《人工智能》课程系列

基本的搜索技术*

武汉纺织大学数学与计算机学院

杜小勤

2018/08/20

Contents

*本系列文档属于讲义性质，仅用于学习目的。Last updated on: February 23, 2020。

1 概述

在设计搜索算法之前，需要定义算法的性能标准：

- 完备性 (Completeness)

  当问题有解时，算法能够保证找到解；

- 最优性 (Optimality)

  当问题有解时，算法能够保证找到最优解；

- 时间复杂度 (Time Complexity)

- 空间复杂度 (Space Complexity)

在计算机科学中，度量时间与空间复杂度的方式是使用状态空间图的大小。例如 $|V|$ 和 $|E|$，其中 V 是图中顶点 (节点) 的集合，E 是图中边的集合，$|X|$ 表示 X 中元素的个数。

在人工智能领域，状态空间图大多由初始状态、动作和转移模型隐式地表示，并且大多是无限的。在这种情况下，复杂度通常由三个量来表达：分支因子 b、目标节点的最浅深度 d(从根节点到目标节点的最浅步数)、所有路径中的最大长度 m。时间复杂度常常由搜索过程中产生的节点数目来度量，而空间复杂度则由内存中存储的最多节点数目来度量。

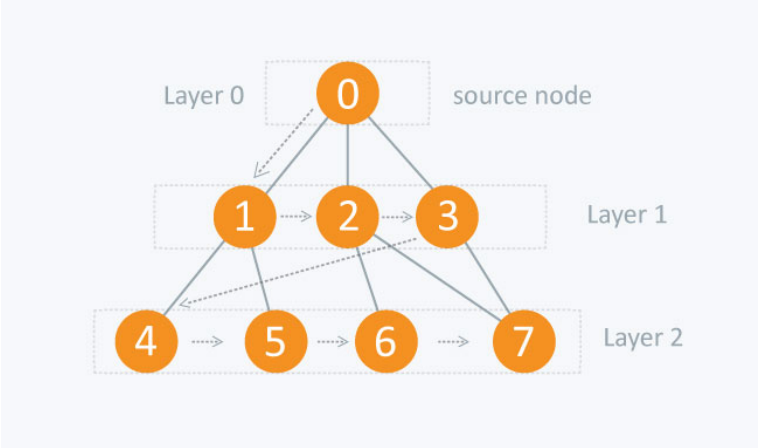

图 2-1: 宽度优先搜索的示意图

考虑到算法描述的方便性及可理解性，将根据实际情况交替使用这 2 种度量方式。

下面首先讨论几种无信息搜索 (盲目搜索) 算法。这类算法仅仅使用了问题提供的状态信息，没有使用任何其它信息，它要做的主要工作是，遍历每个节点并判断后继节点是否是目标节点。无信息搜索算法以节点访问的顺序进行分类，主要可以分为宽度优先搜索与深度优先搜索。有信息的搜索算法被称为启发式搜索算法，它有效地利用了一些启发信息，优先访问那些更接近目标的节点。这类算法也将在后面进行讨论。

2 宽度优先搜索

2.1 基本算法

宽度优先搜索 (Breadth-First Search) 算法是一种遍历或搜索树/图数据结构的简单算法。它先访问根节点，接着访问根节点的所有子节点，然后再访问这些子节点的后继节点，依此类推。这种算法的工作方式是，从根节点开始，由里向外，逐层遍历所有节点——它每次总是访问深度最浅的节点。宽度优先搜索的示意图如图2-1所示。

仔细分析宽度优先搜索算法的工作方式，可知，深度较浅的节点 (老节点) 总是早于深度较深的节点 (新节点) 进行访问。因此，可以使用队列作为管理节点的数据结构：老节点从队列头部出列，新节点从队列尾部入列。算法如下：

```
Input:
    A graph G and a vertex v of G
    came_from is a DICT, initialized with {}
Output:
    return: GOAL or None
    came_from is changed
def BFS(G, v, came_from):
    frontier = Queue()
    frontier.enqueue(v)
    came_from[v] = None
    while not frontier.is_empty():
        v = frontier.dequeue()
        if v is not labeled as discovered:
            if v is a goal:
                return v
            else:
                label v as discovered
                for all edges from v to w in G.adjacentEdges(v):
                    if w is not labeled as discovered:
                        frontier.enqueue(w)
                        came_from[w] = v
    return None
```

分析上述算法，易知，在遍历图时，对图中每个顶点至多访问一次。遍历图的过程实质上是对每个顶点查找邻接点的过程，其耗费的时间取决于 G 的存储结构。如果使用二维数组表示图的邻接矩阵，那么在最坏情况下，时间复杂度为 $O(n^2)$，n 为顶点数；如果使用邻接表 (Adjacency List) 作为图的存储结构，那么在最坏情况下，时间复杂度为 $O(n+e)$，e 为无向图的边数或有向图的弧数。注意，标记顶点的访问标志是重要的一步，否则，图的遍历可能形成无限循环。

