Fringe Search vs. A* for NPC movement

Andru Putra Twinanda - 13506046

Informatics Engineering
School of Electrical dan Informatics Engineering
Institut Teknologi Bandung
Jalan Ganesa 10
e-mail: ndrewh@yahoo.com

ABSTRACT

This paper presents how artificial intelligence (AI) is used in games to solve common problems and provide game features, specifically, non-playing character (NPC) path finding. A* algorithm is a well-known algorithm to solve path finding problem. Beside A* algorithm, in this paper, Fringe Search algorithm is introduced. It is more efficient that A* algorithm. Experimental results show that Fringe Search runs roughly 10-40% faster than highly-optimized A* in many application.

Keywords: AI, NPC, path finding, A*, Fringe Search, faster.

1. INTRODUCTION

Games today are better than at any time in the past. Thanks to the latest powerful hardware platforms, artists are creating near photo-realistic environments, game designers are building finely detailed worlds, and programmers are coding effects more spectacular than ever before.

Unfortunately, this leap in graphics quality and richness of detail has not been matched by a similar increase in the sophistication and believability of artificial intelligence (AI).

Three common problems that computer games must provide a solution for are non-playing character (NPC) movement, NPC decision making, and NPC learning. Solving these problems is the responsibility of the game AI.

A game must provide a way for a NPC to move throughout the game world. When the monster is on one side of the building and the player is on the other, how does the monster negotiate a path through the building to the player? This needs to be done efficiently even when the player is constantly moving throughout the building.

Just like the red, green and blue monsters in pacman game. Those monsters have to get to the pacman, so the player will lose. To determine which path they should take, thay need an algorithm that will take them to the player character, effectively and, more important, efficiently. This is the problem of NPC movement.

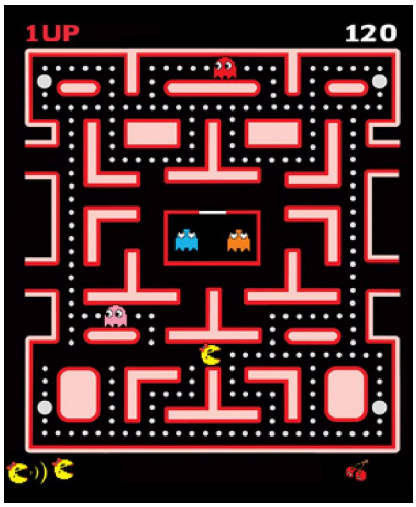

Figure 1. Ms. Pacman Screenshot.
How Could The Monsters Find Ms. Pacman?

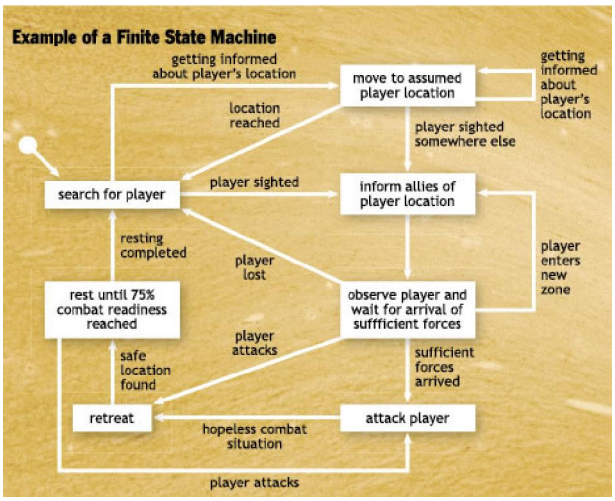

Figure 2.
State Machine of Enemy NPC's AI Movement

2. NPC MOVEMENT

The most popular algorithm to solve a pathfinding problem is the A* algorithm. However, A* algorithm has some weaknesses that make the found path not really efficient. That's the reason a new algorithm is developed, to find a better solution. It is Fringe Serach Algorithm.

2.1 NPC Movement with A* Algorithm

AI Search Methods are utilized to perform path-finding in computer games. Specifically, the A* algorithm is the most widely used search method for path negotiation in games. Game developers like using the A* algorithm because it is so flexible. A* does not blindly search but rather assesses the best direction to explore even if that means backtracking. Ultimately, the A* algorithm will determine the shortest path between two points.

