Sumeet Dinesh Shanbhag

Documentation

Dijkstra Algorithm:

Dijkstra's algorithm is a shortest path algorithm that finds the shortest path between two nodes in a graph. It works by maintaining a priority queue of nodes, starting with the source node, and repeatedly visiting the node with the lowest cost. The cost of visiting a node is the distance from the source node to the current node. The algorithm updates the cost of the neighbors of the current node and adds them to the priority queue, repeating this process until the destination node is reached or no more nodes are left in the priority queue. The algorithm uses a greedy approach and does not takes into account the heuristics, making it a complete and optimal algorithm for finding the shortest path in a graph with non-negative edge weights.

A* Algorithm:

A* algorithm is a shortest path algorithm that uses both the cost from the source node and an estimation of the remaining cost to the destination node, called the heuristic, to find the shortest path. It works by maintaining a priority queue of nodes, starting with the source node, and repeatedly visiting the node with the lowest cost, which is calculated as the sum of the cost from the source node and the heuristic. The algorithm updates the cost and the heuristic of the neighbors of the current node and adds them to the priority queue, repeating this process until the destination node is reached or no more nodes are left in the priority queue. The algorithm uses a combination of a greedy approach and heuristics to make the search more efficient, making it a complete and optimal algorithm for finding the shortest path in a graph with non-negative edge weights. The performance of the algorithm depends on the accuracy of the heuristics used.

Code Explanation:

1. Dijkstra Algorithm:

```python
class Node:
    def __init__(self, row, col, is_obs, h):
        self.row = row        # coordinate
        self.col = col        # coordinate
        self.is_obs = is_obs  # obstacle?
        self.g = None         # cost to come (previous g + moving cost)
        self.h = h            # heuristic
        self.cost = None      # total cost (depend on the algorithm)
        self.parent = None    # previous node
```

- Create node objects for each position in the grid.

```python
start_node = nodes[start[0]*cols + start[1]]
goal_node = nodes[goal[0]*cols + goal[1]]
start_node.g = 0
```

- Set the g cost of the start node to 0.

```python
heap = [start_node]
while len(heap) > 0:
    curr_node = heap.pop(0)
    if curr_node.row == goal_node.row and curr_node.col == goal_node.col:
        found = True
        goal_node.parent = curr_node.parent
        break
```

- Add the start node to a heap (a min heap, where the node with the smallest g cost is always at the front).
- While the heap is not empty, pop the node with the smallest g cost from the heap.
- Check if the popped node is the goal node. If so, break out of the loop.

```
for i, j in [(0, 1), (0, -1), (1, 0), (-1, 0), (1, 1), (1, -1), (-1, 1), (-1, -1)]:
    new_row = curr_node.row + i
    new_col = curr_node.col + j
    if new_row >= 0 and new_row < rows and new_col >= 0 and new_col < cols:
        new_node = nodes[new_row*cols + new_col]
        if new_node.is_obs == 0:
            new_g = curr_node.g + 1
            if i != 0 and j != 0: # for diagonal moves
                new_g += 0.5
            if new_node.g is None or new_g < new_node.g:
                new_node.g = new_g
                new_node.parent = curr_node
                heap.append(new_node)
steps += 1
```

- For each possible move (up, down, left, right, and diagonals), calculate the new g cost of the node.
- If the new g cost is lower than the current g cost of the node, or if the node has not been visited yet, update the g cost of the node and set its parent to the current node.
- Add the node to the heap.
- Repeat steps 4-8 until the goal node is found or the heap is empty.

```
if found:
    node = goal_node
    while node is not None:
        path.append([node.row, node.col])
        node = node.parent
    path.reverse()

    print(f"It takes {steps-1} steps to find a path using Dijkstra")
else:
    print("No path found")
return path, steps-1
```

- If the goal node is found, construct the path by following the parent pointers from the goal node back to the start node.

