Research of the Path Finding Algorithm A* in Video Games

Daohong Liu *

Experimental High School Attached to Beijing Normal University, Beijing, China

* Corresponding author email: liudaohong2024@outlook.com

Abstract. Pathfinding is an essential component of many video games. This review provides a simple overview of the classical pathfinding algorithm A*, including a brief summary of the A* algorithm, as well as a few optimizations and applications of A* in video games. In the gaming industry, A* is a widely used pathfinding algorithm. Based on breadth first search and heuristic search, A* uses a heuristic function to quickly find the shortest path in a game map. A common optimization of A* is to use a binary heap to save the open queue, a variety of heuristic functions are also commonly used to increase the accuracy and speed of A*. This paper summarizes work done on the A* algorithm. The goal of this paper is to give researchers and developers a simple overview of the A* algorithm and its applications in video games.

Keywords: Path Finding; Games; Search; A* Algorithm; Video Games.

1. Introduction

With the increasing popularity of video games, more and more games are starting to use artificial intelligence to improve their gameplay. In video games, there is often a main character that the player controls directly, while all the other characters (NPCs) must be controlled by AI. Therefore, the majority of games require a pathfinding algorithm that can efficiently compute an acceptable path from point A to point B for NPCs to traverse in the game map. Despite the progress seen in recent years, pathfinding still presents challenges to game developers. Pathfinding algorithms are given system resources. Routes must be calculated in real time, and there are often a large number of moving units to account for. Paths must also look sufficiently realistic to players.

At the core of the pathfinding process is the search algorithm. The most well-known and widely used search algorithm is no doubt the A* algorithm [1]. With a search tree developing from the initial vertex till the ending vertex is located, A*employs a heuristic function to explore a search network in the best-first way [2].

The A* algorithm is a heuristic search algorithm based on breadth first search, generally using the heuristic function f(n). A* uses the node with the smallest f(n) value as the next node in the path in each cycle, and thus efficiently and correctly calculates the shortest path between two nodes.

This paper gives a brief summary of A*, it's optimizations and its applications. The author's intention is to summarize research on the A* algorithm, and give researchers and developers unfamiliar with pathfinding an overview of A*. In the next section, we introduce regular state-space search, and discuss its differences with heuristic search, before describing the A* algorithm in Section 3. Section 4 presents the possibility of using a binary heap to optimize A*. Section 5 describes some alternate heuristic, and Sections 6 discusses the application of A* in video games.

2. Regular State-Space Search

Depth first search utilizes the mathematical idea of recursion. The procedure is as follows (as shown in Figure 1): First access vertex a, the first vertex of the graph. Then access the vertex b, which is adjacent to a. Then access c which is adjacent to b, …, until there are no adjacent vertices when m is reached. Then return to the previous vertex c and continue. The entire process is completed with recursion [3]. When all vertices have been accessed, the graph search is complete. The access order for the vertices in the graph below using depth first search is a-b-c-n-m-d-e-f-g-h.

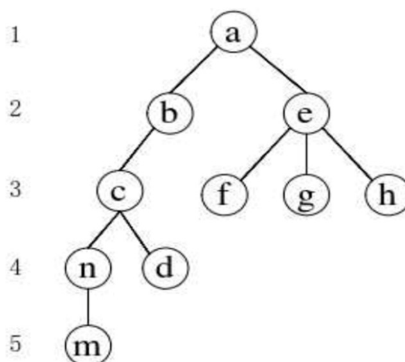

Fig 1. The sketch map of tree structure

Breadth first search uses a queue to manage accessing order. The process is as follows: Take the first vertex a, then add the vertices b and e which are adjacent to an into the queue [4]. Access vertex a, then pop out the first vertex in the queue b, put the vertex c which is adjacent to b into the queue, the items in the queue are now (e, c). After b is accessed, take out the first item in the queue, which is vertex e. Push the f, g, h which are adjacent to e into the queue …, repeat until the queue is empty. The access order for the vertices in the graph in Figure 1 using breadth first search is a-b-e-c-f-g-h-n-d-m.

When the state space is small, BFS and DFS are excellent choices for pathfinding. However, when the state space is large, the number of nodes that need to be searched through increases rapidly, resulting in extreme inefficiency. This is why heuristic searching is required. Heuristic searching evaluates each position to be searched with a heuristic function, eliminating the majority of the redundant searching process, thereby greatly increasing efficiency.

