were saved. The number of landing sites used for each source node connecting to the base station were added together (IPNC). This value was used as a simple representation of the number of landing sites that would be used without the modified algorithm and is of use when comparing the algorithms.

After the simulations, the results were formatted to the desired output for the communication protocol. The output format displays the two paths from each node to the base station starting at different positions. Two checks were performed to verify that the output data complied with the requirements.

6.2 Conclusion

The modified Dijkstra and Bellman-Ford algorithms were simulated to test the different properties of the modified algorithms. For the results to be comparable, the source nodes were placed on the same landing sites for both algorithms. To test the number of landing sites saved for a scenario, the CPNC value was subtracted from the IPNC value. Results of both algorithms show that the number of landing sites saved increases exponentially as the number of source nodes increase; this is because a rise in the number of source nodes will increase the chances for paths to share landing sites.

There is only a small difference between the number of landing sites saved using the modified Dijkstra's- and -Bellman-Ford's algorithm. The difference in the average number of landing sites saved for 2 to 8 source nodes is not more than 1 landing site in both algorithms. For 9 and 10 source nodes, the difference in landing sites saved is less than 2 landing sites, on average, in the modified Bellman-Ford's favour.

The initial thought was that the average number of landing sites saved would measure the effectiveness of the algorithms. Although the number of landing sites saved is very similar, there is a difference in the effectiveness of the algorithms. The number of landing sites saved is also dependent on the IPNC value. The IPNC values of the modified Bellman-Ford's algorithm is less than the modified Dijkstra's algorithm and the differences increases as the source nodes increase. Using 10 source nodes, the IPNC for the modified Bellman-Ford's algorithm is 7.2 landing sites less on average. With the IPNC already hinting the modified Bellman-Ford's superiority, the effectiveness can be measured using the CPNC value since the source nodes are placed on the same landing sites. For each number of source nodes, the modified Bellman-Ford algorithm averages less landing sites than the modified Dijkstra's algorithm. When using 10 source nodes, the CPNC of the modified Bellman-Ford algorithm averages 5.85 less landing sites, which puts this algorithm at a slight advantage.

The number of landing sites saved for both algorithms are dependent on the spread of the source nodes. To show this, six source nodes were evenly spread into 90°, 180°

and 360° sectors. To limit the variables, the distance from the base station was kept the same by keeping the IPNC value constant. The results indicate that the average number of landing sites used for both algorithms more than doubled when the source nodes were spread from 90° to 360°. It was found the average landing sites used increased by 86.3% and 59.8%, from 90° to 180° and 29.6% and 27.9% from 180° to 360° for the modified Dijkstra and Bellman-Ford's algorithms respectively. It is clear that a small difference in the spread of the source nodes can make an enormous difference in the number of landing sites saved. The comparison showed that the modified Bellman-Ford algorithm is less dependent on the spread of source nodes than the modified Dijkstra's algorithm. This is caused by the negative edge weights assigned to certain existing landing sites for the modified Bellman-Ford algorithm, which forces the algorithm to share more paths.

One of the biggest differences between the modified Dijkstra's- and -Bellman-Ford's algorithm is the running time. The computational complexity of the modified Dijkstra's algorithm allows the running time to increase linearly as source nodes are added. With the modified Bellman-Ford's algorithm, the computational complexity increases with each added source node, causing the running time to increase rapidly. The running times are increasing at different rates as source nodes are added. The average running time for the modified Bellman-Ford's algorithm using 2 source nodes is 3 minutes and 48 seconds. The modified Dijkstra's algorithm has an average running time of 2 minutes and 12 seconds when using 10 source nodes. There is an enormous difference in the elapsed time between the modified Dijkstra and Bellman-Ford's algorithm and even more so as the source nodes increase.

