Four Common Algorithms for Path Finding of a Know Maze

Jiankun Dong, Lingjie Yuan

BU MET College

Department of Computer Science

CS566A2, Prof. Alexander Belyaev

Fall, 2023

Contents

Abstract

There are many strategies for finding a path in a maze. This report explores the following common algorithms used for pathfinding: Breath First Search, Depth First Search, Dijkstra's Algorithm, and A* Algorithm. For the 4 algorithms above, the report discusses the implementation of each algorithm, the performance of each, and their strengths and weaknesses. BFS and DFS are used as the baseline for a more detailed discussion of Dijkstra's and A* algorithms.

Introduction

The maze used in this project is a grid of values "1" and "0", in which "1" represents a wall and "0" is a path. Each value in the grid is called a "Node" in the maze. The maze must have a starting node and an end node as the goal. When navigating the maze, each algorithm is allowed to examine one node at a time, essentially simulating a player exploring the maze. After finding the goal, a path is then formed as a list of nodes from start to finish that when followed, guides the player from the beginning to the goal.

As this project is done by 2 people remotely on personal PCs, it's hard to gauge the efficiency based on runtime. Even when using the same PC with the same setup, a manually generated maze is often not complex enough for the more efficient algorithms to generate a readable execution time (i.e., problem solved too fast). The efficiency of the algorithm is evaluated based on how many nodes it checks, in addition to the program running time. Each algorithm is run on the same mazes. The mazes are generated by hand. Moreover, when examining a path, the number of nodes (length of the path) is most significant, and the number of "turns" made by the

explorer is ignored. In real-world scenarios, turning can make a significant impact on determining how good a path is.

In this report, we assume the maze is “dense” and the explorer is “node-sized”. By “dense”, it is assumed that there is at least 1 wall neighboring each node, and there are not 2 parallel paths that are not separated by a wall. Thus, for movement in the grid, the diagonal movement is restricted. Furthermore, by having a “node-sized” explorer, the optimization of detailed movement within and between grids is ignored over the emphasis on the actual path.

Algorithms and Implementation

DFS

The strategy followed by depth-first search (DFS) is to search deeper in the graph whenever possible. DFS explores edges out of the most recently discovered vertex that still has unexplored edges leaving it. Once all of this vertex’s edges have been explored, the search backtracks to explore edges leaving the vertex from which this vertex was discovered. This process continues until we have discovered all the vertices that are reachable from the source vertex. The implementation of DFS is based on stack, which is last in first out (LIFO).

For the worst case, the algorithm needs to explore the whole maze before finding a correct path, resulting in the dreaded O(N) worst-case runtime. Another drawback for DFS is that it does not find the shortest path as the first path it returns. It needs to exhaust the map to find the best path for the maze.

BFS

Breadth-first search (BFS) is one of the simplest algorithms for searching a graph and the archetype for many important graph algorithms, like Dijkstra's algorithm we mention below. BFS expands the frontier between discovered and undiscovered vertices uniformly across the breadth of the frontier. That is, the algorithm discovers all vertices at distance k from source, before discovering the vertices at distance k+1. Since nodes are processed in the same order as they were added to the queue, i.e., first in first out (FIFO), the implementation of BFS is based on the queue.

For the worst case, the procedure time of BFS is O(4*V), which explores all the vertices in the maze in 4 directions. BFS finds the shortest path directly at the cost of memory and time, which makes this algorithm more suitable for sparse and small-sized mazes.

Dijkstra's Algorithm

Dijkstra's algorithm solves the single-source shortest-paths problem on a weighted, directed graph for the case in which all edge weights are nonnegative. This algorithm maintains a set S of vertices whose final shortest-path weights from the source vertex have already been determined. Dijkstra's algorithm repeatedly selects the vertex not in the S set with the minimum shortest-path estimate, adds this vertex to the S set, and relaxes all edges leaving this vertex.

