第3章

状态空间搜索

作为一类系统性探索问题空间并进行求解的方法，状态空间搜索通过状态空间法对问题进行表示，实现对现实世界问题状态、状态间转移规则的表示；同时，使用图搜索算法在问题表示的基础上进行求解，以找到从初始状态到达目标状态的一条最优（或较优）路径。本章首先阐述状态空间搜索的基本概念和方法，随后选取基于盲目搜索和启发式搜索的典型实验案例，分别应用于路径规划问题、八数码问题和规划问题的求解，并提供了实验要求和编程要点。

3.1 状态空间搜索基础

3.1.1 状态空间的基本概念

在状态空间法中，可以将现实世界的问题分解表示为若干可以互相转换的状态，以及它们之间的转换关系。状态空间搜索的核心思想是将待求解问题抽象表示为一个状态空间，然后在此空间内进行搜索，目的是寻找从初始状态到目标状态的代价最小或较小的路径。待求解问题各要素与状态空间图各要素的对应关系如表 3.1 所示。

表 3.1 待求解问题各要素与状态空间图各要素的对应关系

待求解问题	状态空间图
某步骤或某时刻的局面	节点
初始局面	初始节点
目标局面	目标节点
操作算子（状态转换规则）	边
转换所需的代价	边上的权
完成操作序列所需的代价	路径的代价

续表

待求解问题	状态空间图
问题的解	从初始节点到目标节点的一条路径
求解过程	寻找路径过程

3.1.2 通用图搜索框架

在搜索过程中，从初始状态出发，通过不断探索和尝试，逐步发现状态空间中更多的节点和边，探索到目标状态时，就找到了从初始节点到目标节点的路径。按照“边构图边寻路”的思想，可以设计通用图搜索框架。在通用图搜索流程中，为了对搜索过程中已扩展和待扩展的节点进行区分，可以设置 Open 表和 Closed 表对已知节点进行存储，其中 Closed 表存储已知且已扩展节点，Open 表存储已知待扩展节点。

通用图搜索流程如图 3.1 所示。每次循环时，均从 Open 表中选择一个节点 n 进行扩展（步骤②和④），将节点 n 移到 Closed 表中，将 n 的后继节点 $\{n_i\}$ 放入 Open 表中（步骤⑤），循环成功结束的条件是正在扩展的节点为目标节点（步骤③）。如果待求解的问题无解，那么按照上述流程执行下去，最终将耗尽 Open 表中的所有待扩展节点，仍然找不到目标节点。因此，问题无解时的退出条件是 Open 表为空（步骤①）。

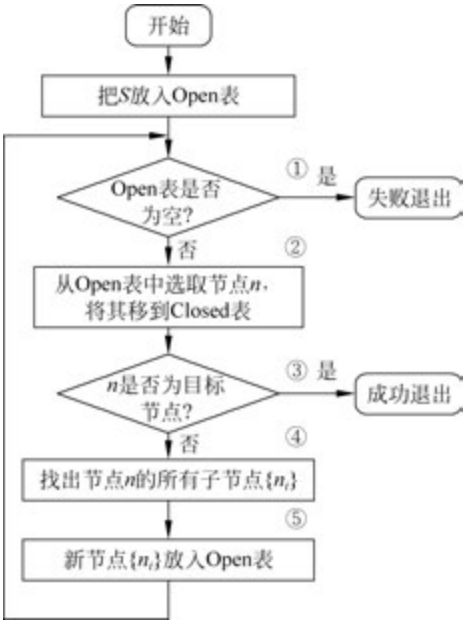

图 3.1 通用图搜索的流程图

按照上述流程，状态空间图是随着探索过程逐步构建的。开始时只有初始节点是已知的，然后通过对 Open 节点的扩展，逐步探索更多的节点和边，直到找到目标节点。图 3.2 演示了状态空间图的逐步探索过程，已知节点范围包含 Open 节点和 Closed 节点，其中 Open 节点处于已知节点范围的边缘。对 Open 节点的扩展，可以使已知范围不断扩大，当目标节点成为 Closed 节点后，可以终止搜索。该搜索算法通常不需要构建完

整的状态空间图，如果算法设计得当，那么即使很复杂的问题也只需要探明小部分状态空间图就可以实现问题求解。

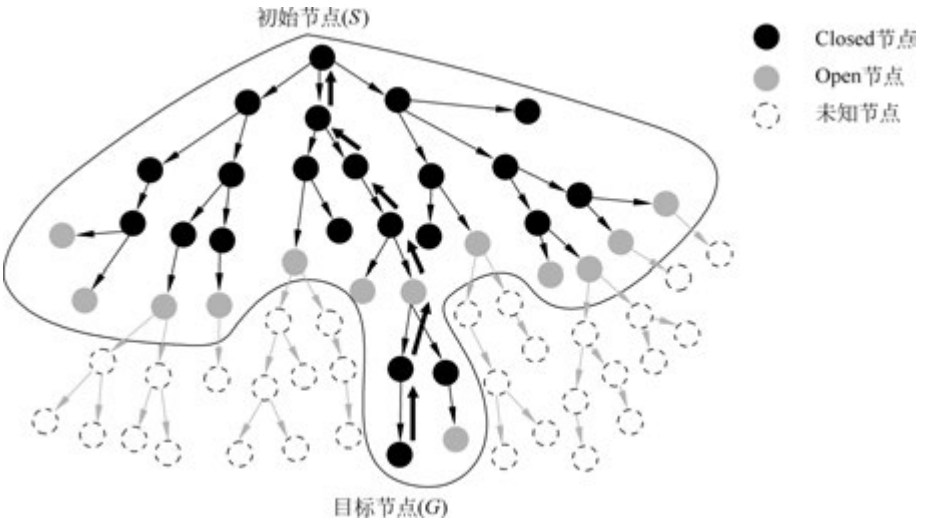

图 3.2 状态空间图的逐步探索过程

在解决实际问题时，仅找到目标节点往往是不够的，一般还需要给出从初始节点到达目标节点的解路径。由于每个节点只有一个父节点，因此从目标节点出发，反方向回溯解路径将较容易实现。如图 3.2 所示，从目标节点出发，逐级回溯其父节点，直到初始节点，就找到了解的逆路径(粗箭头所示)。这就需要在搜索过程中记录节点之间的父子关系，即在节点扩展时，为新扩展出来的每个子节点标记其父节点信息。

在具体实现时，通用图搜索流程中还有很多细节问题需要考虑，包括步骤②中的选点策略、步骤④中的节点数据结构设计和转换规则的形式化描述、步骤⑤中的重复发现节点的处理等。其中，步骤②中的选点策略是算法设计的关键。采用何种选点策略对于图搜索算法的性能有决定性影响，根据不同选点策略，通用图搜索框架可衍生出不同的图搜索算法。

3.1.3 盲目搜索算法

盲目搜索算法采用的是较为简单的选点策略，即在节点选取过程中不对当前节点接近目标节点的程度进行评估分析，而是仅依靠各节点的发现次序等进行选点。盲目搜索算法包括宽度优先搜索算法、深度优先搜索算法和代价优先搜索算法及上述算法的相应变体。

宽度优先搜索算法(Breadth First Search，BFS)按照接近初始节点的程度由近及远地依次扩展节点。这是一种简单、常用的图搜索算法，其中宽度是相对于搜索树而言的。宽度优先搜索算法总是在搜索树的宽度方向上扩展完毕后再往深度方向扩展，如图 3.3 所示。

深度优先搜索(Depth First Search，DFS)总是优先扩展搜索树中深度更深的节点。如图 3.4 所示。深度优先搜索与宽度优先搜索的流程类似，扩展 Open 表节点时，都是从

Open 表中取出第一个节点，不同之处在于深度优先搜索算法将新扩展出来的节点加入 Open 表的头部。

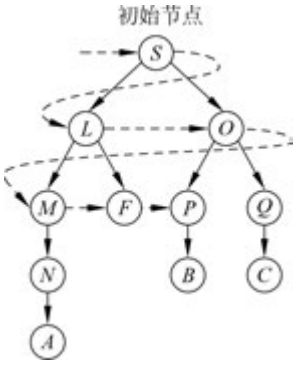

图 3.3　宽度优先搜索过程示意图

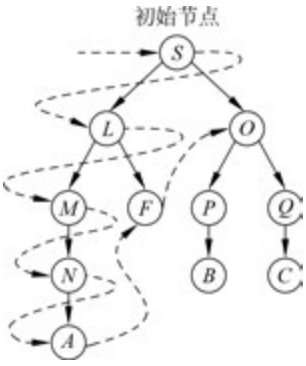

图 3.4　深度优先搜索过程示意图

代价优先搜索算法也称为一致代价搜索(Uniform Cost Search，UCS)算法。在代价优先搜索算法中，仍是从 Open 表取出第一个节点进行扩展，不同之处在于将新发现的节点放入 Open 表后，需要对 Open 表中的节点按照历史累积代价值(记为 g，简称代价值)从小到大排序，将 g 值最小的节点排在 Open 表的头部。

