华东师范大学数据科学与工程学院实验报告

课程名称：AI 基础　　**年级：**2022 级　　**上机实践日期：**2024 年 4 月 14 日

指导教师：杨彬　　**姓名：**唐硕

上机实践名称：project1　　**学号：**10225101447

一、 **实验任务**

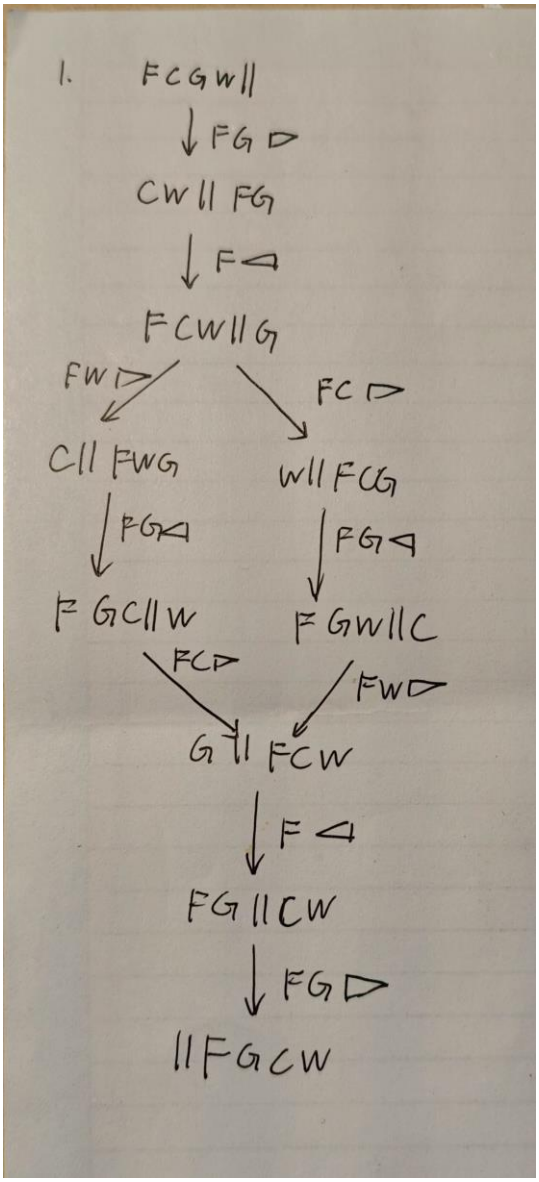
图最短路径问题，八数码问题，迷宫问题

二、 **使用环境**

……

三、 **实验过程**

零：



一、 图最短路径问题

1， BFS

```
def BFS(graph,node):
    queue=[]
    seen=set()
    queue.append(node)
    seen.add(node)

    while (len(queue)>0) :
        ver =  queue.pop(0)
        notes = graph[ver]
        for i in notes:
            if i not in seen:
                queue.append(i)
                seen.add(i)
                pre[i]=ver
    return pre
```

用 queue 存取接下来要访问的节点，用字典 pre 记录前驱节点，首先将根节点加入，seen 维护已经访问过的节点，在 queue 非空时不断 pop 出子节点，遇到没有访问过的节点加入队列和 seen，更新 pre。最后根据返回的 pre 一路回溯累加距离。

```
sions\ms-python.debugpy-2024.2.0-win32-x6
4 5
1 2
2 3
3 4
1 3
1 4
1
PS D:\Code\AI-fundamental\Porject1>
```

2， 朴素版迪杰斯特拉

```
graph=[]
dist=[]
st=[]       #已确定
graph=[[0 for i in range(n)] for j in range(n)]
for _ in range(n):
    dist.append(10000)
    st.append(-1)
for _ in range(0,m):
    char=input().split(' ')
    x=int(char[0])
    y=int(char[1])
    z=int(char[2])
    if(x==y):
        continue
    else:
        graph[x-1][y-1]=z
st[0]=1
for _ in range(0,n):
    dist[_]=graph[0][_]
# print(dist)
for _ in range(1,n):
    t = -1
    for k in range(0,n):
        if (st[k]!=1 and (dist[k] < dist[t] or t == -1)):
            t = k
    st[t]=1
    for k in range(0,n):
        dist[k] = min(dist[k], dist[t] + graph[t][k])
    if(st[n-1]==1):
        print(dist[n-1])
if(st[n-1]==-1):
```

graph、dist 和 st 分别用于存储图的邻接矩阵、节点的最短距离和节点的状态（已确定或未确定）。Dijkstra 的主循环里，从起点出发逐步确定最短路径。每次循环找到距离起点最近且未确定最短路径的节点，并更新其最短距离和状态。检查终点状态变为确定时输出距离。找不到解时 st[]对应位置为-1，输出-1

```
sions\ms-python.debugpy-2024.2.0-win32-x
3 3
1 2 2
2 3 1
1 3 4
3
PS D:\Code\AI-fundamental\Porject1>
```

