ATHABASCA UNIVERSITY

PATHFINDING STRATEGY FOR MULTIPLE NON-PLAYING CHARACTERS AND AGENTS IN A 2.5D GAME WORLD

BY

JASON MACGREGOR

A thesis essay submitted in partial fulfillment

Of the requirements for the degree of

MASTER OF SCIENCE in INFORMATION SYSTEMS

Athabasca, Alberta

February, 2009

ATHABASCA UNIVERSITY

The undersigned certify that they have read and recommend for acceptance the thesis project "PATHFINDING STRATEGY FOR MULTIPLE NON-PLAYING CHARACTERS AND AGENTS IN A 2.5D GAME WORLD" submitted by JASON MACGREGOR in partial fulfillment of the requirements for the degree of MASTER OF SCIENCE in INFORMATION SYSTEMS.

____

Steve Yeung, Ph.D.

Supervisor

____

Chair

____

Examiner

Date: __________________________

DEDICATION

I dedicate this essay to my wonderful wife Kristine. Without her unflagging support and understanding this essay would never have been completed.

ABSTRACT

This essay focuses on pathfinding in a 2.5D game world for Non-playing Characters (NPC) and agents. The primary focus is to investigate and determine the optimal pathfinding strategy based on several measures such as steps and time taken to reach the goal. For this investigation the most popular pathfinding algorithms and interaction schemes will be analyzed and tested using standard analysis techniques and test applications. The result of this analysis and testing is that it is shown that the A* algorithm is optimal using the Manhattan distance heuristic. Unfortunately, our testing did not reveal an optimal choice among the three interaction scheme that was tested which were cooperative, non-cooperative and competitive.

ACKNOWLEDGEMENTS

I would like to take this opportunity to thank the faculty at Athabasca University's School of Computing for preparing me for the challenge of producing this essay. A very special thanks goes out to Dr. Leung for accepting the task of supervising this project. His interest, suggestions and advice meant a lot to me. Thank you all.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

CHAPTER 1

INTRODUCTION

An essential part of most strategy games involves the movement of Non-Player Characters (NPC) and/or Agents in a seemingly intelligent manner within the game. In the computer gaming industry, this intelligently moving of objects has become more significant and complex as games have increased in complexity and can be found in many different styles of games from real time strategy to first person shooters.

The intelligent movement of objects, specifically NPC and Agents, is known as pathfinding and in the most general sense simply means finding the best route from an initial starting point to a destination. The term best in this context means finding the optimal route, which is path having the smallest cost over the entire set of paths [6], based on certain pre-defined criteria such as using minimal amount of system resources or finding the optimal path in an acceptable amount of time.

All pathfinding strategies, which are simply the decided upon pathfinding solution used within an application, must take into account all possible obstacles that can be encountered in the application environment and must have a tactic to deal with them. As the study of pathfinding has matured, many algorithms and strategies have been developed that address this question of finding the optimal path from an initial starting point to a destination while navigating around or accepting a penalty and going though obstacles. Some of these strategies are optimal only for certain types of environments while others work for all types of environments but may require resources beyond what can be allocated for the task. These issues resulted in the search for an all purpose strategy that could find the path from one location to the next in the optimal number of steps while not requiring resources that would make using it a determent to the application in general.

Statement of the Purpose

The main purpose of this paper is to investigate and design a workable pathfinding strategy in a 2.5D game world for multiple NPC and players' Agents that is optimal in terms of steps taken and resources required. In pursuit of this purpose I will consider fixed location obstacles and moving NPCs and Agents. The strategy that will be developed will work for both cooperative environments where NPCs and Agents must find non-colliding routes from one location to separate destinations with knowledge of the routes of others [22], as well as non-cooperative environments where NPC and Agents have no knowledge of each others plans [22]. This paper will also address Antagonistic Pathfinding, also known as Competitive Pathfinding, where the NPC or Agents try to maximize their performance measure which may minimize other NPC or Agents performance measure [14, 22].

Research Problem

The research problem that this paper is trying to address is: what is the optimal pathfinding strategy in regards to resources required, time needed to discover the path and the number of steps required for the NPC or Agent get to the destination in a 2.5D game world. This problem has many facets that play a part in developing an overall solution.

This discussion will focus primarily on the most popular algorithm for pathfinding, the A* (pronounced A Star) algorithm. In our discussion of the A* algorithm

I will discuss several other popular algorithms too be used in comparison to A* as a means to illuminate the history, features and reasons for the popularity of the A* algorithm. Additionally, I will discuss several variants of the A* algorithm that have been proposed to address specific perceived deficiencies or as replacements for the A* algorithm. Several other success factors will also be taken into consideration such as a discussion on search depths for the various algorithms and variants of A* with a focus on resource consumption.

In our study, it will also be assumed that the 2.5d environment we are using in our study is fully observable, which means the NPC or Agent is fully aware of all aspects that are relevant to its choice of action [14, 25]. This gives the NPC or Agent the complete State of the environment in which it resides, at each point in time [14].

In order to round out this topic, I will provide, in figures, tables and various examples, an in-depth discussion of the algorithms in question followed by a real world comparison of the algorithms in a prototype environment that will demonstrate the effectiveness of the discussed algorithms, comparing the best possibilities to each other using basic metrics as the comparison method.

Significance of Problem

The study of pathfinding and strategies is significant for several reasons. Firstly, the technology that goes into computer gaming is almost always at the cutting edge of what is possible with graphics, hardware, artificial intelligence, etc, and pathfinding is no different. Pathfinding algorithms can tax the CPU and memory resources of computers to the point that they are unable to address all requests for resources. An analysis of the various pathfinding algorithms and strategies can help to identify the areas where pathfinding algorithms may cause hardware and performance issues, but also gives the programmer insight into how to mitigate these issues or even overcome them.

Additionally, the study of pathfinding opens up other significant areas of interest in computing, specifically the field of Artificial Intelligence (AI). AI is the field of study that aims to encompass computational techniques for performing tasks that apparently require intelligence when performed by humans [25] and deals directly with the intelligence of machines [26]. AI is a very large and fragmented field that has many sub areas, topics and sub fields.

AI can also be viewed as the study and design of intelligent agents [26] such as those referred to as Non-Playing Characters (NPC) and agents. The study of pathfinding algorithms presents a basic look into the field of AI and how it can be used in real world applications such as the Roomba robotic, in home vacuum cleaner developed and sold by iRobot Corporation to the Mars Pathfinder operated by NASA. While both these examples go beyond this discussion, the basis for both examples and the technology behind how they operate is based on the same questions and techniques that will be discussed in this paper. Both of these examples need to be able to navigate their environment in the most efficient manner and pathfinding, in one way or another, makes this possible.

In order to understand pathfinding, think of an open field. It is easy to see how to move an object from one location to the next but what if there is a wall between the two locations. How does the object now move to the next location by the most efficient path? Now suppose that the object in question is an NPC or agent. How does a programmer embed in the NPC or agent the ability to maneuver inside a virtual game world in the most efficient and least memory intensive way.

Limitations

The area of study known as Pathfinding is a very large topic that has been addressed by many researchers over the years. As such, it is not possible to discuss all possible pathfinding strategies and algorithms that have been introduced by all this research. In the scope of this paper, it is only possible to discuss the most popular and widely used strategies and algorithms, primarily Dijkstra's algorithm, Best-First Search (BFS) and the A* algorithm.

Definition of Terms

In order to develop a base of knowledge for the topic addressed by this paper, this section will define several important terms that are used throughout the paper.

Firstly, what is a NPC? A NPC is a character in a game that is not controlled by a human. Its actions are dictated by a series of pre-defined rules that it follows during the course of a game. These rules may include any actions that are seen as being 'in character' for the NPC and are dependant on the type of game and the game designers whim. The NPC may interact with a Player Character (PC) and may have its actions triggered by this interaction.

Similar to a NPC, an Agent is an entity that is not controlled by a human. It observes its environment and interacts with it to satisfy a pre-programmed goal or set of goals. With agents, there may be an element of learning or knowledge retention involved that assists them in achieving their goals. There may be many different varieties of agents in a game and they can be simple to very complex depending on the requirements as seen by the game designer.

The next important term to understand is the term 2.5D. What do we mean by 2.5D? Tile based games tend to have one of two different perspectives, either a top-down 2D perspective or an isometric 2.5D perspective which allows the graphical representation of the game to represent three-dimensional objects in two-dimensions.

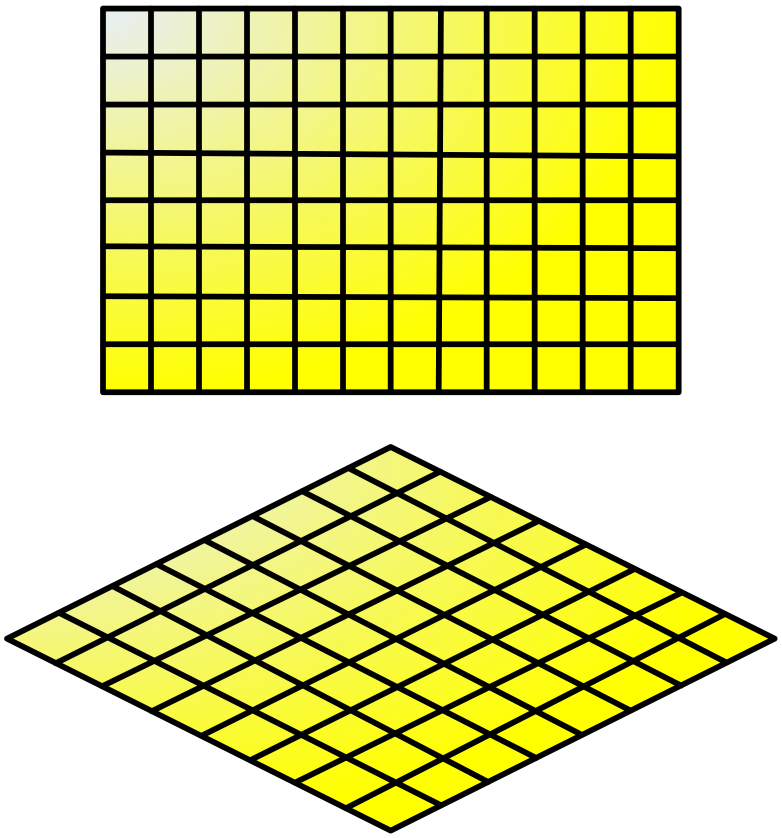


Figure 1: (a) 2D Tile Based Grid (b) 2.5D tile Based Isometric Grid

In a 2.5D isometric perspective all objects are to scale no matter where they are in the space. This differs from real perspective in which all objects reduce in size the farther they recede into the background. This is an important distinction and must be fully understood in order to completely understand what is meant by isometric prospective and 2.5D. Following is an example that illustrates this concept.

In the two figures below, figure (A) shows true perspective in which the rectangle that is perceived to be in the background is smaller than the rectangle perceived to be in the front. In figure (B) the rectangles are the same size even for the rectangle perceived to be in the background.

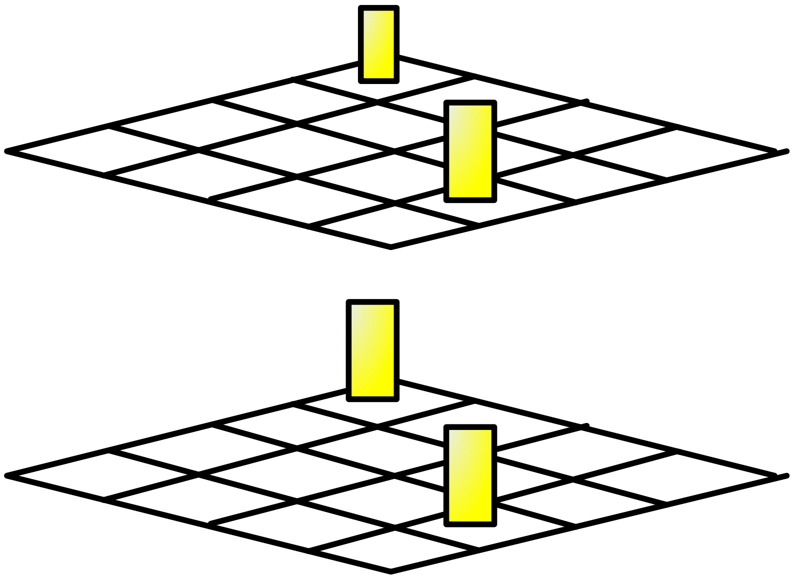


Figure 2: (a) True Perspective (b) Isometric Perspective

One of the benefits of isometric perspective and its lack of perspective scaling, which keeps all the objects the same size no matter their placement in space, is that it makes isometric perspectives easy and reliable to measure [34] because all the proportions are maintained. This can be seen in the following figure of a cube that has all its edge equal in length.

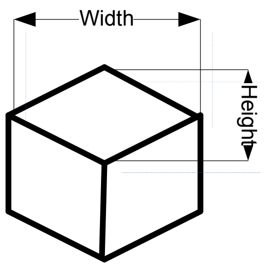


Figure 3: Isometric Perspective

Below is a basic example of a 2.5d environment as it could look in a game environment.

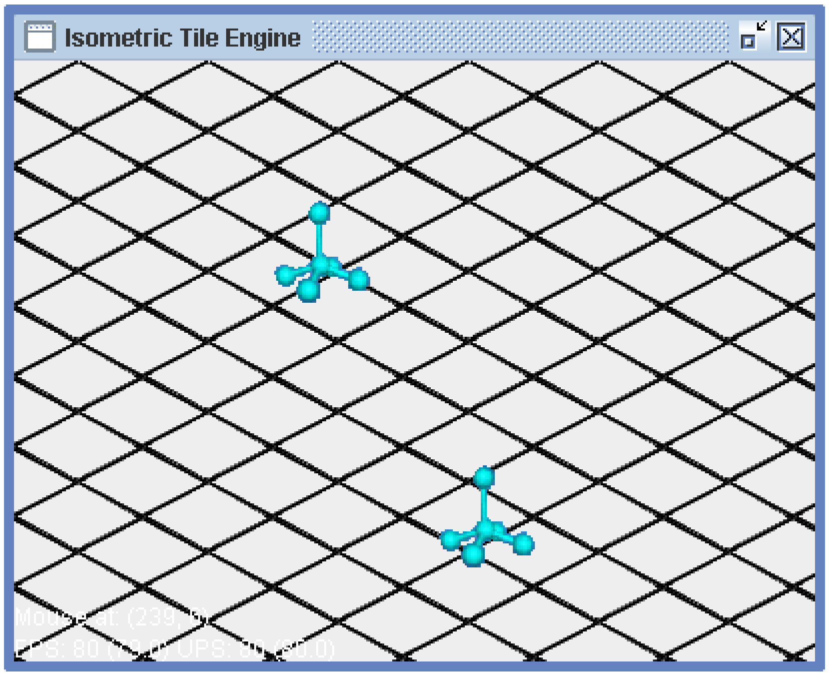


Figure 4: An Example isometric environment

An element of isometric game worlds is that some objects, represented by NPCs or Agents can move independently of each other or the background within the environment. This helps to create a sense of depth for the gamer [29].

Pathfinding is a sub-topic of the larger topic AI. AI has several different definitions depending on the source [14] but they all generally align along two main dimensions which are thought processes and reasoning and secondly, behaviour [14]. In a very basic sense this breaks down to how systems think and how they act with the degree of AI being how closely they think or act like human beings or think or act rationally.

In AI, the Turing Test was proposed by Alan Turing to provide a satisfactory definition of intelligence [14]. In its most basic form, the Turing test a computer shows it is intelligent if it can pass a test that compares its results to that of an undeniably intelligent entity which, in this case, are human beings. A computer passes the test if a human interrogator, after posing some questions, cannot tell whether the resulting responses are from a person or not. If the computer passes it is intelligent [14, 30].

For the study of pathfinding strategies, this is important because the ultimate result of developing a strategy is to have the NPC and Agent objects in the game world appear to move as if they were controlled by a human being. In essence, they should be able to pass a limited type of Turing Test.

The distinction between Cooperative, Non-Cooperative and Antagonistic or Competitive environments was discussed previously. In a Cooperative environment all the NPCs and Agents have knowledge about all other NPCs and Agents and take this into consideration in developing an individual route that does not interfere with any other routes from a starting point to a destination [14, 27]. An example of this is Doubles Tennis where there are a team of two players facing off on each side of the net. When the ball is hit to one side both players evaluate where the ball will land and make decisions on where to move to depending on if they feel they can reach the ball or they feel their partner can reach it. Thus they developed plans for themselves based on the actions of the other player.

The Non-Cooperative environment is one in which the NPC and Agents have no knowledge what so ever of each others actions or plans and must try to predict each others future movements [22].

Antagonistic or Competitive Pathfinding is pathfinding where all the NPCs or Agents goals are in conflict. Each NPC or Agent tries top reach its goal while, at the same time, preventing other Agents from reaching theirs. Basically, Agents make plans to move from their starting point to their destination without taking into consideration any information from the other NPCs or Agents and does not concern itself if the path it selects will be in conflict with another NPC or Agent, in fact this conflict is agreeable. This gives rise to what are known as adversarial search problems which are problems that arise when we try to plan ahead in a world where other NPCs or Agents are planning against us [14, 22].

In discussing the topic of this paper another distinction must be defined, that of fully observable and partially observable. As discussed previously, an environment is fully observable if all aspects that are relevant to the choice of action are known at each point in time [14]. This makes it more convenient for the NPC or Agent because they do not have to maintain an internal state to keep track of the world in which they interact. If they desire to move they can reference the fully observable state of the environment at that time to use in determining the optimal action path.

Opposed to fully observable is partially observable in which not all states of the environment are known at each point in time [14]. An NPC or Agent in a partially observable environment needs to maintain an internal view of their world they use when making action decisions and add to it as they move throughout their environment.

For the discussion contained in this essay, we will only be concerned with NPCs and Agents in fully observable environments.

In our discussion of algorithms we will be using graphs and maps to help represent the area within which the algorithm works. This area can be comprised of many different shapes such as squares, hexagons or even a spatial representation of a game tree and does not need to be a map in the literal sense [11]. One of the easiest representations to understand for our discussion is the graph. The graph contains all the elements that will be dealt with in this paper.

To start, a graph is a graphical representation of a set of objects that are connected by links that are commonly called edges [35]. These object that are being represented are often called vertices or nodes depending on the preference of the author [11, 35]. Nodes or vertices are data structures that represent points on a map [31]. Each point on the map or graph has a representative node that contains information representing that point. For this paper both node and vertex will be used interchangeably.

One of the hardest terms to define is what is meant by the cost of a node [31]. The cost of a node represents the desirability that a path should pass through the node. An example of this arises in most strategy games when there are mountains contained in the terrain. The cost for an NPC or Agent to pass through the mountains as part of their path is higher then if they were simply passing through an open field. The cost of a node can be based on several factors such as time, energy, money or terrain [31].

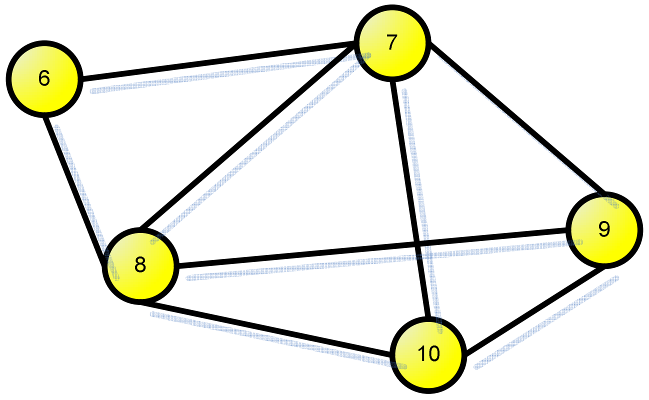


Figure 5: A labeled graph with 5 vertices/nodes and 8 unlabelled edges

As mentioned previously, we will be discussing the area in which the algorithms work; this area is commonly referred to as a state space or a search space. A search space is the set of all possible solutions in a search area [14]. As such, a search space represents all of the paths between vertices in our algorithms.

A state space is what a state space search searches in. A state space search is a process in which successive states of an instance are considered with the intention of finding a goal state with a desired property [36]. This essentially means that various possible solutions are tried in an attempt to reach a goal state. This has an obvious tie in to the subject of this paper in that we are searching through an area attempting to find a path from a starting point to a goal.

The state space is implicitly defined by the initial state which is the start for the search and the successor function which is the description of all possible actions available to an agent [14]. The state space forms a graph where the nodes are states and the edges or arcs between the nodes are actions [14]. Additionally, a path in the state space is a sequence of states connected by a sequence of actions [14].

Organization of the Remaining Chapters

Aside from the Introduction chapter, this essay is divided into four additional chapters, which are briefly described below. These remaining chapters support the statement of purpose of this paper and provide the necessary detail to reach a conclusion.

Chapter two will be a review of previous research and related literature that acts as the basis of knowledge for this paper. This will include a critical appraisal of current research from print sources and the internet discussing various pathfinding strategies including the A* algorithm and variants, as well as, pathfinding related techniques. While various pathfinding strategies will be discussed, this section will not engage in any in-depth discussions of the algorithms beyond that which is needed to explain the pertinent research and findings in light of the statement of purpose of this paper.

Chapter three will focus on discussing various pathfinding strategies and comparing them in order to discern what is the optimal pathfinding strategy for our 2.5D game world. The result of this chapter will be to identify, through analysis only, the optimal pathfinding strategy for our 2.5D environment. This result will then be tested in a 2.5D test environment which is the subject of the following chapter.

Chapter four will deal with taking the research from the previous chapters and use it to answer the statement of purpose. This will be accomplished by using a test environment that will run and measure each of the algorithms and then compare the results to determine the overall best algorithm. These results will be compared to our analysis of the algorithms preformed in the previous chapters to see if our analysis was proved correct. This will take into account the various success factors and will end with a recommendation of which pathfinding strategy is optimal under the assigned conditions. The overall comparison of these algorithms will take into account several different test environments designed to test the abilities of the algorithms as well as several different interaction schemes. The first of these interaction schemes takes into account the environment where the NPC and agents act in a cooperative manner, the second will address a Non-Cooperative scheme and, lastly, the Antagonistic or Competitive scheme will be discussed.

Chapter five will bring together the research from the previous sections and present the conclusions and recommendations. It will also identify any future areas of research in this field.

CHAPTER 2

REVIEW OF RELATED RESEARCH

There has been an abundance of research into pathfinding strategies over the years. Following is a collection of research that has been conducted that address the statement of purpose of this paper. The breath of research in this area made the act of gathering the best articles difficult but I have tried to maintain a common goal of only addressing research that directly relates to the topic at hand without getting into esoteric research that does not add to the overall base of knowledge needed to understand this topic.

In a discussion of pathfinding there are several algorithms that should be discussed along with their accompanying research. Following is a brief introduction to these algorithms and research. The intent of the following research review is to assist the reader in seeing the historical and logical progression of research and study in pathfinding strategies. This section will then wrap up with a brief synopsis of various research that has been carried out with regards to the A* algorithm. This research will be further expanded in the following chapter.

Dijkstra's Algorithm

The first pathfinding strategy to be discussed is Dijkstra's algorithm which was first conceived by E. W. Dijkstra in 1959 in his paper *A Note on Two Problems in Connexion with Graphs* [1, 2]. In this paper Dijkstra discusses what he sees as problems with graphs and proposes a solution in the form of an algorithm designed to find the shortest paths from a starting point to all the other points in a weighted, directed graph which simply means that it is a graph, or set of connected items, whose edges, or connections between its points, are in ordered pairs meaning that each edge can be followed from any point to any other point and each of these edges has a weight or numeric value associated with it [1, 3].

In [1], Dijkstra solves two basic problems that arise when we consider a graph with *n* vertices, which are simply an item on a graph and can be referred to as points or nodes, that have all pairs connected by a positive length edge [1, 4]. The problems that Dijkstra solves are as follows: “Problem1: construct the tree of minimal total length between the *n* vertices. (A tree is a graph with one and only one path between any two vertices.) Problem 2: find the path of minimal total length between two given vertices” [1].

