问题求解Agent：基于目标Agent的一种

无信息的搜索算法：除了问题定义本身外没有任何其他信息

搜索算法

有信息的搜索算法：利用给定的知识引导能够更有效地找到解

环境任务简化：限定问题的解是一组有固定顺序的行动

**3.1问题求解Agent**

如果 **Agent** 能采纳一个目标（goal) 并试图去满足它，最大化性能度量的问题就可能会简化。

目标形式化：考虑状态集合中目标能被满足的状态子集

问题形式化：在给定目标下确定需要考虑哪些行动和状态的过程

搜索：寻找这样的行动序列

执行

**3.1.1良定义的问题及解**

问题组成的形式化描述：

1. Agent的初始状态
2. 描述Agent的可能行动 定义了问题的状态空间
3. 对每个行动的描述：转移模型
4. 目标测试
5. 路径耗散函数（不一定需要，如八皇后问题）

**上述元素通常组织在一起成为一个数据结构，并以此作为问题求解算法的输入**

**问题的解就是从初始状态到目标状态的一组行动序列。**

**解的质量由路径耗散函数度量，所有解里路径耗散值最小的解即为最优解。**

**3.1.2问题的形式化**

精确定义合适的抽象层次

**3.2问题实例**

**增量形式化**（incremental formulation)：包括了算符来增加状态描述，从空状态开始；对于八皇后问题，即每次行动添加一个皇后 到状态中去。

**完整状态形式化**（complete-state formulation) ，8 个皇后都在棋盘上并且不断移动。

**3.3通过搜索求解**

**3.3.1最佳优先搜索BFS**

**开放列表（Open List）：优先队列，用于存储待探索的节点，按​​启发式函数值​​排序，优先级最高的节点（值最小）最先被处理**

**​​封闭列表（Closed List）​​：记录已探索的节点，避免重复访问，一般用哈希表**

1. **初始状态​​**
   * **将起始节点加入开放列表，并记录其启发式函数值（例如曼哈顿距离、欧氏距离等）**
2. **​​节点选择与扩展​​**
   * **​​取出优先级最高的节点​​：从开放列表中取出启发式值最小的节点作为当前节点**
   * **​​检查终止条件​​：若当前节点是目标节点，结束搜索并返回路径；否则继续扩展**
3. **​​子节点生成与处理​​**
   * **​​生成子节点​​：遍历当前节点的所有邻接节点（或子节点）**
   * **​​子节点状态判断​​：**
     + **​​未在开放/封闭列表中​​：计算其启发式值，加入开放列表**
     + **​​已在开放列表中​​：若新路径的启发式值更优，则更新该节点的优先级**
     + **​​已在封闭列表中​​：若新路径更优，则将其移回开放列表并更新优先级**
4. **​​更新列表​​**
   * **将当前节点移入封闭列表，标记为已探索**
5. **​​循环终止条件​​**
   * **重复上述步骤，直到找到目标节点或开放列表为空（搜索失败）**

**优点​​：**

**相比盲目搜索（如BFS、DFS），显著减少搜索空间，提升效率。**

**​​缺点​​：**

**​​不完备性​​：若启发式函数设计不当，可能无法找到解。**

**​​非最优性​​：可能找到可行解但非最短路径（与A算法不同，A结合了实际代价g(n)与启发式代价h(n)）**

**3.3.2搜索数据结构**

搜索树构造过程的数据结构

对树中的**每个节点**，定义**四个元素**：

1. State：状态空间中的状态
2. Parent指针：搜索树中产生该结点的结点（即父结点）
3. Action：父结点生成该结点时所采取的行动
4. Path-Cost：代价，一般用g(n)表示，指**从初始状态**到达该结点的路径消耗

**3.3.4问题求解算法的性能**

**完备性（有解时能保证找到解）最优性、时间复杂度、空间复杂度**

复杂度通常由下列三个量来表达：

b, 分支因子：任何结点的最多后继数；

d, 目标结点所在的**最浅**的**深度**（如从根结点到目标状态的步数)

