Pathfinding Algorithms in Multi-Agent Systems

Amitha Arun

Master of Science, Computer Animation and Visual Effects

August, 2014

Contents

List of Figures

Abstract

Multi agent systems can be used to simulation traffic and pedestrian activity. The main purpose of such a simulation could be for research purposes, traffic management as well as for crowd simulations and realism in games. The multi-agent system for traffic and pedestrian activity involves two main agents - the vehicles and the pedestrians. These two agents respond to a traffic signal as well as to each other in the given environment.

Although many such simulations are existent, many of them don't consider the inclusion of realistic pathfinding algorithms. This simulation also takes into consideration a small amount of errors in the behavior of agents by using adjustments in the pathfinding algorithm.

Keywords: multi-agent systems, pathfinding

Chapter 1

Introduction

Simulation of multi-agent systems find their use in not only the visual effects and gaming industry but also in other areas such as robotics, logistics, study of artificial intelligence etc. For visual effects, the use of multi-agent systems can be seen in the form of crowd simulation and flocking systems. For Gaming, the same is used for purposes such as planning the movement and behavior of a programmed enemy. What is seen as the main purpose for the existence of multi-agent system in a game environment - is the need for the agents to have human-like response to stimuli.

One of the concepts that goes hand in hand with multi-agent systems is pathfinding. Pathfinding is used to determine the shortest path from the initial node to the destination. They could be used to either control the enemy to help find its way to you or for you to able to control your player on a terrain. There could be more elaborate and complex uses of the pathfinding algorithm in games. The main drawback of using these pathfinding algorithms is the lack of a realistic navigation along the path generated by them.

Pathfinding is is still a problem in modern games as it continues to deliver unbelievable movement in the agents. Some mistakes with respect to incorrect pathfinding are often quite amusing, for instance, there are times when the enemy gets stuck in between the waypoints on the terrain.

So while the enemy's animation continues running, he is stuck in collision with an obstacle. There are instances when the enemy finds a rather long and unnatural path to you rather than the shortest one. Some other times, the enemies just walk through solid obstacles! T (2010) This calls for a not only an efficient pathfinding algorithm, but also one that appears more life-like and intelligent. At the same time, the path shouldn't be too intelligent, but must make room for human errors.

Figure 1.1: *In this example, the enemy walks through the obstacle with ease T (2010).*

Figure 1.2: *The algorithm finds a long path rather than a short one T (2010).*

There are several pathfinding algorithms existent today. The most common being - Breadth First Search, Dijkstra's Algorithm and A* algorithm. Each of these are improvements of each other, having very similiar techniques for computing the path.

Problem statement

The aim of my project is to create simple agents on a grid that respond to external stimuli as well as use different pathfinding methods to establish the difference between them and how thay can be improved for better navigation. Most importantly, I have focussed on the basic implementation details of each of the pathfinding algorithms to aid in explanation of these algorithms.

Objectives and contributions

The objective of this thesis is to understand the kind of pathfinding algorithms best suited for a kind of grid or terrain and to aid in understanding of the implementation of such pathfinding algorithms.

Structure

The chapter Background talks about the previous work related to the general topics in pathfinding. The Theory chapters goes into the details of the components of AI programming i.e. the use of agents' brain and body. The Algorithms chapter is focussed only on the heuristics and pathfinding algorithms and goes into details about its runtime complexity and implementation.The design and applications unit discusses the design of the classes for this project and the actual implementation examples. Finally, the conclusion elaborates the limitations, future work and summary of the entire thesis.

Chapter 2

Background

Much research has been done and continues to be done in the field of multi-agent systems and artificial intelligence. A multi-agent system can be thought of as an intelligent system that is capable of making decisions and can potentially imitate a human brain. When an environment contains several of these agents interacting with each other, they are known as multi-agent systems. A lot of information about the multi-agent systems can be discussed but I will focus this thesis on the pathfinding specifics.

In a game environment, the players find their way through an area with the use of pathfinding algorithms. A pathfinding algorithm determines a route between two or more points in a given graphical representation of a terrain. In order for a game to come together, we must create agents and an environment and have them interact with each other. In the next few sections, we discuss - Pathfinding Algorithms, Geometry in Games, Artificial Intelligence and Finite State Machines.

2.1 Pathfinding Algorithms

There have been numerous attempts at pathfinding in the past. Although the concept of pathfinding is almost an instinctive behavior of mobile living beings, mathematical research into this area began much later. One

of the earliest mentions of pathfinding could be the 'Travelling salesman' problem which was conceived by William Rowan Hamilton in the 1800s. This problem, which dealt with finding the shortest route through multiple cities by visiting them only once, gained interest only in the 1900s and solutions to this problem were vastly encouraged. One of the earliest pathfinding solutions was known as the Depth First search, developed by Charles Pierre Tremaux. Following this, many new pathfinding algorithms such as - Breadth First Search, Bellman-Ford's, Dijkstra's and A* have come into use.

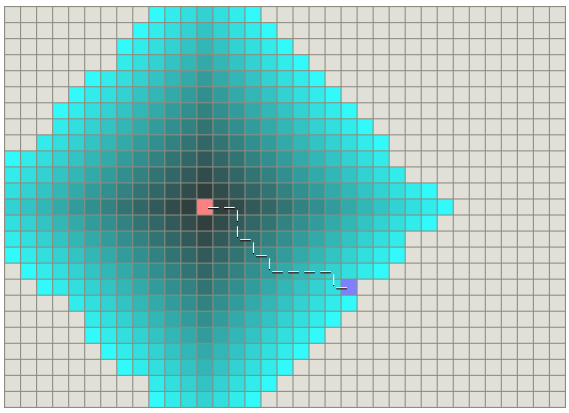

Figure 2.1: *The nodes searched using Dijkstra's Algorithm Patel (2010).*

Modern pathfinding methods used in games, are based on the Dijkstra's algorithm, the most popular being the A* algorithm. A* algorithm is an improvement on Dijkstra as it uses heuristics to ensure better performance of the pathfinding Hart *et al.* (1972). According to what is described by the authors of the paper, if the heuristic (n) defined is the lower bound of the minimal cost from the source node to the destination, then A* would find the minimum cost path to the goal. There are many new pathfinding algorithms based off this method, namely - D*, Theta*, Field D* etc. Theta* is almost identical to the A* algorithm but it returns consistently straight line paths by considering all possible paths as successorsNash (2010). D* is an incremental search which continually improves the search and corrects its information when it receives the same. Likhachev *et al.* (2005)

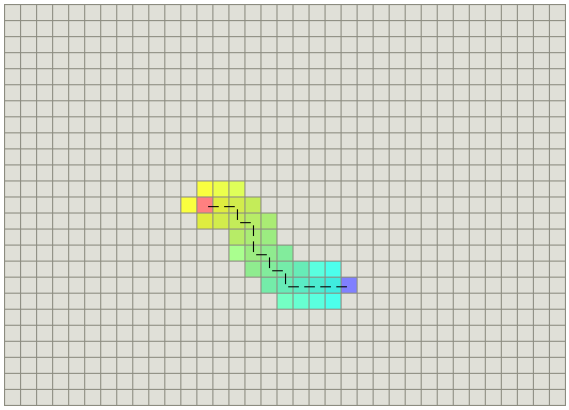

Figure 2.2: *he nodes searched using A* Algorithm Patel (2010).*

2.2 Geometry in Games

For games, the pathfinding algorithms are typically used on a particular kind of geometry. This geometry could be specified in terms of either - navigation meshes or waypoints, as explained earlier. All the pathfinding algorithms either use waypoints or navigation meshes to define the walkable ares on their terrain. For obvious reasons, the use of navigation meshes are the first step to remove the bugs in the pathfinding algorithms. Navigation meshes are a set of two dimensional polygons used to define accessible areas in a game environment Snook (2000). Each polygon represents a node which can be used as input to the pathfinding algorithms.

Figure 2.3: *The terrain with waypoints T (2010).*

Navigation meshes require just a few nodes to specify large areas that

the agent can walk on - as opposed to the waypoints method which create a node for every walkable area on the terrain. But the ease of use of waypoints is far easier and it is simpler to understand. Most of the games created in the Unity 3D game engine, use the Recast and Detour library created by Mikko Mononen to generate navigation meshes for their terrain.

