第3章 搜索基本策略

盲目搜索

启发式搜索

随机搜索

问题求解算法的性能

（1）完备性：如果存在一个解答，该策略是否保证能够找到并结束？

（2）最优性：如果存在不同的几个解答，该策略可以发现是否最高质量的解答？

（3）时间复杂性：需要多长时间可以找到解答？

（4）空间复杂性：执行搜索需要多大存储空间？

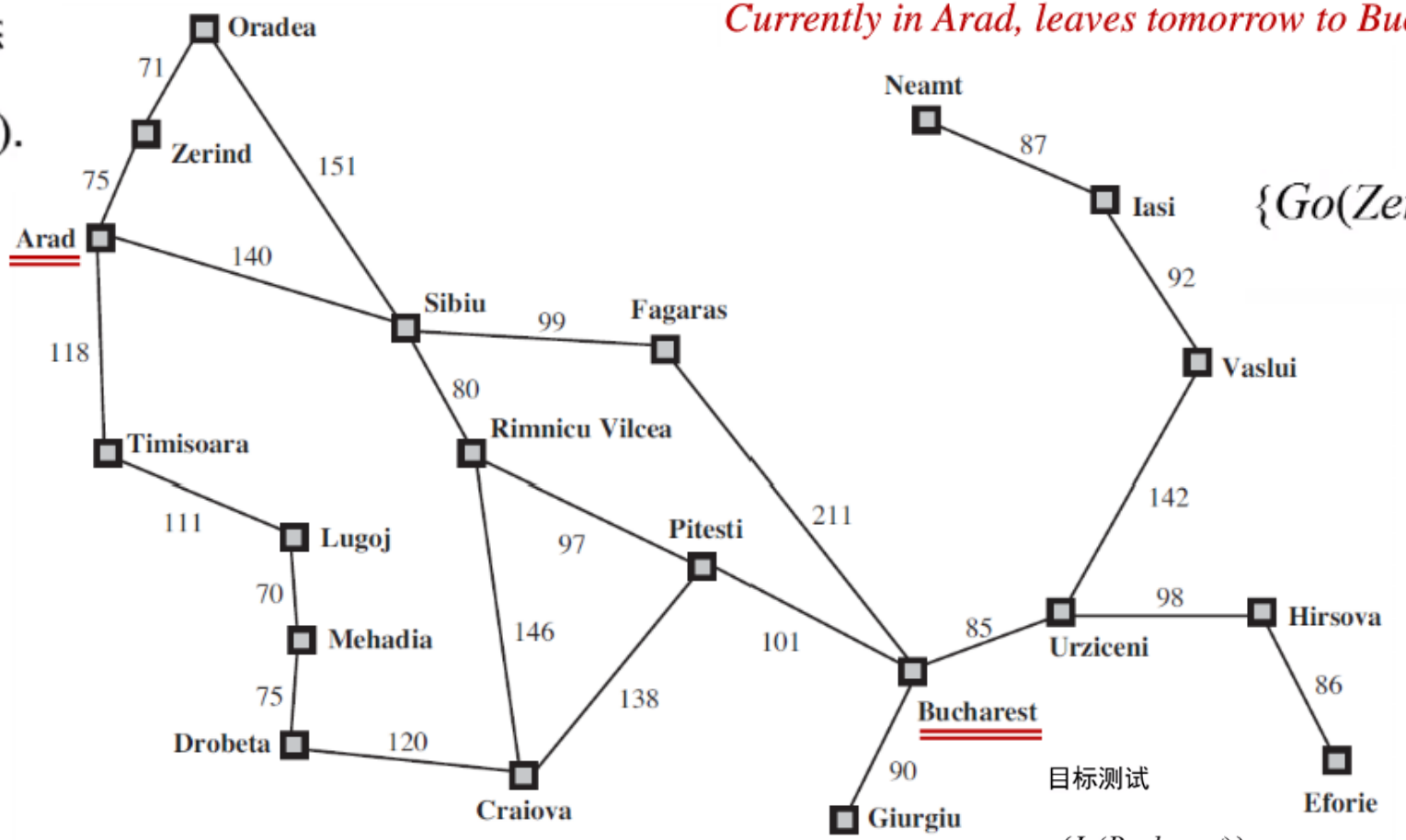
时间复杂性和空间复杂性用如下术语来度量：

- b -- maximum branching factor of the search tree.
  搜索树的最大分支因子。
- d -- depth of the shallowest solution.
  最浅解的深度。
- m -- maximum depth of the search tree.
  搜索树的最大深度。

罗马尼亚问题

初始状态

In(Arad).



Currently in Arad, leaves tomorrow to Bucharest

动作

{Go(Zerind), Go(Sibiu), Go(Timisoara)}.

