人工智能

搜索策略

主讲: 赵国亮

内蒙古大学电子信息工程学院

April 26, 2020

目录

搜索策略

搜索

搜索是人工智能中的一个基本问题, 并与推理密切相关, 搜索策略的优劣, 将直接影响到智能系统的性能与推理效率.

适用情况

不良结构或非结构化问题; 难以获得求解所需的全部信息; 更没有现成的算法可供求解使用.

搜索

依靠经验, 利用已有知识, 根据问题的实际情况, 不断寻找可利用知识 (过程知识和算法框架的构建知识), 从而构造一条代价最小的推理路线, 使问题得以解决的过程称为搜索.

搜索的类型

按是否使用启发式信息

- 盲目搜索: 按预定的控制策略进行搜索, 在搜索过程中获得的中间信息, 并不改变控制策略.
- 启发式搜索: 在搜索中加入了与问题有关的启发性信息, 使得搜索朝着正确的方向进行, 加速问题的求解过程, 迅速找到最优解.

按问题的表示方式

- 状态空间搜索: 用状态空间法来求解问题所进行的搜索.
- 与或树搜索: 用问题归约法来求解问题时所进行的搜索.

状态空间法

状态空间表示方法

- 状态 (State):
  是表示问题求解过程中每一步问题状况的数据结构, 它可表示为:

$$S_k = \{S_{k0}, S_{k1}, \cdots\} \tag{1}$$

  当对每一个分量都给以确定的值时, 就得到了一个具体的状态.
- 操作 (Operator)
  也称为算符, 它是把问题从一种状态变换为另一种状态的手段. 操作可以是一个机械步骤, 一个运算, 一条规则或一个过程. 操作可理解为状态集合上的一个函数, 它描述了状态之间的关系.

- 状态空间 (State space) 描述一个问题的全部状态以及这些状态之间的相互关系. 常用三元组表示为 (S, F, G), 其中, S 为问题的所有初始状态的集合; F 为操作的集合; G 为目标状态的集合.

状态空间问题求解

状态空间法求解问题的基本过程:

- 首先为问题选择适当的“状态”及“操作”的形式化描述方法;
- 然后从某初始状态出发, 每次使用一个“操作”, 递增地建立起操作序列, 直到达到目标状态为止; 由初始状态到目标状态所使用的算符序列就是问题的一个解.

例 2.1

二阶梵塔问题. 设有三根钢针, 它们的编号分别是 1 号、2 号和 3 号. 在初始情况下, 1 号钢针上穿有 A, B 两个金片, A 比 B 小, A 位于 B 的上面. 要求把这两个金片全部移到另一根钢针上, 而且规定每次只能移动一个金片, 任何时刻都不能使大的位于小的上面.

解: 设用 $S_k = S_{k0}, S_{k1}$ 表示问题的状态, 其中, S_{k0} 表示金片 A 所在的钢针号, S_{k1} 表示金片 B 所在的钢针号.

全部可能的问题状态共有以下 9 种:

$$S_0 = (1,1)\ S_1 = (1,2)\ S_2 = (1,3)\ S_3 = (2,1)\ S_4 = (2,2) \tag{2}$$

$$S_5 = (2,3)\ S_6 = (3,1)\ S_7 = (3,2)\ S_8 = (3,3) \tag{3}$$

- 问题的初始状态集合为 $S = \{S_0\}$.
- 目标状态集合为 $G = \{S_4, S_8\}$.

初始状态 S_0 和目标状态 S_4、S_8 如图1所示,

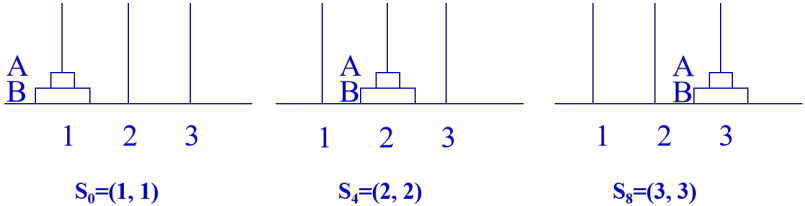


图 1: 二阶梵塔问题的初始状态和目标状态

操作分别用 A(i, j) 和 B(i, j) 表示

- A(i, j) 表示把金片 A 从第 i 号钢针移到 j 号钢针上;
- B(i, j) 表示把金片 B 从第 i 号钢针一到第 j 号钢针上. 共有 12 种操作, 它们分别是:

$$\mathrm{A}(1,2)\ \mathrm{A}(1,3)\ \mathrm{A}(2,1)\ \mathrm{A}(2,3)\ \mathrm{A}(3,1)\ \mathrm{A}(3,2)$$
$$\mathrm{B}(1,2)\ \mathrm{B}(1,3)\ \mathrm{B}(2,1)\ \mathrm{B}(2,3)\ \mathrm{B}(3,1)\ \mathrm{B}(3,2).$$

从初始节点 (1, 1) 到目标节点 (2, 2) 及 (3, 3) 的任何一条路径都是问题的一个解. 其中, 最短的路径长度是 3, 它由 3 个操作组成. 例如, 从 (1, 1) 开始, 通过使用操作 $\mathrm{A}(1,3), \mathrm{B}(1,2)$ 及 $\mathrm{A}(3,2)$, 可到达 (3, 3).

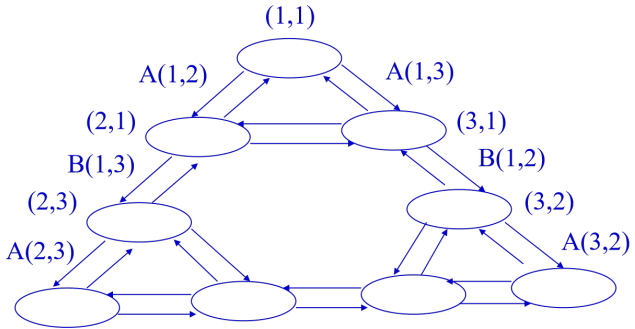


图 2: 二阶梵塔的状态空间图

作业

用 matlab 或 Python 语言实现图 2的二阶梵塔问题的程序, 并绘制流程图.

简称 M-C 问题

例 2.2

修道士 (Missionaries) 和野人 (Cannibals) 问题). 设在河的一岸有三个野人、三个修道士和一条船, 修道士想用这条船把所有的人运到河对岸, 但受以下条件的约束:

- 一是修道士和野人都会划船, 但每次船上至多可载两个人;
- 二是在河的任一岸, 如果野人数目超过修道士数, 修道士会被野人吃掉.

如果野人会服从任何一次过河安排, 请规划一个确保修道士和野人都能过河, 且没有修道士被野人吃掉的安全过河计划.

用 matlab 或 Python 语言实现图 2的二阶梵塔问题的程序, 并绘制流程图.

解: 首先选取描述问题状态的方法. 在这个问题中, 需要考虑两岸的修道士人数和野人数, 还需要考虑船在左岸还是在右岸. 从而可用一个三元组来表示状态

$$S = (m, c, b)$$

其中, m 表示左岸的修道士人数, c 表示左岸的野人数, b 表示左岸的船数.

右岸的状态可由下式确定

- 右岸修道士数 $m' = 3 - m$;
- 右岸野人数 $c' = 3 - c$;
- 右岸船数 $b' = 1 - b$.

在这种表示方式下, m 和 c 都可取 0、1、2、3 中之一, b 可取 0 和 1 中之一. 因此, 共有 $4 \times 4 \times 2 = 32$ 种状态.

这 32 种状态并非全有意义, 除去不合法状态和修道士被野人吃掉的状态, 有意义的状态只有 16 种:

$$
\begin{array}{llll}
S_0 = (3,3,1) & S_1 = (3,2,1) & S_2 = (3,1,1) & S_3 = (2,2,1) \\
S_4 = (1,1,1) & S_5 = (0,3,1) & S_6 = (0,2,1) & S_7 = (0,1,1) \\
S_8 = (3,2,0) & S_9 = (3,1,0) & S_{10} = (3,0,0) & S_{11} = (2,2,0) \\
S_{12} = (1,1,0) & S_{13} = (0,2,0) & S_{14} = (0,1,0) & S_{15} = (0,0,0)
\end{array}
$$

有了这些状态, 还需要考虑可进行的操作.

操作

操作是指用船把修道士或野人从河的左岸运到右岸, 或从河的右岸运到左岸.

每个操作都应当满足如下条件:

- 一是船至少有一个人 (m 或 c) 操作, 离开岸边的 m 和 c 的减少数目应该等于到达岸边的 m 和 c 的增加数目;
- 二是每次操作船上人数不得超过 2 个;
- 三是操作应保证不产生非法状态.

因此, 操作应由条件部分和动作部分:

- 条件: 只有当其条件具备时才能使用.
- 动作: 刻划了应用此操作所产生的结果.

