Dynamic Search Algorithms in Minecraft World

Haiwen Zhou, Cyrus Chen

CSCI4511w Writing 04

May 11, 2018

1 Team Members

Haiwen Zhou 5085530 zhou0416

Cyrus Chen 4936191 chen3736

2 Abstract

Goal oriented searching in dynamic terrain has been studied for a long time and various path finding algorithms have been introduced to solve this kind of problems. To find the best way to approach the goal in a closed environment, algorithms like A*, D*lite, etc... are planned to be tested in Minecraft world and test them in different situations. Make comparisons between different algorithms in different testing environments. The testing environments will include dynamic terrain and static terrain, might focus on dynamic terrain more since it's more interesting and more relevant to artificial intelligence. The discussion in this paper will be on the topic of algorithm comparison and choosing, depends on their performances in different situations and work done by other research. The theoretically best performing algorithm and variation will be chosen to the be coded and tested in the actual Minecraft environment.

3 Introduction

In real-life scenario, automatic driving, unmanned aerial vehicle are commonly used in military and aerospace engineering. Applying these technologies involved a lot of path finding algorithms, which have become an developed field in artificial intelligence, and usually the cost of the projects isn't worth the risk. So simulation could be an alternative choice for testing the existing approaches in certain environment. Like Minecraft world, one of the most powerful and famous stimulating game environment, people can do basically everything as long as it's logically possible. Testing and comparing different search algorithm in Minecraft world will serve multiple purposes, it is an implementation in virtual world, and it is also an simulation of the real world running algorithm.

Possible environments in Minecraft world for applying search algorithms includes: mining diamonds or finding other loot, solving mazes, finding shortest path to a specific location with or without a map. We will be implementing search algorithms in the game, it's like a "Bot" for the game. Minecraft is already there and ready to be used. Similar bot programs exist for other purpose (PvP, looting blocks), but the algorithms the use are unknown. More specifically, there will be two main categories that requires search algorithms and they lead to two different goal-oriented panth finding problems: Shortest path problem, and path finding problem in dynamic terrain.

To find subjects in closed form environment in dynamic terrain, incremental heuristic search runs faster than classical heuristic search by reusing information from previous searches to speed up its current search[1]. Classical heuristic search A*,for example, is not efficient when multiple agents are searching at the same time and also when the map is too large [2]. However, they are still considered as one of the best algorithms in the field. Dijkstra's Algorithm is also in our list duo to it's powerful performance and easy to implement.

There will be further discussion of planned algorithms in the following sections.

4 Related Work and Algorithms

This section will discuss the search algorithms we selected to be implemented in Minecraft world, a comparison of their performance in different situations an advantages of each.

4.1 A* Search Algorithm

In 2009, W. Zeng and R. L. Church published their study on the topic of real road based shortest path finding using A* search.[3] A* search was developed at 1968 and using an estimate cost function to find the best path between the origin and the destination.

A* search has two cost function. g(x) stands for the real cost to reach a node x. and h(x) stands for an approximate cost from node x to goal node. It is called the heuristic function. The heuristic function should never overestimate the cost for the algorithm solution, this is called an admissible heuristic. The total cost of each node is calculated by f(x)=g(x)+h(x).

A* search only expands a node if it seems promising. That mean, it only moves on to the node if it has the lowest total cost. A* search only focuses on reaching the goal node from the current node, so it does not calculate the cost for other nodes unless their coast is lower than the new explored nodes. It guarantees an optimal solution, if the heuristic function is admissible, which means the heuristic function should never overestimate the cost. Later variants of A* search, like iterative deepening A* search, D* search, are either focus on some type of situation (different type of map), or optimizing performance, but they all keep the core idea of A* search which is using a heuristic function to compute the estimate cost. This is extremely powerful when the graph is huge, since the algorithm will not search the nodes with a really high estimate cost. However, when the graph is small, or the estimate value is not actuate, a lot of time and space are wasted on computing the estimate value.

