中山大学数据科学与计算机学院

计算机科学与技术专业-人工智能

本科生实验报告

（2018-2019 学年秋季学期）

课程名称：**Artificial Intelligence**

教学班级	计科 2 班	专业（方向）	计算机科学与技术
学号	16337341	姓名	朱志儒

实验题目

无信息搜索

实验内容

· 算法原理

1) 深度优先搜索

对于无向连通图，首先访问图中某一个起始顶点 v，然后由 v 出发，访问与 v 相邻且未被访问的任一顶点 w_1，再访问与 w_1 相邻且未被访问的任一顶点 w_2，重复该过程，当不能再继续向邻接顶点访问时，依次回退到最近被访问的顶点，如果它还有邻接的未被访问的顶点，则访问该顶点并重复上述过程，直至图中所有的顶点均被访问过为止。

2) 宽度优先搜索

对于无向连通图，从图中的某个顶点 v 出发，访问 v 后，依次访问 v 的各个未被访问过的邻接点 $w_1, w_2, \ldots$，然后顺序访问 w_1 的各个未被访问的邻接点，w_2 的各个未被访问过的邻接点，直至连通图中所有的顶点都被访问过为止。

3) 一致代价搜索

一致代价搜索是宽度优先搜索的改进版，与宽度优先搜索不同的是它维持一个优先队列 ，每次出队列的顶点满足根顶点到该顶点的距离最小，如果根顶点到队列中所有的顶点的距离相等，则一致代价搜索退化成宽度优先搜索。

4) 迭代加深搜索

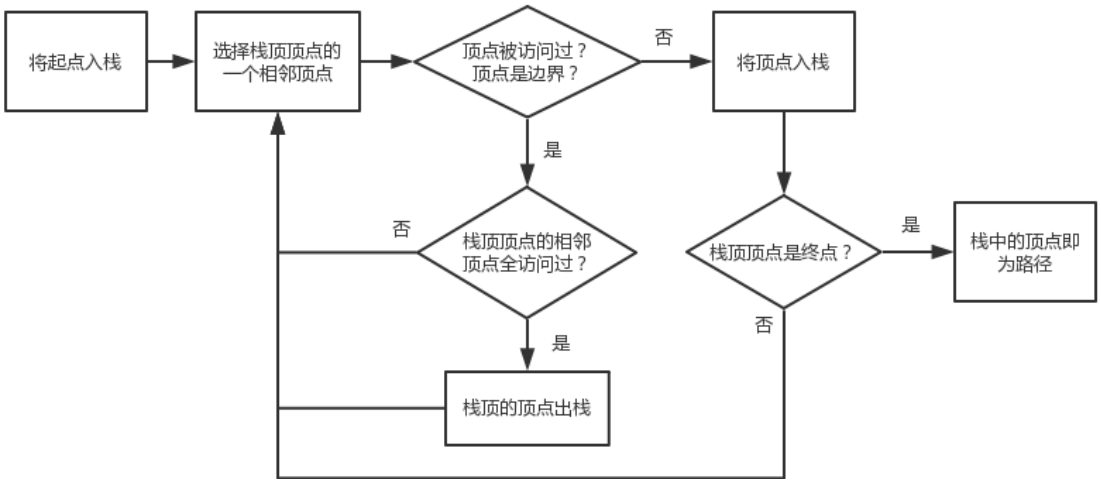
迭代加深搜索是深度优先搜索的改进版，它限制深度优先搜索的递归层数。

基本步骤：

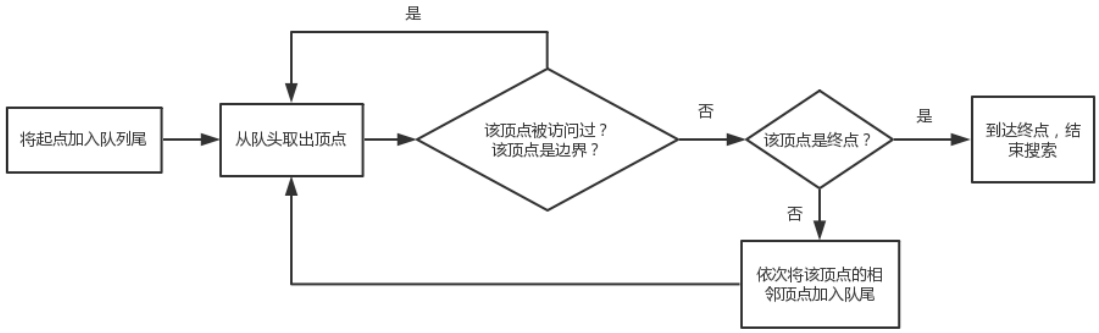
1. 设置一个固定的深度 depth=1，即只搜索初始状态
2. DFS 进行搜索，限制层数为 depth，如果找到目标顶点，则结束搜索，如果没有找到目标顶点，则继续第 3 步
3. 如果第 2 步没有找到目标顶点并且存在顶点未被访问则 depth 加 1，如果第 2 步没有找到目标顶点并且图中所有顶点都被访问过则表示没有答案，结束搜索。