还可以采用 OPEN-CLOSED 表作为数据结构，实现 BFS 算法[1]：

[1]注意，在算法伪代码中，虽然将已访问节点表 (CLOSED) 定义成列表对象，但是，为便于有效地检查节点是否已经被访问过，可以使用哈希表存放已访问节点。在 Python 中，可以使用字典

```
Input:
    A graph G and a vertex v of G
    came_from is a DICT, initialized with {}
Output:
    return: GOAL or None
    came_from is changed
def BFS(G, v, came_from):
    open = [v]
    closed =[]
    came_from[v] = None
    while open is not empty:
        remove an element from the front of open, call it v
        if v is a goal:
            return v
        else:
            put v on closed
            for all edges from v to w in G.adjacentEdges(v):
                if w is not in closed:
                    put w at the back of open
                    came_from[w] = v
    return None
```

BFS 算法是完备的，原因在于，假设最浅的目标节点位于有限的深度 d 处，那么 BFS 算法在访问完所有比它浅的节点之后，一定能够访问到该目标节点且通过目标测试。在以步数作为代价函数的前提下，BFS 搜索到的目标节点一定是最优的。

下面从另一个角度分析 BFS 算法的时间与空间复杂度。假设目标节点的深度为 d，且搜索树中的每个节点都有 b 个子节点，那么，第 0 层是根节点，1 个节点；第 1 层是 b 个子节点；第 2 层是 b^2 个子节点；...；第 d 层是 b^d 个子节点。在最

对象。

Depth	Nodes	Time	Memory
2	110	.11 milliseconds	107 kilobytes
4	11,110	11 milliseconds	10.6 megabytes
6	10^6	1.1 seconds	1 gigabyte
8	10^8	2 minutes	103 gigabytes
10	10^{10}	3 hours	10 terabytes
12	10^{12}	13 days	1 petabyte
14	10^{14}	3.5 years	99 petabytes
16	10^{16}	350 years	10 exabytes

图 2-2: BFS 的时间与空间需求

坏情况下，如果生成节点时进行目标检测，那么已访问的节点总数为：

$$1 + b + b^2 + b^3 + ... + b^{d-1} \tag{1}$$

或者，如果访问节点时进行目标检测，那么已访问的节点总数为：

$$1 + b + b^2 + b^3 + ... + b^d \tag{2}$$

因此，时间复杂度为 $O(b^d)$。

BFS 算法运行时，需要保存生成的节点，根据搜索树的分支特征，它的个数为 $O(b^d)$，这就是算法的空间复杂度[2]。

我们来看一下图2-2，可以体会到情况到底有多糟糕！图中，设分支因子 $b = 10$，算法以 10^6 节点/秒的速度计算，每个节点所占的空间为 1000 字节[3]。分析图中数据，可以发现：

- 内存需求问题比执行时间更令人无法接受

  例如，深度 $d = 12$ 时，执行时间为 13 天，而内存需求为 $1P$，这是一个很大的容量，一般情况下，难以得到满足。幸运的是，还有其它内存需求较少的搜索算法；

- 时间需求也是一个很严重的问题

  例如，深度 $d = 16$ 时，执行时间为 350 年。一般来讲，指数级别复杂度的搜索问题不能使用这种搜索算法，除非问题的规模很小；

[2]在考虑 came_from 数据结构的情况下，空间复杂度的量级不变。

[3]统计数据中没有计入根节点。

2.2 一致代价搜索

基本的 BFS 算法没有显式地考虑每步的代价，或者说，它缺省地将节点的深度当作代价，在此情况下，它可以找到最优节点，因为它总是优先访问深度最浅的节点。

如果每步的代价不一样，那么基本的 BFS 算法将会失效，因为通常情况下，它优先访问的节点仅仅是离根节点最近，而不一定是代价最低。

实际上，在此情况下，可以显式地为每步引入代价函数或一致代价 (Uniform Cost)，并对基本的 BFS 算法进行修改：不再优先访问深度最浅的节点，而是优先访问当前代价最低的节点，这种算法被称为一致代价搜索 (Uniform Cost Search, UCS) 算法。可以看出，基本的 BFS 算法是一致代价搜索算法的特例。我们引入优先级队列用于存储待访问的节点，改进后的算法如下：

```
Input:
    A graph G and a vertex v of G
    came_from is a DICT, initialized with {}
Output:
    return: GOAL or None
    came_from is changed
def UCS(G, v, came_from):
    frontier = PriorityQueue()
    cost_so_far = {}
    frontier.enqueue(v, 0)
    cost_so_far[v] = 0
    came_from[v] = None
    while not frontier.is_empty():
        v = frontier.dequeue()
        if v is a goal:
            return v
        else:
            for all edges from v to w in G.adjacentEdges(v):
                new_cost = cost_so_far[v] + G.cost(v, w)
```

```
            if w not in cost_so_far or new_cost < cost_so_far[w]:
                cost_so_far[w] = new_cost
                priority = new_cost
                frontier.put(w, priority)
                came_from[w] = v
    return None
```

注：可以把 frontier.put 理解为“如果该项已经存在，则用新的优先级替换；否则，按优先级插入该项”[4]。

除了按路径代价将节点插入优先级队列外，UCS 算法与 BFS 算法还有一个明显不同的地方：在生成节点时，增加了节点新代价是否更小的测试，即如果该节点已经生成了，那么需要测试新代价是否要比老代价好。