3， 堆优化版 Dijkstra

```
from queue import PriorityQueue as PQ
pq=PQ()
```

使用优先队列

```
pq.put((0,0))
dist[0]=0
while(not pq.empty()):
    t=(pq.get())
    if(st[t[1]]==1):
        continue
    else:
        st[t[1]]=1

    for k in range(0,n):
        if(dist[k]>dist[t[1]] + graph[t[1]][k]):
            dist[k]=dist[t[1]] + graph[t[1]][k]
            pq.put((dist[k],k))
    if(st[n-1]==1):
        print(dist[n-1])
```

优先队列里存放的（dist[k],k）第一个是距离，第二个是节点序号，默认会按照距离升序排序，每次 pq.get 可以拿到距离最短的相邻节点，把循环变成对数的复杂度

```
PS D:\Code\AI-fundamental\Porject1>  d:; c
sions\ms-python.debugpy-2024.2.0-win32-x64
3 3
1 2 2
2 3 1
1 3 4
3
PS D:\Code\AI-fundamental\Porject1>
```

二、 八数码问题

1．DFS 解决可解性问题

```python
def dfs(char,index):
    global checked
    if char==goal:
        print(1)
        exit(0)
    if char in checked:
        return 0
    else:
        checked.append(char)
    tmp=char
    if index<6:
        new=move(tmp,index,'down')
        dfs(new,index+3)
    if index%3!=0:
        new=move(tmp,index,'left')
        dfs(new,index-1)
    if index%3!=2:
        new=move(tmp,index,'right')
        dfs(new,index+1)
    if index>2:
        new=move(tmp,index,'up')
        dfs(new,index-3)
    return 0
```

通过递归，下左右上的顺序，走到一个方向无法前进时回溯，如果走完无解则输出-1，找到解时输出 1 表示可解。修改递归限制能够运行，时间较长。

```
PS D:\Code\AI-fundamental\Porject1>  d:; cd 'd:
sions\ms-python.debugpy-2024.2.0-win32-x64\bund
2 3 4 1 5 x 7 6 8
1
PS D:\Code\AI-fundamental\Porject1>
```

2.BFS 求解八数码最小移动步数

网上看到教程将状态矩阵变成一个 9 位数字存效率很高，需要两个转换函数：

```
int toInt() {
    int now = 0;
    for(int i = 0; i < 3; i++)
        for(int j = 0; j < 3; j++)
            now = now * 10 + mat[i][j];
    return now;
}
void toMatrix(int s) {
    int div = 100000000;
    for(int i = 0; i < 3; i++)
        for(int j = 0; j < 3; j++) {
            mat[i][j] = (s / div) % 10;
            if(!mat[i][j]) zx = i, zy = j;
            div /= 10;
        }
}
```

因此需要把 x 变成 0：

```
 string input;
 getline(cin, input);
 input.erase(std::remove(input.begin(), input.end(), ' '), input.end());
 std::replace(input.begin(), input.end(), 'x','0'), input.end();
```

开始前计算逆序数验证可解：

```
int inversionCount(const std::string& str) {
    int count = 0;
    for (size_t i = 0; i < str.size(); ++i) {
        if (str[i] == '0') continue;
        for (size_t j = i + 1; j < str.size(); ++j) {
            if (str[j] != '0' && str[j] < str[i]) {
                ++count;
            }
        }
    }
    return (count % 2 == 0) ? 0 : -1;
}
```

BFS 主函数：

```
void bfs(int s) {
    if(s == ::end) return ;
    bool flag;
    state[s] = 1, state[::end] = 2;
    ans[s] = 0, ans[::end] = 1;
    q1.push(s), q2.push(::end);
    while(!q1.empty() && !q2.empty()) {
        flag = 0;
        int t;
        if(q1.size() > q2.size()) {
            t = q2.front();
            q2.pop();
        }else {
            t = q1.front();
            q1.pop();
            flag = 1;
        }
        toMatrix(t);
        for(int i = 0; i < 4; i++) {
            int num;
            int nx = dx[i] + zx;
            int ny = dy[i] + zy;
            if(nx >= 0 && nx < 3 && ny >= 0 && ny < 3) {
                swap(mat[zx][zy], mat[nx][ny]);
                num = toInt();
                if(!ans.count(num)) {
                    ans[num] = ans[t] + 1;
                    state[num] = state[t];

                    if(flag) q1.push(num);
                    else q2.push(num);

                }else if(state[t] + state[num] == 3){
                    cnt = ans[t] + ans[num];
                    return;
                }
                swap(mat[zx][zy], mat[nx][ny]);
            }
        }
    }
```