Dijkstra’s algorithm solves the problem of finding the shortest path from a starting point to the destination in a graph as long as the edges between the points are not negative [1, 5]. In order to find the shortest path from the starting point to the destination, Dijkstra’s algorithm has to discover all the paths to all points on the graph meaning that in order to find the shortest path from the source to the destination we have to find the path from the source to all other points. This problem is sometimes called the single-source shortest paths problem [5] and occurs because Dijkstra’s algorithm always visits the closet unvisited node from the starting node and, as such, does not search towards the goal node [12].

Although Dijkstra’s algorithm does successfully find the shortest path from a starting vertex to a destination vertex in a graph and as such is admissible, it is easy to see that it does have substantial overhead associated with it due to the single-source shortest paths problem. As such it is not considered optimal for most pathfinding applications.

Best-First Search (BFS) Algorithm

Another common algorithm, the BFS, is another algorithm that attempts to be optimal in regards to pathfinding. The BFS algorithm has been discussed in research by many researchers, this paper focuses on the work of Stuart J. Russell and Peter Norvig entitled *Artificial Intelligence – A Modern Approach*. This text gives a very good overview of several search strategies that fall under the informed (heuristic) search strategy umbrella which includes the family of BFS strategies. It must be remembered that the field of study called Pathfinding falls under the very large and encompassing topic of AI. As such, there is a large selection of AI research that directly references the topic of Pathfinding.

The BFS algorithm is an informed search strategy algorithm meaning that it uses specific knowledge that is obtained beyond the definition of the problem in order to solve the problem [14]. It is easy to see the informed search algorithms can be more efficient then uninformed search algorithms due to this additional knowledge that helps the informed algorithm to solve the problem. There is a whole family of BFS algorithms based on their different evaluation functions. The evaluation function is used by the algorithm to select a node for expansion and is commonly denoted by $f(n)$ [14].

A key component of the evaluation function is called the heuristic function [14]. The heuristic function, commonly denoted as $h(n)$, is the estimated cost of the cheapest path from the current node, denoted as n, to the goal node [14]. The current node is the node that is being evaluated at that time.

$$\boxed{h(n)}$$

Figure 6: Common Identifier for the Heuristic Function

The BFS algorithm explores a graph by expanding the most promising node based on its heuristic [8, 10]. In a basic sense, the BFS algorithm acts completely opposite of Dijkstra's algorithm. Instead of always picking the closest node to the starting node, it always picks the node that is closest to the goal node on the assumption that by doing this it will find the solution quickly hence the BFS algorithm being called greedy [14]. Since we do not know the distance to the goal node this has to be estimated [12]. This estimate is a heuristic and is used in the evaluation function. A heuristic is an algorithmic technique that usually, but not always, works or gives an approximate answer [9]. As such, the BFS algorithm evaluates nodes by using just the heuristic function, which is denoted as follows:

$$\boxed{f(n) = h(n)}$$

Figure 7: BFS evaluation function

<u>The A* Algorithm</u>

The next logical step in pathfinding research came in 1968 with the introduction of the A* algorithm by [6]. They introduced the A* algorithm in their research paper *A Formal Basis for the Heuristic Determination of Minimum Cost Paths*. In this paper they developed an algorithm that could find the shortest path similar to Dijkstra's algorithm, but uses a heuristic to guide it in finding the best path similar to the BFS algorithm [8].

In their work on developing the A* algorithm, [6] expanded upon the heuristic evaluation function of the BFS algorithm. They showed that A* is admissible, meaning that it is guaranteed to find an optimal path from a starting vertex to a preferred goal node for any graph [6].

The A* algorithm changes the evaluation function, *f(n),* first seen in the BFS algorithm to the following:

$$f(n) = g(n) + h(n)$$

Figure 8: A* evaluation function

Where ***g(n)*** is the actual cost of an optimal path from the source to the current node, the cost to reach the node and ***h(n)*** is the actual cost of an optimal path from the source node to the goal node, the cost to get from the node to the goal [6, 11, 14].

Making this change in the evaluation function made the A* algorithm admissible but also, compared to Dijkstra's algorithm, uses reduced memory and time requirements. Effectively, A* combines the best characteristics of Dijkstra's algorithm and the BFS algorithm.

While the memory and time requirements for the A* algorithm is less then that of Dijkstra's algorithm they are still considered high and at times prohibitive, especially when it has to deal with a large state space which is commonly a map or graph. Surprisingly, time requirements to run A* does not appear to be the biggest drawback, it seems that because the algorithm maintains a list of nodes in memory it will actually run out of space in memory before it runs out of time and as such is, it is not practical for large scale pathfinding problems [14].

Even with the drawbacks mentioned, the A* algorithm is by far the most popular pathfinding algorithm used in gaming and similar applications today [11]. As such, much of the attention of researchers has been directed to studying A* and trying to improve it in general or for specific applications. Following is a brief sampling of some of the research that has suggested variations to the A* algorithm. The research that is being discussed is presented in no specific order unless it is deemed that understanding of one variation is needed to understand another.

Table 1: Comparison of Pathfinding algorithms

Algorithm	**Uses Heuristic?**	**Speed**	**Admissible**
Dijkstra's	No	$O((n+m) \log n)$ [33]	Yes
BFS	Yes	Depends on heuristic	Yes
A*	Yes	Depends on heuristic	Yes

<u>Iterative Deepening A*</u>

One of the problems with the A* algorithm is that, similar to Dijkstra's algorithm, it is very memory intensive [14]. Many of the variations come out of the need to lower this memory cost. One of the simplest ways proposed to reduce the memory requirements of the A* algorithm involves adapting the idea of iterative deepening to the heuristic function. Adding iterative deepening to the heuristic results in a variation of the A* algorithm called the iterative-deepening A* (IDA*) algorithm [14].

The IDA* algorithm started out in the 1960s as a way for programmers and researchers to address finding a reliable mechanism to control the time consumption of tournament chess games [15]. Essentially, they hit upon the idea of extending the idea of searching only one depth of the state space to subdividing up the task into separate depth-first searches with successively deepened search horizons or cost thresholds [15].

In [15], the author explains that IDA* performs a series of cost-bounded depth-first searches while successively increasing the cost thresholds. This means that IDA*, at each iteration of the algorithm, does a search and removes all nodes that exceed the set fixed cost bound that was determined at the first iteration of the algorithm. The cost bound is set to the heuristic estimate of the initial state [15] and, with each iteration, is increased to the minimum path value that exceeds the previous bound [15, 17]. In simple terms, this means that IDA* addresses the memory issues seen with the A* algorithm by not adding nodes to memory that do not satisfy the depth determined by the iterative-deepening function.

[16] Discusses the problem with BFS which is that it must store all the nodes in memory in order to determine the best node to expand next. For large environments, this requires a significant amount of memory which grows linearly with the running time and increases exponentially with the size of the problem [16, 17].

The author sees iterative deepening as one of the possible solutions. Iterative deepening performs a series of depth-first searches, each with a depth limit that is one greater then the previous iteration [16, 17]. This continues until a solution is found. As such, iterative deepening uses less memory then A* and still is admissible.

In [18], iterative deepening is seen as a way to impose an artificial limit that reduces the total search space on the search algorithm. This works to lower the memory demands that is a problem with algorithms such as A*. As mentioned previously, IDA* means that the algorithm is called repeatedly in iterations, starting with a small value for the path length which increases on each iteration. If the call to the algorithm fails to find a path it will gradually relax the limit in subsequent calls to the algorithm until the search succeeds or the limit reaches the maximum value which would mean that a path is not possible [18].

The research in [18] explains that there are several advantages to using IDA* over A*. The first is obvious and has been mentioned previously, that IDA* uses less memory then A* for most calls because the artificial limit results in fewer nodes being examined and expanded [17, 18]. The additional benefit identified is that IDA* speeds up the detection of almost-straight paths and improves performance in environments where there are many other agents attempting to pass the same point at the same time [18]. This additional feature of IDA* will be further elaborated on in the following chapter.

The purpose of this study was to introduce the IDA* algorithm which is an extension of the A* algorithm. It differs from A* in that it determines a successively deepened search horizon on each iteration that is used by the algorithm to limit the number of nodes that are expanded and as such maintained in memory. It addresses the memory issues of the A* algorithm.

Abstraction

Another newer variant of the A* algorithm that was first proposed in 2004 is called Hierarchical Path-Finding A * (HPA*). The research in [19] proposes a hierarchical approach for reducing the complexity of path-finding on grid-based maps which extends the A* algorithm by creating an abstraction of the map into linked local clusters [19].

The HPA* does not rely on a specific topology and will work with our 2.5D environment, as well, it gives the advantage of being relatively simple compared to other extensions for the A* algorithm and is easy to implement [19]. The authors suggest that HPA* is up to 10 times faster then compared to highly optimized A* [19].

[20] Extends the work proposed by the HPA* algorithm. In this paper the authors bring together several related pieces of work on using abstraction for pathfinding with a focus on implementing the algorithm with the least amount of memory usage [20]. In fact the authors conclude that their proposed technique uses only about 3% additional memory but results in computing paths up to 100 times faster then just the A* algorithm by itself [20].

Abstraction means transforming a large map into smaller representations of the larger map in order to speed pathfinding. This is a natural concept for humans as we use abstraction for solving many large problems [20]. In abstraction a high level path is determined using the abstract pieces of the state space which can be a graph or map; this then leads to determining the near optimal path in the larger non-abstracted, local sections of the map.

The purpose of the above study was to discuss the topic of abstraction in regards to the A* algorithm. Abstraction involves creating representations of a larger state space. This allows A* to perform a high level search that coupled with a lower level search ensures that the agent is moving in the near optimal route to the goal.

Real Time Adaptive A*

Abstraction works in speeding up the A* algorithm but there are many other ways to increase the performance of the A* algorithm. One way that has been propose is called Real-Time Adaptive A* (RTAA*) [21]. With RTAA*, the heuristic for the A* algorithm gets updated between searches that use real-time search methods [21]. RTAA* uses real-time search methods combined with the A* algorithm.

Real-time search methods restrict the search to a small part of the state space which can be reached from the current state or node with a small number of action executions [21]. Basically, the agent determines the state space that will be used for the initial search. The agent then searches the state space and determines what the path will be. Once determined the path is executed to a point on the resulting trajectory. This process is then repeated until the agent reaches the goal state [21].

Real-time search methods do not plan the complete path to the goal state but create a path for the agent to move through to the final goal state. This can be seen as being similar to the abstraction method of pathfinding in that they both do not find a complete path but, instead determine parts of the path using waypoints.

With real-time search methods, this usually results in shorter search times due to the smaller search spaces but larger trajectory costs due to the entire path being broken up and determined separately [21]. A side effect of these benefits is that real-time search methods are adept at pathfinding for very large state spaces.

With RTAA*, the A* algorithm is the algorithm of choice that finds the points on the trajectory or, eventually, the goal state. It is a novel approach to pathfinding using real-time search methods and the A* algorithm to create a real-time heuristic search method. In RTAA* the local search state space is determined in a fine grained way while the heuristic for each state is updated in the local search space very quickly [21] resulting in a A* like algorithm that is overall faster then original A* and is considered by the authors, easy to implement [21].

The purpose of this study was to introduce real-time search methods to the A* algorithm which address the performance issues of the A* algorithm.

Cooperative Pathfinding A* Variations

Cooperative pathfinding was described previously in this paper as pathfinding involving many agents that must find non-colliding routes to separate destinations given full information about the routes of the other agents [22].

In his paper, [22], David Silver discusses several variations on the A* algorithm that work well in a cooperative environment and work to solve the multi-agent pathfinding problem. Specifically he discusses Cooperative A* (CA*), which searches space-time for a non-colliding route, Hierarchical Cooperative A* (HCA*), which use an abstract heuristic to boost performance and finally, Windowed Hierarchical Cooperative A* (WHCA*), which limits the space-time search depth to a dynamic window which spreads the computation of the path over the duration of the route [22].

CA* breaks up the task of moving multiple agents into individual single agent searches that is performed in three dimensional space-time. These single agent searches use a reservation table that represents the shared knowledge about each of the agents planned routes. The path entries in this table are considered impassable and are avoided during pathfinding searches for subsequent agents [22]. This enables the agents to find non-colliding paths to their goals.

HCA* uses abstraction of the state space in order to determine the paths for the multiple agents. In the case of HCA*, the abstraction is a simple single domain, 2-dimensional map with all the agents removed that ignores both the time dimension and the reservation table [22]. With the state space abstracted this way, abstract distances can be viewed as perfect estimates of the distance to the goal, ignoring any and all possible interactions with the other agents [22]. This result is an admissible and consistent heuristic with the inaccuracy of the heuristic being determined only by how much the agent must deviate from its path to avoid contact with other agents [22].

CA* and HCA* have several drawbacks. Firstly, they do not address what to do with the agents once they reach there goal. This becomes an issue if the goal was in a narrow point, the agent could prevent other agents from passing the point resulting is a backup and pathfinding chaos [22]. The agent must move off its goal and allow other agents to pass onto their goals. This involves further cooperation beyond simply finding the non-colliding paths to their goals.

The second issue with the above strategies is that they do not address agent ordering. A robust solution that considers the sensitivity of agent ordering varies the agent order and assigns each agent in turn the highest priority for a short period of time [22]. This enables the strategy to solve problems that would be unsolvable with an arbitrary or fixed agent order.

Thirdly, CA* and HPA* must calculate the complete route to a goal in a large three-dimensional state space [22] which is more time and space prohibitive then what is desired.

WHCA* overcomes the above challenges by simply windowing the search. Windowing simply involves taking a smaller set depth of the state space to search rather then the entire state space. This results in the agent searching only a partial trajectory to the goal. Once the partial trajectory is calculated, the agent begins to move, following the route. At regular intervals such as when the agent is half way through the partial route, the window is shifted forward and a new partial trajectory is calculated.

Due to problems that could arise due to the partial trajectory seeming to be the optimal route but in reality being a dead end, only the cooperative search depth is limited to a fixed depth, that is the pathfinding that takes into account all the other agents, while the abstract search, the single domain, 2-dimensional abstraction, continues to be executed to the full depth of the map [22]. This ensures that the agents are headed to their goal on a near optimal path.

While WHCA* is not without its problems. The main problem it faces is that of the horizon effect [23]. The horizon effect is caused when agents plan just enough to push congestion past the windowed horizon resulting in agents behaviors looking bizarre and unreal [23].

[22] Performs experimentation to determine the most effective of the three strategies in various situation using Local Repair A* (LRA*) as a comparison. LRA* has every agent search for a route to its goal using the A* algorithm, ignoring all other agents except those that are its closest neighbors [22]. Once the system begins to run, the agents follow there routes until a possible collision appears imminent. When it appears that an agent is going to try and move into the space of another agent, the original agent’s route is then recalculated to not try and move into the occupied position.

Another possible solution to the multi-agent pathfinding problem can be seen in [23] in which the authors wish to create a WHCA* type strategy but without the horizon effect. Their solution involves using a direction map which essentially replaces the static reservation table with a dynamic new data structure that stores the direction that the agent last moved as each location [23]. This enables the strategy to determine ahead of time if there is the possibility of a collision and allow it to prevent it occurring without the agents explicitly communicating their plans.

The purpose of this study was to identify several variations of the A* algorithm that address and overcome the multi-agent pathfinding problem that affects multiple agents in a cooperative state space.

Comparison of A* Algorithm Extensions

For comparison and reference, below is a matrix of the A* algorithm changes that have been discussed in this chapter. This is designed as a comparison of the changes made to the A* algorithm and the item that the change addresses in the A* algorithm or previous extensions.

Table 2: Comparison of changes to A*

Algorithm	Changes to A*	Addresses
IDA*	Multiple depth-first searches with successively deepened search horizons	A* memory issues
HPA*	Abstraction	A* Performance
RTAA*	Updates heuristic between searches and real-time search methods	A* Performance
CA*	It is cooperative. Breaks up task of moving multiple agents into individual single agent searches. Also uses a reservation table.	Multi-agent pathfinding problem
HCA*	Abstraction plus CA* changes minus the reservation table.	Multi-agent pathfinding problem
WHCA*	Uses windowing for search.	Multi-agent pathfinding problem plus the drawbacks found in CA* and HCA*
Direction Map A	Direction Maps	All items from WHCA* plus the Horizon effect

* **NOTE:** This change was not given a name in the research, as such, Direction Map A* is a description of the algorithm changes and not the name of the algorithm.

Alternative Heuristics

The most common heuristic for the A* algorithm is Manhattan Distance, sometimes called city block distance [14, 24]. As with all heuristics for the A* algorithm, it estimates the remaining distance to the goal. Manhattan distance does this by summing the horizontal and vertical distances to the goal. While this is considered ‘good enough’ for most applications there has been research into developing better heuristics that have better performance.

While this is a large topic that is beyond the scope of this paper, I will discuss a new heuristic and the related research around it. In the next chapter we will look into several additional heuristics and investigate how they affect the performance of the A* algorithm.

[24] Discusses several optimizations for the A* and IDA* algorithms. Among them are two optimizations that are relate directly to A*. Firstly, the author discusses an optimization that involves the use of an array of stacks rather then a priority queue for maintaining the open list of nodes. This does offer some interesting possibilities but is beyond the scope of this paper and does suggest a further topic for research.

In regards to heuristics, [24] proposes a new heuristics called ALTBestp [24]. The author discusses this heuristic in relation to the Manhattan distance heuristic and the ALT heuristic. The Manhattan heuristic was discussed above, the ALT heuristic works by first pre-computing the distance to all points from a given point, such as the starting point, and then uses these pre-computed distances to calculate an admissible heuristic [24].

The author believes that a better variation on the ALT heuristic is the ALTBestp heuristic which uses pre-computed distances from P points but instead of taking the maximum value over the h values computed with the P points at each node of the search, it selects among the P points the one that gives the highest h value at the root node [24]. For all the nodes in the search, it chooses as the h value the maximum of the ALT value computed with the selected point and of the Manhattan distance [24]. The advantage of the ALTBestp heuristic is that is takes less time at each node, making the overall search process faster while at the same time not being any worse then other heuristics such as Manhattan distance [24].

The purpose of this study was to develop a new heuristic that would increase the performance of the A* algorithm beyond the performance it normally gets when implemented using the Manhattan distance heuristic. Additionally, this research goes to show that different heuristics can be developed that address performance issues with A*.

CHAPTER 3

PATHFINDING STRATEGIES ANALYSIS

The preceding chapter discussed research that has been conducted into various pathfinding strategies and algorithms over the years. This chapter builds upon this previous research by analyzing several algorithms and pathfinding strategies, specifically Dijkstra's algorithm, BFS and A*, by subjecting them to an in-depth technical analysis that will look at how the algorithms function with a goal of being able to deduce what we should see in our actual testing of the algorithms and pathfinding strategies.

This chapter will focus on discussing the identified popular pathfinding strategies and comparing them in order to discern what is the optimal pathfinding strategy for our 2.5D game world. The result of the investigation that is described in this chapter was to identify the optimal pathfinding strategy for our 2.5D environment based on analysis of the main algorithms. The chapter that follows this one will take the results of the analysis conducted in this chapter and evaluate and compare them in a test environment to see if the results of the analysis were correct.

Evaluation Criteria

The success of any given pathfinding algorithm or strategy depends on the requirements and assumptions of the application that is being evaluated. For our investigation, we considered the pathfinding strategies in regards to expected memory usage, relative speed and the ability to find the shortest path from the starting node to the goal node.

Dijkstra’s algorithm

To start our technical discussion of Pathfinding algorithms we will start with Dijkstra’s Algorithm. As discussed in the previous chapter, Dijkstra’s algorithm was first put forward by E. W Dijkstra who developed his namesake algorithm in 1959 [1, 2, 5]. This algorithm is a graph search algorithm that searches for and discovers the shortest path from a point in a graph, the source, to a destination [2, 5]. Dijkstra’s algorithm is greedy [33] which means that it attempts to make the locally optimal choice at each step with the overall goal of finding the globally optimal solution [1, 32, 33].

Following is an example of Dijkstra’s algorithm in action on a basic grid based state space. This state space presents an obstacle that must be navigated around in order to reach the goal. As can be seen in this image, Dijkstra’s algorithm ends up having to search almost all of the state space in order to discover the shortest path making it admissible but not necessarily optimal. It is easy to see that Dijkstra’s algorithm would require a significant amount of time and memory in order to discover a path from the starting point to the goal.

In the following figure, the starting point is represented by the green square; the goal is represented by the red square, the black squares represent obstacles which cannot be passed through, the purple squares are the squares in the state space in which the algorithm searches in discovering the shortest path to the goal which is represented by the yellow squares.

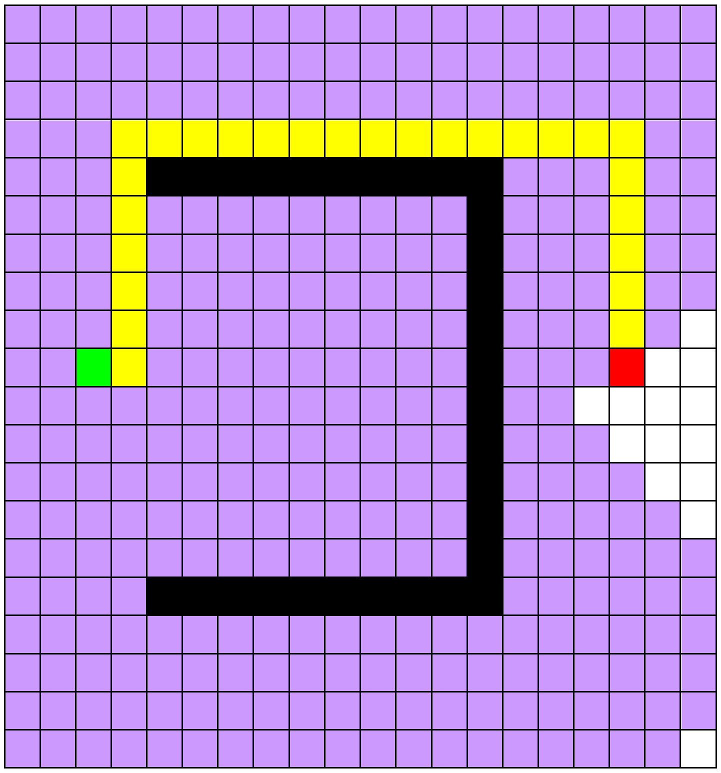
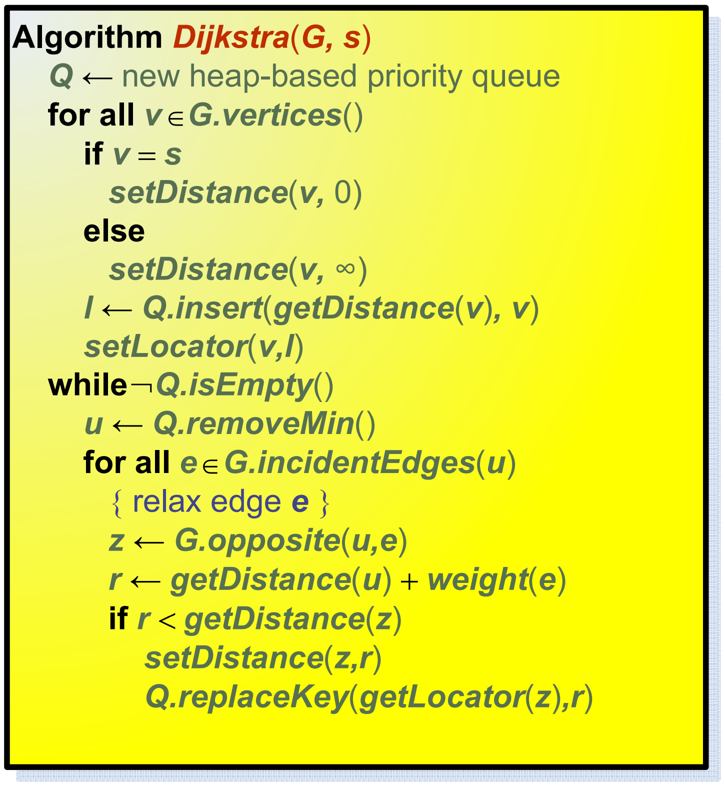


Figure 9: Grid based example of Dijkstra's algorithm

This example is interesting in that it shows how Dijkstra's algorithm functions when presented with a simple obstacle. The next question that must be addressed is why does this algorithm need to expand so many nodes when searching for a path to the goal? The answer lies in the way Dijkstra's algorithm works. In order for it to find a path to the goal it must discover all the possible paths from the starting node to all possible points, this is called the single-source shortest paths problem [5] and is the biggest downfall of Dijkstra's algorithm [5, 14]. It requires a large amount of memory in order to calculate the optimal path to a goal with the size of the state space being a primary factor for determining how much memory is required.