显然，UCS 算法是最优的。第 1 个原因是，当 UCS 算法选择某个节点 n 进行访问时，就说明它已经找到了到达节点 n 的最优路径 (如若不然，假设还存在另一条到达 n 的路径，其代价更低，那么 UCS 算法早就对其进行了访问)。第 2 个原因是，由于每步的代价是非负的，随着节点的增加，路径代价绝不会变低。因此，UCS 算法将按节点的最优路径顺序地访问节点。易知，第 1 个被选择访问的目标节点一定是最优解。

UCS 算法对最优路径的步数并不关心，它只关心路径的总代价，它也不会因为节点已经被生成过而拒绝再次生成该节点 (因为要找出最小访问代价)，并且算法每次选择代价最低的节点进行访问。因此，如果存在零代价路径重新导向那些已经访问过的节点，那么整个搜索可能会陷入死循环。在此情况下，UCS 算法就不是完备的。为了避免这个问题，必须要确保每步代价至少是某个小的正值常数 ϵ。此时，UCS 算法才是完备的[5]。

因为 UCS 算法是由路径代价而不是搜索深度来引导的，所以算法复杂度不能简单地用 b 和 d 来表示。设 C^* 为最优解的代价，每步代价至少为 ϵ，那么在最坏情况下，算法的时间和空间复杂度都为 $O(b^{1+\lfloor C^*/\epsilon \rfloor})$。一般而言，$b^{1+\lfloor C^*/\epsilon \rfloor}$ 要比 b^d

[4]在实际实现时，如果明确知道问题有解，可以简单地使用 frontier.enqueue 方法替换 put 方法，即无论存在与否，均按优先级插入该项。这种变形的处理方法，不会影响算法的功能实现。原因在于，即使该项已经存在，新插入项的优先级高，必将优先访问。另外，算法在找到一个目标后会停止运行，即使 frontier 优先级队列里还包含其它节点。算法实现，请参考4.8。

[5]在每步非零代价的情况下，即使 ϵ 很小，它的累加代价还是会慢慢变大。因此，形成死循环的节点不再位于优先级队列的顶端，从而给其它节点的访问创造了机会。

大很多，原因解释如下。

BFS 算法以步数最短作为最优条件，而 UCS 算法以代价最小作为最优条件，最优条件的差异是很大的，这是 UCS 算法要比 BFS 算法复杂的根源之一。前者每次可以访问的节点集由当前节点的子节点个数决定，是一个与分支因子有关的量。而后者可以访问的节点集是由当前节点的子节点及其它节点的当前代价决定的。通常，它要访问的节点个数是一个比分支因子大的量。在目标节点之前，如果搜索树中包括了很多代价较小的节点，那么这些节点都需要一一探索。即使在找到目标节点时，BFS 算法会立即终止[6]，而 UCS 算法还需要检查其它的目标解，因为它的目标是找出最小代价的解[7]。因此，与 BFS 算法相比，UCS 算法需要做更多的工作。

3 深度优先搜索

3.1 基本算法

深度优先搜索 (Depth-First Search, DFS) 也是一种遍历或搜索树/图数据结构的简单算法[8]。它具有天然的递归性质，并且紧密地与回溯 (Backtracking)[9]结合在一起。DFS 算法是对树的先序遍历 (Preorder Traversal) 算法的推广。

图3-3是深度优先搜索的示意图。

DFS 算法有递归与非递归 2 种形式。下面先给出它的递归形式：

```
Input:
    A graph G and a vertex v of G
    came_from is a DICT, initialized with {}
Output:
    return: GOAL or None
    came_from is changed
```

[6]在 BFS 算法下，每步代价都一样，找到了目标，就意味着该目标解一定是最优的。

[7]目标节点的比对，可以在生成节点时进行，也可以在访问节点时进行。显然，此处指的是第 1 种情况。

[8]早在 19 世纪，法国数学家 Charles Pierre Trémaux 就研究了 DFS 方法，用于解决迷宫问题。

[9]回溯意味着，它从某个节点开始，如有可能，会一直穷尽递归地“向前”遍历，一直进行到当前路径上已经没有更多的节点为止，然后原路返回到上一次被访问的节点，再接着遍历它的某个未被访问的子节点。此过程循环往复，直至所有的节点都被访问。