Bfs 的速度不太能通过，在网上查到双向 BFS，从起点和终点同时搜索，记录层数和状态，状态用 1 和 2 表示被起点找到，被终点找到，每次扩展时如果发现新的一层如果状态和为 3 说明这个点能同时被起点和终点连接，是一条通路，移动步数记录在 ans 里，循环每次只走一步，通过 flag 来记录本次是从起点开始还是终点开始，然后往四个方向扩展子节点，如果子节点没被访问过就根据 flag 加入对应的 queue，然后设置子节点的状态和所需移动步数，直到出现一步子节点扩展之后发现该子节点可以被另一端到达就是找到了解，只需拿该子节点当前深度加上另一节点需要的深度就是所需移动的步数。

```
PS D:\Code\C>  & 'c:\Use
-btsnjfxz.yoc' '--stdout
xe=D:\Software\Mingw64\x
2 3 4 1 5 x 7 6 8
19
PS D:\Code\C>
```

3.迪杰斯特拉

```
int inversionCount(const std::string& str) {
    int count = 0;
    for (size_t i = 0; i < str.size(); ++i) {
        if (str[i] == '0') continue; // Ignore digit 0
        for (size_t j = i + 1; j < str.size(); ++j) {
            if (str[j] != '0' && str[j] < str[i]) {
                ++count;
            }
        }
    }
    return (count % 2 == 0) ? 0 : -1;
}
```

判断

设置输入和目标状态

```
stringstream ss(input);
for (int i = 0; i < 3; ++i) {
    for (int j = 0; j < 3; ++j) {
        string cell;
        ss >> cell;
        if (cell == "x") {
            start.board[i][j] = 0;
            start.x = i;
            start.y = j;
        } else {
            start.board[i][j] = stoi(cell);
        }
    }
}
input.erase(std::remove(input.begin(), input.end(), ' '), input.end());
std::replace(input.begin(), input.end(), 'x','0'), input.end();
int ju=inversionCount(input);
if(ju){
    cout << -1 << "\n";
    return 0;
}

target.board[0][0] = 1; target.board[0][1] = 2; target.board[0][2] = 3;
target.board[1][0] = 4; target.board[1][1] = 5; target.board[1][2] = 6;
target.board[2][0] = 7; target.board[2][1] = 8; target.board[2][2] = 0;
target.x = 1; target.y = 0;
```

为了使用上面的函数还是把 x 替换为 0，

```
        return node.steps;
    }

    if (node.steps > dist[node.state]) {
        continue;
    }

    int dx[] = {0, 0, -1, 1};
    int dy[] = {-1, 1, 0, 0};

    for (int k = 0; k < 4; ++k) {
        int nx = node.state.x + dx[k];
        int ny = node.state.y + dy[k];

        if (nx < 0 || nx >= 3 || ny < 0 || ny >= 3) {
            continue;
        }

        State next = node.state;
        swap(next.board[node.state.x][node.state.y], next.board[nx][ny]);
        next.x = nx;
        next.y = ny;

        int newDist = node.steps + 1;
        if (!dist.count(next) || newDist < dist[next]) {
            dist[next] = newDist;
            pq.push(HeapNode(next, newDist));
        }
    }
```

主体部分类似，找到不在距离确定集合里，同时离集合里节点最近的节点，更新下一步，从优先队列中取出步数最小的节点。如果当前节点为目标状态，则返回当前步数，表示找到了最短路径。如果当前节点的步数大于已知的最短步数，有更短的路径到达该节点，跳过。否则计算新的步数 newDist，并更新最短步数和优先队列。在结构体里面设置了比较两个节点步数大小和比较两个矩阵是否相等的方法；

```
    size_t hash() const {
        size_t h = 0;
        for (int i = 0; i < 3; ++i) {
            for (int j = 0; j < 3; ++j) {
                h = h * 10 + board[i][j];
            }
        }
        return h;
    }
```