对于算法的评价，可以从定量和定性两个角度使用四个指标展开。分别是时间复杂度，即找到解所需的时间，可以用扩展的总节点数来衡量；空间复杂度，即搜索过程所需的内存空间，可以用算法需要存储的最大节点数来衡量；完备性，即对于有解的问题一定能够找到解；最优性，即找到的解是所有解中代价最小的解。表 3.2 比较了三种盲目搜索算法的特点。

表 3.2　不同盲目搜索算法的特点

盲目搜索算法	Open 表数据结构	时间复杂度	空间复杂度	完备性	最优性
宽度优先搜索	队列	$O(b^d)$	$O(b^d)$	满足	满足(仅对单位代价问题)
深度优先搜索	堆栈	$O(b^m)$	$O(bl)$	不满足	不满足
代价优先搜索	优先级队列	$O(b^{\lceil C^*/\varepsilon \rceil})$	$O(b^{\lceil C^*/\varepsilon \rceil})$	满足	满足

表 3.2 中，b 为分支因子，即搜索树中节点的平均分支数目；d 为目标节点的深度，即从根节点到目标节点的最短路径长度；m 为搜索树的最大深度；l 为深度限制；C^* 为最优路径的代价，任意两个节点间的代价$>\varepsilon>0$；$\lceil\ \rceil$表示向上取整。

3.1.4　启发式搜索算法

在求解中与问题相关的能够用于简化搜索过程的信息称为启发信息(heuristic information)。如果在选择要扩展的节点时利用与问题相关的启发信息评估节点接近目标节点的程度，然后选取最有希望的节点进行扩展，那么该搜索算法称为启发式搜索算法。

对于如图 3.5 所示的搜索问题，S 和 G 分别是初始节点和目标节点，节点 m 是已扩

展节点，节点 n 和 p 是由节点 m 扩展出来的待扩展节点。当前任务是从 n 和 p 中选取一个节点进行扩展，即比较经过 n 和经过 p 的两条解路径 $S\rightarrow\cdots\rightarrow n\rightarrow\cdots\rightarrow G$ 和 $S\rightarrow\cdots\rightarrow p\rightarrow\cdots\rightarrow G$ 是最优解路径的可能性。

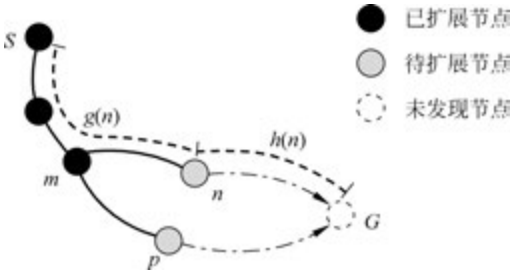

图 3.5 代价评估函数设计示意图

函数 $f(n)$ 表示节点 n 所在解路径的代价评估值。从 S 到 n 的路径代价值和从 n 到 G 的路径的预估值分别表示为函数 $g(n)$ 和 $h(n)$，有

$$f(n)=g(n)+h(n) \tag{3.1}$$

$h(n)$ 称为节点 n 的启发函数，体现了搜索的启发信息。假设从 n 到 G 的路径的实际最小代价为 $h^*(n)$，$h(n)$ 是 $h^*(n)$ 的估计值。

当 $f(n)=g(n)+h(n)$，且满足 $h(n)\leqslant h^*(n)$ 时，对应 A^* 算法，即最佳图搜索算法(best graph search method)。如果对于所有的节点 n，启发函数都满足 $h(n)\leqslant h^*(n)$，那么称该启发函数满足可纳性(admissibility)。A^* 算法具有最优性，即 A^* 算法必定结束在最优解路径上。

A^* 算法设计的关键是启发函数 $h(n)$ 的设计，启发函数应满足两个要求：一是尽量准确地反映实际代价；二是满足可纳性。因为不同问题中代价的含义不同，所以启发函数的设计也要具体问题具体分析。

对于同一个问题，可能存在多种满足可纳性条件的启发函数。A^* 算法的搜索效率很大程度上取决于启发函数 $h(n)$。如果对同一个问题有两个启发函数：$h_1(n)$ 和 $h_2(n)$，对于所有节点，都有 $h_1(n)\leqslant h_2(n)$，那么称 $h_2(n)$ 比 $h_1(n)$ 更具信息。更具信息的 $h_2(n)$ 对应算法扩展的节点数一定小于或等于 $h_1(n)$ 对应算法扩展的节点数，$h_2(n)$ 对应算法扩展的节点集是 $h_1(n)$ 对应算法扩展的节点集的子集。

3.2 基于盲目搜索和 A^* 搜索的路径规划

3.2.1 路径规划实验背景与要求

1. 实验背景

路径规划是指在特定的环境中，依据某些优化标准(如最短路径、最少时间等)，找到一条从初始点到目标点的最优或可行路径的过程。利用人工智能算法解决路径规划问题，要点在于如何利用状态空间表示法对问题进行描述，并利用不同的搜索算法得到解路径。在搜索过程中，通常采用搜索的总节点数、路径代价等指标对算法性能进行评价。

本实验以吃豆人(Pacman)实验框架作为基础,主要完成 search 问题集下的基于盲目搜索算法(宽度优先、深度优先和代价优先等)和 A* 搜索的路径规划问题求解。Pacman 实验主要文件的描述见表 3.3。

表 3.3 Pacman 实验主要文件

编号	文件名	内 容
1	search. py	需要实现的搜索算法所在的文件
2	searchAgents. py	搜索 Agent 的定义文件
3	pacman. py	运行 Pacman 游戏的文件,定义游戏状态等
4	game. py	Pacman 游戏底层运行机制的定义文件
5	util. py	运行搜索算法所需数据结构的定义文件

2. 实验目的和要求

1) 实验目的

通过本实验掌握盲目搜索算法和 A* 算法的基本原理和实现方法;针对不同的搜索算法,选择不同的数据结构作为 Open 表的数据类型;根据不同的搜索算法确定解路径的代价;能够选取合适的评估指标对不同算法进行分析和对比。

2) 实验要求

独立实现不同的盲目搜索算法和 A* 算法,并进行调试,确保代码正常运行。同时,从定性和定量角度分析对比不同算法的性能和效率,给出测评结果并撰写实验报告。

3.2.2 搜索算法设计

1. 深度优先搜索

对于深度优先搜索算法,其 Open 表通常对应于堆栈结构,算法的实现过程如下。

1) 参数定义与初始化

包括如下步骤。

(1) 定义 Open 表变量,对应于堆栈结构。

(2) 定义用于保存路径的变量,同样对应于堆栈结构。

(3) 定义变量,分别用于记录 Closed 表和目标路径。

(4) 将初始状态放入 Open 表中,并用其对当前状态变量 S 进行赋值。

2) 搜索过程

判断当前状态 S 是否是目标状态,如果是,则返回目标路径;否则,进行如下循环。

(1) 如果当前状态 S 未在 Closed 表中,则将其加入到 Closed 表中。

(2) 扩展当前状态 S,获得其所有后继状态信息(含状态、实现动作和代价)。将每个状态信息加入到 Open 表,将状态对应的实现动作加入到目标路径,并将生成的目标路径加入到保存路径的变量。

(3) 用路径变量中的第一个值为目标路径赋值,用 Open 表中的第一个值为当前状态 S 赋值。

2. 宽度优先搜索

对于宽度优先搜索算法，其 Open 表通常对应于队列结构，算法的实现过程如下。

1) 参数定义与初始化

包括如下步骤。

(1) 定义 Open 表变量，对应于队列结构。

(2) 定义用于保存路径的变量，同样对应于队列结构。

(3) 定义变量，分别用于记录 Closed 表和目标路径。

(4) 将初始状态放入 Open 表中，并用其对当前状态变量 S 进行赋值。

2) 搜索过程

判断当前状态 S 是否是目标状态，如果是，则返回目标路径；否则，进行如下循环。

(1) 如果当前状态 S 未在 Closed 表中，则将其加入 Closed 表中。

(2) 扩展当前状态 S，获得其所有后继状态信息(含状态、实现动作和代价)。将每个状态信息加入 Open 表，将状态对应的实现动作加入目标路径，并将生成的目标路径加入保存路径的变量。