In order to illustrate the functionality of Dijkstra's algorithm, following is an implementation of Dijkstra's algorithm in pseudo-code that we will use to explain how the algorithm works. There are many different ways to write pseudo-code. This paper uses the style put forward in [33]. This pseudo code style is used through this paper when describing code snippets. An explanation of this pseudo code style can be found in [33].



```
Algorithm Dijkstra(G, s)
  Q ← new heap-based priority queue
  for all v ∈ G.vertices()
    if v = s
      setDistance(v, 0)
    else
      setDistance(v, ∞)
    l ← Q.insert(getDistance(v), v)
    setLocator(v,l)
  while ¬Q.isEmpty()
    u ← Q.removeMin()
    for all e ∈ G.incidentEdges(u)
      { relax edge e }
      z ← G.opposite(u,e)
      r ← getDistance(u) + weight(e)
      if r < getDistance(z)
        setDistance(z,r)
        Q.replaceKey(getLocator(z),r)
```

Figure 10: Dijkstra's Algorithm in pseudo-code [33]

The above pseudo-code for Dijkstra's algorithm shows the inner workings of Dijkstra's algorithm. As an explanation the steps involved in this algorithm, the following table describes the mode of operation for the algorithm, which explains the step by step process of executing the algorithm.

Table 3: Dijkstra's Algorithm mode of operation

Point	Mode of Operation of Dijkstra's Algorithm
1.	The input to the algorithm is an undirected weighted graph ***G*** with non-negative edge weights and a vertex ***s***.
2.	A new heap-based priority queue ***Q*** is initialized to hold all the vertices of ***G*** using the Distance label as keys.

3.	For each vertex ***v*** in graph ***G,*** assign the values in the distance list to infinity except for the starting vertex which is set to 0.
4.	While there are vertices in ***Q***, pull a new vertex ***u*** into the cloud.
5.	For each vertex ***e*** that is adjacent to ***u*** such that ***e*** is in ***Q,*** perform the relaxation procedure.
6.	The output of this algorithm is a label for ***u***, for each vertex ***u*** of ***G***, such that the distance is the length of a shortest path from ***s*** to ***u*** in ***G***.

To further enhance the understanding of this algorithm, following is a step by step example of it in action based on an example that can be found at [42]. The key for the example is as follows, the boxes are the nodes and are referenced by the letter that is above or below them; the edges are the arrows with their weight being the number contained in their centre and their direction being specified by the arrow head.

With figure 1, we start off with a graph ***G*** and vertex ***s*** which is point one. From this figure we see the initial state of a graph prior to running the algorithm against it. All nodes have an infinite cost except the source node which has a cost of zero. This references point three.

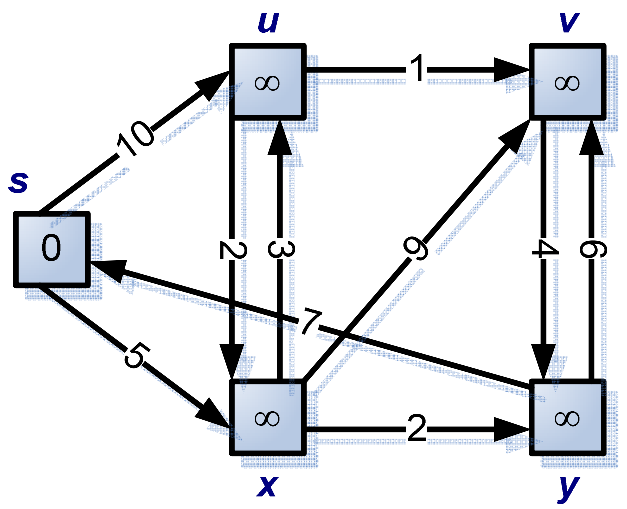


Figure 11: Example execution of Dijkstra's algorithm - Step 1 [42]

In step two of the algorithm, we first choose the node that is closest to the starting node ***s***. We initialize the distance to ***s***, denoted by d[s] to 0. We then add it to our priority queue ***Q*** and then relax all the nodes that are adjacent to the source node ***s***. We then update the predecessor, which is a structure for storing a path through a graph, for all nodes updated. This is represented by the red arrows in the figures.

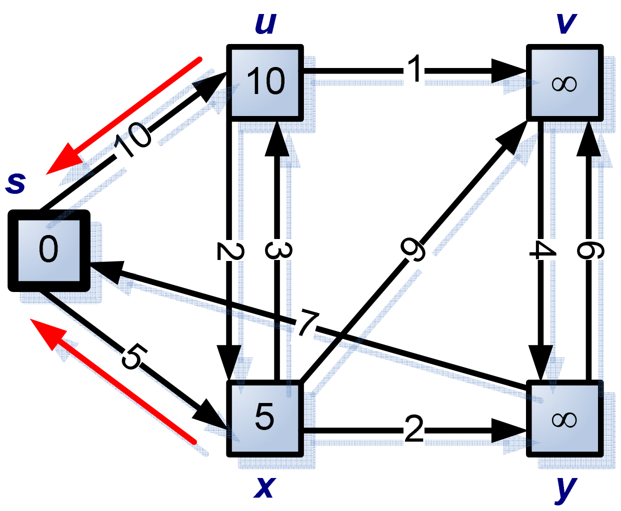


Figure 12: Example execution of Dijkstra's algorithm - Step 2 [42]

Step three of Dijkstra's algorithm proceeds as follows, we choose the closet node which, in this example, would be node ***x***. We relax all the nodes that are adjacent to node ***x***. We then update the predecessors for nodes ***u***, ***v*** and ***y*** which, again, are represented by the red arrows in the figures.

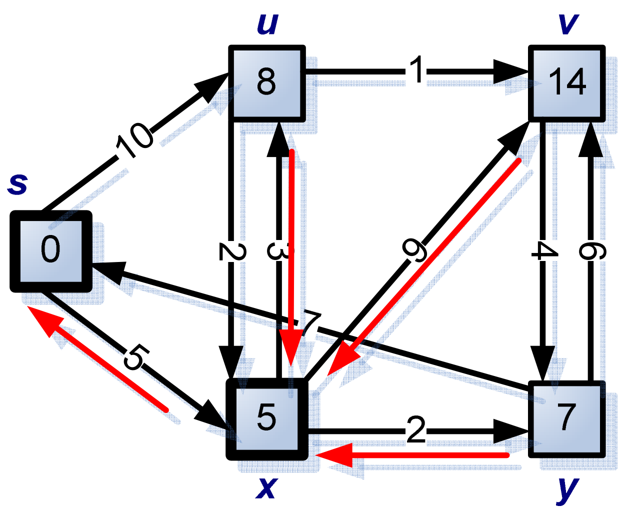


Figure 13: Example execution of Dijkstra's algorithm - Step 3 [42]

Step four follows similar to step three. In this step we again choose the closet node to our last closet node which in this case is node ***y***. We add it to ***Q*** and Relax node ***v*** and adjust its predecessor.

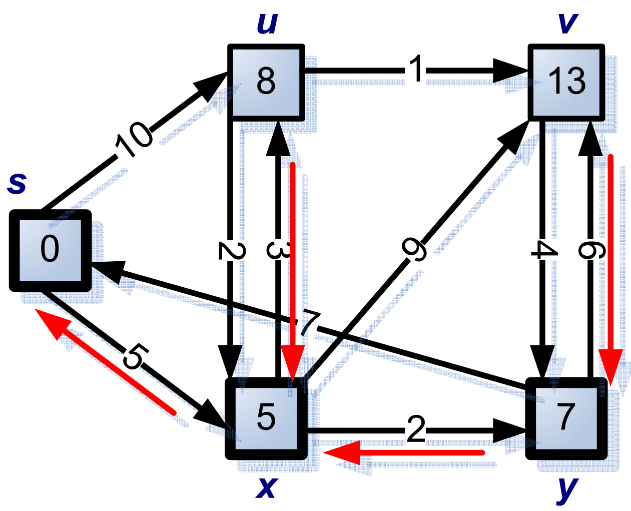


Figure 14: Example execution of Dijkstra's algorithm - Step 4 [42]

Step five with node ***u*** being the closet. We choose this node and adjust its neighbor node ***v***.

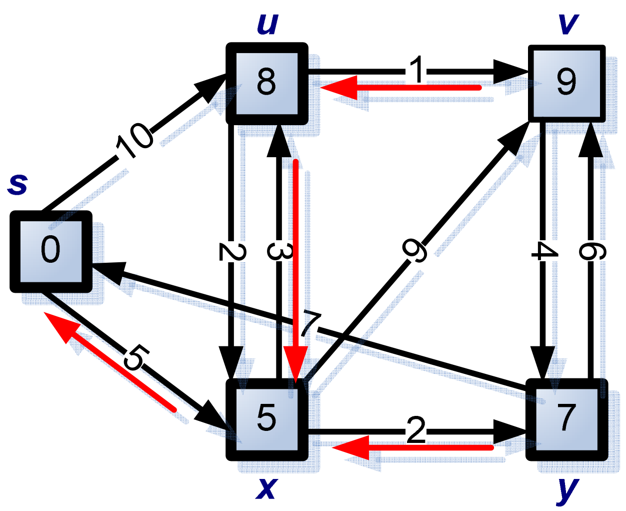


Figure 15: Example execution of Dijkstra's algorithm - Step 5 [42]

The final steps for this graph are as follows: we add node ***v***. The predecessor list now contains the shortest paths from each node to the source node ***s***.

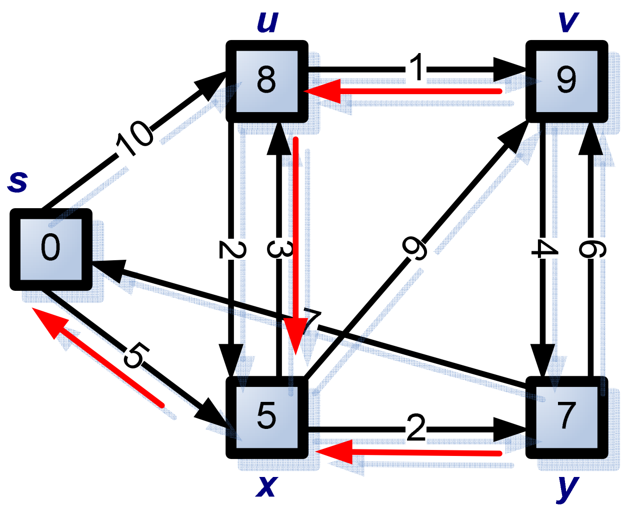


Figure 16: Example execution of Dijkstra's algorithm - Step 6 [42]

The above series of figures represents the outcome of the research conducted by Dijkstra to create an algorithm for finding the shortest path from a source to a destination. The above example shows how the algorithm works with a graph but this example can be extrapolated to a 2.5d isometric environment very easily.

As mentioned in the previous chapter, Dijkstra’s algorithm solves the single-source shortest paths problem by finding the shortest path from a single-source to all other vertices in a specific graph. This can be seen in the above example as each and every node is touched by the algorithm and the distance from each to the source is calculated for each.

Best-First Search

BFS works in a similar manner as Dijkstra’s algorithm except that it uses an estimate, called a heuristic, of how far away a vertex is from the goal, which was discussed in the previous section [8]. BFS is admissible, it will find a path to the goal if one exists but it is not guaranteed to find the shortest path but it does find the path faster then happens with Dijkstra’s algorithm [8].

This concept that BFS will find a path although it may not be the most optimal is easily seen in the figure below. In this figure we can see that BFS searches less of the state space then Dijkstra’s algorithm but it can also result in the expansion of nodes that are not actually on the shortest path to the goal. This results in a path to the goal that contains more steps then Dijkstra’s algorithm being run on the same state space but BFS does use less memory then Dijkstra’s algorithm, although it uses more then the A* algorithm which will be shown in the next section.

In the following figure, the starting point is represented by the green square; the goal is represented by the red square, the black squares represent obstacles which cannot be passed through, the purple squares are the squares in the state space in which the algorithm searches in discovering the shortest path to the goal which is represented by the yellow squares.

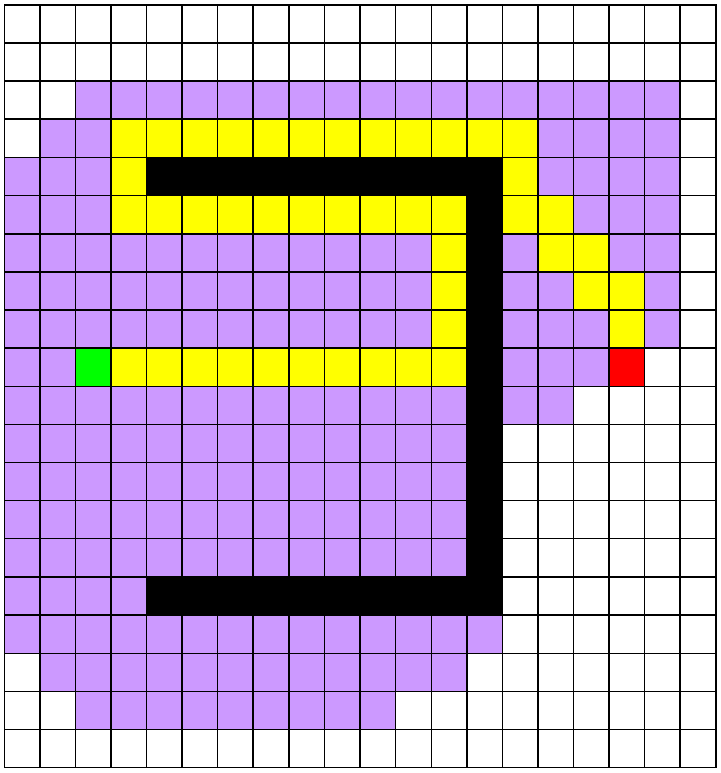


Figure 17: Grid based example of the BFS algorithm

BFS runs much quicker, in most cases, then Dijkstra's algorithm due to it not needing to expand as many nodes in search of a path to the goal. This is a result of the heuristic function being used to guide the algorithm towards the goal [8]. An example of this concept can be seen if the goal is oriented towards the south of the state space, the algorithm will direct its search towards the south instead of in all directions as Dijkstra's algorithm would.

<u>A* Algorithm</u>

The A* algorithm can be seen as a combination of Dijkstra's algorithm and BFS. It finds the shortest path to the goal the same as Dijkstra's algorithm but does it in a much quicker way just like BFS and with less memory usage then both Dijkstra's algorithm and the BFS algorithm. This is shown in the following example that uses the same map as the previous examples for Dijkstra's algorithm and the BFS algorithm.

In the following figure, the starting point is represented by the green square; the goal is represented by the red square, the black squares represent obstacles which cannot be passed through, the purple squares are the squares in the state space in which the algorithm searches in discovering the shortest path to the goal which is represented by the yellow squares.

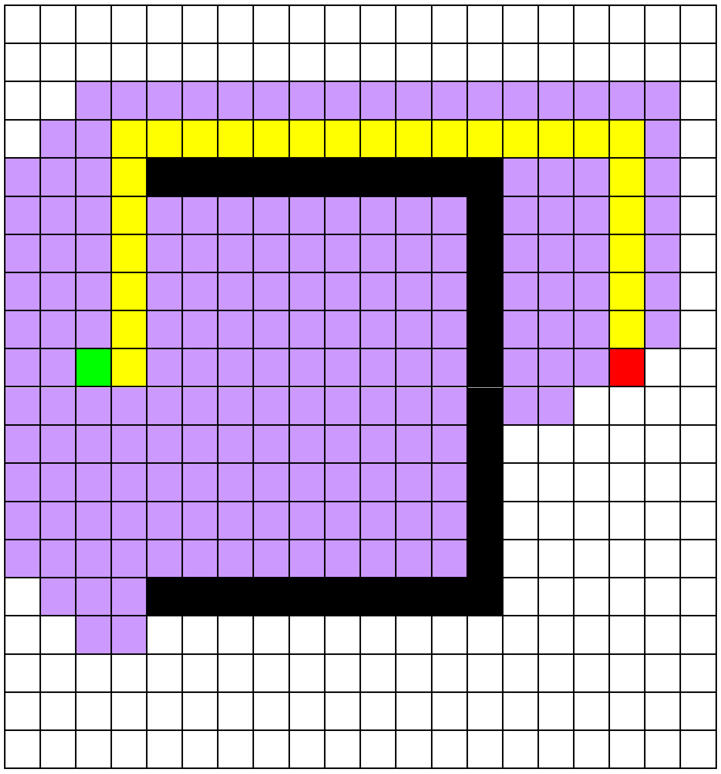
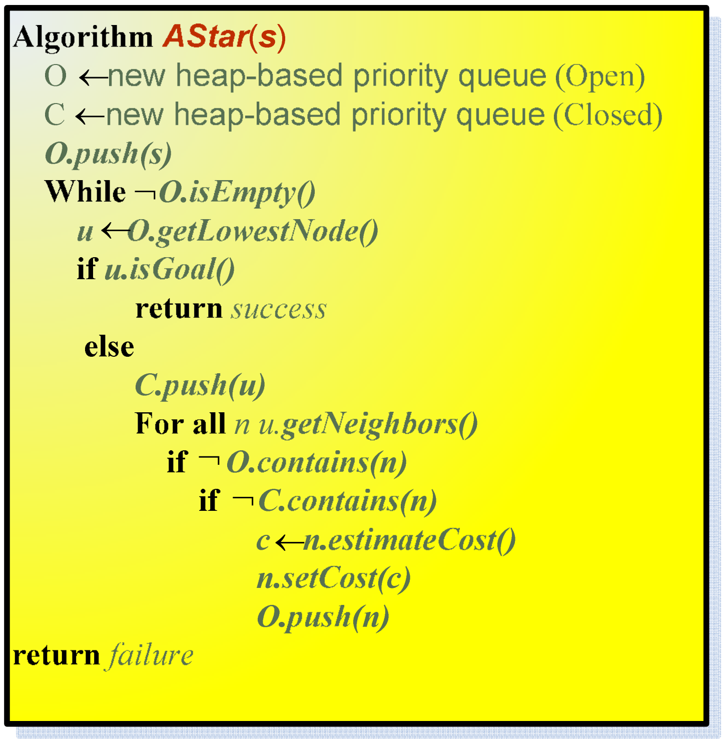


Figure 18: Grid based example of the A* algorithm

This example shows that the search area for A* is smaller then that of BFS due to the use of a refined heuristic (see chapter 2 for details) but it found a path the same as Dijkstra's algorithm. This was accomplished with less memory usage then Dijkstra's algorithm and the BFS algorithm.

The following figure shows a pseudo-code listing of the A* algorithm. There are several different variations of pseudo-code for the A* algorithm as can be seen in [2, 5, 33]. This implementation is based on analysis of several different pseudo-code variants by the author in order to create a variant in the style of [33].



```
Algorithm AStar(s)
   O ←new heap-based priority queue (Open)
   C ←new heap-based priority queue (Closed)
   O.push(s)
   While ¬O.isEmpty()
      u ←O.getLowestNode()
      if u.isGoal()
           return success
       else
           C.push(u)
           For all n u.getNeighbors()
              if ¬ O.contains(n)
                 if ¬C.contains(n)
                      c ←n.estimateCost()
                      n.setCost(c)
                      O.push(n)
return failure
```

Figure 19: A* algorithm in pseudo-code

Examination of the pseudo-code for an algorithm only gives part of the analysis for an algorithm. To support the above pseudo-code is a table followed by a basic flowchart of the mode of operation for the A* algorithm that adds additional comprehension to the above pseudo-code.

Table 4: A* Algorithm mode of operation

Point	Mode of Operation of A* Algorithm
1.	The input to the algorithm is an undirected weighted graph ***G*** with non-negative edge weights and starting vertex ***s***.
2.	Create the open list of nodes ***O***, initially containing only the starting node ***s***

3.	Create the closed list of nodes ***C***, initially empty
4.	If ***O*** is empty, exit with failure
5.	If ***u*** is a goal node, exit successfully with the solution obtained by tracing a path along the pointers from ***s*** to ***u*** in ***G***.
6.	Expand node ***u***, of its successors that are not already ancestors of ***u*** in ***G***. Install these members as successors of ***u*** in ***G***. Each successor is identified by ***n***.
7.	Establish a pointer to ***n*** from each of those members that were not already in ***G*** (i.e., not already on either ***O*** or ***C***). Add these members to ***O***. For each member ***O*** or ***C***, redirect its pointer to ***n*** if the best path so far is through ***n***. For each member already on ***C***, redirect the pointers of each of its descendants in ***G*** so that they point backward along the best paths found so far to these descendants.
8.	Go to Step 4.

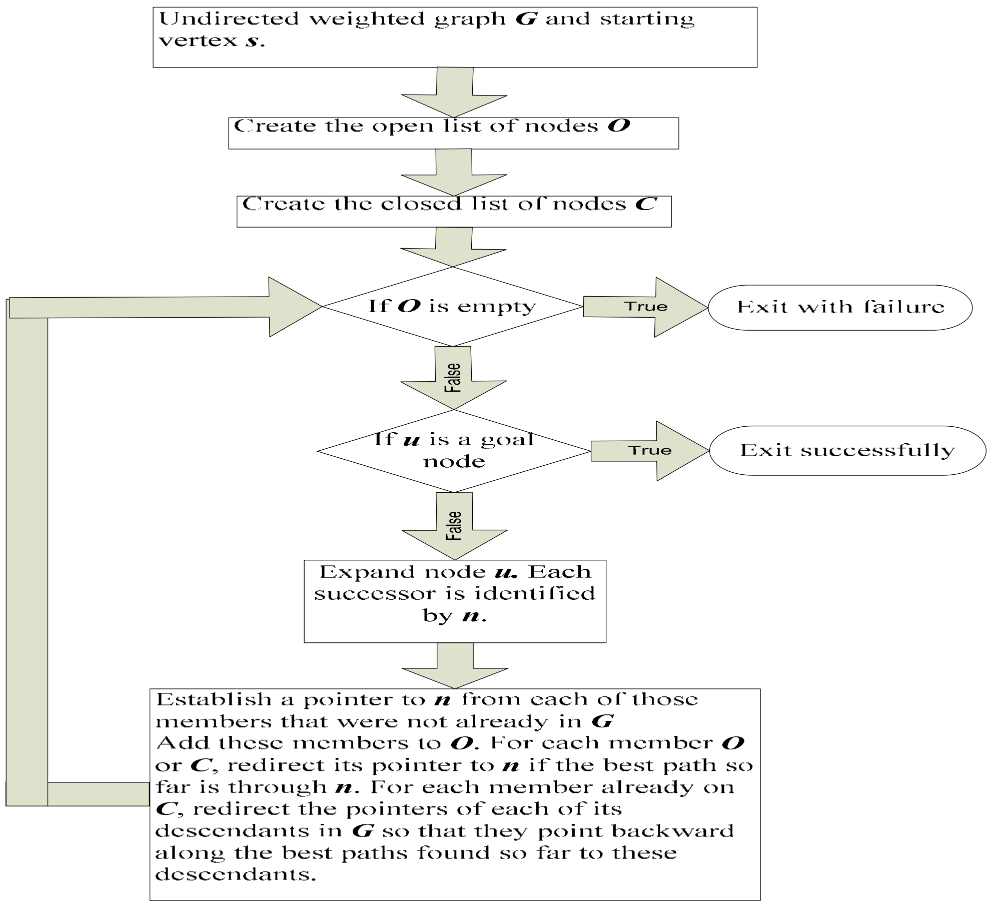

Undirected weighted graph G and starting vertex s.
Create the open list of nodes O
Create the closed list of nodes C
If O is empty
True
Exit with failure
False
If u is a goal node
True
Exit successfully
False
Expand node u. Each successor is identified by n.
Establish a pointer to n from each of those members that were not already in G Add these members to O. For each member O or C, redirect its pointer to n if the best path so far is through n. For each member already on C, redirect the pointers of each of its descendants in G so that they point backward along the best paths found so far to these descendants.

