Anujay Sharma

Basic Search Algorithms

Coding and Algorithms Explanations

Common Utilities:

Class - > Node:

We use Nodes for the algorithm explained below. Node is a class which can be used to create objects as nodes of the search tree.

The Node class has 7 attributes namely,

1) row -> Co-ordinate of row (int)
2) col -> Co-ordinate of column (int)
3) is_obs -> If object is obstacle (bool)
4) g -> The cost to come (int)
5) h -> The heuristic cost (int)
6) cost -> The total cost (int)
7) parent -> Parent of the current node (Node)

Every time is node is explored the above parameters are assigned to that node. These attributes help in assigning costs, backtracking and accessing the row and column numbers.

Function -> _neighbour:

The _neighbour function gives out all the possible neighbors of the current node in four directions. These directions are in the order,

"right, down, left, up”, which means "[0, +1], [+1, 0], [0, -1], [-1, 0]"

Only the neighbors who are inside the grid and are not obstacles are appended to the list.

ds

Algorithms:

Breadth First Search (BFS):

Breadth first search is a graph traversal algorithm for searching data points in a tree. It starts at the source of the tree or the root and the explores all the nodes at one depth before moving on to the next depth. The decision at one depth could be taken randomly or according to a specified order.

eg, if we want to a family member for a particular generation inside the family tree, we will use BFS for fast results.

Code structure and explanation:

Pseudo Code:

BFS (grid, start, goal):

Initialize Start Node

Add Node to Visited Set

Stack Start node in Queue

loop -> if queue not empty or found

if: node < - goal:

found <- True

break

else: add possible neighbors

check visit and bound criteria

append neighbors to queue

parent < - source node

pop current node <- source node

break

Explanation:

At first, we initialize the start node and add it to the visited list. Since the source node would not have any parent, its parent attribute would be none. We then begin our loop to add the valid neighbors. Valid neighbors are the ones which are inside the grid, are not obstacles and have not already been visited.

ds

These neighbors get assigned a parent and are added to a stack/queue. One by one we pop the neighbors from the beginning of the stack as in to follow FIFO system. Once the goal is achieved, we break the loop.

The path is achieved by logging on to the parent of the node and looping until the root node is not achieved.

Depth First Search (DFS):

Depth first search is a graph traversal algorithm for searching data points in a tree. It starts at the source of the tree or the root and the explores deeper into the tree until either a goal is achieved, or a dead end is met. If it meets a dead end, it goes back to the branch as start another traversal down the depth.

eg, if we want to a family member who is known to be an ancestor and is many depths below the current generation. In such cases DFS would be faster as compared to BFS.

Code structure and explanation:

Pseudo Code:

DFS (grid, start, goal):

Initialize Start Node

Add Node to Visited Set

Stack Start node in Queue

loop -> if queue not empty or found

if: node < - goal:

found <- True

break

else: add possible neighbors

reverse the stack

check visit and bound criteria

append neighbors to queue

parent < - source node

pop last node <- source node

break

ds

Explanation:

At first, we initialize the start node and add it to the visited list. Since the source node would not have any parent, its parent attribute would be none. We then begin our loop to add the valid neighbors. Valid neighbors are the ones which are inside the grid, are not obstacles and have not already been visited.

These neighbors get assigned a parent and are added to a stack/queue. We stack the queue in reverse of our priority order. The node on top of the stack is then picked for exploration. This follows the LIFO system.

The path is achieved by logging on to the parent of the node and looping until the root node is not achieved.

Dijkstra:

Dijkstra is a graph traversal algorithm for searching data points in a tree. It is like BFS however in this case we also choose the weights of the connected paths. This step is important in comparing the shortest cost to be explored by the parent node.

eg, if we want to go from one place in a city to another but also have delays like traffic, bad road conditions etc. which could cause a longer duration for travel. We would weigh our options carefully for such instances.

