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。搜索策略的优劣直接影响到智能系统的性能和推理效率。

搜索的基本概念

搜索的含义

根据问题的实际情况不断寻找可利用的知识，**构造出一条代价较少的推理路线**，使问题得到圆满解决的过程称为搜索。包括两个方面：

- 找到从初始事实到问题最终答案的一条推理路径
- 找到的这条路径在时间和空间上复杂度最小

搜索的分类

按是否使用启发信息可分为：

- 盲目搜索（Uninformed search）：按预定的控制策略进行搜索，搜索过程中获得的中间信息不改变控制策略。由于搜索总是按预定的路线进行，并没有考虑到问题本身的特性，因此这种搜索具有盲目性，效率不高，不利于求解复杂问题。
- 启发式搜索（Heuristic search，Informed search）：利用问题领域相关的信息作为启发信息，用来指导搜索朝着最有希望的方向前进，提高搜索效率并力图找到最优解。

并不是每一类问题都容易抽取启发信息，所以在很多情况下仍然需要盲目搜索。

按问题的表示方式可分为：

- 状态空间搜索：用状态空间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所进行的搜索。

状态空间法和问题归约法是人工智能中最基本的两种问题表示和求解方法。

搜索策略常用评价指标

- 完备性（Completeness）：如果问题有解，算法就能找到，称此搜索方法是完备的。
- 最优性（Optimality）：如果解存在，总能找到最优解。
- 空间复杂度（Time Complexity）
- 时间复杂度（Space Complexity）

问题的状态空间表示

状态空间法是人工智能中最基本的问题 求解方法，它所采用的问题表示方法称为状态空间表示法。状态空间法的基本思想是用“状态”和“操作”来表示问题和求解问题的。

状态、操作和状态空间

- 状态（state）：状态是表示问题求解过程中每一步问题状况的数据结构，可用如下形式表示：

$$S_k = \{S_{k0}, S_{k1}, \ldots\}$$

  当每一个分量都给予确定的值时，就得到一个具体的状态。

- 操作（operator）：也称为算符，它是把问题从一种状态变化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）：是有一个问题的全部状态以及这些状态之间的相互关系所构成的集合。可用一个三元组表示：

$$(S, F, G)$$

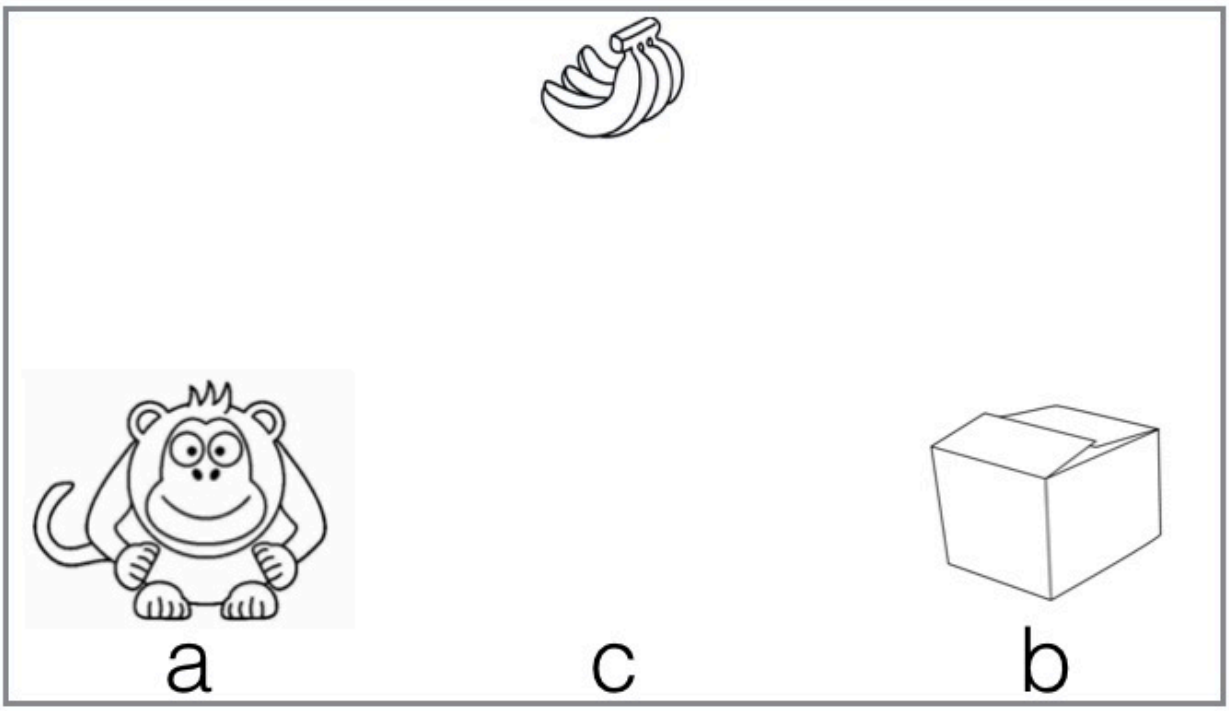
  其中，S为问题的所有初始状态的集合、F为操作的集合、G为目标状态的集合。

- 状态空间图：状态空间也可用一个 赋值的有向图来表示，该有向图称为状态空间图。状态空间图中，节点表示问题的状态，有向边表示操作。

状态空间法求解问题的基本过程

1. 首先为问题选择适当的“状态”和“操作”的形式化描述方法。
2. 然后从某个初始状态出发，每次使用一个满足前提条件的“操作”并产生新的状态，递增地建立起操作的序列，直到到达目标状态为止。
3. 此时，**由初始状态到目标状态所使用的算符（操作符）序列就是该问题的一个解**。