Figure 2.4: *The terrain with navigation mesh T (2010).*

2.3 Artificial Intelligence

Artificial Intelligence is defined as the study of creating machines that have to potential to exhibit human intelligence. For games in particular, Artificial intelligence creates an 'illusion of intelligence' rather than intelligent processing Buckland (2004). The first attempts at using Game AI was the computerised version of the game Nims, which was programmed to win most of of the timeof Play (2015). In the 1950s, a programming computer for playing chess was created immediately followed by a programmed version of noughts and crosses. With the proliferation of hardware and software, some of the first few popular video games came into existence - Mario, Pacman, Donkey Kong, Legend of Zelda etc. Games today are varied, and the AI used in these games have also seen a growth. Most of the AI in these games are scripted for NPCs. In the video game 'Creatures' Grand (1997), the AI is programmed to talk,

eat and protect itself from other characters and was hence considered a milestone in the Game AI scene. Later games like F.E.A.R and Far Cry have a more sophisticated AI that takes on multiple tasks at a time and can respond sufficiently to the environment that it is present in. In general, AI for games could deal with many topics such as robotics, pathfinding, graphics and finite state machines.

Figure 2.5: *The gameplay for the game F.E.A.R Productions (2005).*

Finite State Machines can be used to define the AI of a system which assumes one of many states at a given time. Based on an action, the state of the system can change from one to another but also ensuring that only one state is possible at that time.

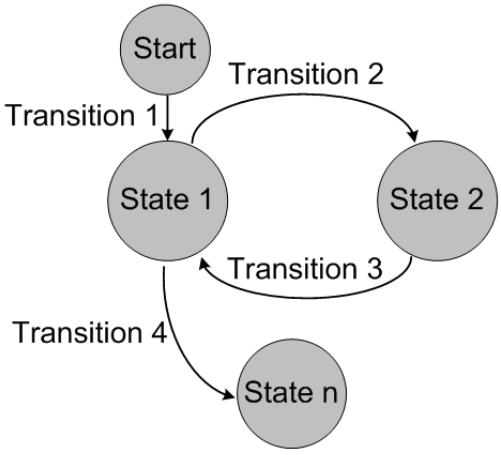

Figure 2.6: *An FSM showing states and transitions Productions (2005).*

Other ways of specifying the intelligence in a system could be the use of neural networks or fuzzy logic. These are relatively more advanced

than a simple finite state machine but a proper Artificial Intelligence system could be chosen based on the requirement of the game agents.

Chapter 3

Theory

An agent is said to be an entity with knowledge, goals and actions Woolridge and Wooldridge (2001). A multi-agent system comprises an environment with two or more of these agents interacting with each other based on the conditions of the environment.

The main topics that need to be addressed to simulate an agent in an environment are - agent brain, agent body.

3.0.1 Agent Brain

The brain of the agent is the central unit that makes decisions on its behalf. It reads the state of the environment from the body and makes decisions as to what action needs to be taken. After this action is executed, the agent may be put into a new state or may continue to remain in the state that it was in. The instructions given by the brain may be as simple as an if condition, which when satisfied, a statement may be executed. These instructions may also be very complex, like in video games, where the agent could be doing several tasks at once.

The instruction itself could be specified in the form of a finite state machine, which is what I have considered for my project. The environment that I have created is that of a street that consists of the two agents - pedestrians and cars along with the presence of a traffic light that goes

between red and green at certain time intervals. The agents each assume two states. The pedestrians could be either in the 'Walking' state or in the 'Waiting' state. The default state of a pedestrian is Walking but when the signal is green and the pedestrian is at a particular location, the pedestrian must enter the 'Stopped' state. The cars could be in the state 'Driving' by default and when the signal is red and the cars are at a particular position, they must enter the 'Stopping' state. The signal is just a continuous change between red and green at an interval of a couple of seconds. A state 'orange' could also be added to the signal to prevent risk of collisions.

The brains for the agents is scripted in python using simple if else statements.

The brain for the pedestrian would appear as shown:

Signal is green and the pedestrian is at the crossing change state to Waiting State remains as Walking and position to be updated

For the car, the brain would have the following structure: Signal is green and the car is at the crossing change state to Stopping State remains as Driving and position to be updated

The signal just switches between red, green and orange every 5 - 10 seconds.

The update script simply checks the current position and increments it to the next position that is determined by the pathfinding algorithm.

3.0.2 Agent Body

The body of an agent is the point of contact between the brain and the outside environment. It is synonymous with a sensor that checks the environment and relays this information to the brain for it to take some action. When the brain determines the action to be taken, the body responds to this and then checks the environment again and this cycle continues. The agent body holds information about the physical properties of the agent like - position, speed, strength, mass etc. For

the purpose of my simulation, the following are the physical properties assigned to the agents.

Position The important physical property with respect to this simulation is the current position of the agent. Since the pathfinding algorithms determine the path from the source to the destination, the current position is continuously updated so that the agent follows the path. The position is also the result of the forces due to flocking as well as collision detection.

Velocity The velocity of the agent can also be gathered by using this change in position. This is also affected by the flocking forces.

Acceleration This determines the rate of change of velocity of the agents.

3.0.3 Flocking

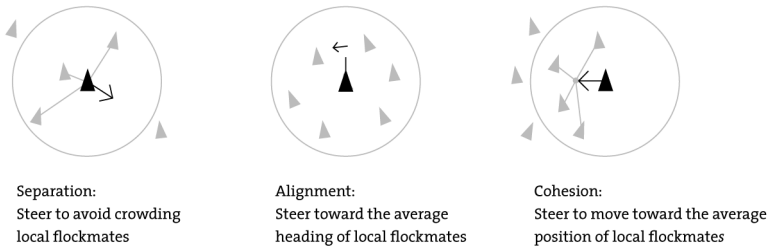

Figure 3.1: *Separation, Cohesion and Alignment Sebastian (2013).*

The study of crowds demonstrates the basic rules of flocking - Alignment, Separation and Cohesion Reynolds (1987). The rules of flocking are:

1. Separation: All agents must avoid collisions with its neighbours.
2. Alignment: Each agent must match direction with its neighbours.
3. Cohesion: Each agent attempts to steer towards the center of the crowd.

3.0.4 Collision

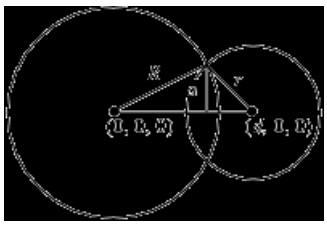

Figure 3.2: *Sphere-sphere collision Weisstein (2015).*

It is important to ensure that none of the agents collide with each other. One of the simpler methods for collision detection is sphere-sphere collision detection. The idea is to take the radii of the two spheres involved in the collision and add them up and compare them to the distance between the centers of the spheres. If this distance is less than the radii of the spheres, it means that a collision has occurred. On detecting the collision, the response is to simply reverse the direction of the agent. This is also added to the list of forces to compute the final position.

There is no requirement for collision detection in the pathfinding algorithm since the input to the algorithm itself checks for obstacles. This results in a path that avoids the obstacles without relying on the collision detection to do it later on. So, the collision detection is meant only for the agents colliding among each other.

Chapter 4

Pathfinding Algorithms and their Comparison

4.0.5 Heuristics

Before beginning the section about the pathfinding algorithms it would be worth mentioning heuristics. A general description of a heuristic would be a rule of thumb or argument that can be used to solve a computational problem Rouse (2009). It is basically an observational rule that makes a particular computation tend towards optimality. The type of heuristic that could be used would completely depend on the type of problem. So there is no 'one-type fits all' kind of heuristic that makes the solution more optimal; It completely problem-dependent. An admissible heuristic is one that does not overestimate the minimum cost of path between the source to the goal. But most pathfinding algorithms require a heuristic that would be consistent or monotonic. A consistent heuristic is one who's value does not decrease on traversing down a path web. The choice of the heuristics have an impact on the way the algorithms arrive at the solution. For the general A* pathfinding algorithm Patel (2010):

1. if the value of $h(n)$ is 0, the A* algorithm gets converted to the Dijkstra's Algorithm where the shortest path is guaranteed to be found.

2. if the value of $h(n)$ is lower than the cost to reach the goal, the shortest path is guaranteed but the process is slowed down.

3. if $h(n)$ is, the shortest will be found very fast but it is not easy to determine such an ideal heuristic.

4. if $h(n)$ is greater than the cost from source to goal, the shortest path is not guaranteed but the program would run faster.