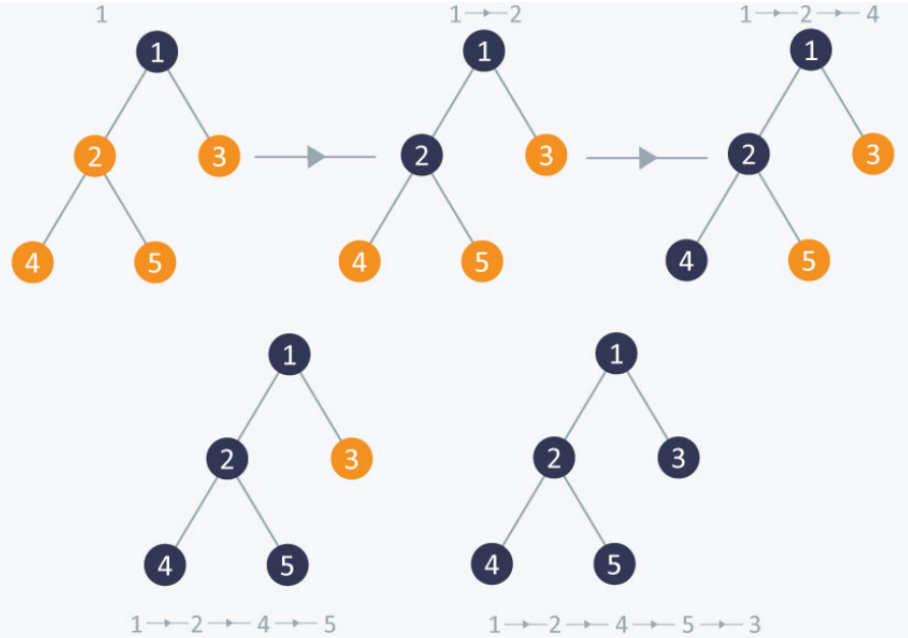

图 3-3: 深度优先搜索的示意图

```
def DFS(G, v, came_from):
    if came_from == {}:
        came_from[v] = None
    if v is a goal:
        return v
    label v as discovered
    for all edges from v to w in G.adjacentEdges(v):
        if vertex w is not labeled as discovered:
            came_from[w] = v
            result = DFS(G, w, came_from)
            if result is a goal:
                return result
    return None
```

该算法总是试图沿着一条路径一直搜索下去，并很快推进到搜索树的最深层，直到不能继续时，再回溯选择另一个未扩展节点继续搜索，这是深度优先搜索名称的由来。

与 BFS 算法类似，DFS 算法耗费的时间取决于 G 的存储结构。如果使用二

维数组表示图的邻接矩阵，那么在最坏情况下，时间复杂度为 $O(n^2)$，n 为顶点数；如果使用邻接表作为图的存储结构，那么在最坏情况下，时间复杂度为 $O(n+e)$，e 为无向图的边数或有向图的弧数。

BFS 算法使用 FIFO 队列，而 DFS 算法则使用 LIFO 线性结构 (最新生成的节点最先被访问)。因此，很自然地，可以使用 Stack 数据结构实现非递归形式的 DFS 算法：

```
Input:
    A graph G and a vertex v of G
    came_from is a DICT, initialized with {}
Output:
    return: GOAL or None
    came_from is changed
def DFS(G, v, came_from):
    frontier = Stack()
    frontier.push(v)
    came_from[v] = None
    while not frontier.is_empty():
        v = frontier.pop()
        if v is not labeled as discovered:
            if v is a goal:
                return v
            else:
                label v as discovered
                for all edges from v to w in G.adjacentEdges(v):
                    frontier.push(w)
                    came_from[w] = v
    return None
```

上述 2 种 DFS 算法以相反的顺序访问子节点：在递归算法中，第 1 个子节点最先进入递归；在 Stack 形式的非递归算法中，第 1 个子节点最先进入 Stack，最后 1 个子节点最后进入 Stack，但它最先被弹出访问。

还有一种常见的非递归形式，采用 OPEN-CLOSED 表作为数据结构：

```
Input:
    A graph G and a vertex v of G
    came_from is a DICT, initialized with {}
Output:
    return: GOAL or None
    came_from is changed
def DFS(G, v, came_from):
    open = [v]
    closed =[]
    came_from[v] = None
    while open is not empty:
        remove an element from the front of open, call it v
        if v is a goal:
            return v
        else:
            put v on closed
            for all edges from v to w in G.adjacentEdges(v):
                if w is not in closed:
                    put w at the front of open
                    came_from[w] = v
    return None
```

从 OPEN-CLOSED 表的实现方式看，BFS 算法与 DFS 算法的唯一差别在于，子节点添加到 OPEN 表的位置不同，BFS 将子节点添加到 OPEN 表的末尾，而 DFS 将子节点添加到 OPEN 表的开头。正是由于这个差异，才使得子节点被访问的顺序不同。

在无限的状态空间中，如果遇到了无限而又无法到达目标节点的路径，无论是图搜索，还是树搜索，DFS 算法都会失败。在这种情况下，该算法不具有完备性。在有限的状态图中，如果采取措施避免重复状态，那么 DFS 算法是完备的。根据 DFS 算法的运行特点，易知，在最坏情况下，搜索路径的最大深度 m 会远远大于目标节点的最浅深度 d，因而，DFS 算法也不具有最优性。

DFS 算法的时间复杂度取决于访问的节点个数。在最坏情况下，将访问状态

树中 $O(b^m)$ 个节点，其中，m 是所有路径中的最大深度。需要注意的是，m 可能要比 d(目标节点的最浅深度) 大得多，而且，如果状态树是无限的，m 可能就是无限的。与 BFS 算法相比，DFS 算法的时间复杂度处于劣势地位。