例：设房间里有一只猴子位于a处，在c处上方的天花板上有一串香蕉，猴子想吃但摘不到。b处有一个箱子，如果猴子站在箱子上就能摘到。



猴子摘香蕉问题

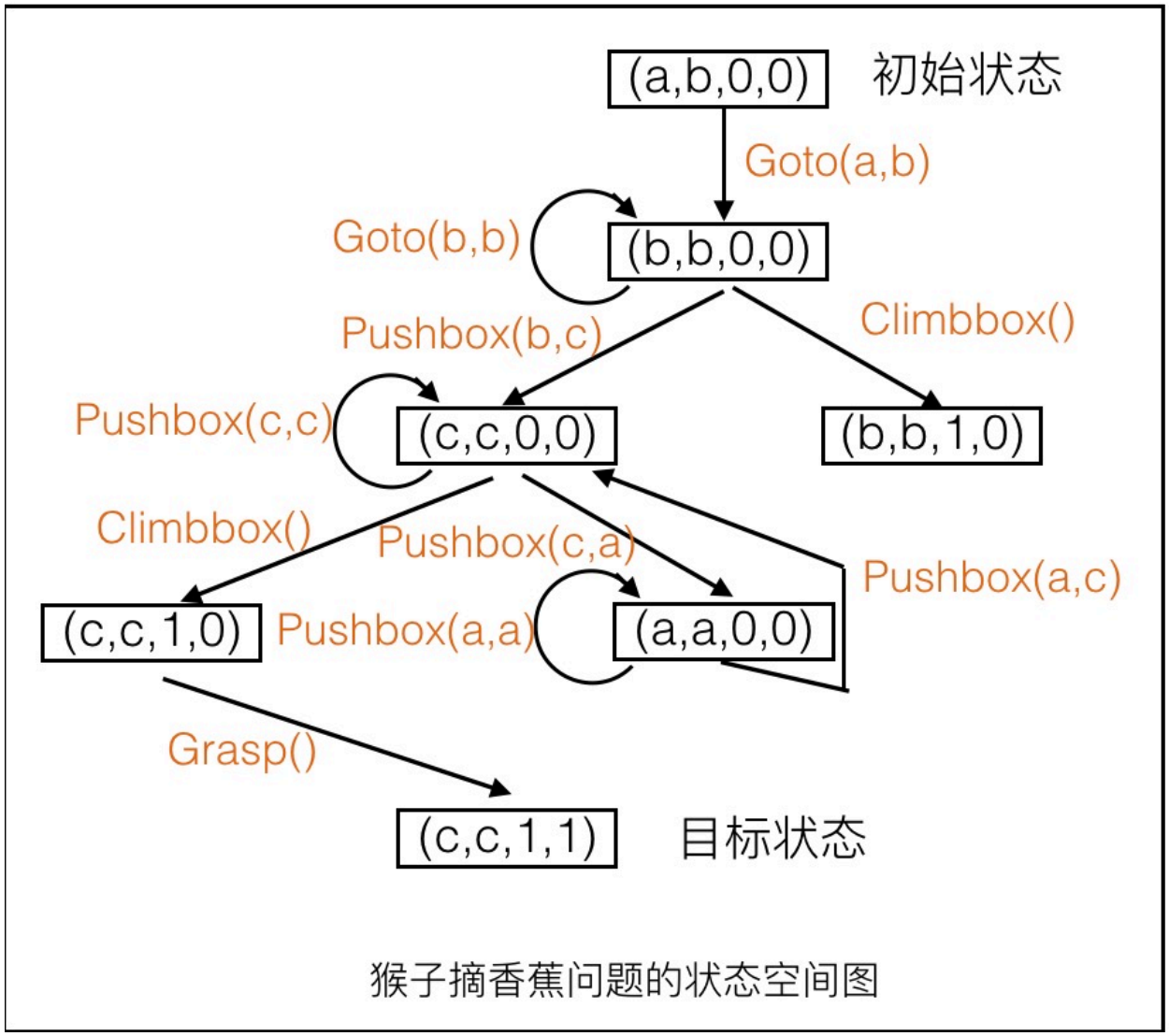
问题的状态可用四元组(w,x,y,z)来表示。其中，w表示猴子的水平位置；x表示箱子的水平位置；y表示猴子是否在箱子上，当猴子在箱子上时，y取1，否则y取0；z表示猴子是否拿到香蕉，拿到取1，否则取0。初始状态：$(a,b,0,0)$，目标状态：$(c,c,1,1)$。

定义操作符：

- $Goto(u,v)$：猴子从u处走到v处。条件:$(w=u \quad AND \quad x=0)$。状态变化：$(u,x,0,G) \implies (v,x,0,G)$。
- $Pushbox(u,v)$：猴子推着箱子从u处移动到v处。条件：$(w=u \quad AND \quad x=u \quad AND \quad y=0)$。状态变化：$(u,u,0,y) \implies (v,v,0,y)$。
- $Climbbox()$：猴子爬到箱子上。条件：$(x=y \quad AND \quad y=0)$。状态变化：$(w,x,0,z) \implies (w,x,1,z)$。
- $Grasp()$：猴子拿到香蕉。条件：$(x=c \quad AND \quad w=c \quad AND \quad y=1 \quad AND \quad G=0)$。状态变化：$(c,c,1,0) \implies (c,c,1,1)$。

不难看出猴子拿到香蕉的最佳操作序列：

$$\{Goto(a,b), Pushbox(b,c), Climbbox(), Grasp()\}$$



猴子摘香蕉问题的状态空间图

状态空间的盲目搜索

尽管盲目搜索性能不如启发式搜索，但由于启发式搜索需要抽取与问题本身有关的特征信息，而这种特征信息的抽取往往比较困难，因此盲目搜索仍不失一种有效的搜索策略。状态空间的盲目搜索是一种基于问题状态空间表示的盲目搜索算法。根据状态空间采用的数据结构不同，可分为图搜索算法和树搜索算法，树搜索算法包括一般树和代价树的盲目搜索算法。

