**170021928**
**CS5001 A2 REPORT**
**SEARCH**

## 1. COMPLETED PARTS

Part 1: Depth First Search; Breadth First Search.
Part 2: Best First Search; A* Search, Distance Heuristics: Manhattan, Chebyshev & Euclidian.

## 2. HOW TO RUN

### Search1

```
java -jar Search1.jar  [param1] [param2]
```

**Param1** should be a choice of map from **1 - 8**. Maps 1 to 6 are the ones provided, 7 and 8 are my additions.

**Param2** should be a choice of algorithm, either **DF** for depth search, or **BF** for breadth first.

### Search2

```
java -jar Search2.jar  [param1] [param2] [param3]
```

**Param1** should be a choice of map from **1 - 8**. Maps 1 to 6 are the ones provided, 7 and 8 are my additions.

**Param2** should be a choice of algorithm, either **DF** for depth first search, **BF** for breadth first, **BestF** for best first, or **A*** for A*.

**Param3** should be a string representing the distance heuristic to be used, either **e** for Euclidean, **m** for Manhattan, **c** for Chebyshev or **com** for manhattan and chebyshev combined.

## 3. LITERATURE REVIEW

Search algorithms are useful whenever an agent needs to find a route from one location to another. Example applications include autonomous vehicles (Dolgov et al. 2008), AI for video games (Millington and Funge 2009) and pathfinding for machines and robotics - particularly for instances where the robot is required to make navigational decisions without human input for example in space exploration, or as in this assignment, search and rescue in a dangerous location.

Zaheer states that a search problem can be formally defined as consisting of four components, namely the the initial state (the starting position of the agent), the successor function (the method by which the next possible moves are identified), the goal test (to determine whether the agent has reached the goal state) and the path cost (the cost of the path from the initial state to the goal) (Zaheer 2006). I would argue that a fifth component is necessary: the method by which it is determined which of the successor states to explore next. Indeed, this choice is the crux of search in AI, and what differentiates each of the algorithms implemented here.

### Uninformed search

If the agent is not aware of the location of its goal, the search is uninformed. In this case, the agent cannot simply move closer towards the goal until it is reached, but must explore the state space until the location of the goal is found. This is considered a brute force approach, and does not require any pre-existing knowledge about the state space. (Korf 2010)

### Breadth first search

Breadth first search works by exploring nodes in order of their depth from the initial state - for example, it would expand all the nodes at depth 1 before moving on to depth 2. This ensures that an optimal solution will be found, but depending on the breadth of the search tree, it could take a long time and be costly in terms of memory required, as each layer of the tree must be saved in order to be able to create the next layer. Depending on the breadth of the tree, this could mean storing up to the entire search state.

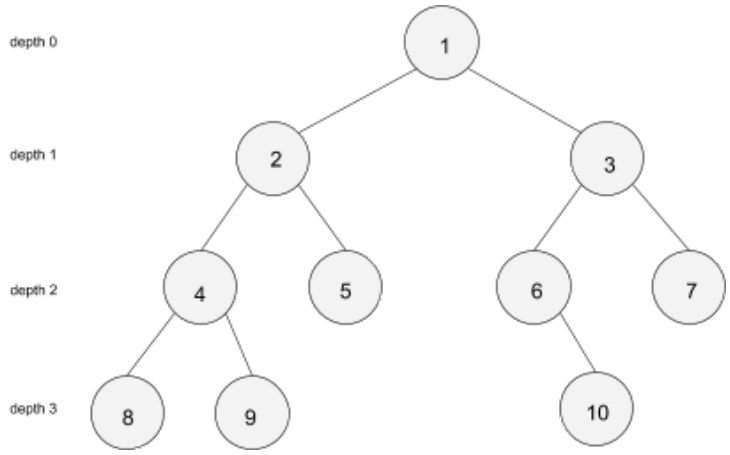



*Diagram by me.*

**Depth first search**

By contrast depth first search works by continually expanding nodes until the maximum depth is reached. This has the benefit of making it unnecessary to store the entire breadth of the tree in memory, but the drawback of not guaranteeing to find an optimal path, and potentially wandering off down the left most path indefinitely if the tree is infinite. (Korf 2010)