Some of the heuristics that have been used for the implementation of the pathfinding algorithms are listed below:

Manhattan Distance Manhattan distance is the distance that would be travelled if the path taken from one point to another is in right angles. It is simply calculated as:

$\sum_{i=1}^{n} |p_i - q_i|$

where, $p_i - q_i$ is the difference between the two points in the x, y or z coordinates. This would usually be used if the agent is allowd to move freely in 4 directions on the grid.

Euclidean Distance This is the length of the line segment connecting two points. It is usually preferred if any direction of movement can be considered. It is calculated as:

$\sum_{i=1}^{n} (q_i - p_i)^2$

Diagonal Distance For diagonal distance, we consider the cost of moving diagonally which saves you the cost of taking non-diagonal steps.

4.0.6 Dijkstra's Algorithm

Dijkstra's algorithm, which is a shortest path finding algorithm, was invented by Edsger Dijkstra. It finds the shortest path from a node to all other nodes in the graph. Dijkstra's algorithm works by adding the

closest neighbours of the start node, examining them and then moving on to its neighbours. The pseudocode is as shown below:

```
Input: Graph G = (V, E), directed or undirected; positive edge lengths {l_e : e ∈
  E}; vertex s ∈ V
Output: For all vertices u reachable from s, dist(u) is set to the distance from
  s to u.
  procedure DIJKSTRA(G, l, s)

    for all u ∈ V do
        dist(u) = ∞
        prev(u) = nil
    dist(s) = 0

    H = MAKEQUEUE(V)                          ▷ using dist-values as keys
    while H is not empty do
        u = DELETEMIN(H)
        for all edges (u, v) ∈ E do
            if dist(v) > dist(u) + l(u, v) then
                dist(v) = dist(u) + l(u, v)
                prev(v) = u
                DECREASEKEY(H, v)
```

Figure 4.1: *Pseudocode showing Dijkstra's Algorithm Sanjoy Dasgupta (2006)*

Algorithm According to the pseudocode, the algorithm first scans the source and sets its distance to itself as 0 and its predecessor as undefined. All the other nodes in the graph are added to a queue. As long as this queue isnt empty, the algorithm repeatedly checks for a minimum value for distance in the queue and removes it. For each of this nodes neighbours, if a shorter path to them has been found, the distance is updates to this value and its predecessor is set to the current node. When using this algorithm for games, the same applies with the only difference being the way that the nodes are represented. We may be dealing with a grid represented as waypoints or navigation meshes. The algorithm benefits with the use of a priority queue which provides as a faster alternative to a queue.

Runtime Complexity If we consider a basic implementation of Dijkstra's Algorithm (where each node finds the shortest distance to every other node), the algorithm runs the main loop as many times as the

number of vertices and each vertex is investigated once in the loop. This makes the best and the worst case complexity equal to n^2 where n is the number of nodes.

The running time complexity of Dijkstra's algorithm has a relation to the kind of queue or heap used in the implementation. There is one main while loop that executes in a graph G(V,E) using Dijkstra's Algorithm which extracts a vertex from a queue. If there are V vertices, the running time can be equated to O(V). The pop operation in a binary heap implementation of the priority queue has runtime complexity of O(log V) which imples that the total time for the main loop is (V log V). At this point the tested node is discarded and its neighbours are considered and tested for an improved path. This test is performed with complexity O(E). The entire algorithm runs with a worst case complexity of $\theta((|E| + |V|)log|V|)$.

If, however, a fibonacci heap is used, the complexity changes to $\mathcal{O}((|E|+|V|)log|V|)$ which is considered as the best case. The average case is $\mathcal{O}(|E| + |V|log\frac{|E|}{|V|}log|V|)$.

Implementation For Dijkstra's Algorithm, the implementation is fairly simple. The idea is to read an adjacency matrix which is basically a matrix that describes a graph by denoting the edges and costs assigned to each of these edges. There is a vector called the distance vector which is initialized to zero. We could use an priority queue to store the visited nodes. For as long as the queue is not empty, we must, search the child nodes for the destination. If not found, it means that a path does not exist.

Applications Dijkstra's algorithm can best be used in a situation where there is a requirement of finding the path from one node to possibly all the other nodes. Dijkstra's Algorithm would, hence, be suitable to applications such as communication between routers. If the nodes that are expanded are considered, Dijkstra's Algorithm tends to expand more nodes than its counterparts but is, nevertheless, guaranteed to find

the shortest path. The kind of path that may result from the Dijkstra algorithm and A* algorithm may be comparable but the simple difference lies in the number of nodes expanded by the two algorithms.

1. A value is popped fromm the queue and its neighbours are inspected.

2. If a new distance vector is calculated, it is pushed into the queue.

A simpler approach would be to follow the A* pathfinding algorithm and set the heauristic value to be zero. This means that the next node to be investigated is not selected based on a pre-determined heuristic. This ensures that all the neighbouring nodes have an equal chance of being investigated and the heuristic doesn't play a role in making that educated guess.

4.0.7 A* Algorithm

This algorithm is a modified version of the Dijkstra's Algorithm. According to the paper Hart *et al.* (1972), there are two approaches to pathfinding called the Mathematical approach and the Heuristic approach. The mathematical approach involves the orderly testing of nodes in a graph to ultimately reach the destination. The heuristic approach, on the other hand, uses special knowledge about the kind of problem to gather information about the a more efficient shortest path. This algorithm combines the two approaches - it uses information about the problem to later determine a mathematical solution to find a path.

Algorithm Most algorithms need to expand the number of nodes they search and the size this expansion would vary from algorithm to algorithm. The idea would be to expand fewer nodes and this may be possible by making a very informaed decision about the nodes to be expanded. The algorithm for A* explains how it chooses the nodes to be expanded. The algorithm begins with adding the source node to a list of nodes to be considered - called open list. This is the list of nodes that need to be

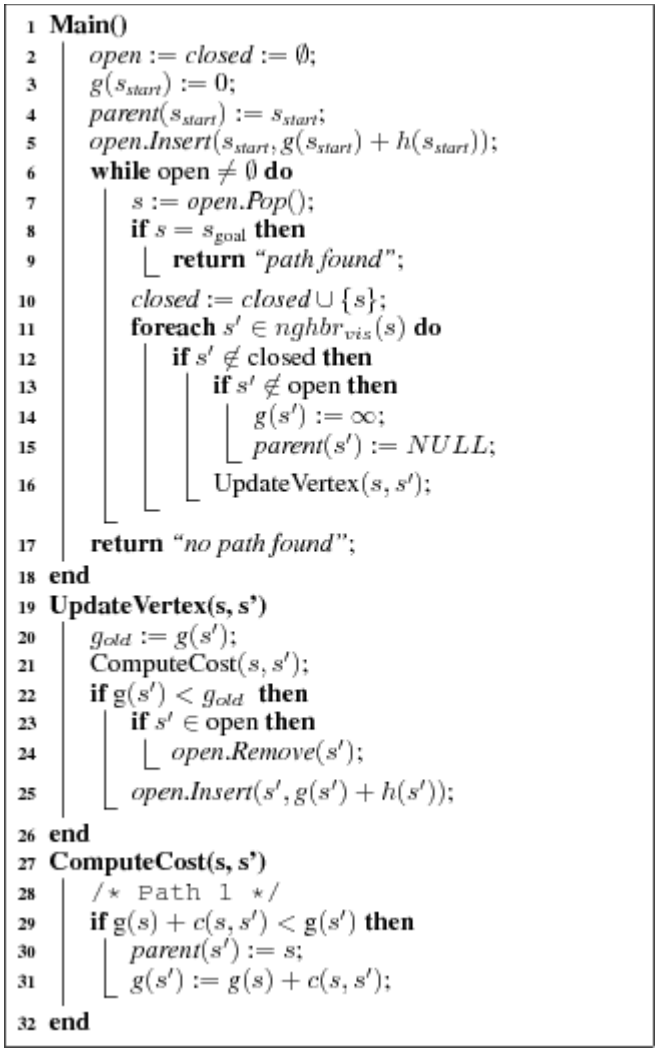

Figure 4.2: *Pseudocode showing A* Algorithm Daniel* et al. *(2010)*

checked out. The neighbours of this nood need to be checked out subsequently, and its parents need to be marked to its predecessor. Drop the first node from the open list and add it to the closed list - which contains nodes that needn't be checked again. From the list of open nodes, we need to pick one. This can be done by considering the equation $\hat{f}(n) = \hat{g}(n) + \hat{h}(n)$, which is known as the evaluation function. $\hat{g}(n)$ is the movement cost from the source to the given node and $\hat{h}(n)$ is the distance from the node being considered to the destination (also known as the heuristic). The most common way to calculate this heuristic value is using Manhattan method, although, there are other ways of obtaining the same value. This is akin to an educated guess in terms of pathfind-

ing. Hence we choose the next node to be expanded by considering the smallest value of $\hat{f}(n)$.

