CS 470 Spring 2011
Project 1

Colby Blair

Due February 18th, 2011

Abstract

In Artificial Intelligence, we can test intelligent agents on their ability to navigate maps to reach a goal position. For each position, there is a choice of options that we can take, and how process these options depends on which search we choose. With each step, our new possible routes increases exponentially. The search algorithm must be able to handle all these possible routes one by one, and build routes efficiently. This report will test the Breadth First Search, discuss the methodology, and highlight the pros and cons of this type of search.

With all searches, there is a tradeoff between best Path Cost, Time Complexity, and Space Complexity. If one needs absolute best (least) Path Cost, Lowest Cost Search is the best across all map systems. Iterative Deepening By Cost Search finds the lowest cost path and is slightly better than Lowest Cost Search, but its Time Complexity eclipses all other searches. A* searches can be better than all searches, but you have to develop really good heuristics (estimates on future cost). This is tough to do for all possible maps, so without a lot of work in heuristics, Lowest Cost Search is the best bet all around.

Contents

1 Introduction

Searching through maps comes down to 3 factors to trade off: Path Cost, Time Complexity (time spent searching), and Space Complexity (memory of past and next locations). Some searches find a good combonation of all three, but if a search is needed that finds the optimum path, we have to pay a price.

2 Path Tracing

An extra feature to searching is to trace the path an individual took to get to the location. There are several ways to do this. The way the search did this here is each node on the Value Matrix had a Parent Pointer. This pointer was set to Null initially. Then, whenever a node (or cell) was added to the frontier queue, the corresponding cell node on the Value Matrix had its parent pointer set to the corresponding Value Matrix cell node of the node that added.

For example, in the figures for Breadth Search below, when cell node 'start' added cell node 3 to the frontier queue, Value Matrix cell node 3's parent was set to Value Matrix cell node 'start'. Since cell node 3 could only ever be added once, it will only ever have its parent set once. Thus, a route back to the cell node 'start' will always be found anywhere in the explored Value Matrix.

3 Breadth First Search

3.1 Procedure

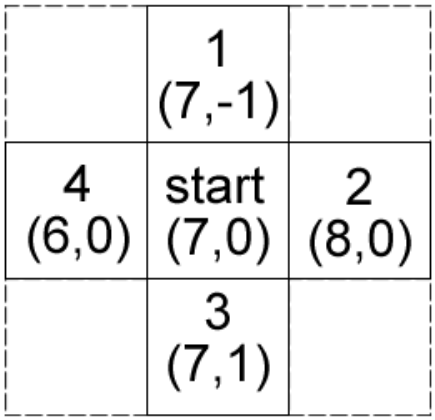
Consider a map like the following:

```
Map Parameters:
Size: (15, 20)
Start: (7, 0)
Goal: (7, 18)
Map before searchL
    M     M     M     h     h     f     f     f     f     f     f     f     f     f     f
    M     M     M     M     M     h     h     f     f     f     f     f     f     f     f
    h     M     M     M     h     h     h     f     f     F     F     F     f     f     f
    f     h     M     h     f     f     F     F     F     F     F     F     F     f     f
    f     h     h     h     f     f     F     F     F     F     F     F     F     F     F
    f     f     f     f     F     F     F     F     F     F     F     F     f     f     f
    r     r     r     r     f     F     F     F     F     F     F     f     f     f     f
    f     f     f     r     r     f     f     F     F     F     f     f     f     f     f
    R     R     f     f     r     r     r     f     F     F     F     F     f     f     f
    f     R     f     f     f     f     r     F     F     F     F     F     F     f     f
    f     R     f     f     f     W     W     W     W     W     F     F     F     F     F
    f     R     f     f     W     W     W     W     W     W     W     W     F     F     F
    f     R     R     f     f     f     W     W     W     W     W     W     W     r     r
    f     f     R     R     R     R     f     f     f     f     W     W     f     f     f
    f     f     f     f     f     R     R     R     f     f     f     f     f     f     f
    f     f     f     f     f     f     f     R     f     f     f     f     f     f     f
    h     f     f     f     f     f     f     R     R     R     R     R     R     R     R
    M     h     h     f     f     f     f     f     f     f     f     f     f     f     f
    M     h     h     f     f     f     f     f     f     f     f     f     f     f     f
    M     M     h     h     h     f     f     f     f     f     f     f     f     f     f
```

The search starts at cell x = 7, y = 0. Each cell has a cost signified by a character (R = low cost road, M = high cost mountain), but Breadth First Search does not care. It will simply find the shortest distance to the goal (there may be multiple that are equal). The only time this algorithm considers cost is when it is going off the side of the map or going into W (water), both of which have an invalid cost of 0 and are not considered.

This Breadth First Search uses 3 data structures: a Frontier Queue, an Explored List, and a Value Matrix. The only technical notes past these is that the Value Matrix is a matrix (or 2 dimensional array) of 'node' classes. Each 'node' class has members name, val, x, and y (where val equals the cost of the cell). Value_Matrix[0][0] then equals node('M',10,0,0), etc. Later we will see that each cell node also has a node parent pointer, but ignore this for now.