A* have a list of different variants that serves by different functionality, like famous D* lite and MPGAA* we discussed earlier this semester, there are potential advantages and limit in different variant of the algorithm. To find the best search algorithm for the problem, we went through several papers comparing the search algorithms which includes two classes from incremental heuristic search, one is extended A* class like Adaptive A* and latest MPGAA* and the other one is D*, D* Lite class. [4]

In Hernandez's paper, it gives a improved version of A* which outperforms D*-Lite by reusing paths found by previous A* searches when re-planning is needed. More specifically, each time a search episode finishes, MPAA* updates the h values of many of the states of the search space, making them more informative [5]. So the cost of path from start position to goal position is perfect after the update. And it gave experiments that proved that MPAA* outperforms D*Lite almost always for path-planning in partially known terrain in situations that are comparable to indoor and outdoor navigation except on maze maps. And D*Lite does better only in problems in which very few optimal path exist [5].

Also an updated version of MPAA* in another paper from the same author, MPGAA* was compared with older algorithms on dynamic terrain. By the experiment of eight-neighbor grids, it gives that the actual memory requirement are very similar for all algorithms and they all grow linearly on the size of the grid.[6]. Przybylski's paper introduced several incremental path-planning algorithms in partially known terrain, including MPGAA*, D* and D* Lite. And it concludes that for typical two-dimensional, navigation problems D* Extra Lite outperforms both D*Lite (optimized version) and MPGAA*[7]. For the project, we will try MPGAA*, since it's more understandable than D* class and have similar performance compared with D*Extra Lite.

4.2 Dijkstra's Algorithm

Dijkstra's algorithms can be considered as a special case for A* search: when the heuristics is zero. It was conceived by Edsger W. Dijkstra in 1956 and published a few years later.[8] The algorithm is finding the shortest paths between nodes in a graph. The most common variant fixes a node to be the source node, then find the shortest paths to all other nodes, and form a shortest-path tree. It is optimal and uninformed, which means it will find the guaranteed shortest path (optimal) and it has no need to know the target node before the algorithm take place.

Dijkstra's algorithm has only one function, which is real cost value from source to each node: f(x)=g(x). It finds the optimal solution, the shortest path from the source to all other nodes by only considering real cost. The procedure is easy to understand as well: From the source node, go to the node with lowest real cost, then mark that node as visited and mark its cost. Then go to the unvisited node with lowest cost, since the cost is calculated as the real cost from the source node, so the algorithm will finish go back and force and different route to a certain node might be found. In this situation, if the new route has a lower cost, then replace that cost with the old one. This way, when all nodes are visited, all paths are cost are guaranteed to be the lowest. It saves a lot of time by not computing the estimate cost (heuristic function in A* search), but it does computes paths for unnecessary nodes to make sure that the solution is optimal.

The first thing needed for Dijkstra's algorithm is to find a variation which works in dynamic terrain since what we are looking for is a strong multi-purpose algorithm which will perform good overall. Using different algorithm for different situation will result in a heavy space complicity. Researchers form ITMO University conducted a research on the topic of Dijkstra's based algorithm for terrain generation.[9]It gives the general idea of how to generate dynamic terrain using an advanced weight functions and the same idea would apply to doing search.

UAV (unmanned aerial vehicle) are one of the areas that taking the most advantages from motion planning. A variant of Dijkstra's algorithm built for UAV by Medeiros and his colleagues using the called the Elevation-based Dijkstra Algorithm. This algorithm will find the best route for UAV from initial place to the target place. The reason we considering this algorithm is that the distance is not the only thing this algorithm taking into account. It also introduces to the idea of Visibility Graphs, which is the visible state of the UAV.

Evaluate the items inside the vision range and determine the next move will be similar to what our goal.[9]

In general, Dijkstra's algorithm could be implemented for dynamic terrains but in a trade off for its performance. There are only limited resources on this subject, possibly due to the fact that Dijkstra's algorithm won't be as powerful in such situations and A* search can easily take the place (they are similar by any means.)

4.3 Floyd-Warshall Algorithm

Floyd-Warshall algorithm is an interesting but also straightforward one. It iterative through the node 3 time. All non-directed path can be split into two parts. For example from A to C can be split into A to B, then B to C. Find the shortest path from A to be, then find the shortest path from B to C and combine them together becomes the shortest path from A to C though B. Each iteration the node pass through will increment by 1. This means, first iteration only allows direct path. Second iteration will allowed one node between the origin and the destination. The iteration will take n times (n = number of nodes).[10]

This algorithm is clear and takes small amount of space, since all it need to keep in the memory is the shortest path between two nodes from last iteration. However, it takes a pretty long time to compute since it need to iterate all the nodes 3 times. Compare to A* search, it wastes too much time on computing unnecessary nodes. The main advantage of Floyd-Warshall algorithm is simplicity, but it is not favorable by applications, applications looks at performance.