Usually a priority queue (LIFO) would be used to store the checked out nodes. Also in order to backtrack to find the shortest path, the parent node could be stored for each of the checked out nodes. When the destination is reached, the shortest path can be determined by starting from the destination and tracking it all the way to the source.

Dijkstra's Algorithm is actually a special case of A* algorithm where the heuristic value is not considered at all or $\hat{h}(n) = 0$.

Post processing of the path returned by A* can make it seemingly more realistic. However this often results in increased runtime at the cost of a better and possibly shorter path Daniel *et al.* (2010). One possible method would be - after finding the path from source to destination, the current vertex checks if its has a line of to its successor after the first. If it does, the post smoothing removes the intermediate path and makes a direct jump between these vertices. It continues this process for the subsequent successors to obtain a more direct path. This may calculate a better path than the A* but is not guaranteed to provide the true short path or a even a smooth one. Since it only refines an already calculated A*, the changes are minor but somewhat effective.

Runtime Complexity For a general case, we consider the time complexity of the algorithm to be $nlog(n)$.The runtime of this algorithm is mainly dependent on the heuristic. In the worst case scenario that the destination is found in an unbounded area, the complexity is $\mathcal{O}(b^d)$, where b is the number of nodes expanded and d is the length of the shortest path. If the goal isn't found, however, the algorithm continues infinitely assuming that the space isn't confined. If the data structure for the nodes is a tree, the complexity is found to be $|h(x) - h^*(x)| = \mathcal{O}(logh^*(x))$, where h^* is the optimal heuristic.

Implementation Details A lot of materials are available on the algorithm used for the A* algorithm and so, this section is a focus on just

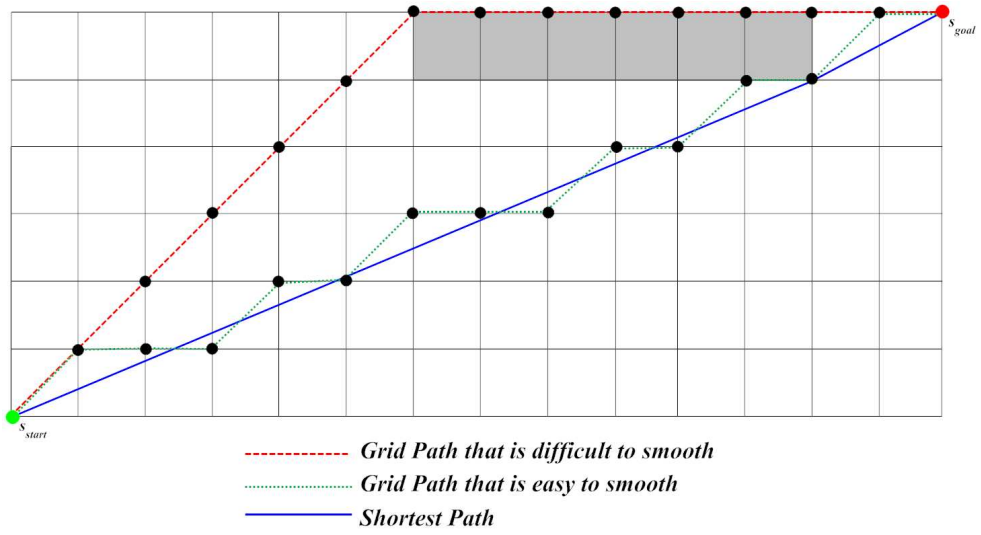

Figure 4.3: *Paths obtained after different post smoothing techniquesDaniel* et al. *(2010).*

the implementation details of this algorithm. The idea is to get a terrain that has either waypoints or graph nodes. Some of the implementation details that can be gathered from the algorithm are listed below:

1. The choice of the data structure to be used was the priority queue to store the open list of nodes that have been checked out. The idea is to push the start node into the openlist after initializing the open list and the closed list to zero.

2. Each node is represented as having a row, column, its $\hat{f}(n)$, $\hat{g}(n)$ and $\hat{h}(n)$ values. A pointer to this node is used to store the element removed from the priority queue.

3. As the algorithm suggests, the source is pushed into the open list and the new successors are determined with the help of a heuristic value. This could be Manhattan distance, Euclidean distance or Diagonal distance.

Applications The most common pathfinding algorithms to be used is the A* pathfinding algorithm. Dijkstra's Algorithm is a special case of the A* pathfinding algorithm since the heuristic value that is considered in A* is set as 0 for Dijkstra's. The main use of A* pathfinding is seen

in games to usually control the player or the enemy AI. The number of nodes expanded of the algorithm is much lesser than the Dijkstra's Algorithm.

4.0.8 Theta* Algorithm

There have been many variants of the A* algorithm and one other interesting one is the Theta* algorithm. Although there has been conclusive proof that a better or shorter path than what comes out of the A* algorithm is practically impossible to obtain, the need for alternate algorithms exist. The shortest path returned by A* is blocky and unrealistic and any agent that follows this path looks unreal. The problem with A* is that the shortest path on the grid doesn't actually mean the same thing as a continuous short path. So, the idea behind the conception of theta* was to not constrain the path to graph edges but to allow the parent edges to be from any angle. For the most part, this algorithm is just a few statements different from A* but it is an effective way to make sure that our path is the shortest.

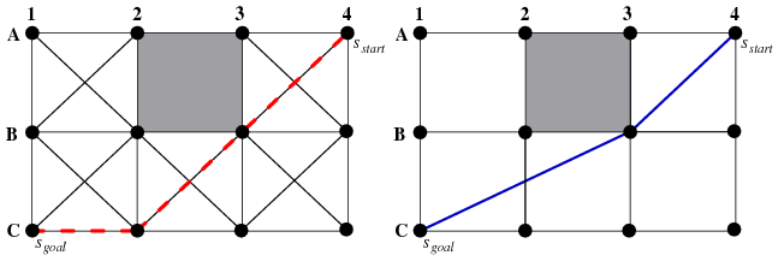

Figure 4.4: *Difference between grid path and shortest pathDaniel* et al. *(2010).*

Algorithm The only difference in the algorithm is that when the parent of the selected node is set, we may consider any of the nodes to the parent as long as the length is equivalent to the original as well as the parent has a line of sight from the current node as well as to the destination. As seen in the figure, the actual change in path produced by the two algorithms may be subtle but when it comes to the actual movement of the agent, this makes all the difference.

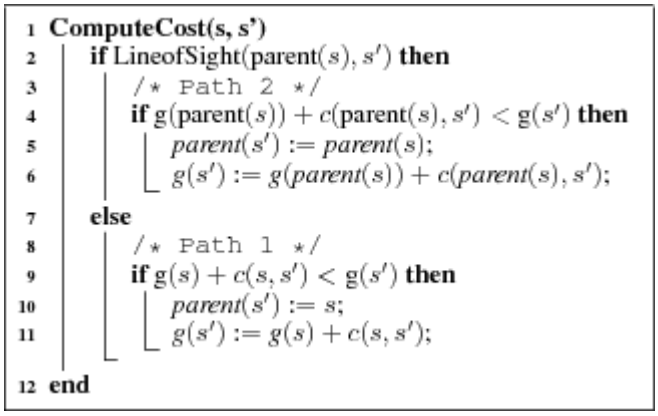

Figure 4.5: *Pseudocode showing parts of Theta* Algorithm Daniel* et al. *(2010)*

The Line of Sight simply draws a straight line between two points on the display without it being interrupted by the presence of a node in between. This line is possible only if there is no node in between that would cut off the connection between these two nodes.

Runtime Complexity Since the algorithm for Theta* is more or less exactly same as A*, the runtime complexity would reflect similiar results. The only difference is that Theta* may be slightly slower but with the use of post smoothing, this can be improved. Also if visibility graphs are used over grid graphs, theta* is significantly faster than its counterpart.

