Basic Algorithm Implementation

Pranav Moorthy

1. BREADTH FIRST SEARCH

Explanation for implemented code:

```
step 1: Create an empty list q, coor and visited
step 2: Create a object of type Node for the starting co-ordinates with a null parent
step 3: Append node to the list q
step 4: while q is not empty repeat steps 4.1 to 4.6
        4.1 pop first element and assign to prevNode
        4.2 append to visited
        4.3 Visit the neighbours and ignore them if they are obstacles or exceed map dimensions
        4.4 if they are not in q and not in visited
                4.4.1 Create a Node with the row and column and the parent of the node is prevNode
                4.4.2 Append to q
        4.5 If goal is reached
                4.5.1 Assign True to found
                4.5.2 Trace the parents with the help of parent_trace which returns path
        4.6 increment steps
step 5: return path and steps
```

```
parent_trace(path, finalNode):
step 1: Assign finalNode to rp
step 2 : while rp.parent is not null repeat steps 2.1 to 2.2
        2.1  append [rp.parent.row, rp.parent.col] to path
        2.2 Assign rp.parent to rp
step 3 : Return path
```

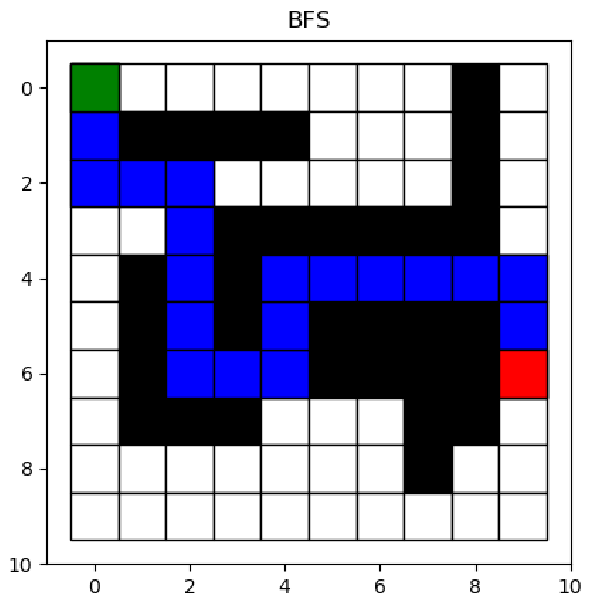
Explanation:

In the Breadth first search algorithm, A node is created for the starting point and appended to the list q. The neighbours of the node are searched and are appended provided they satisfy the following conditions:
a) They are not obstacles
b) They are within the grid
c) They are not visited and not in the queue

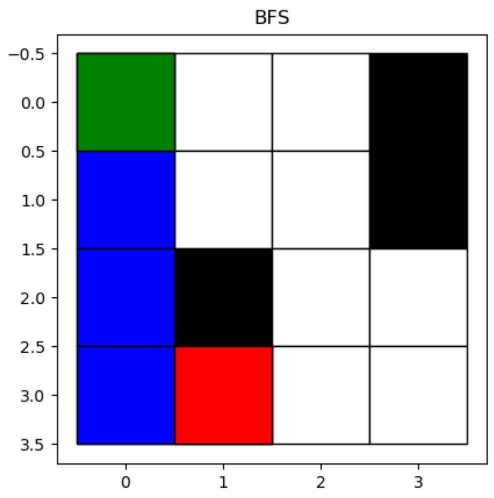
Nodes are created with the current node as the parents of the new Node and the new Nodes are appended to the queue. If the goal is reached parent_trace is used to trace the parents of the goal upto the start point as the parent of the start node is none. This returns the shortest path.