基本概念

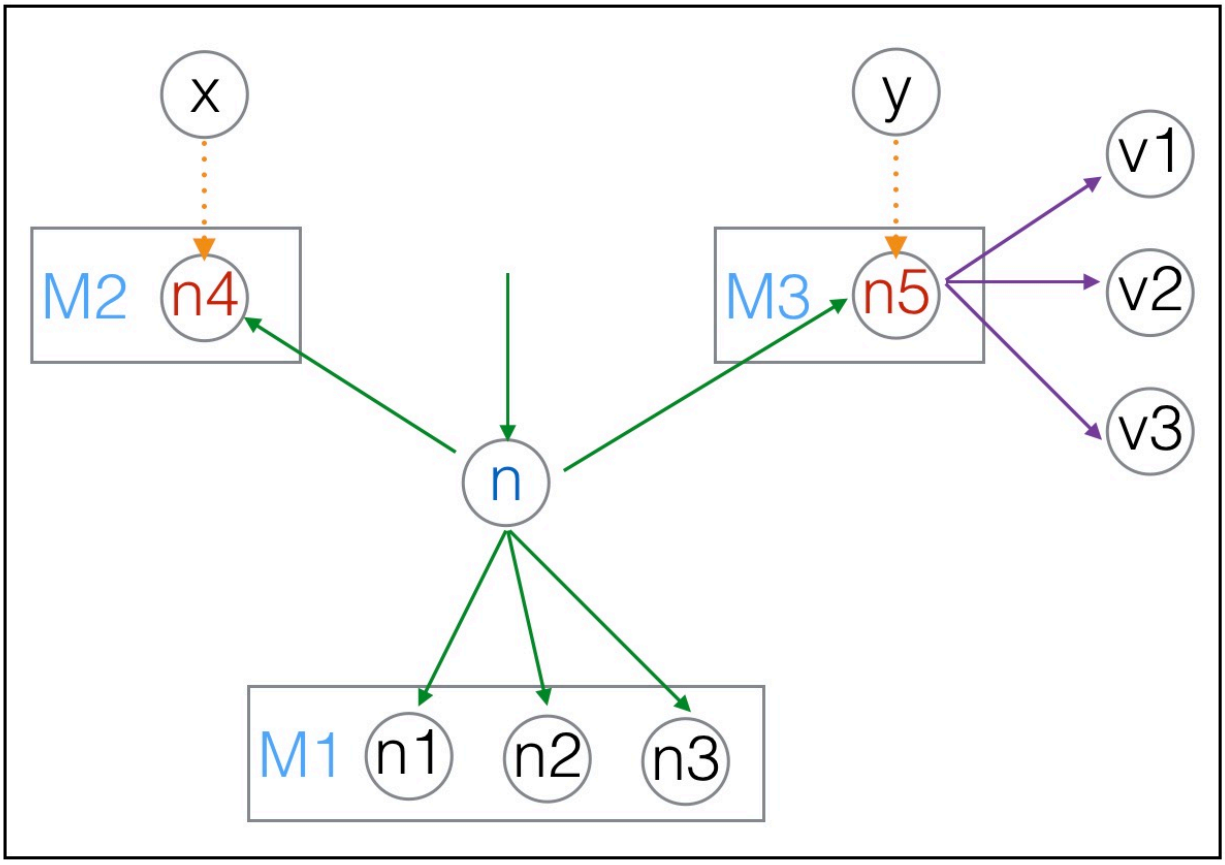
1. 扩展节点：对某一节点（状态）选择合适的操作符作用在节点上使产生后继子节点（状态）的操作。即，类似数据结构中的寻找邻接点，但这里的邻接点是选择操作后产生的。
2. $Open$和$Closed$表：这两个表用来存放节点，$Open$表存放未扩展节点，$Closed$表存放已扩展节点和待扩展节点。

图搜索的一般过程

1. 建立一个只含初始状态节点S的搜索图G，建立一个$Open$表，用来存放未扩展节点，将S放入$Open$表中；
2. 建立一个$Closed$表，用来存放已扩展和待扩展节点，初始为空；
3. 若$Open$为空，则失败，退出；
4. 选择$Open$表中的第一个节点，将其移到$Closed$表中，称此节点为n节点；
5. 若n为目标节点，则成功，退出；
6. 扩展n节点，生成n的后继节点集合$M = M1 + M2 + M3$，其中n的后继节点分为3中情况。设$M1$表示图G中新节点（最新生成的）；$M2$在图中已经存在，处于$Open$表中；$M3$在图G中已经存在，且已经在$Closed$表中。(1)对$M1$型节点，加入到图G中，并放入$Open$表中，设置一个指向父节点n的指针；(2)对$M2$型节点，已经在$Open$表中，确定是否需要修改父节点指针；(3)对$M3$型节点，已经在$Closed$表中，确定是否修改器父节点指针；是否修改其后裔节点的指针。

   按照某一控制策略，重新排序$Open$表。

   返回第3步。



如上图，M1是图G中最新产生的节点；M2已经在$Open$表中了，说明有不同父节点产生，需确定谁作为其父节点，如图中的$n4$需要确定父节点是n还是x；$M3$已经在$Closed$表中，由不同父节点产生，需确定谁为其父节点，此外要确定其生产的子节点是否有效。

状态空间的盲目搜索

一般树的盲目搜索主要包括广度优先算法和深度优先算法两种：

- 广度优先搜索：也称为宽度优先搜索，是一种先生成的节点先扩展的策略。这种策略的搜索过程是：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$n+1$层节点搜索。$Open$表中的节点总是按进入的先后顺序排序。
- 深度优先搜索：和广度优先搜索基本相同，主要区别在于$Open$表中的节点排序不同。在深度优先搜索中，最后进入$Open$表的节点总是排在最前面，即后生成的节点先扩展。