(3) 用路径变量中的第一个值为目标路径赋值，用 Open 表中的第一个值为当前状态 S 赋值。

3. 代价优先搜索

对于代价优先搜索算法，其 Open 表通常对应于优先级队列结构，算法的实现过程如下。

1) 参数定义与初始化

包括如下步骤。

(1) 定义 Open 表变量，对应于优先级队列结构。

(2) 定义用于保存路径的变量，同样对应于优先级队列结构。

(3) 定义变量，分别用于记录 Closed 表和目标路径。

(4) 将初始状态放入 Open 表中，并用其对当前状态变量 S 进行赋值。

2) 搜索过程

判断当前状态 S 是否是目标状态，如果是，则返回目标路径；否则，进行如下循环。

(1) 如果当前状态 S 未在 Closed 表中，则将其加入 Closed 表中。

(2) 扩展当前状态 S，获得其所有后继状态信息(含状态、实现动作和代价)。对于每个后继状态，计算其路径代价。如果后继状态未在 Closed 表中，则将该状态信息加入 Open 表，并使用其路径代价作为优先级；同时，将生成的目标路径加入保存路径的变量，并使用其路径代价作为优先级。

(3) 用路径变量中的第一个值为目标路径赋值，用 Open 表中的第一个值为当前状态 S 赋值。

4. A^* 搜索

对于 A^* 搜索算法，其 Open 表同样对应于优先级队列结构，算法的实现过程与代价

优先搜索一致。与代价优先搜索的区别在于，A* 搜索算法中新生成节点的代价不仅考虑了已产生代价 $g(n)$，还包括启发式代价 $h(n)$。

3.2.3 程序实现代码框架

1. 深度优先搜索算法

在 search.py 文件中找到深度优先搜索算法的代码框架并进行编程实现，代码如下。

```
def depthFirstSearch(problem):
    # 该函数实现了深度优先搜索(Depth First Search, DFS)算法
    # problem: 问题对象，包含初始状态、目标状态以及获取后继状态的方法
    # 返回值是从初始状态到目标状态的路径(动作列表)

    # 使用栈(Stack)来存储待探索的节点
    open = util.Stack()

    # 使用栈(Stack)来存储路径(动作列表)
    path = util.Stack()

    # 记录已访问过的状态，避免重复访问
    closed = []

    # 存储当前路径
    goalpath = []

    # 将初始状态压入栈中
    open.push(problem.getStartState())

    # 从栈中弹出一个状态作为当前状态
    currstate = open.pop()

    # 循环直到找到目标状态
    while not problem.isGoalState(currstate):
        # 如果当前状态未被访问过
        if currstate not in closed:
            # 将当前状态标记为已访问
            closed.append(currstate)

            # 遍历当前状态的所有后继状态
            for next, action, cost in problem.getSuccessors(currstate):
                # 将后继状态压入栈中
                open.push(next)

                # 将当前路径加上新动作后压入路径栈中
                path.push(goalpath + [action])

        # 从路径栈中弹出最新的路径
        goalpath = path.pop()

        # 从栈中弹出下一个待探索的状态
        currstate = open.pop()

    # 返回从初始状态到目标状态的路径
    return goalpath
```

2. 宽度优先搜索算法

在 search.py 文件中找到宽度优先搜索算法的代码框架并进行实现，代码如下。

```
def breadthFirstSearch(problem):
    # 该函数实现了宽度优先搜索(Breadth First Search,BFS)算法
    # problem: 问题对象,包含初始状态、目标状态以及获取后继状态的方法
    # 返回值是从初始状态到目标状态的路径(动作列表)

    # 使用队列(Queue)来存储待探索的节点
    open = util.Queue()

    # 使用队列(Queue)来存储路径(动作列表)
    path = util.Queue()

    # 记录已访问过的状态,避免重复访问
    closed = []

    # 存储当前路径
    goalpath = []

    # 将初始状态压入队列中
    open.push(problem.getStartState())

    # 从队列中弹出一个状态作为当前状态
    currstate = open.pop()

    # 循环直到找到目标状态
    while not problem.isGoalState(currstate):
        # 如果当前状态未被访问过
        if currstate not in closed:
            # 将当前状态标记为已访问
            closed.append(currstate)

            # 遍历当前状态的所有后继状态
            for next, action, cost in problem.getSuccessors(currstate):
                # 将后继状态压入队列中
                open.push(next)

                # 将当前路径加上新动作后压入路径队列中
                path.push(goalpath + [action])

        # 从路径队列中弹出最早的路径
        goalpath = path.pop()

        # 从队列中弹出下一个待探索的状态
        currstate = open.pop()

    # 返回从初始状态到目标状态的路径
    return goalpath
```

3. 代价优先搜索算法

在 search.py 文件中找到代价优先搜索算法的代码框架并进行实现，代码如下。

```
def uniformCostSearch(problem):
    # 该函数实现了代价优先搜索(Uniform Cost Search, UCS)算法
    # problem: 问题对象,包含初始状态、目标状态以及获取后继状态的方法
    # 返回值是从初始状态到目标状态的最小代价路径(动作列表)

    # 使用优先队列(PriorityQueue)来存储待探索的节点,按路径代价排序
    open = util.PriorityQueue()

    # 使用优先队列(PriorityQueue)来存储路径(动作列表),按路径代价排序
    path = util.PriorityQueue()

    # 记录已访问过的状态,避免重复访问
    closed = []

    # 存储当前路径
    goalpath = []

    # 将初始状态压入优先队列中,初始路径代价为 0
    open.push(problem.getStartState(), 0)

    # 从优先队列中弹出一个状态作为当前状态
    currstate = open.pop()

    # 循环直到找到目标状态
    while not problem.isGoalState(currstate):
        # 如果当前状态未被访问过
        if currstate not in closed:
            # 将当前状态标记为已访问
            closed.append(currstate)

            # 遍历当前状态的所有后继状态
            for next, action, cost in problem.getSuccessors(currstate):
                # 计算从初始状态到后继状态的路径代价
                pathCost = problem.getCostOfActions(goalpath + [action])

                # 如果后继状态未被访问过
                if next not in closed:
                    # 将后继状态压入优先队列中,按路径代价排序
                    open.push(next, pathCost)

                    # 将当前路径加上新动作后压入路径优先队列中,按路径代价排序
                    path.push(goalpath + [action], pathCost)

        # 从路径优先队列中弹出代价最小的路径
        goalpath = path.pop()

        # 从优先队列中弹出下一个待探索的状态
        currstate = open.pop()

    # 返回从初始状态到目标状态的最小代价路径
    return goalpath
```

4. A* 搜索算法

在 search.py 文件中找到 A* 搜索算法的代码框架并进行实现,A* 搜索算法代码如下。

```
def aStarSearch(problem, heuristic = nullHeuristic):
    # 该函数实现了 A* 搜索算法
    # problem: 问题对象,包含初始状态、目标状态以及获取后继状态的方法
    # heuristic: 启发式函数,用于估计从当前状态到目标状态的代价(默认为 nullHeuristic,
# 即无启发式)
    # 返回值是从初始状态到目标状态的最小代价路径(动作列表)

    # 使用优先队列(PriorityQueue)来存储待探索的节点,按总代价(路径代价 + 启发式代价)
# 排序
    open = util.PriorityQueue()

    # 使用优先队列(PriorityQueue)来存储路径(动作列表),按总代价排序
    path = util.PriorityQueue()

    # 记录已访问过的状态,避免重复访问
    closed = []

    # 存储当前路径
    goalpath = []

    # 将初始状态压入优先队列中,初始总代价为 0
    open.push(problem.getStartState(), 0)

    # 从优先队列中弹出一个状态作为当前状态
    currstate = open.pop()

    # 循环直到找到目标状态
    while not problem.isGoalState(currstate):
        # 如果当前状态未被访问过
        if currstate not in closed:
            # 将当前状态标记为已访问
            closed.append(currstate)

            # 遍历当前状态的所有后继状态
            for next, action, cost in problem.getSuccessors(currstate):
                # 计算从初始状态到后继状态的路径代价
                pathCost = problem.getCostOfActions(goalpath + [action])

                # 计算总代价: 路径代价 + 启发式代价
                totalCost = pathCost + heuristic(next, problem)

                # 如果后继状态未被访问过
                if next not in closed:
                    # 将后继状态压入优先队列中,按总代价排序
                    open.push(next, totalCost)

                    # 将当前路径加上新动作后压入路径优先队列中,按总代价排序
                    path.push(goalpath + [action], totalCost)

        # 从路径优先队列中弹出总代价最小的路径
        goalpath = path.pop()

        # 从优先队列中弹出下一个待探索的状态
        currstate = open.pop()

    # 返回从初始状态到目标状态的最小代价路径
    return goalpath
```