Ultimately, comparing these algorithms to each other is a daunting task. The first attempt was to compare the number of landing sites saved by each of the algorithms. The results showed that the modified Dijkstra and Bellman-Ford's algorithm saved a similar number of landing sites. The number of landing sites saved is not a good measure of effectiveness since it is dependent on the IPNC value. The number of landing sites used can thus be used as a measure of performance, since the source nodes were placed on the same landing sites. The modified Bellman-Ford algorithm averaged 5%

less landing sites than the modified Dijkstra's algorithm for each number of source nodes. It was found both algorithms are dependent on the spread of the source nodes. Time performance results show that the modified Dijkstra's algorithm averages a run time of more than a hundred times faster than the modified Bellman-Ford's algorithm with 10 source nodes, and the difference will increase as source nodes are added. It is important to remember that Bellman-Ford has the advantage of assigning negative edge weights to landing sites with an energy advantage.

This study indicates that it was possible to select and modify shortest path algorithms to create a redundant communication path between the UAV monitoring a rhino and a base station. The modified algorithms used less landing sites to connect all source nodes with the base station than the original algorithms.

6.3 Recommendations

Recommending one of the modified algorithms depends on the properties of the network and the application. If the application has existing infrastructure or repeater poles close to the base station that has an energy advantage, the Bellman-Ford algorithm has a great advantage over Dijkstra's algorithm. The modified Bellman-Ford's algorithm will ensure that the nodes with an energy advantage are regularly utilised and will use less landing sites that require a UAV. The modified Dijkstra's algorithm will treat all the nodes equally and will not make use of the additional energy available at certain nodes.

Probably the most important aspect is whether or not there are any time constraints. If there are no time constraints, the modified Bellman-Ford algorithm would be the better option. The performance was measured using the CPNC value. By placing the source nodes on the same landing sites, the modified Bellman-Ford's algorithm continuously occupied less landing sites than the modified Dijkstra's algorithm. The modified Dijkstra's algorithm has excellent time performances with a decent performance regarding the number of landing sites used, but if there is no time constraint

the modified Bellman-Ford is the algorithm to use.

Time constraints will always be present, which brings another aspect into play, the number of source nodes that will be used. For situations where 6 or fewer source nodes are used, the modified Bellman-Ford algorithm is still a viable choice as the average running time would be an hour, which is an acceptable timeline. In cases where more than 6 source nodes are used, the modified Dijkstra's algorithm is the best option. The modified Dijkstra's algorithm will deliver a network in less than 3 minutes, where the running time of the modified Bellman-Ford's algorithm does not surpass its ability to use less landing sites than the modified Dijkstra's algorithm.

The biggest trade-off for picking the modified Dijkstra or Bellman-Ford's algorithm is time versus the number of nodes. If enough time is available, the modified Bellman-Ford's algorithm has the edge over the modified Dijkstra's algorithm regarding using less landing sites. The modified Bellman-Ford's algorithm using nodes with energy advantages are meaningless if there are strict time constraints.

6.4 Future work

Some of the figures show a resemblance to Steiner minimum trees. Steiner minimum trees are a strong candidate to consider for comparison with the current algorithms. Steiner minimum trees find the shortest path to connect several given points. With this shortest path, points can be connected to each other using intermediate junctions called Steiner points [40].

In this study, one base station was used. Future investigations can incorporate two or more base stations. Base stations do not have energy constraints and can be placed on existing infrastructure. If two or more base stations are placed on proper geographical positions, it could reduce the number of UAVs required to relay the data to the base stations.

Future work may also include a different arrangement of the landing positions. Landing positions can be placed closer to each other; the communication ranges will overlap and ensure no uncovered areas exist. The landing positions can also be placed further from each other for elevated geographic positions that have better communication ranges. This can be applicable to areas with ridges and towers.

Finally, the study can be expanded by adapting the landing site grid. Landings sites can be eliminated for areas inaccessible to the rhino including dams, roads, camping areas and areas near power lines.

Bibliography

[1] D. Biggs, F. Courchamp, R. Martin, and H. P. Possingham, "Legal trade of africas rhino horns," *Science*, vol. 339, no. 6123, pp. 1038–1039, March 2013.