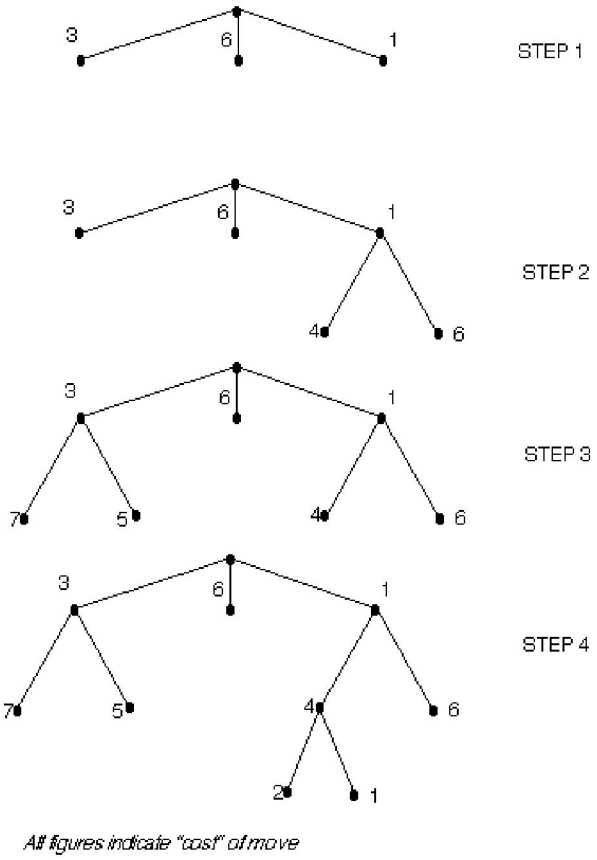

Figure 3.
A* Algorithm Spanning Tree

The A* algorithm finds a path between two nodes in a graph. The nodes also store information that is essential to the A* algorithm like graph position. The cost of a node takes into account various factors like how much energy it would take to travel a path. The job of the A* algorithm is to find the shortest path between two nodes with the least cost.

Typical A* algorithms have three main attributes, fitness, goal, and heuristic or *f*, *g*, and *h* respectively. The typical formula is expressed as:

$$f(n) = g(n) + h(n) \qquad (1)$$

where:

- *f(n)* is the score assigned to node n
- *g(n)* is the actual cheapest cost of arriving at n from the start
- *h(n)* is the heuristic estimate of the cost to the goal from n

The algorithm is seen below:

```
priorityqueue Open
list Closed


AStarSearch
   s.g = 0  // s is the start node
   s.h = GoalDistEstimate( s )
   s.f = s.g + s.h
   s.parent = null
   push s on Open
   while Open is not empty
      pop node n from Open  //n has the
lowest f
      if n is a goal node
         construct path
         return success
      for each successor n' of n
         newg = n.g + cost(n,n')
         if n' is in Open or Closed,
          and n'.g < = newg
                skip
         n'.parent = n
         n'.g = newg
         n'.h = GoalDistEstimate( n' )
         n'.f = n'.g + n'.h
         if n' is in Closed
            remove it from Closed
         if n' is not yet in Open
            push n' on Open
      push n onto Closed
   return failure // if no path found
```

Also stated before, though it's one of the favorites, A* algorithm has some limitations. There are situations where A* may not perform very well, for a variety of reasons. The more or less real-time requirements of games, plus the limitations of the available memory and processor time in some of them, may make it hard even for A* to work well. A large map may require thousands of entries

in the Open and Closed list, and there may not be room enough for that. Even if there is enough memory for them, the algorithms used for manipulating them may be inefficient.

The quality of A*'s search depends on the quality of the heuristic estimate, *h(n)*. If *h* is very close to the true cost of the remaining path, its efficiency will be high. On the other hand, if it is too low, its efficiency gets very bad. In fact, breadth-first search is an A* search, with *h* being trivially zero for all nodes this certainly underestimates the remaining path cost, and while it will find the optimum path, it will do so slowly.

2.2 NPC Movement with Fringe Search Algorithm

Consider Figure 4: each branch is labeled with a path cost (1 or 2) and the heuristic function h is the number of moves required to reach the bottom of the tree (each move has an admissible cost of 1).