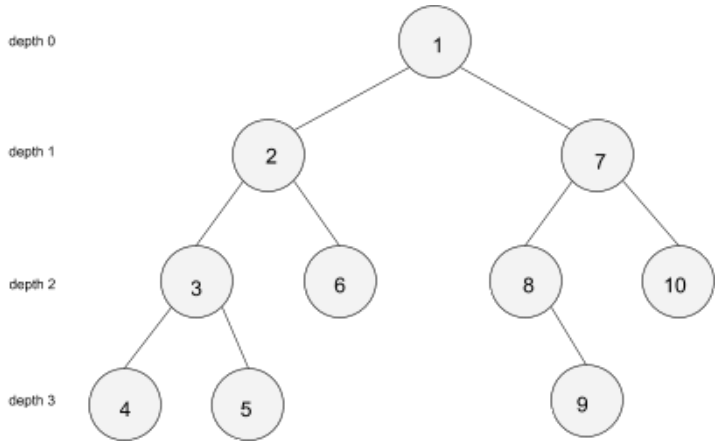



*Diagram by me.*

**Informed search**

When the goal state is known, that knowledge can be used to search in a more directed manner, using each node's distance from the goal to inform the direction of the search.

**Best first search**

Best first search uses the estimated distance from the goal of each possible successor node to order the frontier, giving priority to those nodes with the shortest estimated distance. This distance could be calculated by a number of methods – but more on that later. Best first offers a good trade off between the time and memory required to find a good path, and the proximity of that path to an optimal one. Best first does not guarantee to find the shortest possible path from the initial state to the goal, but it generally gets close and is more efficient than both breadth first and A*. Best first is a good algorithm to use if getting getting a close to optimal path is acceptable and time or memory is costly.

**A* search**

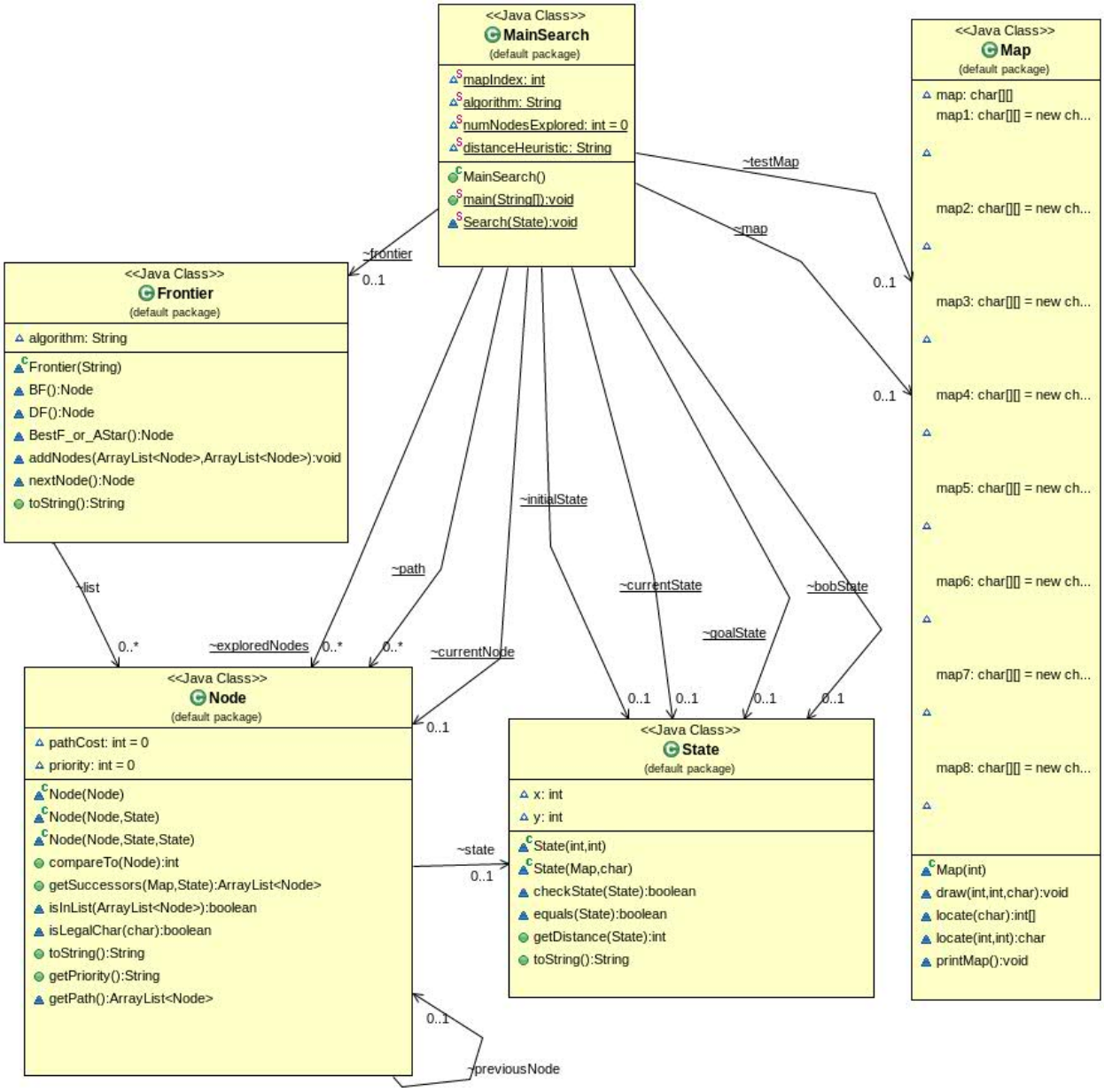
A* is actually the same algorithm as depth first, with the only difference being that A* prioritises the successor nodes by the sum of their estimated distance to the goal state, plus the known cost of the path to that node. This means that A* is guaranteed to find the optimal path if one is possible, but also that it expands many nodes in the process. A* is best used when finding the shortest possible path is more important that the time or memory spent searching for it, or when the state space is small enough that it doesn't matter.