[2] N. Leader-Williams *et al.*, *World trade in rhino horn: a review*. United Kingdom: Traffic, 1992.

[3] K. Nowell, "Assessment of rhino horn as a traditional medicine," *TRAFFIC report to CITES Secretariat pursuant to contract CITES Project No. S-389, Geneva, Switzerland*, 2012.

[4] R. Martin, "A legal trade in rhino horn: Hobsons choice," *Self published. Retrieved from www. rhinoresourcecenter. com pdf les*, pp. 1–36, 2011.

[5] C. So-In, C. Phaudphut, S. Tesana, N. Weeramongkonlert, K. Wijitsopon, U. KoKaew, B. Waikham, and S. Saiyod, "Mobile animal tracking systems using light sensor for efficient power and cost saving motion detection," in *Communication Systems, Networks & Digital Signal Processing (CSNDSP), 2012 8th International Symposium on*. IEEE, 2012, pp. 1–6.

[6] "Remote Piloted Aircraft Systems," http://www.caa.co.za/Pages/RPAS/Remotely%20Piloted%20Aircraft%20Systems.aspx, 2016, [Online; accessed 8-October-2016].

[7] B. A. Forouzan, *Data Communications and Networking 5th edition*. New York: McGraw-Hill, 2013.

[8] A. A. Okandeji, M. R. Khandaker, and K.-K. Wong, "Wireless information and power transfer in full-duplex communication systems," *arXiv preprint arXiv:1603.02182*, 2016.

[9] L. E. Frenzel, *Principles of electronic communication systems 3rd edition*. Boston: McGraw-Hill, 2008.

[10] "The definition of redundant," http://dictionary.reference.com/browse/redundant?s=t, 2015, [Online; accessed 1-November-2015].

[11] "The definition of source," http://dictionary.reference.com/browse/source?s=t, 2015, [Online; accessed 1-November-2015].

[12] O. Oguike, "Recursive shortest route algorithm using abstract data type, graph," *African Journal of Computing & ICT*, vol. 6, no. 2, 2013.

[13] S. K. S. Gupta and P. K. Srimani, "Adaptive core selection and migration method for multicast routing in mobile ad hoc networks," *IEEE Transactions on Parallel and Distributed Systems*, vol. 14, no. 1, pp. 27–38, 2003.

[14] F. Iqbal and F. A. Kuipers, "Disjoint paths in networks," *Wiley Encyclopedia of Electrical and Electronics Engineering*.

[15] L. Guo and H. Shen, "On the complexity of the edge-disjoint min–min problem in planar digraphs," *Theoretical computer science*, vol. 432, pp. 58–63, 2012.

[16] D. Mehta, B. O'Sullivan, C. Ozturk, and L. Quesada, "Computing distance-bounded node-disjoint paths for all pairs of nodesan application to optical core network design," in *Reliable Networks Design and Modeling (RNDM), 2015 7th International Workshop on*. IEEE, 2015, pp. 71–77.

[17] J. W. Suurballe and R. E. Tarjan, "A quick method for finding shortest pairs of disjoint paths," *Networks*, vol. 14, no. 2, pp. 325–336, 1984.

[18] M. German, A. Castro, X. Masip-Bruin, M. Yannuzzi, R. Martinez, R. Casellas, and R. Munoz, "On the challenges of finding two link-disjoint lightpaths of minimum

total weight across an optical network," in *14th European Conference on Networks and Optical Communications (NOC) and the 4th Conference on Optical Cabling and Infrastructure, Valladolid, Spain*, 2009.

[19] M. Pascoal, "Implementations and empirical comparison for k shortest loopless path algorithms," *The Ninth DIMACS Implementation Challenge: The Shortest Path Problem*, 2006.

[20] A. W. Brander and M. C. Sinclair, "A comparative study of k-shortest path algorithms," in *Performance Engineering of Computer and Telecommunications Systems*. Springer, 1996, pp. 370–379.

[21] J. Y. Yen, "Finding the k shortest loopless paths in a network," *management Science*, vol. 17, no. 11, pp. 712–716, 1971.

