无信息搜索解决迷宫问题

（1）算法原理

- 使用**5种算法**解决迷宫问题。

1. 存储结构：声明一个树结点存储数据，每个结点存储当前点的二维坐标、父亲结点、从起始到当前点的路径代价。
2. BFS

   将初始点放入队列，进入循环，每次从队列头取出结点，判断当前结点是否为终点，如果是直接返回，否则对当前结点进行扩展，从四个运动方向进行遍历，每个运动方向中更新当前的结点坐标，如果越界、结点已经访问过或者是墙则放弃当前的运动方向探索，否则生成新的结点，将当前结点位置置为已访问，加入队列。

3. DFS

   使用递归，对每一次递归，判断当前结点是否是终点，如果是，返回True，否则对当前结点进行扩展，遍历四个运动方向，对每个方向，更新坐标，判断新的坐标是否越界、是否已经访问过或是墙，如果是放弃当前方向的探索，否则生层新的结点，将当前结点的位置设为已访问，递归调用下一个DFS函数，判断返回值，如果是True，则直接返回True，否则要进行回溯，将该节点的访问从已访问设为未访问，继续下一个方向探索。

4. UCS一致代价

   基于BFS，方法跟BFS大致相同，不同之处在于UCS队列是使用优先队列，每次从队列中选出路径代价最小的结点进行扩展，同时在设置结点是否访问is_visited也有不同，将结点加入队列前不会设置is_visited为True，而是将结点从队列取出进行扩展时才设置is_visited为True，这是为了更新最佳的路径，每次将结点a加入队列前，首先检查队列中是否有相同状态的结点b，如果有则进行比较a与b的路径代价，如果a路径代价比b小，则将a替换b，否则不进行替换，并且不再将a加入队列，如果没有相同状态的结点，则将a加入队列。

5. 迭代加深搜索

   在DFS基础上进行扩展，设置步数递增循环，每一次循环调用一次深度受限搜索，指定当前搜索的最大深度，深度设置为探索的步数，因为每增加一层，步数加1，每次调用深度受限函数，都要初始化根结点、访问矩阵is_visited，调用DFS函数，步骤跟DFS一样，当搜索达到指定深度但又没有找到终点，返回False，否则返回True。

6. 双向搜索

   从起点和终点同时进行检测，依次将起点和终点分别加入两个队列front_queue和back_queue，为了达到最优先，采用BFS思想，一层层遍历，找到最优解，因此每次循环分别对两个队列的结点进行扩展，首先从起点队列front_queue取出结点a，，对a进行扩展不同方向的结点a1，此时判断back_queue中是否有结点状态与a1状态相同，如果相同，说明已找到通路，基于BFS原理，可知此时为最优路径，直接返回，否则继续扩展；

同理，从back_queue取出结点b，对b从四个方向扩展b1，判断front_queue中是否有结点状态与b1相同，有则说明已经相同，可以返回，否则继续扩展，直到两者连通。

(2) 伪代码

1. BFS

```
Algorithm BFS
    node <- Tree_node(start_state,parent=None,path_cost=0)
    push node to my_queue
    while my_queue.size>0 do #直到队列为空
        get node from the front of my_queue
        if node.state == destination_state do  #到达终点
            return
        for move in every movement do  #各个运动方向
            update node.state to new_state
            if new_state crossing the border or visited or hit the wall do  #不符合条件
                continue
            else
                newNode <-Tree_node(new_state,node,node.path_cost+1)
                update newNode.state to visited  #添加结点为已访问
                push newNode to my_queue  #入队
```

2. DFS

```
Algorithm DFS
    function DFS(node): return success or failure
        if node.state == destination_state do  #到达终点
            return True  #返回正确结果给上一级调用的函数
        for move in every movement do  #各个运动方向
            update node.state to new_state
            if new_state crossing the border or visited or hit the wall do  #不符合条件
                continue
            else
                newNode <-Tree_node(new_state,node,node.path_cost+1)
                update newNode.state to visited  #添加结点为已访问
                #递归调用DFS函数，如果返回结果是False，则回溯，将结点的状态置为not visited
                if not DFS(node) do
                    update newNode.state to not visited
                #否则返回正确给上一级调用的函数
                else
                    return True
     #没有成功找到路径，返回错误给上一级函数
     return False
```

3. UCS一致代价

```
Algorithm UCS
    node <- Tree_node(start_state,parent=None,path_cost=0)
    push node to priority_queue  #初始结点入队
    while my_queue.size>0 do  #直到结点为空
        get min path_cost node from the priority_queue #从优先队列取出最小路径代价的结点
        if node.state == destination_state do #到达终点
            return
        update newNode.state to visited  #出队才添加结点为已访问
        for move in every movement do #检测各个运动方向
            update node.state to new_state
            if new_state crossing the border or visited or hit the wall do #不符条件
                continue
            else
                newNode <-Tree_node(new_state,node,node.path_cost+1)
                #寻找队列中是否有与该结点状态相同的，如果有比较路径代价，小于则替换，否则不作替换，不入
队
                for item in priority_queue.items do
                    if newNode.state = item.state do
                        if newNode.path_cost < item.path_cost do
                            replace item with newNode
                         else
                            do nothing
                #如果队列中没有与该结点状态相同的，入队
                if newNode not in priority_queue do
                    push newNode to priority_queue
```