然而，DFS 算法的最大优势在于空间复杂度。搜索时，DFS 算法只需要存储一条从根节点到叶节点的路径及路径上每个节点的所有未访问的兄弟节点。在最坏情况下，路径的最大深度为 m，那么 DFS 算法只需要保存 $O(bm)$ 个节点[10]。例如，使用如图2-2中的条件，并假设位于目标深度的节点没有子节点，那么当深度 $d = 16$ 时，DFS 算法只需要 $156K$ 字节，而 BFS 算法却需要 10^{19} 字节，这是一个巨大的差异。这一优势，使得 DFS 算法在 AI 的许多领域得到应用，例如约束满足问题、命题逻辑可满足性、逻辑程序设计等。

回溯搜索 (Backtracking Search, BS) 是 DFS 算法的变形，所需的内存空间更少。在 BS 算法中，每次只产生一个后继子节点，而不是一次生成所有的后继子节点，但是每个被部分访问的节点要记住下一个待访问的后继子节点。因此，内存空间的复杂度是 $O(m)$，而不是 $O(bm)$。BS 算法还避免了大状态节点的复制问题，从而减少了空间与时间消耗——直接修改当前的状态节点，以生成后继子节点，而不是像其它算法那样复制状态节点后再修改。为了实现这一功能，回溯时必须能够撤销以前所做的修改。这一技术是机器人装配等应用成功的关键，这些应用具有大状态节点，常规技术难以奏效。

3.2 深度受限的搜索

深度受限的搜索 (Depth-Limited Search, DLS) 在搜索到一定深度 l 时回溯或停止搜索，用来解决 DFS 算法在无限状态空间中搜索时存在的无穷路径问题。

然而，深度 l 的选择是一个棘手的问题。如果 $l < d$，即路径最大深度小于目标最浅深度，那么 DLS 算法就不是完备的。如果 $l > d$，该算法也不是最优的。DLS 算法的时间复杂度是 $O(b^l)$，空间复杂度是 $O(bl)$。

[10]没有考虑 came_from 数据结构的内存需求。即使考虑的话，在同等条件下，DFS 算法的内存需求仍然占很大的优势。况且，came_from 并不是必需对象，可以采取措施只保存一条最短路径。

3.3 迭代加深的 DFS

迭代加深 DFS(Iterative Deepening Depth-First Search, IDDFS) 是一种迭代使用 DFS 算法进行搜索的策略，可用来确定目标节点所处的最浅深度 d。其执行过程是，从深度 0 开始，执行一次 DFS 算法 (相当于 $l = 0$ 的深度受限搜索)，接着从深度 1 开始，执行一次 DFS 算法 (相当于 $l = 1$ 的深度受限搜索)，依此类推，直到找到目标 (当 $l = d$ 时，就能找到目标节点)。算法如下：

```
Input:
    A graph G and a vertex v of G
    came_from is a DICT, initialized with {}
Output:
    return: GOAL or None
    came_from is changed
def IDDFS(G, v, came_from):
    came_from[v] = None
    for depth from 0 to MAX_DEPTH:
        found = DLS(G, v, depth, came_from)
        if found != None:
            return found
    return None

def DLS(G, v, depth, came_from):
    if depth == 0:
        if v is a goal:
            return v
        else
            return None
    elif depth > 0:
        locally label v as discovered
        for all edges from v to w in G.adjacentEdges(v):
            if vertex w is not labeled as discovered:
```

```
                came_from[w] = v
                found = DLS(G, w, depth-1, came_from)
                if found != None:
                    return found
        return None
```

在上面的算法中，DLS 表示深度受限搜索函数，返回：None，表示目标节点为空；非 None，目标节点对象。

IDDFS 算法具有 DFS 与 BFS 算法的优点。它的空间需求与 DFS 算法类似，为 $O(bd)$。与 BFS 算法一样，当分支因子有限时，该算法是完备的；当路径代价是深度的非递减函数时，该算法也是最优的。

关于时间复杂度，给人的直觉是，该算法做了“许多”重复的工作，因为算法要从 $l = 0$，一直迭代到 $l = d$，而每次迭代搜索生成的状态树都会包括前一次的状态树，只是深度更深一层。但是，仔细分析的话，代价并不是很大。原因是，深度 d 的节点只生成一次，深度 $d-1$ 的节点生成 2 次，深度 $d-2$ 的节点生成 3 次，依此类推，一直到根节点的子节点，其生成次数为 d；但是，越靠近根节点，节点数就越少，因而它们对总体代价的影响并不大。因此，算法生成的节点总数为：

$$db + (d-1)b^2 + (d-2)b^3 + ... + 2b^{d-1} + b^d \tag{3}$$

时间复杂度仍然是 $O(b^d)$，量级与 BFS 算法一样。例如，当 $b = 10$，$d = 5$ 时，IDDFS 算法与 BFS 算法需访问的节点总数分别为：

$$\begin{aligned} &50 + 400 + 3000 + 20000 + 100000 = 123450 \\ &10 + 100 + 1000 + 10000 + 100000 = 111110 \end{aligned} \tag{4}$$