2. A* Algorithm:

```python
class Node:
    def __init__(self, row, col, is_obs, h):
        self.row = row        # coordinate
        self.col = col        # coordinate
        self.is_obs = is_obs  # obstacle?
        self.g = None         # cost to come (previous g + moving cost)
        self.h = h            # heuristic
        self.cost = None      # total cost (depend on the algorithm)
        self.parent = None    # previous node
```

- Create node objects for each position in the grid.

```python
start_node = nodes[start[0]*cols + start[1]]
goal_node = nodes[goal[0]*cols + goal[1]]
start_node.g = 0
start_node.cost = start_node.g + start_node.h
```

- Set the g cost of the start node to 0 and calculate the h cost for each node.

```python
heap = [start_node]
while len(heap) > 0:
    curr_node = heap.pop(0)
    #steps += 1
    if curr_node.row == goal_node.row and curr_node.col == goal_node.col:
        found = True
        goal_node.parent = curr_node.parent
        break
```

- Add the start node to a heap (a min heap, where the node with the lowest f cost is always at the front)
- While the heap is not empty, pop the node with the lowest f cost from the heap.
- Check if the popped node is the goal node. If so, break out of the loop.

```
for i, j in [(0, 1), (0, -1), (1, 0), (-1, 0), (1, 1), (1, -1), (-1, 1), (-1, -1)]:
    new_row = curr_node.row + i
    new_col = curr_node.col + j
    if new_row >= 0 and new_row < rows and new_col >= 0 and new_col < cols:
        new_node = nodes[new_row*cols + new_col]
        if new_node.is_obs == 0:
            new_g = curr_node.g + 1
            if i != 0 and j != 0: # for diagonal moves
                new_g += 0.5
            if new_node.g is None or new_g < new_node.g:
                new_node.g = new_g
                new_node.cost = new_g + new_node.h
                new_node.parent = curr_node
                heap.append(new_node)
heap = sorted(heap, key=lambda x: x.cost) # sort heap based on cost (f = g + h)
steps += 1
```

- For each possible move (up, down, left, right, and diagonals), calculate the new g cost and the f cost (g + h) of the node.
- If the new f cost is lower than the current f cost of the node, or if the node has not been visited yet, update the g and f cost of the node and set its parent to the current node.
- Add the node to the heap.
- Repeat steps 4-8 until the goal node is found or the heap is empty.

```
if found:
    node = goal_node
    while node is not None:
        path.append([node.row, node.col])
        node = node.parent
    path.reverse()

    print(f"It takes {steps} steps to find a path using A*")
else:
    print("No path found")
print(path)
return path, steps
```

- If the goal node is found, construct the path by following the parent pointers from the goal node back to the start node.

Pseudo Code:

I. Dijkstra Algorithm:

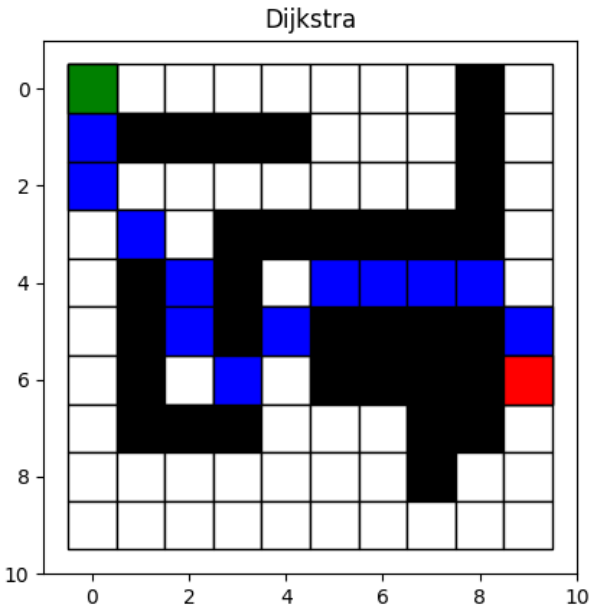
```
function Dijkstra(Graph, source):

     for each vertex v in Graph.Vertices:
         dist[v] ← INFINITY
        prev[v] ← UNDEFINED
        add v to Q
     dist[source] ← 0

     while Q is not empty:
        u ← vertex in Q with min dist[u]
        remove u from Q

        for each neighbor v of u still in Q:
            alt ← dist[u] + Graph.Edges(u, v)
            if alt < dist[v]:
                dist[v] ← alt
                prev[v] ← u

    return dist[], prev[]
```