4. 迭代加深搜索

```
Algorithm iterative_deeping_search

    function iterative_deeping_search () return success or failure
        #逐步加深搜索深度
        for limit <- 1 to 10000 do
            在当前深度找到直接返回
            if depth_limit_search(limit) do
                return True
        return False

    function depth_limit_search (limit) return success or failure
        initial node and visited set #初始化结点和访问集
         return DFS(node,limit) #调用深度搜索函数

    function DFS(node,limit): return success or failure
        if node.state == destination_state do  #到达终点
            return True  #返回正确结果给上一级调用的函数
        if limit <= 0 do #如果已经达到指定深度，没有找到结点，返回失败
            return False
        for move in every movement do  #各个运动方向
            update node.state to new_state

            if new_state crossing the border or visited or hit the wall do  #不符合条件
```

```
                continue
            else
                newNode <-Tree_node(new_state,node,node.path_cost+1)
                update newNode.state to visited  #添加结点为已访问
                #递归调用DFS函数，如果返回结果是False，则回溯，将结点的状态置为not visited
                if not DFS(node,limit-1) do
                    update newNode.state to not visited
                #否则返回正确给上一级调用的函数
                else
                    return True
    #没有成功找到路径，返回错误给上一级函数
    return False
```

5. 双向搜索

```
Algorithm Bidirectional search
    node1 <- Tree_node(start_state,parent=None,path_cost=0)
    push node1 to front_queue  #从初始点出发的前端队列
    node2 <- Tree_node(destination_state,parent=None,path_cost=0)
    push node2 to back_queue  #从终点出发的后端队列

    while front_queue.size>0 or back_queue.size>0 do  #遍历直到两个队列均为空
        #处理前端队列
        if front_queue not null do
            get node from the front of front_queue #出队
            for move in every movement do #检测各个运动方向
                update node.state to new_state
                if new_state crossing the border or hit the wall do
                    continue
                else
                    newNode <-Tree_node(new_state,node,node.path_cost+1)
                    #如果新结点能在后端队列找到，说明已经交汇，找到通路，返回
                    if newNode.state in back_queue do
                        return
                    if newNode is not visited do
                        update newNode.state to visited
                        push newNode to front_queue
        #处理后端队列
        if back_queue not null do
            get node from the front of back_queue
            for move in every movement do #各个运动方向检测
                update node.state to new_state
                if new_state crossing the border or hit the wall do
                    continue
                else
                    newNode <-Tree_node(new_state,node,node.path_cost+1)
                    #如果新结点能在前端队列找到，说明已经交汇，找到通路，返回
                    if newNode.state in front_queue do
                        return
                    if newNode is not visited do
                        update newNode.state to visited
```

```
                        push newNode to back_queue
```

(3) 关键代码

1. 存储结构：使用类存储结点的状态、父结点、路径代价。

```
class Tree_Node(object):
    def __init__(self,state,parent,path_cost):
        self.state = state #结点状态，二维坐标
        self.parent = parent #父结点
        self.path_cost = path_cost #从起始点到当前结点的路径代价
```

2. BFS

- 初始化结点，加入队列

```
    root = Tree_Node((start_x, start_y), None, 0) #初始化起始点
    my_queue = []
    my_queue.append(root)  #入队
```

- 每次从队列头取出结点，判断是否到达终点。

```
        node = my_queue.pop(0) #从队列头取出结点
        #到达终点，返回
        if node.state[0] == end_x and node.state[1] == end_y:
            print(count)
            return node
```

- 检测各个运动方向，更新x、y坐标，判断是否越界。

```
        for i in range(len(move)):
            x = node.state[0] + move[i][0] #更新状态坐标
            y = node.state[1] + move[i][1]
            #如果越界，不进行扩展
            if x < 0 or x >= maze_x or y < 0 or y >= maze_y:
                continue
```

- 如果没有访问或者不是碰墙，进行扩展，创建新结点，入队。

```
            if is_visited[x][y] == 0 and maze_list[x][y] != '1':
                path_cost = node.path_cost + 1 #更新路径代价
                is_visited[x][y] = 1 #更新访问集
                state = (x,y) #更新状态
                parent = node #设置父结点
                #创建新结点
                newNode = Tree_Node(state,parent,path_cost)
                my_queue.append(newNode) #入队
                count += 1
```

3. DFS

- DFS采用递归调用搜索，进入函数，判断当前结点是否到达终点。

```
    #如果到达终点，返回True给上一级
    if (node.state[0] == end_x and node.state[1] == end_y):
        dst_node_list.append(node)
        return True
```

- 检测各个运动方向，更新x、y坐标，检查是否越界。

```
        x = node.state[0] + move[i][0]
        y = node.state[1] + move[i][1]
        #如果越界，不扩展
        if x < 0 or x >= maze_x or y < 0 or y >= maze_y:
            continue
```

- 如果新的状态没有被访问并且没有碰墙，则创建新结点，递归调用DFS，并且根据返回值判断是否要回溯，如果是False，则回溯，将访问集更新，否则返回True给上一级。

```
    #如果结点没有访问并且不碰墙，扩展
    if is_visited[x][y] == 0 and maze_list[x][y] != '1' :
        is_visited[x][y] = 1
        state = (x,y)
        parent = node
        path_cost = node.path_cost + 1
        newNode = Tree_Node(state,parent,path_cost)#创建新结点
        #递归调用DFS，根据返回结果，如果是False，则回溯，将访问集更新
        if not DFS(end_x,end_y,maze_x,maze_y,is_visited,move,maze_list,newNode,dst_node_list):
            is_visited[x][y] = 0
            #否则返回True给上一级
        else :
            return True
```