操作的表示: 用符号 P_{ij} 表示从左岸到右岸的运人操作, 用符号 Q_{ij} 表示从右岸到左岸的操作, 其中: i 表示船上的修道士人数, j 表示船上的野人数.

操作集: 本问题有 10 种操作可供选择

$$F = \{P_{01}, P_{10}, P_{11}, P_{02}, P_{20}, Q_{01}, Q_{10}, Q_{11}, Q_{02}, Q_{20}\}. \quad (4)$$

下面以 P_{01} 和 Q_{01} 为例来说明这些操作的条件和动作.

操作符号	条件	动作
P_{01}	$b = 1, m = 0$ 或 $3, c \geq 1$	$b = 0, c = c - 1$
Q_{01}	$b = 0, m = 0$ 或 $3, c \leq 2$	$b = 1, c = c + 1$

例 2.3

猴子摘香蕉问题. 在讨论谓词逻辑知识表示时, 我们曾提到过这一问题, 现在用状态空间法来解决这一问题.

解: 问题的状态可用 4 元组

$$(w, x, y, z) \tag{5}$$

表示. 其中:

- w 表示猴子的水平位置;
- x 表示箱子的水平位置;
- y 表示猴子是否在箱子上, 当猴子在箱子上时, $y = 1$, 否则 $y = 0$;
- z 表示猴子是否拿到香蕉, 当拿到香蕉时 $z = 1$, 否则 $z = 0$.

所有可能的状态

- $S_0 : (a, b, 0, 0)$; 初始状态
- $S_1 : (b, b, 0, 0)$;
- $S_2 : (c, c, 0, 0)$;
- $S_3 : (c, c, 1, 0)$;
- $S_4 : (c, c, 1, 1)$. 目标状态

允许的操作

- Goto(u): 猴子走到位置 u, 即

$$(\mathsf{w}, \mathsf{x}, 0, 0) \to (\mathsf{u}, \mathsf{x}, 0, 0).$$

- Pushbox(v): 猴子推着箱子到水平位置 v, 即

$$(\mathsf{x}, \mathsf{x}, 0, 0) \to (\mathsf{v}, \mathsf{v}, 0, 0).$$

- Climbbox: 猴子爬上箱子, 即

$$(\mathsf{x}, \mathsf{x}, 0, 0) \to (\mathsf{x}, \mathsf{x}, 1, 0).$$

- Grasp; 猴子拿到香蕉, 即 $(\mathsf{c}, \mathsf{c}, 1, 0) \to (\mathsf{c}, \mathsf{c}, 1, 1)$.

这个问题的状态空间图如图3所示. 不难看出, 由初始状态到目标状态的操作序列为:

{Goto(b), Pushbox(c), Climbbox, Grasp}.

猴子摘香蕉问题的解

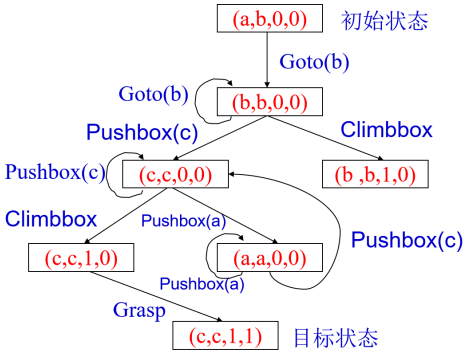


图 3 一阶基模的状态空间图

问题归约法

当问题较复杂时, 可通过分解或变换, 将其转化为一系列较简单的子问题, 然后通过对这些子问题的求解来实现对原问题的求解.

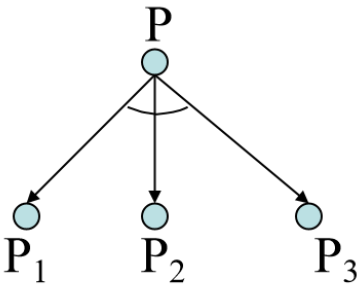
分解

如果一个问题 P 可以归约为一组子问题 $P_1, P_2, \cdots, P_n$, 并且只有当所有子问题 P_i 都有解时原问题 P 才有解, 任何一个子问题 P_i 无解都会导致原问题 P 无解, 则称此种归约为问题的分解. 即分解所得到的子问题的“与”与原问题 P 等价.

等价变换

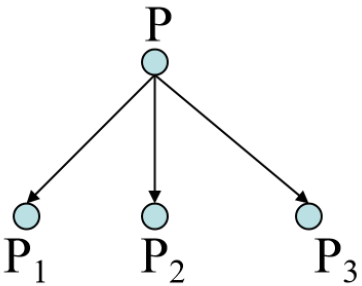
如果一个问题 P 可以归约为一组子问题 $P_1, P_2, \cdots, P_n$, 并且子问题 P_i 中只要有一个有解则原问题 P 就有解, 只有当所有子问题 P_i 都无解时原问题 P 才无解, 称此种归约为问题的等价变换, 简称变换. 即变换所得到的子问题与原问题 P 等价.

与树——分解



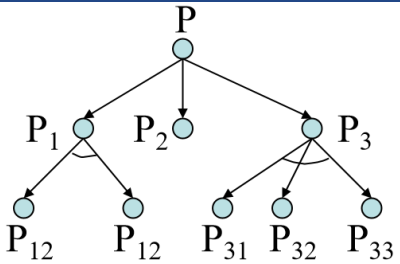
与树

或树——等价变换



或树

与/或树



与/或树

图 6: 与/或树

- 端节点与终止节点
  - 在与/或树中, 没有子节点的节点称为端节点; 本原问题所对应的节点称为终止节点. 可见, 终止节点一定是端节点, 但端节点却不一定是终止节点.
- 可解节点与不可解节点
  在与/或树中, 满足以下三个条件之一的节点为可解节点:
  1. 任何终止节点都是可解节点.
  2. 对“或”节点, 当其子节点中至少有一个为可解节点时, 则该或节点就是可解节点.
  3. 对“与”节点, 只有当其子节点全部为可解节点时, 该与节点才是可解节点.
- 同样, 可用类似的方法定义不可解节点:

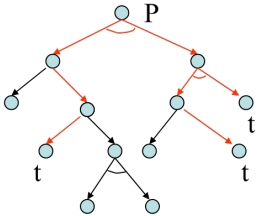
1. 不为终止节点的端节点是不可解节点.
2. 对“或”节点, 若其全部子节点都为不可解节点, 则该或节点是不可解节点.
3. 对“与”节点, 只要其子节点中有一个为不可解节点, 则该与节点是不可解节点.

解树

- 由可解节点构成, 由这些可解节点可以推理出初始节点, 可解节点的子树为解树. 在解树中一定包含初始节点.

下图给出的与或树中, 用红线表示的子树是一个解树. 在该图中, 节点 P 为原始问题节点, 用 t 标出的节点是终止节点. 根据可解节点的定义, 很容易推出原始问题 P 为可解节点.

问题归约求解过程就实际上就是生成解树, 即证明原始节点是可解节点的过程. 这一过程涉及到搜索的问题, 对于与/或树的搜索将在后面详细讨论.



解树

三阶梵塔问题

例 2.4

要求把 1 号钢针上的 3 个金片全部移到 3 号钢针上, 如图8所示.

解: 这个问题也可用状态空间法来解, 不过本例主要用它来说明如何用归约法来解决问题.

问题的形式化表示方法

设用三元组 (i, j, k) 表示问题在任一时刻的状态, 用“$\rightarrow$”表示状态的转换. 上述三元组中

- i 代表金片 C 所在的钢针号
- j 代表金片 B 所在的钢针号
- k 代表金片 A 所在的钢针号

图 8: 与/或树

问题归约方法

原问题可分解为以下三个子问题

1. 把金片 A 及 B 移到 2 号钢针上的双金片移动问题. 即 (1, 1, 1)→(1, 2, 2)
2. 把金片 C 移到 3 号钢针上的单金片移动问题. 即 (1, 2, 2)→(3, 2, 2)
3. 把金片 A 及 B 移到 3 号钢针的双金片移动问题. 即 (3, 2, 2)→((3, 3, 3)

其中, 子问题 (1) 和 (3) 都是一个二阶梵塔问题, 它们都还可以再继续进行分解; 子问题 (2) 是本原问题, 它已不需要再分解.