· **流程图**

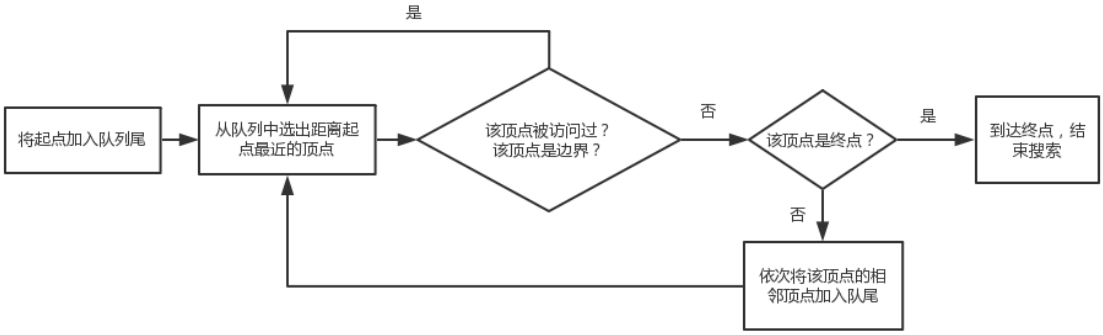
1) **深度优先搜索**



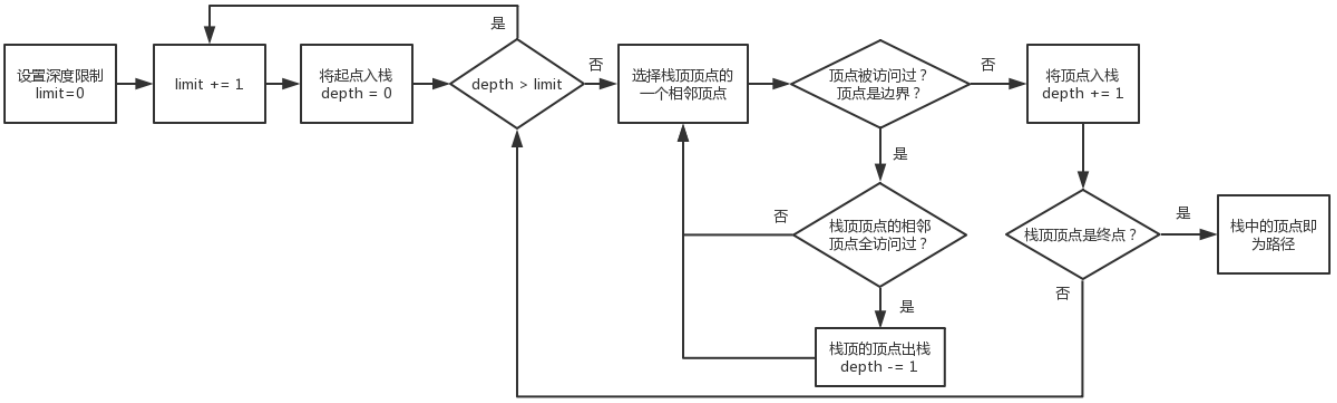
2) **宽度优先搜索**



3) **一致代价搜索**



4) 迭代加深搜索



· 关键代码

1) 深度优先搜索

```
def search(self, pos, end, pre):
    if self.maze[pos[0]][pos[1]] == '%':
        # 遇到迷宫中的墙壁
        return
    if pos[0] == end[0] and pos[1] == end[1]:
        # 到达目的地
        self.path[pos[0]][pos[1]] = (pre[0], pre[1], pre[2] + 1)
        return
    if self.visited[pos[0]][pos[1]] == 1:
        # 已经访问过该顶点
        return
    # 记录父顶点
    self.path[pos[0]][pos[1]] = (pre[0], pre[1], pre[2] + 1)
    # 将顶点标记为已访问
    self.visited[pos[0]][pos[1]] = 1
    # 递归访问邻接顶点
    if pos[1] - 1 >= 0:
        self.search((pos[0], pos[1] - 1, pos[2] + 1), end, pos)
    if pos[0] + 1 < 18:
        self.search((pos[0] + 1, pos[1], pos[2] + 1), end, pos)
    if pos[0] - 1 >= 0:
        self.search((pos[0] - 1, pos[1], pos[2] + 1), end, pos)
    if pos[1] + 1 < 36:
        self.search((pos[0], pos[1] + 1, pos[2] + 1), end, pos)
```