4. UCS

- 由于是基于BFS，因此有关BFS的实现不再重复，主要描述UCS算法改进的地方。
- 每次从队列中取出路径代价最小的结点出队。

```
        min = my_queue[0].path_cost
        index = 0
        #寻找最小路径代价的结点
        for i in range(len(my_queue)):
            if min > my_queue[i].path_cost:
                min = my_queue[i].path_cost
                index = i
        node = my_queue.pop(index)
```

- 将结点入队前更新访问集改为出队时更新访问集。

```
        is_visited[node.state[0]][node.state[1]] = 1
```

- 创建新结点a入队前，先检查队列中是否有结点b状态与a相同，如果有，比较a与b的路径代价，如果a<b，则将a替换b，否则不替换，a也不入队，如果没有找到这样结点b，a正常入队。

```
        newNode = Tree_Node(state,parent,path_cost)
        index1 = 0
        is_find = 0
        for k in range(len(my_queue)):
            if my_queue[k].state == state:
                if my_queue[k].path_cost > path_cost:
                    index1 = k
                    is_find = 1
                    break
                is_find = 2
                break
        if is_find == 1:
            my_queue.pop(index1)
            my_queue.append(newNode)
        elif is_find == 2:
            continue
        else:
            my_queue.append(newNode)
            count += 1
```

5. 迭代加深搜索

- 由于是基于DFS，因此有关DFS的实现不再重复，主要描述迭代加深算法改进的地方。
- 递增限制深度，从0到1000开始递增，调用深度受限函数depth_limit_search，判断返回值，如果为True，直接返回，否则继续递增，最终如果没有找到，返回False。

```
    for limit in range(1000):
        result = depth_limit_search(start_x,start_y,end_x,end_y,maze_x,
                                    maze_y,move,maze_list,dst_node_list,limit)
        if result == True:
            return True
    return False
```

- 进入depth_limit_search函数，初始化初始结点，访问集，调用DFS函数，如果DFS在当前深度找到终点，返回True，否则返回False。

```
    is_visited = np.zeros((maze_x,maze_y))
    is_visited[start_x][start_y] = 1
    root = Tree_Node((start_x,start_y),None,0)
    return DFS(end_x,end_y,maze_x,maze_y,is_visited,move,maze_list,root,dst_node_list,limit)
```

- DFS函数实现跟上面DFS相似，不同的就是加入limit判断，每次递归调用DFS，limit都会减一，每次进入DFS函数，在判断是否到达终点后，再判断limit是否为0，如果是，说明已经达到指定深度，不能再进行扩展，因此返回False。

```
DFS(end_x,end_y,maze_x,maze_y,is_visited,move,maze_list,newNode,dst_node_list,limit-1)
```

```
    if (limit <= 0) :
        return False
```

6. 双向搜索

- 由于是基于BFS，因此有关BFS的实现不再重复，主要描述双向搜索算法改进的地方。
- 将一个队列改为两个队列，一个前端队列，一个后端队列。

```
    start_node = Tree_Node((start_x, start_y), None, 0)
    end_node = Tree_Node((end_x,end_y),None,0)
    front_queue = []
    front_queue.append(start_node) #初始点出发的队列
    back_queue = []
    back_queue.append(end_node) #从终点出发的队列
```

- 每次遍历分别遍历两个队列，以前端队列为例，取出结点，对结点进行扩展，判断新扩展的结点的状态是否在后端队列中出现，如果出现说明两个队列扩展发生交集，连成通路，此时找到路径。

前端队列代码举例，后端队列不重复补充：

```
    if maze_list[x][y] != '1':
        path_cost = node.path_cost + 1
        state = (x, y)
        parent = node
        newNode = Tree_Node(state, parent, path_cost)
        #在后端队列寻找相同结点
        for k in range(len(back_queue)):
            #找到说明路径连通，直接返回
            if newNode.state == back_queue[k].state:
                return (newNode,back_queue[k])
    if is_visited[x][y] == 0 and maze_list[x][y] != '1':
        is_visited[x][y] = 1
        front_queue.append(newNode)
```