## 4. THE PROBLEM: BOB AND THE ROBOT

Bob is stuck and cannot move! Fortunately, the robot is at hand to carry him to safety. We must find the optimal route to collect Bob and take him to the goal.

This search problem is represented as a state space consisting of a grid world, represented by a two dimensional char array. The world may contain obstacles (represented by ‘X’ in the array) which are impassable. The initial state is the agent’s (in this case, the robot’s) starting position in the state space (represented by ‘I’). Our goal is to get to the state where Bob is (‘B’), and then to the goal state (‘G’) by the shortest possible path. A successor function determines the possible actions the robot can take in the current state, namely moving one space to the north, south, east or west, as long as that space is not filled by an obstacle or outside of the map. The path cost is the number of actions (the length of the path) that the robot must take to reach first Bob, and then the goal.

## 5. IMPLEMENTATION




MainSearch controls the program and implements the search function. The Frontier object keeps track of the frontier with an ArrayList of Nodes and contains methods to update itself according to the algorithm used. The Node object contains Node constructors and various methods to access information about a given Node. The State object provides methods to get information about and compare the current state, specifically regarding the location of the robot, Bob and the goal. The Map object holds the various grid worlds and methods to edit and print them.

## 6. SEARCH STRATEGY

When the program is run, 3 arguments are taken: which map to use (from 1 - 8), the algorithm (depth first, breadth first, best first or A*) and the distance heuristic to be used if the search is best first or A* (Euclidean, Chebyshev or Manhattan).

The distance heuristic and algorithm choices are saved in their respective variables to be accessed at need. A new Map object is created to represent the state space given the user's map choice, along with a second Map which is a copy of the first used for printing the explored area and path.

The State constructor is then called three times to find the locations of Bob, the robot and the goal, and create State objects to represent their locations within the state space. The currentState instance of the State object is created to keep track of the current location of the robot.

An empty array list of Nodes is created to keep track of the Nodes that have already been explored.

A new Node is created to hold information about the current Node being explored, including its state, priority and previous node.

And finally, a new Frontier object is created to hold the frontier and allow us to update it.