Figure 20: Flowchart for A* Algorithm

To further enhance the understanding of the functioning of the A* algorithm, following is an example of the algorithm in action. This example shows a fictional search for Toronto starting in Vancouver. Each of the nodes represented by ovals is labeled with their heuristics. They are labeled $f = g + h$ where the h values are the fictional straight-line distances to Toronto and f is the result of the heuristic function.

The following Figure displays the initial state for the search. It represents the starting node of Vancouver.

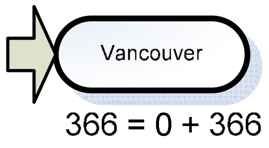


Figure 21: A* example initial state

The next figure displays the extension of Vancouver showing the next nodes with their values. Of the three resulting nodes, the shortest path to the goal passes through Calgary, represented by the arrow. This is determined by Calgary having the lowest value heuristic of 393.

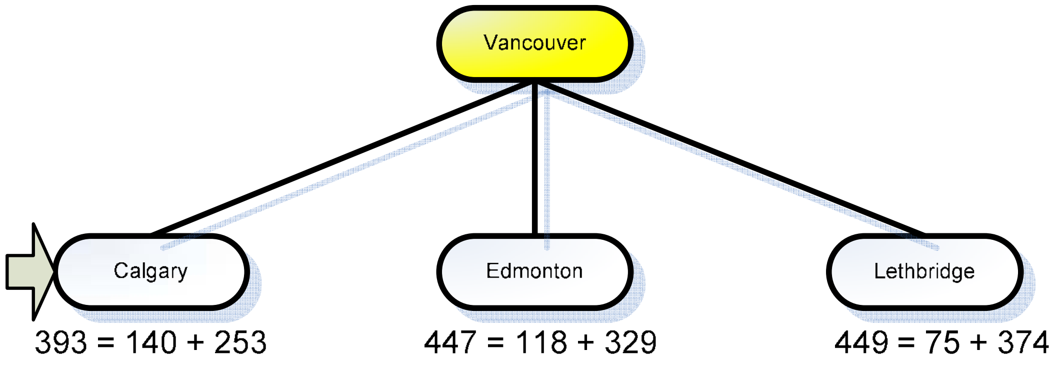


Figure 22: A* example after expanding Vancouver

The next step is to expand the Calgary node. This results in three nodes with the Saskatoon node having the lowest heuristic.

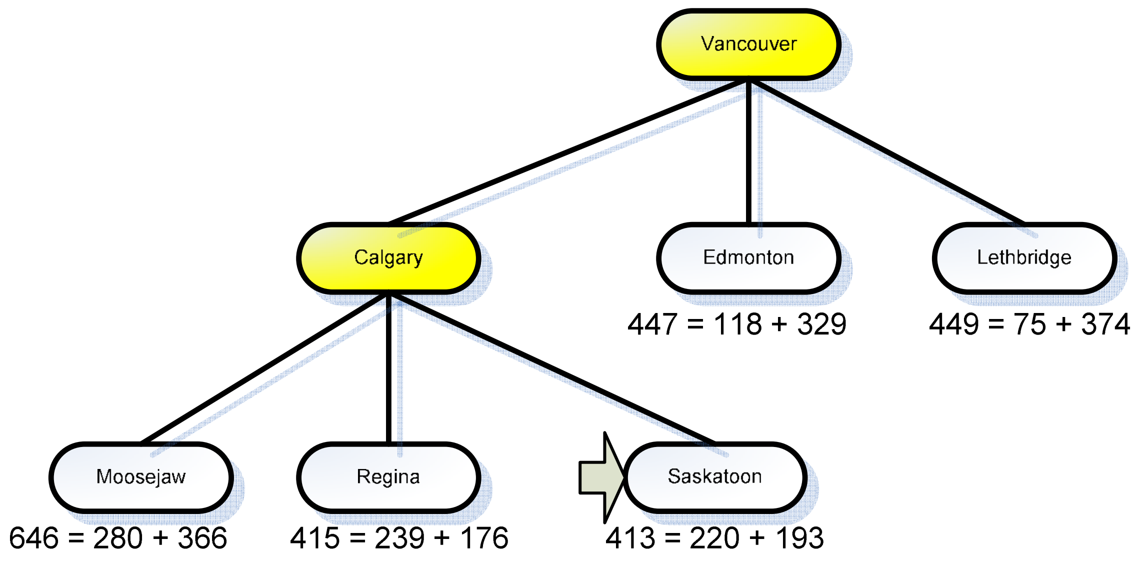


Figure 23: A* example after expanding Calgary

The next step is for the Saskatoon node to be expanded as follows.

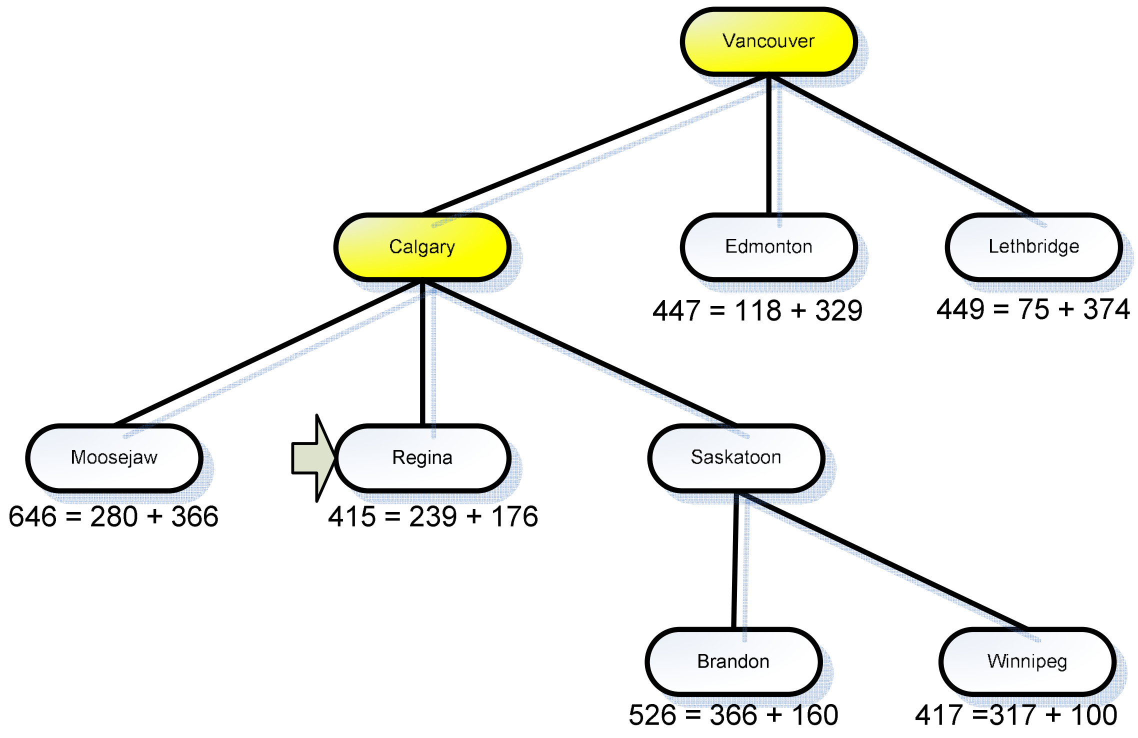


Figure 24: A* example after expanding Saskatoon

The expansion of Saskatoon results in an interesting situation. Each of the nodes expanded from Saskatoon have larger heuristics then the Reina node that was expanded from the Calgary node. This results in the next node to be expanded being the Regina node.

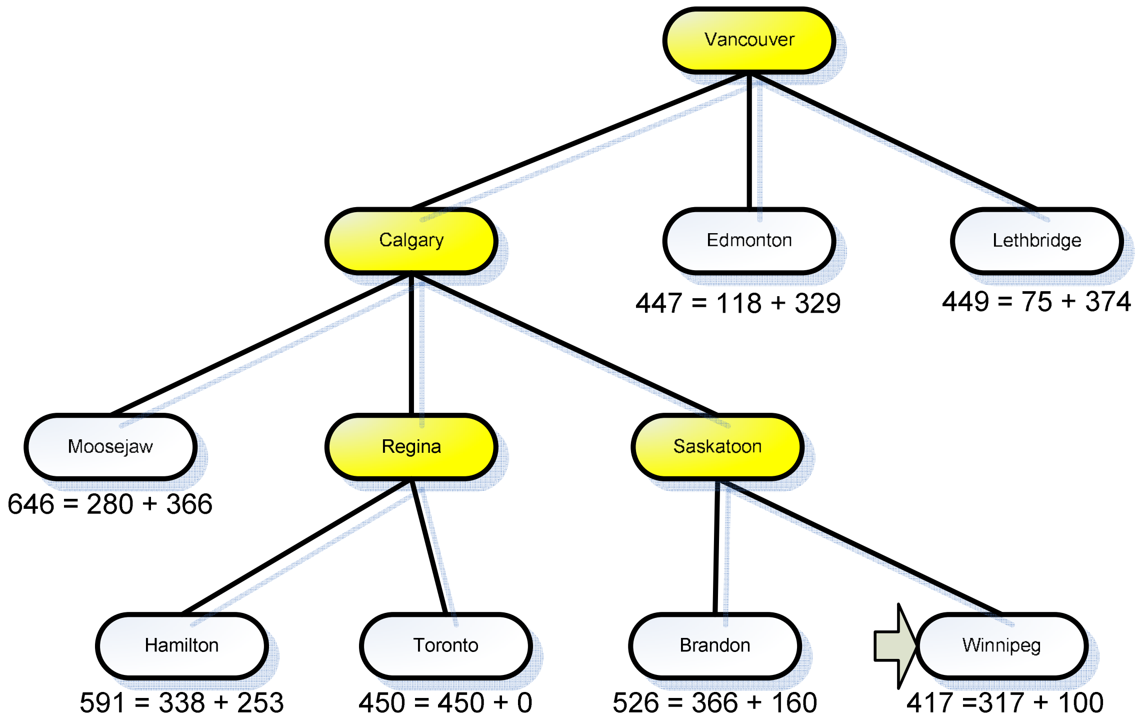


Figure 25: A* example after expanding Regina

Once the Regina node is expanded, the goal node Toronto is shown but since the heuristic is larger the Winnipeg, Winnipeg is the next node to be expanded since it has the lowest heuristic of the visible nodes.

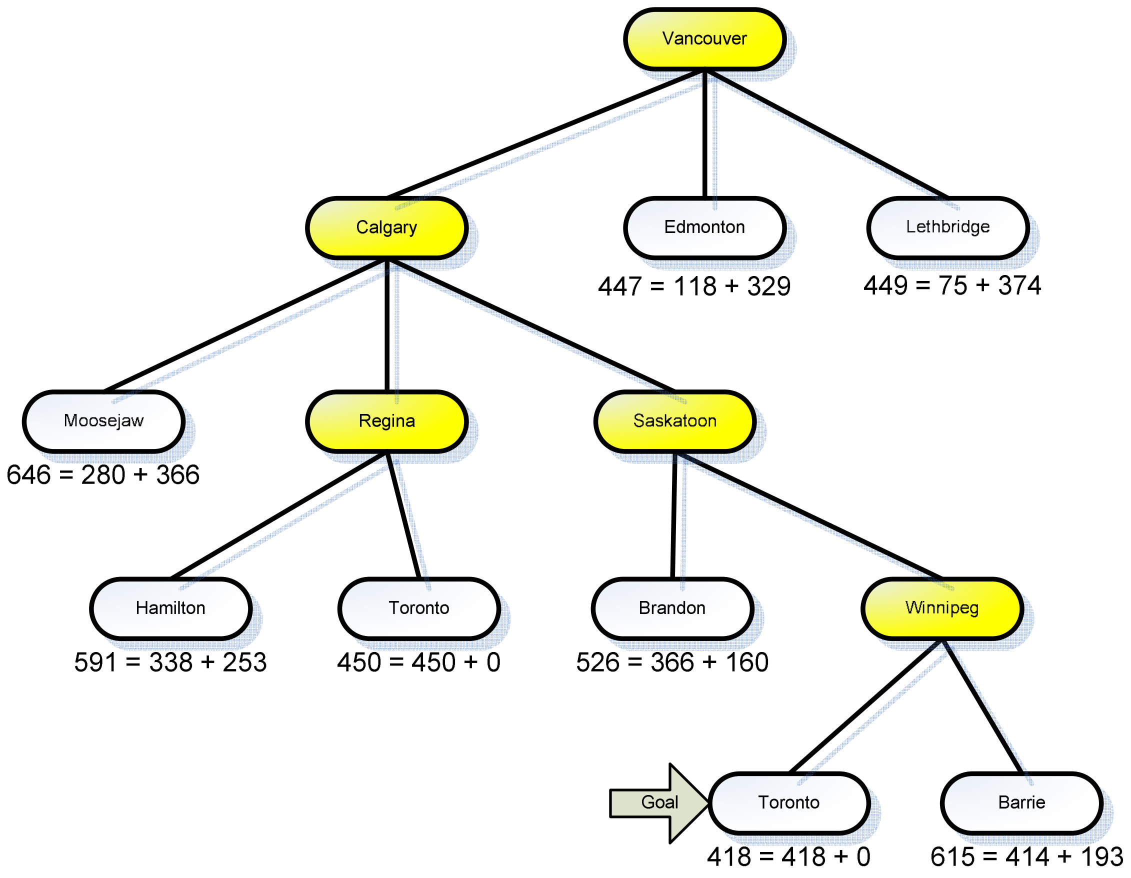


Figure 26: A* example after expanding Winnipeg. Goal reached.

The expansion of Winnipeg results in the goal node Toronto being made visible. This is the final goal node over the Toronto node exposed from the Regina node because it has a lower heuristic then the node under the Regina node.

Heuristics – Speed vs. Accuracy

As mentioned previously, a heuristic is the estimated cost of the cheapest path from the current node to the goal node [14]. The use of heuristics is one of the key elements of the A* algorithm and helps to distinguish it from many other algorithms such as Dijkstra's algorithm. The BFS algorithm uses a heuristic, in fact the A* algorithm actually belongs in the Best-first algorithm family [10, 40], but the A* algorithm expands upon the simply heuristic used for the BFS algorithm to calculate a better path to the goal.

In the A* algorithm, the heuristic is used to control of its behavior and it is important to choose a good heuristic function [37] of which there are several that we will take note of. The following section describes the most common, general purpose heuristics used with the A* algorithm and compares them in an investigation of which heuristic should be considered optimal.

When dealing with heuristics and the A* algorithm it is important to decide what is the desired result of running the algorithm as there can be several outcomes that can be the result of using different heuristics. The most important qualities that must be decided on are the speed at which the algorithm determines the shortest path and how important it is to achieve the optimal shortest path. This can be seen as tradeoff between speed and the accuracy of finding the shortest path. Both of these qualities are interrelated and a balance must be found between them when determining which is optimal based on what is considered important at the time [37].

If a heuristic is decided upon because it determines a path quickly it is very likely that its accuracy is not the best although it is admissible. On the other hand if a heuristic determines the shortest path possible then it usually is not the fastest because of the need for it to expand more nodes as it determines the shortest path to the goal node.

In regards to the accuracy of the heuristic, it is possible for the A* algorithm to find a shortest path but it may not be the shortest path. The heuristic determines what qualifies as the shortest path. To illustrate this point, the following figures show examples of using Manhattan Distance, Euclidean Distance and simple diagonal path heuristics with the A* algorithm.

Figure twenty seven shows an example of the shortest path determined using the Manhattan Distance heuristic. According to [38], Manhattan distance is the distance between two points measured along the axes at right angles. Manhattan Distance is calculated by taking the starting point p1 = (x_1, y_1) and the goal node is p2 = (x_2, y_2) and calculating [39]:

$$h(n) = |x_1 - x_2| + |y_1 - y_2|$$

Figure 26: Manhattan distance equation

As we can see from figure twenty seven, Manhattan distance finds a path to the goal but it is not the most accurate as we can easily visualize a shorter path.

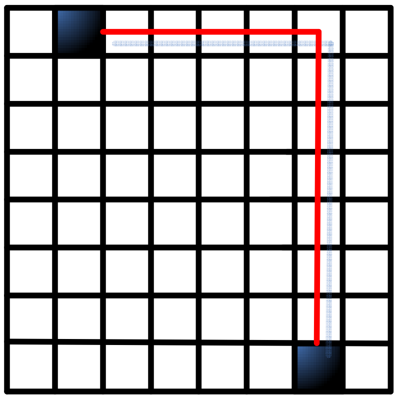


Figure 27: Manhattan distance example

This shorter path can be seen in the following figure. Figure twenty nine shows an example the shortest path determined using the Euclidean distance heuristic. According to [39] it can be defined as the straight line distance between two points. Euclidean distance determines the most direct path from the starting point to the goal without consideration of the grid that makes up the environment. It can be calculated using: the starting point p1 = (x_1, y_1) and the goal node is p2 = (x_2, y_2) and calculating [39]:

$$h(n) = \sqrt{((x_1 - x_2)^2 + (y_1 - y_2)^2)}$$

Figure 27: Euclidean distance equation

This heuristic is much more accurate then the previous example of Manhattan distance as it creates a path to the goal that is a direct line.

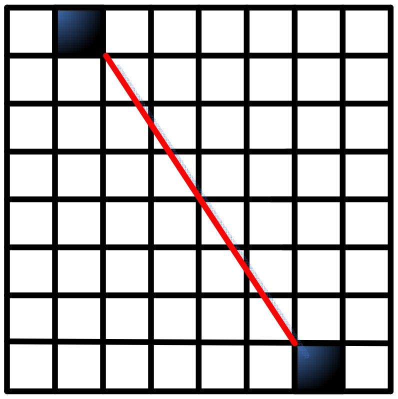


Figure 28: Euclidean distance example

The third example shows the shortest path if we were to find it using Diagonal distance which attempts to use a perfectly diagonal line, as determined by the grid. A straight line from the starting node moves towards the goal node until the diagonal line can be used. Diagonal distance combines aspects of both Manhattan and Euclidean distance heuristics [12, 37]. It takes several computational steps to determine the diagonal distance from one node to another. The equation for determining the diagonal distance is as follows: having the starting point p1 = (x_1, y_1) and the goal node is p2 = (x_2, y_2) the equations are [12]:

First determine the number of diagonal steps that can be taken:

$$d = \min\{\, |x_1 - x_2|,\ |y_1 - y_2| \,\}$$

Figure 29: Diagonal distance, determine number of diagonal steps

Then determine the number of straight steps that need to be taken:

$$s = (|x_1 - x_2| + |y_1 - y_2|) - 2 * d$$

Figure 30: Diagonal distance, Determine number of straight steps

The final step is to determine the heuristic value using:

$$h(n) = s + \sqrt{2} * d$$

Figure 31: Diagonal distance, determine heuristic value

As can be inferred from the equations to calculate Diagonal distance, this heuristic will not be the fastest heuristic since it effectively triples the amount of work that is done in the other heuristic equations.

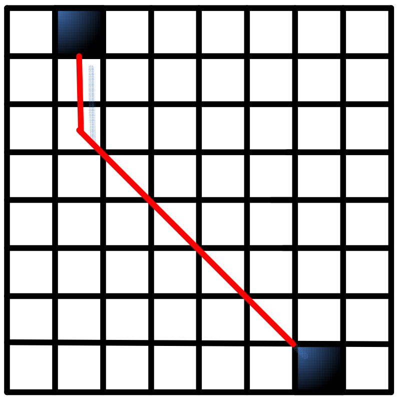


Figure 32: Shortest Path using the diagonal path

As we can see, these heuristics determine a path between two points that is considered by each to be the shortest path but each determines a different shortest path depending on what the heuristic determines it to be. So from this we can see that selection of a heuristic is very important.

As mentioned previously, the most accurate heuristic may not be the best for speed. From our examples above this is indeed the case. The heuristic that determines the optimal shortest path, determined by the least amount of steps taken from the starting node to the goal node, is determined using the Euclidean distance heuristic. The Euclidean distance heuristic usually underestimates the actual cost to the goal node by a significant amount meaning that, while it does determine an admissible, optimal shortest path, it does so by visiting a large number of nodes, some unnecessarily, which in turn causes the heuristic to take a relatively long time to determine the path [12, 37]. As well as expanding more nodes then may be necessary, the Euclidean distance heuristic is also computationally more expensive then the Manhattan distance heuristic simply because it involves two additional multiplication operations as well as the taking of the square root [12].

The Manhattan distance, sometimes called city block distance [14], is the most common heuristic [14, 37] used with the A* algorithm. Manhattan distance is preferred because it is admissible and computationally inexpensive with its only drawback being that it tends to overestimate the actual minimum cost to the goal resulting in the path being found not being the optimal solution [8, 12, 37]. For most applications this drawback is not seen as being important and can easily be overlooked.

The diagonal distance heuristic is also admissible and does find a near optimal, shortest path but due to the overhead of requiring three equations to determine the heuristic value this heuristic is extremely computationally expensive and should only be considered when speed is not a factor and diagonal movement is desired [12, 37].

There are other additional heuristics, several of which were identified in the research review chapter of this paper, but they tend to be variants of one of the three heuristics discussed above or have specific qualities that make them only valuable in certain environments.

Interaction Schemes

An additional topic that needs to be addressed in regards to pathfinding strategies is the interaction of the NPCs and agents with in the state space. This essentially deals with how the NPCs and agents respond to one another during pathfinding when there is more then one NPC trying to pathfind in the environment. There are several schemes that have been adapted to address this form of interaction. These schemes involve the NPCs working together and sharing information called cooperative, the NPCs not sharing information and not hindering each other in any way called non-cooperative and, finally, where they act do not share any information but do interfere with each other if their paths interfere with each other commonly referred to as competitive.

These three pathfinding interaction schemes do not in themselves, affect the outcome of an NPC developing a path to a goal, i.e. they do not affect the amount of memory used, but they can have an effect on the number of steps that an NPC will take to it's goal since the type of interaction scheme being used will effect how the NPCs act when they are faced with situations in which they must decide how to solve a pathfinding problem such as navigating through a small bottleneck.

With cooperative pathfinding, the NPCs can work together to identify where there will be a possible bottleneck in their paths and work to resolve the issue such that all the NPCs can reach their goals in the least amount of steps. This could involve a strategy where some of the NPCs have to find another way around the obstacle or the creation of a strategy that sees all the NPCs pass through the bottleneck in the least amount of steps and shortest possible time.

Non-cooperative pathfinding would see the NPCs not care if another NPC got in their way they would just continue as if nothing was in the way. In this interaction schema it is usually the case that the NPCs can pass through each other with out having to react to a collision with another NPC.

With competitive pathfinding, the situation exists where each NPC is only concerned with itself and getting to its goal even at the expense of other NPCs. In this interaction scheme, if as bottleneck occurs it is possible that some NPCs may not be able to reach their goal, will have to wait until the bottleneck velars or will have to find another path to the goal assuming that the bottleneck will not clear.

These schemes will be discussed further in the next section in the context of how they are implemented for testing in the test environment.

Result of Analysis

From our analysis based investigation, we can come to the conclusion that the A* algorithm is the optimal algorithm for pathfinding in grid based state spaces such as a

2.5D, isometric environment. This is reflected in the following table which summarizes the results of the previous sections in this chapter.

Table 5: Comparison of Algorithms Based on Evaluation Criteria

Comparison of Algorithms based on Evaluation Criteria			
Algorithm	Memory Usage	Relative Speed	Finds Shortest Path
Dijkstra's Algorithm	High	Slow	True
BFS	Medium	Fast	False
A* Algorithm	Medium	Fast	True

In regards to heuristics, our analysis seems to point to Manhattan distance being the best heuristic. This can be concluded by balancing speed and accuracy where Dijkstra's algorithm is accurate but can be relatively slower due to the large number of nodes that must be expanded in its search for the goal. As well, the BFS algorithm is relatively slow because it has to perform three calculations for each node that it expands. The heuristic that seems to balance our two qualities is Manhattan distance which finds a relatively accurate path without expanding too many nodes or having to perform a series of calculations.