4.4 Summary

W. Zeng and R. L. Church then did a comparison between A* search algorithms and Dijkstra's algorithms.[3] The results show that Dijkstra's algorithms outperform on the speed of computation in most cases, but A* search performs better when the destination is far away from the origin. This is as expected since A* search will requires a lot of resources on computing the estimate value. In real life map application use, graphs are tend to be small due to fact that long distance normally won't requires a map (airplanes or trains). Floyd-Warshall algorithm is great but it is the time complexity is too much for application use ($O(n^3)$).

Benjamin Zhan and Charles Noon compared different shortest path finding algorithms and a few of other algorithm have some exciting performance like incremental graph and the threshold algorithm. [11] Meanwhile, various variants of all those algorithms have different performance as well.

Since most algorithms have their advantages in some situations and what we expected ours to do is solving different problems in similar way. The testing environments will be different throughout the whole process and our testing environments includes different type of search problems. Therefore the general idea is using A* search for search problems and set the heuristic to 0 (which is then becoming the Dijkstra's algorithms) for shortest path problem. D* lite is one of the best performing algorithm and it is considered to be implemented after A* if possible.[12] Due to the fact the A* search has the best fitness with Dijkstra's algorithms, we will take advantages of this and implement the best algorithm for our environments.

5 Approach

As mentioned previously, Minecraft with its build-in robot and highly customized map will be used as the stimulating environment. To find out the most efficient algorithms, an moving agent will be controlled to find a shortest path to a specific location. A* and D* Lite algorithms will be performed the agent in an edited land, like mazes which would be both downloaded from others' works and self made maps. Detailed discussion on testing environment will be in the next section. The general idea of implementing search algorithm in the Minecraft will be through a bot program. A bot program operates a controlled character that doing easy jobs set up for the player to achieve certain goals. For the best practice of our own implementation, bot program with build in algorithm will be considered first. In fact, due to the popularity of the game is reduced compare to earlier years and the way bot programs running is different nowadays, it is hard to find such a bot that suit our purpose. 3 different bot programs are selected to be used. Phase Bot, AI Bot, and Torch Bot. They are all open source bot programs distributed at Github.

The first bot program being tested is Phase Bot[13], it is the most powerful bot and it self is a simulation of the Minecraft client. It is written in assembly-like language, and with a combination of Java and Rust to build and run. Since Minecraft is written in Java, therefore it will be easier to understand and interact with when building tools using the same language. The other reason we choose this bot program is it have built-in A* search path finding algorithm. However, for simple purposes only such as point to point path finding. Due to the unfamiliarity with Java and Gradle running on Windows machines and Rust programming language, the testing of the bot been struggle. The Phase Bot was built and tested by the original author 3 years ago, even though Minecraft allows to run older versions, but different version of Java becomes an issue at the end. However the implementation of A* search is an important aid to our project.

The A* search implementation (with our modification) is using a hash-map to store the actual value with a named location. Since Minecraft using a three-dimensional coordinate location system associated with floating point numbers to indicate certain location. For example, the center of the world is located at some point with a coordinate of (0, 0, 0), straight below is (0, 0, -xx). Have access to this coordinate allow the players to position themselves, and made it easy for us to make a estimate distance value. For the A* search, the heuristics (estimated value)is using the straight point to point calculation based on Euler–Lagrange equation: $AB = \sqrt{(x_2 - x_1)^2 + (y_2 - y_1)^2 + (z_2 - z_1)^2}$