目标测试

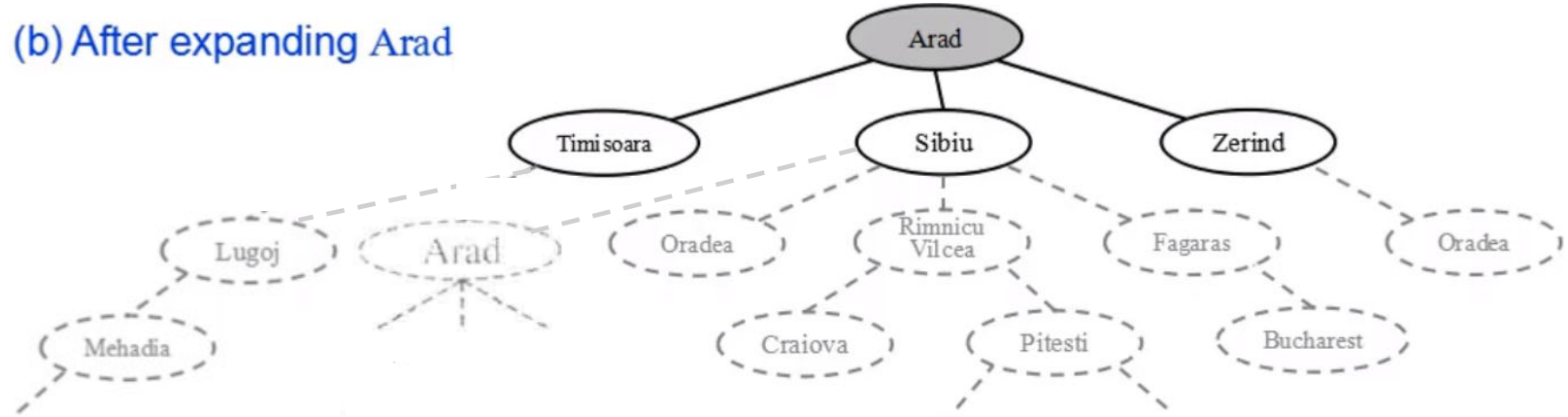
{In(Bucharest)}.

Tree Search 树搜索

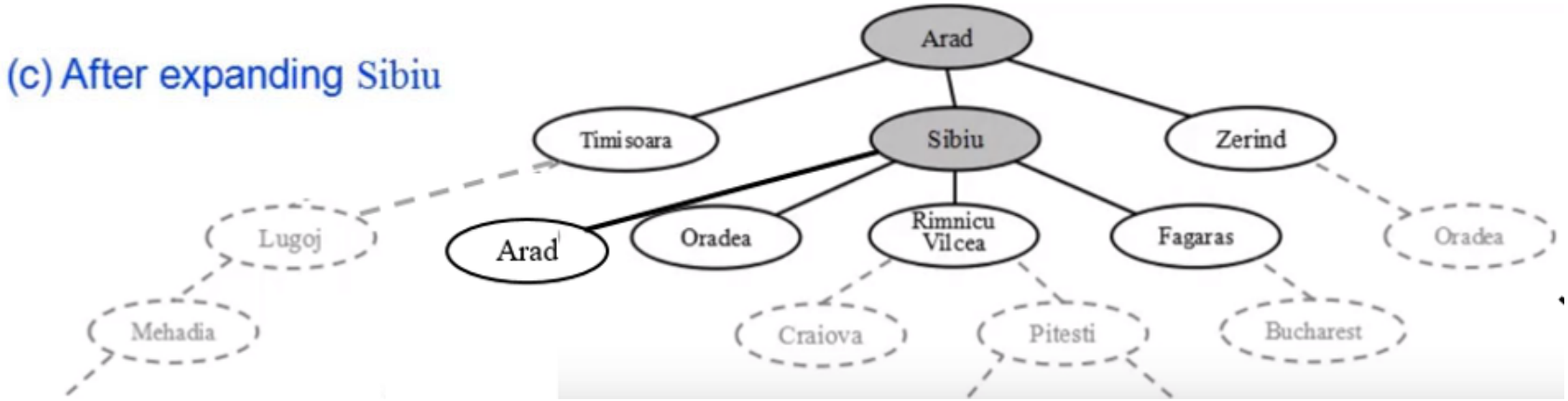
Shaded: the nodes that have been expanded.
阴影：表示该节点已被扩展。
Outlined: the nodes that have been generated but not yet expanded.
粗实线：表示该节点已被生成，但尚未扩展。
Faint dashed lines: the nodes that have not been generated.
浅虚线：表示该节点尚未生成。