The next chapter will take the results from the analysis of the pathfinding strategies identified in this chapter and use a test environment to test if our analysis is in fact correct.

CHAPTER 4

PATHFINDING STRATEGIES TESTING

This chapter focuses on describing the details of the testing that was performed in order to address the statement of purpose for this paper which is to investigate and design a workable pathfinding strategy in a 2.5D game world for multiple NPC and players' agents that is optimal in terms of steps taken and resources required.

In the previous chapter we looked at an analysis of the most popular algorithms, Dijkstra's, BFS and A*, and gave the results of this analysis as being that A* with the Manhattan distance heuristic is optimal. This chapter contains the results of using a 2.5D test environment to evaluate the relative strengths and weaknesses of the various algorithms under specific conditions which will be addressed in relation to the conclusions from our previous analysis.

Test Environment

The test environment consisted of a 2.5D, isometric tiling application that was easy to configure and extend. The original application was developed by Dr. Andrew Davidson and is taken from his book 'Killer Game Programming in Java'. The application itself is very basic but contains much of the necessary functionality to enable individual sprites to move over an isometric environment based on preset algorithms.

In order to use this application for testing, I first removed much of the 'game like' features that would be of no benefit to our testing. As well, I removed the coloration of environment floor to make it easier to follow the sprites as they move through the environment.

I then added an implementation of Dijkstra's algorithm and BFS algorithm. An implementation of the A* algorithm was already included in the application. I verified that this implementation of the A* algorithm was correct and decided that it would suffice for our testing needs. Additionally, for the A* algorithm, I added implementations of the Euclidean and Diagonal distance heuristics. Again the application already contains a good implementation of the Manhattan distance heuristic.

Additionally, the test application was extended to include an implementation of collision detection for the NPCs. When an NPC tries to enter a node that will contain another NPC after the next step, the collision detection functionality will cause the first NPC to add an additional step to its path. This additional step will be the same as the current step and will result in the NPC spending two steps in the same node. This essentially causes the NPC to wait one step for the node it wishes to move to be clear of other NPCs.

The goal node for each of the NPCs also uses collision detection. The collision detection for the goal node was already in place in the form of collision detection for the player character of the unmodified application. When an NPC enters the location of the player character the application signals a game over state and the application stops. When multiple NPCs are being used, all the NPCs must reach the goal node before the application signals this condition.

The movement functionality of the NPCs is as follows. The NPCs are given a priority based on the closeness of the NPC to the goal and where the NPCs are the same distance, the one closest to row one is the more significant. Each NPC, based on this priority, develops its path to the goal based on the pathfinding characteristics determined for the NPC. These characteristics include which pathfinding algorithm it will use, Dijkstra's algorithm, the BFS algorithm or the A* algorithm, which also requires that a heuristic be chosen ahead of time. Other characteristics include determining if collision detection is active and if the NPCs will work cooperatively.

Once each NPC developed its path to the goal node, each NPC in turn, based on their priority, will move one step from their path. This continues until the goal is reached by the NPC. For our testing it was decided that the goal node would not move but if this was not the case and the goal did move then the NPCs would develop a new path to the goal each time it moved.

As will be described later in this chapter, several test environments were used for our testing. Each of the test environments were identical, except for the placement of the obstacles. The obstacles for our testing are fixed and not moving which ensures that we are able to test our different pathfinding schemes evenly.

Through manipulation of the source code I was able to control the starting locations and number of the NPCs as well as the starting location for the goal node. I also was able to adjust the timer to count seconds to two decimal places for accuracy.

The time for the NPCs to reach the goal was recorded but is not the only measurement. An additional measurement is the number of steps that the NPCs take to get to the goal. This measurement is our prime measurement for comparing our algorithms because the size of the environment does not ensure that the differences in speed for calculating the algorithms will be noticeable. This will be manipulated by having several different environments, one with no obstacles and others with varying numbers and designs of obstacles. As well, during different tests, the NPCs will be set to react to collisions from other NPCs by letting them pass through or forcing the other NPC to stop and wait until the path is clear or go around the first NPC.

In our test environment, the NPCs act in certain ways due to the overall design of the test application. Firstly, the NPCs can only move in a diagonal path, north east, north west, south west and south east relative to their present position, using row number one as the top of the environment, as relative north.

As discussed previously in this paper, three interaction schemes were considered and included in our testing, these are cooperative pathfinding, non-cooperative pathfinding and competitive pathfinding.

Cooperative pathfinding was implemented by prioritizing the NPCs. The priority for the NPCs was set by considering which NPC was closer to the goal by a direct distance. For NPCs that were the same direct distance from the goal, the NPC that was on the row closest to the top of the grid was determined to be of higher priority then those that were lower. This was a heuristic and not based on any rule.

Once the NPCs were prioritized, the most significant NPC was instructed to develop its path to the goal before the other NPCs; this essentially ensured that the first NPC would develop the best path to the goal. Following the first, the second NPC developed its path but it was instructed to consider points where it would collide with the first NPC and take action to ensure that this interference did not occur. This was accomplished by adding an extra step to the path of the second NPC where a collision was going to happen. This extra step was the same as the previous step and would basically cause the second NPC to wait one step until the first NPC moved away.

The third NPC would then follow performing the same collision avoidance carried out by the second NPC but comparing itself first to the first NPC and then the second. All other NPCs would follow this process comparing their path to all other paths until they are all complete.

The cooperative interaction scheme is similar to our collision detection scheme discussed previously except for one important difference. The cooperative process happens before the NPCs start to navigate the environment, essentially done before they start, where as, collision detection happens on the fly as each NPC takes a step.

This cooperative approach ensures that no NPC interferes with another and that all NPCs involved will have the best possible route to the goal taking into consideration possible collisions and their priority.

For non-cooperative pathfinding, the implementation involved allowing the NPCs to develop their paths independently of the other NPCs and to enable them to pass through one another without any consideration of collision detection.

Competitive pathfinding was implemented by enabling collision detection but not having the NPCs develop their paths with any knowledge from the other NPCs. For competitive pathfinding during our testing, the number of collisions were counted and recorded.

Another factor that affects the test application is the software and hardware that the test application is running on. This software and hardware has a direct effect on the speed at which the test application runs. The test application was written in and run on Java version 1.6.0_11 running on Microsoft Windows XP with service pack 3. The hardware was a Toshiba laptop with a T5600 Intel Core 2 CPU running at 1.83 GHz with 3.00 GB of RAM.

Tests Overview

Our testing was performed using five test environments. A test environment is the environment that the NPCs must navigate to the goal. They are distinguished by the obstacles they contain as each test environment has a different number and placing of obstacles. Each Test environment has the same number of tests carried out on it and these tests are grouped by interaction strategy for easier reference. The different groups are the single NPC group, the cooperative group, the non-cooperative group and the competitive group. Tests within each group can be referenced by their test number. This terminology will be used throughout this chapter to reference and identify specific tests.

As mentioned in the discussion of the test environment, it is a relatively simple matter to change the settings and perform tests with different variables. As such, it was decided that we would attempt to perform as many tests as possible that have different variables that can have an effect on the overall outcome of running the test. These variables included varying the number of NPCs attempting to reach the goal, varying the interaction between the NPCs and, when applicable, testing different heuristics while testing different pathfinding algorithms.

For each test, there were three iterations conducted with an average taken at the end. The reason for the three tests was to ensure that our results for the test were consistent and did not contain any anomalies that would skew our view of the test. For each test the number of steps to the goal, the number of collisions with other NPCs if collision detection was enabled and the time it took the NPC to reach the goal were all recorded.

There were variables that did not change from test to test, these included the basic design of the environment which can be seen in figure thirty four. As can be seen from this image, the environment was a basic 2.5D isometric grid with a height of 12 cells and a width of 16 cells. The red colored cell represents the goal node which did not be change during the primary tests and only changed for the race tests that are described as test environment 5. The numbered, green colored cells represent the starting locations for the NPCs. The numbers were used for identification as well as prioritization. When only one NPC was required for a test, the number one NPC was always used. The other NPCs, numbers two and three, were only used during tests which were designed to look for differences in the three different pathfinding relationships.

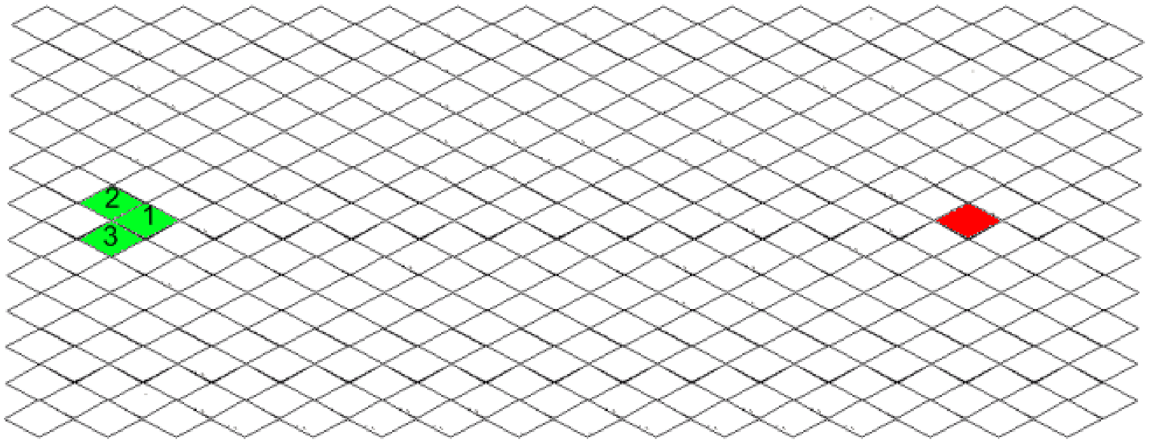


Figure 33: Test Environment 1

The basic test environment, called test environment 1 for simplicity, only enabled very simplified testing and only truly proved that all the tested algorithms were admissible when there are no obstacles for them to work around. In this environment, a relationship test was also conducted but with all the NPCs moving at the same speed there was really no true result to be taken away from this testing. For the testing of the A* algorithm, different heuristics were used with only occasionally minor differences. We believe that differences in the speed of running each of the heuristics were not seen due to the relative small size of the environment and that without any obstacles to navigate around the time to calculate a path for each of the heuristics would be very close to each other.

The second environment that was tested included some basic obstacles for the NPCs to navigate and can be seen in the following figure and will be referred to as test environment 2.

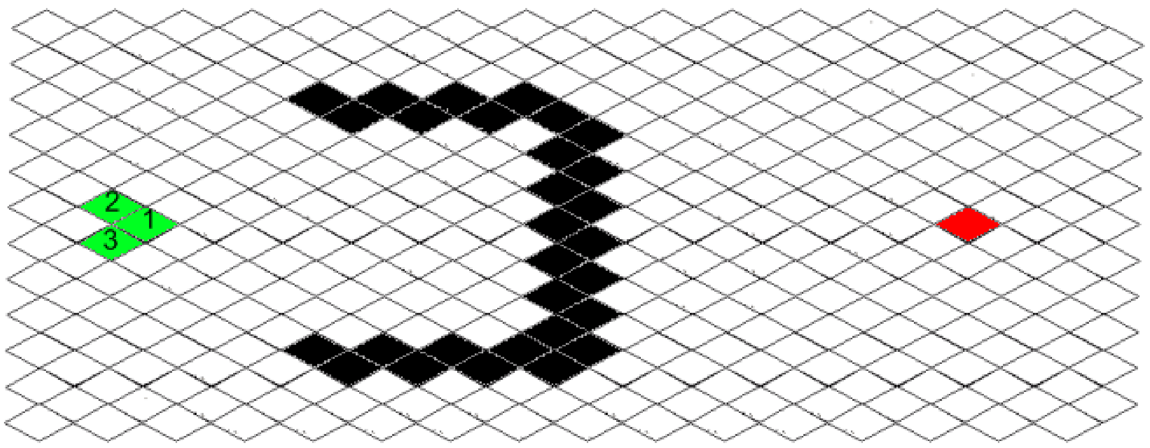


Figure 34: Test Environment 2

Test environment 2 contains a large obstacle in the center that can cause an NPC to enter it and have to add many extra steps in order to work its way out. The intention of this group of testing was to investigate which of the algorithms would not be able to navigate the obstacle without a penalty in the form of steps and time. From our previous work it was expected that the BFS algorithm would have the biggest problem navigating this type of obstacle.

Other tests that were conducted using test environment 2 included testing how the different heuristics for the A* algorithm reacted to the obstacle and how different pathfinding relationship among the NPCs would be affected by this obstacle.

The next test environment, named test environment 3, was configured as seen in the following figure.

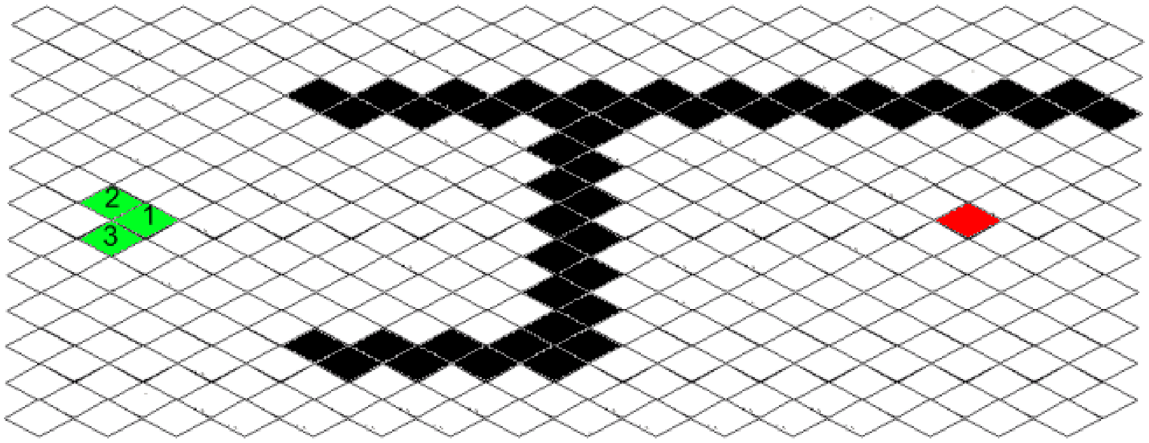


Figure 35: Test Environment 3

Test environment 3 is very similar to test environment 2 except that it extends the top arm of the obstacle to make a barrier that cannot be navigated to the goal. The testing for this test environment followed the same steps and performed the same tests as the previous test environment, test environment 2.

The next test environment, named test environment 4, was designed to test the effects of different pathfinding relationships among the NPCs, especially competitive pathfinding. Under competitive pathfinding, it would be expected that an NPC can block another NPC from moving forward or force it to find another path. Under our environment, using competitive pathfinding, it was expected that the blocked NPC would be forced to wait one step until the path clears.

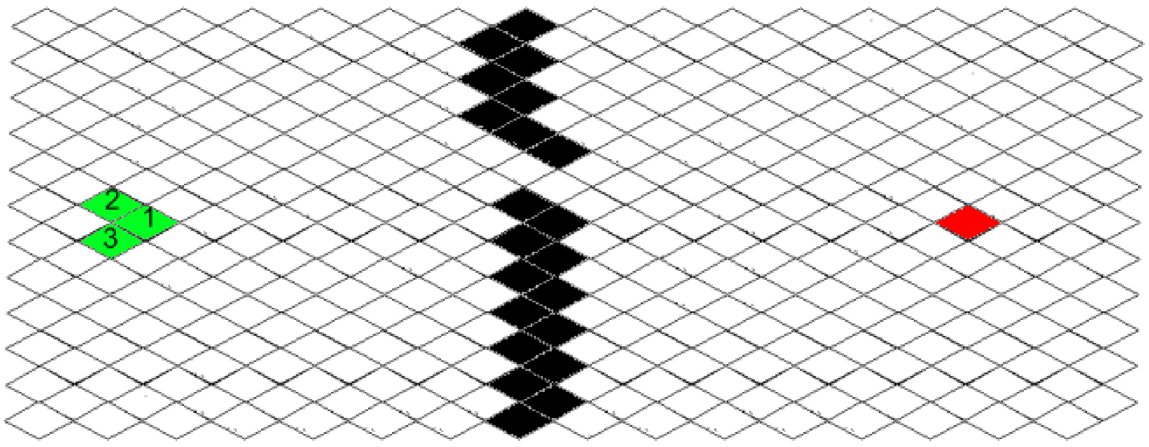


Figure 36: Test Environment 4

Test environment 4 had an obstacle that was specifically designed to force the NPCs to path find to a specific point that only one NPC at a time would be able to navigate around. This would force the other NPCs to reevaluate their pathfinding decisions if they are working cooperatively or using collision detection

The final test environment, test environment 5, can be seen in **figure**. It could be called 'The Race' because it sets the goal into the center of the environment and sets an NPC at an equal distance in each corner. Around the goal was placed different obstacles that are the same for each NPC so as not to give one an advantage over the other.

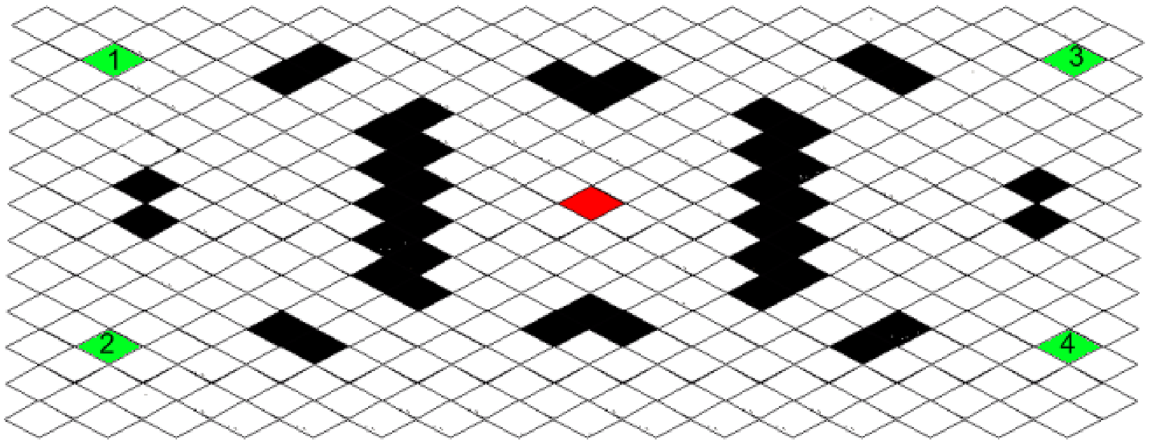


Figure 37: Test Environment 5 ('The Race')

Each of the NPCs was assigned a different algorithm to use to develop a path to the goal. The assignment of algorithms was as follows:

Table 6: Algorithm assignments 1

NPC	Algorithm
1	A* using Manhattan Distance
2	Dijkstra's
3	BFS
4	A* using Euclidean Distance

An additional test was conducted with the following NPC to algorithm assignments:

Table 7: Algorithm assignments 2

NPC	Algorithm
1	A* using Manhattan Distance
2	A* using Diagonal Distance
3	A* using Euclidean Distance
4	A* using Manhattan Distance

The reasoning behind this algorithm assignments was to determine which of the algorithms, with all things being equal, would be the fastest and which of the heuristics used by A* would be the fastest. From our previous research we had already made assumptions based only on analysis, this was the opportunity to determine if the conclusions from this analysis were correct.

Findings and Results

The following sections contain the results from the testing that was performed as well as observations and general conclusions based on the numbers presented. Special attention will be paid to results that reinforce or are contrary to our previous analysis conducted in chapter three of this paper.

The tables containing our summarized test results contain columns that show the variables that were manipulated for each test. They are, from left to right as seen in the tables below, the test number, the interaction strategy for the NPCs in the test, the algorithm being used, if applicable, the heuristic being used and the NPCs involved in the testing identified by their priority number. Following these columns are the columns that contain the metrics for the test averaged from the three testing iterations performed. These metrics include the number of steps taken by the NPC to reach the goal, the number of collisions, if applicable, and the average of the times that were recorded for that particular NPC during the testing iterations rounded up to two decimal points.

The final column, 'Test Avg Time' takes the times recorded for the NPCs in a multi NPC test and averages them. This gives an overall timing for the NPCs to reach the goal and was used for comparison between the different algorithms within the same or different pathfinding strategies but not between different test environments. The 'Test Avg Time' was used for analysis of tests within testing groups to identify if they reinforced our analysis or was contrary to it.

It is important to note that for all the tests performed, the time recorded is relative to that specific test. Due to the possibility of anomalies in the overall testing environment between tests, comparison of times between tests should be taken with a degree of inaccuracy and only generalizations should be made. Use of the times to determine exact conclusions between tests was not done. The timings for tests were taken as generalizations when doing comparisons between different tests.

There are many anomalies that could result in the times for a test to be unreliable such as the conditions of the computer during the testing. During testing all effort was taken to ensure that as few as possible processes were running on the computer that could affect the tests being run.

It was hoped that by performing three identical iterations for each test and then taking the averages would help to eliminate as many anomalies as possible.

<u>Test Environment 1</u>

In the following table are the summarized results for test environment 1. The complete results from testing can be found in Appendix A.

Table 8: Summarized results for test Environment 1

Test Environment 1					Average			
Test	Interaction Strategy	Algorithm	Heuristic	NPC	Steps Taken	Hits	Time (seconds)	Test Avg Time (seconds)
1	Single NPC	Dijkstra's	N/A	N/A	24	N/A	3.45	
2	Single NPC	BFS	N/A	N/A	24	N/A	3.01	
3	Single NPC	A*	Manhattan	N/A	24	N/A	3.13	
4	Single NPC	A*	Euclidean	N/A	24	N/A	3.28	
5	Single NPC	A*	Diagonal	N/A	24	N/A	3.20	
6	Cooperative	Dijkstra's	N/A	1	24	N/A	3.48	3.61
		Dijkstra's	N/A	2	25	N/A	3.61	
		Dijkstra's	N/A	3	26	N/A	3.75	
7	Cooperative	BFS	N/A	1	24	N/A	3.40	3.53
		BFS	N/A	2	25	N/A	3.53	
		BFS	N/A	3	26	N/A	3.66	
8	Cooperative	A*	Manhattan	1	24	N/A	3.49	3.62
		A*	Manhattan	2	25	N/A	3.62	
		A*	Manhattan	3	26	N/A	3.74	
9	Cooperative	A*	Euclidean	1	24	N/A	3.48	3.61
		A*	Euclidean	2	25	N/A	3.61	
		A*	Euclidean	3	26	N/A	3.75	
10	Cooperative	A*	Diagonal	1	24	N/A	3.47	3.60
		A*	Diagonal	2	25	N/A	3.60	
		A*	Diagonal	3	26	N/A	3.73	
11	Cooperative	A*	Manhattan	1	24	N/A	3.50	3.63
		A*	Euclidean	2	25	N/A	3.63	
		A*	Diagonal	3	26	N/A	3.76	
12	Non-Cooperative	Dijkstra's	N/A	1	24	N/A	3.44	3.52

		Dijkstra’s	N/A	2	25	N/A	3.56	
		Dijkstra’s	N/A	3	25	N/A	3.56	
13	Non-Cooperative	BFS	N/A	1	24	N/A	3.45	3.53
		BFS	N/A	2	25	N/A	3.58	
		BFS	N/A	3	25	N/A	3.58	
14	Non-Cooperative	A*	Manhattan	1	24	N/A	3.44	3.53
		A*	Manhattan	2	25	N/A	3.58	
		A*	Manhattan	3	25	N/A	3.58	
15	Non-Cooperative	A*	Euclidean	1	24	N/A	3.49	3.58
		A*	Euclidean	2	25	N/A	3.62	
		A*	Euclidean	3	25	N/A	3.62	
16	Non-Cooperative	A*	Diagonal	1	24	N/A	3.46	3.54
		A*	Diagonal	2	25	N/A	3.59	
		A*	Diagonal	3	25	N/A	3.59	
17	Non-Cooperative	A*	Manhattan	1	24	N/A	3.50	3.59
		A*	Euclidean	2	25	N/A	3.63	
		A*	Diagonal	3	25	N/A	3.63	
18	Competitive	Dijkstra’s	N/A	1	24	0	3.61	3.75
		Dijkstra’s	N/A	2	25	0	3.75	
		Dijkstra’s	N/A	3	26	1	3.90	
19	Competitive	BFS	N/A	1	24	0	3.36	3.49
		BFS	N/A	2	25	0	3.49	
		BFS	N/A	3	26	1	3.61	
20	Competitive	A*	Manhattan	1	24	0	3.47	3.61
		A*	Manhattan	2	25	0	3.60	
		A*	Manhattan	3	26	1	3.77	
21	Competitive	A*	Euclidean	1	24	0	3.41	3.54
		A*	Euclidean	2	26	1	3.67	
		A*	Euclidean	3	25	0	3.54	
22	Competitive	A*	Diagonal	1	24	0	3.49	3.62
		A*	Diagonal	2	25	0	3.62	
		A*	Diagonal	3	26	1	3.76	
23	Competitive	A*	Manhattan	1	24	0	3.54	3.67

		A*	Euclidean	2	25	0	3.67
		A*	Diagonal	3	26	1	3.81

Test environment 1 did not contain any obstacles for the NPCs to have to navigate around. As such, this group of testing can act as a baseline for the best results for an NPC to reach the goal.