The basic version of Theta* is not optimal and there alternatives possible for this algorithm based on requirement - Angle-Propagation Theta*, Field D* etc. The Field D* algorithm is again, very similiar to the A*, but when checking for neighbour, it finds a value on the perimeter that is at the same distance from the concerned node. Angle-Propagation Theta* propagates ranges of angles to check if they have line of sight. This ensures that the actual runtime is much less than the basic Theta*.

Implementation The open list can be represented using a priority queue just like in case of the A* algorithm. For all of the successors, a child node is created who's parent is set as the predecessor. The algo-

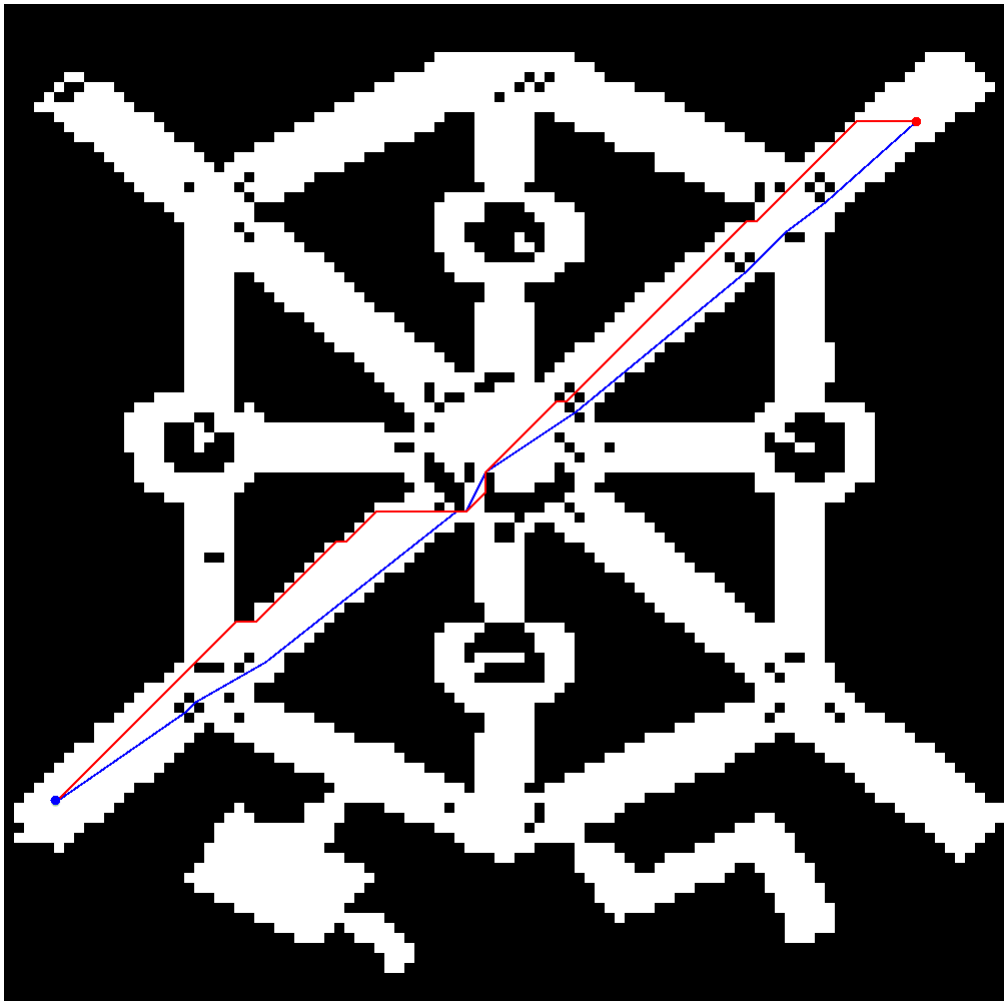

Figure 4.6: *Trace of the Theta* Algorithm Daniel* et al. *(2010).*

rithm is fairly straightforward in terms of the implementation.

Applications Theta* was introduced for the prime purpose of creating true short paths which are also realistic. It could be most suited for game AI or realistic crowd simulations for the most part. It doesn't necessarily improve the performance or arrive at an optimal solution. It achieves a smoother and believeable path for the agents.

4.0.9 Real-Time A* Algorithm

If we are considering a truly intelligent agent, we must definitely take into account the errors he would make in navigation. For instance, If there is a pedestrian who doesn't know the way to the exit, he would probably walk a few yards, pause, look around, move in the wrong direction and then proceed in the right direction after obtaining more information. For a realistic pathfinding, the agent may not be aware of all his surroundings but would rather just be aware of a part around him and the general direction of the destination. A normal implementation of the pathfinding algorithms may not have the capacity to deal with such a scenario because this would involve revisiting older nodes and that would mean an increased cost. This problem, however, does not have a solution as research is still being conducted in this area. A real time approach to would have the following characteristics Korf (1990):

1. Consideration of a limited search horizon because there is a limit to how far ahead the vision system can see.
2. Actions must be committed before their ultimate consequences are known
3. Finding a more realistic path comes at the cost of forgoing the shortest path and time.

In the minimin lookahead search Korf (1990), we make decisions based on the information we can gather by making a move. Information about the predecessors of the node are made before the actual move and after moving, this process is repeated. The decision of a move can be made by the best option available but after the move, it is possible to backtrack to the previous state. We can still use the A* algorithm having the calculation $\hat{f}(n) = \hat{g}(n) + \hat{h}(n)$ but we must ensure to keep the value $\hat{g}(n)$ constant as the cost should be the actual number of moves instead of their cost of execution. There is use for the open stack and closed list like in A* algorithm except that when all moves lead to a closed state, the agent can backtrack until it finds one that hasn't been closed. But the Real-Time A* Algorithm is explained below:

Algorithm Real-Time A* Algorithm doesn't only backtrack when a dead-end is encountered but also when is believes that a better solution may be available due to it. This can be possible by using a normal A* with an additional conjecture that the cost of backtracking and finding the solution would be less than going forward. The main difference would be in the calculation of $\hat{g}(n)$ which would now be the distance of the node from current state and not from the source as before. It basically stores the $\hat{h}(n)$ values (calculated by the minimin lookahead) of previous states on a stack and then updates the $\hat{g}(n)$ value to their actual distance from the current state and then moves to the new node having least$\hat{g}(n)+\hat{h}(n)$ value. The functionality of an open and closed list can be created by a single list with all of the predecessor nodes.

Runtime Complexity The Real-Time A* Algorithm is sure to find the goal state in graph with non-negative edge costs and finite heuristics values, if the goal is ultimately reachable from the other states. The solution cannot be checked for efficiency as it defeats its purpose but it can be compared for accuracy and this is dependent on its heuristic function. The quality of the solution in using this algorithm increases with the increase in the search area. As specified in the paper Korf (1990), the computation for every move increases exponentially with the increase in the size of the area to be searched but the length of the search asymptotically reaches the optimal value at greater horizon values.

Implementation The main data structure used to store the nodes along with their values is a hash table. This hash table is indexed by the value of $\hat{f}(n)$. As the algorithm suggests, the source is initialized and is added to the hash table. On adding the successors into the hash table, the first $\hat{f}(n)$ value is chosen as the hash table would be sorted in ascending order. The current node's heuristic is updated to the second lowest value of $\hat{f}(n)$ in the hash table.

Application Another realistic behavior of agents would be the errors that could be introduced in the simulation. The best use of the LRTA*

algorithm may be to simulate human errors in navigation and for this reason, it could be used in gaming or simulations to imitate an agent with a very small or limited view of the terrain or grid that it is walking on.

4.0.10 Other Pathfinding Algorithms

There are numerous other pathfinding algorithms that have come into the picture and each one has its specific use.

D* D* or Dynamic A* was conceived mainly for the purpose of robotics in order to dynamically obtain the path to destination Stentz (1995). It can modify paths optimally in case the mobile robot (with a sensor attached) makes a move and realises it needs to recompute the path. The algorithm is very similiar to the A* algorithm having the open and closed lists. The D* algorithm denotes the presence of an obstacle by giving a high value to its OBSTACLE variable. The presence of empty spaces is given by the variable EMPTY. The main difference is in the fact that the paths are computed locally keeping in mind that the sensors have a very limited range of vision. Also, the robotic agent would make a close-to-monotonic progress towards the goal. There are also two more variables - RAISE and LOWER which denote the increased or decreased cost of node which are added to the closed list. This increase or descrease of cost is brought about by the presence of GATE, which computes additional cost of possibly opening a gate and then walking through it. The algorithm first computes an initial path from the start to the destination and then modifies it based on certain costs during the travel. It produces an optimal path ensuring that at every state, the robot travels an optimal path to the destination.

