A*-based Pathfinding in Modern Computer Games

Article · November 2010

CITATIONS
209

READS
58,062

2 authors:

Xiao Cui
Victoria University
3 PUBLICATIONS **242** CITATIONS
SEE PROFILE

Hao Shi
SOAS University of London
31 PUBLICATIONS **505** CITATIONS
SEE PROFILE

A*-based Pathfinding in Modern Computer Games

Xiao Cui and Hao Shi

School of Engineering and Science, Victoria University, Melbourne, Australia

Summary
Pathfinding in computer games has been investigated for many years. It is probably the most popular but frustrating game artificial intelligence (AI) problem in game industry. Various search algorithms, such as Dijkstra's algorithm, bread first search algorithm and depth first search algorithm, were created to solve the shortest path problem until the emergence of A* algorithm as a provably optimal solution for pathfinding. Since it was created, it has successfully attracted attention of thousands of researchers to put effort into it. A long list of A*-based algorithms and techniques were generated. This paper reviews a number of popular A*-based algorithms and techniques from different perspectives. It aims to explore the relationship between various A*-based algorithms. In the first section, an overview of pathfinding is presented. Then, the details of A* algorithm are addressed as a basis of delivering a number of optimization techniques from different angles. Finally, a number of real examples of how the pathfinding techniques are used in real games are given and a conclusion is drawn.

Key words: Pathfinding, A*, A* optimization, Computer game

1. Introduction

Pathfinding generally refers to find the shortest route between two end points. Examples of such problems include transit planning, telephone traffic routing, maze navigation and robot path planning. As the importance of game industry increases, pathfinding has become a popular and frustrating problem in game industry. Games like role-playing games and real-time strategy games often have characters sent on missions from their current location to a predetermined or player determined destination. The most common issue of pathfinding in a video game is how to avoid obstacles cleverly and seek out the most efficient path over different terrain.

Early solutions to the problem of pathfinding in computer games, such as depth first search, iterative deepening, breadth first search, Dijkstra's algorithm, best first search, A* algorithm, and iterative deepening A*, were soon overwhelmed by the sheer exponential growth in the complexity of the game. More efficient solutions are required so as to be able to solve pathfinding problems on a more complex environment with limited time and resources.

Because of the huge success of A* algorithm in path finding [1], many researchers are pinning their hopes on speeding up A* so as to satisfy the changing needs of the game. Considerable effort has been made to optimize this algorithm over the past decades and dozens of revised algorithms have been introduced successfully. Examples of such optimizations include improving heuristic methods, optimizing map representations, introducing new data structures and reducing memory requirements. The next section provides an overview of A* techniques which are widely used in current game industry.

2. A* algorithm

A* is a generic search algorithm that can be used to find solutions for many problems, pathfinding just being one of them. For pathfinding, A* algorithm repeatedly examines the most promising unexplored location it has seen. When a location is explored, the algorithm is finished if that location is the goal; otherwise, it makes note of all that location's neighbors for further exploration. A* is probably the most popular path finding algorithm in game AI (Artificial Intelligence) [2].

```
1.  Add the starting node to the open list.
2.  Repeat the following steps:
    a.  Look for the node which has the lowest
        f on the open list. Refer to this node
        as the current node.
    b.  Switch it to the closed list.
    c.  For each reachable node from the current
        node
         i.  If it is on the closed list, ignore
             it.
        ii.  If it isn't on the open list, add it
             to the open list. Make the current
             node the parent of this node. Record
             the f, g, and h value of this node.
       iii.  If it is on the open list already,
             check to see if this is a better
             path. If so, change its parent to the
             current node, and recalculate the f
             and g value.
    d.  Stop when
         i.  Add the target node to the closed
             list.
        ii.  Fail to find the target node, and the
             open list is empty.
3.  Tracing backwards from the target node to the
    starting node. That is your path.
```

Fig. 1 Pseudocode of A* [3].

In the standard terminology used when talking about A*, *g(n)* represents the exact cost from starting point to any

Manuscript received January 5, 2011
Manuscript revised January 20, 2011

point *n*, *h(n)* represents the estimated cost from point *n* to the destination, and *f(n)=g(n)+h(n)*. Fig. 1 lays out the algorithm step-by-step.