广度优先搜索是一种完备策略，即只要问题有解，它就一定可以找到解。并且，广度优先搜索找到的解，还一定是路径最短的解。缺点是盲目性太大，当目标节点和初始节点相距较远时，将产生许多无用的节点，从而效率较低。把树的分支因子（度）即树中最大的子节点数设为a，搜索深度为b，则时间复杂度为a^b，空间复杂度为a^b。

深度优先搜索是一种非完备策略，即对某些本身有解的问题，采用深度搜索可能找不到最优解，也可能根本找不到解。常用的解决办法就是增加一个深度限制，当搜索达到一定深度还没有找到解，停止深度搜索，向宽度发展（有界深度优先搜索）。时间复杂度为a^b，空间复杂度为a*b。

广度优先搜索的算法：

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。如是，则得到问题的解，成功推出；
5. 若节点n不可扩展，则转到第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，然后转向第2步。

这里的BFS和DFS与数据结构中的算法唯一区别是不一定要遍历所有节点，只要达到目标节点算法即结束。

代价树的盲目搜索

在一般树搜索策略中实际上进行了一种假设，认为状态空间中各边的代价都相同且为一个单位量。从而可用路径长度来代替路径的代价。对实际生活中的问题而言，它们的状态空间中的各个边的代价不可能相同，如城市交通问题，各城市之间的距离是不同的。

代价树及其代价计算：

通常我们把每条边上都标有其代价的树称为代价树。在代价树中，可以用$g(n)$表示初始节点S_0到节点n的代价，用(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有：

$$g(n_2)=g(n_1)+c(n_1,n_2)$$

通常，最短路径不一定是最小代价路径，最小代价路径不一定是最短路径。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路径。代价树也可以使用BFS和DFS。

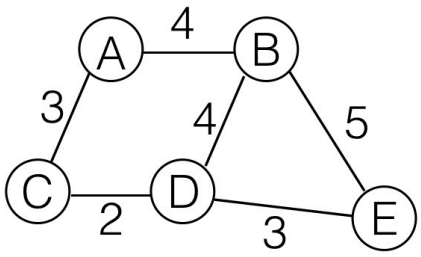
代价树的广度优先搜索

代价树的广度优先搜索也称为分枝界限法。它每次从$Open$表中选择节点或往$Closed$表中存放节点时，总是选择代价最小的节点。也就是说，$Open$表中的节点按照代价大小排序。

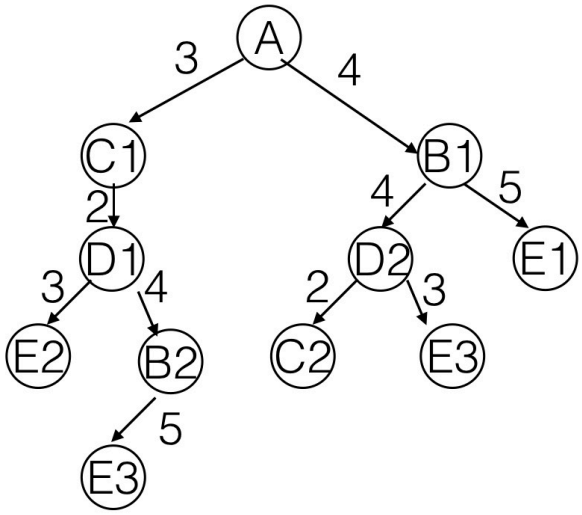
代价树的广度搜索算法如下：

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，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，如是，则找到问题的解，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$n_i(i=1,2,3,...)$，将这些子节点放入$Open$表中，并为每个子节点设置指向父节点的指针。按公式(4)计算各子节点的代价，并根据各节点的代价对$Open$表中的全部节点按从小到大的顺序重新排序。然后转第2步。

例：城市交通问题，图中数字表示两个城市之间的交通费用，即代价。用代价树广度优先搜索，求A到E费用最小的交通路线。



城市交通图



城市交通图的代价树

把一个网络图转化为代价树的方法是：从起始节点A开始，把与它直接邻接的节点作为其子节点。对其他节点也做同样处理。但当一个节点已作为某个节点的直系先辈节点时，就不能再作为这个节点的子节点。

按广度优先搜索可得最优解：A -> C -> D -> E

代价树的深度优先搜索

代价树的深度优先搜索和代价树的广度优先搜索的步骤也基本相同，它们之间的主要区别在于，每次选择最小代价节点的方法不同。广度优先搜索每次都是从$Open$表中的全体节点中选择一个代价最小的节点，而深度优先搜索则是从刚扩展的子节点中选择一个代价最小的节点。代价树的深度优先搜索策略也是一种非完备的策略。

状态空间的启发式搜索

启发式搜索，又叫做有信息搜索。利用问题领域的相关信息来帮助搜索，使得搜索朝着最有希望的方向前进，提高搜索效率。

启发性信息

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。启发性信息一般有以下三种：

1. 用于决定下一个要扩展的节点；
2. 用于决定产生哪些子节点；
3. 用于决定从搜索图中修剪或抛弃哪些节点。

估价函数

用来估计节点重要性的函数称为估价函数。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：

$$f(n)=g(n)+h(n)$$

其中：$g(n)$是从初始节点S_0到节点n，已经走过的路径代价估计通常即用实际花费代价，也比较容易计算。$h(n)$是从n到达目标S_g的最优路径的估计代价，必须根据问题特性，利用启发信息估计，搜索的启发性即体现在$h(n)$上，所以把$h(n)$称为**启发函数**。

A算法

又叫最好优先算法(Best First Search)，有序搜索(Ordered Search)。在图搜索算法中，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。