[22] B. V. Cherkassky, A. V. Goldberg, and T. Radzik, "Shortest paths algorithms: Theory and experimental evaluation," *Mathematical programming*, vol. 73, no. 2, pp. 129–174, 1996.

[23] T. H. Cormen, *Introduction to algorithms 3rd edition*. Massachusetts: MIT press, 2009.

[24] S. M. Kumari and N. Geethanjali, "A survey on shortest path routing algorithms for public transport travel," *Global Journal of Computer Science and Technology*, vol. 9, no. 5, pp. 73–76, 2010.

[25] S. E. Dreyfus, "An appraisal of some shortest-path algorithms," *Operations research*, vol. 17, no. 3, pp. 395–412, 1969.

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

[27] P. F. Felzenszwalb and R. Zabih, "Dynamic programming and graph algorithms in computer vision," *Pattern Analysis and Machine Intelligence, IEEE Transactions on*, vol. 33, no. 4, pp. 721–740, 2011.

[28] A. Schrijver, "On the history of the shortest path problem," *Documenta Mathematica*, pp. 155–167, 2012.

[29] J. M. Cargal, "Shortest paths," *Discrete Mathematics for Neophytes*, vol. 9, 2003.

[30] R. Bellman, "On a routing problem," *Quarterly of applied mathematics*, pp. 87–90, 1958.

[31] J. Ma, K.-p. Li, and L.-y. Zhang, "A parallel floyd-warshall algorithm based on tbb," in *Information Management and Engineering (ICIME), 2010 The 2nd IEEE International Conference on*. IEEE, 2010, pp. 429–433.

[32] K. Gutenschwager, S. Völker, A. Radtke, and G. Zeller, "The shortest path: Comparison of different approaches and implementations for the automatic routing of vehicles," in *Proceedings of the Winter Simulation Conference*. Winter Simulation Conference, 2012, p. 291.

[33] M. A. Djojo and K. Karyono, "Computational load analysis of dijkstra, a*, and floyd-warshall algorithms in mesh network," in *Robotics, Biomimetics, and Intelligent Computational Systems (ROBIONETICS), 2013 IEEE International Conference on*. IEEE, 2013, pp. 104–108.

[34] S. Hougardy, "The floyd-warshall algorithm on graphs with negative cycles," *Information Processing Letters*, vol. 110, no. 8, pp. 279–281, 2010.

[35] H. Wang, J. Zhou, G. Zheng, and Y. Liang, "Has: Hierarchical a-star algorithm for big map navigation in special areas," in *Digital Home (ICDH), 2014 5th International Conference on*. IEEE, 2014, pp. 222–225.

[36] W.-J. Seo, S.-H. Ok, J.-H. Ahn, S. Kang, and B. Moon, "An efficient hardware architecture of the a-star algorithm for the shortest path search engine," in *INC, IMS and IDC, 2009. NCM'09. Fifth International Joint Conference on*. IEEE, 2009, pp. 1499–1502.

[37] I. S. AlShawi, L. Yan, W. Pan, and B. Luo, "Lifetime enhancement in wireless sensor networks using fuzzy approach and a-star algorithm," *Sensors Journal, IEEE*, vol. 12, no. 10, pp. 3010–3018, 2012.

[38] J. Cargal, "Discrete mathematics for neophytes: Number theory," *Probability, Algorithms, and Other Stuff*, 1988.

[39] C. J. van der Spuy, "The design of a communication protocol for a reconfigurable wireless animal tracking mesh network," M. Eng. thesis, North-West University, Potchefstroom, South Africa, Nov. 2015.

[40] G. Robins and A. Zelikovsky, "Minimum steiner tree construction," *The Handbook of Algorithms for VLSI Phys. Design Automation*, pp. 487–508, 2009.