It allows us two things: first we don't need to store it but the program will have access to

the heuristics value at anytime; also this make the heuristics consistent (by means it is also admissible). By the definition, for every node N and each successor P of N, the estimated cost of reaching the goal from N is no greater than the step cost of getting to P plus the estimated cost of reaching the goal from P. That is: h(N) ¡= c(N,P) + h(P) and h(G) = 0. Consistent heuristics means that all the path-finding through this implementation of A* search will guarantees that the result is optimal. Heuristics function is the core of A* search and a good heuristics will make a optimal output with easy algorithm to calculate. Dynamic Weighting from Prof. Pohl [14] uses the function $f(n) = g(n) + (1 + \varepsilon w(n))h(n)$
$w(n) = \begin{cases} 1 - \frac{d(n)}{N} & d(n) \leq N \\ 0 & \text{otherwise} \end{cases}$ and d(n) is the depth of the search and N is the anticipated length of the solution path was our original thought on the cost function before.

Each potential path is calculated and stored as a linked list, a linked list node is a location point in the Minecraft world where the coordinate is known and the cost can be calculated. A linked list is best for this job since it's simply chained nodes. There is no need to find the index of each node as long as they are ordered, which is a must for the bot to take the path. Calculate the cost from one to the next, then move on to the next. Linked list's limitation also prevent any potential error like mixing orders or jumping between different node. All the different path will the be compared in an array list. The cost will then be compared in a loop.

At this point, the general approach is as follows:

- 1. Giving the start location and target location.
- 2. Find all the path lead to the target location and store them in linked lists.
- 3. Store all the linked list in an array list.
- 4. Loop through all the linked list in the array.

- 5. For each linked list, calculate the total cost by adding up the cost node to node from the star location to the target location.
- 6.Compare all the total costs and give the best output.
- 7. In dynamic terrains, repeat this cycle every a new node (block or not) is found.

However, numbers of support functions are required to help. Problems at this stages and some possible solutions to them:

- Minecraft is open world sandbox game, you can always find a new way to the target location, simply walk to the right one more block and come back. As long as the location is known then it is a possible solution to the problem. The way to limit this is using the cost function when selecting paths. When a path's cost is high enough, do not consider it. However, situations like going around a mountain or solve a maze will requires path much longer than it estimates value. Limit the node being visit is another approach, try to force the paths not to duplicate from each other.
- Walkable block and non-walkable block. Even though in Minecraft, most blocks are walk-able and there is a way to go to everywhere: make a way using the blocks on you own, but it is against our purpose. Those will serve the purpose as a roadblock, those will include: lava blocks, two blocks highers than the current walking horizon level (the jumping for Minecraft is one block) and other potential places cannot be crossed. Helper functions Can-Continue will give the result of any roadblocks appear in a path.
- Target might not be a location. Although this can be an later on added function, but it seems reasonable to target a animal or another player in the Minecraft, since those object are move-able, the whole become dynamic.

The final way we used to implementing the algorithms is using the AI Bot software [15] since it is more straightforward, and then implemented our search algorithm as a plug-in function. The AI Bot runs as another player therefore a Minecraft server is required, it act like another player joining the server with IP address and password, then controlled through the chat line. The algorithm implementation are as discussed with a few other helper functions on Minecraft world physics.

6 Experiment design and results

This section will be discussing different testing environment build inside Minecraft world to test and compare the algorithms implemented, along with the results of the tests.

6.1 Experiment Setup and Goal

The goal of the experiment is to compare the performance of the two algorithms, A* and D* Lite in both a closed form maze which includes mixture of small and large mazes and an open environment with moving objects. Since D* Lite was proven runs faster than A* so the results of the experiment will be used to find the ratio of the run-time between the two and which environment is more suitable for each. First, to reduce the bias from the experiment and also considering the time cost of the process, the moving speed of the bot has been set to a maximized value.

50 rounds for each algorithm will be tested on randomly chosen sizes of a maze . A cutoff point of a big/small maze will be set as 20*20 grid in Minecraft world: If a maze have a side built more than 20 units than it would be treated as a big maze and if a side built less than 20 units, it will be treated as a small maze.