Let the search begin! First, the robot must find Bob. This is implemented with a search method within a while loop. The search will be repeated until the current State is equal to Bob's state.

**Inside the search function:**

The current node is added to the explored nodes array, and the number of nodes explored counter is incremented. Keeping track of the total number of nodes explored will make it easier to compare different algorithms and heuristics later.

Some info is printed out for the user, including a map and a list of the explored states. This was very useful when writing and debugging the program, and makes it easier for the user to see how the search method is working.

Based on the current node, the next possible legal nodes to explore are calculated - this is the successor function, implemented using a Node class method getSuccessors which is passed the current map and the current goal state (either Bob or the goal) and checks which of the four possible moves are legal (e.g. not out of bounds, and not blocked by an obstacle). A new node for each legal move is constructed by creating a State object for each legal node, and passing it to the Node constructor, along with the current goal State. It's necessary to pass the goal State since when using Best First and A* search, the distance from the current state to the goal must be calculated in order to assign the node a priority within the frontier.

Assigning the priority of a node is done at the time of its construction using the State method getDistance, which calculates the distance from a given State to the goal State according to the user selected heuristic:

- Euclidean distance is the straight line distance between the two States - as the crow flies, if the crow has an excellent sense of direction. In my implementation this is multiplied by 100 and cast to an int before being used as the distance, since in order to sort the frontier I have implemented the Comparable interface, which requires an int.

- Chebyshev distance can be thought of as the largest of the differences between the respective coordinates of the two states - i.e. the difference between x1 and x2, and y1 and y2 are calculated, and the largest of those is used.
- The Manhattan distance is the distance between the two states using only horizontal or vertical paths, like navigating city blocks.

Now I have identified and created new nodes to represent the successor states which are legally reachable from the current state (which, if using Best First or A* have also been given priorities based on their distance from the goal), I must check if they should be explored and if so, add them to the frontier.

The addNodes method within the Frontier class does this by looping through the list of legal nodes, and checking if their states have already been explored. If so, the node is discarded, as there is no point exploring them again. Likewise, if the state already exists within the frontier the node is not kept - unless I'm using A* search and the node in question has a lower total path cost than the existing node in the explored nodes list. In this case, the new node is kept as I want the lowest path cost possible to any given state, and the existing node with the same state is removed from the frontier. Nodes with states that have not been explored and do not exist within the frontier (or do exist within the frontier, but have a lower path cost) are all added to the frontier. Since the frontier is represented by an array list, these new nodes are simply added to the back of the list.

Some information is then printed out for the user, namely the frontier to check which nodes have been added, along with the current state.

And here's the interesting part! The nextNode method is responsible for providing the next Node from the frontier for the robot to explore, given which search algorithm is being used - this is the crux of the search strategy and where the different algorithms are implemented.

- Breadth First uses a queue, so the first node into the frontier is the first out. To implement this, simply return the node at index 0 of the frontier.
- Depth First uses a stack, so the last node added is the first node explored. For this, just return the last node in the frontier.
- Best First is more complex as it takes into account the estimated distance from the current goal state of each node, which is represented by their priority attribute. (In the code, the higher the number given to the Node's priority attribute, the lower the priority of that node, since it corresponds to the estimated distance from the goal - so a node with a priority of 1 has the highest priority.) To get the next node, I order the frontier - this is done by implementing Java's Comparable interface and overriding its compareTo method to compare the priorities of two nodes, which allows me to sort the frontier by the priority of the nodes within using Collections.sort. The next node is then at index 0 in the frontier.
- A* is essentially the same as Best First, the only difference being the heuristic used to order the nodes. In Best First, it is just the Node's estimated distance from the current goal state, but in A*, it is the sum of that estimated distance to goal, plus the length of the actual path to that Node from the initial state (the path cost). This is calculated when each node is constructed - if A* is the algorithm used, a node's priority is set as its estimated distance from the goal plus its path cost. (The path cost is calculated by recursively looping accessing the node's previous node attribute until null is reached.) Because of this, there is no need to sort the frontier any differently in the nextNode method - so as with Best First I just sort by priority and return the node at index 0.