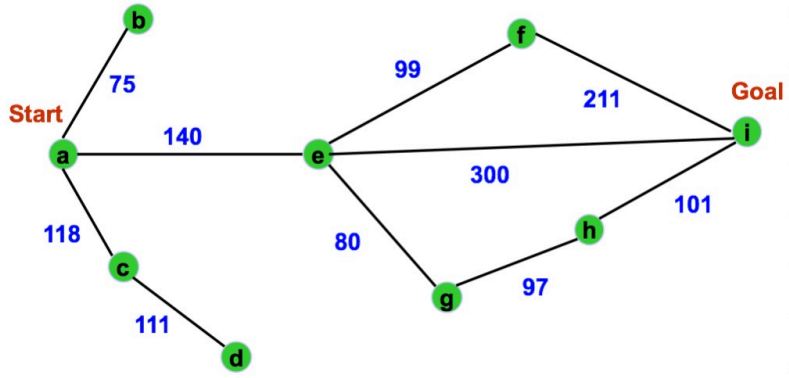
根据搜索过程中选择扩展节点的范围，启发式搜索算法可分为全局择优搜索算法和局部择优搜索算法。全局是每当需要扩展节点时总是从$Open$表的所有节点中选择一个估价函数值最小的节点进行扩展。局部则是每当需要扩展节点时，总是从刚生产的子节点中选择一个估价函数值最小的节点进行扩展。

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$n_i(i=1,2,...)$，计算每一个子节点的估价值$f(n_i)(i=1,2,...)$，并为每一个子节点设置指向其父节点的指针，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。

A算法是完备的，非最优的。时间复杂度和空间复杂度都为$O(a^b)$。

例：用A算法求从a城市出发到达i城市的路径。$g(n)$用从a城市到n城市走过的实际距离。$h(n)$由表格给出，可以设想为n到达i的直线距离。



当前城市	h(n)
a	366
b	374
c	329
d	244
e	253
f	178
g	193
h	98
i	0

用改造的$Open$表和$Closed$表表示求解过程。表中当前节点包含了从a的搜索路径，以及评估函数值。如aeg(413)表示当前节点为g，搜索路径为从a到e，在到g，评估函数值$f(g)=g(g)+h(g)$。其中$g(g)$为从a到e，再到g两段路的实际距离之和，即$g(g)=140+80=220$。$h(g)$为g到i的启发函数值，查表的$h(g)=193$。即$f(g)=g(g)+h(g)=220+193=413$.

求解过程：

OPEN表	当前扩展	CLOSED表
a(366)	a	a
ab(449),ac(447),ae(393)	e	a, e
ab(449),ac(447),aei(440),aef(417),aeg(413)	g	a, e, g
ab(449),ac(447),aei(440),aef(417),aegh(415)	h	a, e, g, h
ab(449),ac(447),aei(440),aef(417),aeghi(418)	f	a, e, g, h, f
ab(449),ac(447), aei(440),aeghi(418),aefi(450)	i	a, e, g, h, f, i

虽然本题得到的解为aeghi为最优解，但**A算法不保证能取得全局最优解**。

爬山搜索算法

爬山搜索算法是一种贪婪算法。评估函数$f(n)=g(n)+h(n)$中，令$g(n)=0$，即$f(n)=h(n)$，A算法即成为爬山搜索算法。

优点：搜索效率高

缺点：常常陷入局部最优，而失去全局最优。

使用爬山搜索算法求解A算法例题，求解过程如下：

OPEN表	当前扩展	CLOSED表
a(366)	a	a
ab(374),ac(329),ae(253)	e	a, e
ab(374),ac(329),aei(0),aef(178), aeg(193)	i	a, e, i

爬山搜索得到的解为aei，可见不是全局最优解，但是和A算法相比，爬山搜索效率高。

等代价搜索

即：Dijkstra算法，评估函数$f(n)=g(n)+h(n)$中，令$h(n)=0$，$f(n)=g(n)$,且$g(n)$为实际路径代价，此时A算法称为等代价搜索，因为$h(n)=0$，没有了启发性，所以属于盲目搜索。

性能：最优性，即找到全局最优解。完备性。时空复杂度同A算法。

问题：搜索效率低。

等代价搜索求解A算法例题，求解过程如下：

OPEN表	当前扩展	CLOSED表
a(0)	a	a
ab(75),ac(118),ae(140)	b	a,b
ac(118),ae(140)	c	a,b,c
ae(140),acd(229)	e	a,b,c,e
acd(229),aeg(220),aef(239),aei(440)	g	a,b,c,e,g
acd(229), aef(239),aei(440),aegh(317)	d	a,b,c,e,g,d
aef(239),aei(440),aegh(317)	f	a,b,c,e,g,d,f
aei(440),aegh(317),aefi(450)	h	a,b,c,e,g,d,f,h
aei(440),aefi(450),aeghi(418)	i	a,b,c,e,g,d,f,h,I

A*算法

如上所说的A算法都没有对估价函数$f(n)$做任何限制。实际上，估价函数对搜索过程十分重要，若选择不当，则有可能找不到问题的解，或者找不到最优解。A*算法就是对估价函数加上一些限制后得到的一种启发式搜索算法。

假设$f^*(n)$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的最小代价值。估价函数$f(n)$则是$f^*(n)$的估计值。显然，$f^*(n)$应由一下两个部分组成：一部分是从初始节点S_0到节点n的最小代价，记为$g^*(n)$；另一部分是从节点n到目标节点S_g的最小代价，记为$h^*(n)$，当问题有多个目标节点时，应选取一种代价最小的一个。因此：

$$f^*(n)=g^*(n)+h^*(n)$$

- $g(n)$是对$g^*(n)$的估计，但$g(n)$不一定就是从初始节点S_0到节点n的真正最小代价，很有可能从初始节点S_0到节点n的真正代价还没有找到，故$g(n)\geq g^*(n)$且$g(n)\geq 0$。
- $h(n)$是$h^*(n)$的下界，即对任意节点均有$h(n)\leq h^*(n)$。