A* has several useful properties which have been proved by Hart, Nilsson and Raphael in 1968 [4]. First, A* is guaranteed to find a path from the start to the goal if there exists a path. And it is optimal if *h(n)* is an admissible heuristic, which means *h(n)* is always less than or equal to the actual cheapest path cost from n to the goal. The third property of A* is that it makes the most efficient use of the heuristic. That is, no search method which uses the same heuristic function to find an optimal path examines fewer nodes than A*.

Although A* is the most popular choice for pathfinding in computer games, how to apply it in a computer game depends on the nature of the game and its internal representation of the world. For example, in a rectangular grid of 1000×1000 squares, there are 1 million possible squares to search. To find a path in that kind of map simply takes a lot of work. Thus, reducing the search space may significantly speed up A*. Several optimizations are discussed in Section 3.

3. A* Optimizations

The following sub-sections discuss several potential optimizations of A* from four different perspectives and reviews some popular A*-based algorithms.

3.1 Search Space

In any game environment, AI characters need to use an underlying data structure – a search space representation – to plan a path to any given destination. Finding the most appropriate data structure to represent the search space for the game world is absolutely critical to achieving realistic-looking movement and acceptable pathfinding performance. As you can see in the above example, a simpler search space will mean that A* has less work to do, and less work will allow the algorithm to run faster. Examples of such representations include rectangular grid (Fig. 2a), quadtree (Fig. 2c), convex polygons (Fig. 2d), points of visibility (Fig. 2e), and generalized cylinders (Fig. 2f).

The following sub-sections review two popular A*-based algorithms which optimize A* algorithm by reducing the search space.

3.1.1 Hierarchical Pathfinding A* (HPA*)

Hierarchical pathfinding is an extremely powerful technique that speeds up the pathfinding process. The complexity of the problem can be reduced by breaking up the world hierarchically. Consider the problem of travelling from Los Angeles to Toronto. Given a detailed roadmap of North America, showing all roads annotated with driving distances, an A* implementation can compute the optimal travel route but this might be an expensive computation because of the sheer size of the roadmap. However, a hierarchical path finding would never work at such a low level of detail. Using abstraction can quickly find a route. The problem described above might be solved more efficiently by planning a large-scale route at the city level first and then planning the inter routes at each city passing through.

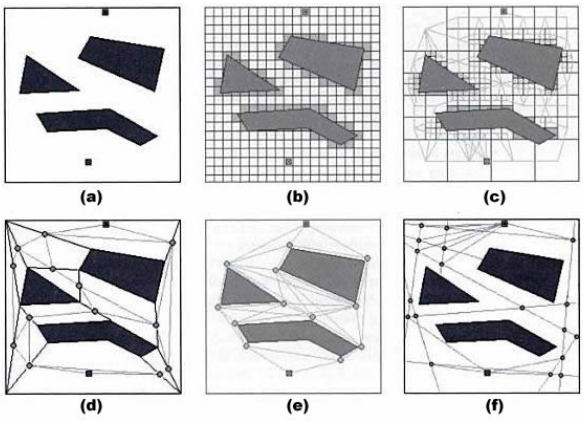


Fig. 2 Five ways to represent search space [5].

A much faster A*-based search algorithm giving nearly solutions named HPA* is described in [6]. This is a domain-independent approach. The hierarchy can be extended to more than two levels, making it more scalable for large problem spaces. A three-step process is applied. The first step is to travel to the border of the neighborhood that contains the start location. Then, the second step is to search for a path from the border of the start neighborhood to the border of the goal neighborhood. This step is done at an abstract level, where search is simpler and faster. The last step is to complete the path by travelling from the border of the goal neighborhood to the goal position. HPA* has been proved that it is 10 times faster than a low-level A* in [6]. The potential problem of this technique is that the cost increases significantly when adding a new abstraction layer.

3.1.2 Navigation Mesh (NavMesh)

NavMesh is another popular technique for AI pathfinding in 3D worlds. A NavMesh is a set of convex polygons that describe the “walkable” surface of a 3D environment. It is a simple, highly intuitive floor plan that AI characters can use for navigation and pathfinding in the game world.

Fig. 3b shows an example of NavMesh. A character moves from the starting point in pol2 to the desired destination in *pol4*. In this case, the starting point is not

in the same polygon as the desired point. Thus, the character needs to determine the next polygon it will go to. Repeat this step until both the character and the goal are located in the same polygon. Then, the character can move to the destination in a straight line.
Compared with a waypoint graph as shown in Fig. 3a, NavMesh approach is guaranteed to find a near optimal path by searching much less data. And the pathfinding behavior in a NavMesh is superior to that in a waypoint graph [7].

