Simple Search

Project One

Allison Ivey

ETSU Graduate  2/3/2014

Part One

# Map solutions

## FlagFlagFlagFlagFlagFlagMap One

**Analysis:**

**Search Times**:

**Best Time**: DFS

**Difference in Time**: 2.046875

**Path Cost**:

**Best Performing**: BFS

**BFS** = 108

**DFS** = 112

**Difference in Path**: 4

**Map Characteristics:**

**Start to End Relationship**: Diagonal

NW to SE

**BFS Better Turn Performance:** 4

**DFS Better Turn Performance:** 2

### **DEPTH FIRST BREADTH FIRST**

|  |  |
| --- | --- |
| Key | |
|  | BFS turns done in less moves than the DFS |
| Flag | DFS turns done in less moves than the BFS. Generally seen during a right-hand turn when the car is traveling North. |
| Flag | Example of the inefficiency of the DFS’s inability to travel the Manhattan distance along the path. |

## Map two

### **DEPTH FIRST BREADTH FIRST**

**Analysis:**

**Search Times**:

**Best Time**: DFS

**Difference in Time**: 1.125

**Path Cost**:

**Best Performing**: BFS

**BFS** = 83

**DFS** = 89

**Difference in Path**: 6

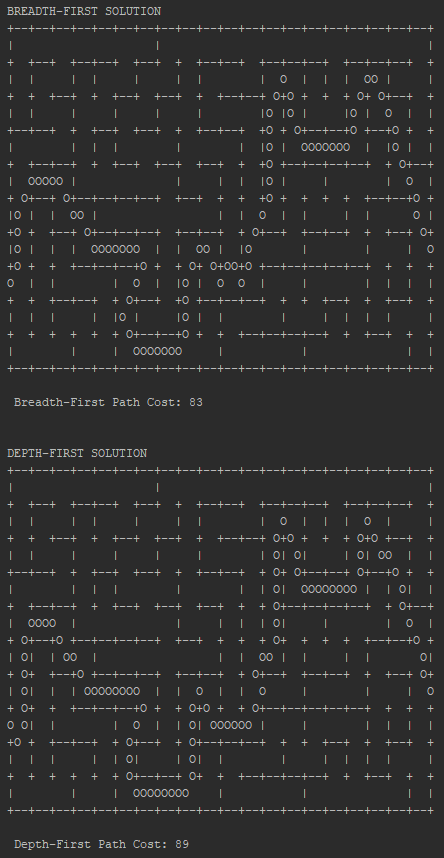
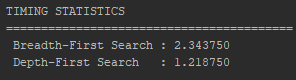
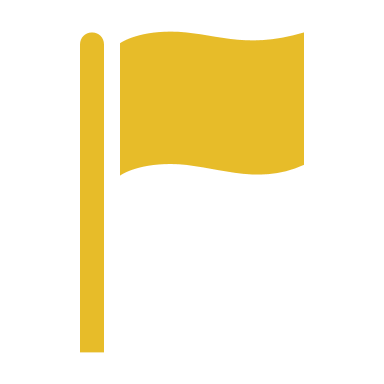
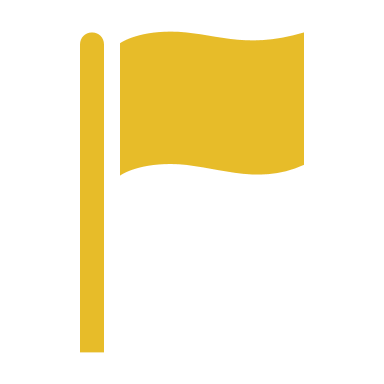
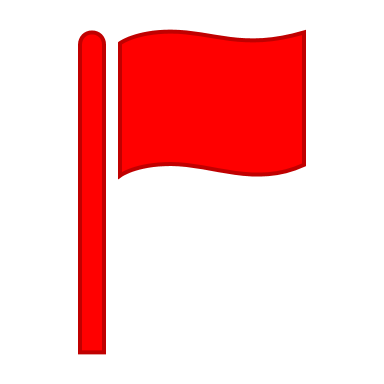
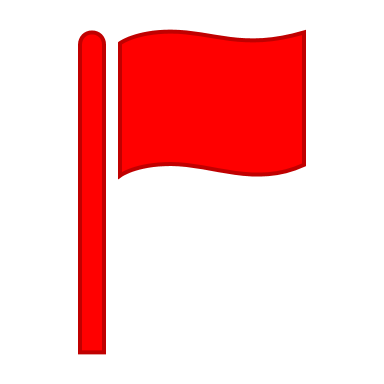
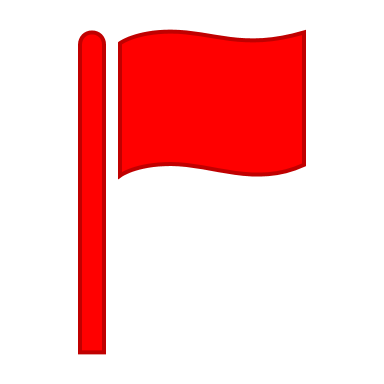
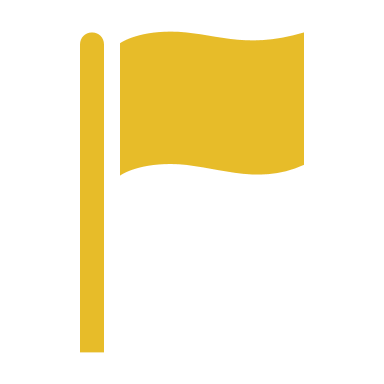
**Map Characteristics:**