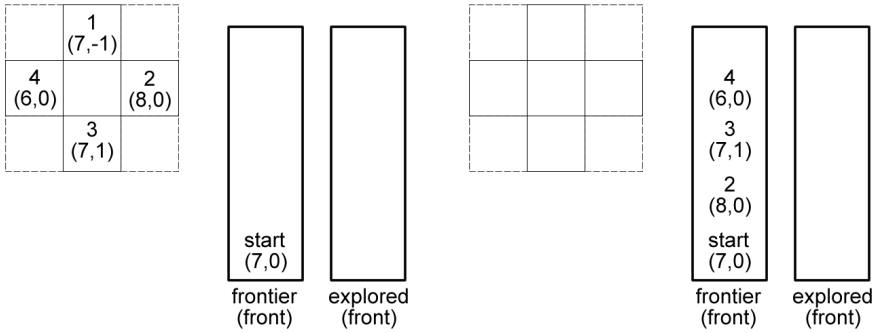
Let's consider our start section of the map:

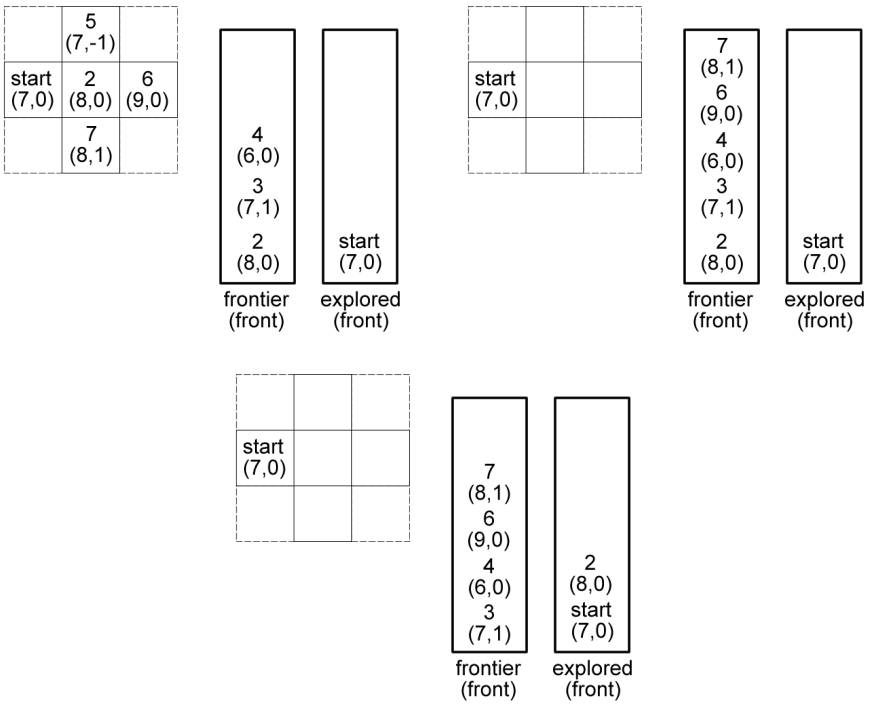


Cells above are labeled in the order the algorithm adds them to a queue.

1. 'start' is added.
2. while the front of the queue is not the goal:
   (a) front's neighbors are added to the end of the queue
   (b) front is popped from the queue

So each iteration would look something like the following:





This would continue on until the node (or cell) on the front of frontier queue was the goal. Notice that the search does not add nodes outside the bounds of the map or matrix (i.e. nodes 1 and 5 above). Also, nodes (or cells) that already exist in the explored list are not re-added to the frontier list.

3.2 Procedure Summary

This search is called a Breadth First Search because at every cell, there is a decision tree of possible new cells to move to. When the search processes every neighbor cell first before processing those neighbor cells' neighbors, the search has covered the breadth of the top level of the decision tree before moving on. In contrast, if the search was at the 'start' cell, processed cell 3 from above, and then processed all of cell 3's neighbors before looking at cell 4, the search would be searching as a Depth First Search.

The Depth First search would be searching each possible 1st neighbor until the last possible neighbor, and then backing up from the bottom of the de-

cision tree. Cell 4 above wouldn't get considered until every possible path from cell 3 was exhausted with no route to the goal.

3.3 Completeness, Efficiency (Cost and Path Length), and Complexity

The Breadth First Search is a **Complete** search, meaning it will find the goal if one exists. In **Efficiency**, it will always find the shortest path, but almost never find the least cost path. Thus, it will find the shallowest solution in the decision tree. The **Time Complexity** (or time it takes to find the goal) is $\approx B^d$, or about the number of each neighbor a cell could have to the power of the max depth of the decision tree. The **Space Complexity** (memory used, or storage all new neighbors to the frontier queue) is $\approx B^d$.

This search is advantageous to the Depth First Search, which is only sometimes complete (only when the depth is finite). This search will also find the shallowest solution (shortest path), which is not gaurenteed or even likely for Depth First Search. Depth first search will almost always have less Time Complexity (quicker) with an upper bound of B^d, and will have less Space Complexity (uses less memory). In practicality, very big maps would have to use Depth First Search for the Complexity reasons.