如果确实很在意搜索树的重复生成问题，可以考虑混合使用两种搜索算法——先尽量使用 BFS 算法，在内存空间达到极限前，再对待访问节点执行 IDDFS 算法。

一般而言，当搜索空间较大而目标节点的深度未知时，IDDFS 算法总是优先选择的无信息搜索方法。

从整体效果上看，IDDFS 算法将 DFS 与 BFS 算法结合在一起，其中，BFS 算法将步数作为代价函数。自然地，也可以将 DFS 与 UCS 算法结合在一起，其中，UCS 算法为每步引入代价函数，步数只是该代价函数的特例。这种新算法被命名为迭代加长 DFS(Iterative Lengthening Depth-First Search, ILDFS) 算法，它

在确保最优化的同时也避免了大量的内存需求，也是一种有价值的搜索算法。然而，不幸的是，与 UCS 算法存在的问题一样，ILDFS 算法也需要在执行时间上付出较大的额外开销。

4 实践

4.1 SimpleGraph 类

SimpleGraph 类实现了简单的图数据结构，程序如下：

```
class SimpleGraph:
    def __init__(self):
        self.edges = {}

    def adjacentEdges(self, v):
        return self.edges[v]

G1 = SimpleGraph()
G1.edges = {
    '1' : ['2', '3', '4'],
    '2' : ['5', '6'],
    '3' : [],
    '4' : ['7', '8'],
    '5' : ['9', '10'],
    '6' : [],
    '7' : ['11', '12'],
    '8' : [],
    '9' : [],
    '10' : [],
    '11' : [],
    '12' : []
}
```

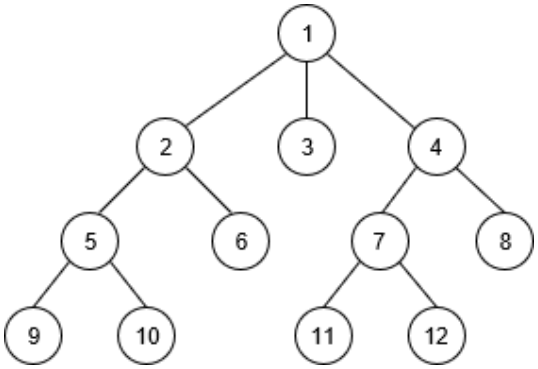

图 4-4: G1 的结构图

在程序中，定义了 G1 图对象，方便后期使用，如图4-4所示。

4.2 Graph 类

Graph 类实现了简单的图数据结构，允许每步代价不一样。程序如下：

```
class Graph:
    def __init__(self):
        self.edges = {}

    def adjacentEdges(self, v):
        return [e[0] for e in self.edges[v]]

    def cost(self, v, w):
        l = [e[1] for e in self.edges[v] if e[0] == w]
        if len(l) > 0:
            return l[0]
        else:
            return MAX_COST

MAX_COST = 10000
G2 = Graph()
G2.edges = {
```

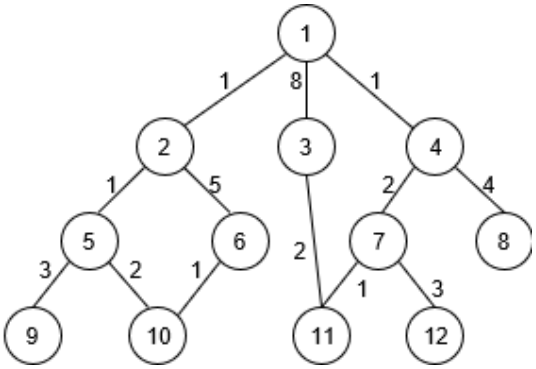

图 4-5: G2 的结构图

```
    'A' : [('B',1), ('C',8), ('D',1)],
    'B' : [('E',1), ('F',5)],
    'C' : [('K',2)],
    'D' : [('G',2), ('H',4)],
    'E' : [('I',3), ('J',2)],
    'F' : [('J',1)],
    'G' : [('K',1), ('L',3)],
    'H' : [],
    'I' : [],
    'J' : [('F',1)],
    'K' : [('C',2)],
    'L' : []
}
```

在程序中，定义了 G2 图对象，方便后期使用，如图4-5所示。节点之间的数值表示代价。

4.3 Queue 类

Queue 类实现了简单的队列，程序如下：

```
class Queue:
    def __init__(self):
        self.items = []
```

```
4
5       def enqueue(self, item):
6           self.items.insert(0,item)
7
8       def dequeue(self):
9           return self.items.pop()
10
11      def is_empty(self):
12          return self.items == []
13
14      def size(self):
15          return len(self.items)
16
17  def main():
18      q = Queue()
19      q.enqueue(1)
20      q.enqueue(2)
21      q.enqueue(3)
22      q.enqueue(4)
23      q.enqueue(5)
24      print(q.size())
25      while not q.is_empty():
26          print(q.dequeue())
27      print(q.size())
28
29  if __name__ == '__main__':
30      main()
```

4.4 PriorityQueue 类

PriorityQueue 类实现了简单的优先队列，程序如下：

```python
import heapq

class PriorityQueue:
    def __init__(self):
        self.elements = []

    def enqueue(self, item, priority):
        heapq.heappush(self.elements, (priority, item))

    def dequeue(self):
        return heapq.heappop(self.elements)[1]

    def is_empty(self):
        return self.elements == []

    def size(self):
        return len(self.elements)

def main():
    pq = PriorityQueue()
    pq.enqueue(3, 3)
    pq.enqueue(2, 2)
    pq.enqueue(1, 1)
    pq.enqueue(4, 4)
    pq.enqueue(5, 5)
    print(pq.size())
    while not pq.is_empty():
        print(pq.dequeue())
    print(pq.size())

if __name__ == '__main__':
```

```
32      main()
```