（4）创新点&优化

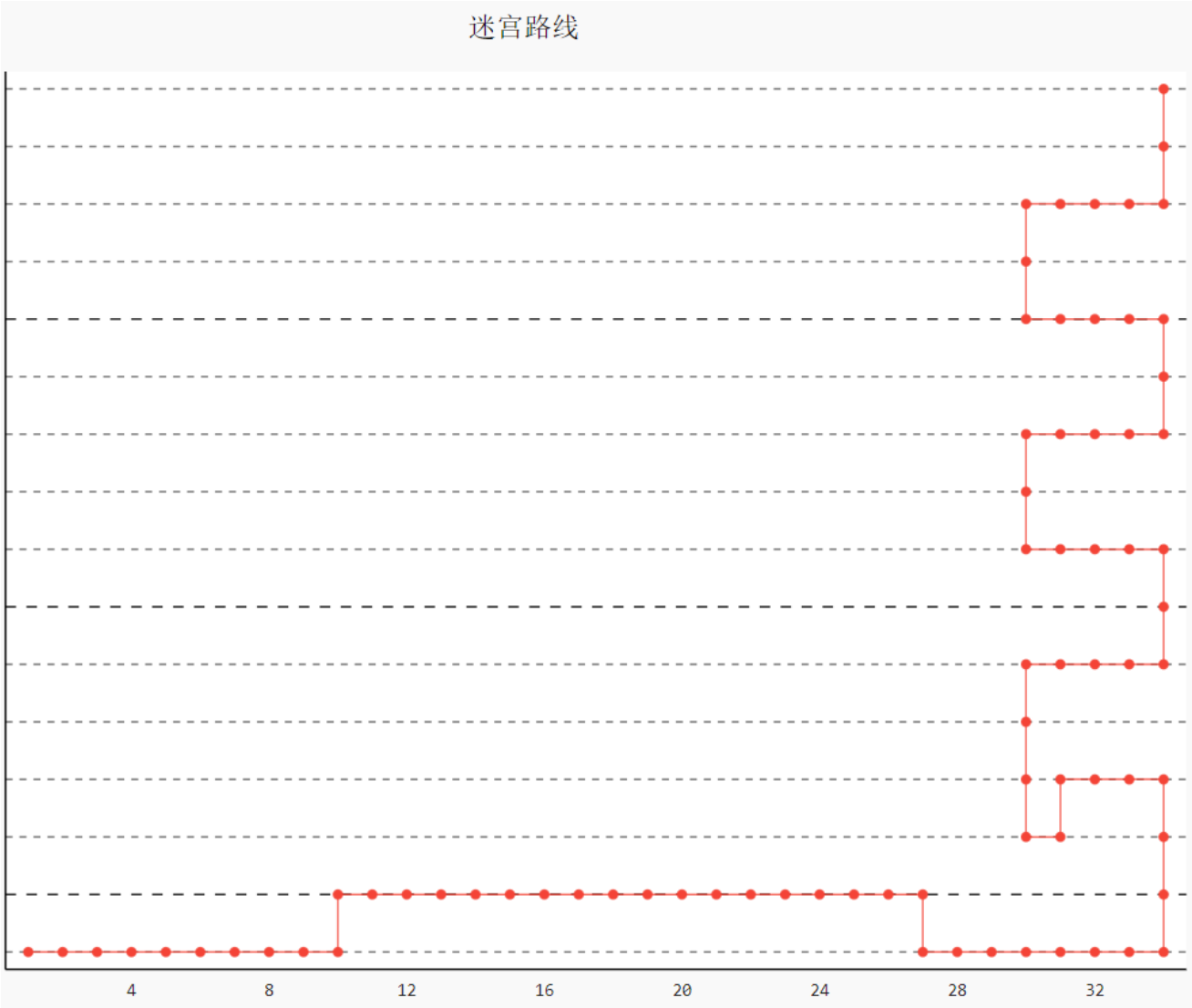
1. 实现迭代加深搜索时，一开始是结点从队列取出扩展时，才将结点加入已访问集，然而检测结果并不能得到最优路径，为了进一步优化，改成结点进队列前就将结点加入已访问集，同时根据DFS特点，当DFS扩展找不到终点时，这时需要进行回溯，而回溯的过程需要将其探索的点从已访问集中取出，这样探索其他方向时也可以经过这些点，从而能够找到最优路径。
2. 在实现双向搜索时，如果两个队列往自己的方向顺序优先扩展，最后交集往往找不到最优路径，可以进行改进，每次都是一层层遍历扩展，从而避免个别结点扩展花费的路径代价，最后保证路径的最优性。

