RBE 550: Basic Search Algorithms Implementation Report

Joel John
jjohn2@wpi.edu

Worcester Polytechnic Institute — February 8, 2022

1 Code and Algorithm Explanation

In addition to the main BFS, DFS, Dijkstra and A* functions, there are other functions which are made as they are used frequently in all the algorithms. These are :-

- vcheck()
- findpath()
- findnext()

vcheck()

This function is called to check whether a node has been visited or not. It takes four inputs, which are the grid, the v list, the x position and the y position of the node in question.
The v list is of the same dimensions as the grid, with all its elements set to -1. It can be imagined as a matrix, where each node is mapped to a specific x and y combination in the matrix. The -1 indicates that the node at those coordinates is not visited. The grid list on the other hand is filled with 1s and 0s, the 1s indicate which coordinates are obstacles and the 0s indicate a safe coordinate.

The function first evaluates if the coordinates are within the boundaries of the grid. It continues if it is, but immideatly returns a 'False' if not.
If the coordinates are valid, it then checks whether the node in question is visited by checking the v list and if the node is an obstacle by checking the grid list. If the node is not visted and not an obstacle, the function returns a 'True' otherwise it defaults to a 'False'

findpath()

This function is called to find a path from the start position to the goal position using the input, which is the goal node. When a node is created, it is made to store info about its parent node. We can backtrack a path using this information.
The search algorithms creates the goal node and passes it to the function. The function extracts the coordinate information from the node and inserts it into a list at position 0, it then does the same to the parent node of the goal node and then the parent node of the parent node. This loop continues till there is no parent node which is the start node. The start node coordinates are inserted into the list and the list is returned. This is the path.

findnext()

This function is called to find valid neighbour nodes to the node in question. It takes three inputs, the grid, the node in question and the goal position. The v list is made global by the search algorithm functions so it does not need to be passed into the function as an input.

The function checks if the neighbours of the node in question are valid and not visited by calling vcheck(). The order of the check is right, down, left, up. If the check returns true, then the nodes for those coordinates are created and initialized. The parent node for these new nodes would be the node in question (The node passed to

the function). The cost to come would be the cost to come of the parent node +1, as the cost to move each step is set to 1. The heuristic would be the manhattan distance to the goal and the total cost would be the cost to come of the node + the heuristic of the same node. These nodes are then appended to a list which is returned.

All information that the algorithm needs are put into the node in this function. This information is available to all algorithms but certain information is only used in certain algorithms.

1.1 Breadth First Search - BFS

```
while(found==False):

    n=n+findnext(grid,n[0],goal)
    n.pop(0)
    steps=steps+1

    if(len(n)==0):
        break

    for i in n:
        v[i.row][i.col]=0

    if(n[0].row==goal[0] and n[0].col==goal[1]):
        found=True
        goal_node=n[0]
```

Figure 1: BFS Algorithm snippet

BFS is a search algorithm that is used to explore nodes in an unweighted graph, in this case the graph is a grid. From its start position, it visits all its neighbours before moving on to the deeper node.
A list 'v' is used to keep track of the visited nodes and a queue 'n' is used to tell the function which node to evaluate next. The starting node is created and initialized and appended to the list. The list 'v' is also updated for the start node.

As seen in Fig.1 The function calls findnext() to find the neighbours of the node using the first node in the queue (position 0). The list that is returned from findnext() is added to the queue. The first node in the list is removed, indicating that it has been evaluated. After this, a variable made to count the steps of the function can be incremented by 1. The function checks if the queue is empty. If it is empty it means that there is no path found and the function can stop.
findnext() would sometimes return nodes which are already in the queue. As the graph is unweighted, these nodes are duplicates which do not need to be evaluated. To overcome this, 'v' is updated to have the coordinates for all the nodes in the queue as visited. Now, findnext() is prevented from returning nodes that are already in the queue. Finally a check is performed to see if the first node in the list has the same coordinates as the goal position. If true, the function stops evaluating the queue as a path has been found. Otherwise the function loops back to find the neighbours of the first node in the queue (position 0) and this loop is continued till the goal is found. The steps variable should be incremented to account for the goal node.

findpath() is called using the goal node as the input to return the path list.
As the graph is unweighted, whenever a node is found for the first time we can be sure that there is no shorter path to that node. Thus BFS will always find the shortest path from start to goal.