**Start to End Relationship**: Across W to E

**BFS Better Turn Performance:** 3

**DFS Better Turn Performance:** 3

**Analysis**:



## FlagMap three

**Analysis:**

**Search Times**:

**Best Time**: DFS

**Difference in Time**: 1.90625

**Path Cost**:

**Best Performing**: BFS

**BFS** = 99

**DFS** = 113

**Difference in Path**: 14

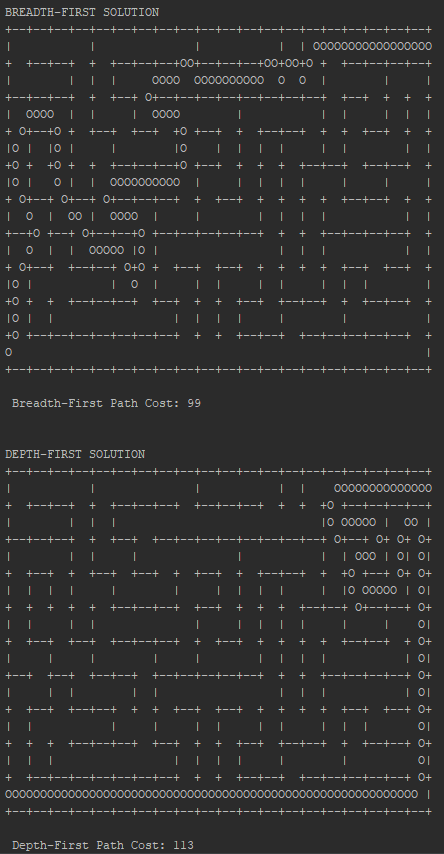
**Map Characteristics:**

**Start to End Relationship**: Diagonal

SW to NE

\*Map too different to compare turn outcomes.