2) 宽度优先搜索

检测顶点是否可以访问：

```
    def is_available(self, pos):
        if self.maze[pos[0]][pos[1]] == '%':
            # 遇到迷宫中的墙壁
            return False
        if self.visited[pos[0]][pos[1]] == 1:
            # 已经访问过该顶点
            return False
        return True
```

宽度优先搜索：

```
    def search(self, start, end):
        start = (start[0], start[1], 0)
        self.path[start[0]][start[1]] = start
        # 声明边界队列
        q = queue.Queue()
        q.put(start)
        while not q.empty():
            pos = q.get()
            if pos[0] == end[0] and pos[1] == end[1]:
                # 到达目的地
                break
            # 将顶点标记为已访问
            self.visited[pos[0]][pos[1]] = 1
            # 依次访问邻接顶点
            if pos[0] - 1 >= 0:
                if self.is_available((pos[0] - 1, pos[1])):
                    # 如果该顶点可以访问，则记录父顶点，并将其加入边界队列
                    self.path[pos[0] - 1][pos[1]] = (pos[0], pos[1],
pos[2] + 1)
                    q.put((pos[0] - 1, pos[1], pos[2] + 1))
            if pos[0] + 1 < 18:
                if self.is_available((pos[0] + 1, pos[1])):
                    # 如果该顶点可以访问，则记录父顶点，并将其加入边界队列
                    self.path[pos[0] + 1][pos[1]] = (pos[0], pos[1],
pos[2] + 1)
```

```
                    q.put((pos[0] + 1, pos[1], pos[2] + 1))
            if pos[1] - 1 >= 0:
                if self.is_available((pos[0], pos[1] - 1)):
                    # 如果该顶点可以访问，则记录父顶点，并将其加入边界队列
                    self.path[pos[0]][pos[1] - 1] = (pos[0], pos[1],
pos[2] + 1)
                    q.put((pos[0], pos[1] - 1, pos[2] + 1))
            if pos[1] + 1 < 36:
                if self.is_available((pos[0], pos[1] + 1)):
                    # 如果该顶点可以访问，则记录父顶点，并将其加入边界队列
                    self.path[pos[0]][pos[1] + 1] = (pos[0], pos[1],
pos[2] + 1)
                    q.put((pos[0], pos[1] + 1, pos[2] + 1))
        path = []
        pos = end
        while pos != start:
            # 得到从起始顶点到目的地的路径
            path.append((pos[0], pos[1], self.path[pos[0]][pos[1]][2]))
            pos = self.path[pos[0]][pos[1]]
        for item in path:
            self.maze[item[0]][item[1]] = str(item[2])
        # 绘制从起始顶点到目的地的路径
        for line in self.maze:
            for i in line:
                if i != '%':
                    print("{:^3s}".format(i), end='')
                else:
                    print("{:%^3s}".format(i), end='')
            print()
        print()
```