Tree Search 树搜索

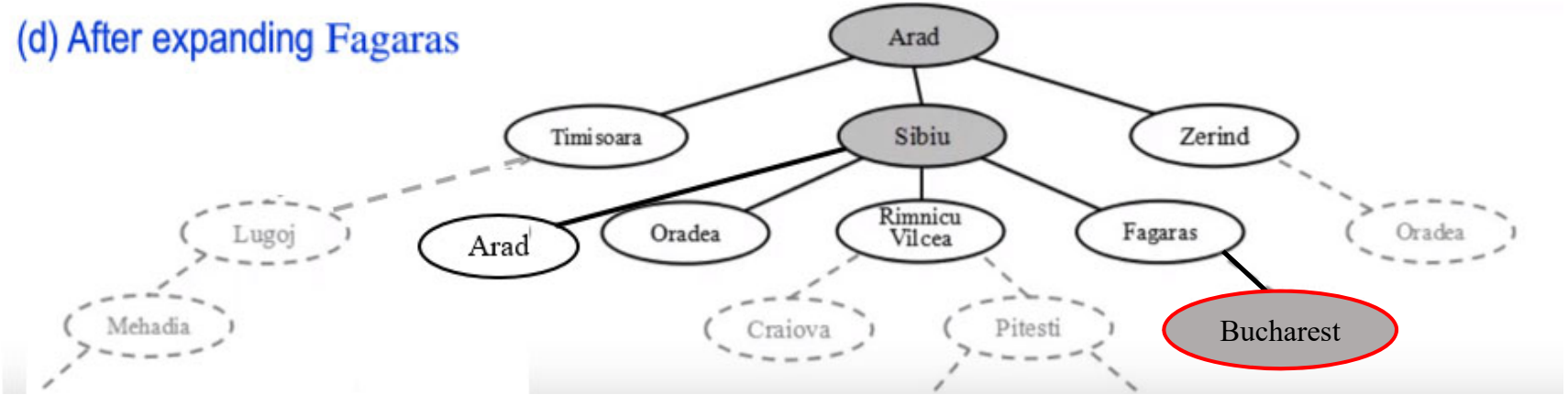
(b) After expanding Arad

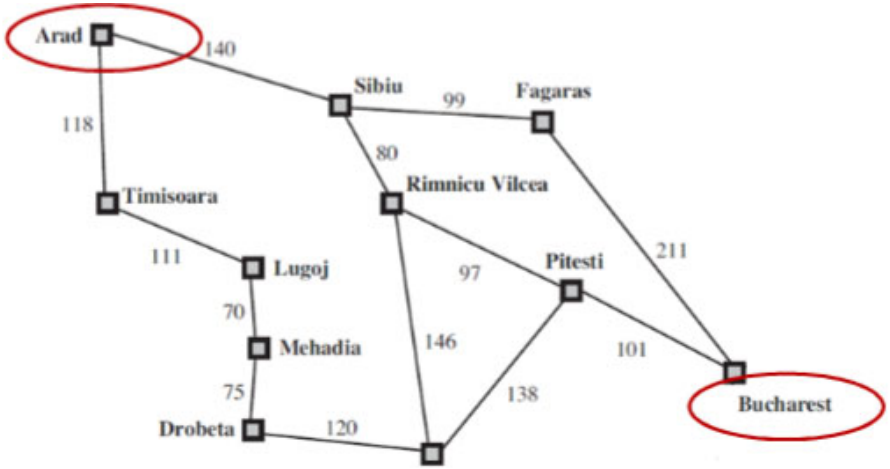


(c) After expanding Sibiu



Tree Search 树搜索





A General Tree-search Algorithm　一种通用的树搜索算法

```
function TREE-SEARCH(problem) returns a solution, or failure
    initialize the frontier using the initial state of problem
    loop do
        if the frontier is empty then return failure
        choose a leaf node and remove it from the frontier
        if the node contains a goal state then return the corresponding solution
        expand the chosen node, adding the resulting nodes to the frontier
```

The *frontier* (also known as *open list*): an data structure, to store the set of all leaf nodes.

该*frontier*（亦称*open list*）：一种数据结构，用于存储所有的叶节点。

The process of expanding nodes on the *frontier* continues until either a solution is found or there are no more states to expand.

在*frontier*上扩展节点的过程持续进行，直到找到一个解、或没有其它状态可扩展。

A General Tree-search Algorithm　存在的问题

In（Arad）是搜索树中的重复状态，生成了一个有环的路径。这样的有环路径意味着罗马尼亚问题的完整搜索树是无限的。这样会导致算法失败，使有解的问题无法求解。

有环路径是冗余路径的一种特殊情况。当两个状态之间的迁移路径不止一条时，这种情况就会发生。如果只关心最终目标，那么对到达任一给定状态都没有必要记录超过一条的路径。

A General Graph-search Algorithm　一种通用的图搜索算法

```
function GRAPH-SEARCH (problem) returns a solution, or failure
    initialize the frontier using the initial state of problem
    initialize the explored to be empty
    loop do
        if the frontier is empty then return failure
        choose a leaf node and remove it from the frontier
        if the node contains a goal state then return the corresponding solution
        add the node to the explored
        expand the chosen node, adding the resulting nodes to the frontier
            only if not in the frontier or explored
```

The *explored* (aka *closed list*) is an data structure to remember every expanded node.

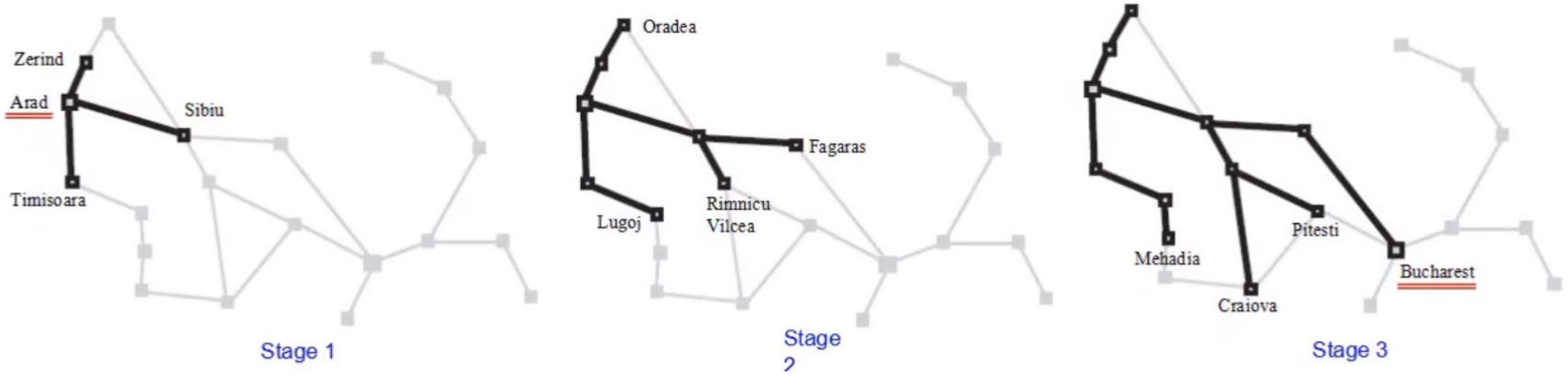
该*explored*（亦称*closed list*）：一种数据结构，用于记忆每个扩展节点。

The nodes in the *explored* or the *frontier* can be discarded.