（5）实验结果及分析

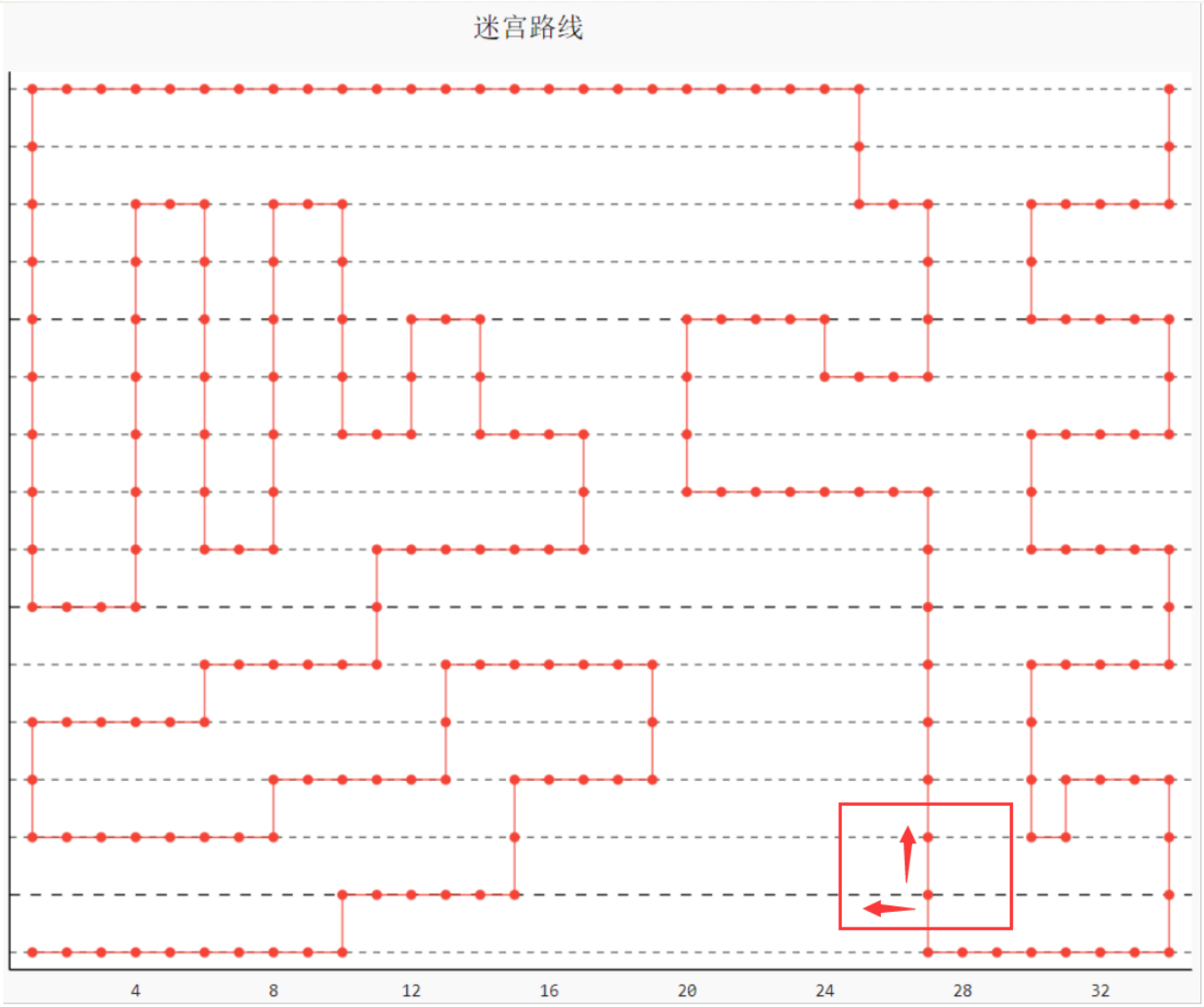
1）实验结果展示

1. DFS：DFS不具备最优性，探索的路径取决于运动方向选择的顺序。（设移动一次为一步）

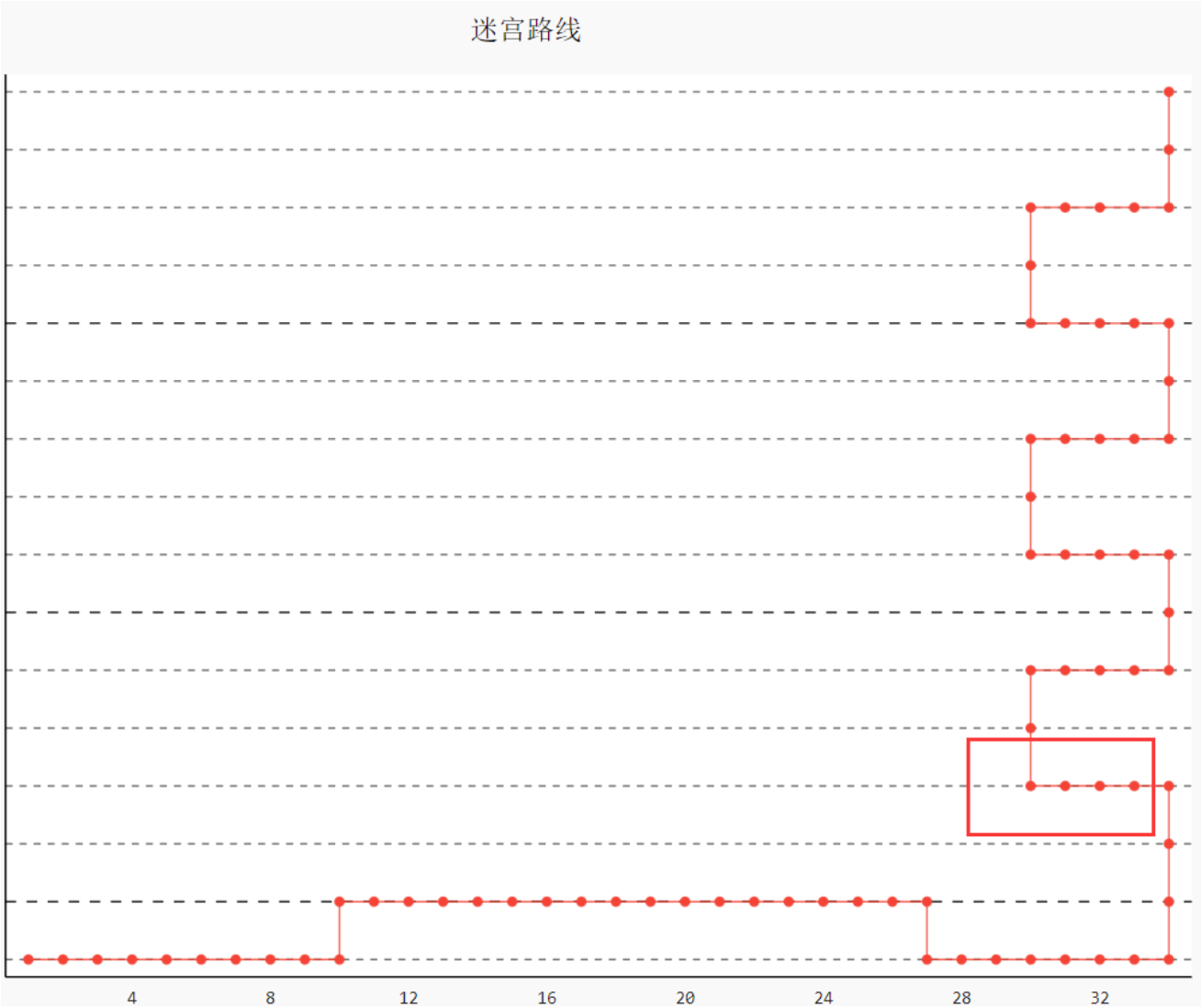
- 运动顺序：下、左、上、右。 路径代价为76。因为一开始结点探索下面结点，因此往下走，一直探索，最终找到终点。