三阶梵塔问题的分解过程

可用如下图9的与/或树来表示. 在该与/或树中, 有 7 个终止节点, 它们分别对应着 7 个本原问题. 如果把这些本原问题从左至右排列起来, 即得到了原始问题的解:
(1, 1, 1)→(1, 3, 3), (1, 3, 3)→(1, 2, 3), (1, 2, 3)→(1, 2, 2), (1, 2, 2)→(3, 2, 2), (3, 2, 2)→(3, 2, 1), (3, 2, 1)→(3, 3, 1), (3, 3, 1)→(3, 3, 3)

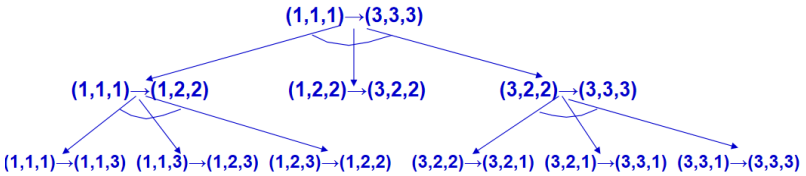


图 9: 与/或树

一般图搜索过程

先把问题的初始状态作为当前扩展节点对其进行扩展, 生成一组子节点, 然后检查问题的目标状态是否出现在这些子节点中. 若出现, 则搜索成功, 找到了问题的解; 否则, 按照某种搜索策略从已生成的子节点中选择一个节点作为当前扩展节点.

重复上述过程, 直到目标状态出现在子节点中或者没有可供操作的节点为止. 所谓对一个节点进行“扩展”是指对该节点用某个可用操作进行作用, 生成该节点的一组子节点.

算法的数据结构和符号约定

数据结构和符号约定

- Open 表: 用于存放刚生成的节点
- Closed 表: 用于存放已经扩展或将要扩展的节点
- S_0: 用表示问题的初始状态
- G: 表示搜索过程所得到的搜索图
- M: 表示当前扩展节点新生成的且不为自己先辈的子节点集.

一般图搜索过程

图搜索过程

1. 把初始节点 S_0 放入 Open 表, 并建立目前仅包含 S_0 的图 G;
2. 检查 Open 表是否为空, 若为空, 则问题无解, 失败推出;
3. 把 Open 表的第一个节点取出放入 Closed 表, 并记该节点为节点 n;
4. 考察节点 n 是否为目标节点. 若是则得到了问题的解, 成功退出;
5. 扩展节点 n, 生成一组子节点. 把这些子节点中不是节点 n 先辈的那部分子节点记入集合 M, 并把这些子节点作为节点 n 的子节点加入 G 中.

一般图搜索过程

1. 针对 M 中子节点的不同情况, 分别作如下处理:

    1. 对那些没有在 G 中出现过的 M 成员设置一个指向其父节点 (即节点 n) 的指针, 并把它放入 Open 表. (新生成的)
    2. 对那些原来已在 G 中出现过, 但还没有被扩展的 M 成员, 确定是否需要修改它指向父节点的指针. (原生成但未扩展的)
    3. 对于那些先前已在 G 中出现过, 并已经扩展了的 M 成员, 确定是否需要修改其后继节点指向父节点的指针. (原生成也扩展过的)

算法的几点说明

1. 按某种策略对 Open 表中的节点进行排序.
2. 转第2步.

(1) 上述过程是状态空间的一般图搜索算法, 它具有通用性, 后面所要讨论的各种状态空间搜索策略都是上述过程的一个特例. 各种搜索策略的主要区别在于对 Open 表中节点的排列顺序不同.

例 3.1

广度优先搜索把先生成的子节点排在前面, 而深度优先搜索则把后生成的子节点排在前面.

(2) 在第 (5) 步对节点 n 扩展后, 生成并记入 M 的子节点有以下三种情况:

① 该子节点来从未被任何节点生成过, 由 n 第一次生成;
② 该子节点原来被其他节点生成过, 但还没有被扩展, 这一次又被 n 再次生成;
③ 该子节点原来被其他节点生成过, 并且已经被扩展过, 这一次又被 n 再次生成.

总结

以上三种情况是对一般图搜索算法而言的. 对于盲目搜索, 由于其状态空间是树状结构, 因此不会出现后两种情况, 每个节点经扩展后生成的子节点都是第一次出现的节点, 不必检查并修改指向父节点的指针.

(3) 在第 (6) 步针对 M 中子节点的不同情况进行处理时, 如果发生当第 ② 种情况, 那么, 这个 M 中的节点究竟应该作为哪一个节点的后继节点呢？一般是由原始节点到该节点路径上所付出的代价来决定的, 哪一条路经付出的代价小, 相应的节点就作为它的父节点. 所谓由原始节点到该节点路径上的代价是指这条路经上的所有有向边的代价之和.

如果发生第③种情况, 除了需要确定该子节点指向父节点的指针外, 还需要确定其后继节点指向父节点的指针. 其依据也是由原始节点到该节点的路径上的代价.

(4) 在搜索图中, 除初始节点外, 任意一个节点都含有且只含有一个指向其父节点的指针. 因此, 由所有节点及其指向父节点的指针所构成的集合是一棵树, 称为搜索树.

(5) 在搜索过程的第 (4) 步, 一旦某个被考察的节点是目标节点, 则搜索过程成功结束. 由初始节点到目标节点路径上的所有操作就构成了该问题的解, 而路径由第 (6) 步所形成的指向父节点的指针来确定.

(6) 如果搜索过程终止在第 (2) 步, 即没有达到目标, 且 Open 表中已无可供扩展的节点, 则失败结束.

广度优先

主要思想

从初始节点 S_0 开始逐层向下扩展, 在第 n 层节点还没有全部搜索完之前, 不进入第 $n+1$ 层节点的搜索. Open 表中的节点总是按进入的先后排序, 先进入的节点排在前面, 后进入的节点排在后面.

算法描述

(1) 把初始节点 S_0 放入 Open 表中;
(2) 如果 Open 表为空, 则问题无解, 失败退出;

(3) 把 Open 表的第一个节点取出放入 Closed 表, 并记该节点为 n;

(4) 考察节点 n 是否为目标节点. 若是, 则得到问题的解, 成功退出;

(5) 若节点 n 不可扩展, 则转第 (2) 步;

(6) 扩展节点 n, 将其子节点放入 Open 表的尾部, 并为每一个子节点设置指向父节点的指针, 然后转第 (2) 步.

例 3.2

在 3×3 的方格棋盘上, 分别放置了表有数字 1、2、3、4、5、6、7、8 的八张牌, 初始状态 S_0, 目标状态 S_g, 如图10所示. 可以使用的操作有

空格左移, 空格上移, 空格右移, 空格下移

即只允许把位于空格左、上、右、下方的牌移入空格. 要求应用广度优先搜索策略寻找从初始状态到目标状态的解路径.

八数码难题

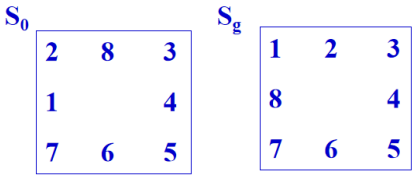


图 10: 与/或树

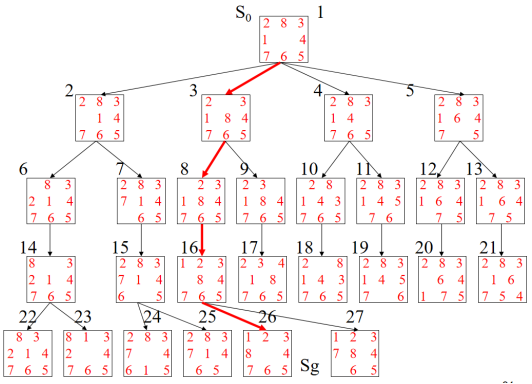


图 11: 与/或树

深度优先搜索

基本思想

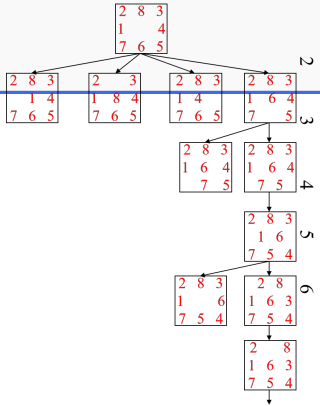
从 S_0 开始, 在子节点中选择一最新生成的节点, 如果该子节点不是目标节点且可以扩展, 则扩展该子节点, 然后再在此子节点的子节点中选择一个最新生成的节点进行考察, 依此向下搜索, 直到某个子节点既不是目标节点, 又不能继续扩展时, 才选择其兄弟节点进行考察.

算法描述

1) 把初始节点 S_0 放入 Open 表中;
(2) 如果 Open 表为空, 则问题无解, 失败退出;
(3) 把 Open 表的第一个节点取出放入 Closed 表, 并记该节点为 n;
(4) 考察节点 n 是否为目标节点. 若是, 则得到问题的解, 成功退出;
(5) 若节点 n 不可扩展, 则转第 (2) 步;
(6) 扩展节点 n, 将其子节点放入 Open 表的首部, 并为每一个子节点设置指向父节点的指针, 然后转第 (2) 步.

例 3.3

八数码难题



八数码难题的深度优先搜索如下图

图 12: 与/或树

代价树的广度优先搜索

在代价树中, 可以用 $g(n)$ 表示从初始节点 S_0 到节点 n 的代价, 用 $c(n_1, n_2)$ 表示从父节点 n_1 到其子节点 n_2 的代价. 这样, 对节点 n_2 的代价有: $g(n_2) = g(n_1) + c(n_1, n_2)$. 代价树搜索的目的是为了找到最佳解, 即找到一条代价最小的解路径.