利用 hash 去重，实际上和上面的 bfs 一样是把状态变成一个 9 位十进制数字。

```
PS D:\Code\C> & 'c:\Users\Gav1n\.vscode\extensions\ms-vscode.cpptools-1.19.9-win32-x64\debugAdapters\bin\WindowsDebugLauncher.exe' '--stdin=Microsoft-MIEngine-In-0flcls1e.rp4' '--stdout=Microsoft-MIEngine-Out-pqhtrup4.lhr' '--stderr=Microsoft-MIEngine-Error-k5jtan4h.xam' '--pid=Microsoft-MIEngine-Pid-1tcdi12e.5f' '--dbgExe=D:\Software\Mingw64\x86_64-8.1.0-release-win32-seh-rt_v6-rev0\mingw64\bin\gdb.exe' '--interpreter=mi'
2 3 4 1 5 x 7 6 8
19
```

4.A*输出一条可行路径

输入判断是否有解

```
string input;
getline(cin, input);
input.erase(std::remove(input.begin(), input.end(), ' '), input.end());
std::replace(input.begin(), input.end(), 'x','0'), input.end();
int ju=inversionCount(input);
if(ju){
    cout << "unsolvable" << "\n";
    return 0;
}
```

还是选择用一穿数字表示一个状态，同样与 BFS 类似，在大循环里面每次遍历四个子状态，估价函数是计算当前值与目标值的曼哈顿距离。

```
string ans;
string target;
target = "123456780";
dist[input] = 0;
pq.push({Manhattan_Distance(input) , input});

while(!pq.empty())
{
    string s = pq.top().second;
    pq.pop();
    if(s == "123456780")
        break;
    int k = s.find('0');

    for(int i = 0; i < 4; i++) {
        int x = k / 3 + dx[i], y = k % 3 + dy[i];
        if(x < 0 || x >= 3 || y < 0 || y >= 3)
            continue;
        string t(s);
        swap(t[k], t[x * 3 + y]);

        if(!dist.count(t) || dist[t] > dist[s] + 1) {
            dist[t] = dist[s] + 1;
            path[t] = {i, s};
            pq.push({dist[t] + Manhattan_Distance(t), t});
        }
    }
}
```

代价函数：

```
int Manhattan_Distance(string &s) {
    int ret = 0;
    for(int i = 0; i < 9; i++)
    {
        if(s[i] != '0')
            ret += abs(i / 3 - (s[i] - '1') / 3) + abs(i % 3 - (s[i] - '1') %
    }
    return ret;
}
```

```
PS D:\Code\C>  & 'c:\User
-exu5bnrc.4tk' '--stdout=
xe=D:\Software\Mingw64\x8
2 3 4 1 5 x 7 6 8
ldruullddrurdllurrd
PS D:\Code\C>
```

三．迷宫问题

```
def valid(matrix, visited, x, y):
    n, m = len(matrix), len(matrix[0])
    return 0 <= x < n and 0 <= y < m and matrix[x][y] == 0 and not visited[
```

判断移动是否越界

```
n, m = map(int, input().split())
matrix = []
for _ in range(n):
    row = list(map(int, input().split()))
    matrix.append(row)
global maze
maze = [[0 for _ in range(m)] for _ in range(n)]
```