II. A* Algorithm

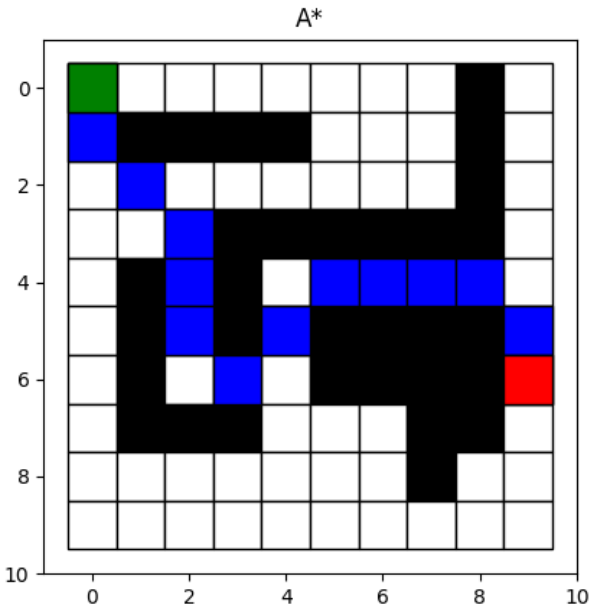
```
// A* (star) Pathfinding
// Initialize both open and closed list
let the openList equal empty list of nodes
let the closedList equal empty list of nodes
// Add the start node
put the startNode on the openList (leave it's f at
zero)
```

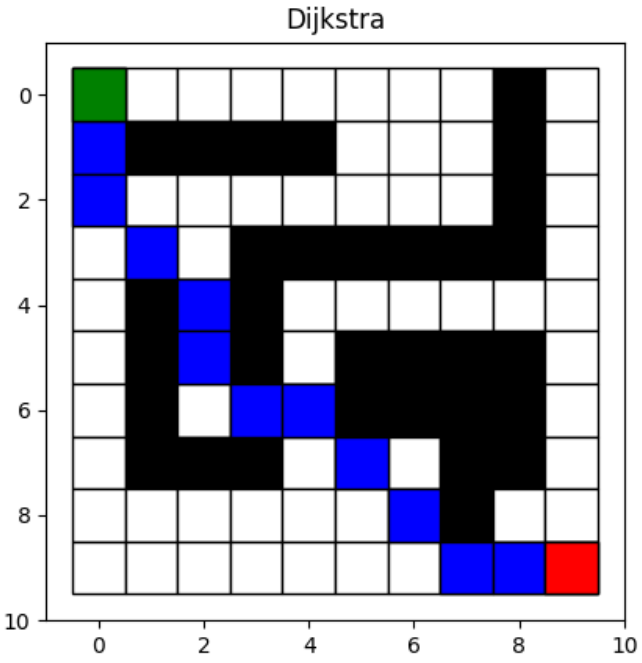
```
// Loop until you find the end
while the openList is not empty
    // Get the current node
    let the currentNode equal the node with the
least f value
    remove the currentNode from the openList
    add the currentNode to the closedList
    // Found the goal
    if currentNode is the goal
        Congratz! You've found the end! Backtrack
to get path
    // Generate children
    let the children of the currentNode equal the
adjacent nodes

    for each child in the children
        // Child is on the closedList
        if child is in the closedList
            continue to beginning of for loop
        // Create the f, g, and h values
        child.g = currentNode.g + distance between
child and current
        child.h = distance from child to end
        child.f = child.g + child.h
        // Child is already in openList
        if child.position is in the openList's
nodes positions
            if the child.g is higher than the
openList node's g
                continue to beginning of for loop
        // Add the child to the openList
        add the child to the openList
```

Test cases:
Start : (0,0)