Focussed D* This is an enhanced version of D* that combines the A* heuristics with the D* algorithm Stentz (1995). The performance of the focussed D* is better than the original D*. A* can be considered as a

special case of the focussed D* where the arc costs are constant and the pathfinding is not dynamic.

Lifelong Planning A* Lifelong Planning A* is an incremental search version of A* pathfinding Koenig *et al.* (2004). For initialization, it contains a value $g(s)$ for all the values which correspond to the same for A*. This keeps track of the distance of the node from the source. There is also an $rhs(s)$ value that is a one step lookahead based on the $g(s)$ values. Each vertex is investigated twice at most. The main idea of the algorithm is that it uses the basic A* pathfinding on the first iteration of the algorithm and saves each state of its search for faster subsequent searches. Incremental search algorithms are said to find optimal solutions which are faster than solving each of the problems from scratch.

D* Lite The main purpose of the dynamic algorithms is to compute paths in unknown terrains. The D* Lite is an incremental search algorithm which uses the information it obtains from previous searches to aid in computing new paths (like in LPA*). This D* Lite is very close to the D* algorithm but is shorter and just as efficient.

Chapter 5

Design and Implementation

The class diagram for the entire project implementation is shown in the figure below. The description of the classes are elaborated in this section.

5.0.11 Crowd

This class is responsible for the creation and management of the agents. The class has a method that is responsible for creation of an agent at a random position concentrated at the particular area on the grid. It also has a method for checking and storing the neighbours of each of the agents and calculates the same on several threads at the same time. The class also makes sure so destroy the agents that have reached their destination.

5.0.12 Agents

This class represents each agent in the system and is perhaps the most indispensible of the classes. It contains the script for the brains of each of the agents. Based on the options read from the user interface, the appropriate pathfinding algorithm is chosen and the path from source to destination is returned to the object. There is a method that updates the position of the agents after a time interval. The update class calls the python script for the calculations and change in position and reads

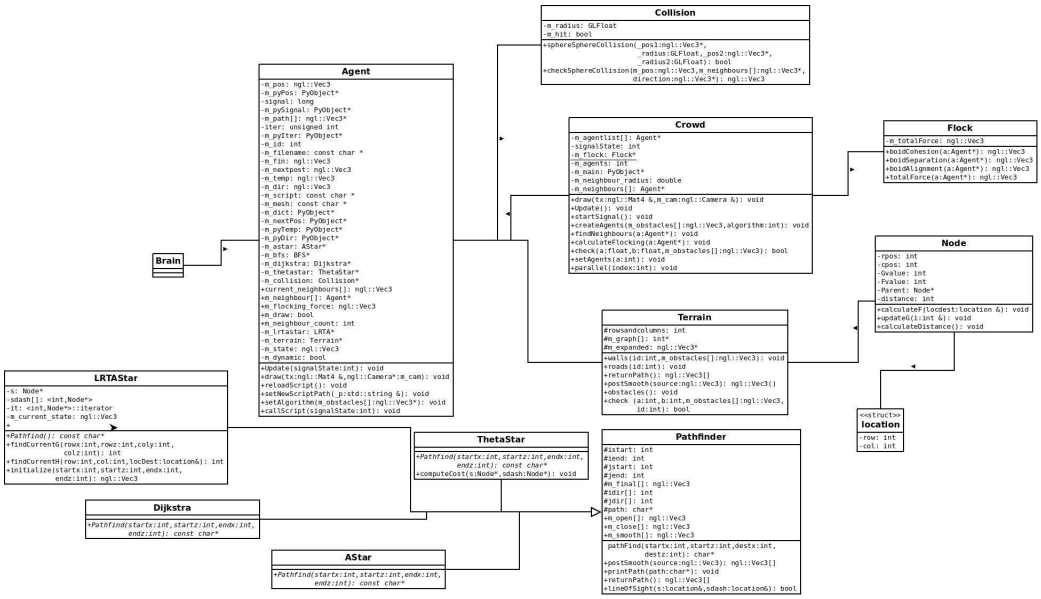

Figure 5.1: *Class Diagram for the project.*

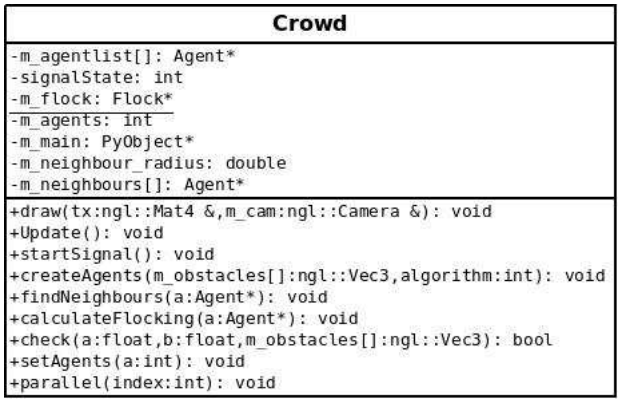

Figure 5.2: *Crowd Class.*

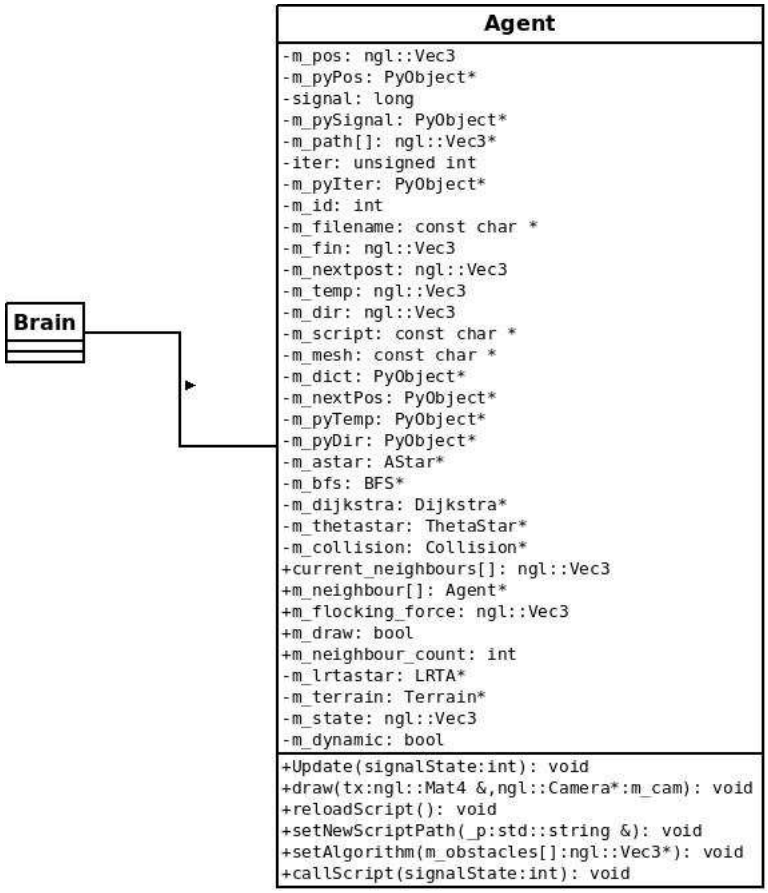

Figure 5.3: *Agent Class*

the new position for the next iteration. This class also has a method to draw the agent at the position returned by the python script. It also has separate methods to find the dynamic path.

5.0.13 Terrain

The Terrain class holds the information about the grid for the pathfinding algorithms. There are some abstract methods present in this class that are common to all the derived classes of this class. This class has the attributes and methods that describe the terrain and the waypoints. This class also takes care of defining the walkable areas and the obstacles on the grid before the pathfinding is called.