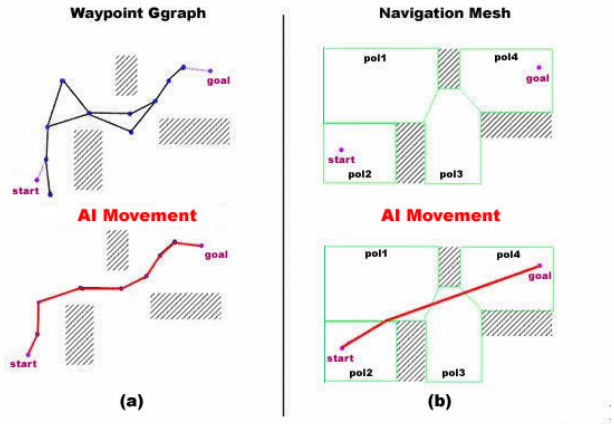


Fig. 3 Different representations of waypoint graph and NavMesh [7]

Creating navigation meshes that are highly simplified and easy for pathfinding is critical to achieving a good pathfinding. Tozour [9] describes how to construct a good navigation mesh and proposes a number of optimization techniques for navigation mesh.

3.2 Heuristic Function

The secret to the success of A* is that it extends Dijkstra's algorithm by introducing heuristic approach. Dijkstra's algorithm is guaranteed to find a shortest path in a connected weighted graph as long as none of the edges has a negative value but it is not efficient enough because all the possible states must be examined. However, A* algorithm improves the computational efficiency significantly by introducing a heuristic approach. Using a heuristic approach means, rather than an exhaustive expansion, only the states that look like better options are examined. The heuristic function used in A* algorithm is to estimate the cost from any nodes on the graph to the desired destination. If the estimated cost is exactly equal to the real cost, then only the nodes on the best path are selected and nothing else is expanded. Thus, a good heuristic function which can accurately estimate the cost may make the algorithm much quicker.
On the other hand, using a heuristic that overestimates the true cost a little usually results in a faster search with a reasonable path [10]. Fig. 4 shows the growth of the search using various heuristic costs while trying to overcome a large obstacle.
When the heuristic equals to zero (shown in Fig. 4a), A* algorithm turns to Dijkstra's algorithm. All the neighboring nodes are expanded. When the heuristic uses the Euclidean distance to the goal (shown in Fig. 4b), only the nodes that look like better options are examined. When the heuristic is overestimated a little (shown in Fig. 4c), the search pushes hard on the closest nodes to the goal. Thus, overestimating the heuristic cost a little may result in exploring much fewer nodes than non-overestimation heuristic approaches. However, how much should the cost be overestimated is a tricky problem. No general solution exists at present.

3.3 Memory

Although A* is about as good a search algorithm as you can find so far, it must be used wisely; otherwise, it might be wasteful of resources. A* algorithm requires a huge amount of memory to track the progress of each search especially when searching on large and complex environments. Reducing the required memory for pathfinding is a tricky problem in game AI. There has been a lot of work on this area.

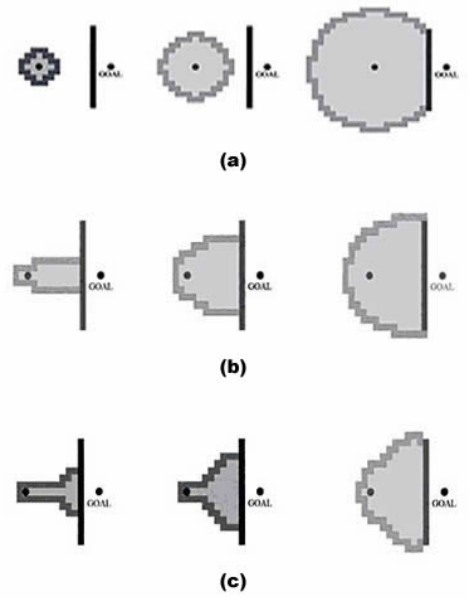


Fig. 4 Comparison between different heuristics [10].