Code structure and explanation:

Pseudo Code:

Dijkstra (grid, start, goal):

Initialize Start Node

Add Node to Visited Set

Stack Start node in Queue

loop -> if queue not empty or found

if: node < - goal:

found <- True

break

else: add possible neighbors

check visit and bound criteria

append neighbors to queue

ds

parent < - source node

cost to come <- assign to new node

sort -> lowest costing node

pop first node from sort <- source node

break

Explanation:

At first, we initialize the start node and add it to the visited list. Since the source node would not have any parent, its parent attribute would be none. We then begin our loop to add the valid neighbors. Valid neighbors are the ones which are inside the grid, are not obstacles and have not already been visited.

These neighbors get assigned a parent. Here we also assign a cost to the new node. This cost is the total cost from start node. This cost helps us in identifying which node we need to explore. The lowest costing node is explored first.

The path is achieved by logging on to the parent of the node and looping until the root node is not achieved.

A*:

A* is a graph traversal algorithm for searching data points in a tree. It is like Dijkstra however in this case we also choose some heuristics for the end goal. This is done to get a sense of direction towards the goal.

eg, A taxi driver in a city can go through several routes. They can make a choice based on the conditions of road, traffic etc. and the length of the route. The best decision is the one which incorporates both

Code structure and explanation:

Pseudo Code:

A* (grid, start, goal):

Initialize Start Node

Add Node to Visited Set

Stack Start node in Queue

loop -> if queue not empty or found

if: node < - goal:

ds

found <- True

break

else: add possible neighbors

check visit and bound criteria

append neighbors to queue

parent < - source node

cost to come <- assign to new node

heuristic <- assign to new node

total cost <- assign to new node

sort -> lowest costing node

pop first node from sort <- source node

break

Explanation:

At first, we initialize the start node and add it to the visited list. Since the source node would not have any parent, its parent attribute would be none. We then begin our loop to add the valid neighbors. Valid neighbors are the ones which are inside the grid, are not obstacles and have not already been visited.

These neighbors get assigned a parent. Here we also assign a cost to the new node. This cost is the total cost to traverse from start node. In addition to this we also add a heuristic cost. This cost is the cost to reach the goal. The total cost helps us in identifying which node we need to explore. The lowest total costing node is explored first.

The path is achieved by logging on to the parent of the node and looping until the root node is not achieved.

Overall Observations:

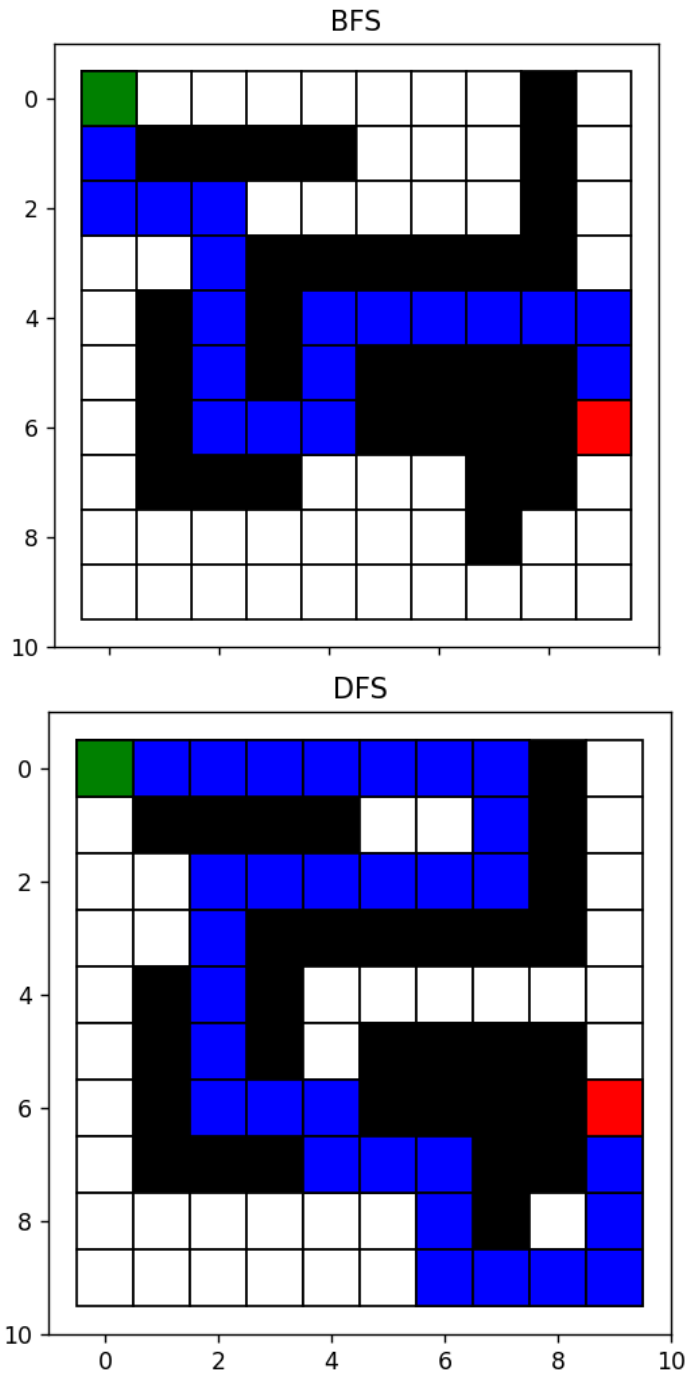
On running these algorithms, we find that A* gives the shortest path in most cases. DFS could also give a short path but if it takes a wrong node in priority then the number of steps as well the path increases drastically. In our assignment since the weight for each node is 1, we achieve almost similar performance from Dijkstra and BFS algorithms.