m，状态空间中**任何路径的最大长度**。

评价搜索算法的有效性，我们可以只考虑**搜索代价**，或可以使用**总代价=搜索代价+解代价**

**3.4无信息搜索策略**

**3.4.1广度优先搜索**

先扩展根结点，接着扩展根结点的所有后继，然后再扩展它们的后继，依此类推。一般地，**在下一层的任何结点扩展之前，搜索树上本层深度的所有结点都应该已经扩展过。**

**每次总是扩展深度最浅的结点：将边缘结点组织成FIFO队列实现**

**目标测试**在结点**生成的时候**，而非被扩展的时候

目标结点一经生成，它一定是最浅的目标结点（不一定最优）

**如果**路径代价是**基于结点深度**的**非递减函数**，宽度优先搜索是**最优的**。

时间复杂度***O(b^d)*** 空间复杂度***O(b^d)*** ，b:分支因子，d:解的深度

**3.4.2一致代价搜索**

一致代价搜索对任何单步代价函数都是**最优的**

一致代价搜索（uniform-cost search) 扩展的是路径消耗最小的结点，而非深度最浅的结点，可以通过将边缘结点集组织成按**g**值排序的队列来实现，结点的g值等于初始结点到该节点的最小代价。

与宽度优先搜索的不同：

1. 按路径代价对队列进行排序
2. **目标检测**应用于结点被选择**扩展时**，而不是在结点生成的时候进行(为了避免选择次优路径)
3. 对解路径的步数并不关心，只关心路径总代价。

**如果存在零代价行动就可能陷入死循环 如果每一步的代价都大于等于某个小的正值常数，那么一致代价搜索是完备的。**

空间复杂度***O(bm)*** ，b:空间分支因子 m:最大深度

**3.4.3深度优先搜索**

深度优先搜索（(kpth-first search) 总是扩展搜索树的**当前边缘结点集**中**最深**的结点

搜索很快推进到搜索树的最深层，那里的结点**没有后继**。当那些结点扩展完之后，就**从边缘结点集中去掉**，然后搜索算法**回溯到下一个还有未扩展后继的深度稍浅的结点。**