3. The A* Search Algorithm

Heuristic search algorithms include local optimal search and best first search, among others. These algorithms all use heuristic functions, but they differ in the strategy they use in selecting an optimal node. Local optimal search selects only the locally optimal node while ignoring all the others, and continues its search. The disadvantage of this method is obvious, as the locally optimal node may not be the globally optimal one, this method has a high chance of resulting in an incorrect path. With best first search, no nodes are discarded (unless they are dead ends). In each step, this method compares the heuristic estimate of the current node with the estimates from the nodes before, thus ensuring that the nodes in the optimal path are not discarded.

A* is a best first search algorithm with additional constraints, used to compute the shortest path from point a to point b in a two-dimensional grid. A problem must be abstracted into a two-dimensional grid before A* can be used on it.

Given a two-dimensional grid, with A as the starting node and B as the ending node, the process of A* is as follows [5]:

1) Starting from point A, we treat it as a node to be processed, and save it in the "open" queue. The "open" queue is a queue of nodes that have not yet been searched, but will be searched next. There is only one node in it currently, but it will eventually contain many more. The optimal path may pass through a node in the "open" queue (or not).
2) Search through the reachable or traversable nodes surrounding A, skip the non-traversable ones. Add these nodes to the "open" queue. Save A as their "parent node".
3) Delete A from the "open" queue, and add it into the "close" queue. The "close" queue saves all the nodes that don't need to be searched through again. Then choose the node with the lowest heuristic estimate in the "close" set. Every heuristic search algorithm has a heuristic function. The heuristic function of A* is $f(n)=g(n)+h(n)$, where $g(n)$ is the cost of traveling from the starting node to the current node and $h(n)$ is the estimate value of the cost to travel

from the current node to the target node. To continue searching, we select the node in "open" with the lowest f(n) value, then complete the following steps.

4) Find the node with the lowest f(n) value and delete it from "open", then add it to "close".
5) Check all adjacent nodes, if they are not yet in the "open" queue, then add them to the "open" queue. Save these nodes as child nodes of the current node. Skip the nodes that are already in the "close" set or aren't traversable.
6) If an adjacent node is already in the "open" queue, check if the route that passes through that node is better. In other words, check if g(n) is lower if the new route is taken. If it is, then queue the parent node of the new node as the current node. Repeating these steps, the values of f(n) will be updated one by one, and new nodes will be added into the "close" set, until B is added. If we plot a route starting from B and with each node connecting to its parent node, A will be reached. Such a route is the shortest path from A to B. If B is never added to the "close" set, then B is not reachable from A.

Figure 2 is a flowchart of this process.

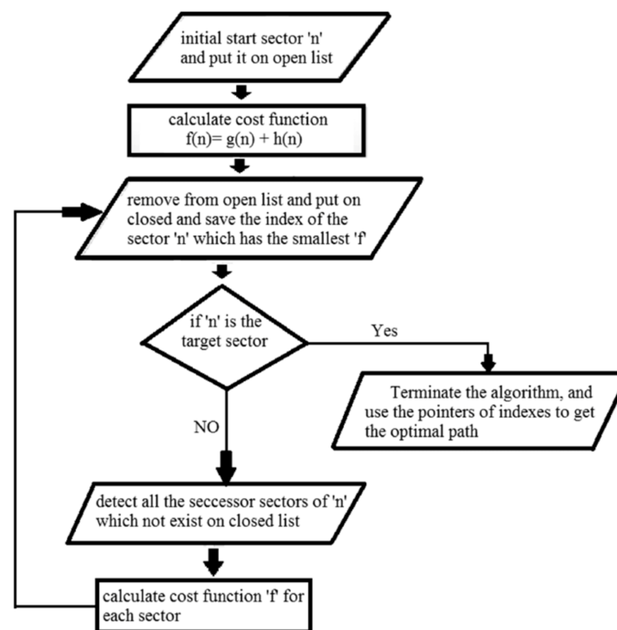

Fig 2. Flowchart of A* algorithm [9]

4. Binary Heap Optimization

The slowest part of A* is when the algorithm searches through the "open" queue to find the node with smallest value f(n). The time required depends on the game map size. There could dozens, or millions of nodes. When using A*, repeatedly searching through this queue can slow down A* significantly. However, the time required to search through open largely depends on data structure open is saved in. In most circumstances, the best data structure for this purpose is the binary heap [3, 4].