A*算法特点：

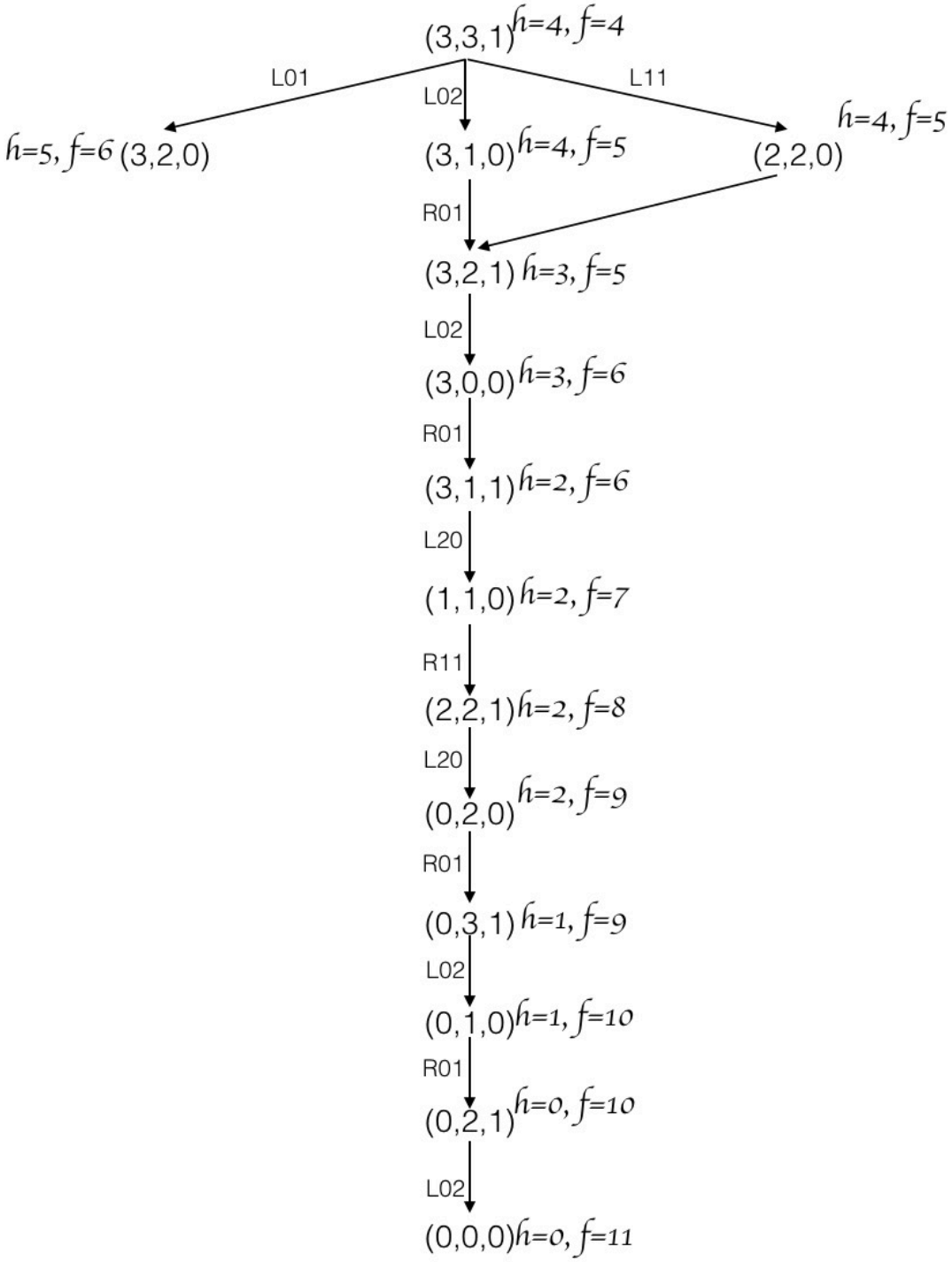
- 最优性：保证最优解存在时能找到最优解。
- 可采纳性：算法能在有限步终止，并能找到最优解。

例：修道士和野人问题。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(n,c,b)表示问题的状态。

对A*算法，首先确定估价函数。设$g(n)=d(n),h(n)=m+c-2b$，则有：

$$f(n)=g(n)+h(n)=d(n)+m+c-2b$$

其中，$d(n)$为节点的深度。可知$h(n)\leq h^*(n)$满足A*算法限制条件。搜索图如下：（符号L_{ij}表示从左岸到右岸的运人操作，R_{ij}表示从右岸到左岸的运人操作，i和j分别表示传上的修道士和野人数）



修道士和野人问题的搜索树

小结

- $f(n)=g(n), h(n)=0, g(n)$为实际路径代价：等价搜索，效率不高，能得到最优解。
- $f(n)=h(n), g(n)=0$：爬山搜索，效率极高，往往陷入局部最优。
- $f(n)=g(n)+h(n), g(n)>0, h(n)>0$：A算法，设计估计函数时要充分考虑$g(n)$和$h(n)$的数量级相当。否则当$g(n)>>h(n)$时，接近等价搜索；$h(n)>>g(n)$时，接近爬山搜索。
- $f(n)=g(n)+h(n), g(n)>0$且$g(n)\geq g^*(n), h(n)\leq h^*(n)$：A*算法，为了兼顾搜索效率，在满足上述条件前提下，要尽可能取较大的$h(n)$值，尽可能提高搜索效率。
- $f(n)=0$即$g(n)=0, h(n)=0$：盲目搜索。

问题归约表示法

问题归约法是不同于状态空间方法的另一种形式化方法，其基本思想是对问题进行分解或变换。当一个问题比较复杂时，如果直接求解往往比较困难，此时可通过分解或变换把它转化为一系列简单的问题，然后通过对这些简单问题的求解来实现对原问题的求解。

