通过搜索进行问题求解

一、问题求解智能体

智能体可以执行以下4个阶段的问题求解过程：

- **目标形式化**（goal formulation）：目标通过限制智能体的目的和需要考虑的动作来组织其行为
- **问题形式化**（problem formulation）：智能体刻画实现目标所必需的状态和动作——进而得到这个世界中与实现目标相关的部分所构成的抽象模型
- **搜索**（search）：在真实世界中采取任何动作之前，智能体会在其模型中模拟一系列动作，并进行搜索，直到找到一个能到达目标的动作序列。这样的序列称为**解**（solution）。
- **执行**（execution）：现在智能体可以执行解中的动作，一次执行一个动作。

搜索问题和解

搜索**问题**（problem）的形式化定义如下

- **状态空间**（state space）：可能的环境**状态**（state）的集合
- **初始状态**（initial state）
- 一个或多个**目标状态**（goal state）的集合
- **行动**（action）：在某一状态s可以采取的行动
- **转移模型**（transition model）给出在状态s中执行动作a所产生的状态
- **动作代价函数**（action cost function）给出在状态s中执行动作a从而转移到状态s'的数值代价。

问题形式化

模型（model）：一种抽象的数学描述

抽象（abstraction）：从表示中剔除细节的过程

抽象层级（level of abstraction）：选择的抽象状态和动作对应于大量具体的世界状态和动作序列

二、问题示例

标准化问题

真实世界问题

三、搜索算法

树搜索算法

该算法通过**扩展**（expand）节点，从初始状态形成各条路径，并试图找到一条可以达到某个目标状态的路径。搜索树中的每个**节点**（node）对应于状态空间中的一个**状态**（state），搜索树中的边对应于动作，树的根对应于问题的初始状态

状态（state）：世界中一个物理配置的表示
节点（node）：一个数据结构，是搜索树的组成部分，包括父节点，子节点，深度和路径代价

最佳优先搜索

在每次迭代中，选择边界上具有最小f(n)值的一个节点，如果它的状态是目标状态，则返回这个节点，否则调用Expand生成子节点。对于每个子节点，如果之前未到达过该子节点，则将其添加到边界；如果到达该子节点的当前路径的代价比之前任何路径都要小，则将其重新添加到边界。该算法要么返回failure，要么返回一个节点（表示一条通往目标的路径）

搜索数据结构

搜索算法需要一个数据结构来跟踪搜索树。树中的节点（node）由一个包含4个组成部分的数据结构表示

- `node.State`：节点对应的状态。
- `node.Parent`：父节点，即树中生成该节点的节点。
- `node.Action`：父节点生成该节点时采取的动作。
- `node.Path-Cost`：从初始状态到此节点的路径总代价。在数学公式中，一般使用$g(node)$表示Path-Cost。

我们需要一个数据结构来存储边界。一个恰当的选择是某种队列（queue），因为边界上的操作有以下几个：

- `Is-Empty( frontier)`：返回true当且仅当边界中没有节点。
- `Pop( frontier)`：返回边界中的第一个节点并将它从边界中删除。
- `Top( frontier)`：返回（但不删除）边界中的第一个节点。
- `Add(node, frontier)`：将节点插入队列中的适当位置

搜索算法使用了3种不同类型的队列：

- 优先队列（priority queue）首先弹出根据评价函数f计算得到的代价最小的节点。它被用于最佳优先搜索
- FIFO队列（FIFO queue），即先进先出队列（first-in-first-out queue），首先弹出最先添加到队列中的节点；它被用于广度优先搜索
- LIFO队列（LIFO queue），即后进先出队列（last-in-first-out queue），也称为栈（stack），首先弹出最近添加的节点；它被用于深度优先搜索

冗余路径

如果搜索算法检查冗余路径，我们称之为**图搜索**（graph search）；否则，称之为**树搜索**（tree-like search）

问题求解性能评估

- **完备性**（completeness）：当存在解时，算法是否能保证找到解，当不存在解时，是否能保证报告失败？
- **代价最优性**（cost optimality）：它是否找到了所有解中路径代价最小的解？
- **时间复杂性**（time complexity）：找到解需要多长时间？可以用秒数来衡量，或者更抽象地用状态和动作的数量来衡量。
- **空间复杂性**（space complexity）：执行搜索需要多少内存？

复杂性可以用3个量来衡量：

- d，最优解的深度（depth）或动作数
- m，任意路径的最大动作数
- b，需要考虑的节点的分支因子（branching factor）或后继节点数

四、无信息搜索策略

无信息搜索算法不提供有关某个状态与目标状态的接近程度的任何线索

广度优先搜索

- 优先扩展最浅的未被扩展的节点
- 边界（frontier）可以实现为一个FIFO队列，即,新节点（总是比其父节点更深）进入队列的队尾，而旧节点，即比新节点浅的节点，首先被扩展
- 完备性：是（如果b 是有限的）
- 时间：$1+b+b^2+b^3+\ldots+b^d=O(b^d)$（假设解的深度为d）空间：$O(b^d)$（所有节点都存储在内存中）
- 代价最优：是（如果所有动作都具有相同代价，否则不一定）

对广度优先搜索来说，内存需求是一个比执行时间更严重的问题

一般来说，除了最小的问题实例，指数级复杂性的搜索问题无法通过无信息搜索求解

Dijkstra算法或一致代价搜索

- 优先扩展代价最小的未被扩展的节点
- 边界（frontier）可以实现为一个按路径代价排序的队列，最浅层的优先
- 完备性：是（假设所有动作的代价>0 ）
- 时间：最坏情况下$O(b^{C^*/\varepsilon})$（这里C^*是最优解的代价）
- 空间：最坏情况下$O(b^{C^*/\varepsilon})$
- 代价最优：是（节点以路径代价增加的顺序扩展）