3) 一致代价搜索

```
    def search(self, start, end):
        begin = (start[0], start[1], 0)
        self.path[start[0]][start[1]] = begin
        # 声明边界队列
        q = [begin]
        while len(q) != 0:
```

```
            # 将距起始顶点最近的顶点从队列中弹出
            index = 0
            minn = q[0][2]
            for i in range(len(q)):
                if minn > q[i][2]:
                    minn = q[i][2]
                    index = i
            pos = q[index]
            del q[index]
            if pos[0] == end[0] and pos[1] == end[1]:
                # 到达目的地
                break
            # 将顶点标记为已访问
            self.visited[pos[0]][pos[1]] = 1
            # 依次访问邻接顶点
            if pos[0] - 1 >= 0:
                if self.is_available((pos[0] - 1, pos[1]), end):
                    # 如果该顶点可以访问，则记录父顶点，并将其加入边界队列
                    self.path[pos[0] - 1][pos[1]] = (pos[0], pos[1],
pos[2] + 1)
                    q.append((pos[0] - 1, pos[1], pos[2] + 1))
            if pos[0] + 1 < 18:
                if self.is_available((pos[0] + 1, pos[1]), end):
                    # 如果该顶点可以访问，则记录父顶点，并将其加入边界队列
                    self.path[pos[0] + 1][pos[1]] = (pos[0], pos[1],
pos[2] + 1)
                    q.append((pos[0] + 1, pos[1], pos[2] + 1))
            if pos[1] - 1 >= 0:
                if self.is_available((pos[0], pos[1] - 1), end):
                    # 如果该顶点可以访问，则记录父顶点，并将其加入边界队列
                    self.path[pos[0]][pos[1] - 1] = (pos[0], pos[1],
pos[2] + 1)
                    q.append((pos[0], pos[1] - 1, pos[2] + 1))
            if pos[1] + 1 < 36:
                if self.is_available((pos[0], pos[1] + 1), end):
                    # 如果该顶点可以访问，则记录父顶点，并将其加入边界队列
                    self.path[pos[0]][pos[1] + 1] = (pos[0], pos[1],
pos[2] + 1)
```

```
                q.append((pos[0], pos[1] + 1, pos[2] + 1))
        path = []
        pos = end
        while pos != begin:
            # 得到从起始顶点到目的地的路径
            path.append((pos[0], pos[1], self.path[pos[0]][pos[1]][2]))
            pos = self.path[pos[0]][pos[1]]
        for item in path:
            self.maze[item[0]][item[1]] = str(item[2])
        # 绘制从起始顶点到目的地的路径
        for line in self.maze:
            for i in line:
                if i != '%':
                    print("{:^3s}".format(i), end='')
                else:
                    print("{:%^3s}".format(i), end='')
            print()
        print()
```

4) 迭代加深搜索

深度受限搜索

```
    def search(self, pos, end, pre, limit):
        # 深度受限搜索
        if pos[2] > limit:
            # 搜索深度达到最大值，结束搜索
            return
        if self.maze[pos[0]][pos[1]] == '%':
            # 遇到迷宫中的墙壁
            return
        if pos[0] == end[0] and pos[1] == end[1]:
            # 到达目的地，记录父顶点
            self.path[pos[0]][pos[1]] = (pre[0], pre[1], pre[2] + 1)
            self.flag = True
            return
        if self.visited[pos[0]][pos[1]] == 1:
            # 已经访问过该顶点
            return
```

```
        # 记录父顶点
        self.path[pos[0]][pos[1]] = (pre[0], pre[1], pre[2] + 1)
        # 将顶点标记为已访问
        self.visited[pos[0]][pos[1]] = 1
        # 递归访问邻接顶点
        if pos[1] - 1 >= 0 and not self.flag:
            self.search((pos[0], pos[1] - 1, pos[2] + 1), end, pos,
limit)
        if pos[0] + 1 < 18 and not self.flag:
            self.search((pos[0] + 1, pos[1], pos[2] + 1), end, pos,
limit)
        if pos[0] - 1 >= 0 and not self.flag:
            self.search((pos[0] - 1, pos[1], pos[2] + 1), end, pos,
limit)
        if pos[1] + 1 < 36 and not self.flag:
            self.search((pos[0], pos[1] + 1, pos[2] + 1), end, pos,
limit)
        # 回溯时将顶点设为未被访问
        self.visited[pos[0]][pos[1]] = 0
```

迭代加深搜索

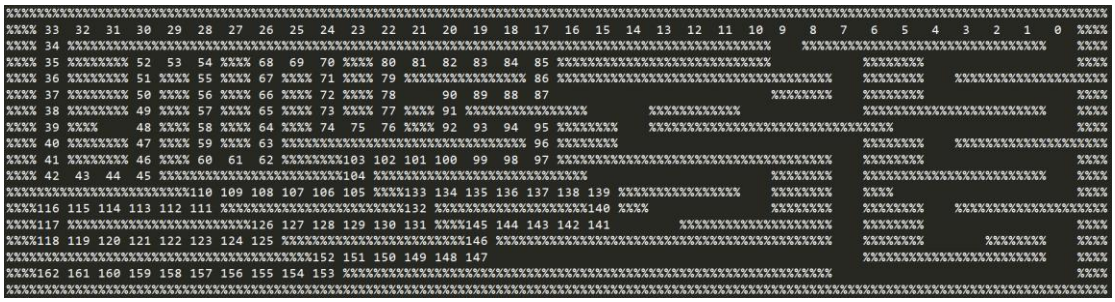
```
    def find_path(self, start, end):
        start = (start[0], start[1], 0)
        cutoff = 0
        while not self.flag:
            # 未到达目的地，则增加搜索深度
            cutoff += 1
            # 清除上次搜索的记录
            self.path = [[0 for i in range(36)] for j in range(18)]
            # 进行深度受限搜索
            self.search(start, end, start, cutoff)
        self.path[start[0]][start[1]] = start
        path = []
        pos = end
        while pos != start:
            # 得到从起始顶点到目的地的路径
            path.append((pos[0], pos[1], self.path[pos[0]][pos[1]][2]))
            pos = self.path[pos[0]][pos[1]]
        for item in path:
```

```
            self.maze[item[0]][item[1]] = str(item[2])
        # 绘制从起始顶点到目的地的路径
        for line in self.maze:
            for i in line:
                if i != '%':
                    print("{:^3s}".format(i), end='')
                else:
                    print("{:%^3s}".format(i), end='')
            print()
        print()
```