*explored*或*frontier*中的节点可以被丢弃。

Graph Search 图搜索

通过图搜索在该罗马尼亚地图上生成一系列搜索路径。



Each path has been extended at each stage by one step. Notice that at 3rd stage, the northernmost city (Oradea) has become a dead end.

两种搜索的比较

对于一个实际搜索问题，应该采用什么形式来表示，是用图还是用树，可以作如下比较：

（1）用树结构，允许搜索图中有相同结点出现

优点：控制简单。因为不用记住曾经走过的路或者扩展失败的结点。

缺点：占空间较大，产生相同结点多，则时空均需要较大的代价。有时会探索到冗余的路径（比如说有环路径），这样就会找不到解。

两种搜索的比较

（2）用图结构，不允许搜索图中有相同结点出现

优点：节省大量空间(相同的结点只存一次）和时间(相同结点不需要重复产生）。

缺点：每产生一个新的结点需判断这个节点是否已生成过，因而控制更复杂，判断也要占用时间。

碰到具体问题时，要权衡二者的利弊。若可能产生大量相同结点，则应采用搜索图。

3.1 盲目的搜索方法

盲目搜索方法又叫非启发式搜索，是按预定的搜索策略进行搜索。该方法具有很大的盲目性，效率不高，不便于复杂问题的求解。

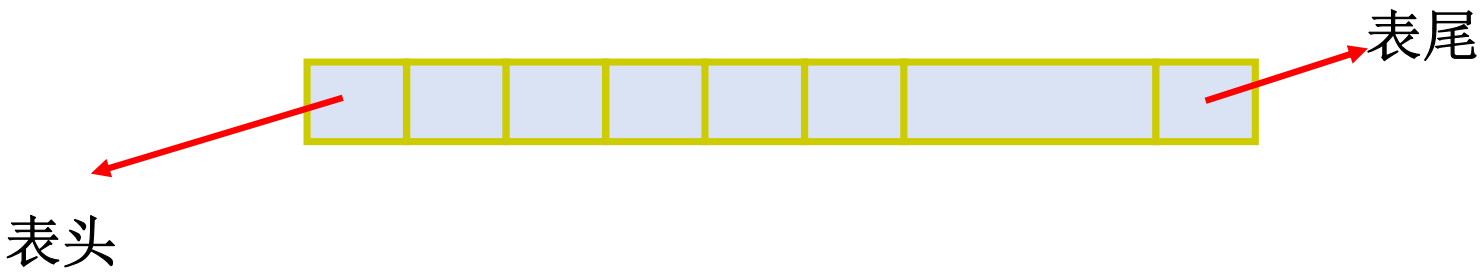
- 宽度优先搜索
- 一致代价搜索
- 深度优先搜索
- 迭代加深搜索

AI操作中的表

- 这里的搜索算法存放节点都采用简单的数据结构：表。
- 假定列表是由字符串组成。每个字符串包含一个或多个字符。在大部分情况下，仅需按其在表中出现的顺序依次移去某些字符。
- 主要操作将是取出下一个符号，这意味着将此符号从表中移去。

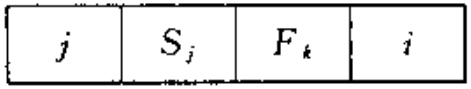
OPEN表和CLOSED表

- open表记载搜索过程中尚未考查的节点和新生成的节点；
- open表支持按指定要求对表中节点排序；
- 搜索算法每次循环时都将open表中的第一个节点移送到closed表的表尾；
- closed表记载搜索过程中已被考查过的节点。



表的内容

open表和closed表的节点的域可如下定义：



j：节点j的序号(通常是节点生成的顺序编号)

S_j：节点j的状态

F_k：生成节点j所使用的算符编号(算符名称)

i：节点j的父节点序号(或是节点j指向父节点i的指针)。

- *n*.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；

Open表适合的数据结构

1.队列：FIFO

2.栈: LIFO

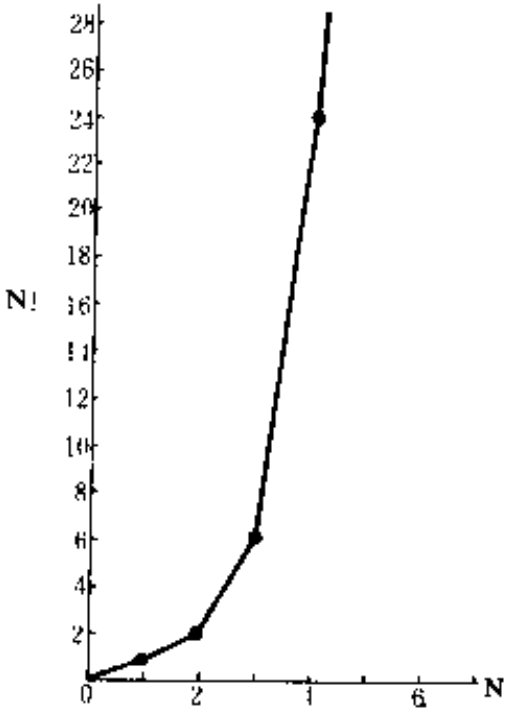
3.优先队列：根据函数计算优先级

- EMPTY?(*queue*) 返回值为真当且仅当队列中没有元素。
- POP(*queue*) 返回队列中的第一个元素并将它从队列中删除。
- INSERT(*element*, *queue*) 在队列中插入一个元素并返回结果队列。