Test Code Result:

Map Result



Test Map output:



RBE 550

Pranav Moorthy

Algorithm Implementation

1. Breadth First search

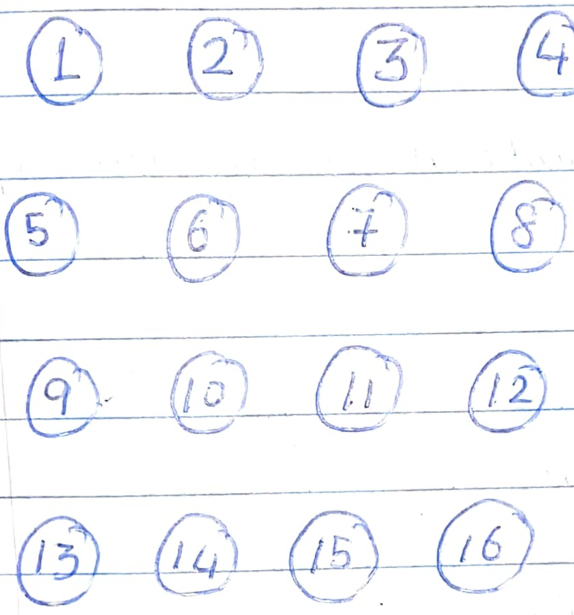
Working of the algorithm:

Let us take the example given in search.py

The map is

$$\begin{matrix} 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 1 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{matrix}$$

We shall name all points as Node with no's from 1 to 16



The exploration order is
[Right, Down, Left, Up]

We start with creating a beginning node & append to a list q

Start in this case is 1 ([0,0])

Note: The parent nodes are given in the blue ink

$q = \textcircled{1}$

Check the neighbours & append if they are

i) not visited

ii) not obstacles

iii) within the grid

$$\therefore \; q = \begin{array}{ccc} \textcircled{1} & \textcircled{2} & \textcircled{5} \\ \phi & 1 & 1 \end{array}$$

pop the first element after this process & add neighbours of the next node till goal reached

$$q = \begin{array}{cccc} \textcircled{2} & \textcircled{5} & \textcircled{3} & \textcircled{6} \\ 1 & 1 & 2 & 2 \end{array}$$

$$q = \begin{array}{cccc} \textcircled{5} & \textcircled{3} & \textcircled{6} & \textcircled{9} \\ 1 & 2 & 2 & 5 \end{array}$$

$$q = \begin{array}{cccc} \textcircled{3} & \textcircled{6} & \textcircled{9} & \textcircled{7} \\ 2 & 2 & 5 & 3 \end{array}$$

$q =$

$q = \textcircled{6}\ \textcircled{9}\ \textcircled{7}$

2 5 ~~7~~ 6

$q = \textcircled{9}\ \textcircled{7}\ \textcircled{13}$

5 ~~7~~ 6 9

$q = \textcircled{7}\ \textcircled{13}\ \textcircled{11}$

~~7~~ 6 9 7

$\cancel{\textcircled{13}}$

$q = \textcircled{13}\ \textcircled{11}\ \textcircled{14}$

9 7 13

$q = \textcircled{11}\ \textcircled{14}\ \textcircled{12}\ \textcircled{15}$

7 13 11 11

$q = \textcircled{14}\ \textcircled{12}\ \textcircled{15}$

13 11 11

The goal node is reached

& the path is traced back to the ~~origu~~ start node

~~The parent of~~

~~Node~~ :

The parents nodes are traced back to the start node

Current Node	(i, j)	Parent Node	(i, j)
14	(3, 1)	13	(3, 0)
13	(3, 0)	9	(2, 0)
9	(2, 0)	5	(1, 0)
5	(1, 0)	1	(0, 0)

$\therefore$ The path is given by

[[0, 0], [1, 0], [2, 0], [3, 0], [3, 1]]

& it takes 10 steps to reach here.

Depth First Search:

Explanation for Implemented code:

```
step 1: Create an empty list q, visited
step 2: Create a object of type Node for the starting co-ordinates with a null parent
step 3: Append node to the list q
step 4: while q is not empty repeat steps 4.1 to 4.6
        4.1 pop the last element of q and assign to prevNode
        4.2 append to visited
        4.3 Visit the neighbours and ignore them if they are obstacles or exceed map dimensions
        4.4 if they are not in visited
                4.4.1 Create a Node with the row and column and the parent of the node is prevNode
                4.4.2 Append to q
        4.5 If goal is reached
                4.5.1 Assign True to found
                4.5.2 Trace the parents with the help of parent_trace which returns path
        4.6 increment steps
step 5: return path and steps
```

```
parent_trace(path, finalNode):
step 1: Assign finalNode to rp
step 2 : while rp.parent is not null repeat steps 2.1 to 2.2
        2.1  append [rp.parent.row, rp.parent.col] to path
        2.2 Assign rp.parent to rp
step 3 : Return path
```