- 运动顺序：下、右、上、左，路径代价为224。由下图可知，在图示标注处，与上图比较，此时在该点可以一直往左走，以更小的路径代价走到终点，但是由于运动方向选择先选择上，再到左，因此往上进行探索，最终花费更多的代价找到终点。



- 运动顺序：右、下、左、上，路径代价：74。跟上上图比较，在图示标注处，由于先往右搜索，因此避免往下搜索走少了2步。

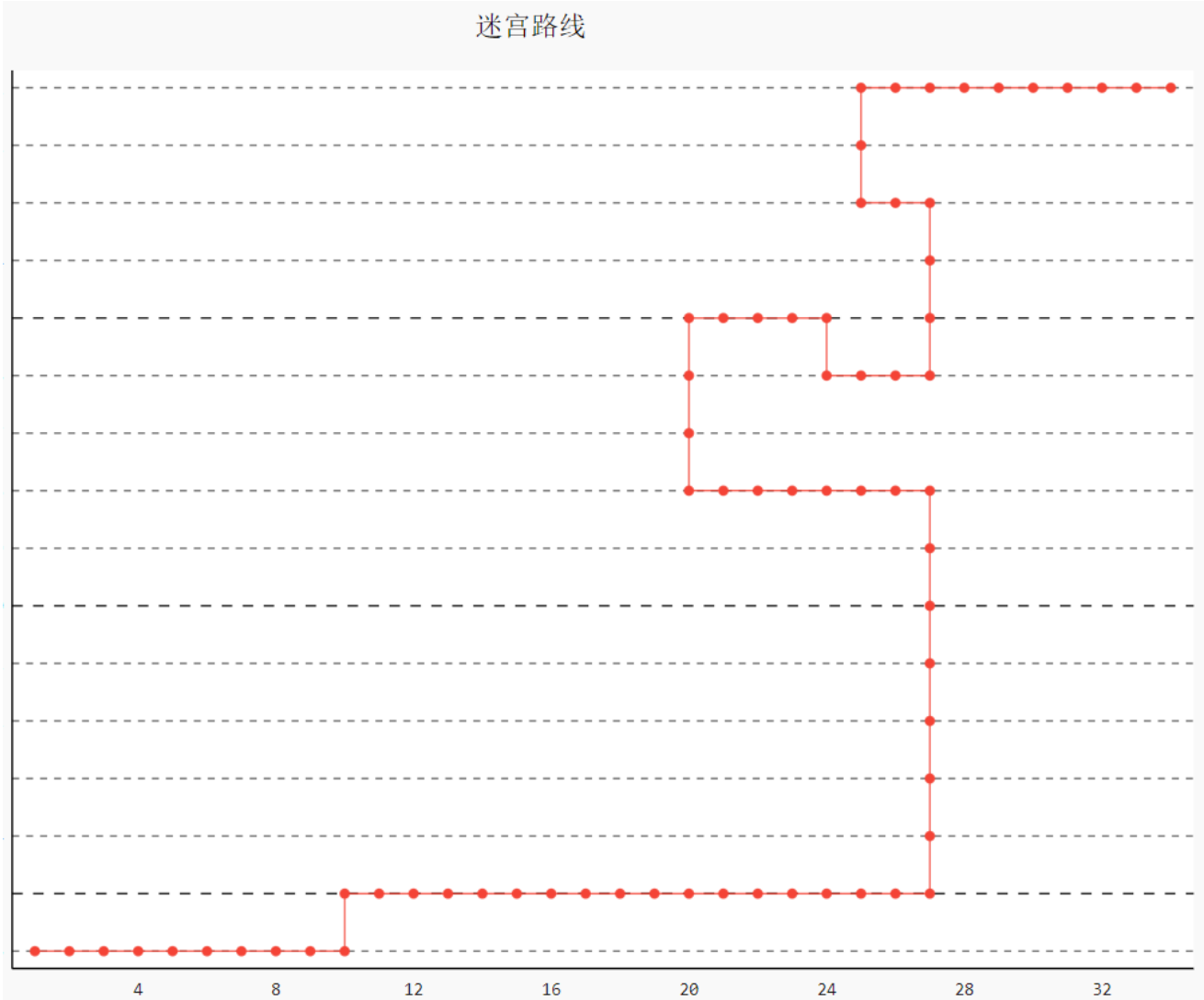


- 运动顺序：左、下、右、上，路径代价：152。

综上：DFS采取深度优先探索，跟方向选择有关，一旦选择一个方向，只要可以继续探索，都会一直走下去，直到找到终点，因此会错误一些最优路径，因此不具备最优性。

2. BFS、UCS、迭代加深搜索均具备最优性，因此其路径图是一致的，路径代价为68。

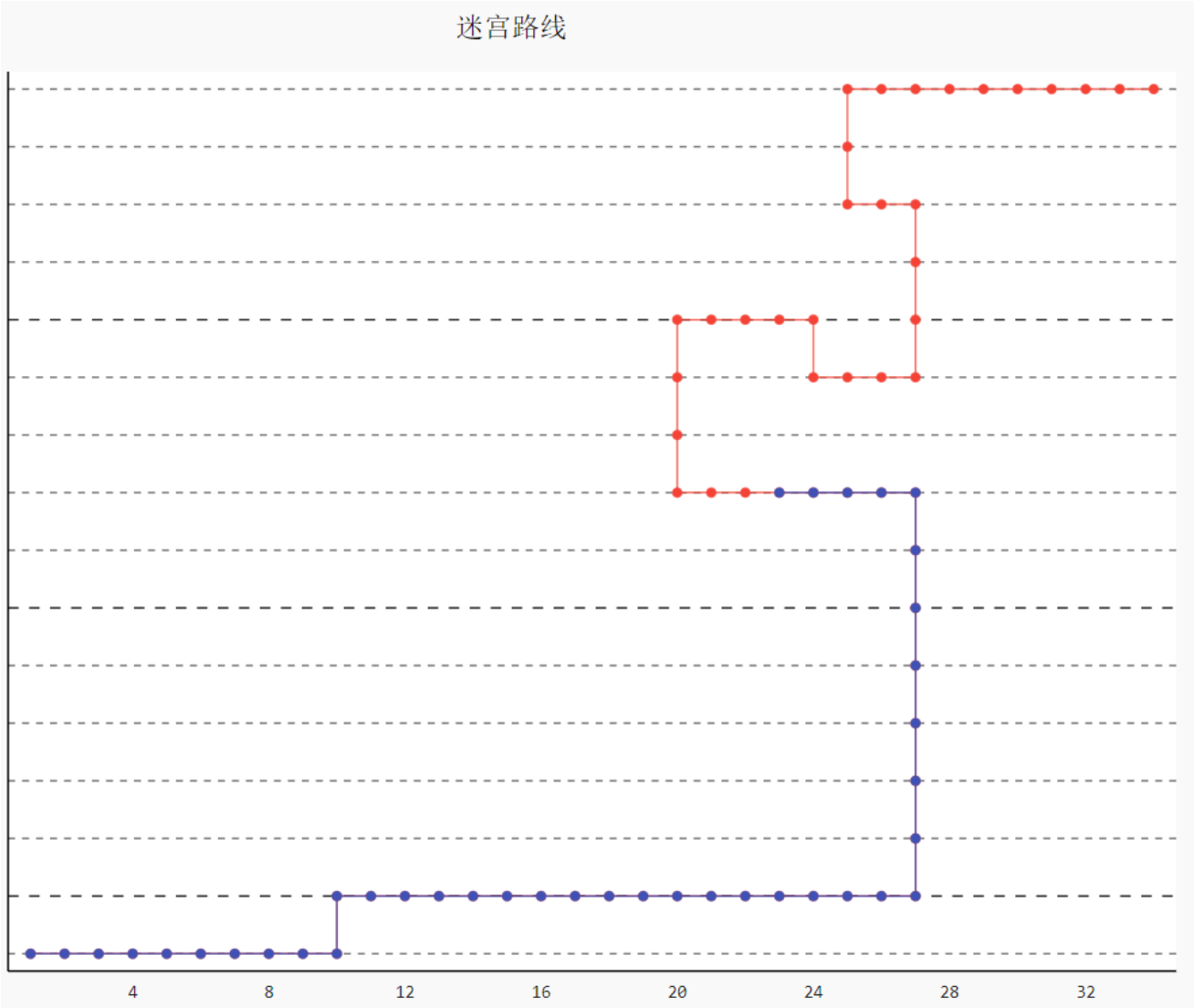
- BFS采取宽度优先策略，每次对结点进行扩展时，四个方向的相邻点作为一层，一层层进行扩展，并且访问过的点不可再访问，那么当路线a经过A点时，如果之后有路线b再经过A点，此时路线b一定比a长，可以保证最后找到终点的路径就是最短的。
- UCS基于BFS，每次在队列中选择路径代价最小的点进行扩展，保证是最小的路径优先扩展，并且探测的点加入队列前，会检查队列是否有该点，如果比较探测的点有更小的路径代价，则进行替换，从而保证最短路径的更新，在本题，由于每一步相当于代价1，因此实际上跟BFS是相同的，具备最优性。
- 迭代加深搜索在DFS基础上加入了层数限制，避免DFS过度往一个方向花费更多的代价，每次扩展四个方向相当于增加一层，当需要扩展深度时，证明之前的层数是没有解的，那么如果有解也只能在新扩展的一层，此时就算DFS首先找到的一个终点，在本题相邻的路径基础上，其路径的代价都是一样，因此是最优路径，具备最优性。