3.4 Program Output

The final output of the Breadth First Search is:

```
Map after search
    M     M     M     h_    h_    f_    f_    f_    f     f     f     f     f     f     f
    M     M     M     M_    M     h     h     f_    f     f     f     f     f     f     f
    h     M     M     M_    h     h     h     f     f     F     F     F     f     f     f
    f     h     M     h_    f     f     F     F     F     F     F     F     F     f     f
    f     h     h     h_    f     f     F     F     F     F     F     F     F     F     F
    f     f     f     f_    F     F     F     F     F     F     F     F     f     f     f
    r     r     r     r_    f     F     F     F     F     F     F     f     f     f     f
    f     f     f     r_    r     f     f     F     F     F     f     f     f     f     f
    R     R     f     f_    r     r     r     f     F     F     F     F     f     f     f
    f     R     f     f_    f     f     r     F     F     F     F     F     F     f     f
    f     R     f     f_    f     W     W     W     W     W     F     F     F     F     F
    f     R     f     f_    W     W     W     W     W     W     W     W     F     F     F
    f     R     R     f_    f     f     W     W     W     W     W     W     W     r     r
    f     f     R     R_    R     R     f     f     f     f     W     W     f     f     f
    f     f     f     f_    f     R     R     R     f     f     f     f     f     f     f
    f     f     f     f_    f     f     f     R     f     f     f     f     f     f     f
    h     f     f     f_    f     f     f     R     R     R     R     R     R     R     R
    M     h     h     f_    f     f     f     f     f     f     f     f     f     f     f
    M     h     h     f_    f_    f_    f_    f_    f     f     f     f     f     f     f
    M     M     h     h     h     f     f     f     f     f     f     f     f     f     f
Explored list contained 266 elements
```

We can see that this search takes the straightest path over until the 'W' water barrier is out of the way. The search goes straight down towards the goal, and then once parallel, cuts straight back across. This is the shortest path on our discrete map of cells. It is in now way the least cost path, but it is the straightest, considering the impassable 'W' water barrier.

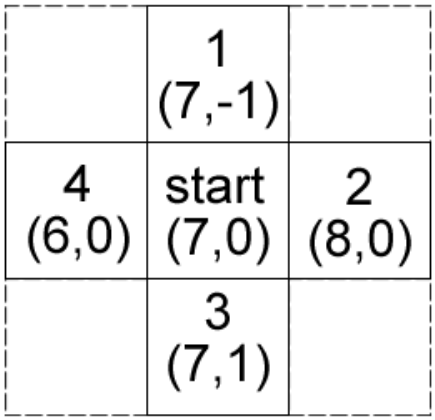
3.5 Breadth Conclusion

The Breadth First Search found the straightest path, but at a high Space Complexity / memory price. The frontier explored list contained 266 cell nodes, the majority of the 15 x 20 map (300 cell nodes). This Space Complexity on a larger map could be a big issue, and a different search algorithm may be needed.

4 Lowest Cost Search

4.1 Procedure

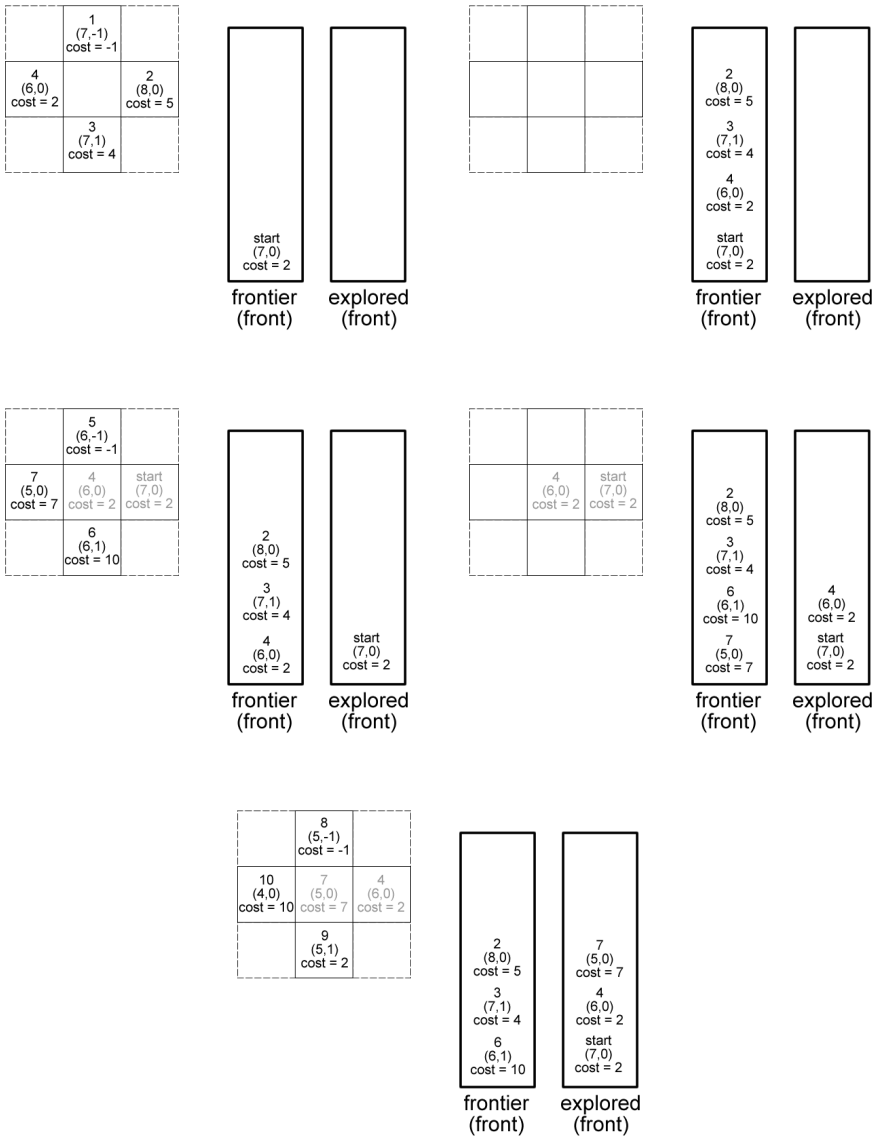
Let's consider again the start section of the map:



The process for Lowest Cost is as follows:

1. while goal cell node is not at the front of the frontier list
   (a) if the cell node at the front of the frontier list is not in the exlpored list:
      i. add the front of the frontier list to the explored list
      ii. pop the front of the frontier list
      iii. push the neighbors of the popped cell node onto the front of the frontier list
         - neighbors are sorted from lowest cost to cost
         - the neigbor with the lowest cost will be on the front of the frontier list
   (b) else pop the front of the frontier list

So each iteration would look like the following:



4.2 Procedure Summary

Search by Lowest Cost is a Depth First Search by Cost. It considers a cell node's neighbors, and picks the lowest cost cell node of those neighbors. It then looks at the lowest cost of that neighbor's neighbors, and pursues through the depth of that decision tree until the end. If it finds the goal,

it finishes. If it hits a dead end, it returns to the previous neighbors and considers the next lowest cost, again until it finds a goal, or again if it hits a dead end, it goes back up the decision tree and considers the next lowest cost neighbors of the next highest level.

4.3 Completeness, Efficiency (Cost and Path Length), and Complexity

Since this search is Depth First, it doesn't offer **Completeness** unless the map is limited. In our example maps, it is, but real world maps may be intractable. Since Least Cost is Depth First, it probably won't find the most **Efficient** path. It probably creates a longer **Path Length** than Breadth and Depth First, but a lower **Cost** than Depth and Breadth. **Time Complexity** is B^D as an upper bound, but usually does better. **Space Complexity** is $B * D$, which is this search's main advantage over other searches. Its Space Complexity is considerably better then many other searches.

4.4 Program Output

The final output of the Search is:
Explored:

```
Map after search:
 Explored Map:
   M    M_   M_   h_   h_   f_   f    f_   f_   f    f    f    f    f    f
   M_   M_   M_   M_   M_   h_   h_   f_   f_   f    f    f    f    f    f
   h_   M_   M_   M_   h    h    h    f_   f_   F    F    F    f    f    f
   f_   h_   M_   h    f_   f_   F_   F_   F    F    F    F    F    f    f
   f_   h    h    h    f_   f    F    F    F    F    F    F    F    F    F
   f_   f_   f_   f_   F_   F    F    F    F    F    F    F    f    f    f
   r_   r    r    r    f    F    F    F    F    F    F    f    f    f    f
   f_   f    f    r    r    f    f    F    F    F    f    f    f    f    f
   R_   R_   f    f    r    r    r    f    F    F    F    F    f    f    f
   f    R_   f    f    f    f    r    F    F    F    F    F    F    f    f
   f    R_   f    f    f    W    W    W    W    W    F    F    F    F    F
   f    R_   f    f    W    W    W    W    W    W    W    W    F    F    F
   f    R_   R_   f    f    f    W    W    W    W    W    W    W    r    r
   f    f    R_   R_   R_   R_   f    f    f    f    W    W    f    f    f
   f    f    f    f    f    R_   R_   R_   f    f    f    f    f    f    f
   f    f    f    f    f    f    f    R_   f    f    f    f    f    f    f
   h    f    f    f    f    f    f    R_   R_   R_   R_   R_   R_   R_   R_
   M    h    h    f    f    f    f    f    f_   f_   f_   f_   f_   f_   f_
   M    h    h    f    f    f    f    f    f_   f_   f_   f_   f_   f_   f_
   M    M    h    h    h    f    f    f    f    f    f    f    f    f    f
```