Binary heap is a special kind of heap. Its structure is similar to a complete binary tree. Binary heap satisfies the characteristic of heaps: the value of the parent node is always smaller or equal to (or larger or equal to) the value of any child node. The child nodes in a binary heap are also binary heaps. When parent nodes are larger or equal to child nodes, the heap is a large root heap; when the parent nodes are smaller or equal to child nodes, the heap is a small root heap. In A*, we are searching for the node with the smallest value of f(n), therefore a small root heap is used. It is possible to add values into a binary heap as well as take values from it while maintaining its integrity in O (log n) time, this is what makes the binary heap ideal for storing the open queue.

In average, an A* algorithm optimized with binary heap runs 2-3 times faster than a regular A* on small datasets. On larger datasets, the optimized A* is orders of magnitude faster.

5. Alternative Heuristics

In A*, heuristic functions that return a value that is smaller or equal to the true distance to a goal are admissible. Manhattan distance is a well-known heuristic on 4 connected maps (maps with nodes that only connect to nodes in the 4 cardinal directions). With two nodes a and b with coordinates (x_1, y_1) and (x_2, y_2), the Manhattan distance between a and b is defined as $\Delta x + \Delta y$. The Octile distance is another well-known heuristic on 8 connected maps. It is defined as $\sqrt{2} \times m + (M - m)$, where $m = min(\Delta x, \Delta y)$ and $M = max(\Delta x, \Delta y)$.

Goldberg and Harrelson [5] developed an admissible heuristic called the ALT heuristic that precomputes a table with true distance d (n, l) for every node n to a given node l, called a landmark. With the two nodes a and b, |d (a, l) – d (b, l) | is a viable heuristic for the distance between a and b. This heuristic uses multiple landmarks and returns the largest estimated distance throughout the landmarks.

A variation of the ALT heuristic known as ALTBestP employs the one landmark that consistently calculates the greatest distance between a and b. The biggest of the two distances between the heuristic and the Manhattan distance is returned as the heuristic evaluation [6].

Manhattan might not be as informative as these two heuristics. For instance, these algorithms could yield +∞ when there is no path connecting a and b, whereas Manhattan will return a finite value. The gateway heuristic and the dead-end heuristic were first introduced by Björnsson and Halldórsson [7]. The map is divided into jumbled regions using the dead end heuristic. It gives all nodes in such scattered locations a value of +∞. A table containing the actual distances between pairs of gates is pre-computed by the gateway heuristic, which detects gateway locations between regions on the game map. Thus, better heuristic accuracy is exchanged for more memory, equal to quadratic the number of gateways. There are other comparable strategies that sacrifice more memory for better accuracy.

6. Application of A* in Video Games

It is a common practice to divide the game map into a large number of tiny squares, we use a two-dimensional array to represent these squares, with the positions of these squares represented by the indexes of the array. We assign 0 to traversable square and 1 to non-traversable ones. Then we turn the positions of NPCs and their destinations into the indexes of the squares they are in. Applying A* to this array will yield a path. The indexes in the path can then be converted into coordinates on the game map. The NPCs can then move to each of these coordinates in turn, forming a complete route. However, if this is done the NPCs will move through the corners of obstacles, this is obviously not realistic [8]. This is where the heuristic function comes in, if the square in a diagonal direction is an obstacle, the g(n) value of this square must be raised.

The requirements of interactive games often impact the possibility of calculating an optimal path. Compared to computing the optimal path, it may be more important to compute a viable one. Most of the time, it is more important to calculate the portion of the path closer to the starting point than the part close to the destination, as the former needs to be executed immediately. Even if it is along a sub-optimal path, a unit should still be moved as soon as possible. In real time games, the time A* takes are often more important than its accuracy [9].

