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 3](#_Toc151296561)

[Introduction 3](#_Toc151296562)

[Algorithms and Implementation 4](#_Toc151296563)

[DFS 4](#_Toc151296564)

[BFS 4](#_Toc151296565)

[Djikstra’s Algorithm 4](#_Toc151296566)

[A\* 4](#_Toc151296567)

[Testing and Evaluation 6](#_Toc151296568)

[DFS 6](#_Toc151296569)

[BFS 6](#_Toc151296570)

[Dijkstra’s 7](#_Toc151296571)

[A\* 7](#_Toc151296572)

[Source Code 8](#_Toc151296573)

[DFS 8](#_Toc151296574)

[BFS 10](#_Toc151296575)

[Dijkstra’s 12](#_Toc151296576)

[A\* 15](#_Toc151296577)

[References 17](#_Toc151296578)

# Abstract

There are many strategies for finding a path of a maze. This report explores the following common algorithms used for path finding: 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 baseline for more detailed discussion on Dijkstra’s and A\* Algorithm.

# Introduction

The maze used in this project is a grid of value “1” and “0”, in which “1” represents a wall and “0” 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, guide 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, 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 checked, in addition to program running time. Each algorithm is run on the same mazes. The mazes are generated by hands. 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 emphasize of the actual path.

# Algorithms and Implementation

## DFS

Depth first search (DFS) is a simple algorithm based on the idea that to completely explore one path before exploring the next path from the previous fork. The implementation of DFS is based on the stack (in python, list is used for stack implementation).

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

## Djikstra’s Algorithm

## 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 Djikstra 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

3 maze grids 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.

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.

## DFS

A black background with many small dots

Description automatically generated

## BFS

A screenshot of a computer program

Description automatically generated

## Dijkstra’s

Notice that the node is represented by its index instead of the true position. For example, “5” = (1,0) when there’s 5 columns each row.

A screen shot of a computer code

Description automatically generated

## A\*

A screen shot of a computer

Description automatically generated

Comparing the results of each algorithm, we can see that Dijkstra’s and BFS without the code that prevents revisiting visited nodes, way more nodes were explored (due to repetition). However, Dijkstra explores less nodes than BFS when neither deal with repeating nodes. A\* yields the overall best performance.

# Source Code

## DFS

import time

# 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)]

class Node():

    def \_\_init\_\_(self,parent=None,position=None):

        self.position = position

        self.parent = parent

    def \_\_eq\_\_(self, other):

        return self.position==other.position

def reconstruct\_path(currNode):

    path = []

    while currNode is not None:

        path.append(currNode.position)

        currNode = currNode.parent

    return path[::-1]

def dfs(grid,start,goal):

    nodecount = 0

    nodeStack = []

    checkedStack = []

    startNode = Node(None,start)

    goalNode = Node(None,goal)

    nodeStack.append(startNode)

    while(nodeStack):

        currNode = nodeStack.pop()

        checkedStack.append(currNode)

        if currNode == goalNode:

            path = reconstruct\_path(currNode)

            return (path,nodecount)

        for dx, dy in MOVEMENTS: # for each possible steps

            neighbor = (currNode.position[0] + dx, currNode.position[1] + dy)

            if 0 <= neighbor[0] < len(grid) and 0 <= neighbor[1] < len(grid[0]) and grid[neighbor[0]][neighbor[1]]!=1:

                nodecount += 1

                # when neighbor is a legal move put it in queue

                neighborNode = Node(currNode,neighbor)

                if neighborNode not in checkedStack:

                    nodeStack.append(neighborNode)

    return (None,nodecount) # no path found

#example call

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)

    for grid in gridLs:

        goal = (len(grid)-1, len(grid[0])-1)

        starttime = time.time()

        path,nodeexplored = dfs(grid,start,goal)

        endtime = time.time()

        if path:

            print("Path found:", path)

        else:

            print("No path found.")

        print("Explored node count: ",nodeexplored)

        print("runtime: ", endtime-starttime)