4.5 Stack 类

Stack 类实现了简单的栈，程序如下：

```
1   class Stack:
2       def __init__(self):
3           self.items = []
4
5       def push(self, item):
6           self.items.append(item)
7
8       def pop(self):
9           return self.items.pop()
10
11      def is_empty(self):
12          return self.items == []
13
14      def peek(self):
15          return self.items[-1]
16
17      def top(self):
18          return self.items[-1]
19
20      def size(self):
21          return len(self.items)
22
23  def main():
24      s = Stack()
25      print(s.is_empty())
26      s.push(1)
```

```
27      s.push(2)
28      s.push(3)
29      s.push(4)
30      print(s.size())
31      print(s.top())
32      print(s.is_empty())
33      while not s.is_empty():
34          print(s.pop())
35
36  if __name__ == '__main__':
37      main()
```

4.6 BFS 算法的队列实现

```
1   from queue import Queue
2   from stack import Stack
3   from simplegraph import *
4
5   def BFS(G, v, goal, came_from):
6       frontier = Queue()
7       frontier.enqueue(v)
8       came_from[v] = None
9       visited = {}
10      while not frontier.is_empty():
11          v = frontier.dequeue()
12          if visited.get(v) == None:
13              if v == goal:
14                  return v
15              else:
16                  visited[v] = True
17                  for w in G.adjacentEdges(v):
18                      frontier.enqueue(w)
```

```
19                      came_from[w] = v
20      return None
21
22  def main():
23      came_from = {}
24      start = '1'
25      goal = '12'
26      found = BFS(G1, start, goal, came_from)
27      print('start:', start)
28      print('goal:', goal)
29      if found != None:
30          s = Stack()
31          s.push(found)
32          found = came_from.get(found)
33          while found != None:
34              s.push(found)
35              found = came_from.get(found)
36          while not s.is_empty():
37              print(s.pop(), end='-')
38          print('end')
39      else:
40          print('Path not found!')
41
42  if __name__ == '__main__':
43      main()
```

运行结果如下:

```
start: 1
goal: 12
1-4-7-12-end
```

4.7 BFS 算法的 OPEN-CLOSED 实现

```
from stack import Stack
from simplegraph import *

def BFS(G, v, goal, came_from):
    open = [v]
    closed = []
    came_from[v] = None
    while len(open) != 0:
        v = open.pop(0)
        if v == goal:
            return v
        else:
            closed.append(v)
            for w in G.adjacentEdges(v):
                if w not in closed:
                    open.append(w)
                    came_from[w] = v
    return None

def main():
    came_from = {}
    start = '1'
    goal = '12'
    found = BFS(G1, start, goal, came_from)
    print('start:', start)
    print('goal:', goal)
    if found != None:
        s = Stack()
        s.push(found)
        found = came_from.get(found)
```

```
31          while found != None:
32              s.push(found)
33              found = came_from.get(found)
34          while not s.is_empty():
35              print(s.pop(), end='-')
36          print('end')
37      else:
38          print('Path not found!')
39
40  if __name__ == '__main__':
41      main()
```

4.8 UCS 算法的实现

```
1   from priorityqueue import PriorityQueue
2   from stack import Stack
3   from graph import *
4
5   def UCS(G, v, goal, came_from):
6       frontier = PriorityQueue()
7       cost_so_far = {}
8       frontier.enqueue(v, 0)
9       cost_so_far[v] = 0
10      came_from[v] = None
11      while not frontier.is_empty():
12          v = frontier.dequeue()
13          if v == goal:
14              return v
15          else:
16              for w in G.adjacentEdges(v):
17                  new_cost = cost_so_far[v] + G.cost(v, w)
18                  if cost_so_far.get(w) == None or new_cost < cost_so_far[w]:
```

```
                    cost_so_far[w] = new_cost
                    priority = new_cost
                    frontier.enqueue(w, priority)
                    came_from[w] = v
    return None

def main():
    came_from = {}
    start = '1'
    goal = '3'
    found = UCS(G2, start, goal, came_from)
    print('start:', start)
    print('goal:', goal)
    if found != None:
        s = Stack()
        s.push(found)
        found = came_from.get(found)
        while found != None:
            s.push(found)
            found = came_from.get(found)
        while not s.is_empty():
            print(s.pop(), end='-')
        print('end')
    else:
        print('Path not found!')

if __name__ == '__main__':
    main()
```