### **DEPTH FIRST BREADTH FIRST**



## FlagFlagFlagFlagMap four

### **DEPTH FIRST BREADTH FIRST**

**Analysis:**

**Search Times**:

**Best Time**: DFS

**Difference in Time**: 0.53125

**Path Cost**:

**Best Performing**: BFS

**BFS** = 46

**DFS** = 51

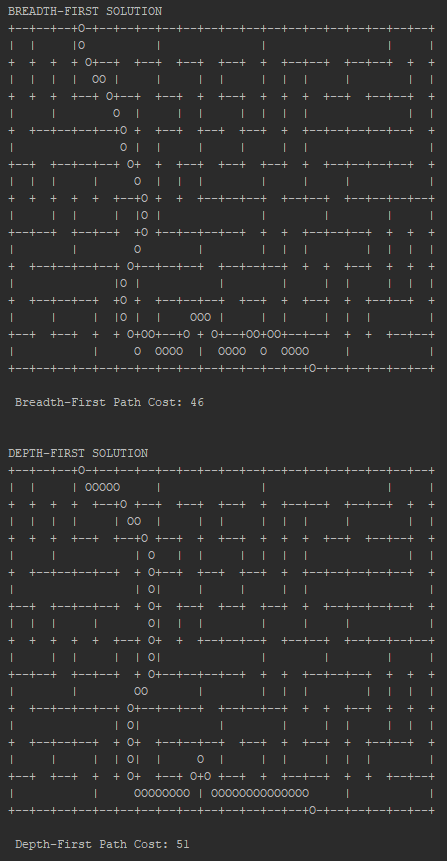
**Difference in Path**: 5

**Map Characteristics:**

**Start to End Relationship**: Across N to S

**BFS Better Turn Performance:** 3

**DFS Better Turn Performance:** 1



## FlagFlagFlagFlagFlagFlagFlagFlagFlagFlagFlagMap five

**Analysis:**

**Search Times**:

**Best Time**: DFS

**Difference in Time**: 0.90625

**Path Cost**:

**Best Performing**: BFS

**BFS** = 141

**DFS** = 154

**Difference in Path**: 13

**Map Characteristics:**

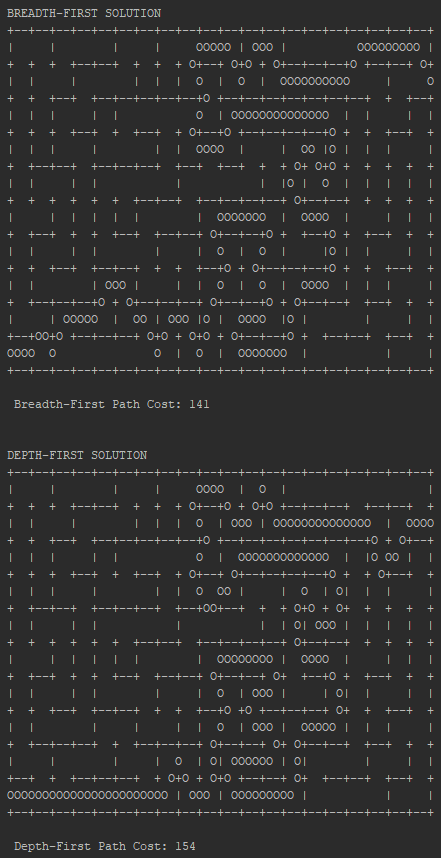
**Start to End Relationship**: Diagonal

SW to NE

**BFS Better Turn Performance:** 6

**DFS Better Turn Performance:** 4

### **DEPTH FIRST BREADTH FIRST**



## FlagFlagFlagFlagFlagFlagFlagFlagFlagMap six

**Analysis:**

**Search Times**:

**Best Time**: BFS

**Difference in Time**: -2.515625

**Path Cost**:

**Best Performing**: BFS

**BFS** = 90

**DFS** = 104

**Difference in Path**: 14

**Map Characteristics:**

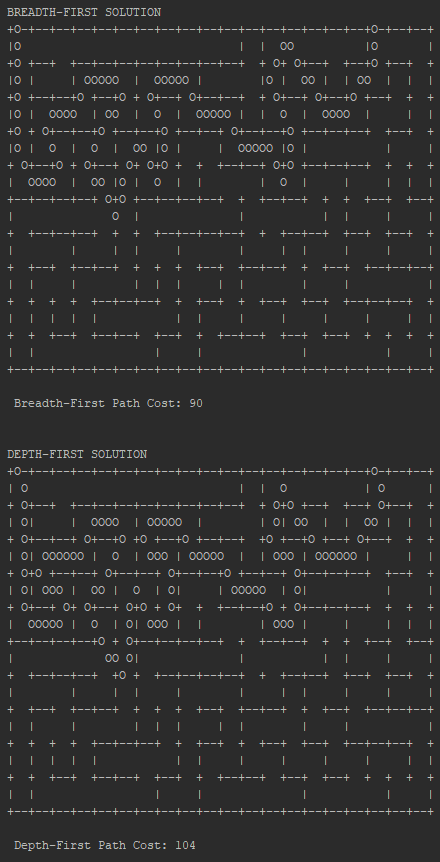
**Start to End Relationship**: Across

NW to NE

**BFS Better Turn Performance:** 6

**DFS Better Turn Performance:** 3

### **DEPTH FIRST BREADTH FIRST**



# Compiled Results

# Conclusion

Which algorithm to chose largely depends on a value judgement. Does the product owner want Herbie to find the shortest path from start to finish or do they want Herbie to make the decision quickly? Under the assumption that a second or two will not make a whole lot of difference to the product owner, it is my recommendation that Herby come equip with mapping equipment that uses the breadth first algorithm.

The breadth first search out performed the depth first search in terms of distance five out of five times. It did well with traversing the maps using Manhattan Directional movements. BFS moved in a zig zag fashion with the goal of arriving at a given destination. The DFS stuck to a single road until it no longer could be ridden and then found another road until it reached its destination. The differences are illustrated well when looking at the green flag on map three. While the BFS was able to weave its way through the maze of roads the DFS followed the low road until it came to a dead end and then changed course.

