**CS570 Artificial Intelligence**

**Spring 2012**

**Project 1**

**Feb 15, 2012**

**Sanqing Yuan**

**Pathfinding**

**Abstract**

In this project, four pathfinding strategies were implemented and evaluated, including Breadth-first search, Uniform-cost search, Iterative deepening by cost and A\* search with two different heuristics functions. C++ programming language was used. It was found that Breadth-first search can always find the shortest path without considering cost but with high time and space complexity. Uniform-cost search and Iterative deepening by cost search are able to find the path with lowest cost but Iterative deepening has higher time complexity because of the overhead. A\* search is the most efficient algorithm to find the lowest cost path because it can find the lowest cost path with the smallest number of visited cells.

**Introduction**

Pathfinding is the process of plotting an efficiently traversable path between points, called nodes [1]. It is one of the greatest challenges in the design of realistic Artificial Intelligence (AI) in computer games. In addition to computer games, pathfinding is also widely used in many other areas such as navigation. Pathfinding is an instance of search problem. Its goal is to find the lowest cost path between the start point and the destination. The solution of a pathfinding problem is an action sequence that can lead the agent moves from the start point to the destination.

The essence of pathfinding is the choice of actions. It depends on which child node to expand next – the so-called search strategy. The search strategies can be roughly divided into two categories, uninformed search and heuristic search. The former one is given no information about the state of destination and heuristic search knows additional information about destination. For each categories, the search strategies can be distinguished further by the order in which nodes are expanded [2]. In this project, some most commonly used pathfinding strategies were implemented and evaluated, including Breadth-first search, Uniform-cost search, Iterative deepening by cost and A\* search with different heuristics functions. It was found that Breadth-first search can always find the shortest path without considering cost. Uniform-cost search is able to find the path with lowest cost. Iterative deepening by cost also can find the lowest cost path but with lots of overhead. A\* search is the most efficient algorithm to find the lowest cost path.

**Algorithms**

In this project, all algorithms are implemented in C++. Two 2-dementional arrays were created. One was used to store the original map and the other was to store explored set of node. A queue class was created for each algorithm. For Breadth-first search, Lowest-cost search and A\* search, the queue class included 2 linked lists. One linked list (point to child node) was used for frontier queue. However, Breadth-first search used a FIFO queue, while the other three algorithms used a priority queue based on cost. Another linked list was used to store backtracking information, whose nodes had the pointer that points to their parent. This linked list is very useful when we return the solution. In addition to these two linked list, Iterative-deepening by cost search need the third linked list in the queue class, which is a normal linked list that stores all pointers of nodes in the second linked list. This third linked list was used to clean the whole queue class after each iteration so we can start over after the cutoff of cost increased. Figure 1-4 showed pseudocodes for each algorithm.

read map from input file and store the information

node <- initial node

**if** node is goal **then** **return** the solution

**else** add it to the *frontier* (FIFO queue)

set *explored* to be an empty set

**loop do**

**if** *frontier* is empty **then return** search fails

**else** POP *frontier* -> node /\* pop the shallowest node in frontier \*/

add this node to *explored* set

**for** each child nodes of current node **do**

**if** child node is not in *explored* or *frontier* **then**

**if** this child node is goal then return solution

**else** add this node to *frontier*

Figure 1 Pseudocode of Breadth-first search

read map from input file and store the information

node <- initial node, Path-cost = 0

add node to *frontier* queue /\*priority queue by cost, lowest cost node pops first\*/

set *explored* to be an empty set

**loop do**

**if** *frontier* is empty **then return** search fails

**else** POP *frontier* -> node /\* lowest-cost node pop first \*/

**if** node state is goal **then** **return** the solution

**else** add it into *explored* set

**for** each child nodes of current node **do**

**if** child node is not in *explored* or *frontier* **then**

add this node to frontier /\* node cost is the cost to reach this node\*/

**else if** child node state is in *frontier* with higher cost **then**

replace that *frontier* node with child

Figure 2 Pseudocode of lowest cost search

Figure 2 Pseudocode of Iterative deepening by cost search

read map from input file and store the information

node <- initial node, Path-cost = 0

add node to *frontier* queue /\*priority queue by cost, lowest cost node pops first\*/