可以看到，在对当前状态进行扩展时，考虑了新生成节点对应的状态、动作和代价。新生成节点的代价包含了已产生代价 $g(n)$ 和启发式代价 $h(n)$。

对于路径规划问题，可以定义曼哈顿距离和欧氏距离两种启发函数。基于曼哈顿距离的启发函数代码如下。

```
def manhattanDistance( xy1, xy2 ):
    return abs( xy1[0] - xy2[0] ) + abs( xy1[1] - xy2[1] )
```

该启发函数返回点 xy1 和点 xy2 之间的曼哈顿距离。

5. 地图选择与算法设置

Pacman 框架提供了 30 多种地图布局，通过在不同大小的地图中运行各类搜索算法，可对算法的性能进行比较。

例如，在 mediumMaze（中型迷宫）地图中运行宽度优先搜索算法可使用如下命令。

```
python pacman.py -l mediumMaze -p SearchAgent -a fn=bfs
```

其中，“-l”为地图布局选项，还可以选择 smallMaze（小型迷宫）、bigMaze（大型迷宫）等地图；“-a”为算法选项，可以对函数 fn 进行选择，bfs 代表宽度优先搜索算法，dfs 代表深度优先搜索，ucs 代表代价优先搜索，astar 代表 A* 搜索算法。

在 A* 搜索算法中，还需要定义启发函数 heuristic。例如，利用 A* 搜索算法解决大型迷宫问题，且启发函数为曼哈顿距离的命令如下。

```
python pacman.py -l bigMaze -z .5 -p SearchAgent -a fn=astar,heuristic
=manhattanHeuristic
```

3.2.4 实验结果分析

给定 4 种地图布局，不同搜索算法的结果对比见表 3.4。

表 3.4 路径规划问题中不同搜索算法的结果对比

地图布局	深度优先搜索		宽度优先搜索		代价优先搜索		A* 搜索	
	路径代价	扩展节点数	路径代价	扩展节点数	路径代价	扩展节点数	路径代价	扩展节点数
微型地图	10	15	8	15	8	15	8	14
小型地图	49	59	19	92	19	92	19	53
中型地图	130	146	68	269	68	269	68	221
大型地图	210	390	210	620	210	620	210	549

从表 3.4 可以看出，对于不同的地图布局，宽度优先搜索算法、代价优先搜索算法和 A* 搜索算法都能找到最优路径，而深度优先搜索算法没有找到最优解。A* 搜索算法由于引入了启发函数，在找到最优路径的同时，其搜索过程中扩展节点数相对于宽度优先搜索和代价优先搜索有所减小。由于这是一个单位代价问题，宽度优先搜索等价于代价优

先搜索。A* 搜索算法在路径规划问题中表现最为优异，尤其是在大规模地图中。在拓展实验中，可进一步比较不同启发函数对于 A* 搜索算法性能的影响。

3.3 基于代价优先和 A* 搜索的八数码问题求解

3.3.1 八数码实验背景与要求

1. 实验背景

八数码问题是人工智能领域中的一个经典问题，提供了一个合适的平台来测试和比较不同的搜索算法。八数码问题如图 3.6 所示。在 3×3 的棋盘上，放有 8 个棋子，每个棋子上标有 1～8 的某一数字。棋盘中留有一个空位，空位周围的棋子可以移动到空位上。

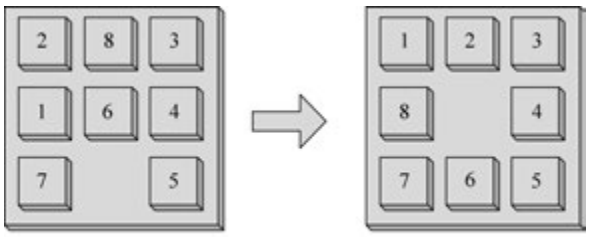

图 3.6 八数码问题示意图

对于八数码问题的求解，可以在定义问题的初始状态、目标状态、动作和转换规则的基础上，采用盲目搜索和启发式搜索算法进行求解。可沿用 Pacman 实验中所定义的深度优先搜索、宽度优先搜索、代价优先搜索和 A* 搜索算法代码，但是对于 A* 搜索算法，需要针对八数码问题重新设计相应的启发函数。八数码问题涉及状态的表示和判重技术。如何有效地表示状态空间以及避免重复访问已访问过的状态是解决问题的关键之一。

2. 实验目的和要求

1) 实验目的

理解八数码问题中状态的表示、动作及转换规则的设计、目标状态的判断等内容；掌握不同启发函数的构造和设计方法，能够对不同启发函数下的结果进行分析和对比，理解更具信息的启发函数的概念和含义。

2) 实验要求

针对八数码问题求解，能够设计不同的启发函数，并进行调试以确保代码正常运行；通过对比不同搜索算法以及 A* 搜索算法在不同启发函数下的结果，从盲目搜索和启发式搜索、不同启发函数的角度进行分析，给出实验结果并撰写实验报告。

3.3.2 八数码实验启发函数设计

启发函数 $h(n)$ 是 A* 搜索算法的关键，良好的启发函数需要尽量准确反映当前节点

到目标节点的实际代价，同时还需要满足可纳性。在具体实现上，可以通过松弛原问题的限定条件（即减少对操作算子的限制）的方式，降低从当前节点 n 到目标节点 G 所需的代价，然后将松弛后的代价作为评估代价。

正常情况下，八数码问题的棋子移动受到其与棋盘中空位的位置关系的约束。在启发函数的构造过程中，假设每个棋子均能不受约束地自由移动，则每个棋子移动至其目标位置的步数一定不大于其受到约束时所需的步数。该松弛条件下，对应的启发函数可表示为

$$h_1(n)=\sum_{k=1}^{8}(|x_{k,n}-x_{k,G}|+|y_{k,n}-y_{k,G}|) \tag{3.2}$$

式中，$(x_{k,n},y_{k,n})$和$(x_{k,G},y_{k,G})$分别为节点 n 和 G 中第 k 个棋子在棋局中的坐标。

此外，还可以设计另外一种启发函数，即只统计未归位棋子的数目，不考虑这些棋子与目标位置的距离，对应公式为

$$h_2(n)=\text{棋局 } n \text{ 中未归位的棋子数} \tag{3.3}$$

棋局中未归位的棋子数一定小于或等于归位整个棋局所需的步数，因此式(3.3)也满足可纳性条件。

3.3.3 程序实现代码框架

关于八数码问题的求解，可沿用 Pacman 问题中 SearchProblem 的定义，以及各类搜索算法的代码。本节主要完成八数码问题状态的构造及启发函数的定义。

1. 八数码问题状态的定义

载入所需的库，具体代码如下。

```
import search
import random
```

定义类名称为 EightPuzzleState，具体代码如下。

```
class EightPuzzleState:
```

定义该类的初始化函数，具体代码如下。

```
    def __init__(self, numbers):
        # 初始化函数，创建一个 3×3 的棋局
        # numbers 是一个包含 9 个数字的列表，表示棋盘的初始状态

        # 初始化棋局的单元格列表
        self.cells = []

        # 复制 numbers 列表，以避免修改原始数据
        numbers = numbers[:]    # 原样复制 numbers

        # 反转 numbers 列表，以便后续使用 pop() 方法时按顺序填充棋局
        numbers.reverse()
```

```
    # 遍历棋局的每一行(共 3 行)
    for row in range(3):
        # 为当前行初始化一个空列表
        self.cells.append([])

        # 遍历棋局的每一列(共 3 列)
        for col in range(3):
            # 从 numbers 列表中弹出最后一个元素,并将其添加到当前行的单元格中
            self.cells[row].append(numbers.pop())

            # 如果当前单元格的值为 0,则记录空位的位置
            if self.cells[row][col] == 0:
                self.blankLocation = row, col
```

定义判断是否为目标状态的函数,具体代码如下。

```
        if (self == problem.goalstate):
          return True
```

定义某一状态下合法操作的函数,具体代码如下。

```
def legalMoves(self):
    # 该函数用于返回当前空位可以移动的合法方向
    # 返回值是一个包含合法移动方向的列表,例如 ['up', 'down', 'left', 'right']

    # 初始化合法移动方向的列表
    moves = []

    # 获取空位的位置(行和列)
    row, col = self.blankLocation

    # 如果空位不在第一行,则可以向上移动
    if row != 0:
        moves.append('up')

    # 如果空位不在最后一行,则可以向下移动
    if row != 2:
        moves.append('down')

    # 如果空位不在第一列,则可以向左移动
    if col != 0:
        moves.append('left')

    # 如果空位不在最后一列,则可以向右移动
    if col != 2:
        moves.append('right')

    # 返回合法移动方向的列表
    return moves
```