3. 双向搜索

- 如果是采取深度优先策略，那么两个队列分别按照指定的方向顺序优先扩展，最后交集的路径往往不是最优的。
- 如果是采取宽度优先策略，两端每次都按照一层层的方式扩展，确保每一层的扩展的路径都是最短的，避免结点深度扩展，最后交集就是最优路径。

由图可知，前端扩展30结点、后端扩展38结点。



2）评测指标展示

- 五种搜索方式的比较。

指标	BFS	DFS	UCS	迭代加深	双向搜索
最优性	有	有	有	有	有
完备性	有	有	有	有	有
时间复杂度	0.0028	0.00085	0.0035	0.105	0.0021
空间复杂度	4^68	74*4	4^68	68*4	4^68

最优性上面已做分析，空间复杂度中，一个结点的存储空间设置为1。

1. BFS：由于运动方向只有4个，迷宫是有限，因此是可以遍历所有直到找到解。同时由于需要一层层遍历，因此需要花费一定的时间，同时存储空间较大。

2. DFS：如果没有添加访问检测，则DFS容易进入探索死循环，添加访问检测，可以保证找到解，由于可取的路径比较多，因此深度优先探索可以比较迅速找到解，不需要探索其他更多的方向，时间是最短的，由于只需要存储一条路径的空间，比较省空间。
3. UCS：在本题中，每一步代价为1是正数，可以确保一定有解，同时等价于BFS，因此时间和空间花费跟BFS接近。
4. 迭代加深：由于是一层层探索，设置层数限制，并且运动方向个数是有限的，因此确保能找到解，同时由于是一层层迭代，需要直到68层才找到解，因此时间花费较多，基于DFS原理，存储一条路径长度，节省更多空间。
5. 双向搜索：由于从两端出发，一层层扩展，最终一定相遇，具备完备性，同时虽然对每个队列探索的路径减少一半，但是由于需要两倍的存储成本和探索成本，因此实际的时间和空间复杂度并没有减少，接近BFS。

拓展思考：分析各种算法的优缺点

1. BFS

- 优点：
    1. 可以找到最优解。
    2. 无论是否设置环检测，只要问题有解，都可以找到解。
    3. 使用队列可以避免DFS栈溢出的问题。
- 缺点：
    1. 内存消耗比较大，需要用比较多的空间存储探索的结点。
    2. 如果目标结点距离开始结点比较远，那么探索过程会产生很多没有用的结点，探索的效率比较低。
- 适用场景：适用于探索子节点个数不多，并且层数不深的树

2. DFS

- 优点：
    1. 节省存储空间，不需要存储探索过的所有结点。
    2. 如果目标点在开始探索的方向上，则很快能够找到解。
- 缺点：
    1. 容易陷入往某个方向不断搜索，找不到最优路径。
    2. 没有设置环检测的条件下，容易陷入无限循环搜索，不具备完备性。
    3. 递归次数过多时，存储的栈容易溢出。
    4. 如果探索的路径比最优路径大很多，则时间复杂度比BFS高很多。
- 适用场景：适用于深度比较深，结点数比较多的树，此时DFS存储空间具有比较大的优势。

3. 一致代价

- 优点：
    1. 在宽度优先的基础上每次选择代价最小的点进行扩展，提高搜索效率，避免探索更多无用结点。
    2. 确保能够找到最优解，同时具有完备性。
- 缺点：
    1. 如果路径代价为负数，则此时无法确保能够找到解。

2. 如果各个代价相同，一致代价退化为BFS，需要更多的存储空间存储结点。
3. 跟BFS相比，BFS探索到解后就终止，而一致代价会检查目标深度所有结点看谁的代价最小，在此情况下，一致代价在同样深度下做了更多无意义的工作。

- 适用场景：深度较浅，同时子结点之间的代价有差异的树。

4. 深度受限

- 优点：
    1. 避免像DFS结点往一个方向无限扩展。
    2. 节省存储空间。
    3. 避免搜索太多无用结点。
- 缺点：
    1. 难以确定指定的深度，如果解的深度大于指定的深度，则无法找到解。
    2. 不具备最优性和完备性。
- 适用场景：能够确定目标结点层数的情况。

5. 迭代加深

- 优点：
    1. 经过步数迭代增加，最终能找到最优解。
    2. 具备DFS节省存储空间的优点。
    3. 容易通过剪枝减少时间开销。
- 缺点：如果目标结点的层数很深，需要进行很多次迭代，从而提高了时间复杂度，需要剪枝减少时间开销。
- 适用场景：深度不深，结点数较多，BFS无法很好解决的情况。

6. 双向探索

- 优点：
    1. 节省存储空间和时间开销。
    2. 如果是采用BFS策略和每一步的代价相同，有限子节点时，具备完备性和最优性。
- 缺点：如果采用DFS策略，不一定能找到最优解。
- 适用场景：搜索时间和空间复杂度比较高，需要降低空间和时间开销的情况。