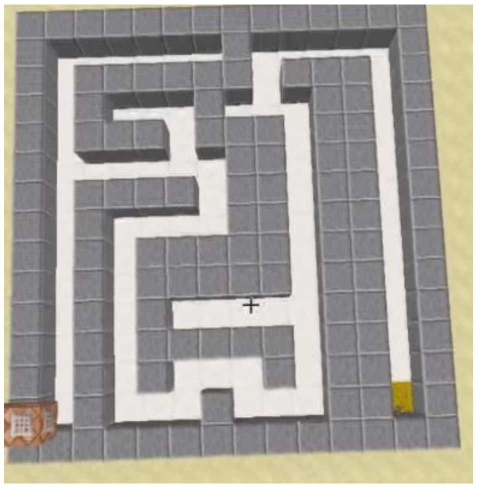

Figure 1: An example of small maze we used in the experiment

For the open environment, 10 rounds for each algorithm will be executed and constant amount of moving object will be put on the environment. Because unlike the maze, there are less elements can be controlled in open environment so the number of rounds are reduced to 10. Additionally, an dynamic maze would also be included in the study, but since the building process is too complex, an existing one (show in Figure 2) downloading from others' workshop will be used in the experiment [16] The maze was built with blocks which can randomly pop up and drop down, which makes some area accessible that originally does not. Under this environment, the test will also run for 10 rounds and be compared with the other two settings.

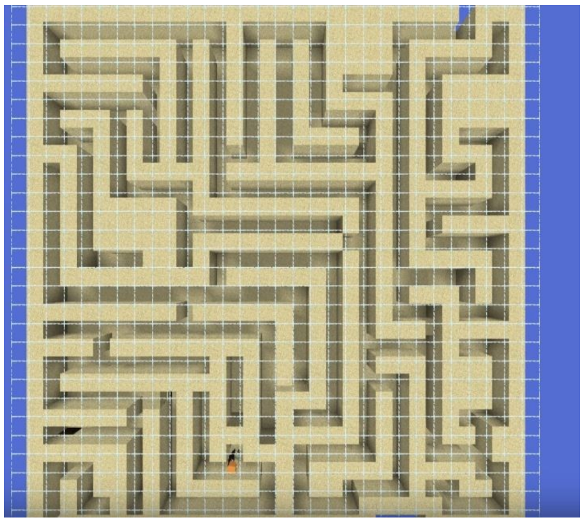

Figure 2: Dynamic maze

For each time an algorithm runs, the total distance of finding the goal and average run-time will be recorded. The initial and final time and distance will be marked and the averaging calculation for accuracy purpose will be done with program.

6.2 Experimental Results

After 120 iterations of implementations on three test environment with two path-finding algorithms, the results were listed in the following tables. Table 1 shows how A* and D*Lite performed on randomly selected mazes, a mixture of a 15*15 grid and 40*40 grid. Table 2 shows their performance on an open environment with moving obstacles and table 3 is on a dynamic maze.

Executing on Static Mazes	
Algorithms being tested	Average Run-Time Ratio
A*	3.5
D* Lite	1

Table 1: Performance ratios wrt. to D* Lite

Executing on the Open Environment with Moving Object	
Algorithms being tested	Average Run-Time Ratio
A*	5.9
D* Lite	1

Table 2: Performance ratios wrt. to D* Lite

Executing on the Dynamic Maze	
Algorithms being tested	Average Run-Time Ratio
A*	7.2
D* Lite	1

Table 3: Performance ratios wrt. to D* Lite

7 Analysis

The performances of the two algorithms were tested on three different situations which includes open environment, five different mazes and one dynamic maze. The run-times were recorded in seconds and the results were averaged and made into ratio form in order to see the comparison more clearly.

Given the results from the tables above, the ratio for the three setting were respectively 1:3.5, 1:5.9 and 1:7.2. It's obvious that D* lite runs faster than A* does on every kind of settings. It's reasonable since theoretically even A* and D* Lite both store the information when the bot meets obstacles, but A* throws away all the calculated information while D* Lite doesn't, which makes it run much slower than D* Lite. Moreover, the differences between the two enlarged when the mazes became bigger and also the environment turned from static to dynamic. To make a more accurate decision on the ratio in dynamic environment, 30 more rounds were performed on the setting 3, the dynamic maze. Since in the last 10 rounds, some dead ends appeared to stuck the bot, and in this case we just deleted the data from the records. As a result, we have nearly 1:7 in every round which was a huge advantage of

D* Lite. So for most scenario in real life, D* Lite is definitely a better choice than A*. But still, A* performed relatively better on small size static maze compared with itself, so for convenience A* can be used in small range searching.