1.2 Depth First Search - DFS

```
def rdfs(grid,pos,goal):
    global found
    global v
    global steps
    global goal_node

    if(found==False):

        if(pos.row==goal[0] and pos.col==goal[1]):
            found=True
            goal_node=pos

        if(v[pos.row][pos.col]==-1):
            v[pos.row][pos.col]=0
            steps=steps+1

        else:
            return

        n=findnext(grid,pos,goal)
        for i in n:
            rdfs(grid,i,goal)
```

Figure 2: The rdfs function used for DFS

DFS is also a search algorithm that is used to explore an unweighted graph, in this case a grid. From its start position, it picks a neighbour node and moves to that node, it keeps doing this until there are no valid neighbour nodes. It will then backtrack and evaluate another neighbour node that was not visited and the process repeats until the goal is found.

DFS can be implemented using a recursive function, a seperate function 'rdfs' is made for the implementation (shown in Fig.2). rdfs() takes three inputs, the grid, the start node and the goal coordinates. The main function initializes the variables needed and calls rdfs().
The rdfs function first checks if the coordinates of node passed to it is the goal coordinates, if it is the recursion stops. Next, a check is made if the node is visited or not, and updates the 'v' list as visited for the node coordinates if it wasnt visited. The steps variable is also incremented as the 'v' list is updated.
findnext() is called using the node passed to rdfs as input. Now, the recursion happens. rdfs() is called for each of the nodes in the list. The steps variable and 'v' need to be made global to reflect all the changes made to it.
When the goal is found, the goal node is stored in a global variable. A global flag variable 'found' is changed to True and the recursion stops. In the main function findpath() is only called if the flag variable is True. Here, the steps count both the start and the goal.

DFS does not find the shortest path. Depending on the structure of the graph and how deep the goal position is, it can give faster results than BFS.

1.3 Dijkstra Algorithm

```
while(found==False):

    m=findnext(grid,n[0],goal)

    for i in n:
        for a in m:

            if(i.row==a.row and i.col==a.col):
                if(i.g<=a.g):

                    m.remove(a)

                else:
                    temp=n.index(i)
                    n.remove(i)
                    n.insert(temp,a)
                    m.remove(a)

    n=n+m

    v[n[0].row][n[0].col]=0
    n.pop(0)
    steps=steps+1

    if(len(n)==0):
        break

    d=n[0].g
    j=0

    for i in range(1,len(n)):
        if(n[i].g<d):
            j=i
            d=n[i].g

    n.insert(0,n[j])
    n.pop(j+1)

    if(n[0].row==goal[0] and n[0].col==goal[1]):
        found=True
        goal_node=n[0]
```

Figure 3: Dijkstra Algorithm code snippet

Dijkstra Algorithm is used on a weighted graph to find the shortest path from the start to goal. It is similar to BFS but Dijkstra will visit the nodes with the smallest 'cost to come' first in the queue. Also the nodes that findnext() returns may not be duplicates. They can have the same node coordinates but may not have the same cost to come. In these cases the node with greater cost to come must be removed and the other node will replace it (shown in Fig.3).

The code is almost the same as the BFS algorithm. The differences are, in the case of dijkstra. The list returned from findnext() is not added to the queue immediately. The elements from that list is compared with the elements in the queue and the nodes are updated and removed such that there is only one node with the same coordinates and that node will have the lowest cost amongst the similar nodes. The list is then added to the queue.
All nodes in the queue are not made to be visited in the 'v' list as findnext() needs to evaluate them. Finally, before the goal check. The node with the smallest cost is found and moved to the start of the list to be evaluated next. The steps variable here does not count the goal position so it is set to start from 1 at the beginning.