Appendix A

Graphical user interface

This appendix will demonstrate the GUI used to perform the simulations of this study. The figures show the base station that can be moved with the spin box, or by entering a position number in the editable area. The buttons required for performing the modified Dijkstra and modified Bellman-Ford are also shown, with an additional button to generate the output for the communication protocol. The algorithm used, the number of sites used and the processing time is shown below the buttons.

A.1 Initial simulation interface

Figure A.1 is the initial display of the GUI. Before starting simulations, the base station is placed on the required landing site. The source nodes are selected by clicking on the respective landing sites. To start the simulation, click on the algorithm that will be used. The simulation is finished when the paths are shown, as in figure A.2 and A.3. To obtain the output for the communication protocol, the output button is pressed.

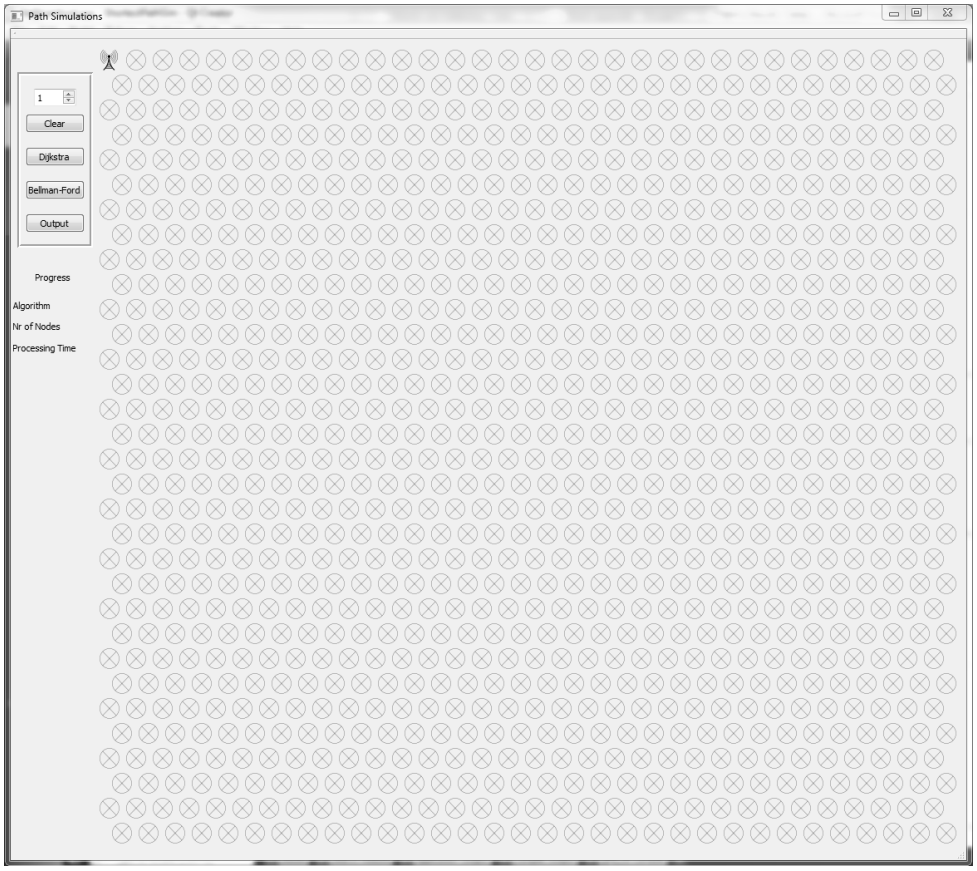


Figure A.1: Simulation

A.2 Simulation with Dijkstra's algorithm

Figure A.2 shows the landing sites used for all source nodes to communicate with the base station. The modified Dijkstra's algorithm is used to determine the paths that each source node uses. The scenario in this figure, with six source nodes, forms part of the results. It is visible from the figure that the modified Dijkstra's algorithm was used, 86 landing sites will be occupied and the running time is 107.5 seconds.