搜索与组合爆炸

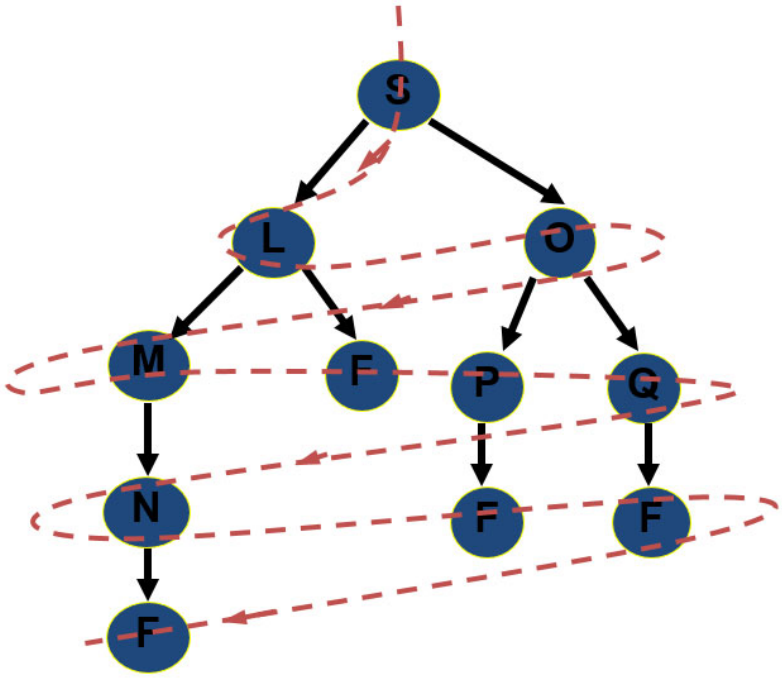
- 3！=3*2*1
- 4！=4*3*2*1
- 。
- 。
- 。
- N!=

指随着优化问题规模的不断增大，决策变量取值的不同组合量、可行解数量以及寻找最优解时需要考虑的组合量也会迅速大幅度增加，且往往是以指数形式增加，最终导致无法从中找到最优解。



宽度优先搜索Breadth-first Search

如果搜索是以同层邻近节点依次扩展节点的，那么这种搜索就叫宽度优先搜索，这种搜索是逐层进行的，在对下一层的任一节点进行搜索之前，必须搜索完本层的所有节点。



宽度优先搜索算法

(1) 初始化open表与closed表，把初始节点(序号1)放入open表中。

(2) 若open＝()，则算法失败终止；否则，转步骤(3)。

(3) 把open表的第1个节点i移出，放入closed表的表尾。

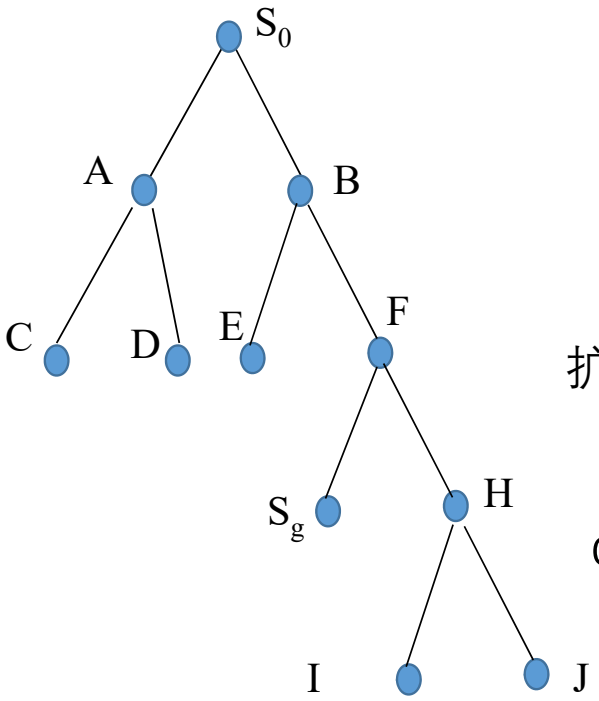
(4) 若节点i的状态S_i是目标状态，则算法成功终止；否则，转步骤(5)。

(5) 若节点i 不可扩展(即在算符集中找不到可用算符)，则转步骤(2)；否则，转步骤(6)。

(6) 逐一用可用算符扩展节点，生成i的所有子节点。若子节点j的状态S_j既不在open表中，也不在closed表中，则节点j是一个新节点。将所有的新子节点逐一放入open表的表尾，并记载子节点各个域的值，转步骤(2)；否则，不把新生成的节点j放入open表中，放弃节点j，转步骤(2)。

Breadth-first Search Algorithm on a Graph 图的宽度优先搜索算法

```
function BREADTH-FIRST-SEARCH(problem) returns a solution, or failure
    node ← a node with STATE = problem.INITIAL-STATE
    PATH-TEST = 0
    frontier ← a FIFO queue with node as the only element
    explored ← an empty set
    loop do
        if EMPTY ? (frontier) then return failure
        node ← POP(frontier)          /* chooses the shallowest node in frontier */
        add node.STATE to explored
        for each action in problem.ACTIONS(node.STATE) do
            child ← CHILD-NODE(problem, node, action)
            if child.STATE is not in explored or frontier then
                if problem.GOAL-TEST(child.STATE) then return SOLUTION(child)
                frontier ← INSERT(child, frontier)
```