定义状态转移规则函数,具体代码如下。

```
def result(self, move):
    # 根据给定的移动方向,生成一个新的八数码状态
    # move 是一个字符串,表示移动方向('up', 'down', 'left', 'right')
    # 返回值是一个新的 EightPuzzleState 对象,表示移动后的八数码状态

    # 获取空位的当前位置(行和列)
    row, col = self.blankLocation

    # 根据移动方向计算空白格的新位置
    if move == 'up':
        newrow = row - 1
        newcol = col
    elif move == 'down':
        newrow = row + 1
        newcol = col
    elif move == 'left':
        newrow = row
        newcol = col - 1
    elif move == 'right':
        newrow = row
        newcol = col + 1
    else:
        # 如果移动方向不合法,抛出异常
        raise "Illegal Move"

    # 创建一个新的 EightPuzzleState 对象,用于存储移动后的八数码状态
    # 初始状态为一个全 0 的列表,后续会被覆盖
    newPuzzle = EightPuzzleState([0, 0, 0, 0, 0, 0, 0, 0, 0])

    # 复制当前八数码状态的单元格到新对象中
    newPuzzle.cells = [values[:] for values in self.cells]

    # 更新八数码状态: 交换空位和目标位置的数字
    newPuzzle.cells[row][col] = self.cells[newrow][newcol]
    newPuzzle.cells[newrow][newcol] = self.cells[row][col]

    # 更新空位的位置
    newPuzzle.blankLocation = newrow, newcol

    # 返回新的八数码状态
return newPuzzle
```

2. 八数码搜索问题的定义

定义类名称为 EightPuzzleSearchProblem,该类有两个输入,具体代码如下。

```
class EightPuzzleSearchProblem(search.SearchProblem,EightPuzzleState):
```

定义类的初始化函数,具体代码如下。

```
    def __init__(self,puzzle, goalstate):
        self.puzzle = puzzle
        self.goalstate = goalstate
        self._expanded = 0
```

其中，puzzle 代表八数码问题的初始状态，goalstate 代表八数码问题的目标状态。

定义获取初始状态函数，具体代码如下。

```
    def getStartState(self):
        return puzzle
```

定义判断是否为目标状态的函数，具体代码如下。

```
    def isGoalState(self,state):
        if state == self.goalstate:
            return True
```

定义获取后继节点的函数，具体代码如下。

```
    def getSuccessors(self,state):
        succ = []
        for a in state.legalMoves():
            succ.append((state.result(a), a, 1))
            self._expanded += 1
        return succ
```

该函数返回结果为由(后继节点，动作，单步代价)构成的三元组的列表，其中后继节点对应状态为父节点通过其可选动作(上、下、左、右)所得的状态。

定义获取动作序列代价函数，具体代码如下。

```
    def getCostOfActions(self, actions):
        return len(actions)
```

该函数返回某特定动作序列的长度，设定八数码问题的单步代价为 1，则路径长度对应动作序列的代价。

3. 启发函数的定义

1) 启发函数 $h_1(n)$的实现

定义该启发函数的具体代码如下。

```
def h1Heuristic(curstate, problem, info={}):
    # h1 启发式函数，用于计算当前状态到目标状态的曼哈顿距离
    # curstate: 当前状态，是一个 EightPuzzleState 对象
    # problem: 问题对象，包含目标状态等信息
    # info: 可选参数，用于传递额外信息(此处未使用)
    # 返回值是当前状态到目标状态的曼哈顿距离总和

    # 获取当前状态和目标状态
    cur = curstate
    aim = problem.goalstate

    # 初始化启发式代价
    h_value = 0

    # 遍历当前状态的每一个单元格
```

```
    for row in range(3):                                    # 遍历每一行
        for col in range(3):                                # 遍历每一列
            # 获取当前单元格的值
            temp = cur.cells[row][col]

            # 如果当前单元格的值不为 0(即不是空位),则计算其曼哈顿距离
            if temp != 0:
                # 遍历目标状态的每一个单元格,寻找与当前单元格值相同的位置
                for rowtemp in range(3):                    # 遍历目标状态的每一行
                    for coltemp in range(3):                # 遍历目标状态的每一列
                        # 如果找到与当前单元格值相同的位置
                        if temp == aim.cells[rowtemp][coltemp]:
                            # 计算曼哈顿距离,并累加到启发式代价中
                            h_value += (abs(row - rowtemp) + abs(col - coltemp))
                            break                           # 找到目标位置后,跳出内层循环

    # 返回曼哈顿距离的总和
    return h_value
```

2) 启发函数 $h_2(n)$的实现

定义该启发函数的具体代码如下。

```
def h2Heuristic(curstate, problem, info={}):
    # h2 启发式函数,用于计算当前状态与目标状态的不一致单元格数量
    # curstate: 当前状态,是一个 EightPuzzleState 对象
    # problem: 问题对象,包含目标状态等信息
    # info: 可选参数,用于传递额外信息(此处未使用)
    # 返回值是当前状态与目标状态不一致的单元格数量(不包括空位)

    # 获取当前状态和目标状态
    cur = curstate
    aim = problem.goalstate

    # 初始化启发函数值
    h_value = 0

    # 遍历当前状态的每一个单元格
    for row in range(3):                    # 遍历每一行
        for col in range(3):                # 遍历每一列
# 如果当前单元格的值与目标状态对应位置的值不一致,并且当前单元格的值不为 0(即不是
# 空位)
            if (cur.cells[row][col] != aim.cells[row][col]) and (cur.cells[row][col] !=
0):
                # 启发函数值加 1
                h_value += 1

    # 返回不一致的单元格数量
return h_value
```

4. 算法性能对比

设定八数码问题的初始状态和目标状态,采用代价优先搜索算法和 A* 搜索算法(使

用两种启发函数)对问题进行求解,对比不同算法及其启发函数的效果。

本实验中用于测试算法效果的目标状态如图 3.7 所示。

图 3.7 八数码问题的目标状态示意图

主程序代码如下。

```
if __name__ == '__main__':
    goalstate = EightPuzzleState([0, 1, 2, 3, 4, 5, 6, 7, 8])   #设定目标状态
    puzzle = createRandomEightPuzzle(25)                        #随机生成初始状态
    print(puzzle)
    problem = EightPuzzleSearchProblem(puzzle,goalstate)        #构造八数码搜索问题

    #使用代价优先搜索进行求解,并显示动作数和扩展节点数
    ucs_path = search.uniformCostSearch(problem)
    print('UCS found a path of %d moves: %s with expanding %d nodes' % (len(bfs_path), str
(bfs_path), problem._expanded))
    problem._expanded = 0

    #使用基于启发函数 h1 的 A* 算法进行求解,并显示动作数和扩展节点数
    astar_path1 = search.aStarSearch(problem,heuristic = h1Heuristic)
    m_expanded1 = problem._expanded
    print('h1 found a path of %d moves: %s with expanding %d nodes' % (len(astar_path1),
str(astar_path1),m_expanded1))

    problem._expanded = 0
    #使用基于启发函数 h2 的 A* 算法进行求解,并显示动作数和扩展节点数
    astar_path2 = search.aStarSearch(problem,heuristic = h2Heuristic)
    m_expanded2 = problem._expanded
    print('h2 found a path of %d moves: %s with expanding %d nodes' % (len(astar_path2),
str(astar_path2),m_expanded2))
```

3.3.4 实验结果分析

给定多种初始棋局,应用不同搜索算法及启发函数的实验结果如表 3.5 所示。对于不同的初始棋局,代价优先搜索算法和 A^* 搜索算法求得的解路径均一致,且为最优解。A^* 搜索算法由于引入了启发函数,在找到最优路径的同时,其搜索过程中扩展节点数相对于代价优先搜索有所减小。同时,更具信息的启发函数对应的 A^* 搜索算法扩展的节点数更少(或至少一样多),搜索效率更高。

表 3.5　不同搜索算法及启发函数下的八数码问题结果对比

初始棋局	代价优先搜索		A^* 搜索(h_1 函数)		A^* 搜索(h_2 函数)	
	路径代价	扩展节点数	路径代价	扩展节点数	路径代价	扩展节点数
	11	2754	11	67	11	158
	11	3210	11	113	11	230
	9	1238	9	35	9	82
	5	152	5	15	5	15