Code Explanation

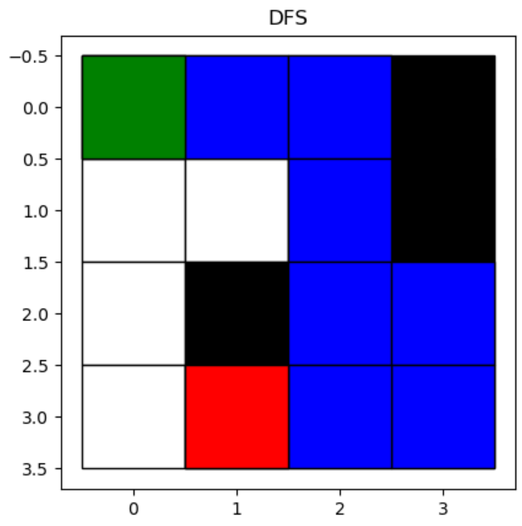
In the Depth first search algorithm, A node is created for the starting point and appended to the list q. The neighbours of the node are searched and are appended provided they satisfy the following conditions:
a) They are not obstacles
b) They are within the grid
c) They are not visited and not in the queue

Nodes are created with the current node as the parents of the new Node and the new Nodes are appended to the queue. Depth first search differs from Breadth First search in the sense that DFS explores a particular path before coming back to the start. This is done by popping the last element of the queue instead of the first.

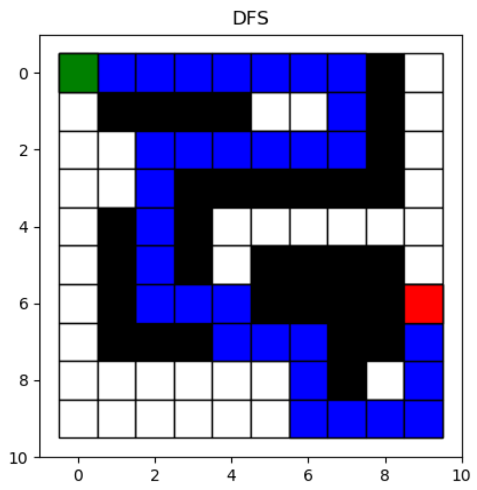
If the goal is reached parent_trace is used to trace the parents of the goal upto the start point as the parent of the start node is none. This returns the shortest path.

ALGORITHM RESULTS:

Test Map Result



Map Results:



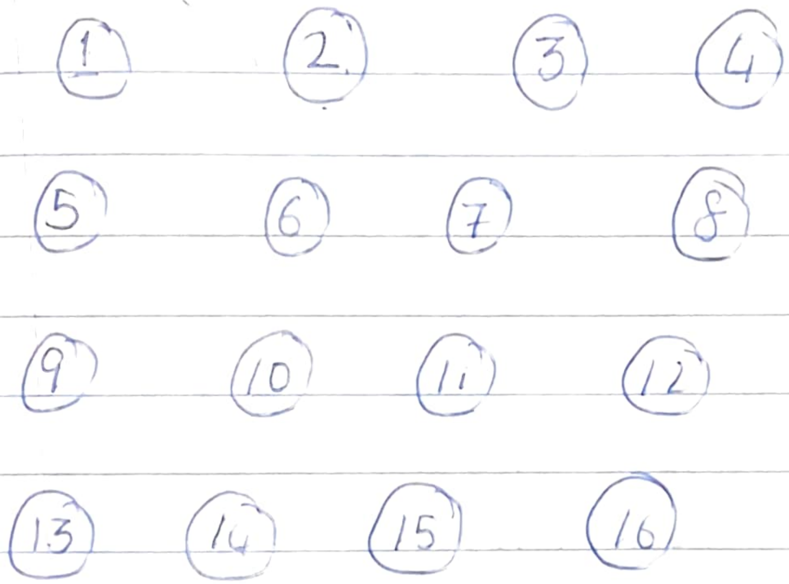
Depth First Search:

Working of the codes :

Let us use the grid given in search.py as example for an demon

0	0	0	1
0	0	0	1
0	1	0	0
0	0	0	0

Let us take all the nodes & number them from 1 to 16.