Path:

```
Path in Map:
   M     M     M     h     h     f     f     f_    f_    f     f     f     f     f     f
   M     M     M     M     M     h     h     f_    f_    f     f     f     f     f     f
   h     M     M     M     h     h     h     f_    f_    F     F     F     f     f     f
   f     h     M     h     f_    f_    F_    F_    F     F     F     F     F     f     f
   f     h     h     h     f_    f_    F_    F_    F     F     F     F     F     F     F
   f_    f_    f_    f_    F_    F     F     F     F     F     F     F     f     f     f
   r_    r_    r_    r_    f_    F     F     F     F     F     F     f     f     f     f
   f_    f     f     r     r     f     f     F     F     F     f     f     f     f     f
   R_    R_    f     f     r     r     r     f     F     F     F     F     f     f     f
   f_    R_    f     f     f     f     r     F     F     F     F     F     F     f     f
   f     R_    f     f     f     W     W     W     W     W     W     F     F     F     F
   f     R_    f     f     W     W     W     W     W     W     W     W     F     F     F
   f     R_    R_    f     f     f     W     W     W     W     W     W     W     r     r
   f     f     R_    R_    R_    R_    f     f     f     f     W     W     f     f     f
   f     f     f_    f_    f_    R_    R_    R_    f     f     f     f     f     f     f
   f     f     f     f     f     f     f_    R_    f     f     f     f     f     f     f
   h     f     f     f     f     f     f     R_    R_    R_    R_    R_    R_    R_    R_
   M     h     h     f     f     f     f     f     f_    f_    f_    f_    f     f     f_
   M     h     h     f     f     f     f     f_    f_    f_    f_    f_    f_    f_    f_
   M     M     h     h     f     f     f     f_    f_    f_    f_    f_    f_    f_    f_
Total Cost: 86
Explored list contained 68 elements
Path length: 49
Frontier List contained 65 elements
Total of all Explored Lists contained 68 elements
  (different in iterative depth searches only)
 Space Complexity (E List + F Queue): 133
 Time Complexity (E List Total, = E List except on Iter Depth): 68
```

4.5 Lowest Cost Conclusion

Lowest Cost Search is a reasonably good search to pick; it gets the next best cost path for a lot less Time Complexity. It could potentially be beat all around by A* Search with good heuristics, but it's algorithm has no dependency on any system, and performs with less Complexity than A*'s worst case scenario

5 Iterative Deepening By Cost Search

5.1 Procedure

The process for Iterative Deepening is simply the Least Cost with Max Cost incremented until the goal is reached:

max cost = 2

1. while goal cell node is not at the front of the frontier list
   (a) if the cell node at the front of the frontier list is not in the explored list AND its cost <= max cost
      i. add the front of the frontier list to the explored list
      ii. pop the front of the frontier list

iii. push the neighbors of the popped cell node onto the front of the frontier list
   - neighbors are sorted from lowest cost to highest cost
   - the neigbor with the lowest cost will be on the front of the frontier list

(b) else pop the front of the frontier list

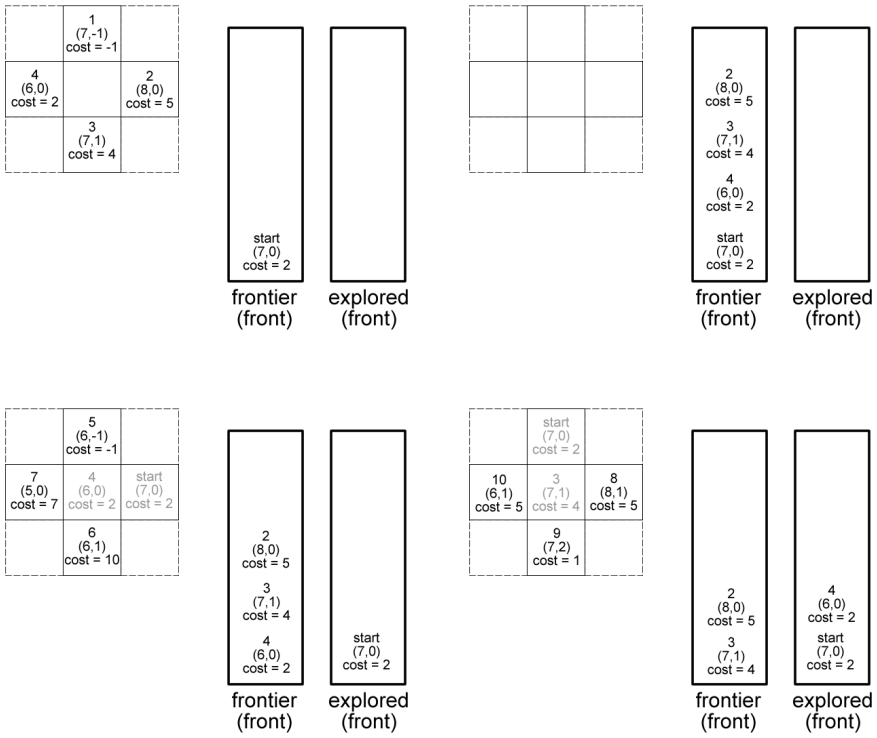
(c) if the frontier list is empty, or if the frontier list is empty AND cost > max depth

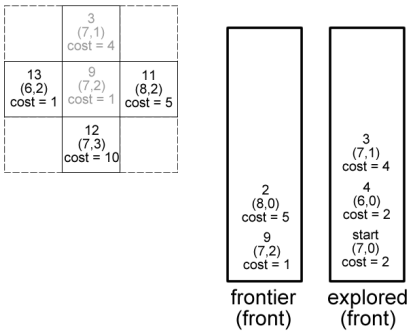
i. clear out the frontier list, leaving only the start cell node

ii. clear out the current explored list

iii. increment max cost by 1

So a process with a **max cost of 7** would look like:





5.2 Procedure Summary

Iterative Deepening by Cost was very similar to Search By Cost, except it placed an 'artifical' max cost that. The advantage to this was that it would stop at the max cost depth, and then explore its neighbors, and then all the neighbors above the max cost depth in the search tree. And then the search would explore those neighbors' children back down to the max cost depth. If the whole tree didn't have the goal, then the max cost would be incremented, and a whole other level of children were available on the bottom of the tree.