DIJKSTRA $(\mathbf{G}, \mathbf{w}, \mathbf{s})$

$INITIALIZE\_SINGLE\_SOURCE(G, s)$
$S = \emptyset$
$Q = G.V$ $//all\ the\ vertices\ in\ G\ with\ their\ distance\ form\ source\ vertex(s)$
$while\ Q \neq \emptyset$
$u = EXTRACT.MIN(Q)$ $//extract\ the\ closest\ vertex\ in\ (V - S)$

$S = S \cup \{u\}$ //*add this closest vertex to set S*
for each vertex $v \in G.Adj[u]$
$RELAX(u, v, w)$ //*relax each edge leaving u to update vertices' distance in Q*

Dijkstra's algorithm fits the greedy strategy since it always chooses the closest vertex to add to set S. This guarantees that we can find the optimal path, and save a lot of time compared with BFS and DFS. However, the disadvantage of Dijkstra's algorithm is that it can only see the cost accumulated so far, and it has no information about the goal location, thus it will explore the next position in every direction. In the Dijkstra's code part, we need to transverse the four directions (up, right, down, and left) to determine the next point.

A*

A* algorithm takes advantage of a heuristic value for each node to help determine the optimal path. It combines the shortest path searching from Dijkstra and the searching based on the heuristic value. For each node, a value $f(node)$ is calculated:

$$f(node) = h(node) + g(node).$$

There are multiple options for calculating a heuristic value for a node, as it's a representation of how far the node is to the goal. Because diagonal movements are restricted, the most common and basic Manhattan Distance is used, in which:

$$h(node) = abs(Xgoal - Xnode) + abs(Ygoal - Ynode).$$

The combination of restricted movement and the usage of Manhattan distance for the heuristic value also prevents forming of cross-path from being generated in a complex maze (J. Liu, J. Yang, H. Liu, X. Tian, and M. Gao Oct. 2017). Using only 4 directional movement also resolves the sawtooth path issue (Gang Tang, CongQiang Tang, Christophe Claramunt, Xiong

Hu, Peipei Zhou 2021).This also dramatically simplifies the problem for other 3 algorithms in comparison.

The searching process of the A* star stores the checked nodes in “Closed list” and the nodes to be checked in “Open list”. Each time a node with the least $f(node)$ is popped from the open list, the algorithm does the following:

1. Check if the node is the goal. If so, generate the path; if not, add the node to closed list.
2. For each of the neighboring nodes that is not a wall/obstacle.
3. If the neighboring node is NOT in the closed list, calculate the node’s $h(node)$ and $g(node)$.
4. If the neighboring node is already in the Open list, update and sort the open list if the new $g(node)$ value is smaller.
5. Add the neighboring node into the Open list and sort the list.

A class Node is created to store the node’s position, the parent node (for generating the path), $h(node)$, $g(node)$ and $f(node)$. Comparison between nodes is based on the $f(node)$ value.

A min heap is used for the “Open list” to take advantage of the fact that the root will be the node with the smallest $f(node)$.

Path is generated by following the last node’s parent node and storing them in a list, finally reverting the order of the list.

Testing and Evaluation

Three maze grids, shown below, are used across the 4 algorithms. For each algorithm, the number of nodes explored is counted. The number of nodes explored together with execution time determines how efficient the algorithm is.

```
#MAZE 1 #

0 0 0 0 0
0 1 1 0 0
0 0 0 1 0
0 0 1 1 1
0 0 0 0 0
```

```
#MAZE 2 #

0 1 1 0 1 0 1
0 0 0 0 0 0 0
0 1 1 0 1 1 0
0 0 1 0 0 1 0
1 0 1 0 1 1 0
1 0 0 0 1 0 0
```

```
#MAZE 3 #

0 0 0 0 0
0 1 0 1 0
0 0 0 1 0
0 1 1 1 0
1 1 0 1 0
```

DFS

The results of the DFS algorithm to find the path of the maze are shown below.

```
#TEST 1 #

* 0 0 0 0
* 1 1 0 0
* 0 0 1 0
* 0 1 1 1
* * * * *

Explored node count:  26
runtime:  0.0
```

```
#TEST 2 #

* 1 1 0 1 0 1
* 0 0 * * * *
* 1 1 * 1 1 *
* * 1 * 0 1 *
1 * 1 * 1 1 *
1 * * * 1 0 *

Explored node count:  48
runtime:  0.0
```

```
#TEST 3 #

* 0 * * *
* 1 * 1 *
* * * 1 *
0 1 1 1 *
1 1 0 1 *

Explored node count:  29
runtime:  0.0
```

To find a path, the procedure running time ranges from $4 \times dis(start,\ end)$ to $4 \times Vertex\ Number$, and the time complexity is $O(V)$, while the space complexity is $O(n)$, n represents the deepest stack layer, i.e., the number of vertices in the longest path.