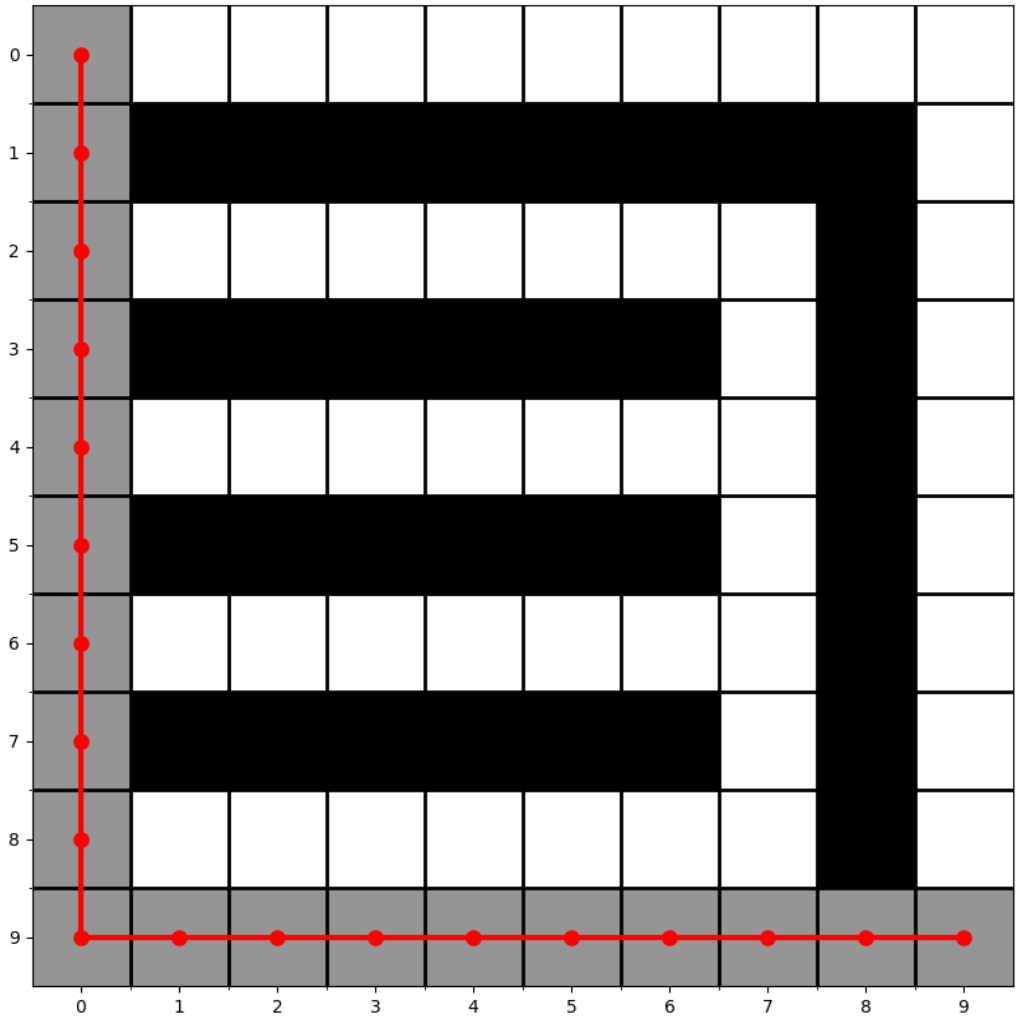
输入记录在 matrix 用于传给函数,maze 记录搜索经过的节点和输入中的墙，用于生成图像。

```
for i in range(len(maze)):
    for j in range(len(maze[0])):
        if maze[i][j] == 2:
            plt.fill([j-0.5, j+0.5, j+0.5, j-0.5], [i-0.5, i-0.5, i+0.5, i+0.5], color='light
```

特定值染成蓝色表示搜索时经过的节点

1. DFS

ii.