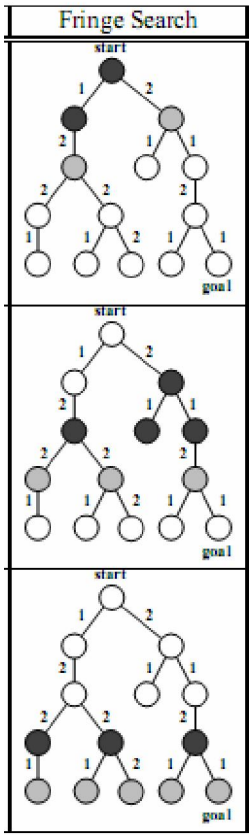

Figure 4.
Spanning Tree For Fringe Search

Fringe algorithm starts out with a threshold of h(start)=4. Two nodes are expanded (black circles)vand two nodes are visited (gray circles) before the algorithm proves that no solution is possible with a cost of 4. The two leaf nodes of the first iteration are saved, and are then used as the starting point for the second iteration. The second iteration has 3 leaf nodes that are used for the third iteration. For the last iteration, Fringe Search algorithm only visits the parts that have not yet been explored. In this example, a total of 9 nodes are expanded and 19 nodes are visited.

Figure 5.
MazeGame Screenshot
It Already Used Fringe Algorithm as NPC Movement AI

This new algorithm is called the Fringe Search, since the algorithm iterates over the fringe (frontier) of the search tree. The data structure used by Fringe Search can be thought of as two lists: one for the current iteration (now) and one for the next iteration (later). Initially the now list starts off with the root node and the later list is empty. The algorithm repeatedly does the following. The node at the head of the now list (head) is examined and one of the following actions is taken:

1. If *f(head)* is greater than the threshold then head is removed from now and placed at the end of later. In other words, we do not need to consider head on this iteration (we only visited head), so we save it for consideration in the next iteration.
2. If *f(head)* is less or equal than the threshold then we need to consider head's children (expand head). Add the children of head to the front of now. Node head is discarded.

When an iteration completes and a goal has not been found, then the search threshold is increased, the later linked list becomes the now list, and later is set to empty.

The pseudo code for the algorithm is seen below:

```
Initialize:
   Fringe F ß (s)
   Cache C[start] ß (0, null),
   C[n] ß null for n ≠ start
   f_limit ß h(start)
   found ß false
   Repeat until found = true or F empty
      f_min ß ∞
      Iterate over nodes n ∈ F from left
         to right:
        (g, parent) ß C[n]
        f ← g + h(n)
        If f > f_limit
          f_min ß min(f,f_min)
          continue
        If n = goal
          found ß true
          break
        Iterate over s ∈ successors(n)
          from right to left:
          g_s ß g +cost(n, s)
          If C[s] ≠ null
            (g',parent) ß C[s]
            If gs ≥ g'
              continue
          If F contains s
            Remove s from F
          Insert s into F after n
          C[s] ← (g_s,n)
        Remove n from F
      f_limit ← f_min
   If found = true
      Construct path from parent nodes in
         cache
```

3. COMPARING A* AND FRINGE

For the experiments two different grid movement models are used: tiles, where the agent movement is restricted to the four orthogonal directions (move cost = 100), and octiles, where the agent can additionally move diagonally (move cost = 150). To better simulate game worlds that use variable terrain costs we also experiment with two different obstacle models: one where obstacles are impassable, and the other where they can be traversed, although at threefold the usual cost. As a heuristic function we used the minimum distance as if traveling on an obstacle-free map (e.g. Manhattan-distance for tiles). The heuristic is both admissible and consistent.