代价树的广度优先搜索算法

(1) 把初始节点 S_0 放入 Open 表中, 置 S_0 的代价 $g(S_0) = 0$;
(2) 如果 Open 表为空, 则问题无解, 失败退出;
(3) 把 Open 表的第一个节点取出放入 Closed 表, 并记该节点为 n;

(4) 考察节点 n 是否为目标. 若是, 则找到了问题的解, 成功退出;
(5) 若节点 n 不可扩展, 则转第 (2) 步;

(6) 扩展节点 n, 生成其子节点 n_i $(i = 1, 2, ...)$, 将这些子节点放入 Open 表中, 并为每一个子节点设置指向父节点的指针. 按如下公式:

$$g(n_i) = g(n) + c(n_i),\ i = 1, 2, ... \tag{6}$$

计算各子结点的代价, 并根据各子结点的代价对 Open 表中的全部结点按由小到大的顺序排序. 然后转第 (2) 步.

城市交通问题

例 3.4

设有 5 个城市, 它们之间的交通线路如左侧子图13所示, 图中的数字表示两个城市之间的交通费用, 即代价. 用代价树的广度优先搜索, 求从 A 市出发到 E 市, 费用最小的交通路线.

解: 代价树如右侧子图13所示. 其中, 红线为最优解, 其代价为 8.

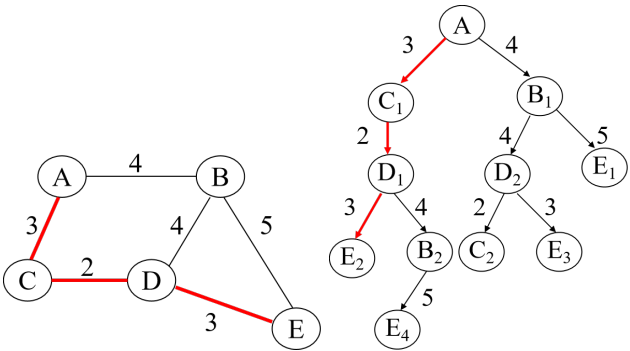


图 13: 城市交通图和城市交通图的代价树

代价树的深度优先搜索算法

(1) 把初始节点 S_0 放入 Open 表中, 置 S_0 的代价 $g(S_0) = 0$;

(2) 如果 Open 表为空, 则问题无解, 失败退出;

(3) 把 Open 表的第一个节点取出放入 Closed 表, 并记该节点为 n;

(4) 考察节点 n 是否为目标节点. 若是, 则找到了问题的解, 成功退出;

(5) 若节点 n 不可扩展, 则转第 (2) 步;

(6) 扩展节点 n, 生成其子节点 $n_i\,(i = 1, 2, \cdots)$, 将这些子节点按边代价由小到大放入 Open 表的首部, 并为每一个子节点设置指向父节点的指针. 然后转第 (2) 步.

启发性信息和估价函数

启发性信息的概念

启发性信息是指那种与具体问题求解过程有关的, 并可指导搜索过程朝着最有希望方向前进的控制信息.

启发性信息的种类

- 有效地帮助确定扩展节点的信息;
- 有效的帮助决定哪些后继节点应被生成的信息;
- 能决定在扩展一个节点时哪些节点应从搜索树上删除的信息.

估价函数的作用

估价函数用来估计节点重要性的函数. 估价函数 $f(n)$ 被定义为从初始节点 S_0 出发, 约束经过节点 n 到达目标节点 S_g 的所有路径中最小路径代价的估计值. 它的一般形式为:

$$f(n) = g(n) + h(n) \tag{7}$$

其中, $g(n)$ 是从初始节点 S_0 到节点 n 的实际代价; $h(n)$ 是从节点 n 到目标节点 S_g 的最优路径的估计代价.

例 4.1

(八数码难题) 设问题的初始状态 S_0, 目标状态 S_g, 如图14所示, 估价函数定义为

$$f(n) = d(n) + W(n) \tag{8}$$

其中: $d(n)$ 表示节点 n 在搜索树中的深度, $W(n)$ 表示节点 n 中 “不在位” 的数码个数. 请计算初始状态 S_0 的估价函数值 $f(S_0)$.

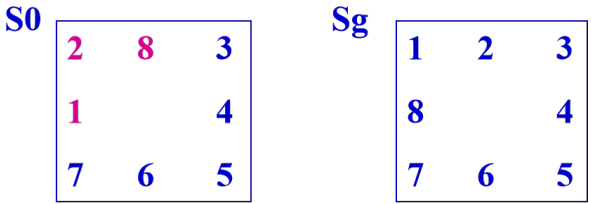


图 14:

解: 取 $g(n) = d(n), h(n) = W(n)$. 它说明是用从 S_0 到 n 的路径上的单位代价表示实际代价, 用结点 n 中“不在位”的数码个数作为启发信息.

一般来说, 某节点中的“不在位”的数码个数越多, 说明它离目标节点越远.

对初始节点 S_0, 由于 $d(S_0) = 0$, $W(S_0) = 3$, 因此有 $f(S_0) = 0 + 3 = 3$.

A 算法

A 算法

图搜索算法中, 如果能在搜索的每一步都利用估价函数 $f(n) = g(n) + h(n)$ 对 Open 表中的节点进行排序, 则该搜索算法为 A 算法.

注

由于估价函数中带有问题自身的启发性信息, 因此, A 算法也被称为启发式搜索算法.

类型

可根据搜索过程中选择扩展节点的范围, 将启发式搜索算法分为全局择优搜索算法和局部择优搜索算法.

- 全局择优: 从 Open 表的所有节点中选择一个估价函数值最小的一个进行扩展.
- 局部择优: 仅从刚生成的子节点中选择一个估价函数值最小的一个进行扩展.

全局择优搜索 A 算法描述

(1) 把初始节点 S_0 放入 Open 表中, $f(S_0) = g(S_0) + h(S_0)$;

(2) 如果 Open 表为空, 则问题无解, 失败退出;

(3) 把 Open 表的第一个节点取出放入 Closed 表, 并记该节点为 n;

(4) 考察节点 n 是否为目标节点. 若是, 则找到了问题的解, 成功退出;

(5) 若节点 n 不可扩展, 则转第 (2) 步;

(6) 扩展节点 n, 生成其子节点 $n_i (i = 1, 2, \cdots)$, 计算每一个子节点的估价值 $f(n_i) (i = 1, 2, \cdots)$, 并为每一个子节点设置指向父节点的指针, 然后将这些子节点放入 Open 表中;

(7) 根据各节点的估价函数值, 对 Open 表中的全部节点按从小到大的顺序重新进行排序; (8) 转第 (2) 步.

例 4.2

八数码难题. 设问题的初始状态 S_0 和目标状态 S_g 如图所示, 估价函数与例58相同. 请用全局择优搜索解决该问题.

解: 该问题的全局择优搜索树如下图所示. 在该图中, 每个节点旁边的数字是该节点的估价函数值.

例 4.3

对节点 S_2, 其估价函数值的计算为: $f(S_2) = d(S_2)+W(S_2) = 1+3 = 4$.

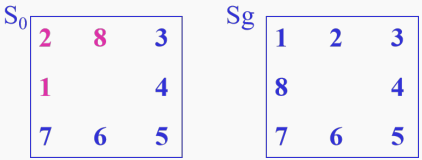


图 15:

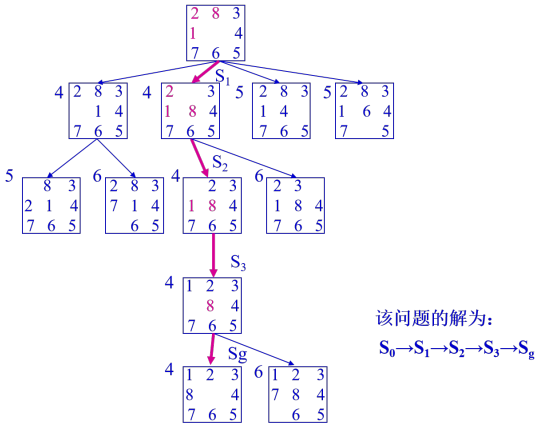


图 16: 八数码难题的全局择优搜索树

Theorem

对无限图, 如果从初始节点 S_0 到目标节点 S_g 有路径存在, 则算法 A* 算法不终止的话, 则从 Open 表中选出的节点必将具有任意大的 f 值.