In the test cases, we can find that the explored node is relatively less, and the running time is short. However, the paths in the second and third mazes are redundant. Only in the first maze, does DFS find the shortest path. That's because we execute the algorithm until there is no path

to go or we find the destination and never go back to check if this path is optimal. Thus, it's easy and fast to find a path using DFS, but we can't guarantee the path we find is the shortest one.

BFS

The results of the BFS algorithm to find the path of mazes are shown below.

```
#TEST 1 #

* 0 0 0 0
* 1 1 0 0
* * 0 1 0
0 * 1 1 1
0 * * * *

Explore node count: 41
runtime:  0.0
```

```
#TEST 2 #

* 1 1 0 1 0 1
* * * * * * *
0 1 1 0 1 1 *
0 0 1 0 0 1 *
1 0 1 0 1 1 *
1 0 0 0 1 0 *

Explore node count: 49
runtime:  0.0
```

```
#TEST 3 #

* * * * *
0 1 0 1 *
0 0 0 1 *
0 1 1 1 *
1 1 0 1 *

Explore node count: 29
runtime:  0.0
```

The BFS algorithm will explore all the vertices in each layer until finds the destination, or transverse all the vertices in the graph. Its running time is nearly $4 \times Vertex\ Number$, and the time complexity is $O(V)$. BFS stores all the vertices for each layer in a queue, thus the space complexity is $O(n)$, n represents the longest queue number, i.e., the largest number of vertices in a layer.

In the test cases, we can find that all the explored nodes are larger than the vertices number. That's because, in addition to traversing all the points in the maze, the algorithm also needs to make some comparisons between the non-graph direction of the edge points. This complete algorithm makes the path found must be the shortest one.

Dijkstra's algorithm

The results of the Dijkstra's algorithm to find the path of mazes are shown below.

```
#TEST 1 #

* 0 0 0 0
* 1 1 0 0
* * 0 1 0
0 * 1 1 1
0 * * * *

Explored node count:  59
runtime:  0.01562356948852539
```

```
#TEST 2 #

* 1 1 0 1 0 1
* * * * * * *
0 1 1 0 1 1 *
0 0 1 0 0 1 *
1 0 1 0 1 1 *
1 0 0 0 1 0 *

Explored node count:  68
runtime:  0.0
```

```
#TEST 3 #

* * * * *
0 1 0 1 *
0 0 0 1 *
0 1 1 1 *
1 1 0 1 *

Explored node count:  35
runtime:  0.0
```

Because Dijkstra's algorithm will update all the undetermined vertex every time, which is V times. The procedure exploring new vertex time will be less than $4 \times Vertex\ Number$. Since we use the queue to implement this algorithm, its time complexity is $O(V^2)$. The space complexity of this algorithm is $O(V)$, which means, at last, the S set should contain all the vertices in the graph.

Dijkstra's algorithm can find the shortest path in the maze, and its time complexity is less than basic BFS. It works in this "find the path in a maze" problem, but it's able to do better in the graphs with weighted edges. We can do more in-depth research in this area in the future.

A*

For A*, the number of time either heapify is called or a node is added to the heap during the main while loop is also counted. This also helps with judging how efficient the A* algorithm is. Because A* explores the node with the least $f(node)$ value in the open list, the min-heap data structure is used to improve the time for finding and retrieving the node with the least f value.

The time complexity of the A* algorithm is largely dependent on how well the heuristic function estimates the cost to the goal. In our example, because the move is limited in 4 directional, the Manhattan distance is sufficient. The distant is inaccurate (underestimate)

when the node leads to a dead-end, and the distant is never overestimated. Therefore, our A* performance varies based on the maze given.