八数码问题可能存在无解的情况，这取决于初始状态与目标状态的逆序数奇偶性。将一个状态表示成一维的形式，求出除 0 之外所有数字的逆序数之和，其中每个数字前面比它大的数字的个数的和称为这个状态的逆序数。若两个状态的逆序数奇偶性相同(同为奇数或偶数)，则可相互到达，否则不可相互到达。为了避免因无解而导致的程序运行时间过长，建议先进行初始状态和目标状态的可到达性判断，再进行搜索求解。

3.4　基于启发式搜索的规划问题求解

3.4.1　规划问题实验背景与要求

1. 实验背景

1) 规划的概念

本实验中的规划是指行动序列的规划，目的是如何根据初始状态和目标状态，来推理获得这个达成目标的动作序列。

经典规划研究的是完全可观察、确定性、静态和单 Agent 任务环境下的行动序列规划问题。规划问题可采用四元组(S, A, s_0, g)表示，其中，S 是用逻辑语句描述的状态集合；A 是动作(也称算子)集合，对于任意 $a \in A$，a：$S \rightarrow S$；s_0 为初始状态；g 为目标状态。规划问题的解为规划问题的动作序列，该序列能够将初始状态 s_0 映射为状态 s'，使得 s'满足 g。

相较于解决搜索问题，规划问题的求解过程将状态、动作和目标的黑箱打开，并采用逻辑语言对这些要素进行明确的表示。这样可以将复杂的目标分解为多个子目标，进而为每个子目标制订相应的计划。

2) 规划语言

对于规划问题，可以采用 STRIPS(Stanford Research Institute Problem Solver)、

PDDL(Planning Domain Definition Language)和 ADL(Action Description Language)等多种语言进行表示。本实验基于 STRIPS 语言对规划问题进行表示。

在 STRIPS 中,采用封闭世界假设,即没有在状态中出现的文字都为假。状态表示为正文字的合取式;目标表示为无变量正文字的合取式形式。

动作通常由以下三部分组成。

(1) 动作定义 ACTION,通常由动作名和参数表组成。

(2) 动作前提 PRECONDITION,它是文字的合取式。

(3) 动作效果 EFFECT,它是文字的合取式,包含正文字和负文字。

在操作语义的层面上,判断一个动作是否可以执行,其前提条件是状态 s 的可满足性:寻找与 s 中匹配的动作前提的最一般合一元。动作的具体化涉及对动作效果施加最一般合一元,即将动作正效果中的文字加入 s 中,并从 s 中删除负效果中的文字。其他不包含在动作效果中的文字保持不变,从而得到一个新状态 s'。

3) 规划问题求解方法

与搜索问题中状态空间图的概念类似,规划问题的所有状态和动作构成了规划问题的状态空间图。在状态空间中,如果定义了初始状态和目标状态,则规划问题的求解可以转化为在状态空间中寻找解路径的过程,可以利用搜索算法进行求解。

对于规划问题的求解,主要的搜索算法包括前向搜索、后向搜索和启发式搜索。前向搜索从初始状态出发,使用问题的动作向前搜索目标状态;而后向搜索则从目标的状态集出发,使用动作的逆向后搜索初始状态。对于前向搜索和后向搜索,可以采用树搜索/图搜索框架下的搜索求解方法,但存在搜索效率低或者陷入死循环等问题。因此,本例采用启发式搜索进行规划问题求解。

4) 问题设置

考虑航空货物运输问题,假设 m 架次飞机表示为 p_1, p_2, …, p_m,n 个机场表示为 a_1, a_2, …, a_n,需要运输的货物表示为 c_1, c_2, …, c_k。在问题设计中,可以根据需要的复杂度设置飞机、机场和货物的数量,飞机、货物所处的初始机场,以及货物最终要到达的目标机场等信息。求解的目标是在给定的初始状态和目标状态下,找到能够完成任务的动作规划序列。

对于此问题,可以定义如下状态和动作。

状态 1:In(c,p)表示货物 c 在飞机 p 中。

状态 2:At(x,a)表示对象(飞机或者货物)在机场 a。

动作 1:Load(装载)。

动作 2:Unload(卸载)。

动作 3:Fly(飞行)。

2. 实验目的和要求

1) 实验目的

通过本实验理解规划问题的状态表示、动作及转换规则的设计、目标状态的判断等内容;掌握基于 A^* 搜索算法的规划问题中启发函数的设计,对不同启发函数下的规划问

题解代价进行分析对比。

2）实验要求

针对规划问题进行求解，独立实现启发函数的设计，开展代码调试以确保代码可正常运行。对比不同启发函数下规划问题的解代价，进行原因分析，给出实验结果并撰写实验报告。

3.4.2 规划问题实验的启发函数设计

假设在规划问题中单个动作代价为1，则路径代价为路径上所有动作的代价之和。通过考察动作的效果文字、目标状态的文字，可以估计达成目标所需的动作数量，从而得到可纳的启发函数。

通常采用两类方法求取规划问题的启发函数，即松弛问题和目标分解。

对于松弛问题，主要有以下三种思路。

（1）忽略动作的前置条件启发式。对于规划问题中定义的每个动作，可以忽略所有前置条件，每个动作可施加于任何状态，此时可以通过单个动作获得至少一个目标文字。因此，解决目标合取式的步数大致为目标中未满足的文字数目。对于该启发式设计思路，存在两个变化因素：一是一个动作的正效果可能被另一动作删除；二是一个动作可能获得多个目标文字。如果子目标独立性假设成立，则上述两个变化因素可消除，启发函数值等于未满足的目标文字数目。

（2）目标的最小覆盖集启发式。利用该启发式时，会忽略动作的前置条件和动作中的负效果，然后寻找最小动作集，若该动作集中所有动作的正效果文字的并集满足目标文字，则启发函数为该最小动作集的元素个数。

（3）忽略删除列表启发式。该启发式通过忽略动作的负效果而得到相应的松弛问题，并将松弛问题的解代价作为原问题的启发函数值。该启发式一般较为精确，但是需要对整个松弛问题进行求解。

对于目标分解启发式，其思路是将目标分解为一组独立的子目标合取式，解决目标的代价近似于独立解决每个子目标的代价之和。此外，还可以通过构建规划图，即从初始状态到目标状态的可达图的近似表示，来求取规划问题的启发函数。

完成规划图构建后，定义获得目标文字的代价估计值为其在规划图中首次出现的层次（Level）。对于单个目标文字，该估计值为可纳的启发函数。基于规划图，可以定义如下三种启发式。

（1）最大层次启发式（Max-level）。该启发式的值对应于所有目标文字首次出现层次值中的最大值，是可纳的启发函数，但不精确。

（2）层次和启发式（Level Sum）。在遵循子目标独立性假设的情况下，该启发式为所有目标文字的层次代价之和，属于不可纳的启发函数。

（3）集合层次启发式（Set-level）。寻找某个文字层，其中包含所有目标文字，且目标文字的任意两个都不互斥，则该层次的值对应于启发式的值。该启发函数是可纳的，且比最大层次启发式更具信息。

3.4.3 程序实现代码框架

本实验代码框架来自GitHub上对《人工智能：现代方法》(第4版)(该书作者为斯图尔特·罗索(Stuart Russell)和彼得·诺维格(Peter Norvig))中规划问题实验内容的实现(aimacode)，根据需要选取其中与本实验相关的代码文件，文件名及主要内容见表3.6。

表3.6 本实验代码框架包含的文件名及内容

编 号	文 件 名	内 容
1	util.py	定义运行搜索算法所需的数据结构
2	search.py	实现常用搜索算法
3	logic.py	定义逻辑相关定义、运算、归结等
4	planning.py	定义规划问题中的动作

1. 航空运输问题类的定义

载入所需的库和类定义，具体代码如下。

```
from aimacode.logic import PropKB
from aimacode.planning import Action
from aimacode.search import (Node, Problem,)
from aimacode.utils import expr
from lp_utils import (FluentState, encode_state, decode_state,)
from my_planning_graph import PlanningGraph
from functools import lru_cache
```