initialize *discard* queue /\*priority queue by cost, lowest cost node pops first\*/

set *explored* to be an empty set

set up initial cost *cutoff*

**loop do**

POP *frontier* -> node /\* lowest-cost node pop first \*/

**if** node state is goal **then** **return** the solution

**else** add it into *explored* set

**for** each child nodes of current node **do**

**if** child node is not in *explored* or *frontier* **then**

**if** child node cost > *cutoff* **then** add it to *discard* queue

**else**

**if** child node state is in *frontier* with higher cost **then**

replace that frontier node with child

**else** add this node to *frontier* /\* node cost is the cost to reach this node\*/

**if** frontier is empty **then**

cutoff <- pop *discard* queue

clean *discard* and *frontier* queue

set explored set to be empty

Figure 3 Pseudocode of Iterative-deepening by cost search

Figure 3 didn’t show all details of the algorithm of iterative-deepening by cost search. The very important thing in this algorithm is to choose the cost cutoff for next iteration. A discard queue was created here to store all nodes that were generated whose path cost exceeds the current cutoff. For the next iteration, the cutoff was set to the lowest path cost of nodes in discard queue in the previous iteration. This can be easily implemented by pop discard queue since it is a priority queue by path cost.

read map from input file and store the information

node <- initial node, Path-cost = 0

add node to *frontier* queue /\*priority queue by cost, lowest cost node pops first\*/

set *explored* to be an empty set

**loop do**

**if** *frontier* is empty **then return** search fails

**else** POP *frontier* -> node /\* lowest-cost node pop first \*/

**if** node state is goal **then** **return** the solution

**else** add it into *explored* set

**for** each child nodes of current node **do**

**if** child node is not in *explored* or *frontier* **then**

add this node to *frontier* /\*node cost here combines the cost to reach the node and the cost to get from the node to the goal \*/

**else if** child node state is in *frontier* with higher cost **then**

replace that frontier node with child

Figure 4 Pseudocode of A\* search

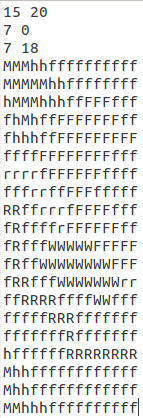
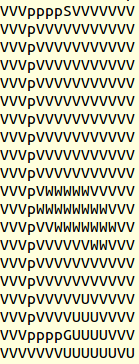
Figure 4 showed the pseudocode for A\* search. It is very similar to lowest-cost search. The only difference is the way to calculate the cost for node. In lowest-cost search, the cost of a node is the cost to reach this node from the start, while in A\* search, the cost of a node is the sum of the cost above and the cost to get from the node to the goal. I used two different heuristics functions to estimate the latter cost.

Heuristics function 1:

Heuristics function 2:

All algorithm worked fine and the results are shown in the next section.

**Results**

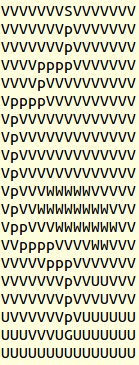
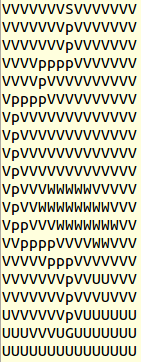
 

**Figure 5** Original map **Figure 6** Breadth first search [unvisited cells (U), visited

cells (V), water (W) , path (p), start point(S) and the goal (G)]. It

has 263 visited cells and 26 steps of path.

Based on figure 5-10, all the algorithms but breadth-first have the same solutions. That is because the last four algorithms take cost into account while breadth-first search only cares about shortest steps. Therefore, although the solution provided by breadth-first search is the shortest, it has higher (89 vs 56, see figure 11) cost compared to the solution given by the other algorithms.