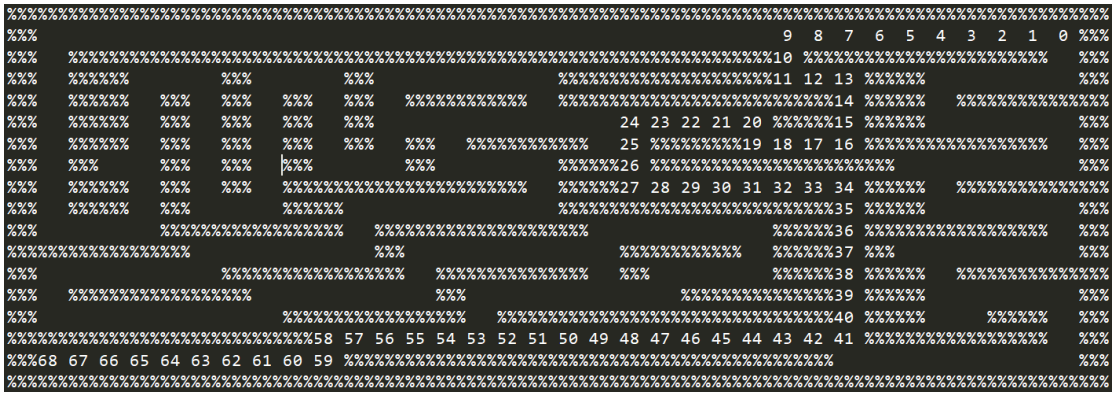
实验结果及分析

· 实验结果展示

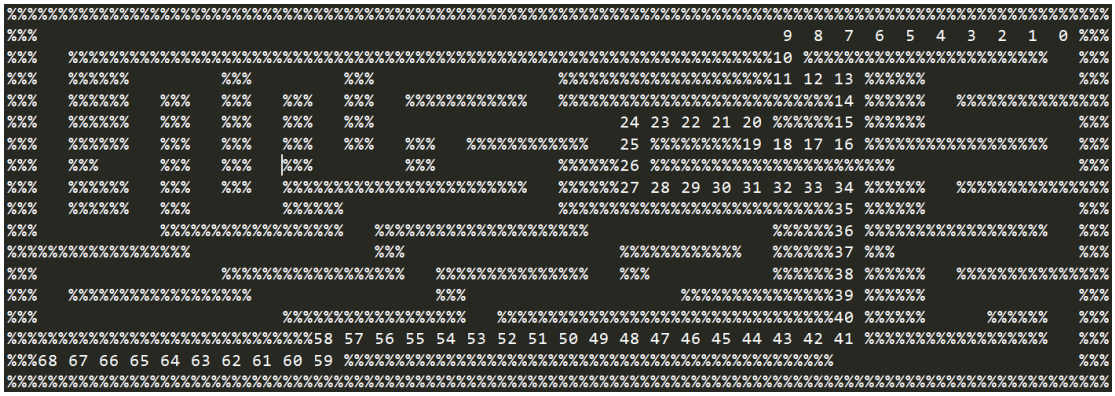
1) 深度优先搜索



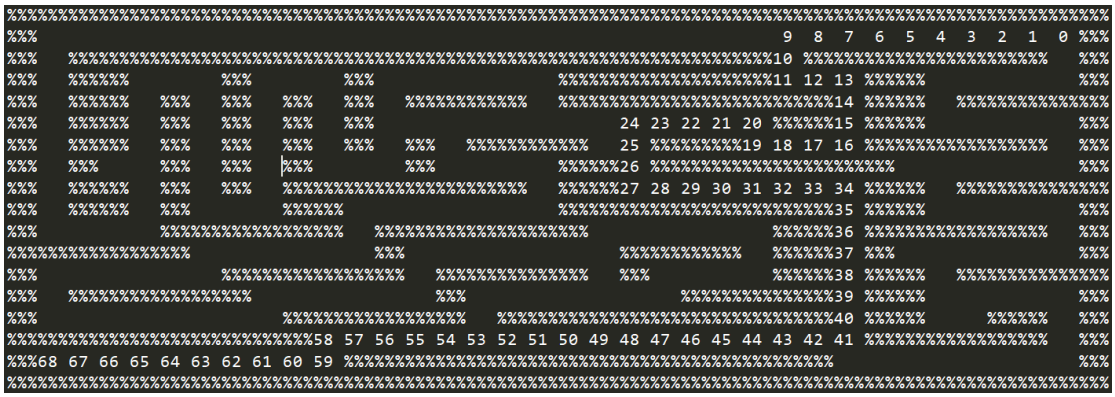
2) 宽度优先搜索



3) 一致代价搜索



4) 迭代加深搜索



时间对比：

```
DFS时间:  0.003988981246948242
BFS时间:  0.011967420578002930
UCS时间:  0.014595737457275390
IDS时间:  0.195989847183227540
```

· 评测指标展示

1) 深度优先搜索

DFS 具有完备性，当问题有解时，DFS 可以保证找到解；

DFS 不具备最优性，从本次实验结果可以看出 DFS 并未找到最短路径；

DFS 的时间复杂度为 $O(b^m)$，即指数级时间复杂度，其中 m 指图中最长路径的长度，b 指每个节点的子节点的最大数目；

DFS 的空间复杂度为 O(bm)，即线性空间复杂度。

2) 宽度优先搜索

BFS 具有完备性，当问题有解时，DFS 可以保证找到解；

BFS 也具有最优性，因为图中顶点到相邻顶点的距离均相等，所以 BFS 探索出到达目的地的路径就是最短路径；

由于本次实验是在探索顶点时判断该顶点是否为目的地，所以时间复杂度为 $O(b^{d+1})$，其中 b 指每个节点的子节点的最大数目，d 指初始位置到目标位置最短路径的长度；

空间复杂度为 $O(b^{d+1})$。

3) 一致代价搜索

一致代价搜索具备完备性，当问题有解时，UCS 可以保证找到解；