Therefore, the worst case of this A* implementation is for a maze that contains many dead-ends, and the optimal path is one that first go further away from the goal than the starting point. For the worst case, A* has the time complexity of $O(VlogV)$ and space complexity of $O(V)$. For the best case, A* would have $O(n)$ for both time and space complexity (n is the number of nodes for the best path), since it would simply follow the best path guided by the accurate heuristic function.

```
#TEST 1 #

* 0 0 0 0
* 1 1 0 0
* * 0 1 0
0 * 1 1 1
0 * * * *

Explored node count:  21
The number of time the a node is push to a heap or heapify is called:  19
runtime:  0.0
#TEST 2 #

* 1 1 0 1 0 1
* * * * * * *
0 1 1 0 1 1 *
0 0 1 0 0 1 *
1 0 1 0 1 1 *
1 0 0 0 1 0 *

Explored node count:  25
The number of time the a node is push to a heap or heapify is called:  25
runtime:  0.0
```

```
#TEST 3 #

* * * * *
0 1 0 1 *
0 0 0 1 *
0 1 1 1 *
1 1 0 1 *

Explored node count:  15
The number of time the a node is push to a heap or heapify is called:  15
runtime:  0.0
```

Conclusion

The time complexity of DFS is $O(V)$ and the space complexity is $O(n)$, it's equal to BFS and less than Dijkstra. But the path DFS finds in a maze is not always the shortest one.

BFS has the time complexity of $O(V)$ and the space complexity of $O(n)$, and it's relatively fast. That's because we marked the visited nodes to avoid repeating.

Since Dijkstra needs to update all the vertices every time, and we use a simple queue to implement this algorithm. This algorithm is the most complex.

The most important part of the A* algorithm is to find an accurate heuristic function. Then A* yields the overall best performance.

	Time Complexity	**Space Complexity**
DFS	$O(V)$	$O(n)$
BFS	$O(V)$	$O(n)$
Dijkstra	$O(V^2)$	$O(V)$
A*	Best heuristic function: $O(n)$ Worst heuristic function: $O(VlogV)$	Best heuristic function: $O(n)$ Worst heuristic function:$O(V)$

V represents the number of 0 in the maze.
For DFS: n represents the number of vertices in the optimal path
For BFS: n represents the largest number of vertices in a layer
For A*: n is the number of nodes in the optimal path

Source Code

DFS

```
1.  class Node():
2.      def __init__(self, parent=None, position=None):
3.          self.position = position
4.          self.parent = parent
5.
6.      def __eq__(self, other):
7.          return self.position == other.position
8.
9.  def reconstruct_path(currNode):
10.     path = []
11.     while currNode is not None:
12.         path.append(currNode.position)
13.         currNode = currNode.parent
14.     return path[::-1]
15.
16. def dfs(grid, start, goal):
17.     nodecount = 0
18.     nodeStack = []
19.     checkedStack = []
20.     startNode = Node(None, start)
21.     goalNode = Node(None, goal)
22.     nodeStack.append(startNode)
23.     while (nodeStack):
24.         currNode = nodeStack.pop()
25.         checkedStack.append(currNode)
26.         if currNode == goalNode:
27.             path = reconstruct_path(currNode)
28.             return (path, nodecount)
29.         for dx, dy in MOVEMENTS:  # for each possible steps
30.             neighbor = (currNode.position[0] + dx, currNode.position[1] + dy)
31.             if 0 <= neighbor[0] < len(grid) and 0 <= neighbor[1] < len(grid[0]) and grid[neigh
    bor[0]][neighbor[1]] != 1:
32.                 nodecount += 1
33.                 # when neighbor is a legal move put it in queue
34.                 neighborNode = Node(currNode, neighbor)
35.                 if neighborNode not in checkedStack:
36.                     nodeStack.append(neighborNode)
37.     return (None, nodecount)  # no path found
```