A unit can either move in a simple manner (such as directly towards the direction of its destination) or move along a pre-computed path. It is often unnecessary to compute the entire path for all units, instead, it is possible to compute a portion of a unit's path every game loop. And then allow the units to move in a simple manner, and continue calculating the path in the next game loop. This way, the time needed to compute an entire path can be separated into many parts, instead of having to compute the whole path all at once.

Generally, A* will exit the main loop after it reaches the target node, but it can also exit prematurely, resulting in a portion of the path. If A* is paused at any time before it reaches the target node, it will return the node with the lowest f(n) in the "open" queue, this node is most likely to reach the destination, so this is reasonable path.

Similar circumstances to the above include: A sufficient number of nodes have already been searched; A* has been running for several milliseconds; the current node is a sufficient distance away from the starting node. Sections of the path can be joined together [10].

Need for pathfinding is not evenly distributed across time. In strategy games, it is a common occurrence for players to select several units at once and order them to move towards a single target all at once. This creates a great deal of strain for pathfinding systems. Observing that the path found for one unit may also be useful for other units is one method to approach this issue. We can find the path from the central point of the starting nodes to the central point of the targets P, then apply the majority of P for all the units, only replacing the first 10 steps and the last 10 steps with individually computed routes. The path that a unit receives is: from the starting point to P [10], then is the shared route P [10 ... len(P)-10], and then the route from P[len(P)-10] to the target node.

This way, the path computed for each unit is very short (around 10 steps), the longest part of the route is shared between all the units, and only needs to be calculated once. The problem with this approach is that if all the units move in exactly the same route, the effect might be unrealistic. This issue can be resolved by allowing the units to take brief, random detours, making slight changes to their paths. Allowing the units to function as a group is another technique to address this issue (we may choose a "leader" unit), and compute a single route for the leader, then use swarm algorithms to make all units move as a whole.

Some variants of A* can be used to process moving targets, or to gradually deepen its understanding of the target. Some of those suitable for computing situations where multiple units converge on a single target turn A* upside down (they attempt to compute a route from the destination to the beginning).

7. Conclusion

This paper has reviewed work done on the A* algorithm, including a description of the principles of the A* algorithm and an explanation of binary heap optimization, as well as a few alternate heuristics and details in application. While the searching process of A* does unavoidably use exhaustive search, it greatly reduces the scale of the search through the use of a heuristic function, greatly increasing searching efficiency. As a basic algorithm in artificial intelligence, A* greatly enriches video games, increasing their playability. A* also has extensive applications in robotics and other aspects of computer science.

References

[1] Y. Björnsson, M. Enzenberger, R. Holte, and J. Schaeffer. Fringe Search: Beating A* at Pathfinding on Game Maps. In China Internet Gaming, 2005:125–132.

[2] P. Hart, N. Nilsson, and B. Raphael. A formal basis for the heuristic determination of minimum cost paths. In Computer Science 1972 4(2):100–107.

[3] Yao Yu, LI Qing, Chen Xi. Optimization of the Application of A * Algorithm in Path Planning. Microelectronics & Computer, 2017, 34(7): 51-55.

[4] X. Guo and P. Guo, A* Algorithm Analysis and Optimization: In Network Game Design, Computer and Information Science and Engineering, 2009:1-4.

[5] A. V. Goldberg and C. Harrelson. Computing the Shortest Path: A* Search Meets Graph Theory. In ACM-SIAM Symposium on Discrete Algorithms, 2005 156–165.

[6] Tristan Cazenave. Optimizations of data structures, heuristics and algorithms for path-finding on maps. In China Internet Gaming, 2006, 27–33.

[7] Yngvi Björnsson and Kári Halldórsson. Improved heuristics for optimal path-finding on game maps. In conference on artificial intelligence and interactive digital entertainment, 2006: 9–14.

[8] N. Sturtevant, A. Felner, M. Barer, J. Schaeffer, and N. Burch. Memory-based heuristics for explicit state spaces. In International Joint Conference on Artificial Intelligence, 2009, 609–614.

[9] Wavefront and A-Star Algorithms for Mobile Robot Path Planning. Available from: https:// www. ResearchGate. net/figure/Flow-chart-of-A-star-algorithm_fig1_319404402

[10] Cui, Xiao & Shi, Hao. A*-based Pathfinding in Modern Computer Games. International journal of computer science and network security, 2011,11(1):125-130.