定义航空运输问题类AirCargoProblem，具体代码如下。

```
class AirCargoProblem(Problem):
    def __init__(self, cargos, planes, airports, initial: FluentState, goal: list):
        """
        初始化AirCargoProblem类。
        :param cargos: 货物列表,字符型数据列表。
        :param planes: 飞机列表,字符型数据列表。
        :param airports: 机场列表,字符型数据列表。
        :param initial: 初始状态,FluentState类对象实例。
        :param goal: 目标状态列表。
        """
        # 将初始状态的正负条件合并为状态映射表
        self.state_map = initial.pos + initial.neg
        # 将初始状态编码为字符串形式
        self.initial_state_TF = encode_state(initial, self.state_map)
        # 调用父类Problem的初始化方法
        Problem.__init__(self, self.initial_state_TF, goal=goal)
        # 初始化货物、飞机和机场列表
        self.cargos = cargos
        self.planes = planes
        self.airports = airports
        # 获取所有可能的动作列表
        self.actions_list = self.get_actions()
```

```
    def get_actions(self):
        """
        获取所有可能的动作,包括装载、卸载和飞行动作。
        :return: 动作列表。
        """
        def load_actions():
            """生成所有装载动作"""
            loads = []
            for cargo in self.cargos:
                for plane in self.planes:
                    for airport in self.airports:
                        # 装载动作的前提条件: 货物和飞机都在同一机场
                        precond_pos = [expr("At({}, {})".format(cargo, airport)),
                                       expr("At({}, {})".format(plane, airport))]
                        precond_neg = []
                        # 装载动作的效果: 货物被装载到飞机上
                        effect_add = [expr("In({}, {})".format(cargo, plane))]
                        # 装载动作的效果: 货物不再在机场
                        effect_rem = [expr("At({}, {})".format(cargo, airport))]
                        # 创建装载动作对象
                        load = Action(expr("Load({}, {}, {})".format(cargo, plane,
airport)),
                                      [precond_pos, precond_neg],
                                      [effect_add, effect_rem])
                        loads.append(load)
            return loads

        def unload_actions():
            """生成所有卸载动作"""
            unloads = []
            for cargo in self.cargos:
                for plane in self.planes:
                    for airport in self.airports:
                        # 卸载动作的前提条件: 货物在飞机上,且飞机在机场
                        precond_pos = [expr("In({}, {})".format(cargo, plane)),
                                       expr("At({}, {})".format(plane, airport))]
                        precond_neg = []
                        # 卸载动作的效果: 货物被卸载到机场
                        effect_add = [expr("At({}, {})".format(cargo, airport))]
                        # 卸载动作的效果: 货物不再在飞机上
                        effect_rem = [expr("In({}, {})".format(cargo, plane))]
                        # 创建卸载动作对象
                        load = Action(expr("Unload({}, {}, {})".format(cargo, plane,
airport)),
                                      [precond_pos, precond_neg],
                                      [effect_add, effect_rem])
                        unloads.append(load)
            return unloads

        def fly_actions():
            """生成所有飞行动作"""
            flys = []
            for fr in self.airports:
                for to in self.airports:
```

```
                    if fr != to:
                        for p in self.planes:
                            # 飞行动作的前提条件：飞机在起飞机场
                            precond_pos = [expr("At({}, {})".format(p, fr))]
                            precond_neg = []
                            # 飞行动作的效果：飞机到达目标机场
                            effect_add = [expr("At({}, {})".format(p, to))]
                            # 飞行动作的效果：飞机不再在起飞机场
                            effect_rem = [expr("At({}, {})".format(p, fr))]
                            # 创建飞行动作对象
                            fly = Action(expr("Fly({}, {}, {})".format(p, fr, to)),
                                         [precond_pos, precond_neg],
                                         [effect_add, effect_rem])
                            flys.append(fly)
            return flys

        # 返回所有装载、卸载和飞行动作
        return load_actions() + unload_actions() + fly_actions()

    def actions(self, state: str) -> list:
        """
        获取当前状态下所有可能的动作。
        :param state: 当前状态(字符串形式)。
        :return: 可能的动作列表。
        """
        possible_actions = []
        # 创建命题知识库
        kb = PropKB()
        # 将当前状态解码并添加到知识库中
        kb.tell(decode_state(state, self.state_map).pos_sentence())
        # 遍历所有动作,检查是否满足前提条件
        for action in self.actions_list:
            is_possible = True
            # 检查正前提条件
            for clause in action.precond_pos:
                if clause not in kb.clauses:
                    is_possible = False
            # 检查负前提条件
            for clause in action.precond_neg:
                if clause in kb.clauses:
                    is_possible = False
            # 如果动作满足前提条件,则添加到可能的动作列表中
            if is_possible:
                possible_actions.append(action)
        return possible_actions

    def result(self, state: str, action: Action):
        """
        执行动作后返回新的状态。
        :param state: 当前状态(字符串形式)。
        :param action: 要执行的动作。
        :return: 新状态(字符串形式)。
        """
        # 创建新的状态对象
```

```
            new_state = FluentState([], [])
            # 解码当前状态
            old_state = decode_state(state, self.state_map)
            # 处理正条件
            for fluent in old_state.pos:
                if fluent not in action.effect_rem:
                    new_state.pos.append(fluent)
            for fluent in action.effect_add:
                if fluent not in new_state.pos:
                    new_state.pos.append(fluent)
            # 处理负条件
            for fluent in old_state.neg:
                if fluent not in action.effect_add:
                    new_state.neg.append(fluent)
            for fluent in action.effect_rem:
                if fluent not in new_state.neg:
                    new_state.neg.append(fluent)
            # 将新状态编码为字符串形式
            return encode_state(new_state, self.state_map)

        def goal_test(self, state: str) -> bool:
            """
            检查当前状态是否满足目标状态。
            :param state: 当前状态(字符串形式)。
            :return: 如果满足目标状态则返回 True,否则返回 False。
            """
            # 创建命题知识库
            kb = PropKB()
            # 将当前状态解码并添加到知识库中
            kb.tell(decode_state(state, self.state_map).pos_sentence())
            # 检查目标状态是否全部满足
            for clause in self.goal:
                if clause not in kb.clauses:
                    return False
            return True
```

2. 针对具体问题的设计

本例设置了由简单到复杂的三个航空运输规划问题,以便考查算法在解决不同难度问题时的性能表现。

1) 问题 1 设计

在问题 1 中,设计货物为 C1 和 C2,飞机为 P1 和 P2,机场为 SFO 和 JFK。初始状态下,C1 在 SFO 机场,C2 在 JFK 机场,P1 停靠在 SFO 机场,P2 停靠在 JFK 机场;目标状态为:货物 C1 在 JFK 机场,C2 在 SFO 机场。问题定义的代码如下。

```
def air_cargo_p1() -> AirCargoProblem:
    cargos = ['C1', 'C2']
    planes = ['P1', 'P2']
    airports = ['JFK', 'SFO']
    pos = [expr('At(C1, SFO)'),
```

```
            expr('At(C2, JFK)'),
            expr('At(P1, SFO)'),
            expr('At(P2, JFK)'),
            ]
    neg = [expr('At(C2, SFO)'),
            expr('In(C2, P1)'),
            expr('In(C2, P2)'),
            expr('At(C1, JFK)'),
            expr('In(C1, P1)'),
            expr('In(C1, P2)'),
            expr('At(P1, JFK)'),
            expr('At(P2, SFO)'),
            ]
    init = FluentState(pos, neg)
    goal = [expr('At(C1, JFK)'),
             expr('At(C2, SFO)'),
             ]
    return AirCargoProblem(cargos, planes, airports, init, goal)
```

2）问题 2 设计

在问题 2 中，设计货物为 C1、C2 和 C3，飞机为 P1、P2 和 P3，机场为 SFO、JFK 和 ATL。初始状态下，C1 在 SFO 机场，C2 在 JFK 机场，C3 在 ATL 机场，P1 停靠在 SFO 机场，P2 停靠在 JFK 机场，P3 停靠在 ATL 机场；目标状态为：货物 C1 在 JFK 机场，C2 在 SFO 机场，C3 在 SFO 机场。问题定义的代码如下。

```
def air_cargo_p2() -> AirCargoProblem:

    cargos = ['C1', 'C2', 'C3']
    planes = ['P1', 'P2','P3']
    airports = ['JFK', 'SFO', 'ATL']
    pos = [expr('At(C1,SFO)'),expr('At(C2,JFK)'),expr('At(C3,ATL)'),
           expr('At(P1,SFO)'),expr('At(P2,JFK)'),expr('At(P3,ATL)')]

    neg = [expr('At(C1,JFK)'),expr('At(C1,ATL)'),
           expr('At(C2,SFO)'), expr('At(C2,ATL)'),
           expr('At(C3,JFK)'), expr('At(C3,SFO)'),

           expr('In(C1,P1)'), expr('In(C1,P2)'), expr('In(C1,P3)'),
           expr('In(C2,P1)'), expr('In(C2,P2)'), expr('In(C2,P3)'),
           expr('In(C3,P1)'), expr('In(C3,P2)'), expr('In(C3,P3)'),

           expr('At(P1,JFK)'),expr('At(P1,ATL)'),
           expr('At(P2,SFO)'), expr('At(P2,ATL)'),
           expr('At(P3,SFO)'), expr('At(P3,JFK)'),
           ]
    init = FluentState(pos, neg)
    goal = [expr('At(C1, JFK)'),
            expr('At(C2, SFO)'),
            expr('At(C3, SFO)')
            ]
    return AirCargoProblem(cargos, planes, airports, init, goal)
```