运行结果如下:

```
start: 1
```

```
goal: 3
1-4-7-11-3-end
```

4.9 DFS 算法的递归实现

```python
from stack import Stack
from graph import *

def DFS(G, v, goal, came_from):
    if came_from == {}:
        came_from[v] = None
    if v == goal:
        return v
    for w in G.adjacentEdges(v):
        if came_from.get(w) == None:
            came_from[w] = v
            result = DFS(G, w, goal, came_from)
            if result == goal:
                return result
    return None

def main():
    came_from = {}
    start = '1'
    goal = '6'
    found = DFS(G2, start, goal, came_from)
    print('start:', start)
    print('goal:', goal)
    if found != None:
        s = Stack()
        s.push(found)
        found = came_from.get(found)
```

```
28          while found != None:
29              s.push(found)
30              found = came_from.get(found)
31          while not s.is_empty():
32              print(s.pop(), end='-')
33          print('end')
34      else:
35          print('Path not found!')
36
37  if __name__ == '__main__':
38      main()
```

4.10 DFS 算法的 Stack 实现

```
1   from stack import Stack
2   from graph import *
3
4   def DFS(G, v, goal, came_from):
5       frontier = Stack()
6       frontier.push(v)
7       came_from[v] = None
8       visited = {}
9       while not frontier.is_empty():
10          v = frontier.pop()
11          if visited.get(v) == None:
12              if v == goal:
13                  return v
14              else:
15                  visited[v] = True
16                  for w in G.adjacentEdges(v):
17                      frontier.push(w)
18                      came_from[w] = v
```

```
19      return None
20
21  def main():
22      came_from = {}
23      start = '1'
24      goal = '11'
25      found = DFS(G2, start, goal, came_from)
26      print('start:', start)
27      print('goal:', goal)
28      if found != None:
29          s = Stack()
30          s.push(found)
31          found = came_from.get(found)
32          while found != None:
33              s.push(found)
34              found = came_from.get(found)
35          while not s.is_empty():
36              print(s.pop(), end='-')
37          print('end')
38      else:
39          print('Path not found!')
40
41  if __name__ == '__main__':
42      main()
```

4.11 DFS 算法的 OPEN-CLOSED 实现

```
1  from stack import Stack
2  from graph import *
3
4  def DFS(G, v, goal, came_from):
5      open = [v]
```

```
6       closed = []
7       came_from[v] = None
8       while len(open) != 0:
9           v = open.pop(0)
10          if v == goal:
11              return v
12          else:
13              closed.append(v)
14              for w in G.adjacentEdges(v):
15                  if w not in closed:
16                      open.insert(0, w)
17                      came_from[w] = v
18      return None
19
20  def main():
21      came_from = {}
22      start = '1'
23      goal = '11'
24      found = DFS(G2, start, goal, came_from)
25      print('start:', start)
26      print('goal:', goal)
27      if found != None:
28          s = Stack()
29          s.push(found)
30          found = came_from.get(found)
31          while found != None:
32              s.push(found)
33              found = came_from.get(found)
34          while not s.is_empty():
35              print(s.pop(), end='-')
36          print('end')
```

```
37      else:
38          print('Path not found!')
39
40  if __name__ == '__main__':
41      main()
```

4.12 IDDFS 算法的实现

```
1   from stack import Stack
2   from graph import *
3
4   MAX_DEPTH = 10
5
6   def IDDFS(G, v, goal, came_from):
7       came_from[v] = None
8       for depth in range(MAX_DEPTH+1):
9           visited = {}
10          found = DLS(G, v, goal, depth, came_from, visited)
11          if found != None:
12              return found
13      return None
14
15  def DLS(G, v, goal, depth, came_from, visited):
16      if depth == 0:
17          if v == goal:
18              return v
19          else:
20              return None
21      elif depth > 0:
22          visited[v] = True
23          for w in G.adjacentEdges(v):
24              if visited.get(w) == None:
```

```
25                  came_from[w] = v
26                  found = DLS(G, w, goal, depth-1, came_from, visited)
27                  if found != None:
28                      return found
29          return None
30
31  def main():
32      came_from = {}
33      start = '1'
34      goal = '6'
35      found = IDDFS(G2, start, goal, came_from)
36      print('start:', start)
37      print('goal:', goal)
38      if found != None:
39          s = Stack()
40          s.push(found)
41          found = came_from.get(found)
42          while found != None:
43              s.push(found)
44              found = came_from.get(found)
45          while not s.is_empty():
46              print(s.pop(), end='-')
47          print('end')
48      else:
49          print('Path not found!')
50
51  if __name__ == '__main__':
52      main()
```

4.13 练习

1. 编写程序，遍历 SimpleGraph.py 中的 G1 对象 (SimpleGraph 对象)；

2. 使用 GUI 图形对象，可视化 G1 对象的结构；

3. 在 G1 图形化程序的基础上，编写 BFS 搜索算法 (各版本) 的动画演示程序；

4. 实现迷宫的宽度优先与深度优先搜索算法；

5 参考文献

1. Wikipedia: Depth-First Search.. Accessed on Aug. 20, 2018.

2. Depth-First Search.. Accessed on Aug. 20, 2018.

3. Wikipedia: Iterative deepening depth-first search.. Accessed on Aug. 20, 2018.

4. Stuart J. Russell, Peter Norvig, 殷建平等译。《人工智能：一种现代的方法》，第 3 版，清华大学出版社。

5. Wikipedia: Breadth-First Search. Accessed on Aug. 21, 2018.

6. Breadth-First Search. Accessed on Aug. 21, 2018.