BFS

```
1.  class Node():
2.      def __init__(self, parent=None, position=None):
3.          self.position = position
4.          self.parent = parent
5.
6.      def __eq__(self, other):
7.          return self.position == other.position
8.
9.  def reconstruct_path(currNode):
10.     path = []
11.     while currNode is not None:
12.         path.append(currNode.position)
13.         currNode = currNode.parent
14.     return path[::-1]
15.
16. def bfs(grid, start, goal):
17.     nodecount = 0
18.     visited = []
19.     nodeQueue = queue.Queue()
20.     startNode = Node(None, start)
21.     goalNode = Node(None, goal)
22.     nodeQueue.put(startNode)
23.     visited.append(startNode)
24.     while (not nodeQueue.empty()):
25.         currNode = nodeQueue.get()
26.         if currNode == goalNode:
27.             path = reconstruct_path(currNode)
28.             return (path, nodecount)
29.         for dx, dy in MOVEMENTS:  # for each possible steps
30.             neighbor = (currNode.position[0] + dx, currNode.position[1] + dy)
31.             if 0 <= neighbor[0] < len(grid) and 0 <= neighbor[1] < len(grid[0]) and grid[neigh
    bor[0]][neighbor[1]] != 1:
32.                 nodecount += 1
33.                 # when neighbor is a legal move put it in queue
34.                 neighborNode = Node(currNode, neighbor)
35.                 if neighborNode not in visited:
36.                     visited.append(neighborNode)
37.                     nodeQueue.put(neighborNode)
38.     return (None, nodecount)  # no path found
```

Dijkstra

```
def create(maze, start):
    Q = []
    for i in range(0,len(maze)):
        for j in range(0,len(maze[0])):
            if maze[i][j] != 1:
                Q.append(i*len(maze[0])+j)
    dis = [float('inf')] * len(Q)
    dis[Q.index(start)] = 0
    return Q, dis

def Path(S, pre, start, end, maze):
    path = []
    point = end
    while point != start:
        path.insert(0, [point//len(maze[0]),point%len(maze[0])])
        point = pre[S.index(point)-1]
    path.append([start // len(maze[0]), start % len(maze[0])])
    path.append([end//len(maze[0]),end%len(maze[0])])
    return path

def Dijkstra(maze, start, end):
    nodecount = 0
    S = []
    dis_determ = []
    start = start[0] * len(maze[0]) + start[1]
    end = end[0] * len(maze[0]) + end[1]
    Q, dis = create(maze, start)
    temp1 = []
    temp2 = []
    pre = []
    while(Q != []):
        point = Q[dis.index(min(dis))]
        S.append(point)
        dis_determ.append(dis[Q.index(point)])
        if temp1 != []:
            pre.append(temp1[temp2.index(point)])
        if point == end:
            path = Path(S, pre, start, end, maze)
            return path, nodecount
        dis.pop(Q.index(point))
        Q.pop(Q.index(point))
        temp1 = []
        temp2 = []
        for point in S:
```

```
45.             for i in range (0,4):
46.                 nextpoint = [int(point/len(maze[0]))+ MOVEMENTS[i][0], point%len(maze[0]
    ) + MOVEMENTS[i][1]]
47.                 if nextpoint[0] < 0 or nextpoint[1] < 0 or nextpoint[0] > len(maze) -
     1 or nextpoint[1] > len(maze[0]) - 1:
48.                     continue
49.                 elif nextpoint[0]*len(maze[0])+nextpoint[1] in Q:
50.                     nodecount += 1
51.                     dis[Q.index(nextpoint[0]*len(maze[0])+nextpoint[1])] = dis_determ[S.index
    (point)]+1
52.                     temp1.append(point)
53.                     temp2.append(nextpoint[0]*len(maze[0])+nextpoint[1])
54.     return False
```