All that is left to do within the search function is to set the current node to the next node to explore as provided by the method described above, and update the currentState with the robot's new location.

Once Bob has been found, the robot must then search for the goal. The exploredNodes and frontier lists are cleared, and the search method is called again within a loop, passing in the goal state as the state to search for.

These two search loops are enclosed within a try block which will catch a NullPointerException - this will only happen if the frontier is empty, but Bob or the goal has not yet been found - meaning that there is no possible path to Bob, or from Bob to the goal. Poor Bob. The user is alerted in this case.

```
Search complete.
No path found. Sorry Bob.

Explored map:
#########X
########XO
######XXOO
######XOOO
#I###XXOBO
###XXXOXXO
#####XX##X
##XXXX####
####G#####
##########
```

When the goal state has been found, the path to it, via Bob, from the initial state is printed out.

```
Explored map:
##X######X
#XOX##B##X
#XXXXXXX##
##########
##########
###XXXX###
##XX######
###X######
#XXX######
G#I#######
Explored in current search: [6,1, 6,0, 7,1, 5,1, 7,0, 5,0, 8,1, 4,1, 8,0, 4,0, 8
Current position: 0,8
Frontier:
State: 0,9 Priority: 0

Found Goal!
Path: [3,9, 4,9, 4,8, 4,7, 4,6, 5,6, 6,6, 7,6, 7,5, 7,4, 7,3, 8,3, 8,2, 8,1, 7,1

OOXOOOOOOX
OXOXOOB##X
OXXXXXXX#O
OOOOOOO##O
OO#######O
O##XXXX#OO
O#XX####OO
##OX#OOOOO
#XXX#OOOOO
GOI##OOOOO

Explored 144 nodes

Path length is 34
```

## 7. TESTING

One of the first things I did was create a Map class and write some methods to allow the explored area to be printed to the console at each step during search as a 2d array. This proved invaluable in testing as I was able to see the area that had been and was yet to be explored at a glance, which made it very easy to see whether the algorithm was behaving as expected.

```
Explored map:
I##OOOOOOO
OO##OOOOOO
OOO##OOOOO
OOOO#XOOXO
OOOX###OOO
OOOOOO#OOO
OOXXOXOOOO
OOOOOOOOBO
OOOOOOOOOO
OOOOOOOOOG
Explored in current search: [0,0, 1,0, 2,0, 2,1, 3,1, 3,2, 4,2, 4,3, 4,4, 5,4, 6,4, 6,5]
Current position: 6,5
Frontier:
State: 5,5 Priority: 3
State: 7,4 Priority: 3
State: 4,5 Priority: 4
State: 6,3 Priority: 4
State: 3,3 Priority: 5
State: 5,2 Priority: 5
State: 2,2 Priority: 6
State: 4,1 Priority: 6
State: 1,1 Priority: 7
State: 3,0 Priority: 7
State: 0,1 Priority: 8
State: 7,5 Priority: 2
State: 6,6 Priority: 2
```

I also included a lot of print statements to show the current explored nodes, the frontier, and the current state of the robot - and once Bob and the goal had or had not been found, the optimal path discovered.

Based on my understanding of the search algorithms and how they should work, this visual feedback was enough for me to identify any problems with the implementation. I ran my program repeatedly with different maps (and created two additional test maps of my own) to ensure that it behaved predictably. I walked through each of these tests step by step, checking that at each stage the nodes were explored in the order I expected.

## 8. EXAMPLES & EVALUATION

In order to facilitate the comparison of different algorithms and distance heuristics over a range of maps, I created a copy of the program and added another class called Compare which enabled me to automatically iterate over the various different combinations of maps, search algorithms and distance heuristics, rather than typing each manually. This doesn’t have a user interface, but by directly modifying the code in the Compare class, I can print out results for each possible combination. This makes comparison a bit easier.