As can be expected with this group of testing, with no obstacles for the NPCs to navigate around, each test ran almost identical to all the others except for the tests that involved the A* algorithm using the Diagonal distance heuristic. These tests saw the NPCs move in long diagonal lines until they reached the goal. Interestingly, all the tests, even the Diagonal distance tests, took the exact same number of steps to reach the goal.

The results for our single NPC tests led us to several conclusions. The first conclusion being that the fastest test was test two which used the BFS algorithm. This was expected as our analysis pointed to BFS being the fastest algorithm if there are no obstacles that need to be navigated around.

Not surprisingly, based on our earlier analysis, Dijkstra’s algorithm was the slowest. Our earlier analysis had told us this would be the case due to the number of nodes that Dijkstra’s algorithm has to expand in order to find a path to the goal node. Additionally, our previous analysis suggested that when using the A* algorithm, the Euclidean and Diagonal distance heuristics should be the slowest among the three heuristics used, which was indeed the case for this test environment.

Not surprisingly, the above results were also the same for the multi NPC tests except for several anomalies such as test twelve which shows the multi NPC non-cooperative test using Dijkstra’s algorithm as being the fastest of all the tests in the non-cooperative group. It is believed that further testing would see the times for this group fall into line with the other groups.

Another interesting result from this testing is that for the cooperative and competitive groups of tests the NPCs arrived at the goal one at a time based on their priorities which was the intended result of both of these pathfinding schemes.

Table 9: Summarized group timings for Test Environment 1

Interaction Strategy	Algorithm	Heuristic	Test Avg Time (seconds)	Scheme Avg (seconds)
Cooperative	Dijkstra's	N/A	3.61	3.60
Cooperative	BFS	N/A	3.53	
Cooperative	A*	Manhattan	3.62	
Cooperative	A*	Euclidean	3.61	
Cooperative	A*	Diagonal	3.60	
Cooperative	A*	Manhattan	3.63	
Non-Cooperative	Dijkstra's	N/A	3.52	3.55
Non-Cooperative	BFS	N/A	3.53	
Non-Cooperative	A*	Manhattan	3.53	
Non-Cooperative	A*	Euclidean	3.58	
Non-Cooperative	A*	Diagonal	3.54	
Non-Cooperative	A*	Manhattan	3.59	
Competitive	Dijkstra's	N/A	3.75	3.61
Competitive	BFS	N/A	3.49	
Competitive	A*	Manhattan	3.61	
Competitive	A*	Euclidean	3.54	
Competitive	A*	Diagonal	3.62	
Competitive	A*	Manhattan	3.67	

From the above table which averages the run times for the different tests and the overall runtime for the group of tests we can see that our expectations from our earlier analysis are correct. The non-cooperative interaction scheme is the fastest amongst the three that we tested.

Test Environment 2

The following table contains the summarized results for test environment 2. The complete results from testing can be found in Appendix A.

Table 10: Summarized results for test Environment 2

Test Environment 2					Average			
Test	Strategy	Algorithm	Heuristic	NPC	Steps Taken	Hits	Time (seconds)	Test Avg Time (seconds)
1	Single NPC	Dijkstra's	N/A	N/A	28	N/A	3.96	
2	Single NPC	BFS	N/A	N/A	36	N/A	4.96	
3	Single NPC	A*	Manhattan	N/A	28	N/A	3.83	
4	Single NPC	A*	Euclidean	N/A	28	N/A	3.93	
5	Single NPC	A*	Diagonal	N/A	28	N/A	3.98	
6	Cooperative	Dijkstra's	N/A	1	28	N/A	4.21	4.16
		Dijkstra's	N/A	2	27	N/A	4.07	
		Dijkstra's	N/A	3	28	N/A	4.21	
7	Cooperative	BFS	N/A	1	36	N/A	4.80	4.92
		BFS	N/A	2	37	N/A	4.92	
		BFS	N/A	3	38	N/A	5.05	
8	Cooperative	A*	Manhattan	1	28	N/A	3.93	3.89
		A*	Manhattan	2	27	N/A	3.80	
		A*	Manhattan	3	28	N/A	3.93	
9	Cooperative	A*	Euclidean	1	28	N/A	3.97	3.93
		A*	Euclidean	2	27	N/A	3.84	
		A*	Euclidean	3	28	N/A	3.97	
10	Cooperative	A*	Diagonal	1	28	N/A	4.00	3.96
		A*	Diagonal	2	27	N/A	3.87	
		A*	Diagonal	3	28	N/A	4.00	

11	Cooperative	A*	Manhattan	1	28	N/A	3.92	3.88
		A*	Euclidean	2	27	N/A	3.79	
		A*	Diagonal	3	28	N/A	3.92	
12	Non-Cooperative	Dijkstra's	N/A	1	28	N/A	4.18	4.09
		Dijkstra's	N/A	2	27	N/A	4.04	
		Dijkstra's	N/A	3	27	N/A	4.04	
13	Non-Cooperative	BFS	N/A	1	36	N/A	4.97	5.05
		BFS	N/A	2	37	N/A	5.10	
		BFS	N/A	3	37	N/A	5.10	
14	Non-Cooperative	A*	Manhattan	1	28	N/A	3.97	3.88
		A*	Manhattan	2	27	N/A	3.84	
		A*	Manhattan	3	27	N/A	3.84	
15	Non-Cooperative	A*	Euclidean	1	28	N/A	4.04	3.95
		A*	Euclidean	2	27	N/A	3.91	
		A*	Euclidean	3	27	N/A	3.91	
16	Non-Cooperative	A*	Diagonal	1	28	N/A	4.10	4.01
		A*	Diagonal	2	27	N/A	3.97	
		A*	Diagonal	3	27	N/A	3.97	
17	Non-Cooperative	A*	Manhattan	1	28	N/A	4.08	3.99
		A*	Euclidean	2	27	N/A	3.94	
		A*	Diagonal	3	27	N/A	3.94	
18	Competitive	Dijkstra's	N/A	1	29	1	4.34	4.20
		Dijkstra's	N/A	2	28	1	4.20	
		Dijkstra's	N/A	3	27	0	4.05	
19	Competitive	BFS	N/A	1	36	0	5.09	5.22
		BFS	N/A	2	37	0	5.22	
		BFS	N/A	3	38	1	5.36	
20	Competitive	A*	Manhattan	1	29	1	4.01	3.88
		A*	Manhattan	2	28	1	3.88	
		A*	Manhattan	3	27	0	3.75	
21	Competitive	A*	Euclidean	1	29	1	4.23	4.10
		A*	Euclidean	2	28	1	4.10	
		A*	Euclidean	3	27	0	3.96	

22	Competitive	A*	Diagonal	1	29	1	4.21	4.08
		A*	Diagonal	2	28	1	4.08	
		A*	Diagonal	3	27	0	3.94	
23	Competitive	A*	Manhattan	1	29	1	4.19	4.06
		A*	Euclidean	2	28	1	4.06	
		A*	Diagonal	3	27	0	3.93	

The main feature of test environment 2 is a large backwards 'C' obstacle placed in the centre of the environment. As predicted, all tests that utilized the BFS algorithm had trouble negotiating around this obstacle resulting in additional steps and time being recorded for these tests.

Similar to test environment 1 we expected that tests utilizing Dijkstra's algorithm would still be slower then other tests, except tests utilizing the BFS algorithm and this was indeed the case. The results of all the tests in all groups within this test environment came out exactly as expected from our initial analysis including what we expected from the different heuristics used for the A* algorithm.

Table 11: Summarized group timings for Test Environment 2

Strategy	Algorithm	Heuristic	Test Avg Time (seconds)	Scheme Avg (seconds)
Cooperative	Dijkstra's	N/A	4.16	4.12
Cooperative	BFS	N/A	4.92	
Cooperative	A*	Manhattan	3.89	
Cooperative	A*	Euclidean	3.93	
Cooperative	A*	Diagonal	3.96	
Cooperative	A*	Manhattan	3.88	
Non-Cooperative	Dijkstra's	N/A	4.09	4.16
Non-Cooperative	BFS	N/A	5.05	
Non-Cooperative	A*	Manhattan	3.88	

Non-Cooperative	A*	Euclidean	3.95	
Non-Cooperative	A*	Diagonal	4.01	
Non-Cooperative	A*	Manhattan	3.99	
Competitive	Dijkstra’s	N/A	4.20	4.26
Competitive	BFS	N/A	5.22	
Competitive	A*	Manhattan	3.88	
Competitive	A*	Euclidean	4.10	
Competitive	A*	Diagonal	4.08	
Competitive	A*	Manhattan	4.06	

While the overall test results were in line with our initial analysis, the overall group timings did present a different picture and were not what we expected. As can be seen in the above table, the cooperative groups of tests, on average, were as fast, actually .02 seconds faster, then the non-cooperative group of tests which were fastest overall in test environment 1.

This goes against what we expected based on our initial analysis. It is expected that with more iterations of testing the expected results would be achieved although it would require additional testing beyond this paper to verify that this is indeed the case.

Test Environment 3

The following table contains the summarized results for test environment 3. The complete results from testing can be found in Appendix A.

Table 12: Summarized results for test Environment 3

Test Environment 3					Average			
Test	Strategy	Algorithm	Heuristic	NPC	Steps Taken	Hits	Time (seconds)	Test Avg Time (seconds)
1	Single NPC	Dijkstra’s	N/A	N/A	28	N/A	3.91	
2	Single NPC	BFS	N/A	N/A	34	N/A	4.55	

3	Single NPC	A*	Manhattan	N/A	28	N/A	3.95	
4	Single NPC	A*	Euclidean	N/A	28	N/A	3.94	
5	Single NPC	A*	Diagonal	N/A	28	N/A	3.98	
6	Cooperative	Dijkstra’s	N/A	1	28	N/A	4.11	4.11
		Dijkstra’s	N/A	2	29	N/A	4.24	
		Dijkstra’s	N/A	3	27	N/A	3.98	
7	Cooperative	BFS	N/A	1	34	N/A	4.89	5.02
		BFS	N/A	2	35	N/A	5.02	
		BFS	N/A	3	36	N/A	5.16	
8	Cooperative	A*	Manhattan	1	28	N/A	4.12	4.12
		A*	Manhattan	2	29	N/A	4.26	
		A*	Manhattan	3	27	N/A	3.98	
9	Cooperative	A*	Euclidean	1	28	N/A	4.14	4.14
		A*	Euclidean	2	29	N/A	4.28	
		A*	Euclidean	3	27	N/A	4.00	
10	Cooperative	A*	Diagonal	1	28	N/A	4.12	4.12
		A*	Diagonal	2	29	N/A	4.26	
		A*	Diagonal	3	27	N/A	3.99	
11	Cooperative	A*	Manhattan	1	28	N/A	4.18	4.18
		A*	Euclidean	2	29	N/A	4.32	
		A*	Diagonal	3	27	N/A	4.05	
12	Non-Cooperative	Dijkstra’s	N/A	1	28	N/A	4.22	4.22
		Dijkstra’s	N/A	2	29	N/A	4.36	
		Dijkstra’s	N/A	3	27	N/A	4.08	
13	Non-Cooperative	BFS	N/A	1	34	N/A	4.95	5.04
		BFS	N/A	2	35	N/A	5.09	
		BFS	N/A	3	35	N/A	5.09	
14	Non-Cooperative	A*	Manhattan	1	28	N/A	4.14	4.14
		A*	Manhattan	2	29	N/A	4.28	
		A*	Manhattan	3	27	N/A	4.00	
15	Non-Cooperative	A*	Euclidean	1	28	N/A	4.21	4.21
		A*	Euclidean	2	29	N/A	4.35	
		A*	Euclidean	3	27	N/A	4.07	
16	Non-	A*	Diagonal	1	28	N/A	4.23	4.22

	Cooperative							
		A*	Diagonal	2	29	N/A	4.36	
		A*	Diagonal	3	27	N/A	4.08	
17	Non-Cooperative	A*	Manhattan	1	28	N/A	4.11	4.11
		A*	Euclidean	2	29	N/A	4.25	
		A*	Diagonal	3	27	N/A	3.98	
18	Competitive	Dijkstra's	N/A	1	28	0	4.21	4.21
		Dijkstra's	N/A	2	29	0	4.35	
		Dijkstra's	N/A	3	27	0	4.07	
19	Competitive	BFS	N/A	1	34	0	4.99	5.13
		BFS	N/A	2	35	0	5.13	
		BFS	N/A	3	36	1	5.26	
20	Competitive	A*	Manhattan	1	28	0	4.12	4.12
		A*	Manhattan	2	29	0	4.26	
		A*	Manhattan	3	27	0	3.98	
21	Competitive	A*	Euclidean	1	28	0	4.18	4.18
		A*	Euclidean	2	29	0	4.32	
		A*	Euclidean	3	27	0	4.04	
22	Competitive	A*	Diagonal	1	28	0	4.12	4.11
		A*	Diagonal	2	29	0	4.25	
		A*	Diagonal	3	27	0	3.98	
23	Competitive	A*	Manhattan	1	28	0	3.86	3.86
		A*	Euclidean	2	29	0	3.99	
		A*	Diagonal	3	27	0	3.74	

Test environment 3 was very close in obstacle layout as test environment 2 except for the addition of the continuation of the top, relative to row one, from the backwards 'C' like obstacle to the far right end of the environment. The reason behind this test environment was to see if creating an obstacle that goes nowhere along with having only a small corridor to the goal would cause any issues with the different algorithms and heuristics.

Again, as expected, all tests utilizing the BFS algorithm were the slowest and took the most number of steps. Additionally, all other tests followed what we expected from our initial analysis although, several tests utilizing Dijkstra's algorithm were lower on average then was expected but not to the point that it would contradict our earlier analysis.

Table 13: Summarized group timings for Test Environment 3

Strategy	Algorithm	Heuristic	Test Avg Time (seconds)	Scheme Avg (seconds)
Cooperative	Dijkstra's	N/A	4.11	4.28
Cooperative	BFS	N/A	5.02	
Cooperative	A*	Manhattan	4.12	
Cooperative	A*	Euclidean	4.14	
Cooperative	A*	Diagonal	4.12	
Cooperative	A*	Manhattan	4.18	
Non-Cooperative	Dijkstra's	N/A	4.22	4.32
Non-Cooperative	BFS	N/A	5.04	
Non-Cooperative	A*	Manhattan	4.14	
Non-Cooperative	A*	Euclidean	4.21	
Non-Cooperative	A*	Diagonal	4.22	
Non-Cooperative	A*	Manhattan	4.11	
Competitive	Dijkstra's	N/A	4.21	4.27
Competitive	BFS	N/A	5.13	
Competitive	A*	Manhattan	4.12	
Competitive	A*	Euclidean	4.18	
Competitive	A*	Diagonal	4.11	
Competitive	A*	Manhattan	3.86	

The timings of the three main testing groups were not what we expected. All three testing groups have almost the same average times. It is not known if additional test iterations would see these timings fall in line with what we expected but this is beyond the scope of this testing.

Test Environment 4

The following table contains the summarized results for test environment 4. The complete results from testing can be found in Appendix A.

Table 14: Summarized results for test Environment 4

Test Environment 4					Average			
Test	Strategy	Algorithm	Heuristic	NPC	Steps Taken	Hits	Time (seconds)	Test Avg Time (seconds)
1	Single NPC	Dijkstra's	N/A	N/A	24	N/A	3.44	
2	Single NPC	BFS	N/A	N/A	24	N/A	3.42	
3	Single NPC	A*	Manhattan	N/A	24	N/A	3.52	
4	Single NPC	A*	Euclidean	N/A	24	N/A	3.52	
5	Single NPC	A*	Diagonal	N/A	24	N/A	3.52	
6	Cooperative	Dijkstra's	N/A	1	24	N/A	3.66	3.80
		Dijkstra's	N/A	2	25	N/A	3.80	
		Dijkstra's	N/A	3	26	N/A	3.93	
7	Cooperative	BFS	N/A	1	24	N/A	3.59	3.73
		BFS	N/A	2	25	N/A	3.73	
		BFS	N/A	3	26	N/A	3.86	
8	Cooperative	A*	Manhattan	1	24	N/A	3.60	3.73
		A*	Manhattan	2	25	N/A	3.73	
		A*	Manhattan	3	26	N/A	3.87	
9	Cooperative	A*	Euclidean	1	24	N/A	3.61	3.74
		A*	Euclidean	2	25	N/A	3.74	
		A*	Euclidean	3	26	N/A	3.88	
10	Cooperative	A*	Diagonal	1	24	N/A	3.64	3.78
		A*	Diagonal	2	25	N/A	3.78	
		A*	Diagonal	3	26	N/A	3.92	

11	Cooperative	A*	Manhattan	1	24	N/A	3.65	3.79
		A*	Euclidean	2	25	N/A	3.79	
		A*	Diagonal	3	26	N/A	3.92	
12	Non-Cooperative	Dijkstra’s	N/A	1	24	N/A	3.69	3.79
		Dijkstra’s	N/A	2	25	N/A	3.83	
		Dijkstra’s	N/A	3	25	N/A	3.83	
13	Non-Cooperative	BFS	N/A	1	24	N/A	3.59	3.68
		BFS	N/A	2	25	N/A	3.72	
		BFS	N/A	3	25	N/A	3.72	
14	Non-Cooperative	A*	Manhattan	1	24	N/A	3.63	3.72
		A*	Manhattan	2	25	N/A	3.76	
		A*	Manhattan	3	25	N/A	3.76	
15	Non-Cooperative	A*	Euclidean	1	24	N/A	3.55	3.64
		A*	Euclidean	2	25	N/A	3.68	
		A*	Euclidean	3	25	N/A	3.68	
16	Non-Cooperative	A*	Diagonal	1	24	N/A	3.65	3.74
		A*	Diagonal	2	25	N/A	3.78	
		A*	Diagonal	3	25	N/A	3.78	
17	Non-Cooperative	A*	Manhattan	1	24	N/A	3.64	3.73
		A*	Euclidean	2	25	N/A	3.78	
		A*	Diagonal	3	25	N/A	3.78	
18	Competitive	Dijkstra’s	N/A	1	24	0	3.64	3.78
		Dijkstra’s	N/A	2	26	1	3.92	
		Dijkstra’s	N/A	3	25	0	3.78	
19	Competitive	BFS	N/A	1	24	0	3.34	3.46
		BFS	N/A	2	25	0	3.46	
		BFS	N/A	3	26	1	3.59	
20	Competitive	A*	Manhattan	1	24	0	3.46	3.59
		A*	Manhattan	2	25	0	3.59	
		A*	Manhattan	3	26	1	3.72	
21	Competitive	A*	Euclidean	1	24	0	3.54	3.68
		A*	Euclidean	2	25	0	3.67	
		A*	Euclidean	3	26	1	3.81	

22	Competitive	A*	Diagonal	1	24	0	3.60	3.73
		A*	Diagonal	2	26	1	3.86	
		A*	Diagonal	3	25	0	3.72	
23	Competitive	A*	Manhattan	1	24	0	3.56	3.69
		A*	Euclidean	2	25	0	3.69	
		A*	Diagonal	3	26	1	3.83	

The obstacles for test environment 4 created a vertical line in the middle of the environment with only a small opening in the middle for the NPCs to pass through while navigating to the goal. The primary purpose of this environment is to test instances where multiple NPCs attempt to pass through the same point at the same time.

As can be expected from this type of environment, tests utilizing the BFS algorithm were once again the fastest or almost fastest. This is because there was no obstacle designed to make these tests pursue a path that does not lead to the goal.

Additionally, all the tests for all the testing groups came out exactly as expected from our previous analysis.

Table 15: Summarized group timings for Test Environment 4

Strategy	Algorithm	Heuristic	Test Avg Time (seconds)	Scheme Avg (seconds)
Cooperative	Dijkstra's	N/A	3.80	3.76
Cooperative	BFS	N/A	3.73	
Cooperative	A*	Manhattan	3.73	
Cooperative	A*	Euclidean	3.74	
Cooperative	A*	Diagonal	3.78	
Cooperative	A*	Manhattan	3.79	
Non-Cooperative	Dijkstra's	N/A	3.79	3.72
Non-Cooperative	BFS	N/A	3.68	
Non-Cooperative	A*	Manhattan	3.72	

Non-Cooperative	A*	Euclidean	3.64	
Non-Cooperative	A*	Diagonal	3.74	
Non-Cooperative	A*	Manhattan	3.73	
Competitive	Dijkstra's	N/A	3.78	3.66
Competitive	BFS	N/A	3.46	
Competitive	A*	Manhattan	3.59	
Competitive	A*	Euclidean	3.68	
Competitive	A*	Diagonal	3.73	
Competitive	A*	Manhattan	3.69	

The testing group timings were, again, not as expected. For this test environment, the competitive interaction scheme was the fastest amongst the three schemes tested. As well, both the cooperative and non-cooperative interaction schemes were similar in their timings where it was expected that the non-cooperative interaction scheme would be the fastest of all the schemes with this test environment.

Test Environment 5

The following table contains the summarized results for test environment 5. The complete results from testing can be found in Appendix A.

Table 16: Summarized results for test Environment 5

Test Environment 5					Average	
Test	Strategy	Algorithm	Heuristic	NPC	Avg Steps	Avg Speed (seconds)
1	N/A	A*	Manhattan	1	16	2.42
	N/A	Dijkstra's	N/A	2	16	2.42
	N/A	BFS	N/A	3	16	2.42
	N/A	A*	Euclidean	4	16	2.42
2	N/A	A*	Manhattan	1	16	2.53
	N/A	A*	Euclidean	2	16	2.53

	N/A	A*	Diagonal	3	16	2.53
	N/A	A*	Manhattan	4	16	2.53

The intention of test environment 5 was to pit the different algorithms and heuristics against one another in order to determine which of them is the fastest. Unfortunately, the design of the application proved to make this round of testing moot.

The testing application was designed to iterate through the NPCs, developing their paths one at a time until they are all complete. They then proceed to implement their paths one step at a time until they reach their goal.

Basically the time to compute the paths do not have an affect on the total time for the NPCs to reach the goal as the time to compute the paths is taken as a whole not by individual NPC. It would be interesting to be able to separate out the times to develop the paths in order to see which strategy is the fastest but that is not possible with this testing application.

Fortunately, this effect does not affect any of the other tests. The individual tests will show the time to compute the path and implement it for a single strategy and the tests utilizing multiple NPCs would include the time taken to calculate and implement the paths for one strategy only so comparison of these times is possible.

CHAPTER 5

CONCLUSIONS AND RECOMMENDATIONS

There were several conclusions that can be taken from the testing that was conducted and discussed in the previous chapter. The first major outcome was that the results of the tests contained in each testing group conformed almost exactly for all the different test environments to the conclusions of the analysis performed in chapter three of this paper. This means that the different tests that were conducted within each test group performed as expected with only a few variations that can be explained by anomalies in the test environment.