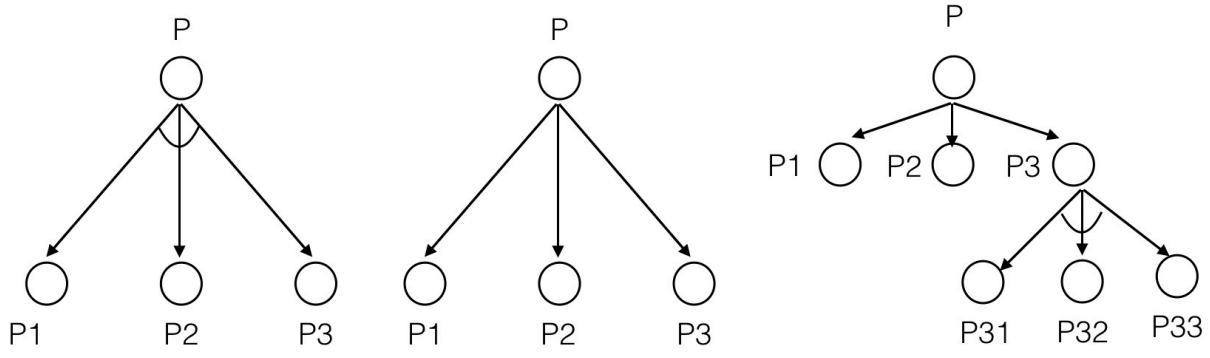
问题的分解与等价变换

- 分解：如果一个问题P可以归约为一组子问题$P_1, P_2, P_3, \ldots, P_n$，并且只有当所有子问题$P_i(i=1,2,\ldots,n)$都有解时，原问题$P$才有解，任何一个子问题$P_i(i=1,2,\ldots,n)$无解，都会导致原问题无解。则称此种归约为问题的分解，即分解所得到的子问题的“与”和原问题P等价。
- 等价变换：如果一个问题P可以归约为一组子问题$P_1, P_2, \ldots, P_n$，并且这些子问题$P_i(i=1,2,\ldots,n)$中只要有一个有解，原问题P就有解，只有当所有子问题$P_i(i=1,2,\ldots,n)$都无解时，原问题P才无解，则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的“或”与原问题P等价。

实际问题中，有可能需要同时采用变换和分解。无论变换还是分解，都是将原问题归约为一系列本原问题。所谓本原问题**是指那种不能（或不需要）再进行分解或变换，且可以直接回答的问题**。本原问题可以作为终止归约的限制条件。

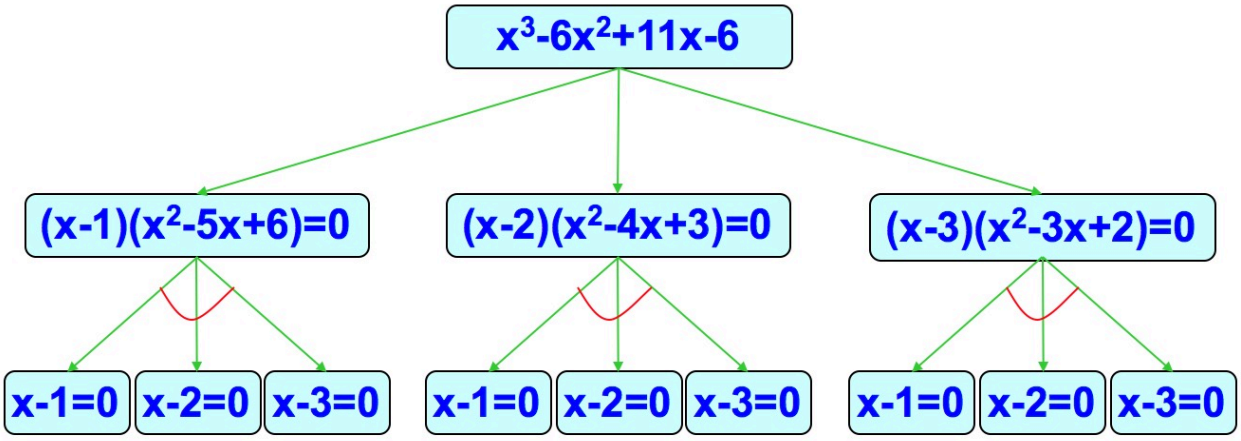
问题归约的与/或树表示

- 与树：原问题P作为父节点，分解的子问题$P_i(i=1,2,\ldots,n)$作为子节点，原问题的可解性与子问题的可解性呈“与”关系。
- 或树：原问题P作为父节点，分解的子问题$P_i(i=1,2,\ldots,n)$作为子节点，原问题的可解性与子问题的可解性呈“或”关系。
- 与或树：即用到分解又用到变换。如图：



与树　　或树　　与或树

例：用与或树表示解方程$x^3-6x^2+11x-6=0$的过程。



- 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：
  - 任何终止节点都是可解节点。
  - 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。
  - 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，用类似的方法定义不可解节点：
  - 不为终止节点的端节点是不可解节点。
  - 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。
  - 对于“与”节点，只要其子节点中有一个为不可解节点，则该节点是不可解节点。
- 解树：由可解节点构成，并且由这些可解节点可以推出初始节点为可解节点的子树为解树。上面的例题就有三棵解树。