## BFS

import queue

import time

# 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)]

class Node():

    def \_\_init\_\_(self,parent=None,position=None):

        self.position = position

        self.parent = parent

    def \_\_eq\_\_(self, other):

        return self.position==other.position

def reconstruct\_path(currNode):

    path = []

    while currNode is not None:

        path.append(currNode.position)

        currNode = currNode.parent

    return path[::-1]

def bfs(grid,start,goal,DNRflag):

    nodecount = 0

    visited = []

    nodeQueue = queue.Queue()

    startNode = Node(None,start)

    goalNode = Node(None,goal)

    nodeQueue.put(startNode)

    if DNRflag:

        visited.append(startNode)

    while(not nodeQueue.empty()):

        currNode = nodeQueue.get()

        if currNode == goalNode:

            path = reconstruct\_path(currNode)

            return (path,nodecount)

        for dx, dy in MOVEMENTS: # for each possible steps

            neighbor = (currNode.position[0] + dx, currNode.position[1] + dy)

            if 0 <= neighbor[0] < len(grid) and 0 <= neighbor[1] < len(grid[0]) and grid[neighbor[0]][neighbor[1]]!=1:

                nodecount += 1

                # when neighbor is a legal move put it in queue

                neighborNode = Node(currNode,neighbor)

                if DNRflag:

                    if neighborNode not in visited:

                        visited.append(neighborNode)

                        nodeQueue.put(neighborNode)

                else:

                    nodeQueue.put(neighborNode)

    return (None,nodecount) # no path found

#example call

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)

    for grid in gridLs:

        for flag in [True, False]:

            goal = (len(grid)-1, len(grid[0])-1)

            starttime = time.time()

            path,nodeexplored = bfs(grid,start,goal,flag)

            endtime = time.time()

            if flag:

                print("No repeating BFS")

            else:

                print("Base implementation BFS, don't check for repeating node")

            if path:

                print("Path found:", path)

            else:

                print("No path found.")

            print(nodeexplored)

            print("runtime: ", endtime-starttime)

## Dijkstra’s

import time

inf = float('inf')

dirc = [[0,1],[1,0],[0,-1],[-1,0]]

#direction = ["right", "down", "left", "up"]

path = []

pre = []

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 = [inf] \* len(Q)

    dis[Q.index(start)] = 0

    return Q, dis

def Path(S, pre, start, end):

    path = []

    point = end

    while point != start:

        path.insert(0, pre[S.index(point)-1])

        point = pre[S.index(point)-1]

    path.append(end)

    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)

            return (path,nodecount)

        dis.pop(Q.index(point))

        for point in S:

            for i in range (0,4):

                nodecount +=1

                nextpoint = [int(point/len(maze[0]))+ dirc[i][0], point%len(maze[0]) + dirc[i][1]]

                if nextpoint[0] < 0 or nextpoint[1] < 0 or nextpoint[0] > len(maze) - 1 or nextpoint[1] > len(maze[0]) - 1:

                    continue

                elif nextpoint[0]\*len(maze[0])+nextpoint[1] in Q:

                    dis[Q.index(nextpoint[0]\*len(maze[0])+nextpoint[1])] = dis\_determ[S.index(point)]+1

                    temp1.append(point)

                    temp2.append(nextpoint[0]\*len(maze[0])+nextpoint[1])

    return False

# Example usage:

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)

    for grid in gridLs:

        goal = (len(grid)-1, len(grid[0])-1)

        starttime = time.time()

        path,nodeexplored = Dijkstra(grid, start, goal)

        endtime = time.time()

        if path:

            print("Path found:", path)

        else:

            print("No path found.")

        print("Number of node explored: ",nodeexplored)

        print("The number of time the a node is push to a heap or heapify is called: ",nodeexplored)

        print("Execution time:", endtime-starttime)

## A\*

import heapq

import time

# 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)]

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]

# Example usage:

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)

    for grid in gridLs:

        goal = (len(grid)-1, len(grid[0])-1)

        starttime = time.time()

        path,nodeexplored,heapCount = astar(grid, start, goal)

        endtime = time.time()

        if path:

            print("Path found:", path)

        else:

            print("No path found.")

        print("Number of node explored: ",nodeexplored)

        print("The number of time the a node is push to a heap or heapify is called: ",heapCount)

        print("Execution time:", endtime-starttime)

# 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.