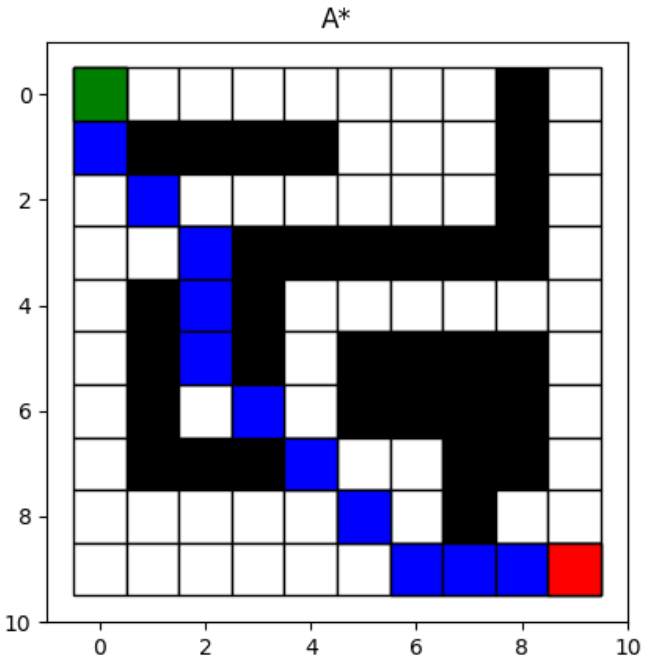


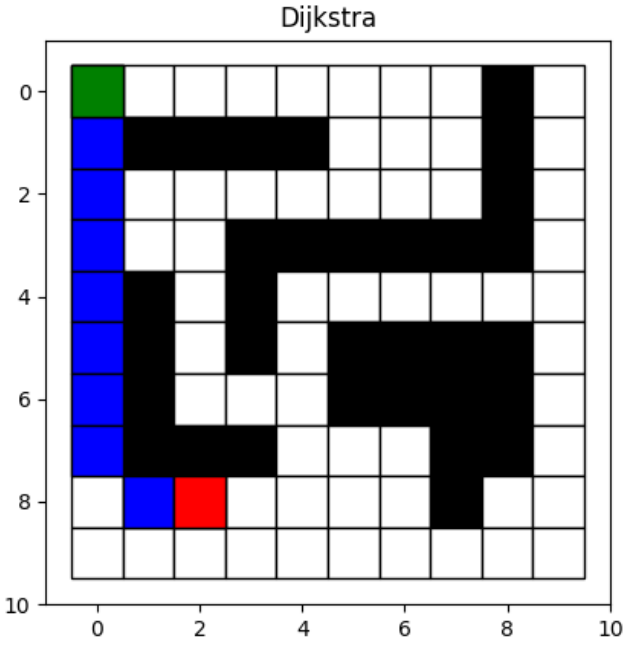
Goal : (6,9)



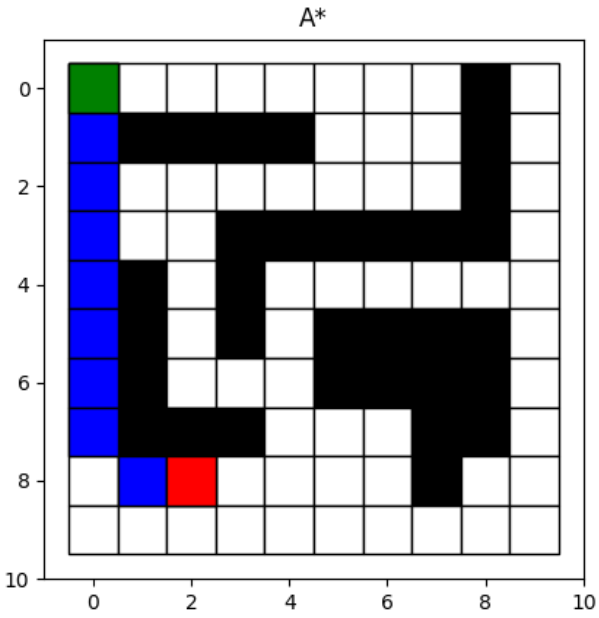


Goal : (9,9)





Goal : (8,2)



References:

- https://www.freecodecamp.org/news/dijkstras-algorithm-explained-with-a-pseudocode-example/
- https://medium.com/@nicholas.w.swift/easy-a-star-pathfinding-7e6689c7f7b2