与或树的启发式搜索

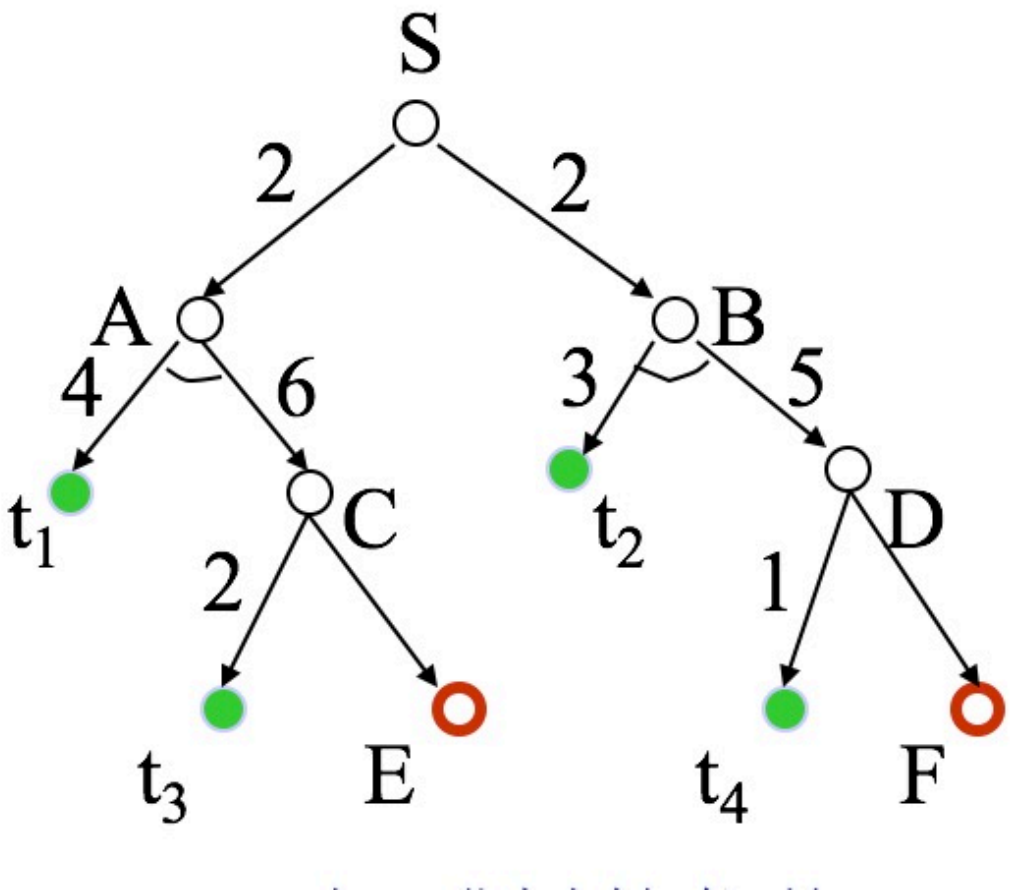
与或树的启发式搜索过程是一种利用搜索过程所得到的启发性信息寻找最优解树的过程。对搜索的每一步，算法都试图找到一个最有希望成为最优解树的子树。最优解树是指代价最小的那棵解树。

解树的代价

在与或树的启发式搜索过程中，解树的代价可按如下方法计算：

1. 若n为终止节点，则其代价$h(n)=0$。
2. 若n为或节点，且子节点为$n_1,n_2,\cdots,n_k$，则n的代价为$h(n)=min\{c(n,n_i)+h(n_i)\},(1\leq i\leq k)$，式中$c(n,n_i)$是节点$n$到其子节点$n_i$的边代价。
3. 若n为与节点，且子节点为$n1,n2,\cdots,n_k$，则n的代价可用和代价法或最大代价法计算。若用和代价法，则其计算公式为：$h(n)=\sum_{i=1}^{n}[c(n,n_i)+h(n_i)]$；若用最大代价法，则其计算公式为：$h(n)=max_{1\leq i<<k}\{c(n,n_i)+h(n_i)\}$。
4. 若n是端节点，但不是终止节点，则n不可扩展，其代价定义为$h(n)=\infty$。
5. 根节点的代价即为解树的代价。

例：设下图是一棵与/或树，它包括两可解树，左边的解树由S、A、t_1、C及t_3组成；右边的解树由S、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、F是不可解端节点；边上的数字是该边的代价。请计算解树的代价。



先计算左边的解树

按和代价：$h(S)=2+4+6+2=14$

按最大代价：$h(S)=(2+6)+2=10$

再计算右边的解树

按和代价：$h(S)=1+5+3+2=11$

按最大代价：$h(S)=(1+5)+2=8$

希望树

为了找到最优解树，搜索过程中的任何时刻都应该选择那些最有希望成为最优解树的一部分的节点进行扩展。由于这些节点及其父节点所构成的与或树最有可能成为最优解树的一部分，因此称它为希望解树，简称希望树。希望解树在搜索过程中不断变化的。希望树的定义如下：

希望解树T

1. 初始节点S_0在希望树T中；
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在希望解树T中的充分必要条件是：$min_{1\le i\le k}\{c(n,n_i)+h(n_i)\}$。
3. 如果n是与节点，则n的全部子节点都在希望解树T中。

与或树的启发式搜索过程

与或树的启发式搜索需要不断的选择、修正希望树，其搜索过程如下：

1. 把初始节点S_0放入$Open$表中，计算$h(S_0)$。
2. 计算希望解树T。
3. 依次在$Open$表中取出T的端节点，放入$Close$表，并记该节点为n。
4. 如果节点n为终止节点，则做下列工作：
    - 标记节点n为可解节点；
    - 在T上应用可解标记过程，对n的先辈节点中的所有可解节点进行标记；
    - 如果初始节点S_0能够被标记为可解节点，则T就是最优解树，成功退出；
    - 否则，从$Open$表中删去具有可解先辈的所有节点；
    - 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：
    - 扩展节点n，生成n的所有子节点；
    - 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；
    - 计算这些子节点及其先辈节点的h值；
    - 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：
    - 标记节点n为不可解节点；
    - 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节点进行标记；
    - 如果初始节点S_0能够被标记为不可解节点，则问题无解，失败退出；
    - 否则，从$Open$表中删去具有不可解先辈的所有节点；
    - 转第2步。