Start Node : 1
Goal Node : 14

In depth first search the decision is made to traverse a path from start node & we come back if the goal is not reached.

This is done by popping the ~~que~~ last element of the list q which is a stack (Last In First Out)

The neighbour ~~explori~~ ~~o~~ appending order is [Up, Left, Down, Right]

~~q = 1~~

~~Vi q = 1 5 2~~

q = 1
pop 1
and append the neighbours

q =	5	2
	1	1

q =	5	6	3
	1	2	2

q =	5	6	7
	1	2	3

q =	5	6	11
	1	2	7

q =	5	6	15	12
	1	2	11	11

q =	5	6	15	16
	1	2	11	12

q =	5	6	15	15
	1	2	11	16

q =	5	6	15	14
	1	2	11	15

We have reached the goal now we shall trace the path back

Node	(i,j)	Parent	(i,j)
14	(3,1)	15	(3,2)
15	(3,2)	16	(3,3)
16	(3,3)	12	(2,3)
12	(2,3)	11	(2,2)
11	(2,2)	7	(1,2)
7	(1,2)	3	(0,2)
3	(0,2)	2	(0,1)
2	(0,2)	1	(0,0)

The path is

[[0,0],[0,1],[0,2],[1,2],[2,2],[2,3],[3,
[3,2],[3,1]]

with 9 steps

A * Algorithm

Explanation for Implemented code

```
step 1: Create an empty list q and visited
step 2: Create a object of type Node for the starting co-ordinates with a null parent with cost as 0 and
heuristic as manhattan distance between start and goal
step 3: Append node to the list q
step 4: while q is not empty repeat steps 4.1 to 4.9
        4.1 Sort q based on sum of cost and heuristic
        4.2 pop first element and assign to prevNode
        4.3 append to visited
        4.4 assign temp as 0
        4.5 Visit the neighbours and ignore them if they are obstacles or exceed map dimensions
        4.6 if they are not in visited and not in q
                4.6.1 Create a Node with the row and column and the parent of the node is prevNode
                4.6.2 Assign cost of node as prevNode.cost +1
                4.6.3 Assign Heuristic as manhattan distance between goal and the row and column of
                        node
                4.6.4 Append to q
        4.7 If they are in q
                4.7.1 if prevNode.cost+1 <current cost
                        4.7.1.1 current cost=prevNode.cost+1

        4.8 If goal is reached
                4.8.1 Assign True to found
                4.8.2 Trace the parents with the help of parent_trace which returns path and break
        4.9 increment steps
step 5: return path and steps
```

```
parent_trace(path, finalNode):
step 1: Assign finalNode to rp
step 2 : while rp.parent is not null repeat steps 2.1 to 2.2
        2.1  append [rp.parent.row, rp.parent.col] to path
        2.2 Assign rp.parent to rp
step 3 : Return path
```

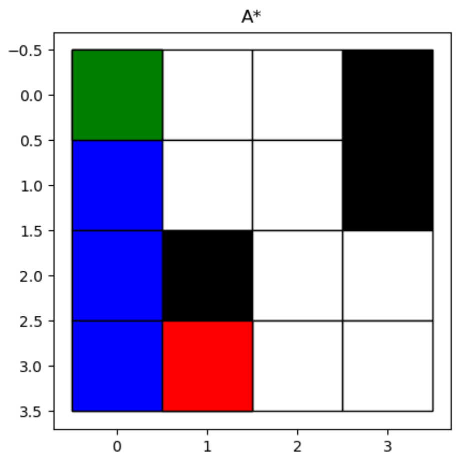
Code Explanation

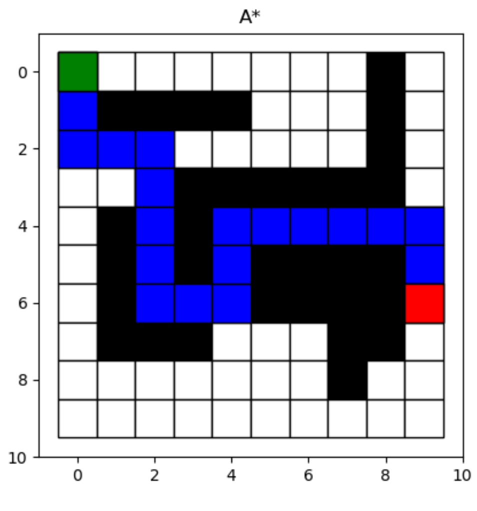
A* algorithm uses heuristics to allocated weight to each and every node. Nodes with higher values of heuristics are to be avoided.