8 Conclusion and future work

Path-finding problems are the quintessential problem in artificial intelligence and two main algorithms A* and D* Lite were compared in this paper. The purpose was to find under what situation the two algorithms can be more efficient and useful. In this paper, three scenarios have been discussed: static mazes, dynamic mazes and open environments with moving objects, since in real word the scenes you wanted to simulate are always dynamic and mostly with changing information. However, from pre-knowledge that A*, which is very similar to Dijkstra, can't actually increase the run-time when there's no information about the place while D* Lite is designed for the partially known terrain situation. And this also proved in the experiment result, D* Lite outperformed nearly 7 times better than A* in dynamic environment, so D* Lite will always be a better choice among the two. However, since A* is easier to code and understand, it can be used in some small range and static searching environment.

In future study, some improved A* algorithms like AA*, MPAA* and repeated A* can be applied to compare with D* Lite, since interestingly these methods have been proved that is faster than D* Lite in some settings. And also non-solid target path-finding is a practical research direction so it will be interesting to apply the existing algorithm to solve this kind of question.

9 Contributions

Most literature research sections of this project is done under cooperation of all team members.

Following sections are lead by the team member indicated, however, with contributions from all members of the team:

Approach section: Cyrus Chen

Experiment design and analysis section: Haiwen Zhou

References

[1] E. W. Dijkstra. A note on two problems in connexion with graphs. *Numerische Mathematik*, 1(1):269–271, Dec 1959.

[2] R. W. Floyd. Algorithm 97: Shortest path. *Commun. ACM*, 5(6):345–, June 1962.

[3] I. Found. Available at: http://www.mediafire.com/file/krr96z4hb1pf6kk/Randomised+Maze.zip.

[4] J. Fowler. Available at: https://github.com/CarbonNeuron/AIBot.

[5] C. Hernández, R. Asín, and J. A. Baier. Reusing previously found a* paths for fast goal-directed navigation in dynamic terrain. In *AAAI*, pages 1158–1164, 2015.

[6] C. Hernández, J. A. Baier, and R. Asín. Making a* run faster than d*-lite for path-planning in partially known terrain. In *ICAPS*, 2014.

[7] C. Hernández, T. Uras, S. Koenig, J. A. Baier, X. Sun, and P. Meseguer. Reusing cost-minimal paths for goal-directed navigation in partially known terrains. *Autonomous Agents and Multi-Agent Systems*, 29(5):850–895, 2015.

[8] S. Koenig and M. Likhachev. Fast replanning for navigation in unknown terrain. *IEEE Transactions on Robotics*, 21(3):354–363, 2005.

[9] S. Koenig, C. Tovey, and Y. Smirnov. Performance bounds for planning in unknown terrain. *Artificial Intelligence*, 147(1-2):253–279, 2003.

[10] S. K. M. L. Y. Liu and D. Furcy. Incremental heuristic search in artificial intelligence.

[11] F. L. L. Medeiros and J. D. S. D. Silva. A dijkstra algorithm for fixed-wing uav motion planning based on terrain elevation. *Advances in Artificial Intelligence – SBIA 2010 Lecture Notes in Computer Science*, page 213–222, 2010.

[12] C. Neuron. Available at: https://github.com/phase/phasebot.

[13] I. Pohl. The avoidance of (relative) catastrophe, heuristic competence, genuine dynamic weighting and computational issues in heuristic problem solving. In *Proceedings of the 3rd International Joint Conference on Artificial Intelligence*, IJCAI'73, pages 12–17, San Francisco, CA, USA, 1973. Morgan Kaufmann Publishers Inc.

[14] M. Przybylski and B. Putz. D* extra lite: A dynamic a* with search–tree cutting and frontier–gap repairing. *International Journal of Applied Mathematics and Computer Science*, 27(2):273–290, 2017.

[15] W. Zeng and R. L. Church. Finding shortest paths on real road networks: the case for a*. *International Journal of Geographical Information Science*, 23(4):531–543, 2009.

[16] F. B. Zhan and C. E. Noon. Shortest path algorithms: An evaluation using real road networks. *Transportation Science*, 32(1):65–73, 1998.