深度优先搜索算法的效率严重依赖于使用的是**图搜索**（避免重复状态和冗余路径，在有限空间状态集**完备**）还是**树搜索**（**不完备**）。

避免重复状态和冗余路径的图搜索，在有限状态空间是完备的，因为它至多扩展所有结点。而树搜索，则不完备

深度优先搜索**不是最优的**（找到目标结点即返回，路径不一定是最佳的）

变形：**回溯搜索**

在回溯搜索中，每次只产生一个后继而不是生成所有后继；每个被部分扩展的结点要记住下一个要生成的结点。这样，内存只需要 O(m)而不是 O(bm)。

**3.4.4深度受限搜索**

设置界限l，深度为 l的结点被当作没有后继对待。

深度受限搜索可能因为两种失败而终止标准：***failure***返回值指示无解；***cutoff***值指示在深度界限内无解

**3.4.5迭代加深的深度优先搜索**

**不断地增大深度限制**，首先为0,接着为1，然后为2, 依此类推直到找到目标。当深度界限达到d，即最浅的目标结点所在深度时，就能找到目标结点。

空间复杂度：***O(bd)*** 时间复杂度为 ***〇(b^d)*** （与宽度优先搜索相近）

当**分支因子有限**时是该搜索算法是**完备**的

当路径代价是**结点深度**的**非递减函数**时该算法是**最优**的

如果要避免一部分重复状态的生成，可以混合使用两种搜索算法，先用宽度优先搜索直到有效内存耗尽，然后对边缘集中的所有结点应用迭代加深的深度优先搜索。

**3.4.6双向搜索**

时间复杂度是 ***O(b^d/2)***。空间复杂度也是 ***O(b^d/2)***。

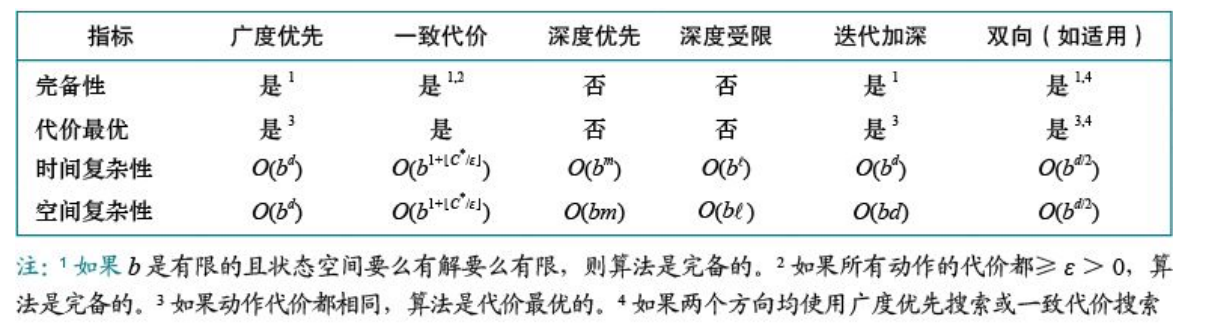
对于最佳优先双向搜索：

若评价函数是路径代价，则是最优的

若不是，则找到的解可能不是最优解，

即使两个方向采用的都是宽度优先搜索；保证最短路径还需要额外搜索

**3.4.7无信息搜索策略对比**



**3.5有信息（启发式）的搜索策略**

评价函数f(n)：代价估计，评估值最低的结点被选择首先进行扩展

启发函数h(n)：结点n到目标结点的最小代价路径的代价估计值

假设启发式信息是任一非负的由问题而定的函数，有一个约束：**若**

**n是目标结点，则h(n)= 0**

**3.5.1贪婪最佳优先搜索**

试图扩展离目标最近的结点，理由是这样可能可以很快找到解。因此，它只用启发式信息，即f(n)=h(n)

图搜索在有限状态空间中是完备的，但在无限状态空间中是不完备的

时间和空间复杂度为O(|V|)，使用好的启发式函数可以大大降低

**3.5.2 A\*搜索：缩小总评估代价**

结点评估：f(n)=g(n)+h(n),

g(n): 从开始结点到结点n的路径代价

h(n): 从结点n到目标结点的最小代价路径的估计值

因此f(n)= 经过结点n到一个目标状态的最优路径的估计代价值

A\*搜索是**完备的**，算法与一致代价搜索类似

**保证最优性的条件：可容许性和一致性**

**保证最优性的条件：可容许性和一致性**

1. h(n)是一个可容许启发式：它从**不会过高估计**到达目标的代价

故**f(n)=g(n)+h(n) <= 经过结点n的解的实际代价**

1. 启发式h(n)是一致的：**h(n) <= c(n, a, n') + h(n')**

**c(n, a, n')** ：结点n通过行动a生成n的后继结点n’的**实际代价**

**A\*算法的最优性**

如果**h(n)**是**可容许的**，那么A\*的**树搜索**版本是**最优的**；

如果**h(n)**是**一致的**，那么**图搜索**的A\*算法是**最优的**

缺点

时间复杂度是 ***O(b^ε^d)***

空间复杂度：极大，在内存中保留了所有已生成的结点

**3.5.2存储受限的启发式搜索**

迭代加深 A\*(IDA\*)算法

弊端：解路径的每次改变都对应于IDA\*中的一次迭代，并且可能

需要重新扩展已经遗忘的结点来重建最佳路径。

递归最佳优先搜索（RBFS）

类似于用目前能达到的最小值剪枝

如果启发式函数 A(/i)是可采纳的，那么 RBFS 算法是最优的

IDA\*和 RBFS 的问题在于它们**使用的内存过于小**了。

因为两个算法都**忘记了它们做过什么**， 所以算法终止时**有些状态可能重复扩多次**。更坏的是，图中的冗余路径会带来复杂度的潜在的指数级的增长

充分利用内存的算法：

MA\* (内存受限 A\*)

SMA\* (简化的 MA\*)

扩展最佳叶结点直到内存耗尽。就是说，要在搜索树中加入新结点就得抛弃一

个旧结点。SMA\*总是丢弃最差的叶结点——即f值最高的结点。

SMA\*把被遗忘结点的值回填给父结点。这样，被遗忘子树的祖先结点可以了解子树的最佳路径。

有了这个信息，当所有其他路径看来比被遗忘路径要差的时候，SMA\*可以重新生成该子树。换句话说，如果结点n的所有子孙结点都被遗忘了，我们不知道从n该走哪条路，但是我们知道从n去别处是否值得。

**3.6启发式函数**

一个约束：当n为目标状态时，h(n)=0

（1）h(n)是一个可容许启发式：它从**不会过高估计**到达目标的代价

故**f(n)=g(n)+h(n) <= 经过结点n的解的实际代价**

（2）启发式h(n)是一致的：**h(n) <= c(n, a, n') + h(n')**

**c(n, a, n')** ：结点n通过行动a生成n的后继结点n’的**实际代价**

有效分支因子