A*

```
class Node():
    def __init__(self, parent = None, position = None):
        self.parent = parent
        self.position = position
        self.g=0
        self.h=0
        self.f=0
    def __eq__(self, other):
        return self.position == other.position
    def __hash__(self):
        return hash(self.position)
    def __lt__(self,other):
        return self.f<other.f

def heuristic(node, goal):
    # This is a simple heuristic (Manhattan distance), because we not allowing diagnoal movement at the moment
    return abs(node[0] - goal[0]) + abs(node[1] - goal[1])

def astar(grid, start, goal):
    nodecount =pushcount = 0
    # grid is the map
    # start is the starting postion
    # goal is the target postion
    start_node = Node(None, start)
    start_node.h = heuristic(start, goal)
    start_node.f = start_node.g+start_node.h
    end_node = Node(None,goal)
    open_list = [] # this is the expending "tip" of the search, use heap because we checking lowest f val neighbor
    close_set = set() # this is searched, no need to keep it as heap, but rather set
    heapq.heappush(open_list, start_node)#organized based on f score

    while open_list: #keep searching until path is found
        current = heapq.heappop(open_list) #consider the node with lowest f score
        if current == end_node:
            path = reconstruct_path(current)
            return (path,nodecount,pushcount)
        #put currnode into the closed list andd look at its neighbors
        close_set.add(current)

        for dx, dy in MOVEMENTS: # for each possible steps
            neighbor = (current.position[0] + dx, current.position[1] + dy)
```

```
            if 0 <= neighbor[0] < len(grid) and 0 <= neighbor[1] < len(grid[0]) and grid[neighbor[0]][neighbor[1]]!=1:
                #don't run into walls and keep within bounds
                neighborNode = Node(current,neighbor)
                if neighborNode in close_set:
                    #ignore node in close set already
                    continue
                nodecount +=1
                neighborNode.h = heuristic(neighbor,goal)
                neighborNode.g = current.g + 1 # Assuming uniform cost for all movements
                neighborNode.f = neighborNode.h+neighborNode.g
                add_flag = True
                for i in range(0,len(open_list)):
                    openNode = open_list[i]
                    if openNode == neighborNode :
                        add_flag = False
                        #when neighborNode is already in open list
                        # if new g score is lower, the new path is better update parent and scores
                        if neighborNode.g<=openNode.g:
                            open_list[i] = neighborNode
                            heapq.heapify(open_list)
                            pushcount+=1
                            break
                if add_flag: #neighborNode not in open list
                    pushcount+=1
                    heapq.heappush(open_list, neighborNode)
    return (None,nodecount,pushcount)  # No path found

def reconstruct_path(currNode):
    path = []
    while currNode is not None:
        path.append(currNode.position)
        currNode = currNode.parent
    return path[::-1]
```

Main Function

```
# Define possible movements (up, down, left, right) disableing : and diagonals)
MOVEMENTS = [(0, 1), (0, -1), (1, 0), (-1, 0)]  # (1, 1), (1, -1), (-1, 1), (-1, -1)]

if __name__ == "__main__":
    grid1 = [[0, 0, 0, 0, 0],
             [0, 1, 1, 0, 0],
             [0, 0, 0, 1, 0],
             [0, 0, 1, 1, 1],
             [0, 0, 0, 0, 0]]
    grid2 = [[0, 1, 1, 0, 1, 0, 1],
             [0, 0, 0, 0, 0, 0, 0],
             [0, 1, 1, 0, 1, 1, 0],
             [0, 0, 1, 0, 0, 1, 0],
             [1, 0, 1, 0, 1, 1, 0],
             [1, 0, 0, 0, 1, 0, 0]]
    grid3 = [[0, 0, 0, 0, 0],
             [0, 1, 0, 1, 0],
             [0, 0, 0, 1, 0],
             [0, 1, 1, 1, 0],
             [1, 1, 0, 1, 0]]
    gridLs = [grid1, grid2, grid3]
    start = (0, 0)
    count=1
    for grid in gridLs:
        goal = (len(grid) - 1, len(grid[0]) - 1)
        starttime = time.time()
        path, nodeexplored = DFS(grid, start, goal)
        #path, nodeexplored = BFS(grid, start, goal)
        #path, nodeexplored = Dijkstra(grid, start, goal)
        #path, nodeexplored = astar(grid, start, goal)
        endtime = time.time()
        print("#TEST", count, "#")
        count += 1
        print()
        if path:
            for i in range(0, len(path)):
                grid[path[i][0]][path[i][1]] = "*"
            for row in grid:
                for element in row:
                    print(element, end=" ")
                print()
            print()
        else:
            print("No path found.")
```

```
45.         print("Explored node count: ", nodeexplored)
46.         print("runtime: ", endtime - starttime)
47.         print()
```

References

Gang Tang, CongQiang Tang, Christophe Claramunt, Xiong Hu, Peipei Zhou. VOLUME 9, 2021. "Geometric A-Star Algorithm: An Improved." *IEEE Access* 59196-59210.

J. Liu, J. Yang, H. Liu, X. Tian, and M. Gao. Oct. 2017. "An improved ant colony algorithm for robot path planning." *Soft Comput.* vol.21, no.19, pp. 5829-5839.

Thomas Cormen and Ronald Rives. " Introduction to Algorithms – Third Edition"