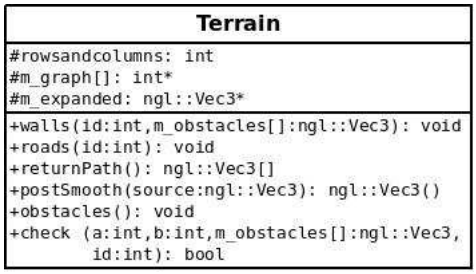

Figure 5.4: *Terrain Class*

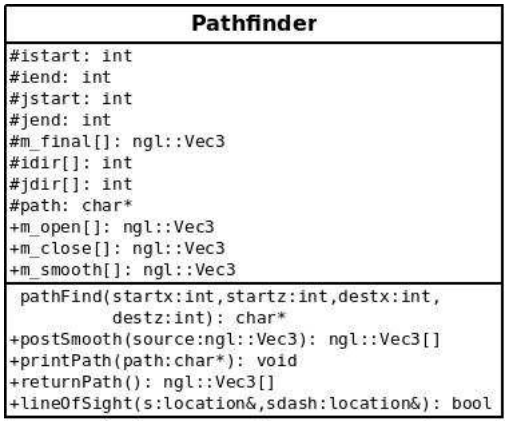

Figure 5.5: *Pathfinder Class*

5.0.14 Pathfinder

The derived classes are the different pathfinding algotihms (AStar, ThetaStar, Dijkstra) which overload the virtual methods according to their specification. It consists of the line of sight algorithm which is an integral part of the Theta* algorithm and post smoothing for the A*.

5.0.15 Node

The Node class is the foundation for the pathfinding to function. The Node class helps to create objects that can store the value of row number, column number, pointer to parent node and values such as distance, G value and F value (for A*). An object of Node class can collectively store all this information and creates a better representation of a node in a graph. It uses a structure location to keep track of the row and column of the node.

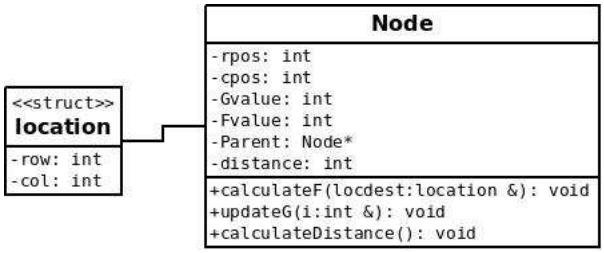

Figure 5.6: *Node Class*

Figure 5.7: *AStar Class*

5.0.16 AStar

This is derived class of the Terrain class. It specifically deals with the A* pathfinding algorithm as specified in the 'Pathfinding Algorithms and their Comparison' section.

5.0.17 ThetaStar

This class implements the Theta* algorithm as specified in one of the previous sections. It uses the line of sight method of its base class as per the specifications of the algorithm.

5.0.18 Dijkstra

This is another derived class of the Terrain class which implements Dijkstra's shortest path algorithm.

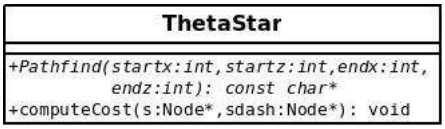

Figure 5.8: *ThetaStar Class*

Figure 5.9: *Dijkstra Class*

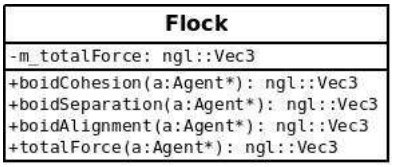

Figure 5.10: *Flock Class*

5.0.19 Flock

Flock class takes care of adding all the forces due to flocking and returns it to the agent class for the calculation.

5.0.20 Collision

The collision class takes care of adding collision detection among agents. This just checks whether a sphere-sphere collision is happening and applies a force in the opposite direction.

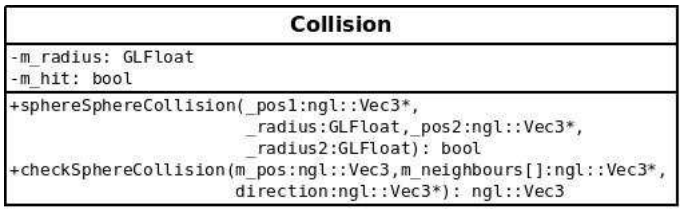

Figure 5.11: *Collision Class*

Chapter 6

Applications & results

The applications, observations and results are presented in this section.

6.0.21 Comparison of the pathfinding algorithms

The comparison of the algorithms is described in the table below. Each of the algorithms is contrasted on the basis of certain criteria as seen in the table.

Pathfinding Algorithm	Dijkstra
General Time Complexity	n^2
Running time for Best Case	$\|E\| + \|V\|)log\|V\|$
Running time for Worst Case	$\theta((\|E\| + \|V\|)log\|V\|)$
Running time for average case	$\|E\| + \|V\|log\frac{\|E\|}{\|V\|}log\|V\|$
Average Nodes expanded(per 400)	292
Data Structures Used	adjacency list,binary/fibonacci heap
Heuristics	no heuristics
Applications	For destination that is unknown or sparse graphs

The features of Dijkstra's Algorithm.

Pathfinding Algorithm	A*
General Time Complexity	$nlog(n)$
Running time for Best Case	$\|h(x) - h^*(x)\| = \mathcal{O}(logh^*(x))$
Running time for Worst Case	infinity if path to goal can't be found/ d^n
Running time for average case	$\|h(x) - h^*(x)\| = \mathcal{O}(logh^*(x))$
Average Nodes expanded(per 400)	64
Data Structures Used	priority queue
Heuristics	Manhattan/Euclidean
Applications	Simple efficient pathfinding

The features of A* Algorithm.

Pathfinding Algorithm	Theta*
General Time Complexity	$nlog(n)$
Running time for Best Case	same as A*
Running time for Worst Case	infinity if path can't be found
Running time for average case	same as A*
Average Nodes expanded(per 400)	291
Data Structures Used	queue
Heuristics	Manhattan/Euclidean
Applications	Realistic movement of agent

The features of Theta* Algorithm.

Pathfinding Algorithm	Real-Time A*
General Time Complexity	-
Running time for Best Case	same as A*
Running time for Worst Case	infinity
Running time for average case	2^n
Average Nodes expanded(per 400)	104
Data Structures Used	hash table
Heuristics	Minimin lookahead
Applications	For terrain that is unknown

The features of LRTA* Algorithm.

In the table n is the number of nodes, E is number of edges and V is the number of vertices and d is the length of shortest path.

Implementation Since the path found by the algorithms are roughly the same, there would more relevance in discussing the execution times of the algorithms.

Number of Agents	5	8	10
A*	9.101	346.6578	588.2921
Dijkstra	587.401	1245.8744	1443.951
Theta*	588.121	1355.7654	1628.491

The executing time in units of the algorithms with respect to the number of agents.

Expanded nodes Expansion of nodes is done in order to reach the goal state. One of the performance measures of an algorithm would be the least number of nodes expanded by the algorithm. The following diagrams show the number of nodes expanded by each of the algorithms.

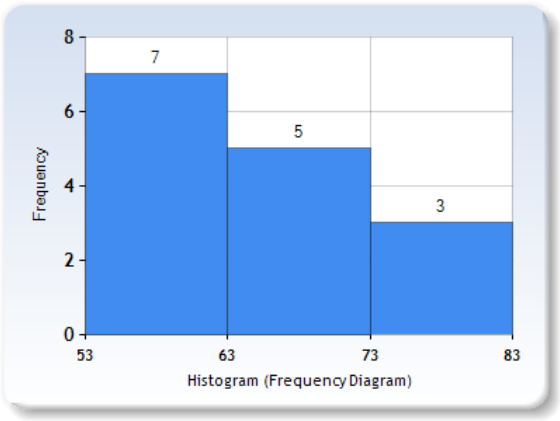

Figure 6.1: *Nodes expanded by A* algorithm.*

As seen in the histograms, the A* algorithm expands the least number of nodes. Since Dijkstra is a basic algorithm, it probably expands the most number of nodes. The LRPA* expansion might vary based on