The most popular way to avoid memory waste is to pre-allocate a minimum amount of memory [10]. The general idea is to dedicate a piece of memory (Node Bank) before A* starts execution. During the execution, if all the memory gets exhausted, create a new buffer to progress the search. The size of this buffer is allowed to change so that less memory is wasted. The size of the

minimum memory mainly depends on the complexity of the environment. Thus, tuning is required before this strategy is applied to a particular application.
Another alternative to reduce space requirements in A* is to compute the whole path in small pieces. This is the core concept behind IDA* (Iterative Deepening A*), which is a very popular variant of A* algorithm. In IDA*, a path is cut off when its total cost *f(n)* exceeds a maximum cost threshold. IDA* starts with a threshold equal to *f(start_node)*, in this case, the threshold is equal to *h(start_node)* because *g(start_node)=0*. Then, neighboring nodes are expanded until either a solution is found that scores below the threshold or no solution is found. In this case, the threshold is increased by one, and another search is triggered. The main advantage of IDA* over A* is that memory usage is significantly reduced.

3.4 Data Structure

Once a node has been initialized from the Node Bank (see Section 3.3), it needs to be put somewhere for fast retrieval. A hash table might be the best choice because it allows constant time storing and looking up of data. This hash table allows us to find out if a particular node is on the CLOSED list or the OPEN list instantaneously.
A priority queue is the best way to maintain an OPEN list. It can be implemented by a binary heap. There is little work on introducing new data structures to maintain OPEN list and CLOSED list more efficiently. Probably introducing a new data structure to store the data can help speed up A* significantly.

4. Relevant Applications in Computer Games

As a popular pathfinding algorithm in game industry, A* algorithm has been applied to a wide range of computer games. Although the algorithm itself is easy to understand, implementation in a real computer game is non-trivial. This section discusses several popular computer games in terms of pathfinding and uses a popular online game as an example to show how the different map representations can impact on the performance of pathfinding.

4.1 Pathfinding Challenge in Game Industry

Age of Empires is a classic real-time strategy game. It uses grids to represent map locations. A 256×256 grid yields 65,536 possible locations. The movement of the military unit can be simplified as if moving an object through a maze. A* algorithm is applied to Age of Empires. Although it looks perfect theoretically, many Age of Empires players are annoyed by the terrible pathfinding. An example of such problems is that when a group of units goes around forest to get to another position, half of them get stuck in the trees as shown in Fig. 5. Such situations always happen especially when the density of forest increases.

Fig. 5 A screenshot of Age of Empires II. [11]

Another strategy game Civilization V uses hexagonal tiles to represent map locations as shown in Fig. 6. A pathfinding algorithm is applied to control the military unit moving to the desired location through a group of “walkable” hexagonal tiles. Similar to Age of Empires, Civilization V still suffers with bad pathfinding although it is the latest game of Civilization series which was released in November 2010.

Fig. 6 A screenshot of Civilization V. [12]

However, compared with strategy games which involve hundreds and thousands of units simultaneously, A* works much better in first-person shooter games like Counter-Strike which only involves a few units moving around at the same time. An explanation might be that the exponential growth in the number of units moving around at the same time makes the game environment much more dynamic and it is hard to provide optimal paths for hundreds and thousands of units in real time using limited CPU and memory resources. Massively

multiplayer online is another example which involves real-time pathfinding intensively, like World of WardCraft.

4.2 Comparison between Map Representations

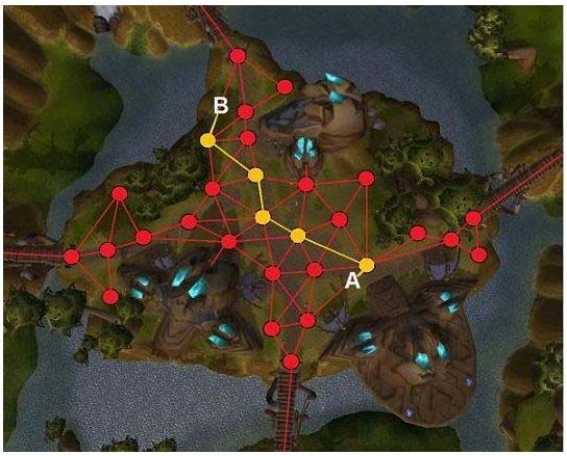
Waypoint graph is a popular technique to represent map locations. Waypoints are a set of points that refer to the coordinates in the physic space. It is designed for navigation purpose and has been applied to a wide range of areas. Most game engines support pathfinding on a waypoint graph. Although it works well for most 2D games, it requires tons of waypoints to achieve an adequate movement in a 3D game because of the complexity of the environment. Thus, a new technique called NavMesh is created. As mentioned in Section 3.1.2, NavMesh only requires a couple of polygons to represent the map. It results in a much more quickly pathfinding because less data is examined.