开始：S_0初始状态放在表尾

Open ：| s_0 |

closed ：| s_0 |

扩展S_0：

Open ：| A | B |

closed ：| s_0 | A |

扩展A：

Open ：| B | C | D |

closed ：| s_0 | A | B |

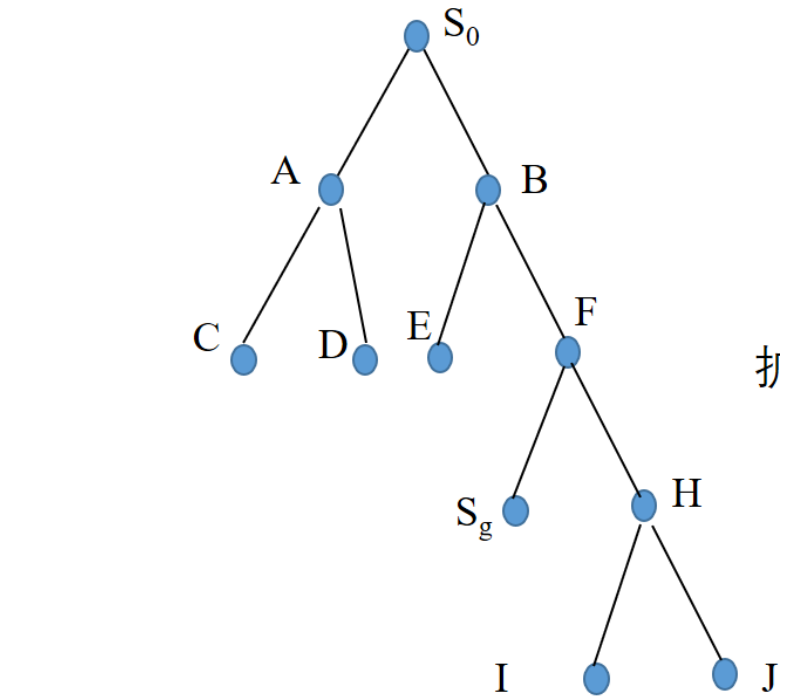
……

closed ：| s_0 | A | B | C | D | E | F | s_g |

open ：| s_g | H |

宽度优先搜索算法的性质

- **完备性：** 满足
- **最优性：** 不一定
- **时间复杂度：** $b + b^2 + b^3 + \ldots + b^d = O(b^d)$
- **空间复杂度：** $O(b^d)$



closed：

	s_0	A	B	C	D	E	F	s_g

open：

							s_g	H

时间和内存需求

Time and Memory Requirements 时间和内存需求

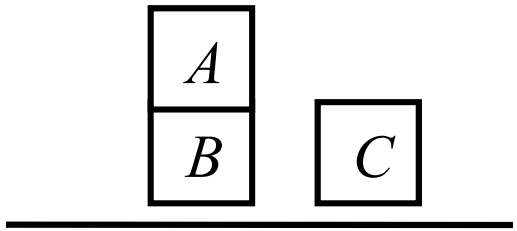
Depth	Nodes	Time	Memory
2	110	.11 milliseconds	107 kilobytes
4	11,110	11 milliseconds	10.6 megabytes
6	10^6	1.1 seconds	1 gigabyte
8	10^8	2 minutes	103 gigabytes
10	10^{10}	3 hours	10 terabytes
12	10^{12}	13 days	1 petabyte
14	10^{14}	3.5 years	99 petabytes
16	10^{16}	350 years	10 exabytes

Assume: branching factor b = 10; 1 million nodes/second; 1000 bytes/node.

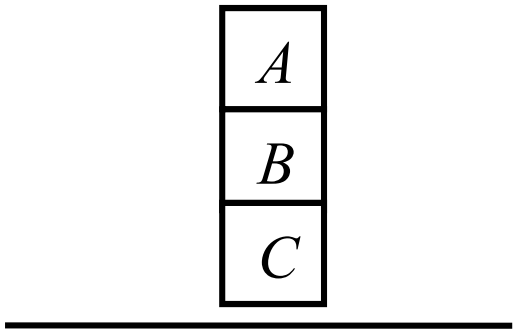
- Memory requirements are a bigger problem, execution time is still a major factor.
  内存的需求是一个很大的问题，而执行时间仍是一个主要因素。
- Breadth-first search cannot solve exponential complexity problems but small branching factor.
  宽度优先搜索不能解决指数复杂性的问题，小的分支因子除外。