Dijkstra Algorithm finds the shortest path as the shortest 'cost to come' is always evaluated first. So when the goal is evaluated we can be sure that the shortest path is found.

1.4 A* Algorithm

```
while(found==False):

    m=findnext(grid,n[0],goal)

    for i in n:
        for a in m:

            if(i.row==a.row and i.col==a.col):
                if(i.cost<=a.cost):
                    m.remove(a)

                else:
                    temp=n.index(i)
                    n.remove(i)
                    n.insert(temp,a)
                    m.remove(a)

    n=n+m

    v[n[0].row][n[0].col]=0
    n.pop(0)
    steps=steps+1

    if(len(n)==0):
        break

    d=n[0].cost
    j=0
    for i in range(1,len(n)):
        if(n[i].cost<d):
            j=i
            d=n[i].g

    n.insert(0,n[j])
    n.pop(j+1)

    if(n[0].row==goal[0] and n[0].col==goal[1]):
        found=True
        goal_node=n[0]
```

Figure 4: A* Algorithm code snippet

A* Algorithm is used on a weighted graph to find the shortest path from the start to goal. It uses a heuristic method to find the shortest path to the goal. The heuristic used here is the manhattan distance to the goal from the node. The total cost of the node would be the 'heuristic' + the 'cost to come'.

As seen in Fig.4, the code is almost the same as the Dijkstra Algorithm. The differences are for A*, instead of comparing the 'cost to come' of the nodes in the list returned by findnext() and the queue. It compares the total cost. Also instead of bringing the node with the smallest 'cost to come' as in Dijkstra, A* brings the node with the smallest total cost.

A* Algorithm is better than Dijkstra as it finds the shortest path using less steps. This is owning to the use of the heuristic as it gives the Algorithm an idea of how far it is from the goal. So it does not need to try to evaluate all the nodes to get to the goal. Its important to note that the A* algorithm will only find the shortest path if the heuristic is admissable. In a square grid, the manhattan distance is an admissable heuristic.

2 Test example, result and explanation

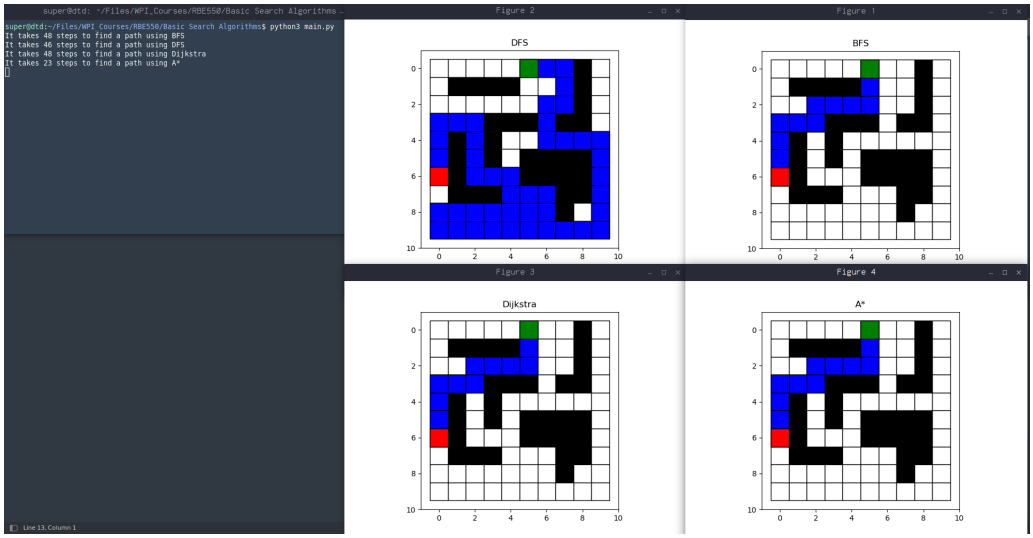

Figure 5: Results of the Algorithms on Map 1

In Fig 5. We see that BFS takes 48 steps, DFS takes 46 steps, Dijkstra takes 48 steps and A* takes 23 steps. We immediately notice that A* takes the least amount of steps followed by DFS and then Dijkstra and BFS. The path found though is the shortest path in all the algorithms but DFS.