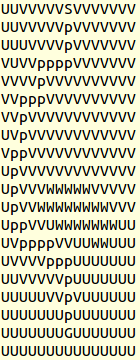
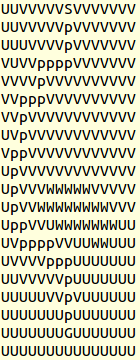
 

**Figure 7** Lowest-cost first search [unvisited cells (U), visited **Figure 8**  Iterative-deepening by cost (unvested cells

cells (V), water (W) , path (p), start point(S) and the goal (G)]. (U), visited cells (V), water (W), path (p), start point

It has 242 visited cells and 30 steps of path. The lowest cost (S) and the goal (G)). It has 242 visited cells and 30

is 56. steps of path. The lowest cost is 56.

**Figure 9** A\* search with h(n) 1 [unvisited cells (U), visited **Figure 10** A\* search with h(n) 2 (unvested cells

cells (V), water (W) , path (p), start point(S) and the goal (G)]. (U), visited cells (V), water (W), path (p), start point

It has 185 visited cells and 30 steps of path. The lowest cost (S) and the goal (G)). It has 185 visited cells and 30

is 56. steps of path. The lowest cost is 56.

**Figure 11** Number of explored cells, cost and steps of path for the four algorithm using map provided by Dr.Soule.

As shown in figure 11, another thing that is worth mentioning is that although the solution from the last four algorithms are the same, the time and space complexity of each algorithm is different. Compared to lowest-cost search, iterative-deepening by cost search has much more overhead. Although the number of visited cells are the same, iterative-deepening by cost search actually visits some of them many times. A\* search, on the other hand, has less explored cells compared to uninformed search methods. The reason is obvious because it have more information regarding the right direction to expand a child node.

**Figure 12** Plot of distance between start and goal, and the number of explored cells for each of the five algorithm. The curves of lowest-cost and iterative-deepening by cost are exactly the same. Also the curves of two A\* search algorithm are exactly the same.

Figure 12 showed us how the number of explored cells increases when the distance between start and goal increases. Note that numbers of explored cells used in figure 12 are the numbers of explored cells when the solution was found. As mentioned above, iterative-deepening by cost search actually visits some of cells many times. If we take this into account, iterative-deepening by cost search should have the largest visited cells among all algorithms. Another thing we have to mention is that in this project, the maps we used have boundaries, which causes that the unvisited cells in map is not infinite. Therefore, the curves shown in figure 12 are approximately linear. Even so, A\* search explores the smallest number of cells before it reaches the goal.

**Conclusions**

In this project, four different pathfinding strategies were implemented and evaluated. Breadth-first search can find the shortest path without considering path cost. Lowest-cost search, iterative-deepening by cost search and A\* search are able to find lowest-cost path. Among these three algorithms, A\* has the best time and space complexity, while iterative-deepening by cost search has lots of overhead compared to lowest-cost search.

However, there are several limitations in this study. First, the maps we used have boundaries. When expanding reaches boundaries, it stops. This happens a lot when we used breadth-first search, lowest-cost search and iterative-deepening by cost search, and actually enormously reduces the number of explored cells, compared to the situation in which we used a map without boundaries and has infinite cells. Because of this, we didn’t see much difference of explored cell numbers between these three search methods and A\* search. Second, for iterative-deepening by cost search, I didn’t count the number of visited cells, including re-visited cells. This number matters because it is the real number that determines time and space complexity. If this is done, the number of visited cells of iterative-deepening by cost search will be much larger than explored cells of lowest-cost search because of overhead. Third, using lowest-cost search, we found the solution in this case, but this is not true for all maps. If the cost of some steps don’t exceeds some small positive constant, the completeness is not guaranteed.

Two heuristic functions were used in this study. One is the straight line between current node and goal, and another one is Hamilton distance. Because agent can’t move diagonally in the map, both heuristic functions meet the condition of admissible. However, if we change the rules and allow diagonal movements, the second heuristic function may not work because the condition of admissible doesn’t hold any more.

**References**

1. Hart, P.E., N.J. Nilsson, and B. Raphael, *A Formal Basis for the Heuristic Determination of Minimum Cost Paths.* IEEE Transactions on Systems Science and Cybernetics, 1968. **4**(2): p. 100-107.

1. Russell, S.J. and P. Norvig, *Artificial intelligence: a modern approach*. 2010: Prentice Hall.