Here are a few example results:

Using the first map only with best first search, and different distance heuristics. Of course only best first and A* search are affected by the distance heuristic choice - breadth and depth first don't take it into account. You can see that the path found by best first search is actually affected to quite a large extent by the measure used to calculate the distance to the goal, and is not guaranteed to be optimal.

```
distance measure: e
map: 1
algorithm: BestF
IOOOOOOOOO
##OOOOOOOO
O####OOOOO
OOOO#XOOXO
OOOX#OOOOO
OOOO###OOO
OOXXOX###O
OOOOOOOOB0
OOOOOOOO##
OOOOOOOOOG
Explored 29 nodes
Path length is 18
```

```
distance measure: m
map: 1
algorithm: BestF
I########O
OOOOOOOO#O
OOOOOOOO##
OOOOOXOOX#
OOOXOOOOO#
OOOOOOOOO#
OOXXOXOOO#
OOOOOOOOB#
OOOOOOOO#
OOOOOOOOOG
Explored 20 nodes
Path length is 20
```

```
distance measure: c
map: 1
algorithm: BestF
I##OOOOOOO
OO##OOOOOO
OOO##OOOOO
OOOO#XOOXO
OOOX###OOO
OOOOOO##OO
OOXXOXO##O
OOOOOOOOB0
OOOOOOOO##
OOOOOOOOOG
Explored 18 nodes
Path length is 18
```

```
distance measure: com
map: 1
algorithm: BestF
I########O
OOOOOOOO#O
OOOOOOOO##
OOOOOXOOX#
OOOXOOOOO#
OOOOOOOOO#
OOXXOXOOO#
OOOOOOOOB#
OOOOOOOO#
OOOOOOOOOG
Explored 20 nodes
Path length is 20
```

Let's try again with A*, which should always find the best path possible, no matter what the distance measure used. You can see below that although A* explores many more nodes in the process and that the path produced may vary, it will always be of the shortest possible length. Combining the manhattan and chebyshev distances (the 'com' for combined distance measure) has no real effect.

```
distance measure: e
map: 1
algorithm: A*
IOOOOOOOOO
##OOOOOOOO
O####OOOOO
OOOO#XOOXO
OOOX#OOOOO
OOOO###OOO
OOXXOX###O
OOOOOOOOB0
OOOOOOOO##
OOOOOOOOOG
Explored 29 nodes
Path length is 18
```

```
distance measure: m
map: 1
algorithm: A*
I#######OO
OOOOOOO#OO
OOOOOOO#OO
OOOOOXO#XO
OOOXOOO##O
OOOOOOOO#O
OOXXOXOO#O
OOOOOOOOB#
OOOOOOOOO#
OOOOOOOOOG
Explored 70 nodes
Path length is 18
```

```
distance measure: c
map: 1
algorithm: A*
I##OOOOOOO
OO##OOOOOO
OOO##OOOOO
OOOO#XOOXO
OOOX###OOO
OOOOOO##OO
OOXXOXO##O
OOOOOOOOB0
OOOOOOOO##
OOOOOOOOOG
Explored 70 nodes
Path length is 18
```

```
distance measure: com
map: 1
algorithm: A*
I#######OO
OOOOOOO#OO
OOOOOOO#OO
OOOOOXO#XO
OOOXOOO##O
OOOOOOOO#O
OOXXOXOO#O
OOOOOOOOB#
OOOOOOOOO#
OOOOOOOOOG
Explored 70 nodes
Path length is 18
```

Comparing breadth first, depth first, best first and a* on map 3, which is more complex. Best first and A* use chebyshev distance here. Again, best first succeeded in finding an optimal path, and explored far fewer nodes than A*. Breadth first also found the shortest path, although it explored many more nodes. *NB: The map is only 10 x 10, but breadth first explored 126 nodes - this is because when Bob is found, we must search again for the best path to the goal.*

```
map: 3
algorithm: BF
OOOOOOOOOO
O########O
O#OOOOXO#O
O#OOOOXO#O
OIOOOXXOBO
OOOXXXOO#O
OOOOOXOO#O
OOXXXXOO#O
OOOOG####O
OOOOOOOOOO
Explored 126 nodes
Path length is 21
```

```
map: 3
algorithm: DF
OOOOOOOOOO
OOOOOOOOOO
OOOOOOXOOO
OOOOOOXOOO
#IOOOXX#BO
#OOXXX###O
#OOOOX#OOO
#OXXXX####
#OOOG##OO#
##########
Explored 73 nodes
Path length is 33
```

```
distance measure: c
map: 3
algorithm: BestF
OOOOOOOOOO
OOOOO###OO
OOOOO#X#OO
OOOO##X##O
OI###XXOBO
OOOXXXO##O
OOOOOX##OO
OOXXXX#OOO
OOOOG##OOO
OOOOOOOOOO
Explored 21 nodes
Path length is 21
```

```
distance measure: c
map: 3
algorithm: A*
OOOOOOOOOO
OOOOO###OO
OOOOO#X#OO
OOOO##X##O
OI###XXOBO
OOOXXXO##O
OOOOOX##OO
OOXXXX#OOO
OOOOG##OOO
OOOOOOOOOO
Explored 82 nodes
Path length is 21
```