ds

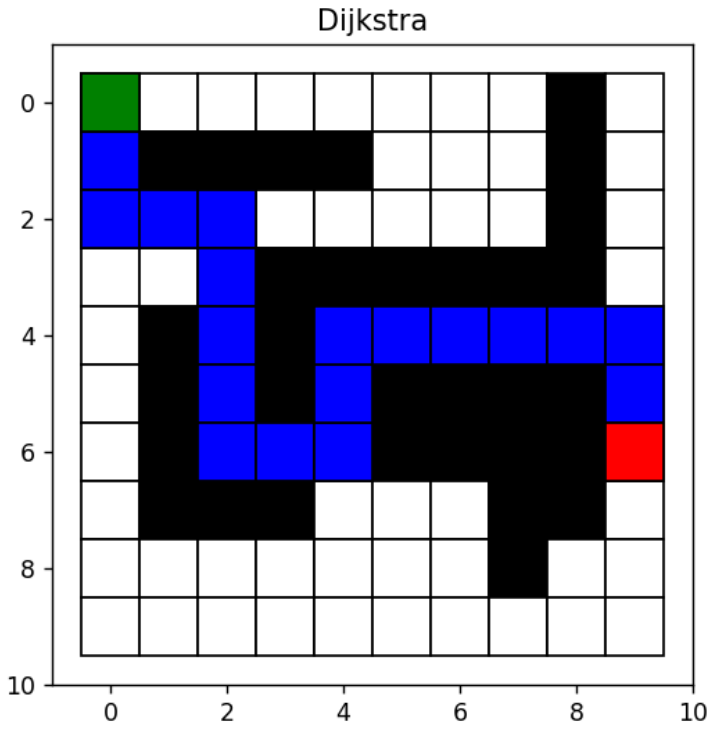
Outputs:

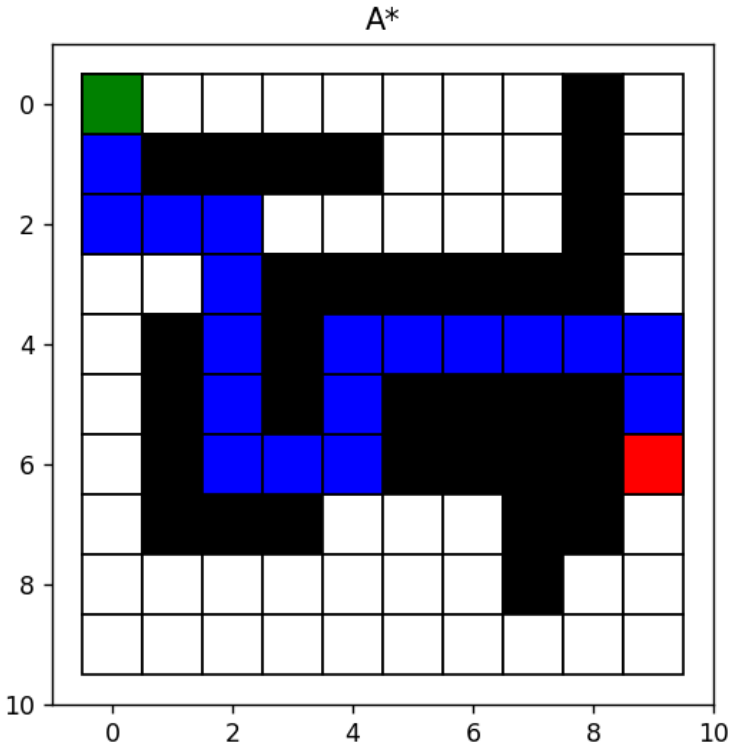
These observations are observed for the default maze, start and goal positions.

```
It takes 64 steps to find a path using BFS
It takes 33 steps to find a path using DFS
It takes 64 steps to find a path using Dijkstra
It takes 51 steps to find a path using A*
```