Five reasons why NavMesh works better than waypoint approaches in 3D games using World of WarCraft as an example are addressed by Tozour in 2008 [8]. It shows the difference between waypoints approaches and NavMesh when representing a complex 3D environment. It uses the town of Halaa in World of WarCraft as an example as shown in Fig. 7. In Fig. 7a, it uses 28 waypoints to represent the possible locations while on the other hand, as shown in Fig. 7b, only 14 convex polygons are used. The movement in Fig. 7b also acts much more like an actual human than the movement in Fig. 7a.

5. Conclusion

This paper systematically reviews several popular A*-based algorithms and techniques according to the optimization of A*. It shows a clearly relational map between A* algorithm and its variants. The core of pathfinding algorithm is only a small piece of the puzzle in game AI. The most challenge is how to use the algorithm to solve tricky problems. A* algorithm is the most popular algorithm in pathfinding. It is hard-pressed to find a better algorithm since A* is provably optimal. A lot of effort has been put into speeding it up by optimizing it from different perspectives. The ways to improve the performance of A* search include optimizing the underlying search space, reducing the memory usage, improving heuristic functions and introducing new data structures.

A potential research is to continue optimizing A* algorithm from these perspectives or to combine multiple optimization techniques into one single solution. Another way to make some contribution to the game AI community is to apply these techniques described above to the real computer games because not all of the techniques described in this paper have been widely used in current game industry. The reason why they are reviewed in this paper is that they are the hottest topics in the academic domain of pathfinding and many researchers are struggling to bring them into real games. It is expected that this research help game industry has a basic understanding about the future research direction in pathfinding.



(a) Navigating from A to B using waypoint graph.

(b) Navigating from A to B on NavMesh.

Fig. 7 Comparison between wapoing graph and NavMesh [8].

References

[1] B. Stout, “Smart moves: intelligent path-finding,” in Game Developer Magazine, pp.28-35, 1996.

[2] Stanford Theory Group, “Amit’s A* page”, http://theory.stanford.edu/~amitp/GameProgramming/AStarComparison.html, accessed October 12, 2010.

[3] N.Nilsson, Artificial Intelligence: A New Synthesis, Morgan Kaufmann Publishers, San Francisco, 1998.

[4] P. Hart, N. Nilsson, and B. Raphael, “A formal basis for the heuristic determination of minimum cost paths,” IEEE Trans.Syst.Sci.Cybernet., vol.4, no.2, pp.100-107, 1968

[5] B. Stout, “The basics of A* for path planning,” in Game Programming GEMS, pp.254-262, Charles River Meida, America, 2000.
[6] A.Botea, M.Mueller, and J.Schaeffer, “Near optimal hierarchical path-finding,” J. GD, vol.1, no.1, pp.7-28, 2004.
[7] Unreal Developer Network, “Navigation mesh reference”, http://udn.epicgames.com/Three/NavigationMeshReference.html, accessed Jan.13, 2011.
[8] Game/AI, “Fixing pathfinding once and for all”, http://www.ai-blog.net/archives/000152.html, accessed September 23, 2010.
[9] P. Tozour, “Building a near-optimal navigation mesh,” in AI Game Programming Wisdom, pp.171-185, Charles River Media, America, 2002.
[10] S. Rabin, “A* speed optimizations,” in Game Programming GEMS, pp.264-271, Charles River Media, America, 2000.
[11] Microsoft Games, “Microsoft Age of Empires”, http://www.microsoft.com/games/empires, accessed October 24, 2010.
[12] Firaxis Games, “Sid Meier’s Civilization V”, http://www.civilization5.com, accessed October 24, 2010.

Xiao Cui is a Ph.D student in School of Engineering and Science at Victoria University, Australia. He completed his master degree in the area of Software Engineering at Australian National University and obtained his Bachelor of Computer Science degree at Victoria University. His research interests include Object-Oriented Software Engineering, Pathfinding Algorithms, Database Management and Game Programming.

Hao Shi is an Associate Professor in School of Engineering and Science at Victoria University, Australia. She completed her PhD in the area of Computer Engineering at University of Wollongong and obtained her Bachelor of Engineering degree at Shanghai Jiao Tong University, China. She has been actively engaged in R&D and external consultancy activities. Her research interests include p2p Network, Location-Based Services, Web Services, Computer/Robotics Vision, Visual Communications, Internet and Multimedia Technologies.