For the majority of tests, the outcomes were as expected for the type of test environment being used, which means that the results were predictable based on the type of obstacles placed in the test environment. The slowest algorithm on average was Dijkstra's algorithm except for those tests where there was an obstacle that interfered with the BFS algorithms ability to develop a direct path to the goal. In these cases, the tests that utilized the BFS algorithm were the slowest. On average, all A* tests were faster then the equivalent test using a different algorithm in the same test group. As well, the heuristics used in conjunction with the A* algorithm preformed as expected with only minor variations.

The Manhattan distance heuristic was generally the fastest followed by Euclidean distance, which was expected to be slower due to the number of nodes it needs to expand in order to develop a path, and finally, the Diagonal distance heuristic was usually the slowest of the three heuristics. This was expected as the Diagonal distance heuristic performs three calculations per node which creates significant overhead.

While the results of the tests within the testing groups conformed to what was expected based on the initial analysis conducted on the algorithms and the various options, the overall average times for each of the test groups that were compared to each other did not conform to what was expected. It was expected that the non-cooperative interaction scheme tests would be the fastest followed by the cooperative and competitive interaction schemes tests. The reasoning behind this was that with non-cooperative interaction there was no additional processing required when this algorithm was used to develop a path to the goal or during the navigation to the goal. As well, the NPCs for the non-cooperative pathfinding tests did not engage in any form of collision detection or avoidance and could actually pass through each other removing the possibility of the other NPCs acting as obstacles to be navigated around.

The cooperative and competitive interaction schemes contain additional processing that, it was thought, would make them slower then the non-cooperative scheme but this did not seem to be the case. This conclusion will be explored further in the following sections.

Implications

There are several implications of these results in regards to the statement of purpose for this paper. The main purpose of this paper was to investigate and design a workable pathfinding strategy in a 2.5D game world for multiple NPCs that is optimal in term of the number of steps taken by the NPCs to the goal and in the resources required.

The testing that was carried out included several different tests using various pathfinding algorithms and interaction schemes and either a single NPC or multiple NPCs. These tests were designed to determine which pathfinding algorithm was optimal in light of the number of steps taken and the resources used, which for our testing was represented by the time required for the NPC to reach the goal.

The single NPC tests gave a baseline for each of the pathfinding algorithms and showed the fewest amount of steps that were needed for the NPC to reach the goal as well as the least amount of time needed by the algorithm to develop the path to the goal and implement it. For each of the multi NPC tests there were at least one NPC that matched the number of steps to the goal as was seen in the single NPC tests. As well, if the test involved using collision detection, which essentially implements the competitive interaction scheme, the NPCs would reach the goal one step ahead of each other suggesting that the collision detection actually forced priority ordering on the NPCs. This affect was also seen when the NPC interaction scheme was cooperative which pre-calculates the paths for all the NPCs prior to their navigating to the goal.

Unlike the cooperative and competitive interaction schemes, the non-cooperative scheme would have the NPCs arrive at the goal in the same order they started in. For our testing, this meant that the number one NPC would arrive first followed by the number two and three NPCs arriving at the same time and usually from the same node meaning that the number two and three nodes paths usually became the same after the few first initial steps. This was made possible because of the absence of collision detection and avoidance.

Another interesting point to make regarding the BFS algorithm and the number of steps taken for the NPC to reach the goal is that the BFS algorithm performance depended greatly on the layout of the obstacles in the test environment. This algorithm was very susceptible to being 'fooled' by obstacles into following paths that did not lead to the goal but if there were no such obstacles, this algorithm usually performed the best. The reason for this lies in the BFS algorithm being a greedy algorithm that always follows the path that appears to be optimal at each step of developing the path with the hope that this will lead to the globally optimal path to goal [41].

Along with the BFS algorithm, the testing results seem to show that all the algorithms were impacted by the design of the obstacles in the testing environment but not to the extent that the BFS algorithm was affected. A look at the results shows that for all testing groups, the overall times for the tests are faster for test environments one and four and longer for test environments two and three. This points to the conclusion that all the algorithms incurred a slight penalty developing a path around the obstacles in test environments two and three which were the backwards ‘C’ shaped obstacles were as test environments one and four had either no obstacles or no obstacles that would not cause a serious problem and penalty for any of the algorithms to develop a path to the goal.

In order to determine the overall best pathfinding algorithm, the following table was developed which summarizes the test results data. This table applies a value to the relative finishing position of the algorithm when compared to the other tests in its testing group. Tests number eleven, seventeen and twenty three were eliminated since these tests comprised three different heuristics. In the table, the lower the number the faster the test in comparison to the other tests and ties are scored by the tied items receiving the same score and the next item receiving a score the same as if there was not a tie before it.

Table 17: Comparison of running times

Test Environment	Test Group	Dijkstra's	BFS	A* Manhattan	A* Euclidean	A* Diagonal
1	Single NPC	5	1	2	4	3
1	Cooperative	3	1	5	3	2
1	Non-Cooperative	1	2	2	5	4
1	Competitive	5	1	3	2	4

2	Single NPC	3	5	1	2	4
2	Cooperative	4	5	1	2	3
2	Non-Cooperative	4	5	1	2	3
2	Competitive	4	5	1	2	3
3	Single NPC	1	5	3	2	4
3	Cooperative	3	5	1	4	1
3	Non-Cooperative	3	5	1	2	3
3	Competitive	4	5	1	3	2
4	Single NPC	2	1	3	3	3
4	Cooperative	5	1	1	3	4
4	Non-Cooperative	5	2	3	1	4
4	Competitive	5	1	2	3	4
	Total	57	50	31	43	51

This table reinforces our conclusions from our analysis discussed in chapter three. It essentially breaks down the results from our testing and gives us a view of which algorithm out of all the algorithms performs the best in most circumstances and test environments.

From our testing results and analysis we can conclude that the A* algorithm using the Manhattan distance heuristic is the best algorithm for most situations and varying environments with fixed obstacles. In fact overall, the A* algorithm seems to perform better then the other algorithms used for this comparison which was the conclusion of our previous analysis.

Table 18: Comparison of interaction schemes

Test Environment	Cooperative	Non-Cooperative	Competitive
1	2	1	3
2	1	2	3
3	2	3	1
4	3	2	1
Total	8	8	8

The above table summarizes the various interaction schemes that were considered during our testing. Each interaction scheme was assigned a value based on the averaged time that the algorithms ran using these schemes. The lower the number, the lower the time taken for the tests to run.

From this table we can see that none of the interaction schemes performed better then the others. This goes against our analysis which suggested that the non-cooperative interaction scheme would perform better then the others because it does not contain the additional calculations used by the cooperative and competitive schemes.

There is no apparent reason why the non-cooperative interaction scheme did not perform better then the other schemes. Additional testing would have to be conducted in order to address this conclusion which seems to go against our analysis.

Future Research

Pathfinding is a large and complex topic with many different facets and, as such, it is impossible to address all possible research directions in one paper. As a result, there is plenty of room for additional future research that can build upon the work of this paper. Some of the possible research directions are discussed here.

This paper dealt exclusively with a fixed goal, fixed obstacle test environment. Further research could look into the performance of the various pathfinding strategies when dealing with a moving goal and/or moving obstacles. This would force the NPCs to constantly reevaluate their paths and develop new paths on the fly.

A further extension to the research performed and documented in this paper would include performing the tests using a larger test environment. This would enable true resource testing where the amount of memory could be monitored if the test environment was large enough. With the resources available to modern computers, it would take an exceptionally large environment to push the memory availability of a computer but it is a possibility and not overly difficult to develop.

Additional testing could adapt the test application to have the NPCs function as true autonomous agents so they could be compared to each other in the same test environment. This would allow for more exact testing where the times recorded from the tests can be compared to each other in the same test directly and exactly without having to consider that the times are generalizations. This would be the best method for truly determining the optimal pathfinding strategy.

The research and in depth analysis conducted for this paper only considered the most popular and widely discussed pathfinding algorithms and interaction schemes which make up a pathfinding strategy. As such there is a lot of additional research that could be done into other algorithms, additional heuristics and additional interaction schemes. Some of the possible research extensions were touched on in the research review section of this paper and can act as a good starting point.

Conclusion

The purpose of this paper was to determine which pathfinding strategy was best under most circumstances in a 2.5D game world. This paper used analysis and testing in an attempt to satisfy this goal with the result being that the A* algorithm using the Manhattan distance heuristic came out as the best algorithm and heuristic combination. Unfortunately, we were not able to effectively make a decision of which interaction scheme was better overall. As such, our recommended pathfinding strategy would, which is a combination of an algorithm and interaction scheme, would have to be the aforementioned A* algorithm utilizing Manhattan distance with any of the three interaction schemes since they performed equally well.

REFERENCES

[1] E. W. Dijkstra, "A note on two problems in connexion with graphs," *Numerische Mathematik*, vol. 1, no. 1, pp. 269-271, December 1959. [Online]. Available: http://dx.doi.org/10.1007/BF01386390

[2] Wikipedia contributors. Dijkstra's algorithm [Internet]. Wikipedia, The Free Encyclopedia; 2009 Feb 5, 06:56 UTC [cited 2009 Feb11]. Available from: http://en.wikipedia.org/w/index.php?title=Dijkstra%27s_algorithm&oldid=268643812.

[3] Paul E. Black, "Dijkstra's algorithm", in Dictionary of Algorithms and Data Structures [online], Paul E. Black, ed., U.S. National Institute of Standards and Technology. 20 September 2006. (accessed 21 Dec 2008) Available from: http://www.itl.nist.gov/div897/sqg/dads/HTML/dijkstraalgo.html

[4] Paul E. Black, "vertex", in Dictionary of Algorithms and Data Structures [online], Paul E. Black, ed., U.S. National Institute of Standards and Technology. 19 April 2004. (accessed 21 December 2008) Available from: http://www.nist.gov/dads/HTML/vertex.html

[5] J. Morris, "Data Structures and Algorithms: Dijkstra's Algorithm"; [online] 1998, [cited 21 December 2008] Available from: http://www.cs.auckland.ac.nz/software/AlgAnim/dijkstra.html

[6] P. E. Hart, N. J. Nilsson, and B. Raphael, "A formal basis for the heuristic determination of minimum cost paths," Systems Science and Cybernetics, IEEE Transactions on, vol. 4, no. 2, pp. 100-107, 1968. [Online]. Available: http://dx.doi.org/10.1109/TSSC.1968.300136

[7] Paul E. Black, "best-first search", in Dictionary of Algorithms and Data Structures [online], Paul E. Black, ed., U.S. National Institute of Standards and Technology. 18 April 2005. (accessed 21 December 2008) Available from: http://www.nist.gov/dads/HTML/bestfirst.html

[8] Amit Patel, "Introduction" [online] 24 Feb 2008, [cited 21 Dec 2008]. Available from: http://theory.stanford.edu/~amitp/GameProgramming/AStarComparison.html

[9] Paul E. Black, "heuristic", in Dictionary of Algorithms and Data Structures [online], Paul E. Black, ed., U.S. National Institute of Standards and Technology. 27 February 2004. (accessed 21 December 2008) Available from: http://www.nist.gov/dads/HTML/heuristic.html

[10] Wikipedia contributors. Best-first search [Internet]. Wikipedia, The Free Encyclopedia; 2009 Feb 3, 10:37 UTC [cited 2009 Feb 11]. Available from: http://en.wikipedia.org/w/index.php?title=Best-first_search&oldid=268226586.

[11] James Matthews, "Basic A* Pathfinding Made Simple" in AI Game Programming Wisdom, S. Rabin, Ed., Rockland, MA: Charles River Media, Inc, 2002. pp. 105-113

[12] D.R. Wichmann and B.C. Wuensche, "Automated Route Finding on Digital Terrains" Proceedings of IVCNZ '04, Akaroa, New Zealand, 21-23 November 2004, pp. 107-112.

[13] Dan Higgins, "How to Achieve Lightning-Fast A*" in AI Game Programming Wisdom, S. Rabin, Ed., Rockland, MA: Charles River Media, Inc, 2002. pp. 133-145

[14] S. Russell and P. Norvig, Artificial Intelligence: A Modern Approach, Englewood Cliffs, New Jersey: Prentice Hall Series in Artificial Intelligence, 1995.

[15] A. Reinefeld and T.A. Marsland, "Enhanced Iterative-Deepening Search", IEEE Transactions on Pattern Analysis and Machine Intelligence, vol. 16, issue 7, July 1994, pp. 701 -710.

[16] R.E. Korf, "Space Efficient Search Algorithms", ACM Computing Surveys (CSUR), vol. 27, issue 3, Sept. 1995, pp 337 – 339.

[17] R.E. Korf, "Depth First Iterative Deepening: An optimal Admissible Tree Search", Artificial Intelligence vol. 27, 1985, pp. 97-109.

[18] Timothy Cain, "Practical Optimizations for A* Path Generation", in AI Game Programming Wisdom, S. Rabin, Ed., Rockland, MA: Charles River Media, Inc, 2002. pp. 145 – 152.

[19] A. Botea, M. Müller, and J. Schaeffer, "Near optimal hierarchical path-finding," Journal of Game Development, vol. 1, no. 1, pp. 7-28, 2004.

[20] N. Sturtevant, "Memory-Efficient Abstractions for Pathfinding", Proceedings of the 3rd Artificial Intelligence and Interactive Digital Entertainment Conference, June 2007, pp. 31-36

[21] Koenig, S and Maxim Likhachev. "Real-time adaptive A*". Proceedings of the fifth international joint conference on Autonomous agents and multiagent systems, 2006, pp. 281 – 288

[22] D. Silver,"Cooperative pathfinding". In Proceedings of the First Artificial Intelligence and Interactive Digital Entertainment conference, 2005, pp.117 - 122.

[23] Renee Jansen and Nathan Sturtevant, “A new approach to cooperative pathfinding (Short Paper)” , Proceedings of 7th International. Conference on Autonomous Agents and Multiagent Systems Padgham, Parkes, Müller and Parsons (eds.), May, 12-16., 2008, Estoril, Portugal, pp. 1401-1404.

[24] Cazenave, T. “Optimizations of data structures, heuristics and algorithms for path-finding on maps”, 2006 IEEE Symposium on Computational Intelligence and Games, May 2006 pp. 27 – 33

[25] S. Tanimoto, The Elements of artificial Intelligence An Introduction Using LISP, Rockland, Maryland, Computer Science Press, Inc., 1987.

[26] Wikipedia contributors. Artificial intelligence [Internet]. Wikipedia, The Free Encyclopedia; 2009 Feb 11, 14:55 UTC [cited 2009 Feb 11]. Available from: http://en.wikipedia.org/w/index.php?title=Artificial_intelligence&oldid=2700079 87.

[27] S.M. LaValle, Planning Algorithms, Cambridge University Press, 2006

[28] Wikipedia contributors. Isometric projection [Internet]. Wikipedia, The Free Encyclopedia; 2009 Feb 10, 22:57 UTC [cited 2009 Feb 11]. Available from: http://en.wikipedia.org/w/index.php?title=Isometric_projection&oldid=26987419 0.

[29] Wikipedia contributors. 2.5D [Internet]. Wikipedia, The Free Encyclopedia; 2009 Feb 6, 13:43 UTC [cited 2009 Feb 11]. Available from: http://en.wikipedia.org/w/index.php?title=2.5D&oldid=268912869.

[30] Wikipedia contributors. Turing test [Internet]. Wikipedia, The Free Encyclopedia; 2009 Feb 11, 17:54 UTC [cited 2009 Feb 11]. Available from: http://en.wikipedia.org/w/index.php?title=Turing_test&oldid=270040193.

[31] Mike Mika and Chris Charla, “Simple Cheap Pathfinding”, in AI Game Programming Wisdom, S. Rabin, Ed., Rockland, MA: Charles River Media, Inc, 2002. pp. 155 – 160.

[32] Wikipedia contributors. Greedy algorithm [Internet]. Wikipedia, The Free Encyclopedia; 2009 Jan 22, 11:30 UTC [cited 2009 Feb 11]. Available from: http://en.wikipedia.org/w/index.php?title=Greedy_algorithm&oldid=265681873.

[33] M.T.Goodrich and R Tamassia, Data Structures And Algorithms in Java, 3rd Edition, John Wiley & Sons, Inc., 2004.

[34] “Isometric Perspective”, [online] [cited 18 Jan 2009] Available from: http://www.kirupa.com/developer/isometric/perspective.htm

[35] Wikipedia contributors. Graph (mathematics) [Internet]. Wikipedia, The Free Encyclopedia; 2009 Jan 28, 22:05 UTC [cited 2009 Feb 11]. Available from: http://en.wikipedia.org/w/index.php?title=Graph_(mathematics)&oldid=267060783.

[36] Wikipedia contributors. State space search [Internet]. Wikipedia, The Free Encyclopedia; 2008 Dec 17, 15:05 UTC [cited 2009 Feb 11]. Available from: http://en.wikipedia.org/w/index.php?title=State_space_search&oldid=258583373.

[37] Amit Patel, "Heuristics" [online] 24 Feb 2008, [cited 18 Jan 2009]. Available from: http://theory.stanford.edu/~amitp/GameProgramming/Heuristics.html

[38] Paul E. Black, "Manhattan distance", in Dictionary of Algorithms and Data Structures [online], Paul E. Black, ed., U.S. National Institute of Standards and Technology. 31 May 2006. (accessed 18 Jan 2009) Available from: http://www.nist.gov/dads/HTML/manhattanDistance.html

[39] Paul E. Black, "Euclidean distance", in Dictionary of Algorithms and Data Structures [online], Paul E. Black, ed., U.S. National Institute of Standards and Technology. 17 December 2004. (accessed 18 Jan 2009) Available from: http://www.nist.gov/dads/HTML/euclidndstnc.html

[40] Wikipedia contributors. A* search algorithm [Internet]. Wikipedia, The Free Encyclopedia; 2009 Jan 27, 14:04 UTC [cited 2009 Feb 11]. Available from: http://en.wikipedia.org/w/index.php?title=A*_search_algorithm&oldid=266743815.

[41] Wikipedia contributors. Greedy algorithm [Internet]. Wikipedia, The Free Encyclopedia; 2009 Jan 22, 11:30 UTC [cited 2009 Feb 11]. Available from: http://en.wikipedia.org/w/index.php?title=Greedy_algorithm&oldid=265681873.

[42] Rashid Bin Muhammad, "Dijkstra's Algorithm", [online], (accessed 18 Feb 2009) Available from: http://www.ecf.utoronto.ca/~writing/bbieee-help.html

APPENDIX A

TESTING RESULTS

Table 19: Test results for Test Environment 1

Test Environment 1					Test 1			Test 2			Test 3			Average		
Test	Interaction Strategy	Algorithm	Heuristic	NPC	Steps	Hits	Time (secs)	Steps	Hits	Time (secs)	Steps	Hits	Time (secs)	Steps	Hits	Time (secs)
1	Single NPC	Dijkstra's	N/A	N/A	24	0	3.40	24	0	3.39	24	0	3.56	24	N/A	3.45
2	Single NPC	BFS	N/A	N/A	24	0	3.06	24	0	2.99	24	0	2.98	24	N/A	3.01
3	Single NPC	A*	Manhattan	N/A	24	0	2.99	24	0	3.21	24	0	3.19	24	N/A	3.13
4	Single NPC	A*	Euclidean	N/A	24	0	3.18	24	0	3.32	24	0	3.34	24	N/A	3.28
5	Single NPC	A*	Diagonal	N/A	24	0	3.18	24	0	3.16	24	0	3.27	24	N/A	3.20
6	Cooperative	Dijkstra's	N/A	1	24	0	3.22	24	0	3.59	24	0	3.62	24	N/A	3.48
		Dijkstra's	N/A	2	25	0	3.34	25	0	3.73	25	0	3.76	25	N/A	3.61
		Dijkstra's	N/A	3	26	0	3.47	26	0	3.88	26	0	3.89	26	N/A	3.75
7	Cooperative	BFS	N/A	1	24	0	3.40	24	0	3.38	24	0	3.43	24	N/A	3.40
		BFS	N/A	2	25	0	3.53	25	0	3.51	25	0	3.56	25	N/A	3.53
		BFS	N/A	3	26	0	3.66	26	0	3.64	26	0	3.69	26	N/A	3.66
8	Cooperative	A*	Manhattan	1	24	0	3.38	24	0	3.53	24	0	3.55	24	N/A	3.49
		A*	Manhattan	2	25	0	3.51	25	0	3.66	25	0	3.68	25	N/A	3.62
		A*	Manhattan	3	26	0	3.63	26	0	3.79	26	0	3.81	26	N/A	3.74
9	Cooperative	A*	Euclidean	1	24	0	3.41	24	0	3.51	24	0	3.52	24	N/A	3.48
		A*	Euclidean	2	25	0	3.54	25	0	3.64	25	0	3.66	25	N/A	3.61

		A*	Euclidean	3	26	0	3.67	26	0	3.78	26	0	3.79	26	N/A	3.75
10	Cooperative	A*	Diagonal	1	24	0	3.49	24	0	3.40	24	0	3.51	24	N/A	3.47
		A*	Diagonal	2	25	0	3.62	25	0	3.53	25	0	3.65	25	N/A	3.60
		A*	Diagonal	3	26	0	3.75	26	0	3.66	26	0	3.78	26	N/A	3.73
11	Cooperative	A*	Manhattan	1	24	0	3.44	24	0	3.54	24	0	3.51	24	N/A	3.50
		A*	Euclidean	2	25	0	3.57	25	0	3.67	25	0	3.64	25	N/A	3.63
		A*	Diagonal	3	26	0	3.72	26	0	3.80	26	0	3.77	26	N/A	3.76
12	Non-Cooperative	Dijkstra’s	N/A	1	24	0	3.22	24	0	3.49	24	0	3.60	24	N/A	3.44
		Dijkstra’s	N/A	2	25	0	3.33	25	0	3.62	25	0	3.74	25	N/A	3.56
		Dijkstra’s	N/A	3	25	0	3.33	25	0	3.62	25	0	3.74	25	N/A	3.56
13	Non-Cooperative	BFS	N/A	1	24	0	3.40	24	0	3.44	24	0	3.50	24	N/A	3.45
		BFS	N/A	2	25	0	3.53	25	0	3.57	25	0	3.63	25	N/A	3.58
		BFS	N/A	3	25	0	3.53	25	0	3.57	25	0	3.63	25	N/A	3.58
14	Non-Cooperative	A*	Manhattan	1	24	0	3.29	24	0	3.48	24	0	3.56	24	N/A	3.44
		A*	Manhattan	2	25	0	3.42	25	0	3.62	25	0	3.70	25	N/A	3.58
		A*	Manhattan	3	25	0	3.42	25	0	3.62	25	0	3.70	25	N/A	3.58
15	Non-Cooperative	A*	Euclidean	1	24	0	3.40	24	0	3.55	24	0	3.52	24	N/A	3.49
		A*	Euclidean	2	25	0	3.53	25	0	3.68	25	0	3.66	25	N/A	3.62
		A*	Euclidean	3	25	0	3.53	25	0	3.68	25	0	3.66	25	N/A	3.62
16	Non-Cooperative	A*	Diagonal	1	24	0	3.51	24	0	3.54	24	0	3.32	24	N/A	3.46
		A*	Diagonal	2	25	0	3.64	25	0	3.67	25	0	3.45	25	N/A	3.59