Comparing breadth first, depth first, best first and a* on some maps of my own creation, using manhattan distance. A* always finds the best route, but as shown below, it requires many more nodes to be searched. For this reason, I expect that best first is often used preferentially in situations where the state space is very large, or time or memory is limited. Breadth first also always finds the shortest path, but requires still more nodes to be searched.

```
map: 7
algorithm: BF
O########X
OOOO#XXO##
OXOX#OXOOB
OXOX#OXXOO
G####XXXXO
XOOXXXXOOO
XXOOOOOOXX
OOOXXXOXOX
OXXX00XXOO
OOOOOOOXOO
Explored 89 nodes
Path length is 26
```

```
map: 7
algorithm: DF
########OX
#OOOOXX#OO
#XOXOOX##B
#XOXOOXXO#
G#OOOXXXX#
X##XXXX###
XX######XX
OOOXXXOXOX
OXXX00XXOO
OOOOOOOXOO
Explored 71 nodes
Path length is 34
```

```
distance measure: m
map: 7
algorithm: A*
O########X
OOOOOXXO##
OXOXOOXOOB
OXOXOOXXO#
G##OOXXXX#
XO#XXXX###
XX######XX
OOOXXXOXOX
OXXX00XXOO
OOOOOOOXOO
Explored 66 nodes
Path length is 26
```

```
distance measure: m
map: 7
algorithm: BestF
O########X
OOOOOXXO##
OXOXOOXOOB
OXOXOOXXO#
G##OOXXXX#
XO#XXXX###
XX######XX
OOOXXXOXOX
OXXX00XXOO
OOOOOOOXOO
Explored 29 nodes
Path length is 26
```

```
map: 8
algorithm: BF
IOOOOOOOOO
#OOOOOOOOO
#OOOOOOOOO
#OOOOXOOXO
#OOXOOOOOO
#OOOOOOOOO
#XXXOXOOOO
##B#XOOOOO
OOO#######
OOOOOOOOOG
Explored 97 nodes
Path length is 18
```

```
map: 8
algorithm: DF
IOOOOOOOOO
#OOOOOOOOO
#OOOOOOOOO
#OOOOXOOXO
#OOXOOOOOO
#OOOOOOOOO
#XXXOXOOOO
##B#XOOOOO
#OO#OOOOOO
#########G
Explored 95 nodes
Path length is 28
```

```
distance measure: m
map: 8
algorithm: BestF
I##OOOOOOO
OO#OOOOOOO
OO#OOOOOOO
OO#OOXOOXO
OO#XOOOOOO
###OOOOOOO
#XXXOXOOOO
##B#XOOOOO
OOO#######
OOOOOOOOOG
Explored 26 nodes
Path length is 22
```

```
distance measure: m
map: 8
algorithm: A*
IOOOOOOOOO
#OOOOOOOOO
#OOOOOOOOO
#OOOOXOOXO
#OOXOOOOOO
#OOOOOOOOO
#XXXOXOOOO
##B#XOOOOO
OOO#######
OOOOOOOOOG
Explored 38 nodes
Path length is 18
```