The advantage is that the search tends to look around much better in its surroundings and explore good low cost paths. The disadvantage is that everytime the max cost depth is increased, the whole search has to start from the top of the search tree (or the start cell node).

5.3 Completeness, Efficiency (Cost and Path Length), and Complexity

Iterative Deepening by Cost was probably the best all around search. It found the best path in my tests, although a A* Search with a great heuristic could beat it. It is **Complete**, like Breadth First; it is like a bottom-up Breadth Search in a way. It is gauranteed to find the most **Cost Efficient** path (unlike Breadth First), but not likely the shortest **Path Length**.

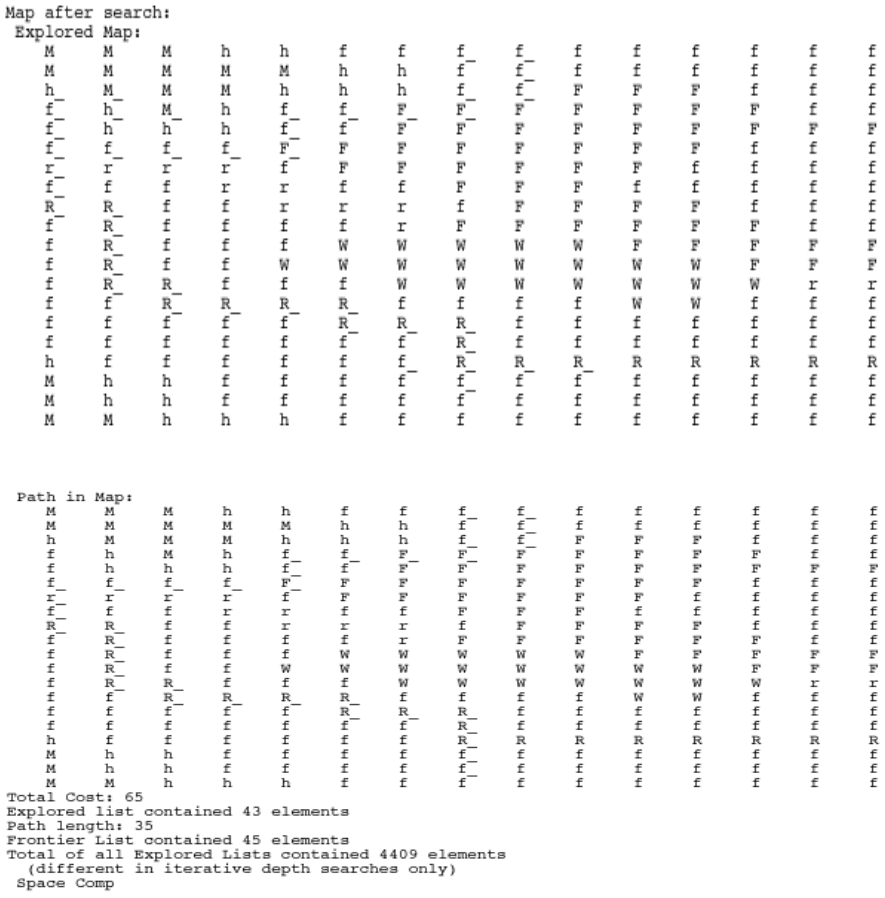
The search's **Space Complexity** is:

$$S_c = |LeastCostPath| + |LastIterationExploredList| <= B * D << B^D \quad (1)$$

-by far better than all other searches, with possibly the exception of an A* Search with a really great Heuristic. The only disadvantage is that the **Time Complexity** is $(i-1) * B^D + |LeastCostPath|$ where i is how many iterations of the max cost depth it took before the goal was found. This is considerably larger than even a Depth First Search.

5.4 Program Output

The final output of the Search is:
Explored Map



5.5 Iterative Deepening by Cost Conclusion

Even though this search's Time Complexity was so enormous, it found the best path for the lowest Space Complexity. This may often be the case in real life. If the answer needs to be quicker, then it will probably not be as Efficient or it may take much more Space Complexity. But if one just needs an answer, this search may be the best until the Time Complexity or map is untractable.

6 A* Search With Multiple Heuristics

6.1 Procedure

1. while the goal cell node is not on the front of the frontier list
   (a) if the front of the frontier list is not in the explored list
      i. add the front of the frontier list to the explored list
      ii. pop the front of the frontier list
      iii. push the neighbors of the popped cell node onto the front of the frontier list
         - neighbors are sorted from lowest cost to highest cost
         - the neigbor with the lowest cost will be on the front of the frontier list
         - cost is determined by the cell node cost + one of two heuristics (see below)
   (b) else pop the front of the frontier list

The process isn't easy to illustrate without looking at the entire graph, bu isn't difficult to understand when reading about the heuristics while looking at the output.