深度优先搜索与内存问题

- 优先扩展最深的未被扩展的节点
- 边界（frontier）可以实现为一个LIFO队列，即后进先出
- 完备性：否（无限状态空间和有环状态空间，而对于树型的有限状态空间，算法是有效且完备的）
- 时间：$O(b^m)$（m是树的最大深度，如果解密集的，可能要比广度优先更快）
- 空间：$O(bm)$（线性空间）
- 代价最优：否，会找到最左边的解，而不管深度或者代价如何

回溯搜索（backtracking search）是深度优先搜索的一种变体，它使用的内存更少

深度受限和迭代加深搜索

深度限制为 l 的深度优先搜索，即将深度 l 上的所有节点视为其不存在后继节点

- 时间复杂性为$O(bl)$
- 空间复杂性为$O(bl)$

迭代加深搜索（iterative deepening search）解决了如何选择一个合适的 l 的问题

- 完备性：是
- 时间：$(d+1)b^0+db^1+(d-1)b^2+\ldots+b^d=O(b^d)$
- 空间：$O(b^d)$

- 代价最优：是（如果动作代价为1，能够被修改来探索代价一致树）
- 存在解时，时间复杂性为$O(b^d)$，不存在解时，时间复杂性为$O(b^m)$
- 当搜索状态空间大于内存容量而且解的深度未知时，迭代加深搜索是首选的无信息搜索方法

双向搜索

同时从初始状态正向搜索和从目标状态反向搜索，直到这两个搜索相遇

无信息搜索算法对比

指标	广度优先	一致代价	深度优先	深度受限	迭代加深	双向（如适用）
完备性	是[1]	是[1,2]	否	否	是[1]	是[1,4]
代价最优	是[3]	是	否	否	是[3]	是[3,4]
时间复杂性	$O(b^d)$	$O(b^{1+\lfloor C^*/\varepsilon \rfloor})$	$O(b^m)$	$O(b^\ell)$	$O(b^d)$	$O(b^{d/2})$
空间复杂性	$O(b^d)$	$O(b^{1+\lfloor C^*/\varepsilon \rfloor})$	$O(bm)$	$O(b\ell)$	$O(bd)$	$O(b^{d/2})$

注：[1] 如果 b 是有限的且状态空间要么有解要么有限，则算法是完备的。[2] 如果所有动作的代价都$\geq \varepsilon > 0$，算法是完备的。[3] 如果动作代价都相同，算法是代价最优的。[4] 如果两个方向均使用广度优先搜索或一致代价搜索

五、有信息（启发式）搜索策略

- **有信息搜索（informed search）策略**使用关于目标位置的特定领域线索来比无信息搜索策略更有效地找到解。
- 线索以**启发式函数**（heuristic function）的形式出现，记为h(n)：h(n) = 从节点n的状态到目标状态的最小代价路径的代价估计值

贪心最佳优先搜索（greedy best-first search）

- 首先扩展h(n)值最小的节点，即看起来最接近目标的节点，因为这样可能可以更快找到解。
- 评价函数$f(n) = h(n)$
- 完备性：否（可能会卡在循环当中，对于有着重复状态检测的有限状态空间是完备的）
- 时间：$O(b^m)$（一个好的启发式函数可以使复杂性大大降低）
- 空间：$O(b^m)$（所有节点都存储在内存中）
- 代价最优：否（如果所有动作都具有相同代价，否则不一定）

A^*搜索

- 主要思想：避免扩展代价已经很高的路径
- 评价函数$f(n) = g(n) + h(n)$
    - $g(n)$ = 从初始节点到节点n的路径代价
    - $h(n)$ = 从节点n的状态到目标状态的最小代价路径的代价估计值
    - $f(n)$ = 经过n到一个目标状态的最优路径的代价估计值
    - 对于可容许的启发式（admissible heuristic）函数，A^*搜索是代价最优的，即$h(n) \leq h^*(n)$
      这里 h*(n) 经过节点n到目标状态的真实代价. ($h(n) \geq 0$, 对于任意目标G，$h(G) = 0$.)

搜索等值线

- 一致代价搜索中，等值线将以初始状态为圆心呈“圆形”向各个方向均匀扩展
- 对于具有好的启发式函数的A*搜索，等值线将朝目标状态延伸，并在最优路径周围收敛变窄

满意搜索：不可容许的启发式函数与加权A^*搜索

$f(n) = g(n) + W \cdot h(n)$，这里 $W > 1$；

内存受限搜索

1. **束搜索**（beam search）对边界的大小进行了限制：只保留具有最优f值的k个节点，放弃其他已扩展节点
2. **迭代加深A^*搜索**（iterative-deepening A search，IDA*）
3. **递归最佳优先搜索**（recursive best-first search，RBFS）

双向启发式搜索

对于正向搜索（以初始状态作为根节点）中的节点，我们用$f_F(n) = g_F(n) + h_F(n)$作为评价函数；对于反向搜索（以某个目标状态作为根节点）中的节点，我们用$f_B(n) = g_B(n) + h_B(n)$作为评价函数

六、启发式函数

启发式函数的准确性对性能的影响

如果对于任意节点n，$h_2(n) \geq h_1(n)$，那么 h_2 占优于 h_1 并且更有利于搜索。

给定任意可容许的启发函数 h_a, h_b,

$h(n) = \max(h_a(n), h_b(n))$ 也是可容许的并且占有于 h_a, h_b

从松弛问题出发生成启发式函数

- 松弛问题中最优解的代价可以作为原问题的一个可容许的启发式函数
- 关键点：松弛问题最优解的代价不大于原问题最优解的代价