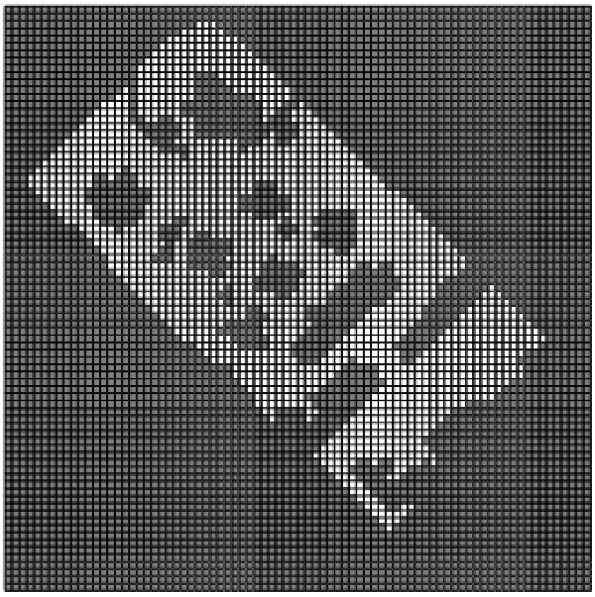

Figure 6.
Example Map

	Octiles		Tiles	
	A*	**Fringe**	**A***	**Fringe**
CPU/msec	1.7	1.3	1.2	0.8
Iterations	25.8	25.8	9.2	9.2
N-visited	583.4	2490.7	607.0	1155.3
N-visited-last	27.7	79.5	54.5	103.8
N-expanded	582.4	586.5	606.0	613.2
N-expanded-last	26.7	30.7	53.5	60.7
P-cost	5637.7	5637.7	6758.6	6758.6
P-length	46.1	46.1	68.6	68.6

Table 1.
Result for pathfinding in Figure 6

The A* algorithm is the de facto standard used for pathfinding search. As can be seen from Table 1, both algorithms expand comparable number of nodes (the only difference is that because of its g-value ordering A finds the target a little earlier in the last iteration). Fringe Search, on the other hand, visits many more nodes than A*. Visiting a node in Fringe Search is an inexpensive operation, because the algorithm iterates over the node-list in a linear fashion. In contrast, though both A* and Fringe search algorithm would give the same path, A* requires far more overhead per node because of the extra work needed to maintain a sorted order. Time-wise the Fringe

Search algorithm outperforms A* by a significant margin, running 25%-40% faster on average depending on the model.

Note that under the passable obstacle model, there is a small difference in the path lengths found by A* and Fringe. This is not a concern as long as the costs are the same (a length of a path is the number of grid cells on the path, but because under this model the cells can have different costs it is possible that two or more different length paths are both optimal cost-wise).

Using buckets, Fringe Search could do partial or even full sorting, reducing or eliminating A*'s best-first search advantage. The expanded-last row in Tables 1 and 2 shows that on the last iteration, Fringe Search expands more nodes (as expected). However, the difference is small, meaning that for this application domain, the advantages of best-first search are insignificant. The ratio of nodes visited by Fringe Search versus A* is different for each model used. For example, in the impassable and passable obstacles model these ratios are approximately 4 and 6, respectively. It is of interest to note that a higher ratio does not necessarily translate into worse relative performance for Fringe Search; in both cases the relative performance gain is the same, or approximately 25%.

The reason is that there is a "hidden" cost in A* not reflected in the above statistics, namely as the Open List gets larger so will the cost of maintaining it in a sorted order.

4. CONCLUSION

NPC needs an AI to move, in this case, to find a way to get to the PC (Playable Characters). In other words, NPC needs a pathfinding algorithm that will take him to the character the fastest. A* algorithm is commonly used, but have some weaknesses. In order to minimize the weaknesses, a new algorithm is developed. It's Fringe Search Algorithm.

Large memories are ubiquitous, and the amount of memory available will only increase. The class of single-agent search applications that need fast memory-resident solutions will only increase. As this paper shows, in this case, A* is not the best choices for some applications. For example, Fringe Search out-performs optimized versions of A*, though they both give the same path, by significant margins when pathfinding on grids typical of game worlds. Compared to A*, Fringe Search avoids the overhead of maintaining a sorted open list. Although visiting more nodes than A* does, the low overhead per node visit in the Fringe Search algorithm results in an overall improved running time.

Fringe Search algorithm can be used for NPC AI movement in high difficulty games, because the enemy NPC can find our character in game faster than NPC using A* algorithm.

Some data is taken unfiltered. There might be some parts that are taken unedited from the resource.

BIBLIOGRAPHY

[1] Yngvi Bjornsson, "Fringe Search: Beating A* at Pathfinding on Game Maps"
[2] AISeek Ltd, "Intelligence For New Worlds"
[3] Greg Alt, "The Suffering: A Game AI Case Study"