Comparing memory usage: here you can see that whilst breadth first will find an optimal solution, to do so it may require up to the entire state space to be stored in memory. This may be unworkable depending on the application. Here is breadth first's explored list and frontier from map 1, right before Bob was found:

```
Explored in current search: [0,0, 1,0, 0,1, 2,0, 1,1, 0,2, 3,0, 2,1, 1,2, 0,3, 4,0, 3,1, 2,2,
1,3, 0,4, 5,0, 4,1, 3,2, 2,3, 1,4, 0,5, 6,0, 5,1, 4,2, 3,3, 2,4, 1,5, 0,6, 7,0, 6,1, 5,2, 4,3,
2,5, 1,6, 0,7, 8,0, 7,1, 6,2, 4,4, 3,5, 1,7, 0,8, 9,0, 8,1, 7,2, 6,3, 5,4, 4,5, 2,7, 1,8, 0,9,
9,1, 8,2, 7,3, 6,4, 5,5, 4,6, 3,7, 2,8, 1,9, 9,2, 7,4, 6,5, 4,7, 3,8, 2,9, 9,3, 8,4, 7,5, 6,6,
5,7, 4,8, 3,9, 9,4, 8,5, 7,6, 6,7, 5,8, 4,9, 9,5, 8,6, 7,7, 6,8, 5,9, 9,6]
Current position: 9,6
Frontier:
State: 8,7
State: 7,8
State: 6,9
State: 9,7
```

Breadth first found the solution to map 1, with a path cost of 18, after exploring 101 nodes. To compare, here is depth first's explored list and frontier for the same map, at the same stage.

```
Explored in current search: [0,0, 0,1, 0,2, 0,3, 0,4, 0,5, 0,6, 0,7, 0,8, 0,9, 1,9, 2,9, 3,9,
4,9, 5,9, 6,9, 7,9, 8,9, 9,9, 9,8, 9,7]
Current position: 9,7
Frontier:
State: 1,0
```

```
State: 1,1
State: 1,2
State: 1,3
State: 1,4
State: 1,5
State: 1,6
State: 1,7
State: 1,8
State: 2,8
State: 3,8
State: 4,8
State: 5,8
State: 6,8
State: 7,8
State: 8,8
State: 9,6
State: 8,7
```

Depth first solved map 1 with a path cost of 40, but had to explore only 40 nodes to do it. This, of course, is difficult to rely on, as the success of the depth first algorithm depends very much on the state space it is exploring - if the search tree is very deep, depth first's frontier would grow and grow (potentially infinitely) - in the same way that if the tree is very broad, breadth first's list of explored nodes would be very large and demand a lot of memory. For this reason the best choice of algorithm really depends on the nature of the state space (at least if you're choosing between depth first and breadth first).

**IMPROVEMENTS**

Naturally had I more time I would have liked to extend the functionality of my system with the additions in Part 3. My code works, but I believe it could do with a bit of tidying up in terms of encapsulation, checking user input, and possibly refactoring some of the methods or restructuring the classes. On the whole I believe my implementation is solid and implements the search algorithms correctly. In particular, my use of maps to visually portray the explored areas and the final path found is useful, as is my testing methodology which allowed me a comprehensive view of the possible results of each algorithm and heuristic.

**WORDCOUNT: 4067**

**REFERENCES**


Dolgov, Dmitri, Sebastian Thrun, Michael Montemerlo, and James Diebel. 2008. "Practical Search Techniques in Path Planning for Autonomous Driving." *Proceedings of the First International Symposium on Search Techniques in Artificial Intelligence and Robotics*. http://ai.stanford.edu/~ddolgov/papers/dolgov_gpp_stair08.pdf.

Korf, Richard E. 2010. "Artificial Intelligence Search Algorithms 22.1." In *Algorithms and Theory of Computation Handbook 2*, 22–22. CRC Press. https://dl.acm.org/citation.cfm?id=1882745.

Millington, Ian, and John Funge. 2009. "Pathfinding." In *Artificial Intelligence for Games*, 197–291. doi:10.1016/B978-0-12-374731-0.00004-9.

Zaheer, Kamran. 2006. "Artificial Intelligence Search Algorithms In Travel Planning." *Algorithms {&} Theory of Computation Handbook*, 36(1--20). http://www.idt.mdh.se/utbildning/exjobb/files/TR0654.pdf.