BFS and Dijkstra Algorithms are similar, the difference being BFS works on an unweighted graph and Dijkstra on a weighted graph. Since the movement cost from one node to its neighbour is 1 for all nodes. BFS and Dijkstra should have the same amount of steps to reach the goal.

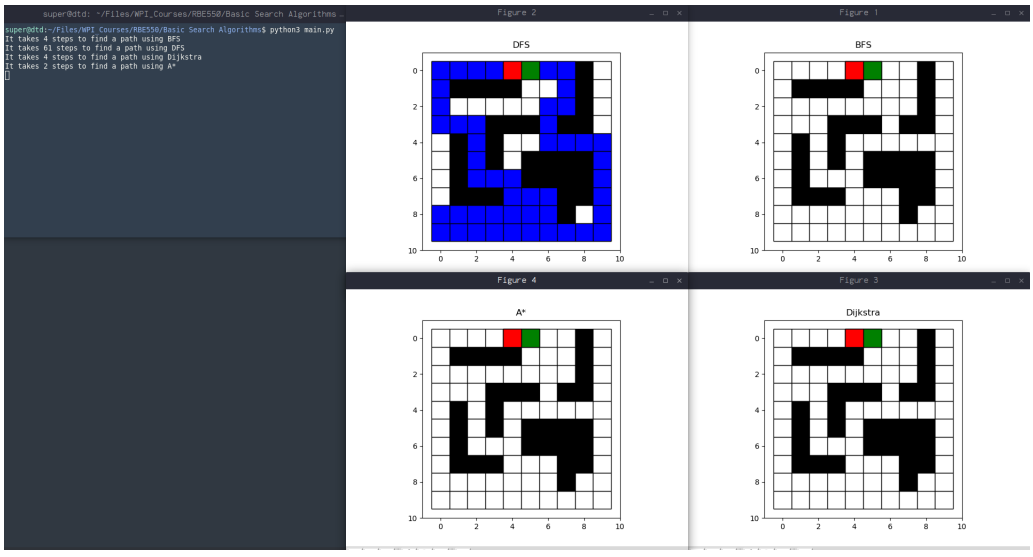

Figure 6: Results of the Algorithms on Map 2

DFS Algorithm has a lesser amount of steps than BFS and Dijkstra in Fig 5., this will not always be the case. For example, as seen in Fig 6. If the goal is nearby to the left of the start, BFS and Dijkstra finds the goal in 4 steps and A* finds it in 2 steps. DFS on the other hand takes 61 steps as this algorithm checks for neighbour nodes in the order right, down, left, up. The algorithm will move to the first valid node evaluated. If there are no valid nodes, it will bactrack one step and try another route.
Since the goal node is to the left of the start node the algorithm would circle around to get to the goal. DFS is only

concerned with finding a path, the path does not have to be the shortest. Therefore if the aim is to just find a path and if the goal node is very deep and not nearby the start node, we can use DFS. As, if it is nearby the start node, the other algorithms perform better.

A* Algorithm performs the best as it finds the shortest path with least amount of steps in any case. This is because the algorithm uses a heuristic which lets it know how far the node is from the goal. So it does not need to try to evaluate all the nodes like in the case of BFS and Dijkstra. The algorithm will evaluate only the nodes with the smallest total cost to the goal among the nodes explored, thereby moving in the direction of goal node. The total cost takes into account the 'cost to come' to the node from the start position and the distance from the node to the goal position. This algorithm is used in weighted graphs.