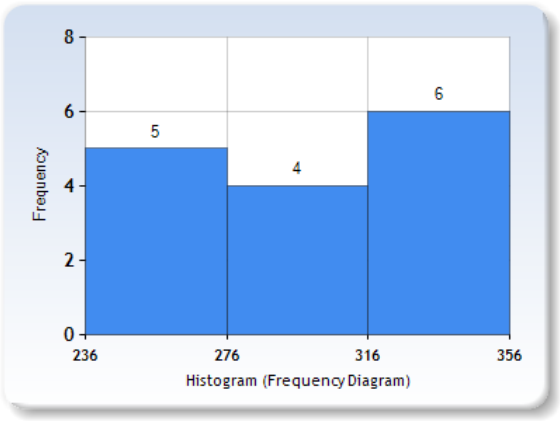

Figure 6.2: *Nodes expanded by Dijkstra algorithm.*

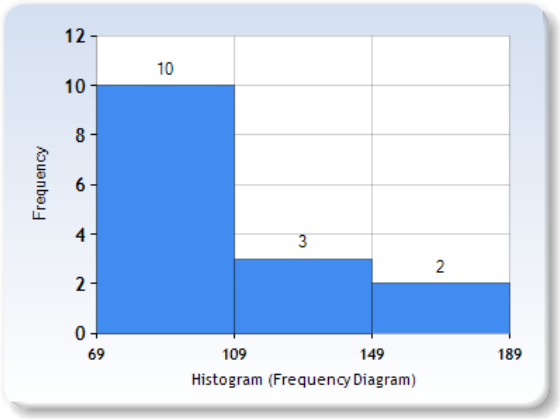

Figure 6.3: *Nodes expanded by Theta* algorithm.*

the scenario since its a dynamic algorithm. This is also why this algorithm has the greatest range. Based on the results, it can be seen that the theta* algorithm actually expands more nodes than expected. The algorithm doesn't focus too much on its efficiency; It rather looks at the realistic path it delivers.

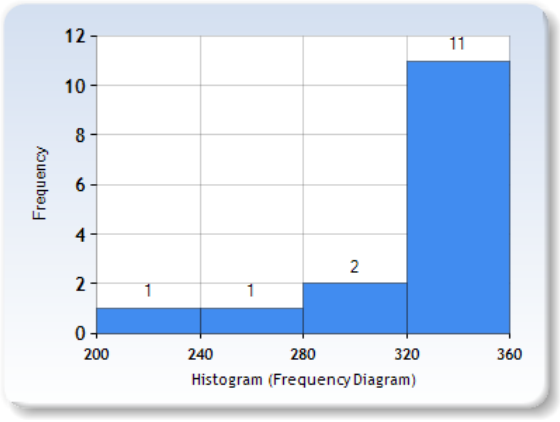

Figure 6.4: *Nodes expanded by LRTA* algorithm.*

Chapter 7

Conclusion

This thesis was directed at studying various pathfinding algorithms and comparing them with respect to their efficiency, realism and accuracy. There is no winning pathfinding algorithm and the deciding factor for the best algorithm depends completely on the scenario of the implementation. If the goal is to simulate the behavior of agents in a particular situation (disaster evacuation, traffic simulation), the realistic approach for pathfinding would be a better choice i.e. Real-time pathfinding with A*. However, if the idea is to obtain the actual shortest path at minimum cost, the approach would be the A* algorithm or Theta* Algorithm. If what we want is more realistic movement of the agents, the post smoothing with A* would work the best. There may be a better suited pathfinding algorithm for all the different requirements and the challenge is to make the coorect choice.

There is much improvement possible in the area of real-time pathfinding and there is no such thing as a solution that has emerged from the ample research that has been done.

7.0.22 Summary

I have implemented the following pathfinding algorithms - BFS, Dijkstra's Algorithm, A* Algorithm, A* with post smoothing, Theta* Algorithm and the real-time A* algorithm. The complexities and the accu-

racy od these algorithms have also been compared. The simulation also demonstrates the behavior of two agents with different brains responding to the environment it is present in along with collision detection and flocking.

7.0.23 Critique and Limitations

The major areas of concern, in this project, were in the large number of modules to be integrated. Firstly, there was a requirement of a basic agent which was controlled by python scripts. The other module was the description of the walkable areas and blocked off areas on the terrain. Finally, there was the implementation of the pathfinding algorithms itself. The biggest challenge was to bring these modules together because although they seemed to work by themselves, they had trouble after integration.

Initially, it seemed difficult to actually put down the algorithms as code. But eventually, the method of actually going through a scientific paper got much easier.

The implementation mainly focussed on the comparison of already implemented algorithms based on criteria such as - efficiency, performance and time complexities.

There are some bugs in the code - especially in areas where the python script is embedded into the C++ code. Also, the Learning Real Time A* has an issue with the backtracking. These issues are minor and do not have a direct consequence on the purpose of the project. Some small errors with A* post smoothing and some GUI components weren't behaving as expected. There were some problems with the collision and flocking behavior but they are rather unimportant.

The main learning I have achieved from this project is a good understanding of design and the use of C++ libraries that I had no previous knowledge of. The implementation is exactly as suggested in the algorithm and the mistakes could only be in the misinterpretation of the algorithms and not the coding. I have become more accustomed to

the coding standards and the use of OpenGL in graphics programming thanks to this project.

This project could be very useful for the understanding of which pathfinding to use under what circumstances.

7.0.24 Future Work

Since the main focus of my project was on the pathfinding algorithms, there needs to be a lot of improvement in the agents brain since they are currently a basic implementation. I would hope to add more complex brains with more states and transitions. An introduction of steering behaviors would have been interesting for the interaction of agents.

There could be more improvement in the speed of the simulation since the multi-threading works only for small bits of the code; It would be more useful to parallelize the entire agent behavior to make the simulation faster.

The entire pathfinding was done using waypoints and a future addition could be the same using navigation meshes.

The improvement could be boundless but the research may lead to more interesting discoveries.

Bibliography

http://www.cs.colostate.edu/~anderson/cs440/index.html/doku.php?id=notes:week4c.

Buckland M., September 2004. *Programming Game AI By Example (Wordware Game Developers Library)*. Jones & Bartlett Learning, 1 edition.

Daniel K., Nash A., Koenig S. and Felner A., 2010. Theta*: Any-angle path planning on grids. *J. Artif. Intell. Res. (JAIR)*, 533–579.

Grand S., 1997. Creatures. CD-ROM.

Hart P. E., Nilsson N. J. and Raphael B., December 1972. ucorrection/u to "a formal basis for the heuristic determination of minimum cost paths". *SIGART Bull.*, (37), 28–29.

Koenig S., Likhachev M. and Furcy D., May 2004. Lifelong planning a*. *Artif. Intell.*, **155**(1-2), 93–146.

Korf R. E., March 1990. Real-time heuristic search. *Artif. Intell.*, **42** (2-3), 189–211.

Likhachev M., Ferguson D., Gordon G., Stentz A. T. and Thrun S., June 2005. Anytime dynamic a*: An anytime, replanning algorithm. In *Proceedings of the International Conference on Automated Planning and Scheduling (ICAPS)*.

Nash A., 2010. Theta* any angle paths. http://aigamedev.com/open/tutorials/theta-star-any-angle-paths/.

of Play N. M., 2015. Video game history timeline. http://www.museumofplay.org/icheg-game-history/timeline/.

Patel A., 2010. Introduction to a*. http://theory.stanford.edu/~amitp/GameProgramming/.

Productions M., 2005. F.e.a.r. CD-ROM.

Reynolds C. W., 1987. Flocks, herds and schools: A distributed behavioral model. In *Proceedings of the 14th Annual Conference on Computer Graphics and Interactive Techniques*, SIGGRAPH '87, New York, NY, USA. ACM, 25–34.

Rouse M., 2009. Flocking like it's 1999. http://whatis.techtarget.com/definition/heuristic.

Sanjoy Dasgupta U. V., Christos Papadimitriou, 2006. Algorithms. http://rosalind.info/glossary/algo-dijkstras-algorithm/.

Sebastian , 2013. Flocking like it's 1999. http://coffeepoweredmachine.com/flocking-like-its-1999/.

Snook G. 2000. 288–304. Simplified 3d movement and pathfinding using navigation meshes. In DeLoura M., editor, *Game Programming Gems*, Charles River Media.

Stentz A., 1995. The focussed d* algorithm for real-time replanning. In *Proceedings of the 14th International Joint Conference on Artificial Intelligence - Volume 2*, IJCAI'95, San Francisco, CA, USA. Morgan Kaufmann Publishers Inc., 1652–1659.

T P., 2010. Pathfinding once and for all. http://www.ai-blog.net/archives/000152.html.

Weisstein E. W., 2015. Sphere-sphere intersection. http://mathworld.wolfram.com/Sphere-SphereIntersection.html.

Woolridge M. and Wooldridge M. J., 2001. *Introduction to Multiagent Systems*. John Wiley & Sons, Inc., New York, NY, USA.