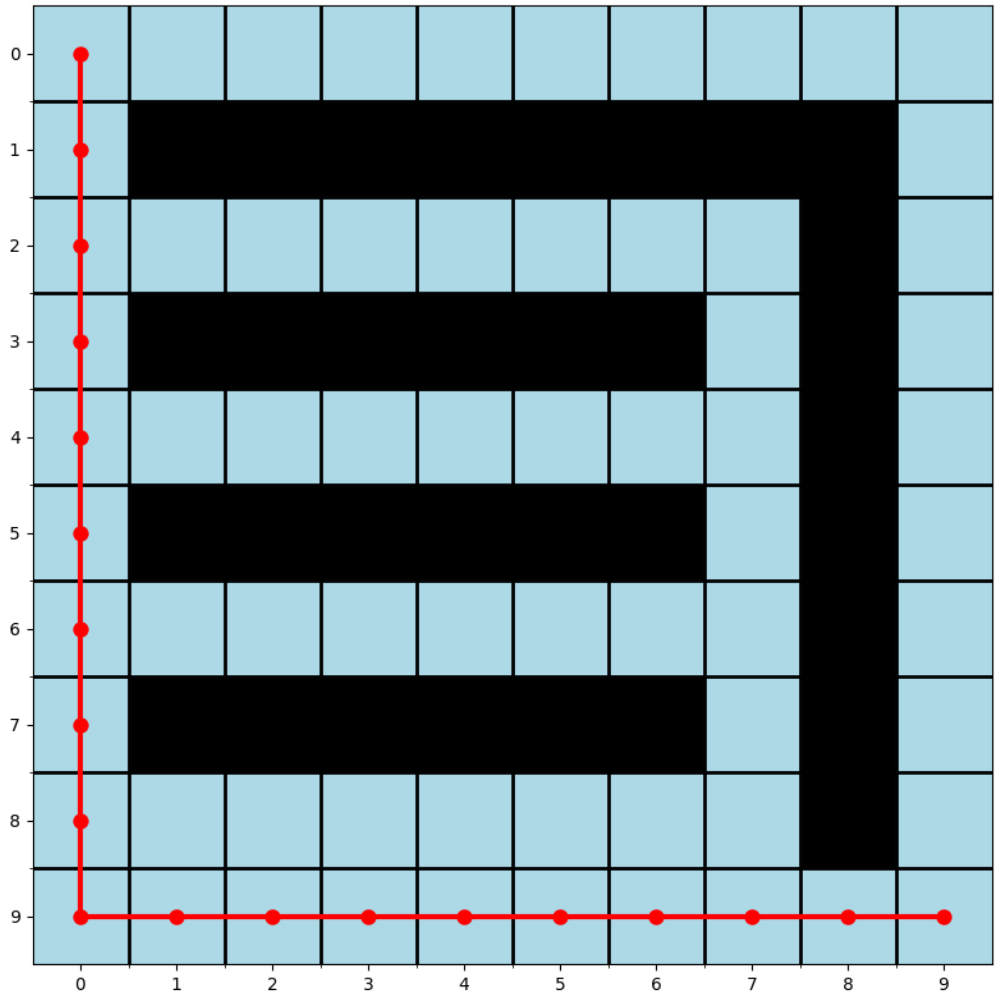


```python
def dfs(matrix, visited, path, x, y, dest_x, dest_y):
    global maze

    if x == dest_x and y == dest_y:
        path.append((x, y))
        return True
    if valid(matrix, visited, x, y):
        maze[x][y]=2
        visited[x][y] = True
        path.append((x, y))
        if (dfs(matrix, visited, path, x + 1, y, dest_x, dest_y) or
                dfs(matrix, visited, path, x, y + 1, dest_x, dest_y) or
                dfs(matrix, visited, path, x - 1, y, dest_x, dest_y) or
                dfs(matrix, visited, path, x, y - 1, dest_x, dest_y)):
            return True
        path.pop()
        visited[x][y] = False
    return False

def find_path_dfs(matrix):
    global maze
    n, m = len(matrix), len(matrix[0])
    visited = [[False for _ in range(m)] for _ in range(n)]
    path = []
    if dfs(matrix, visited, path, 0, 0, n - 1, m - 1):
        # for i in range(n):
        #     for j in range(m):
        #         if visited[i][j]:
        #             matrix[i][j] = 2
        return path
```

iii.

iv. Visited 中的 bool 类型存放是否被访问过，每次递归往下加入 path，失败回溯时 pop 掉，path 中留下的是成功路径节点，传给绘图函数。

1. BFS



v.

```python
def bfs(matrix, visited, start_x, start_y, dest_x, dest_y):
    queue = deque([(start_x, start_y, [])])
    visited[start_x][start_y] = True

    while queue:
        x, y, path = queue.popleft()
        path = path + [(x, y)]

        if x == dest_x and y == dest_y:
            return path

        for dx, dy in [(1, 0), (-1, 0), (0, 1), (0, -1)]:
            nx, ny = x + dx, y + dy
            if valid(matrix, visited, nx, ny):
                queue.append((nx, ny, path[:]))
                visited[nx][ny] = True

    return None
```

vi.