3）问题 3 设计

在问题 3 中，设计货物为 C1、C2、C3 和 C4，飞机为 P1 和 P2，机场为 SFO、JFK、ATL 和 ORD。初始状态下，C1 在 SFO 机场，C2 在 JFK 机场，C3 在 ATL 机场，C4 在 ORD 机场，P1 停靠在 SFO 机场，P2 停靠在 JFK 机场；目标状态为货物 C1 在 JFK 机场，C2 在 SFO 机场，C3 在 JFK 机场，C4 在 ORD 机场。问题定义的代码如下。

```
def air_cargo_p3() -> AirCargoProblem:
cargos = ['C1', 'C2', 'C3','C4']
    planes = ['P1', 'P2']
    airports = ['JFK', 'SFO', 'ATL','ORD']
    pos = [expr('At(C1,SFO)'),expr('At(C2,JFK)'),expr('At(C3,ATL)'),expr('At(C4,ORD)'),
           expr('At(P1,SFO)'),expr('At(P2,JFK)')]
    neg = [
           expr('At(C1,JFK)'),expr('At(C1,ATL)'),expr('At(C1,ORD)'),
           expr('In(C1,P1)'), expr('In(C1,P2)'),

        expr('At(C2,SFO)'), expr('At(C2,ATL)'), expr('At(C2,ORD)'),
        expr('In(C2,P1)'), expr('In(C2,P2)'),

        expr('At(C3,JFK)'), expr('At(C3,SFO)'), expr('At(C3,ORD)'),
        expr('In(C3,P1)'), expr('In(C3,P2)'),

        expr('At(C4,JFK)'), expr('At(C4,SFO)'), expr('At(C4,ATL)'),
        expr('In(C4,P1)'), expr('In(C4,P2)'),

           expr('At(P1,JFK)'),expr('At(P1,ATL)'),expr('At(P1,ORD)'),
           expr('At(P2,SFO)'),expr('At(P2,ATL)'),expr('At(P2,ORD)'),
           ]
    init = FluentState(pos, neg)
    goal = [expr('At(C1, JFK)'),
            expr('At(C3, JFK)'),
            expr('At(C2,SFO)'),
            expr('At(C4,SFO)')
            ]
    return AirCargoProblem(cargos, planes, airports, init, goal)
```

3. 启发函数的定义

对于航空运输问题，设计如下 3 种启发函数。

1）启发函数 $h_1(n)$

定义该启发函数的具体代码如下。该启发函数不是真正的启发式，其取值恒为 1。

```
  def h_1(self, node: Node):
      # 这是一个启发式函数 h_1，它总是返回一个常数 1
      # 注意：这并不是一个真正的启发式函数，因为它不依赖于节点的状态
      h_const = 1
      return h_const
```

2）启发函数 $h_2(n)$

该启发函数为基于规划图的层次和启发式，具体代码如下。

```
def h_pg_levelsum(self, node: Node):
    # 该启发式函数使用了规划图(Planning Graph)的层次和启发式(h_levelsum)
    # 它通过构建规划图并计算层次和来估计从当前节点到目标节点的代价
    # 注意: 该函数依赖于已实现的 PlanningGraph 类

    # 创建规划图对象,传入当前问题的定义和节点的状态
    pg = PlanningGraph(self, node.state)

    # 调用规划图的 h_levelsum 方法,计算层次和启发式值
    pg_levelsum = pg.h_levelsum()

    # 返回计算得到的层次和启发式值
    return pg_levelsum
```

3) 启发函数 $h_3(n)$

该启发式为忽略动作前置条件启发式,具体代码如下。

```
def h_ignore_preconditions(self, node: Node):
    # 该启发式函数为"忽略动作前置条件启发式"
    # 它通过比较当前状态和目标状态的正子句(positive clauses)来计算启发式值
    # 启发式值的计算方式是: 目标状态的正子句数量减去当前状态中已满足的目标子句数量

    # 解码当前节点的状态,获取当前状态的正子句
    state_pos_clauses = decode_state(node.state, self.state_map).pos

    # 获取目标状态的正子句
    goal_pos_clauses = self.goal

    # 初始化匹配计数器
    num_matches = 0

    # 获取目标状态中正子句的总数
    num_pos_goal_clauses = len(goal_pos_clauses)

    # 遍历目标状态的每个正子句
    for goal_pos_clause in goal_pos_clauses:
        # 遍历当前状态的每个正子句
        for state_pos_clause in state_pos_clauses:
            # 如果当前状态的子句与目标状态的子句匹配,则增加匹配计数
            if goal_pos_clause == state_pos_clause:
                num_matches += 1

    # 返回启发式值: 目标状态的正子句数量减去已匹配的子句数量
    return num_pos_goal_clauses - num_matches
```

4. 算法性能对比

选择不同的航空运输问题,利用搜索算法和启发函数对问题进行求解,对比各种算法和启发式的效果。

在该项目的 run_search.py 文件中,通过 main()函数和 run_search()函数实现了规划问题与搜索算法之间的关联。在具体操作时,可以使用命令"python run_search.py -p

参数 1 -s 参数 2"来调用，其中，参数 1 代表问题选项，1～3 分别对应航空运输问题 1～3；参数 2 代表搜索算法选项，其值与搜索算法的对应关系如表 3.7 所示。

表 3.7　参数 2 的值与算法的对应关系

参数 2 的值	对 应 算 法
1	宽度优先搜索
2	宽度优先树搜索
3	深度优先图搜索
4	深度优先搜索
5	代价优先搜索
6	使用启发函数 $h=1$ 的递归最佳优先搜索
7	使用启发函数 $h=1$ 的贪婪搜索
8	使用启发函数 $h=1$ 的 A^* 搜索
9	使用忽略动作前置条件启发式的 A^* 搜索
10	使用规划图层次和启发式的 A^* 搜索

例如，使用启发函数 $h=1$ 的 A^* 搜索算法解决问题 3 的命令如下。

```
python run_search.py -p 3 -s 8
```

3.4.4　实验结果分析

对于给定的 3 个航空运输问题，在不同的搜索算法和启发函数组合下得到的规划结果对比如表 3.8 所示。

表 3.8　不同搜索算法和启发函数组合下的规划结果对比

问题编号	代价优先搜索		A^* 搜索（启发函数恒为 1）		A^* 搜索（忽略动作前置条件）	
	扩展次数	扩展节点数	扩展次数	扩展节点数	扩展次数	扩展节点数
1	55	224	55	224	41	170
2	4852	44030	4852	44030	1450	13303
3	18235	159716	18235	159716	5040	44944

可以发现，对于不同的规划问题，代价优先搜索和采用启发函数恒为 1 的 A^* 搜索实际上是等价的。在解决规划问题时，采用忽略前置条件启发式的 A^* 搜索算法，可有效减少扩展次数和产生的节点数。这一结果不仅验证了启发函数设计的合理性，也为后续的算法优化提供了重要参考。拓展实验可以考虑如何结合实际问题特点，设计更为高效的启发函数，以进一步提升算法的整体性能。

3.5　本章小结

本章主要介绍了状态空间搜索的基本概念、方法和典型应用。状态空间搜索是一种系统性探索问题空间并进行求解的方法，通过状态空间法对问题进行表示，并使用图搜索

算法在问题表示的基础上进行求解，以找到从初始状态到达目标状态的一条最优（或较优）路径。本章首先阐述了状态空间搜索的基础概念和方法，然后通过路径规划问题、八数码问题和规划问题的求解，展示了盲目搜索算法和启发式搜索算法的应用过程。通过对比不同搜索算法的性能，强调了启发式搜索算法在求解效率上的优势，特别是 A^* 搜索算法在路径规划和八数码问题求解中均有优异表现。

同时，针对航空运输这一特定规划问题，设计了不同的启发函数，并通过实验对比了不同搜索算法和启发函数组合下的规划结果。实验结果表明，采用忽略前置条件启发式的 A^* 搜索算法在减少扩展次数和产生的节点数方面具有明显优势，这为后续的算法优化提供了重要参考。通过本章的学习，有助于深入了解状态空间搜索的基本概念、方法和应用，并掌握如何在实际问题中设计和选择有效的启发函数，以提升算法的整体性能。