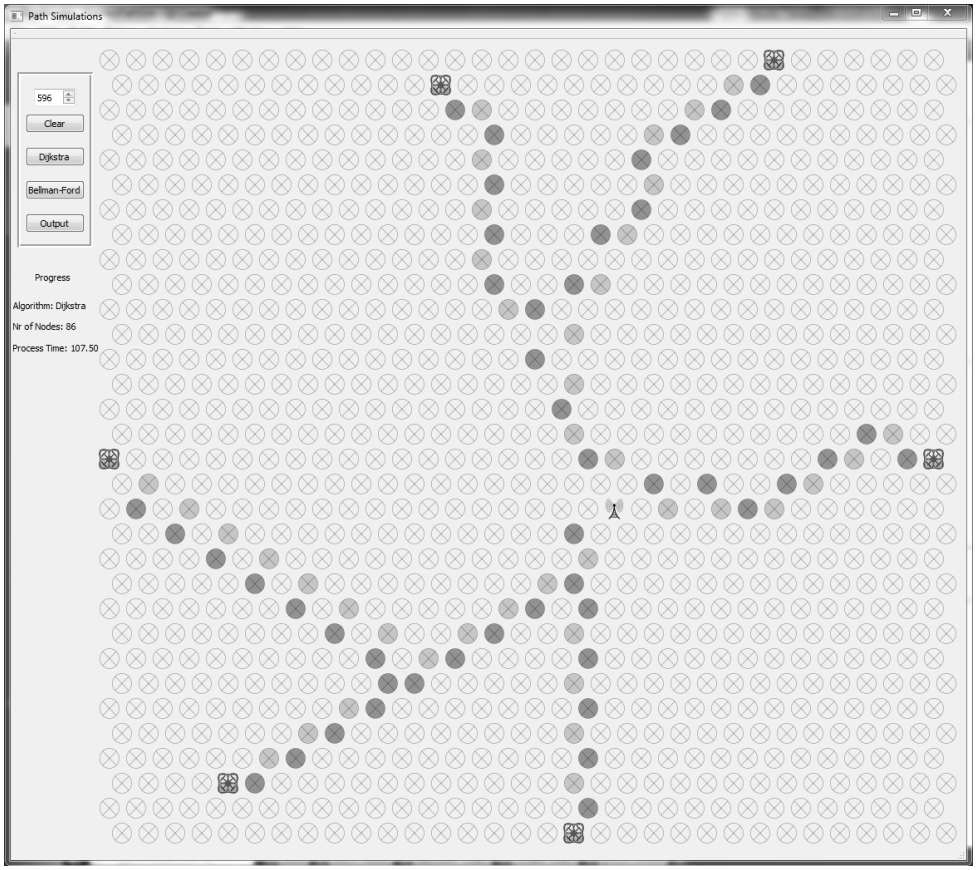


Figure A.2: Simulation Dijkstra

A.3 Simulation with Bellman-Ford's algorithm

The paths in figure A.3 are formed using the modified Bellman-Ford's algorithm. Six source nodes were used and this result was used in one of the scenarios for the results. The figure shows that 60 landing sites will be occupied for all the source nodes to have a redundant communication path to the base station. It is also visible that the processing time is 3156 seconds, which is more than 52 minutes.