积木问题

- 例 通过搬动积木块，希望从初始状态达到一个目的状态，即三块积木堆叠在一起。



(a) 初始状态



(b) 目的状态

积木问题

- 操作算子为*MOVE*（*X*，*Y*）：把积木*X*搬到*Y*（积木或桌面）上面。

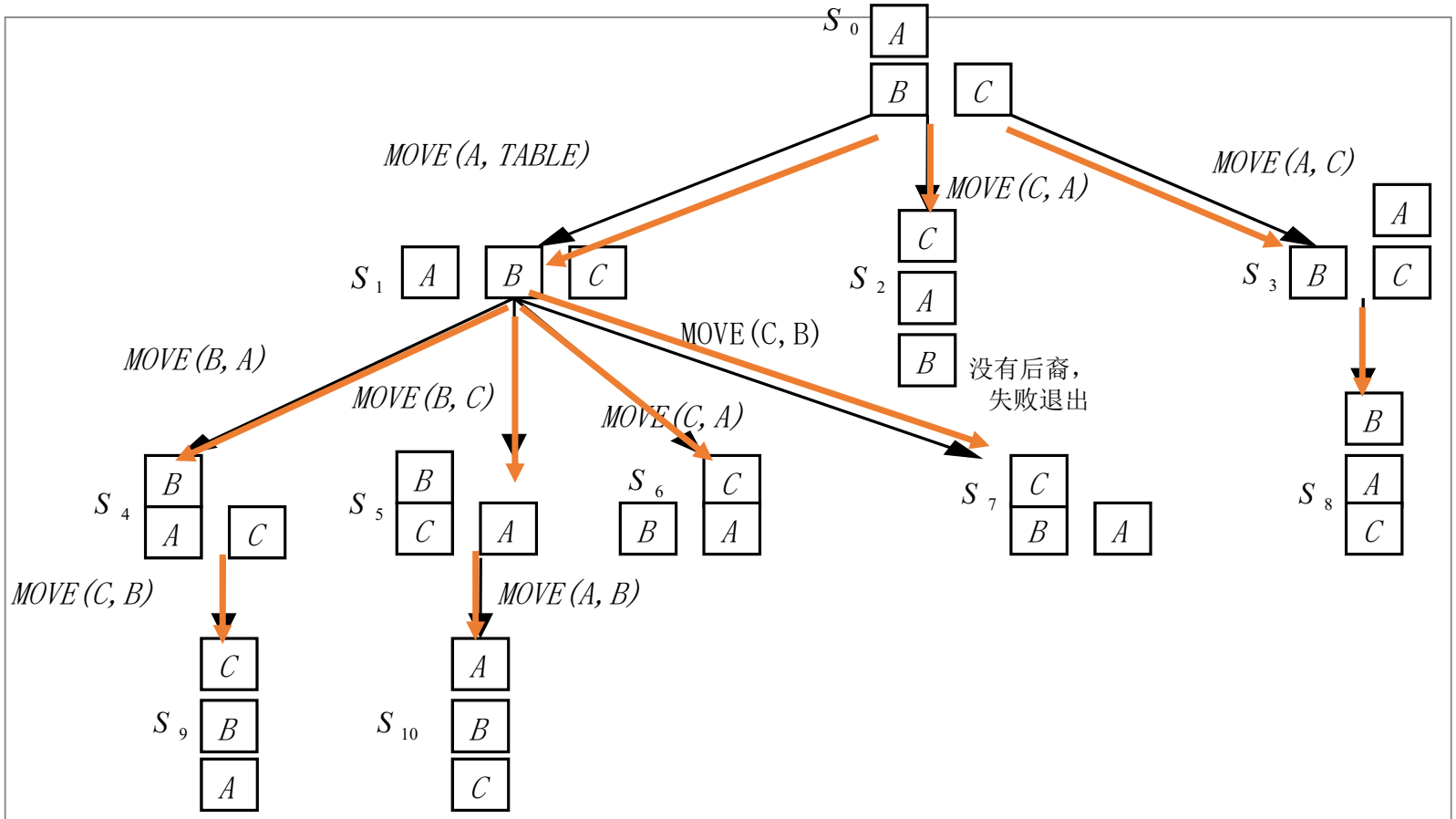
MOVE（*A*，*Table*）：“搬动积木*A*到桌面上”。

- 操作算子可运用的先决条件：

（1）被搬动积木的顶部必须为空。

（2）如果 *Y* 是积木，则积木 *Y* 的顶部也必须为空。

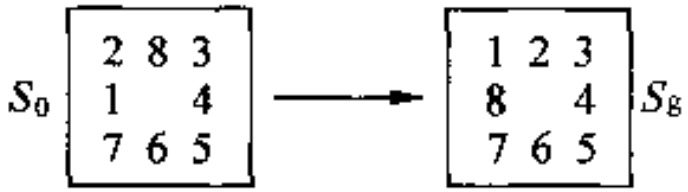
（3）同一状态下，运用操作算子的次数不得多于一次。



积木问题的宽度优先搜索

重排九宫格问题

- 在3x3的方格棋盘中放置1，2，3，4，5，6，7，8八个数字，其中有一个方格是空的。如图所示，设初始状态为S_0，目标状态为S_g，寻找从初始状态到目标状态的路径。



问题空间为：

$$\begin{matrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ a_{31} & a_{32} & a_{33} \end{matrix}$$

重排九宫格问题

各状态间的转换规则为：

$$\begin{matrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ a_{31} & a_{32} & a_{33} \end{matrix}$$

Pr1: 空格上移 ↑

If $\square_{i,\ j}$ and $i \neq 1$ then $a_{i-1,j} \longleftrightarrow \square_{i,\ j}$

Pr2: 空格下移 ↓

If $\square_{i,\ j}$ and $i \neq 3$ then $a_{i+1,j} \longleftrightarrow \square_{i,\ j}$

Pr3: 空格左移 ←

If $\square_{i,\ j}$ and $j \neq 1$ then $a_{i,j-1} \longleftrightarrow \square_{i,\ j}$

Pr4: 空格右移 →

If $\square_{i,\ j}$ and $j \neq 3$ then $a_{i,j+1} \longleftrightarrow \square_{i,\ j}$