An empty list q is created. The starting node is appended with its cost as 0 and heuristic as the distance between goal and the starting node. A loop is executed as long as q is not empty in which q is sorted based on the sum of cost and heuristic. The neighbours are explored and appended to the queue by creating objects of type node with parent as prevNode,cost as prevNode.cost+1 and h as manhattan distance between the point and the goal. Once the goal is reached, The path is backtracked and the shortest path is obtained.

Algorithm Results :

Test Map Result





Map Result

A* algorithm

A* algorithm uses a heuristic which is this case is manhattan distance The sum of the two is used to decide which will be explored next.

We shall use the same example

0	0	0	1
0	0	0	1
0	1	0	0
0	0	0	0

Numbering them ~~from~~ node from 1 to 16.

1	2	3	4
5	6	7	8
9	10	11	12
13	14	15	16

start : 1 goal : 14

explore order = ['R', 'D', 'L', 'U']

g

q =	1	2	5
parent	∅	1	1
h	4	3	3
cost	0	1	1

q =	2	5	3	6
p	1	1	2	2
h	3	3	4	2
cost	1	1	2	2
total	4	4	6	4

~~q =~~	~~5~~	~~3~~	~~6~~	~~9~~
~~p~~	~~1~~	~~2~~		
~~h~~	~~3~~	~~4~~		
~~cost~~	~~1~~	~~2~~		
~~total~~	~~4~~	~~6~~		

q =	5	6	3	9
p	1	2	2	5
h	3	2	4	2
cost	1	2	2	2
total	4	4	6	4

q =	5	6	9	3

q =	6	9	3	7
p	2	5	2	6
h	2	2	4	3
cost	2	2	2	3
total	4	4	6	6

q =	$\boxed{9}$	$\boxed{3}$	$\boxed{7}$	$\boxed{13}$
p	5	2	6	9
h	2	4	3	1
cost	2	2	3	3
total	4	6	6	4

q =	$\boxed{13}$	$\boxed{3}$	$\boxed{7}$	$\boxed{14}$
p	9	2	6	13
h	1	4	3	0
cost	3	2	3	4
total	4	6	6	4

q =	$\bigcirc\!\!\!\!\!\!\!\;14$	$\bigcirc\!\!\!\!\!\!\;3$	$\bigcirc\!\!\!\!\!\!\;7$
p	13	2	6

The goal is reached in
7 steps

Tracing parents

The path is

[[0,0],[1,0],[2,0],[3,0],[3,1]]

Similarities and differences between various algorithms and reason for similar and different results:

From implementing the four algorithms, It is found that they carry with them a lot of similarities and few differences.

Similarities of algorithms include
1. All of these algorithms are executed with a help of either a stack or a queue, very often the difference between one algorithm and another can be changing the data structure. Example would be , to go from bfs to dfs all is needed is to pop the elements from the end instead of the front.

Differences lie in the way the datastructure is treated for instance for A* the queue is sorted by the sum of cost and heuristic whereas in Dikjstra the queue is sorted by the cost. This may lead to A* taking longer periods of time as compared to dikjstra.

The reason the output of various algorithms are same and some are different are:
1. The output obtained may be the optimal output for the given graph which leads to various algorithms giving the same output.
2. The reason some are different are that some algorithms perform better in some situations and worse in others. The difference could be due to the situation not favouring the algorithm.

References:

1. MIT Open Courseware : https://www.youtube.com/watch?v=s-CYnVz-uh4

2. Youtube channels for DFS : https://www.youtube.com/watch?v=smHnz2RHJBY&t=1178s

https://www.youtube.com/watch?v=GazC3A4OQTE

3. For A*: https://www.youtube.com/watch?v=ySN5Wnu88nE&t=665s