The average difference in time between the two search algorithms is marginal at .666665. The difference in step cost between the two was much more significant with an average of 9.3333 additional steps. Given that additional steps result in more wear and tear on a car the benefits to Herbie are more pronounced given the BFS algorithm.

There are times that the DFS algorithm does better than the BFS algorithm. This can be seen in the red flag indicators on each map. When Herbie is traveling in a Northern direction and taking tight right turns he performs better given the DFS rather than the BFS. The BFS performs better when making left turns after traveling south as indicated by the yellow flags. The benefit seen using DFS was inconsistent. Herbie saw more benefit in turning performance given the BFS five out of five times. This fact mitigates any benefit to the use of the DFS algorithm.

The breadth first search consistently out performed the depth first search in terms of path cost. BFS’s ability to zig zag its way to the goal lead to better outcomes. It is recommended that Herbie use the BFS algorithm.

part two

# Map solutions

## map one

Took shorter step but lead to greater total path cost

|  |  |
| --- | --- |
| **A\* Algorithm** | **Greedy Search Algorithm using Heuristic** |
| **Path Cost:** 344 | **Path Cost:** 364 |
| **Steps to End:** 11 | **Steps to End:** 14 |

## map two

Extra Step Taken

|  |  |
| --- | --- |
| **A\* Algorithm** | **Greedy Search Algorithm using Heuristic** |
| **Path Cost:** 312 | **Path Cost:** 312 |
| **Steps to End:** 6 | **Steps to End:** 7 |

## map three

|  |
| --- |
| **A\* Algorithm** |
| **Path Cost:** 301 |
| **Steps to End:** 7 |

# A\* Algorithm analysis

### code Analysis

node = self.start   
  
frontier = PriorityQueue()  
frontier.put(node, 0)  
came\_from = {}  
cost\_so\_far = {}  
  
came\_from[str(node)] = None  
cost\_so\_far[str(node)] = 0  
  
while not frontier.empty():  
 current = frontier.get()  
  
 if current == self.end:  
 break  
 for child in self.get\_neighbors(current):  
 new\_cost = cost\_so\_far[str(current)] +

self.straight\_line\_distance(child, current)  
 if str(child) not in cost\_so\_far.keys() or new\_cost <

cost\_so\_far[str(child)]:  
 cost\_so\_far[str(child)] = new\_cost  
 priority = new\_cost + self.straight\_line\_distance(child, self.end)  
 frontier.put(child,priority)  
 came\_from[str(child)] = current  
  
node = self.end  
self.backtrack(came\_from, node )  
return cost\_so\_far[str(node)