证明: 设 $d^*(n)$ 是 A* 生成的从初始节点 S_0 到节点 n 的最短路经长度, 由于搜索图中每条边的代价都是一个正数, 令这些正数中的最小的一个数是 e, 则有

$$g^*(n) \geq d^*(n) \times e. \tag{12}$$

因为 $g^*(n)$ 是最佳路径的代价, 故有

$$g(n) \geq g^*(n) \geq d^*(n) \times e. \tag{13}$$

又因为 $h(n) \geq 0$, 故有

$$f(n) = g(n) + h(n) \geq g(n) \geq d^*(n) \times e. \tag{14}$$

如果 A* 算法不终止的话, 从 Open 表中选出的节点必将具有任意大的 $d^*(n)$ 值, 因此, 也将具有任意大的 f 值.

Theorem

在 A* 算法终止前的任何时刻, Open 表中总存在节点 n′, 它是从初始节点 S_0 到目标节点的最佳路径上的一个节点, 且满足 $f(n') \leq f^*(S_0)$.

证明: 设从初始节点 S_0 到目标节点 t 的一条最佳路径序列为

$$S_0 = n_0, n_1, \cdots, n_k = S_g \tag{15}$$

算法开始时, 节点 S_0 在 Open 表中, 当节点 S_0 离开 Open 表进入 Closed 表时, 节点 n_1 进入 Open 表. 因此, A* 没有结束以前, 在 Open 表中必存在最佳路径上的节点. 设这些节点中排在最前面的节点为 n′, 则有

$$f(n') = g(n') + h(n'). \tag{16}$$

由于 n' 在最佳路径上, 故有 $g(n') = g^*(n')$, 从而

$$f(n') = g^*(n') + h(n'). \tag{17}$$

又由于 A^* 算法满足 $h(n') \leq h*(n')$, 故有

$$f(n') \leq g^*(n') + h^*(n') = f^*(n'). \tag{18}$$

因为在最佳路径上的所有节点的 f^* 值都应相等, 因此有

$$f(n') \leq f^*(S_0). \tag{19}$$

Theorem

对无限图, 若从初始节点 S_0 到目标节点 t 有路径存在, 则 A^* 算法必然会结束.

Proof.

(反证法) 假设 A^* 不结束
由引理 4.1 知 Open 表中的节点有任意大的 f 值, 这与引理 4.2 的结论相矛盾, 因此, A^* 算法只能成功结束.
Open 表中任一具有 $f(n) < f^*(S_0)$ 的节点 n, 最终都被 A^* 算法选作为扩展的节点.
(证明略) □

下面给出 A* 算法的可纳性

Theorem

A* 算法是可采纳的, 即若存在从初始节点 S_0 到目标节点 S_g 的路径, 则 A* 算法必能结束在最佳路径上.

Proof.

过程分以下两步进行:
先证明 A* 算法一定能够终止在某个目标节点上.
由定理 4.1 和定理 4.2 可知, 无论是对有限图还是无限图, A* 算法都能够找到某个目标节点而结束.
再证明 A* 算法只能终止在最佳路径上. (反证法) □

假设 A* 算法未能终止在最佳路径上, 而是终止在某个目标节点 t 处, 则有

$$f(t) = g(t) > f^*(S_0). \tag{20}$$

但由引理 4.2 可知, 在 A* 算法结束前, 必有最佳路径上的一个节点 n′ 在 Open 表中, 且有

$$f(n') \leq f^*(S_0) < f(t). \tag{21}$$

这时, A* 算法一定会选择 n′ 来扩展, 而不可能选择 t, 从而也不会去测试目标节点 t, 这就与假设 A* 算法终止在目标节点 t 相矛盾. 因此, A* 算法只能终止在最佳路径上.

Lemma

在 A* 算法中, 对任何被扩展的节点 n, 都有 $f(n) \leq f^*(S_0)$.

Proof.

令 n 是由 A* 选作扩展的任一节点, 因此 n 不会是目标节点, 且搜索没有结束. 由引理 4.2 可知, 在 Open 表中有满足

$$f(n') \leq f^*(S_0). \tag{22}$$

的节点 n′. 若 $n = n'$, 则有 $f(n) \leq f^*(S_0)$; 否则, 选择 n 扩展, 必有

$$f(n) \leq f(n'). \tag{23}$$

所以有

A* 算法的最优性

A* 算法的搜索效率很大程度上取决于估价函数 $h(n)$.

一般来说, 在满足 $h(n) \leq h^*(n)$ 的前提下, $h(n)$ 的值越大越好. $h(n)$ 的值越大, 说明它携带的启发性信息越多, A* 算法搜索时扩展的节点就越少, 搜索效率就越高.

算法的最优性定理

Theorem

A* 算法 A_1^* 和 A_2^*, 它们有

$$A_1^* : f_1(n) = g_1(n) + h_1(n), \tag{25}$$
$$A_2^* : f_2(n) = g_2(n) + h_2(n). \tag{26}$$

如果 A_2^* 比 A_1^* 有更多的启发性信息, 即对所有非目标节点均有

$$h_2(n) > h_1(n), \tag{27}$$

则在搜索过程中, 被 A_2^* 扩展的节点也必然被 A_1^* 扩展, 即 A_1^* 扩展的节点不会比 A_2^* 扩展的节点少, 亦即 A_2^* 扩展的节点集是 A_1^* 扩展的节点集的子集.

用数学归纳法证最优性定理

Proof.

(1) 对深度 $d(n) = 0$ 的节点, 即 n 为初始节点 S_0, 如 n 为目标节点, 则 A_1^* 和 A_2^* 都不扩展 n; 如果 n 不是目标节点, 则 A_1^* 和 A_2^* 都要扩展 n.
(2) 假设对 A_2^* 中 $d(n) = k$ 的任意节点 n 结论成立, 即 A_1^* 也扩展了这些节点.
(3) 证明 A_2^* 中 $d(n) = k + 1$ 的任意节点 n, 也要由 A_1^* 扩展. (用反证法)
假设 A_2 搜索树上有一个满足 $d(n) = k + 1$ 的节点 n, A_2^* 扩展了该节点, 但 A_1^* 没有扩展它. 根据第 (2) 条的假设, 知道 A_1^* 扩展了节点 n 的父节点. 因此, n 必定在 A_1^* 的 Open 表中. 既然节点 n 没有被 A_1^* 扩展, 则有

$$f_1(n) \geq f^*(S_0) \tag{28}$$

□

用数学归纳法证最优性定理

即 $g_1(n) + h_1(n) \geq f^*(S_0)$. 但由于 $d = k$ 时, A_2^* 扩展的节点 A_1^* 也一定扩展, 故有

$$g1(n) \leq g2(n), \tag{29}$$

因此有 $h_1(n) \geq f^*(S_0) - g_2(n)$.
另一方面, 由于 A_2^* 扩展了 n, 因此有

$$f_2(n) \leq f^*(0), \tag{30}$$

即 $g_2(n) + h_2(n) \leq f^*(S_0)$, 亦即 $h_2(n) \leq f^*(S_0) - g_2(n)$, 所以有 $h_1(n) \geq h_2(n)$.
这与我们最初假设的 $h_1(n) < h_2(n)$ 矛盾, 因此假设不成立.

在 A* 算法中, 每当扩展一个节点 n 时, 都需要检查其子节点是否已在 Open 表或 Closed 表中.

对已在 Open 表中的子节点, 需要决定是否调整指向其父节点的指针;

对已在 Closed 表中的子节点, 除需要决定是否调整其指向父节点的指针外, 还需要决定是否调整其子节点的后继节点的父指针.

如果能够保证, 每当扩展一个节点时就已经找到了通往这个节点的最佳路径, 就没有必要再去作上述检查为满足这一要求, 我们需要对启发函数 $h(n)$ 增加单调性限制.

解释

如果启发函数满足以下两个条件

(1) $h(S_g) = 0$;
(2) 对任意节点 n_i 及其任一子节点 n_j, 都有

$$0 \leq h(n_i) - h(n_j) \leq c(n_i, n_j), \tag{31}$$

其中 $c(n_i, n_j)$ 是 n_i 到其子节点 n_j 的边代价, 则称 $h(n)$ 满足单调限制.

h(n) 的单调限制

Theorem

如果 h 满足单调条件, 则当 A^* 算法扩展节点 n 时, 该节点就已经找到了通往它的最佳路径, 即 $g(n) = g^*(n)$.