一致代价搜索也具备最优性，因为每次探索的顶点满足从初始位置到该顶点的距离最短；

与宽度优先搜索相同，一致代价搜索的时间复杂度为 $O(b^{d+1})$，其中 b 指每个节点的子节点的最大数目，d 指初始位置到目标位置最短路径的长度；

空间复杂度为 $O(b^{c/(s+1)})$，其中 c 指最短路径的长度，s 指顶点到相邻顶点的最短距离。

4) 迭代加深搜索

迭代加深搜索具备完备性，当问题有解时，UCS 可以保证找到解；

由于本次实验图中顶点到相邻顶点的距离均相等，所以迭代加深搜索具备最优性；

时间复杂度为 $O(b^{d})$，其中 b 指每个节点的子节点的最大数目，d 指初始位置到目标位置最短路径的长度；

空间复杂度与深度优先搜索相同均为线性空间复杂复杂度为 O(bd)。

思考题

这些策略的优缺点是什么？它们分别适用于怎样的场景？

1) 深度优先搜索

优点：线性空间复杂度，在一定条件下不必遍历所有分支而找到目标节点；

缺点：不一定能得到最短路径；

适用场景：只要判断是否能到达目的节点而不必得到最短路径的情况。

2) 宽度优先搜索

优点：在每步代价相同的情况下，可以得到从初始节点到目的节点的最短路径；

缺点：指数级空间复杂度，十分消耗内存；

适用场景：节点的子节点数目不多，图的层次不深的情况。

3) 一致代价搜索

优点：可得到从初始节点到目的节点的最短路径，比宽度优先搜索的空间复杂度底；

缺点：还是指数级空间复杂度，消耗内存，时间复杂度高于宽度优先搜索；

适用场景：需要找到最短路径的场景。

4) 迭代加深搜索

优点：可避免陷入无限的分支，可找到深度最浅的目的节点，线性空间复杂度；

缺点：时间复杂度较高，实际运行时间高于上述三种方法；

适用场景：适用于无限深度的图。