|  |  |
| --- | --- |
| A\* Algorithm Dictionary | |
| Node | An object that holds an ordered pair |
| frontier | A priority queue that is passed a Node object and the Node's Heuristic. It brings the node with the lowest Heuristic to the front of the queue. |
| come\_from | A dictionary that associates a parent ordered pair with its child ordered pair. Only nodes that are visited are added to the dictionary. |
| cost\_so\_far | A dictionary that associates a node with its path cost. It is also the return variable |
| current | The Node with the lowest heuristic is passed to current from the frontier priority queue. |
| end | The goal destination |
| start | The starting destination |
| get\_neighbor | A method that is passed a Node and returns a list of children associated with the given parent |
| child | A loop variable that holds a child from the get\_neighbor method |
| new\_cost | A temporary variable that holds the cost so far of the child in the get\_neighbor loop |
| priority | A temporary variable that holds the calculated heuristic of a given child |
| backtrack | A method that is passed a Node and the came\_from dictionary so that list of the Node's path can be made |

def straight\_line\_distance(self, point1, point2):  
 *'''  
 Returns the straight-line distance between point 1 and point 2  
 '''* distance = math.sqrt((point1.getX() - point2.getX())\*\*2 + (point1.getY() –

point2.getY())\*\*2)  
 return distance

|  |  |
| --- | --- |
| Distance Between Two Points Algorithm | |
| point1 | holds an ordered pair of x and y coordinates |
| point2 | holds an ordered pair of x and y coordinates |
| distance | the return variable that holds the results of the Pythagorean theorem being applied to the separate ordered pairs |
| math | a dictionary provided by python |
| getX | gets the x coordinate from an ordered pair |
| getY | gets the y coordinate from an ordered pair |

### A\* Algorithm Definition

A\* search is a best-first informed search algorithm. A\* uses an estimated cost represented as f(n) to only search optimal nodes. The estimated cost is made up of two parts: g(n) the cost to reach the node and h(n) an admissible heuristic. An admissible heuristic is one that can never be overestimated. In the context of this problem we use the straight-line distance of a node to the goal. This heuristic never changes and can never be overestimated so it is the perfect candidate to use as a heuristic.

We can also be sure that our heuristic is consistent. The consistency of the heuristic allows us to order the nodes in the priority queue helping us find an optimal path. The algorithm is always searching for a path that is better than the one that is has in front of it.

**Parent to Child + Child to Goal >= Parent to Goal = Consistent**

Pythagorean’s Theorem is used as the metric to order the priority queue. If the path cost of the parent, cost of the parent to the child, and the distance from the child to the goal is smaller than another child we know exploring that child path is more optimal.

The progression of the algorithm fallows the rules of the A\* search.

* It begins by setting the path cost of the parent
* It finds all the children of the current parent.
* It then looks at the first child to calculate the cost of the parent plus the cost of the parent to the child.
* It then goes to the next child node and compares that child’s total path cost with the old child’s path cost.
* It does this over and over until the best child is chosen.
* Once the best child is chosen the child is added to the priority queue with a priority that is set by the child’s total path cost plus its distance from the goal.
* The algorithm then loops again and takes the node off the queue that has the lowest number as its priority.
* When it loops again it makes sure that the node that is being measured against is not already a part of the path so far or it is on a shorter path.
* As the algorithm approaches finding the best route to the goal the path costs get larger while the distance to the goal gets shorter.
* Once the goal has been reached the optimal path has been discovered because it has only searched the most optimal branches.

### Greedy algorithm used as a reflex based agent a comparison

The comparison graph above shows an algorithm that does not go back up a tree to find a better path. The algorithm above only goes to the node that is both closer to the goal and closest to the parent. It does not take into account the fact that there are times when the parent node is not the best choice among its sibling nodes, but it still may result in a better path. The major benefit to using the A\* algorithm is its ability to reflect on the decisions made and make changes in the pursuit of a better path. Although both graphs depict a short path the A\* algorithm was able to consistently find the best path with the lowest cost.

Another benefit to the A\* algorithm is its ability to work well when there are competing good options. When the greedy algorithm was confronted with this problem it was not able to produce a path it just kept looping. That is why there is only one map represented for map three. In map one, rather than choose a further node with a better eventual outcome it chose the node that was closest to it. Because the second algorithm did not allow backtracking it was not able to find the optimal path.