证明: 设 A^* 正要扩展节点 n, 而节点序列

$$S_0 = n_0, n_1, \cdots, n_k = n, \tag{32}$$

是由初始节点 S_0 到节点 n 的最佳路径. 其中, n_i 是这个序列中最后一个位于 Closed 表中的节点, 则上述节点序列中的 n_{i+1} 节点必定在 Open 表中, 则有

$$g^*(n_i) + h(n_i) \leq g^*(n_i) + c(n_i, n_{i+1}) + h(n_{i+1}). \tag{33}$$

由于节点 n_i 和 n_{i+1} 都在最佳路径上, 故有

$$g^*(n_{i+1}) = g^*(n_i) + c(n_i, n_{i+1}). \tag{34}$$

所以

$$g^*(n_i) + h(n_i) \leq g^*(n_{i+1}) + h(n_{i+1}). \tag{35}$$

一直推导下去可得

$$g^*(n_{i+1}) + h(n_{i+1}) \leq g^*(n_k) + h(n_k). \tag{36}$$

由于节点 n_{i+1} 在最佳路径上, 故有

$$f(n_{i+1}) \leq g^*(n) + h(n). \tag{37}$$

因为这时 A^* 扩展节点 n 而不扩展节点 n_{i+1}, 则有

$$f(n) = g(n) + h(n) \leq f(n_{i+1}) \leq g^*(n) + h(n). \tag{38}$$

即

$$g(n) \leq g^*(n). \tag{39}$$

但是 $g^*(n)$ 是最小代价值, 应当有

$$g(n) \geq g^*(n), \tag{40}$$

所以有

$$g(n) = g^*(n). \tag{41}$$

Theorem

如果 $h(n)$ 满足单调限制, 则 A^* 算法扩展的节点序列的 f 值是非递减的, 即 $f(n_i) \leq f(n_{i+1})$.

证明: 假设节点 n_{i+1} 在节点 n_i 之后立即扩展, 由单调限制条件可知

$$h(n_i) - h(n_{i+1}) \leq c(n_i, n_{i+1}). \tag{42}$$

即

$$f(n_i) - g(n_i) - f(n_{i+1}) + g(n_{i+1}) \leq c(n_i, n_{i+1}). \tag{43}$$

亦即

$$f(n_i) - g(n_i) - f(n_{i+1}) + g(n_i) + c(n_i, n_{i+1}) \leq c(n_i, n_{i+1}), \tag{44}$$

$$f(n_i) - f(n_{i+1}) \leq 0; \text{即}, f(n_i) \leq f(n_{i+1}). \tag{45}$$

注

以上两个定理都是在 h(n) 满足单调性限制的前提下才成立的. 如果 h(n) 不满足单调性限制, 则它们不一定成立.

在 h(n) 满足单调性限制下的 A* 算法常被称为改进的 A* 算法.

A* 算法应用举例

例 4.4

八数码难题.

$f(n) = d(n) + P(n)$, $d(n)$ 深度, $P(n)$ 与目标距离 $f^* = g^* + h^*$.

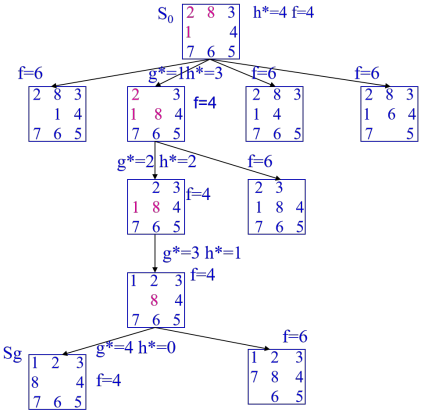


图 17: 八数码难题 h(n)=P(n) 的搜索树

A* 算法应用举例

例 4.5

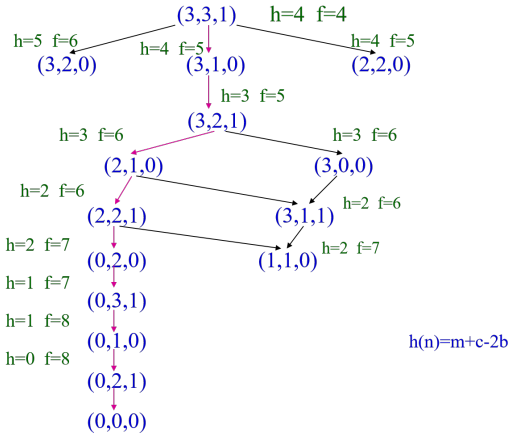
修道士和野人问题.

解: 用 m 表示左岸的修道士人数, c 表示左岸的野人数, b 表示左岸的船数, 用三元组 (m, c, b) 表示问题的状态.
对 A* 算法, 首先需要确定估价函数. 设 $g(n) = d(n), h(n) = m + c - 2b$, 则有

$$f(n) = g(n) + h(n) = d(n) + m + c - 2b \tag{46}$$

其中, $d(n)$ 为节点的深度. 通过分析可知 $h(n) \leq h^*(n)$, 满足 A* 算法的限制条件.

M-C 问题的搜索过程如下图所示.



与/或树的一般搜索

与/或树的搜索过程实际上是一个不断寻找解树的过程. 其一般搜索过程如下:

(1) 把原始问题作为初始节点 S_0, 并把它作为当前节点;

(2) 应用分解或等价变换操作对当前节点进行扩展;

(3) 为每个子节点设置指向父节点的指针;

(4) 选择合适的子节点作为当前节点, 反复执行第 (2) 步和第 (3) 步, 在此期间需要多次调用可解标记过程或不可解标记过程, 直到初始节点被标记为可解节点或不可解节点为止.

上述搜索过程将形成一颗与/或树, 这种由搜索过程所形成的与/或树称为搜索树.

与/或树的广度优先

与/或树的广度优先搜索算法

(1) 把初始节点 S_0 放入 Open 表中;

(2) 把 Open 表的第一个节点取出放入 Closed 表, 并记该节点为 n;

(3) 如果节点 n 可扩展, 则做下列工作:
① 展节点 n, 将其子节点放入 Open 表的尾部, 并为每一个子节点设置指向父节点的指针;

② 考察这些子节点中有否终止节点. 若有, 则标记这些终止节点为可解节点, 并用可解标记过程对其父节点及先辈节点中的可解解节点进行标记. 如果初始解节点 S_0 能够被标记为可解节点, 就得到了解树, 搜索成功, 退出搜索过程; 如果不能确定 S_0 为可解节点, 则从 Open 表中删去具有可解先辈的节点.

③ 转第 (2) 步.

(4) 如果节点 n 不可扩展, 则作下列工作:

① 标记节点 n 为不可解节点;

② 应用不可解标记过程对节点 n 的先辈中不可解解的节点进行标记. 如果初始解节点 S_0 也被标记为不可解节点, 则搜索失败, 表明原始问题无解, 退出搜索过程; 如果不能确定 S_0 为不可解节点, 则从 Open 表中删去具有不可解先辈的节点.

③ 转第 (2) 步.

例 4.6

设有下图所示的与/或树, 节点按标注顺序进行扩展, 其中表有 t_1, t_2, t_3 的节点是终止节点, A, B, C 为不可解的端节点.

搜索过程

(1) 先扩展 1 号节点, 生成 2 号节点和 3 号节点.

(2) 扩展 2 号节点, 生成 A 节点和 4 号节点.

(3) 扩展 3 号节点, 生成 t_1 节点和 5 号节点. 由于 t_1 为终止节点, 则标记它为可解节点, 并应用可解标记过程, 不能确定 3 号节点是否可解.

(4) 扩展节点 A, 由于 A 是端节点, 因此不可扩展. 调用不可解标记过程.

(5) 扩展 4 号节点, 生成 t_2 节点和 B 节点. 由于 t_2 为终止节点, 则标记它为可解节点, 并应用可解标记过程, 可标记 2 号节点为可解, 但不能标记 1 号节点为可解.

(6) 扩展 5 号节点, 生成 t_3 节点和 C 节点. 由于 t_3 为终止节点, 则标记它为可解节点, 并应用可解标记过程, 可标记 1 号节点为可解节点.

(7) 搜索成功, 得到由 1、2、3、4、5 号节点即 t_1、t_2、t_3 节点构成的解树.

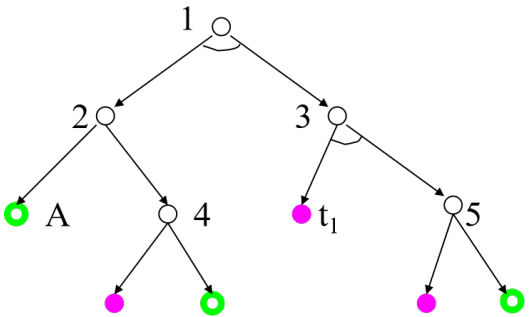


图 19: 与/或树的广度优先搜索

与/或树的深度优先搜索

与/或树的深度优先搜索也可以带有深度限制 d_m, 其搜索算法如下:

(1) 把初始节点 S_0 放入 Open 表中;

(2) 把 Open 表第一个节点取出放入 Closed 表, 并记该节点为 n;

(3) 如果节点 n 的深度等于 d_m, 则转第 (5) 步的第①点;

(4) 如果节点 n 可扩展, 则做下列工作:

① 扩展节点 n, 将其子节点放入 Open 表的首部, 并为每一个子节点设置指向父节点的指针;

② 考察这些子节点中是否有终止节点. 若有, 则标记这些终止节点为可解节点, 并用可解标记过程对其父节点及先辈节点中的可解解节点进行标记. 如果初始解节点 S_0 能够被标记为可解节点, 就得到了解树, 搜索成功, 退出搜索过程; 如果不能确定 S_0 为可解节点, 则从 Open 表中删去具有可解先辈的节点. ③ 转第 (2) 步.