vii. 同样在一次大循环里面遍历四个可能的方向，加入队列，

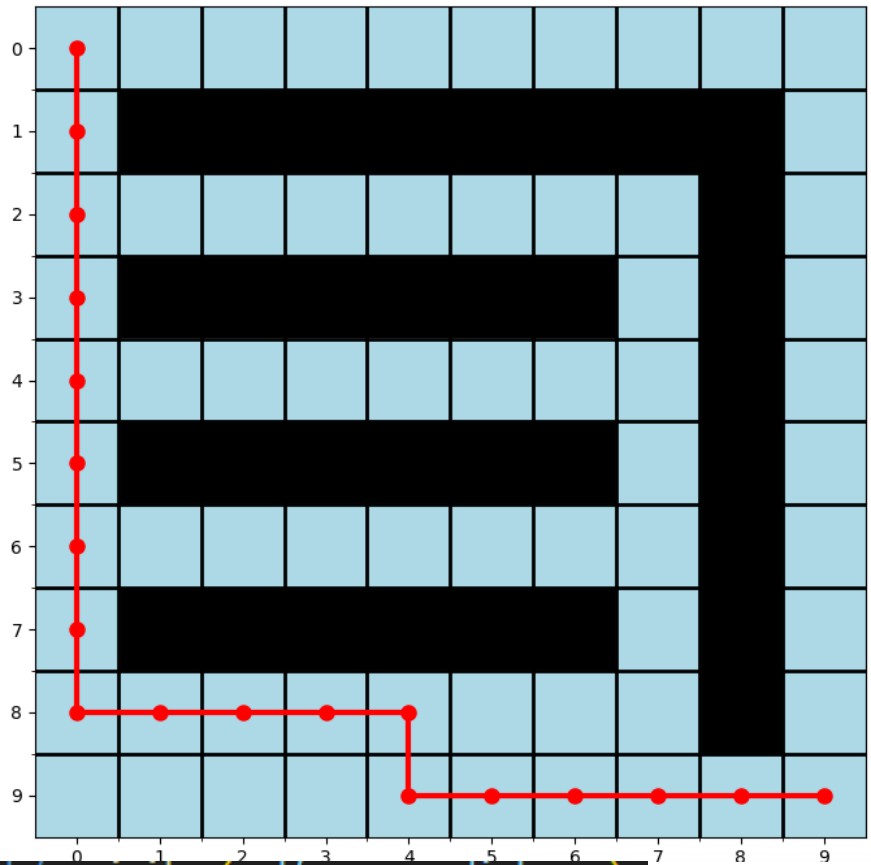
viii. queue.append((nx, ny, path[:]))确保每次的 path 不会改变原先的，使得最终输出时

不会有失败回溯的节点。

ix.
```
if path_bfs:
    for i in range(n):
        for j in range(m):
            if matrix[i][j] == 1:
                maze[i][j] = 4
            if visited[i][j]==True:
                maze[i][j] = 2

    maze[n - 1][m - 1] = 2
```

x. 把迷宫和 visited 结合存到 maze 方便绘图。

1. 迪杰斯特拉

xi.


xii.
```
def __init__(self, x, y, distance):
    self.x = x
    self.y = y
    self.distance = distance
    self.parent = None
```

xiii. 与 DFS，BFS 不同，通过找到路径之后回溯节点的父节点得到 path，

xiv.

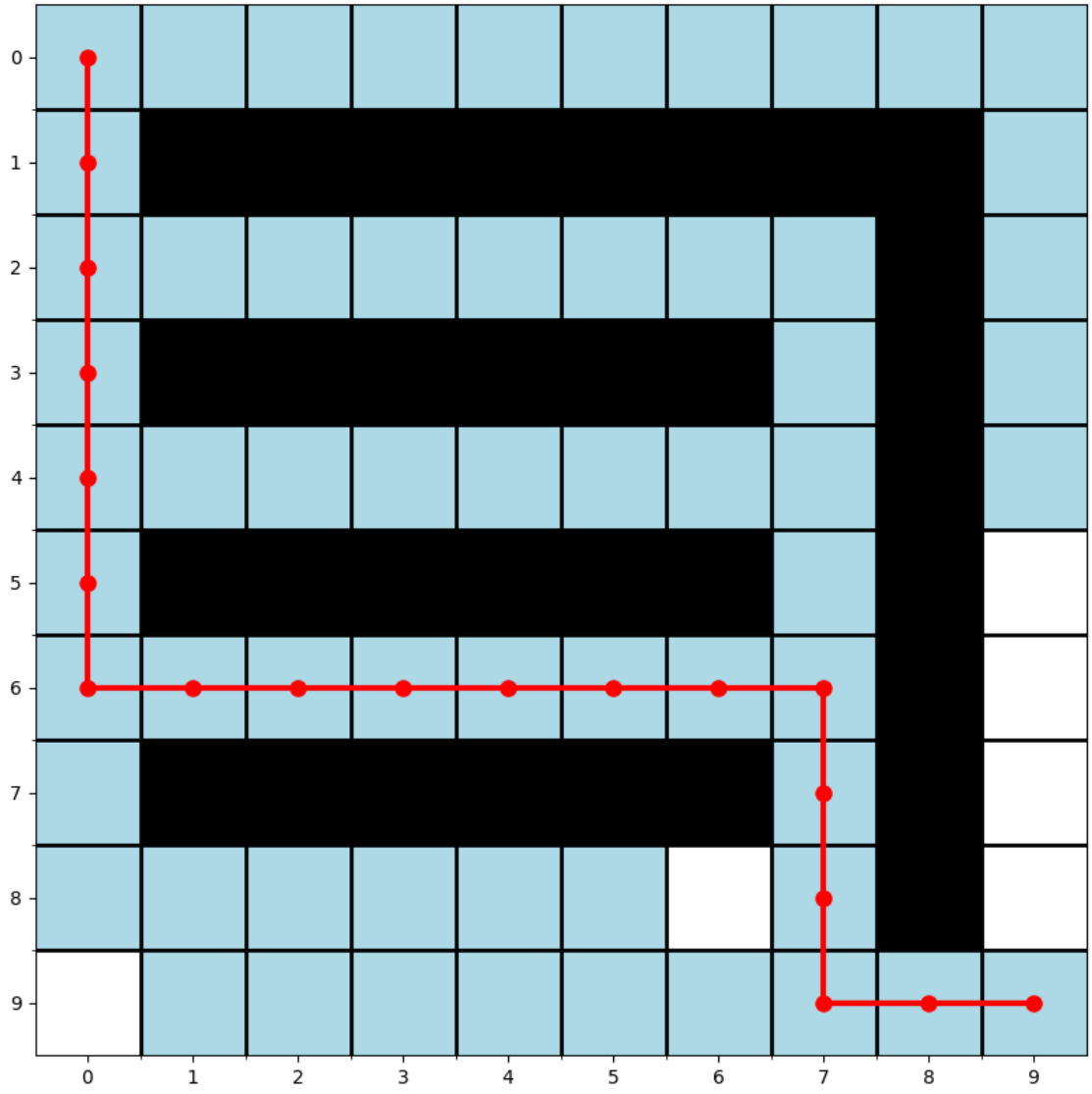
```python
def dijkstra(matrix, start_x, start_y, dest_x, dest_y):
    n, m = len(matrix), len(matrix[0])
    visited = [[False for _ in range(m)] for _ in range(n)]
    directions = [(1, 0), (-1, 0), (0, 1), (0, -1)]
    min_heap = []
    heapq.heappush(min_heap, Node(start_x, start_y, 0))

    while min_heap:
        curr = heapq.heappop(min_heap)
        x, y, distance = curr.x, curr.y, curr.distance

        if x == dest_x and y == dest_y:
            path = []
            while curr:
                path.append((curr.x, curr.y))
                visited[curr.x][curr.y] = True
                curr = curr.parent
            path.reverse()
            return path, visited

        if not visited[x][y]:
            visited[x][y] = True
```