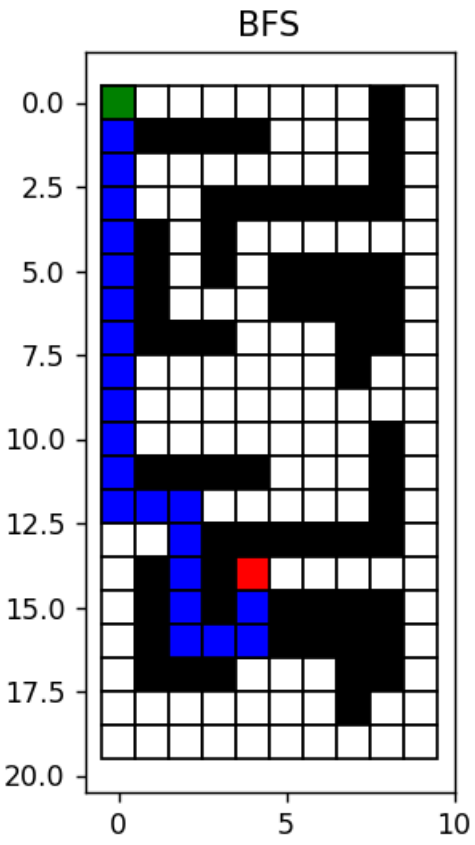
ds

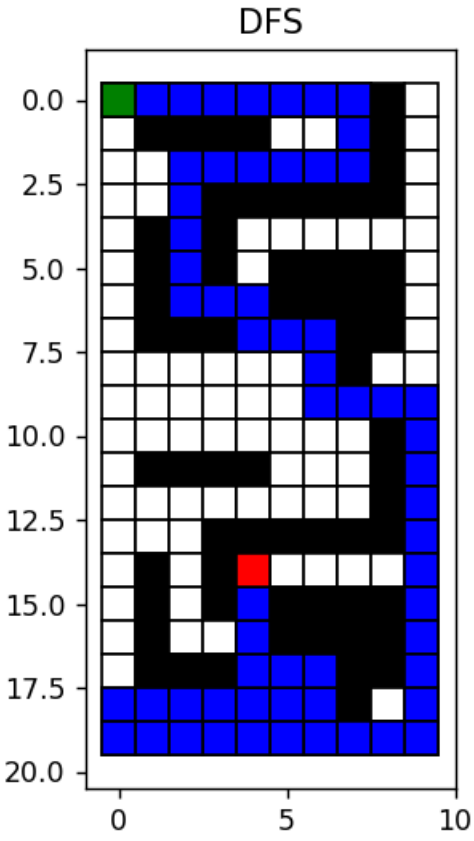


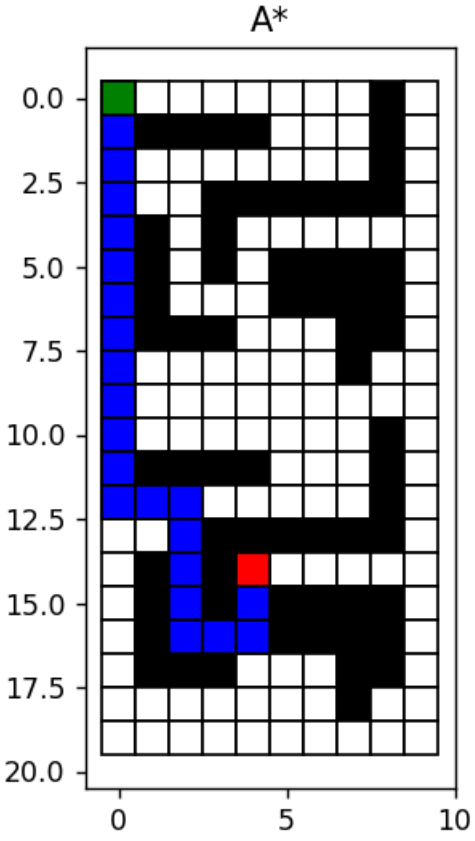


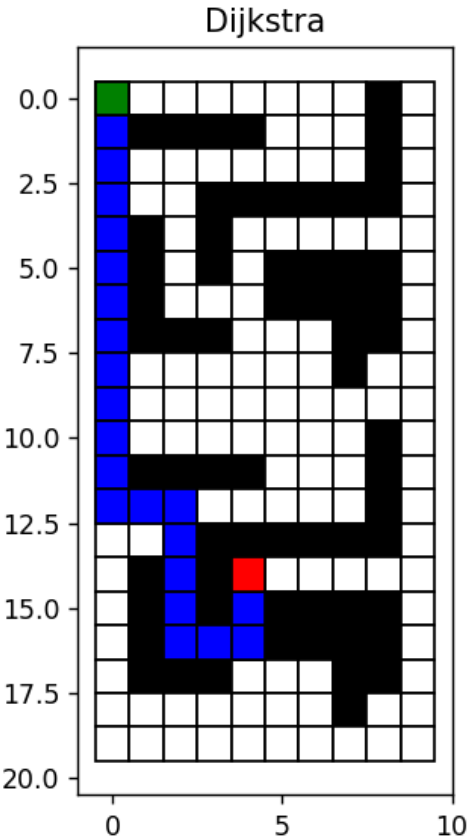
I also got to experiment with some other maps of my own where I applied the same algorithms to see the results.

ds


BFS
0.0
2.5
5.0
7.5
10.0
12.5
15.0
17.5
20.0
0
5
10


DFS
0.0
2.5
5.0
7.5
10.0
12.5
15.0
17.5
20.0
0
5
10


A*
0.0
2.5
5.0
7.5
10.0
12.5
15.0
17.5
20.0
0
5
10


Dijkstra
0.0
2.5
5.0
7.5
10.0
12.5
15.0
17.5
20.0
0
5
10

As we see here DFS takes a very long path to traverse. This is because the priority queue is set in a particular way. We also observe that A* takes the minimum number of steps. Th performances for BFS and Dijkstra are similar because of the unweighted or single weight graph.

References:

1. **YouTube – Abdul Bari [**Lecture Series on Graph Traversals.]
2. **YouTube – NeetCode [**Tree Traversal Order - BFS**]**
3. **Wikipedia – [**A* search algorithm**]**
4. **YouTube – John Levine [**A* Search, Uniform Cost Search**]**

ds