如果节点 n 不可扩展

① 标记节点 n 为不可解节点;
② 应用不可解标记过程对节点 n 的先辈中不可解解的节点进行标记. 如果初始解节点 S_0 也被标记为不可解节点, 则搜索失败, 表明原始问题无解, 退出搜索过程; 如果不能确定 S_0 为不可解节点, 则从 Open 表中删去具有不可解先辈的节点. ③ 转第 (2) 步.

搜索过程

(1) 先扩展 1 号节点, 生成 2 号节点和 3 号节点.

(2) 扩展 3 号节点, 生成 t_1 节点和 5 号节点. 由于 t_1 为终止节点, 则标记它为可解节点, 并应用可解标记过程, 不能确定 3 号节点是否可解.

(3) 扩展 5 号节点, 生成 t_3 节点和 C 节点. 由于 t_3 为终止节点, 则标记它为可解节点, 并应用可解标记过程, 可标记 3 号节点为可解节点, 但不能标记 1 号为可解.

搜索过程

(4) 扩展 2 号节点, 生成 A 节点和 4 号节点.

(5) 扩展 4 号节点, 生成 t_2 节点和 B 节点. 由于 t_2 为终止节点, 则标记它为可解节点, 并应用可解标记过程, 可标记 2 号节点为可解, 再往上又可标记 1 号节点为可解.

(6) 搜索成功, 得到由 1、3、5、2、4 号节点即 t_1、t_2、t_3 节点构成的解树.

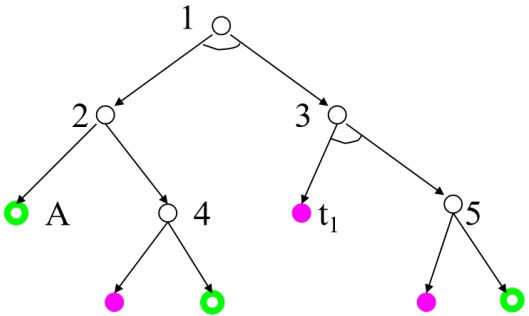


图 20: 与/或树的有界深度优先搜索

对上例, 若按有界深度优先, 且设 $d_m = 4$, 则其节点扩展顺序为: 1, 3, 5, 2, 4.

与/或树的启发式搜索

与/或树的启发式搜索过程

实际上是一种利用搜索过程所得到的启发性信息寻找最优解树的过程. 算法的每一步都试图找到一个最有希望成为最优解树的子树.

最优解树

最优解树是指代价最小的那棵解树. 它涉及到解树的代价与希望树.

解树的代价可按如下规则计算:

- 若 n 为终止节点, 则其代价 $b(n) = 0$;
- 若 n 为或节点, 且子节点为 $n_1, n_2, \cdots, n_k$, 则 n 的代价为:

$$h(n) = \min_{1 \le i \le k} \{c(n, n_i) + h(n_i)\} \tag{47}$$

其中, $c(n, n_i)$ 是节点 n 到其子节点 n_i 的边代价.

- 若 n 为与节点, 且子节点为 $n_1, n_2, \cdots, n_k$, 则 n 的代价可用和代价法或最大代价法.

解树的代价

- 若用和代价法, 则其计算公式为:

$$h(n) = \sum_{i=1}^{k} \{c(n, n_i) + h(n_i)\} \tag{48}$$

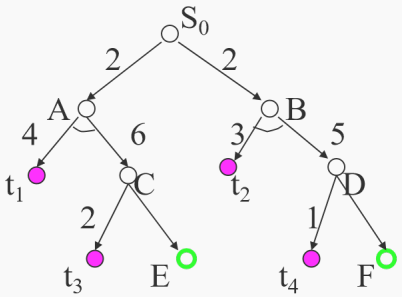
- 若用最大代价法, 则其计算公式为:

$$h(n) = \max_{1 \le i \le k} \{c(n, n_i) + h(n_i)\} \tag{49}$$

- 若 n 是端节点, 但又不是终止节点, 则 n 不可扩展, 其代价定义为 $h(n) = \alpha$. 根节点的代价即为解树的代价.

例 5.1

设下图是一棵与/或树, 左边的解树由 S_0, A, t_1, C 及 t_2 组成; 右边的解树由 S_0, B, t_2, D 及 t_4 组成. 在此与或树中, t_1, t_2, t_3, t_4 为终止节点; E 和 F 是端节点; 边上的数字是该边的代价. 请计算解树的代价.



解:

先计算左边的解树

按和代价: $\mathsf{h}(\mathsf{S}_0) = 2 + 4 + 6 + 2 = 14.$

按最大代价: $\mathsf{h}(\mathsf{S}_0) = (2 + 6) + 2 = 10.$

再计算右边的解树

按和代价: $\mathsf{h}(\mathsf{S}_0) = 1 + 5 + 3 + 2 = 11.$

按最大代价: $\mathsf{h}(\mathsf{S}_0) = (1 + 5) + 2 = 8.$

希望树

希望树

指搜索过程中最有可能成为最优解树的那棵树.

与/或树的启发式搜索过程

就是不断地选择、修正希望树的过程, 在该过程中, 希望树是不断变化的.

希望解树

(1) 初始节点 S_0 在希望树 T.
(2) 如果 n 是具有子节点 $n_1, n_2, \cdots, n_k$ 的或节点, 则 n 的某个子节点 n_i 在希望树 T 中的充分必要条件是？
(3) 如果 n 是与节点, 则 n 的全部子节点都在希望树 T 中.

与/或树的启发式搜索过程如下:

(1) 把初始节点 S_0 放入 Open 表中, 计算 $h(S_0)$;
(2) 计算希望树 T;
(3) 依次在 Open 表中取出 T 的端节点放入 Closed 表, 并记该节点为 n;

续上页

(4) 如果节点 n 为终止节点, 则做下列工作:

①标记节点 n 为可解节点;

② 在 T 上应用可解标记过程, 对 n 的先辈节点中的所有可解解节点进行标记;

③ 如果初始解节点 S_0 能够被标记为可解节点, 则 T 就是最优解树, 成功退出;

④ 否则, 从 Open 表中删去具有可解先辈的所有节点.

⑤转第 (2) 步.

(5) 如果节点 n 不是终止节点, 但可扩展, 则做下列工作:

①扩展节点 n, 生成 n 的所有子节点;

② 把这些子节点都放入 Open 表中, 并为每一个子节点设置指向父节点 n 的指针

③计算这些子节点及其先辈节点的 h 值;

④ 转第 (2) 步.

(6) 如果节点 n 不是终止节点, 且不可扩展, 则做下列工作:

①标记节点 n 为不可解节点;

② 在 T 上应用不可解标记过程, 对 n 的先辈节点中的所有不可解解节点进行标记;

③ 如果初始解节点 S_0 能够被标记为不可解节点, 则问题无解, 失败退出;

④否则, 从 Open 表中删去具有不可解先辈的所有节点.

⑤ 转第 (2) 步.

注

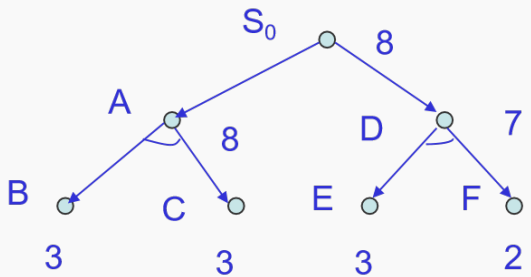
要求搜索过程每次扩展节点时都同时扩展两层, 且按一层或节点、一层与节点的间隔方式进行扩展. 它实际上就是下一节将要讨论的博弈树的结构.

设初始节点为 S_0, 对 S_0 扩展后得到的与/或树如右图所示. 其中, 端节点 B, C, E, F, 下面的数字是用启发函数估算出的 h 值, 节点 S_0, A, D 旁边的数字是按和代价法计算出来的节点代价. 此时, S_0 的右子树是当前的希望树.

和代价法

例 5.2

节点 A 的值 =3+1+2+1+1=8.



扩展节点 E, 得到如下图所示的与/或树. 此时, 由右子树求出的 $h(S_0) = 12$. 但是, 由左子树求出的 $h(S_0) = 9$. 显然, 左子树的代价小. 因此, 当前的希望树应改为左子树.

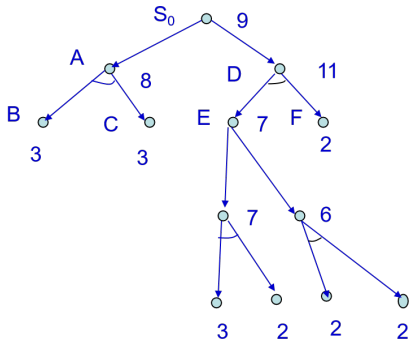


图 23: 扩展节点 E 后得到的与/或树

对节点 B 进行扩展, 扩展两层后得到的与/或树如下图所示. 由于节点 H 和 I 是可解节点, 故调用可解标记过程, 得节点 G, B 也为可解节点, 但不能标记 S_0 为可解节点, 须继续扩展. 当前的希望树仍然是左子树.