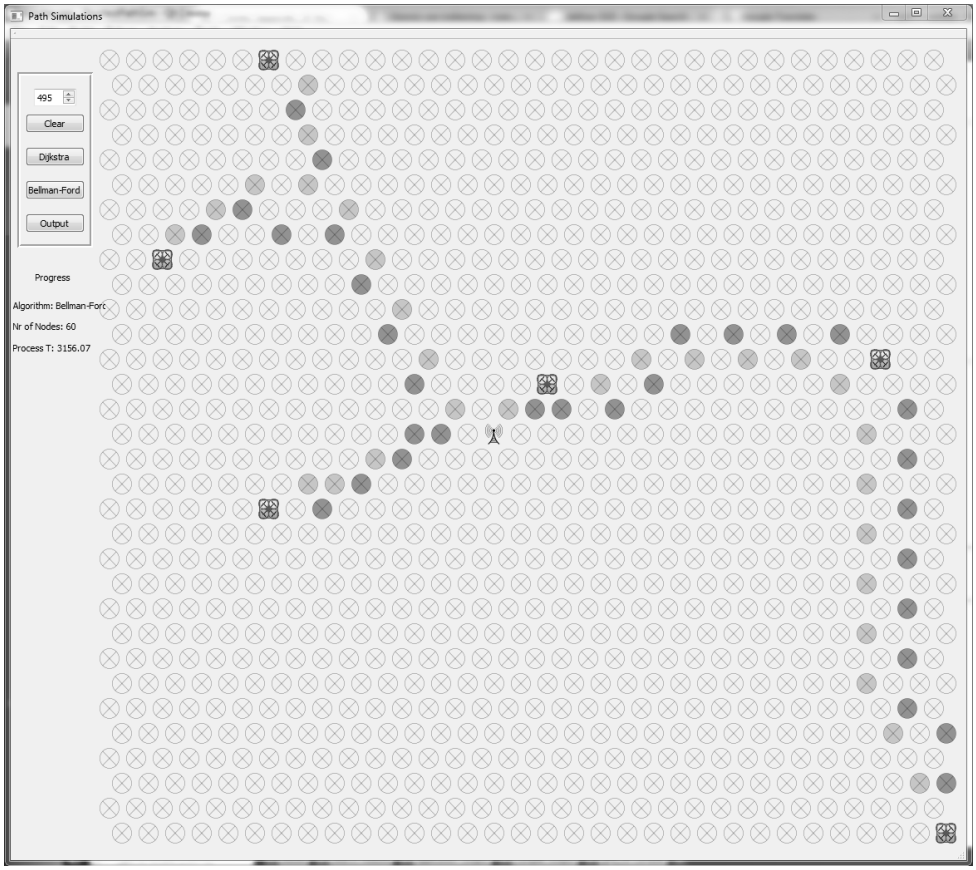


Figure A.3: Simulation Bellman-Ford

Appendix B

Result tables

This appendix contains the values of the results compared in Chapter 5. Each value is an average obtained from multiple simulation scenarios. At 20 scenarios, the average values converged, which can be used to represent the results of the data.

B.1 Results for Dijkstra

In table B.1, the first column represents the number of source nodes used. The second column is the average number of nodes used when the source nodes are connected to the base station individually and added together. The third column is when the source nodes are simulated together, and the final column is the values when column three is subtracted from column two. The columns from two to four are all average values obtained from multiple simulations with the different number of source nodes. The table shows that the average number of source nodes saved when using two source nodes is only 2.2. This indicates that using this implementation of Dijkstra for only two source nodes does not have a significant benefit.

Table B.1: Dijkstra: Averages for increasing number of source nodes

Nodes	IPNC	CPNC	Saved sites
2	26.7	24.5	2.2
3	38.25	32.8	5.45
4	53.3	43.25	10.05
5	68.35	52.1	16.25
6	81.0	59.0	22.0
7	93.85	65.65	28.2
8	107.15	72.4	34.75
9	120.5	76.85	43.65
10	134.6	83.2	51.4

B.2 Results for Bellman-Ford

To clarify the differences between the implementation of Dijkstra and Bellman-Ford's algorithms, table B.2 is used. The first column indicates the number of source nodes used; the second column is the number of landing sites used when these source nodes are connected to the base station individually, and then added together. The third column is when all source nodes are simulated with the implementation of Bellman-Ford's algorithm; the last column is the saved landing sites. These columns display the average landing sites used and saved for the different number of source nodes.

Table B.2: Bellman-Ford: Averages for increasing source nodes

Nodes	IPNC	CPNC	Saved sites
2	25.4	23.55	1.85
3	36.6	31.05	5.55
4	50.35	40.7	9.65
5	65.0	48.9	16.1
6	77.05	54.9	22.15
7	88.7	61.4	27.3
8	101.3	67.55	33.75
9	113.95	72.1	41.85
10	127.45	77.35	50.1