6.2 Heuristics

For heuristics to find the optimal path, they must never overestimate (is admissable). Each of these heuristics underestimate the true cost, but try to approach true cost.

6.2.1 H1 - Manhattan Distance

The first heuristic that the program used was calculating Manhattan Distance. For each cell node, its heuristic was the length of the shortest possible path to the goal. This underestimated true cost by saying each future node on the Manhattan Path would have a cost of 1.

6.2.2 H2 - Cost Guilt By Association

The second Heuristic tried to judge a bad neighborhood to explore by judging a cell node by it immediate neighbors (4 cell nodes surrounding it left, right, top and bottom). If a cell node was in a high cost neighborhood, the heuristic was high. This heuristic overestimates when there are single high cost cell node outliers, but in more continuous landscape tends not to overestimate, since the neighbors will be likely paths.

6.3 Procedure Summary

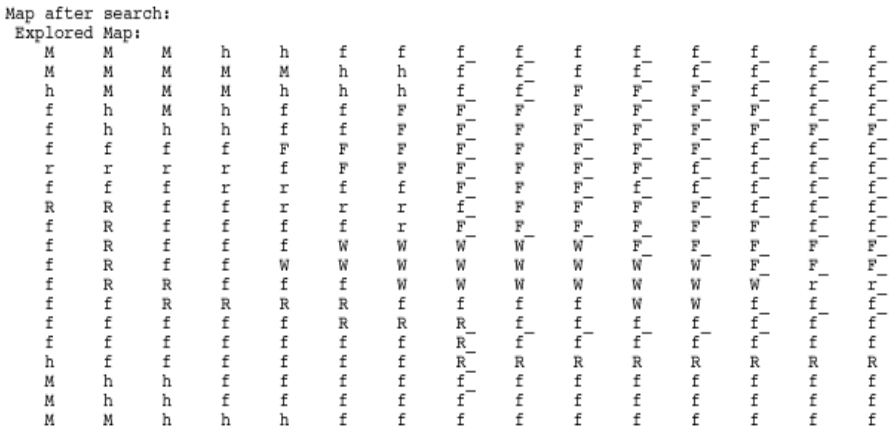
H1 performed better than H2, despite seeming to be dumber. Its ability to underestimate more seemed to help it. H2 should have perhaps taken the least cost of a cell node's neighbors instead of an average.

6.4 Completeness, Efficiency (Cost and Path Length), and Complexity

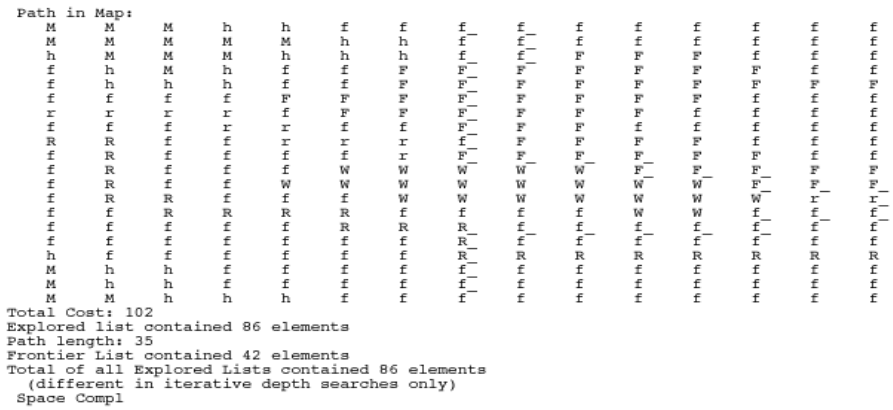
A* was a Depth First Search with a lookahead, so it was not at all **Complete**. With better heuristics, it could surpase Iterative Deepening by Cost's **Space Complexity** as $S_c = |LeastCostPath| + |LengthofExploredList|$, each of which would be less. Both H1 and H2 have a much better **Time Complexities** $T_c = |LengthofExploredList|$. Both H1 and H2 lost slightly to Breadth Search in Cost, but make it up in considerably less complexity.

6.5 Program Output

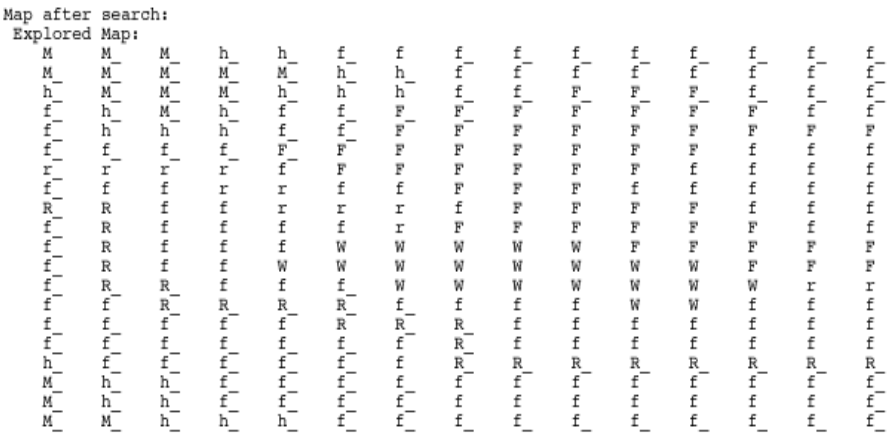
The final output of the A* H1 Search is:
Explored:



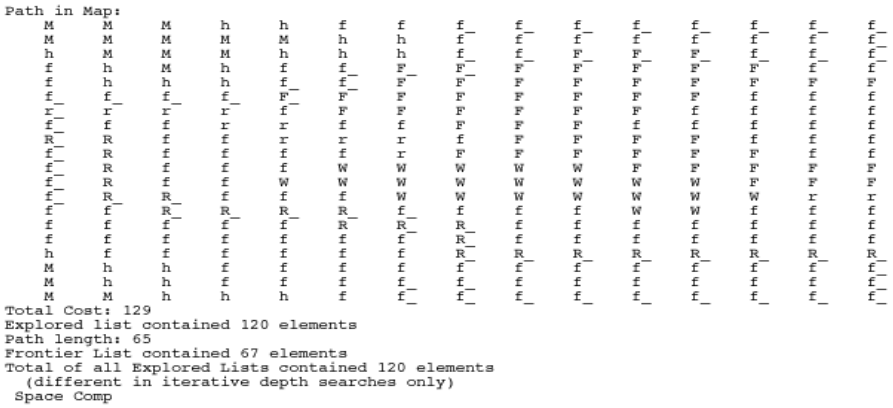
Path:



The final output of the A* H2 Search is:
Explored:



Path:

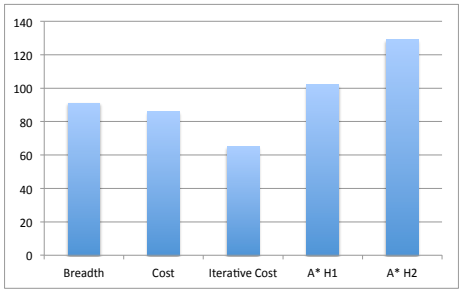


6.6 A* Search Conclusion

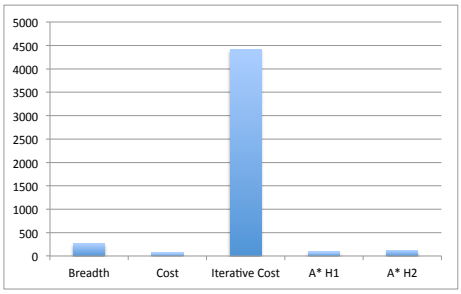
A* Search is a good search, and tends to be the right middle ground between Time Complexity versus Space Complexity. With really good heuristics, it could become better. It is tough to make a good heuristic that is independent of the sampled systems, and I was only able to try with mild success.

7 Algorithm Comparison

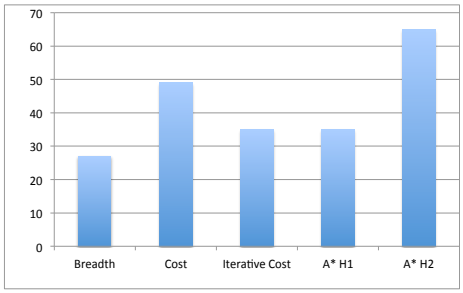
Path Cost	
Breadth	91
Cost	86
Iterative Cost	65
A* H1	102
A* H2	129



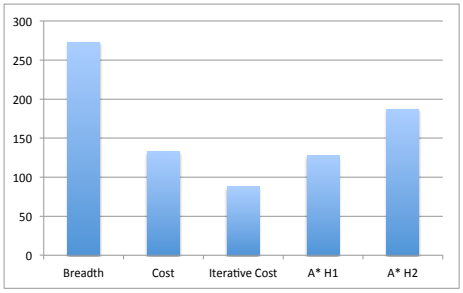
Time Complexity	
Breadth	266
Cost	68
Iterative Cost	4409
A* H1	86
A* H2	120



Path Length	
Breadth	27
Cost	49
Iterative Cost	35
A* H1	35
A* H2	65



Space Complexity	
Breadth	273
Cost	133
Iterative Cost	88
A* H1	128
A* H2	187



From these graphs, Iterative Deepening By Cost Searchfinds has the lowest path cost. But for its slight cost gain, it needs multitudes higher Time Complexity. The next best search is by Lowest Cost. Lowest Cost has still low path cost with low Time Complexity, and has fairly low Space Complexity. After that, A* H1 and H2 have reasonably low path cost, Time and Space Complexity. Breadth First Search only outperforms A*, and has substantially more Space Complexity.

8 Conclusion

From the results, it is clear that if you need a little better path cost, you have to give up a lot of Time Complexity to comute it with Iterative Deepening

by Cost. For a little more path cost, one can use Lowest Cost search and have reasonably low Time and Space Complexity. Breadth First Search gets average path cost results with a bit higher Time Complexity, but Space Complexity is considerably higher and expands fast. Which becomes untractable fast with real world problems. A* Heuristics can potentially outperform all searches if well writen, but can fall apart if they are too dependent on certain systems, and revert back to Breadth First Search performance.

In conclusion, pick Iterative Deepening by Cost Search if you have lots of time to find the best path. Otherwise, pick Lowest Cost Search.