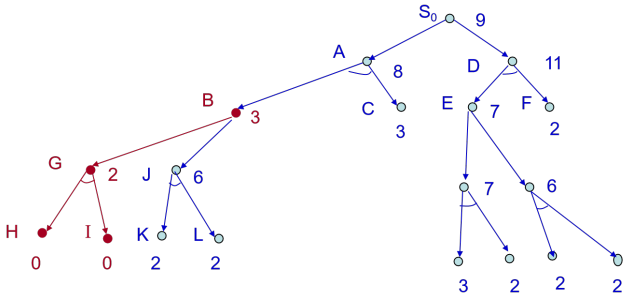
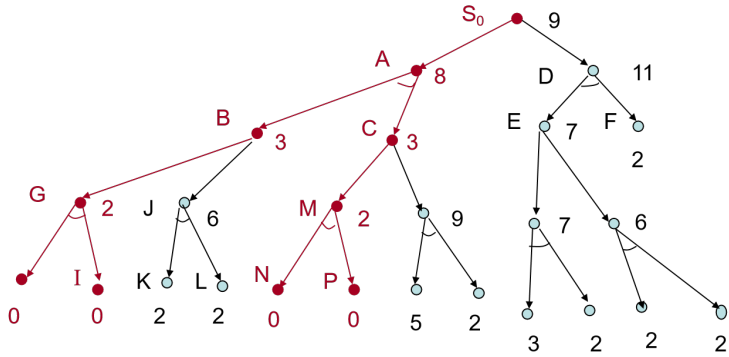


图 24: 扩展节点 B 后得到的与/或树

对节点 C 进行扩展, 扩展两层后得到的与/或树如右图所示. 由于节点 N 和 P 是可解节点, 故调用可解标记过程, 得节点 M, C, A 也为可解节点, 进而可标记 S_0 为可解节点, 这就的到了代价最小的解树. 按和代价法, 该最优解的代价为 9.



博弈树的启发式搜索

博弈的概念

博弈是一类具有智能行为的竞争活动, 如下棋、战争等.

博弈的类型

- 双人完备信息博弈: 两位选手 (例如 MAX 和 MIN) 对垒, 轮流走步, 每一方不仅知道对方已经走过的棋步, 而且还能估计出对方未来的走步.
- 机遇性博弈: 存在不可预测性的博弈, 例如掷币等.

博弈树的特点

若把双人完备信息博弈过程用图表示出来, 就得到一棵与/或树, 这种与/或树被称为博弈树. 在博弈树中, 那些下一步该 MAX 走步的节点称为 MAX 节点, 下一步该 MIN 走步的节点称为 MIN 节点.

(1) 博弈的初始状态是初始节点;

(2) 博弈树中的“或”节点和“与”节点是逐层交替出现的;

(3) 整个博弈过程始终站在某一方的立场上, 例如 MAX 方. 所有能使自己一方获胜的终局都是本原问题, 相应的节点是可解节点; 所有使对方获胜的终局都是不可解节点.

极大极小过程

对简单的博弈问题, 可生成整个博弈树, 找到必胜的策略.

- 对于复杂的博弈问题, 不可能生成整个搜索树, 如国际象棋, 大约有 10120 个节点. 一种可行的方法是用当前正在考察的节点生成一棵部分博弈树, 并利用估价函数 f(n) 对叶节点进行静态估值.

- 对叶节点的估值方法是: 那些对 MAX 有利的节点, 其估价函数取正值; 那些对 MIN 有利的节点, 其估价函数取负值; 那些使双方均等的节点, 其估价函数取接近于 0 的值.

极大极小过程

对非叶节点的值, 必须从叶节点开始向上倒退.

倒退方法:

- 对于 MAX 节点, 由于 MAX 方总是选择估值最大的走步, 因此, MAX 节点的倒退值应取其后继节点估值的最大值.
- 对于 MIN 节点, 由于 MIN 方总是选择估值最小的走步, 因此, MIN 节点的倒推值应取其后继节点估值的最小值.

极大极小过程

这样一步一步的计算倒推值, 直至求出初始节点的倒推值为止. 这一过程称为极大极小过程.

一子棋游戏

例 6.1

设有一个三行三列的棋盘, 如下图所示, 两个棋手轮流走步, 每个棋手走步时往空格上摆一个自己的棋子, 谁先使自己的棋子成三子一线为赢.

设 MAX 方的棋子用 × 标记, MIN 方的棋子用 ○ 标记, 并规定 MAX 方先走步.

一子棋棋盘

棋局1

解: 估价函数 $e(+P)$: P 上有可能使 $\times$ 成三子为一线的数目; $e(-P)$: P 上有可能使 $\circ$ 成三子为一线的数目;
当 MAX 必胜 $e(P)$ 为正无穷大, MIN 必胜 $e(P)$ 为负无穷大
棋局即估价函数: 具有对称性的棋盘可认为是同一棋盘. 如下图所示:

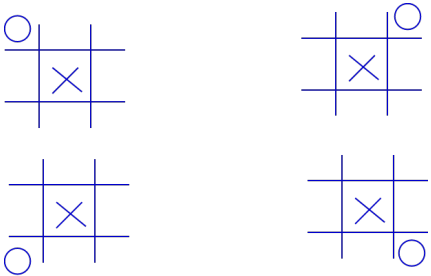


图 27: $e(P) = e(+P) - e(-P)$

一子棋的极大极小搜索

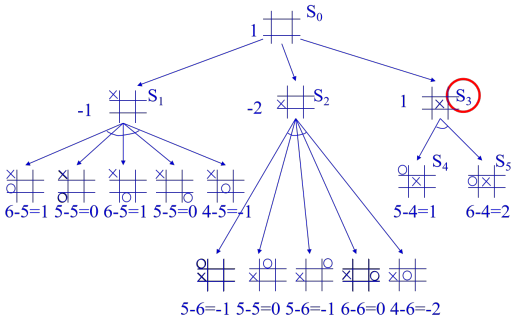


图 28: 一子棋的极大极小搜索

α-β 剪枝

极大极小过程是先生成与/或树, 然后再计算各节点的估值, 这种生成节点和计算估值相分离的搜索方式, 需要生成规定深度内的所有节点, 因此搜索效率较低.

$\alpha - \beta$ 剪枝

如果能边生成节点边对节点估值, 并剪去一些没用的分枝, 这种技术被称为 $\alpha - \beta$ 剪枝.

剪枝方法

剪枝方法

(1) MAX 节点 (或节点) 的 α 值为当前子节点的最大倒推值;
(2) MIN 节点 (与节点) 的 β 值为当前子节点的最小倒推值;
(3) α-β 剪枝的规则如下:

- 任何 MAX 节点 n 的 α 值大于或等于它先辈节点的 β 值, 则 n 以下的分枝可停止搜索, 并令节点 n 的倒推值为 α. 这种剪枝称为 β 剪枝.
- 任何 MIN 节点 n 的 α 值小于或等于它先辈节点的 α 值, 则 n 以下的分枝可停止搜索, 并令节点 n 的倒推值为 β. 这种剪枝称为 α 剪枝.

例 6.2

一个 $\alpha-\beta$ 剪枝的具体例子, 如图29所示. 其中最下面一层端节点旁边的数字是假设的估值.

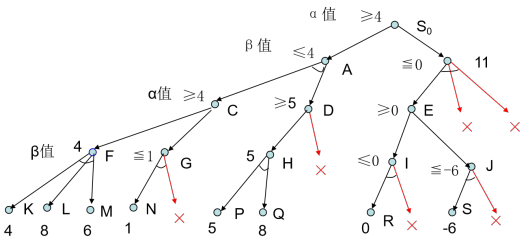


图 29: $\alpha-\beta$ 剪枝

作业

思考

何谓估价函数, 在估价函数中, $g(n)$ 和 $h(n)$ 各起什么作用?

作业

探索

设有如下结构的移动将牌游戏:B B W W E, 其中, B 表示黑色将牌, W 表是白色将牌, E 表示空格. 游戏的规定走法是: (1) 任意一个将牌可移入相邻的空格, 规定其代价为 1; (2) 任何一个将牌可相隔 1 个其它的将牌跳入空格, 其代价为跳过将牌的数目加 1. 游戏要达到的目标什是把所有 W 都移到 B 的左边. 对这个问题, 请定义一个启发函数 $h(n)$, 并给出用这个启发函数产生的搜索树. 你能否判别这个启发函数是否满足下界要求？在求出的搜索树中, 对所有节点是否满足单调限制？