		A*	Diagonal	3	25	0	3.64	25	0	3.67	25	0	3.45	25	N/A	3.59
17	Non-Cooperative	A*	Manhattan	1	24	0	3.41	24	0	3.53	24	0	3.55	24	N/A	3.50
		A*	Euclidean	2	25	0	3.54	25	0	3.67	25	0	3.68	25	N/A	3.63
		A*	Diagonal	3	25	0	3.54	25	0	3.67	25	0	3.68	25	N/A	3.63
18	Competitive	Dijkstra's	N/A	1	24	0	3.52	24	0	3.61	24	0	3.69	24	0	3.61
		Dijkstra's	N/A	2	25	0	3.69	25	0	3.75	25	0	3.82	25	0	3.75
		Dijkstra's	N/A	3	26	1	3.86	26	1	3.89	26	1	3.96	26	1	3.90
19	Competitive	BFS	N/A	1	24	0	3.32	24	0	3.47	24	0	3.29	24	0	3.36
		BFS	N/A	2	25	0	3.45	25	0	3.60	25	0	3.41	25	0	3.49
		BFS	N/A	3	26	1	3.57	26	1	3.73	26	1	3.53	26	1	3.61
20	Competitive	A*	Manhattan	1	24	0	3.26	24	0	3.56	24	0	3.60	24	0	3.47
		A*	Manhattan	2	25	0	3.38	25	0	3.69	25	0	3.73	25	0	3.60
		A*	Manhattan	3	26	1	3.51	26	1	3.92	26	1	3.87	26	1	3.77
21	Competitive	A*	Euclidean	1	24	0	3.50	24	0	3.24	24	0	3.49	24	0	3.41
		A*	Euclidean	2	26	1	3.76	26	1	3.49	26	1	3.75	26	1	3.67
		A*	Euclidean	3	25	0	3.63	25	0	3.36	25	0	3.62	25	0	3.54
22	Competitive	A*	Diagonal	1	24	0	3.38	24	0	3.53	24	0	3.55	24	0	3.49
		A*	Diagonal	2	25	0	3.51	25	0	3.66	25	0	3.68	25	0	3.62
		A*	Diagonal	3	26	1	3.65	26	1	3.80	26	1	3.82	26	1	3.76
23	Competitive	A*	Manhattan	1	24	0	3.46	24	0	3.61	24	0	3.55	24	0	3.54
		A*	Euclidean	2	25	0	3.59	25	0	3.75	25	0	3.68	25	0	3.67
		A*	Diagonal	3	26	1	3.73	26	1	3.88	26	1	3.82	26	1	3.81

Table 20: Test results for Test Environment 2

Test Environment 2					Test 1			Test 2			Test 3			Average		
Test	Strategy	Algorithm	Heuristic	NPC	Steps	Hits	Time (secs)	Steps	Hits	Time (secs)	Steps	Hits	Time (secs)	Avg Steps	Hits	Avg Time
1	Single NPC	Dijkstra's	N/A	N/A	28	0	3.71	28	0	4.00	28	0	4.18	28	0	3.96
2	Single NPC	BFS	N/A	N/A	36	0	4.89	36	0	4.93	36	0	5.06	36	0	4.96
3	Single NPC	A*	Manhattan	N/A	28	0	3.90	28	0	3.68	28	0	3.91	28	0	3.83
4	Single NPC	A*	Euclidean	N/A	28	0	3.86	28	0	3.96	28	0	3.96	28	0	3.93
5	Single NPC	A*	Diagonal	N/A	28	0	3.90	28	0	3.97	28	0	4.06	28	0	3.98
6	Cooperative	Dijkstra's	N/A	1	28	0	4.23	28	0	4.16	28	0	4.23	28	0	4.21
		Dijkstra's	N/A	2	27	0	4.10	27	0	4.03	27	0	4.08	27	0	4.07
		Dijkstra's	N/A	3	28	0	4.23	28	0	4.16	28	0	4.23	28	0	4.21
7	Cooperative	BFS	N/A	1	36	0	4.37	36	0	4.95	36	0	5.07	36	0	4.80
		BFS	N/A	2	37	0	4.49	37	0	5.08	37	0	5.20	37	0	4.92
		BFS	N/A	3	38	0	4.61	38	0	5.21	38	0	5.33	38	0	5.05
8	Cooperative	A*	Manhattan	1	28	0	3.80	28	0	3.96	28	0	4.03	28	0	3.93
		A*	Manhattan	2	27	0	3.68	27	0	3.83	27	0	3.90	27	0	3.80
		A*	Manhattan	3	28	0	3.80	28	0	3.96	28	0	4.03	28	0	3.93
9	Cooperative	A*	Euclidean	1	28	0	3.92	28	0	4.03	28	0	3.97	28	0	3.97
		A*	Euclidean	2	27	0	3.79	27	0	3.90	27	0	3.84	27	0	3.84
		A*	Euclidean	3	28	0	3.92	28	0	4.03	28	0	3.97	28	0	3.97
10	Cooperative	A*	Diagonal	1	28	0	3.93	28	0	4.03	28	0	4.05	28	0	4.00

		A*	Diagonal	2	27	0	3.80	27	0	3.89	27	0	3.91	27	0	3.87
		A*	Diagonal	3	28	0	3.93	28	0	4.03	28	0	4.05	28	0	4.00
11	Cooperative	A*	Manhattan	1	28	0	3.71	28	0	4.02	28	0	4.03	28	0	3.92
		A*	Euclidean	2	27	0	3.58	27	0	3.89	27	0	3.90	27	0	3.79
		A*	Diagonal	3	28	0	3.71	28	0	4.02	28	0	4.03	28	0	3.92
12	Non-Cooperative	Dijkstra’s	N/A	1	28	0	4.11	28	0	4.21	28	0	4.22	28	0	4.18
		Dijkstra’s	N/A	2	27	0	3.97	27	0	4.08	27	0	4.08	27	0	4.04
		Dijkstra’s	N/A	3	27	0	3.97	27	0	4.08	27	0	4.08	27	0	4.04
13	Non-Cooperative	BFS	N/A	1	36	0	4.89	36	0	4.95	36	0	5.06	36	0	4.97
		BFS	N/A	2	37	0	5.01	37	0	5.09	37	0	5.19	37	0	5.10
		BFS	N/A	3	37	0	5.01	37	0	5.09	37	0	5.19	37	0	5.10
14	Non-Cooperative	A*	Manhattan	1	28	0	3.85	28	0	4.03	28	0	4.04	28	0	3.97
		A*	Manhattan	2	27	0	3.73	27	0	3.89	27	0	3.90	27	0	3.84
		A*	Manhattan	3	27	0	3.73	27	0	3.89	27	0	3.90	27	0	3.84
15	Non-Cooperative	A*	Euclidean	1	28	0	3.95	28	0	4.08	28	0	4.08	28	0	4.04
		A*	Euclidean	2	27	0	3.82	27	0	3.95	27	0	3.95	27	0	3.91
		A*	Euclidean	3	27	0	3.82	27	0	3.95	27	0	3.95	27	0	3.91
16	Non-Cooperative	A*	Diagonal	1	28	0	4.08	28	0	4.09	28	0	4.13	28	0	4.10
		A*	Diagonal	2	27	0	3.95	27	0	3.96	27	0	3.99	27	0	3.97
		A*	Diagonal	3	27	0	3.95	27	0	3.96	27	0	3.99	27	0	3.97
17	Non-	A*	Manhattan	1	28	0	4.04	28	0	4.09	28	0	4.10	28	0	4.08

	Cooperative														
		A*	Euclidean	2	27	0	3.91	27	0	3.95	27	0	3.97	27	0	3.94
		A*	Diagonal	3	27	0	3.91	27	0	3.95	27	0	3.97	27	0	3.94
18	Competitive	Dijkstra’s	N/A	1	29	1	4.26	29	1	4.36	29	1	4.40	29	1	4.34
		Dijkstra’s	N/A	2	28	1	4.13	28	1	4.22	28	1	4.26	28	1	4.20
		Dijkstra’s	N/A	3	27	0	3.97	27	0	4.07	27	0	4.11	27	0	4.05
19	Competitive	BFS	N/A	1	36	0	4.93	36	0	5.24	36	0	5.09	36	0	5.09
		BFS	N/A	2	37	0	5.06	37	0	5.38	37	0	5.22	37	0	5.22
		BFS	N/A	3	38	1	5.19	38	1	5.52	38	1	5.36	38	1	5.36
20	Competitive	A*	Manhattan	1	29	1	3.69	29	1	4.11	29	1	4.24	29	1	4.01
		A*	Manhattan	2	28	1	3.57	28	1	3.98	28	1	4.10	28	1	3.88
		A*	Manhattan	3	27	0	3.45	27	0	3.85	27	0	3.96	27	0	3.75
21	Competitive	A*	Euclidean	1	29	1	4.18	29	1	4.23	29	1	4.29	29	1	4.23
		A*	Euclidean	2	28	1	4.05	28	1	4.09	28	1	4.16	28	1	4.10
		A*	Euclidean	3	27	0	3.91	27	0	3.96	27	0	4.02	27	0	3.96
22	Competitive	A*	Diagonal	1	29	1	4.16	29	1	4.21	29	1	4.27	29	1	4.21
		A*	Diagonal	2	28	1	4.02	28	1	4.08	28	1	4.13	28	1	4.08
		A*	Diagonal	3	27	0	3.89	27	0	3.94	27	0	3.99	27	0	3.94
23	Competitive	A*	Manhattan	1	29	1	4.12	29	1	4.24	29	1	4.22	29	1	4.19
		A*	Euclidean	2	28	1	3.99	28	1	4.11	28	1	4.08	28	1	4.06
		A*	Diagonal	3	27	0	3.86	27	0	3.97	27	0	3.95	27	0	3.93

Table 21: Test results for Test Environment 3

Test Environment 3					Test 1			Test 2			Test 3			Average		
Test	Strategy	Algorithm	Heuristic	NPC	Steps	Hits	Time (secs)	Steps	Hits	Time (secs)	Steps	Hits	Time (secs)	Avg Steps	Hits	Avg Time
1	Single NPC	Dijkstra's	N/A	N/A	28	0	3.68	28	0	3.97	28	0	4.09	28	0	3.91
2	Single NPC	BFS	N/A	N/A	34	0	4.35	34	0	4.60	34	0	4.70	34	0	4.55
3	Single NPC	A*	Manhattan	N/A	28	0	3.95	28	0	3.94	28	0	3.97	28	0	3.95
4	Single NPC	A*	Euclidean	N/A	28	0	3.86	28	0	3.96	28	0	4.00	28	0	3.94
5	Single NPC	A*	Diagonal	N/A	28	0	3.92	28	0	4.02	28	0	4.00	28	0	3.98
6	Cooperative	Dijkstra's	N/A	1	28	0	3.89	28	0	4.28	28	0	4.17	28	0	4.11
		Dijkstra's	N/A	2	29	0	4.02	29	0	4.41	29	0	4.30	29	0	4.24
		Dijkstra's	N/A	3	27	0	3.76	27	0	4.14	27	0	4.03	27	0	3.98
7	Cooperative	BFS	N/A	1	34	0	4.86	34	0	4.91	34	0	4.89	34	0	4.89
		BFS	N/A	2	35	0	4.99	35	0	5.05	35	0	5.03	35	0	5.02
		BFS	N/A	3	36	0	5.13	36	0	5.19	36	0	5.17	36	0	5.16
8	Cooperative	A*	Manhattan	1	28	0	4.02	28	0	4.15	28	0	4.19	28	0	4.12
		A*	Manhattan	2	29	0	4.16	29	0	4.29	29	0	4.33	29	0	4.26
		A*	Manhattan	3	27	0	3.89	27	0	4.01	27	0	4.05	27	0	3.98
9	Cooperative	A*	Euclidean	1	28	0	4.08	28	0	4.14	28	0	4.19	28	0	4.14
		A*	Euclidean	2	29	0	4.23	29	0	4.27	29	0	4.34	29	0	4.28
		A*	Euclidean	3	27	0	3.94	27	0	4.00	27	0	4.06	27	0	4.00

10	Cooperative	A*	Diagonal	1	28	0	4.13	28	0	4.06	28	0	4.16	28	0	4.12
		A*	Diagonal	2	29	0	4.27	29	0	4.20	29	0	4.30	29	0	4.26
		A*	Diagonal	3	27	0	4.00	27	0	3.93	27	0	4.03	27	0	3.99
11	Cooperative	A*	Manhattan	1	28	0	4.12	28	0	4.17	28	0	4.25	28	0	4.18
		A*	Euclidean	2	29	0	4.26	29	0	4.31	29	0	4.39	29	0	4.32
		A*	Diagonal	3	27	0	3.98	27	0	4.04	27	0	4.12	27	0	4.05
12	Non-Cooperative	Dijkstra’s	N/A	1	28	0	4.13	28	0	4.24	28	0	4.29	28	0	4.22
		Dijkstra’s	N/A	2	29	0	4.26	29	0	4.38	29	0	4.43	29	0	4.36
		Dijkstra’s	N/A	3	27	0	3.99	27	0	4.10	27	0	4.15	27	0	4.08
13	Non-Cooperative	BFS	N/A	1	34	0	4.82	34	0	5.03	34	0	5.01	34	0	4.95
		BFS	N/A	2	35	0	4.95	35	0	5.16	35	0	5.15	35	0	5.09
		BFS	N/A	3	35	0	4.95	35	0	5.16	35	0	5.15	35	0	5.09
14	Non-Cooperative	A*	Manhattan	1	28	0	4.13	28	0	4.13	28	0	4.15	28	0	4.14
		A*	Manhattan	2	29	0	4.27	29	0	4.27	29	0	4.29	29	0	4.28
		A*	Manhattan	3	27	0	3.99	27	0	3.99	27	0	4.01	27	0	4.00
15	Non-Cooperative	A*	Euclidean	1	28	0	4.23	28	0	4.22	28	0	4.17	28	0	4.21
		A*	Euclidean	2	29	0	4.37	29	0	4.36	29	0	4.31	29	0	4.35
		A*	Euclidean	3	27	0	4.09	27	0	4.08	27	0	4.04	27	0	4.07
16	Non-Cooperative	A*	Diagonal	1	28	0	4.27	28	0	4.23	28	0	4.18	28	0	4.23
		A*	Diagonal	2	29	0	4.40	29	0	4.37	29	0	4.31	29	0	4.36
		A*	Diagonal	3	27	0	4.13	27	0	4.08	27	0	4.04	27	0	4.08

17	Non-Cooperative	A*	Manhattan	1	28	0	4.07	28	0	4.15	28	0	4.12	28	0	4.11
		A*	Euclidean	2	29	0	4.21	29	0	4.28	29	0	4.26	29	0	4.25
		A*	Diagonal	3	27	0	3.94	27	0	4.01	27	0	3.99	27	0	3.98
18	Competitive	Dijkstra’s	N/A	1	28	0	4.11	28	0	4.22	28	0	4.29	28	0	4.21
		Dijkstra’s	N/A	2	29	0	4.25	29	0	4.36	29	0	4.43	29	0	4.35
		Dijkstra’s	N/A	3	27	0	3.98	27	0	4.09	27	0	4.15	27	0	4.07
19	Competitive	BFS	N/A	1	34	0	4.91	34	0	4.99	34	0	5.06	34	0	4.99
		BFS	N/A	2	35	0	5.05	35	0	5.13	35	0	5.20	35	0	5.13
		BFS	N/A	3	36	1	5.19	36	1	5.26	36	1	5.34	36	1	5.26
20	Competitive	A*	Manhattan	1	28	0	4.01	28	0	4.18	28	0	4.16	28	0	4.12
		A*	Manhattan	2	29	0	4.15	29	0	4.32	29	0	4.30	29	0	4.26
		A*	Manhattan	3	27	0	3.88	27	0	4.04	27	0	4.03	27	0	3.98
21	Competitive	A*	Euclidean	1	28	0	4.16	28	0	4.19	28	0	4.20	28	0	4.18
		A*	Euclidean	2	29	0	4.29	29	0	4.32	29	0	4.34	29	0	4.32
		A*	Euclidean	3	27	0	4.02	27	0	4.05	27	0	4.06	27	0	4.04
22	Competitive	A*	Diagonal	1	28	0	4.01	28	0	4.13	28	0	4.21	28	0	4.12
		A*	Diagonal	2	29	0	4.14	29	0	4.26	29	0	4.35	29	0	4.25
		A*	Diagonal	3	27	0	3.87	27	0	3.99	27	0	4.07	27	0	3.98
23	Competitive	A*	Manhattan	1	28	0	3.55	28	0	3.93	28	0	4.10	28	0	3.86
		A*	Euclidean	2	29	0	3.67	29	0	4.07	29	0	4.24	29	0	3.99
		A*	Diagonal	3	27	0	3.43	27	0	3.81	27	0	3.97	27	0	3.74

Table 22: Test results for Test Environment 4

Test Environment 4					Test 1			Test 2			Test 3			Average		
Test	Strategy	Algorithm	Heuristic	NPC	Steps	Hits	Time (secs)	Steps	Hits	Time (secs)	Steps	Hits	Time (secs)	Avg Steps	Hits	Avg Time
1	Single NPC	Dijkstra's	N/A	N/A	24	0	3.24	24	0	3.51	24	0	3.57	24	0	3.44
2	Single NPC	BFS	N/A	N/A	24	0	3.36	24	0	3.44	24	0	3.45	24	0	3.42
3	Single NPC	A*	Manhattan	N/A	24	0	3.50	24	0	3.51	24	0	3.56	24	0	3.52
4	Single NPC	A*	Euclidean	N/A	24	0	3.46	24	0	3.55	24	0	3.54	24	0	3.52
5	Single NPC	A*	Diagonal	N/A	24	0	3.46	24	0	3.55	24	0	3.55	24	0	3.52
6	Cooperative	Dijkstra's	N/A	1	24	0	3.62	24	0	3.69	24	0	3.67	24	0	3.66
		Dijkstra's	N/A	2	25	0	3.76	25	0	3.83	25	0	3.81	25	0	3.80
		Dijkstra's	N/A	3	26	0	3.89	26	0	3.97	26	0	3.94	26	0	3.93
7	Cooperative	BFS	N/A	1	24	0	3.52	24	0	3.60	24	0	3.64	24	0	3.59
		BFS	N/A	2	25	0	3.65	25	0	3.74	25	0	3.79	25	0	3.73
		BFS	N/A	3	26	0	3.78	26	0	3.88	26	0	3.93	26	0	3.86
8	Cooperative	A*	Manhattan	1	24	0	3.47	24	0	3.65	24	0	3.67	24	0	3.60
		A*	Manhattan	2	25	0	3.60	25	0	3.79	25	0	3.81	25	0	3.73
		A*	Manhattan	3	26	0	3.74	26	0	3.93	26	0	3.95	26	0	3.87
9	Cooperative	A*	Euclidean	1	24	0	3.59	24	0	3.60	24	0	3.63	24	0	3.61
		A*	Euclidean	2	25	0	3.73	25	0	3.73	25	0	3.76	25	0	3.74
		A*	Euclidean	3	26	0	3.87	26	0	3.87	26	0	3.90	26	0	3.88
10	Cooperative	A*	Diagonal	1	24	0	3.60	24	0	3.71	24	0	3.62	24	0	3.64

		A*	Diagonal	2	25	0	3.73	25	0	3.85	25	0	3.76	25	0	3.78
		A*	Diagonal	3	26	0	3.87	26	0	3.98	26	0	3.90	26	0	3.92
11	Cooperative	A*	Manhattan	1	24	0	3.58	24	0	3.70	24	0	3.66	24	0	3.65
		A*	Euclidean	2	25	0	3.72	25	0	3.84	25	0	3.80	25	0	3.79
		A*	Diagonal	3	26	0	3.85	26	0	3.98	26	0	3.94	26	0	3.92
12	Non-Cooperative	Dijkstra's	N/A	1	24	0	3.69	24	0	3.70	24	0	3.69	24	0	3.69
		Dijkstra's	N/A	2	25	0	3.83	25	0	3.84	25	0	3.83	25	0	3.83
		Dijkstra's	N/A	3	25	0	3.83	25	0	3.84	25	0	3.83	25	0	3.83
13	Non-Cooperative	BFS	N/A	1	24	0	3.46	24	0	3.66	24	0	3.64	24	0	3.59
		BFS	N/A	2	25	0	3.59	25	0	3.80	25	0	3.78	25	0	3.72
		BFS	N/A	3	25	0	3.59	25	0	3.80	25	0	3.78	25	0	3.72
14	Non-Cooperative	A*	Manhattan	1	24	0	3.61	24	0	3.65	24	0	3.63	24	0	3.63
		A*	Manhattan	2	25	0	3.74	25	0	3.79	25	0	3.76	25	0	3.76
		A*	Manhattan	3	25	0	3.74	25	0	3.79	25	0	3.76	25	0	3.76
15	Non-Cooperative	A*	Euclidean	1	24	0	3.46	24	0	3.58	24	0	3.60	24	0	3.55
		A*	Euclidean	2	25	0	3.59	25	0	3.72	25	0	3.74	25	0	3.68
		A*	Euclidean	3	25	0	3.59	25	0	3.72	25	0	3.74	25	0	3.68
16	Non-Cooperative	A*	Diagonal	1	24	0	3.63	24	0	3.67	24	0	3.64	24	0	3.65
		A*	Diagonal	2	25	0	3.76	25	0	3.81	25	0	3.78	25	0	3.78
		A*	Diagonal	3	25	0	3.76	25	0	3.81	25	0	3.78	25	0	3.78
17	Non-	A*	Manhattan	1	24	0	3.64	24	0	3.62	24	0	3.67	24	0	3.64

	Cooperative															
		A*	Euclidean	2	25	0	3.78	25	0	3.76	25	0	3.80	25	0	3.78
		A*	Diagonal	3	25	0	3.78	25	0	3.76	25	0	3.80	25	0	3.78
18	Competitive	Dijkstra’s	N/A	1	24	0	3.59	24	0	3.68	24	0	3.66	24	0	3.64
		Dijkstra’s	N/A	2	26	1	3.86	26	1	3.95	26	1	3.95	26	1	3.92
		Dijkstra’s	N/A	3	25	0	3.73	25	0	3.82	25	0	3.80	25	0	3.78
19	Competitive	BFS	N/A	1	24	0	3.16	24	0	3.39	24	0	3.46	24	0	3.34
		BFS	N/A	2	25	0	3.28	25	0	3.52	25	0	3.59	25	0	3.46
		BFS	N/A	3	26	1	3.40	26	1	3.65	26	1	3.73	26	1	3.59
20	Competitive	A*	Manhattan	1	24	0	3.37	24	0	3.47	24	0	3.54	24	0	3.46
		A*	Manhattan	2	25	0	3.50	25	0	3.60	25	0	3.68	25	0	3.59
		A*	Manhattan	3	26	1	3.62	26	1	3.74	26	1	3.81	26	1	3.72
21	Competitive	A*	Euclidean	1	24	0	3.50	24	0	3.55	24	0	3.58	24	0	3.54
		A*	Euclidean	2	25	0	3.63	25	0	3.68	25	0	3.71	25	0	3.67
		A*	Euclidean	3	26	1	3.76	26	1	3.82	26	1	3.85	26	1	3.81
22	Competitive	A*	Diagonal	1	24	0	3.56	24	0	3.58	24	0	3.65	24	0	3.60
		A*	Diagonal	2	26	1	3.82	26	1	3.85	26	1	3.92	26	1	3.86
		A*	Diagonal	3	25	0	3.68	25	0	3.71	25	0	3.78	25	0	3.72
23	Competitive	A*	Manhattan	1	24	0	3.46	24	0	3.61	24	0	3.60	24	0	3.56
		A*	Euclidean	2	25	0	3.59	25	0	3.74	25	0	3.74	25	0	3.69
		A*	Diagonal	3	26	1	3.73	26	1	3.88	26	1	3.87	26	1	3.83

Table 23: Test results for Test Environment 5

Test Environment 5					Test 1		Test 2		Test 3		Average	
Test	Strategy	Algorithm	Heuristic	NPC	Steps	Time (secs)	Steps	Time (secs)	Steps	Time (secs)	Avg Steps	Avg Speed
1	N/A	A*	Manhattan	1	16	2.14	16	2.5	16	2.61	16	2.42
	N/A	Dijkstra's	N/A	2	16	2.14	16	2.5	16	2.61	16	2.42
	N/A	BFS	N/A	3	16	2.14	16	2.5	16	2.61	16	2.42
	N/A	A*	Euclidean	4	16	2.14	16	2.5	16	2.61	16	2.42
2	N/A	A*	Manhattan	1	16	2.46	16	2.62	16	2.51	16	2.53
	N/A	A*	Euclidean	2	16	2.46	16	2.62	16	2.51	16	2.53
	N/A	A*	Diagonal	3	16	2.46	16	2.62	16	2.51	16	2.53
	N/A	A*	Manhattan	4	16	2.46	16	2.62	16	2.51	16	2.53