每次大循环内检查四个方向是否可行并添加进最小堆设置父节点信息。

xv.

```python
if not visited[x][y]:
    visited[x][y] = True

    for dx, dy in directions:
        nx, ny = x + dx, y + dy
        if 0 <= nx < n and 0 <= ny < m and matrix[nx][ny] == 0 and not visited[nx][r
            new_distance = distance + 1
            heapq.heappush(min_heap, Node(nx, ny, new_distance))
            min_heap[-1].parent = curr
```

1. A*



xvi.

```python
def a_star(matrix, start_x, start_y, dest_x, dest_y):
    n, m = len(matrix), len(matrix[0])
    visited = [[False for _ in range(m)] for _ in range(n)]
    directions = [(1, 0), (-1, 0), (0, 1), (0, -1)]
    min_heap = []
    heapq.heappush(min_heap, Node(start_x, start_y, 0, heuristic(Node(start_x, start_y, 0, 0), dest_x, dest_y)))

    while min_heap:
        curr = heapq.heappop(min_heap)
        x, y = curr.x, curr.y

        if x == dest_x and y == dest_y:
            path = []
            while curr:
                path.append((curr.x, curr.y))
                visited[curr.x][curr.y] = True
                curr = curr.parent
            path.reverse()
            return path, visited

        visited[x][y] = True

        for dx, dy in directions:
            nx, ny = x + dx, y + dy
            if 0 <= nx < n and 0 <= ny < m and matrix[nx][ny] == 0 and not visited[nx][ny]:
                new_node = Node(nx, ny, curr.g_cost + 1, heuristic(Node(nx, ny, 0, 0), dest_x, dest_y))
                new_node.parent = curr
                heapq.heappush(min_heap, new_node)
```

同样的输出 path 和 visited，与输入合成 maze 给绘图函数，这里是通过记录前驱来回溯使得 path 里能够只留存一条可行路径：

```python
class Node:
    def __init__(self, x, y, g_cost, h_cost)
        self.x = x
        self.y = y
        self.g_cost = g_cost
        self.h_cost = h_cost
        self.f_cost = g_cost + h_cost
        self.parent = None
```

与迪杰斯特拉类似，用优先队列取得较小值。