- 宽度优先搜索算法搜索过程：

1. Open：s_0

   closed：s_0

2. Open：2 3 4 5

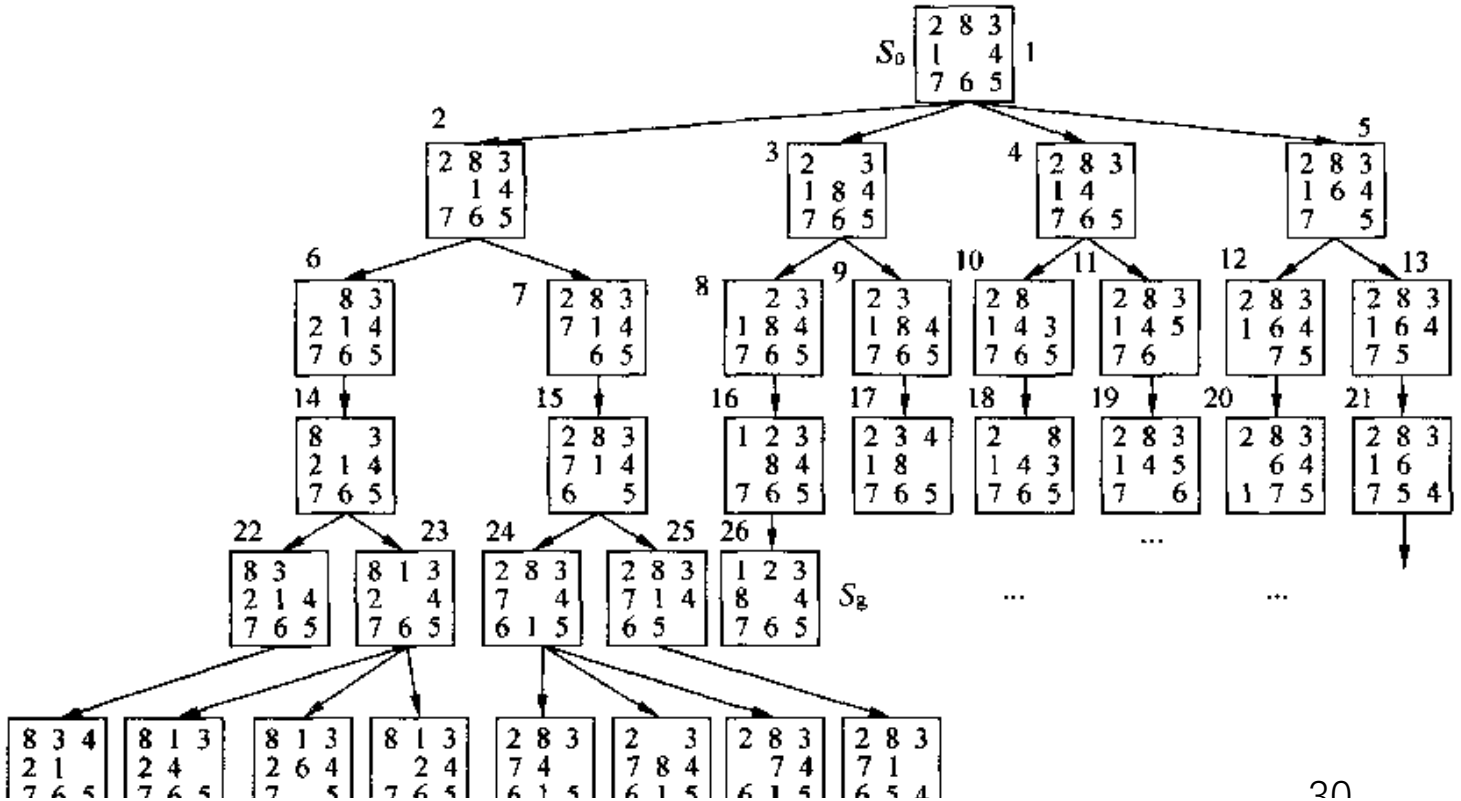
   closed：s_0 2

3. Open：3 4 5 6 7

   closed：s_0 2 3

4. Open：4 5 6 7 8 9

   closed：s_0 2 3 4



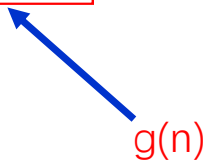
30

宽度优先搜索的特点

- 宽度优先搜索的优点是：若问题有解，当每一步行动代价都相等时，则可找出最优解。
- 宽度优先搜索的缺点是:效率低，组合爆炸问题难以解决。

Uniform-cost Search一致代价搜索

- 搜索策略：扩展最低代价的未扩展节点
- 实现方法：优先队列，按路径代价排序，最低优先



一致代价搜索算法

```
function UNIFORM-FIRST-SEARCH(problem) returns a solution, or failure
  node ← a node with STATE = problem.INITIAL-STATE, PATH-COST = 0
  frontier ← a priority queue ordered by PATH-COST, with node as the only element
  explored ← an empty set
  loop do
    if EMPTY ? (frontier) then return failure
    node ← POP(frontier)          /* chooses the lowest-cost node in frontier */
    if problem.GOAL-TEST(node.STATE) then return SOLUTION(node)
    add node.STATE to explored
    for each action in problem.ACTIONS(node.STATE) do
      child ← CHILD-NODE(problem, node, action)
      if child.STATE is not in explored or frontier then
        frontier ← INSERT(child, frontier)
      else if child.STATE is in frontier with higher PATH-COST then
        replace that frontier node with child
```

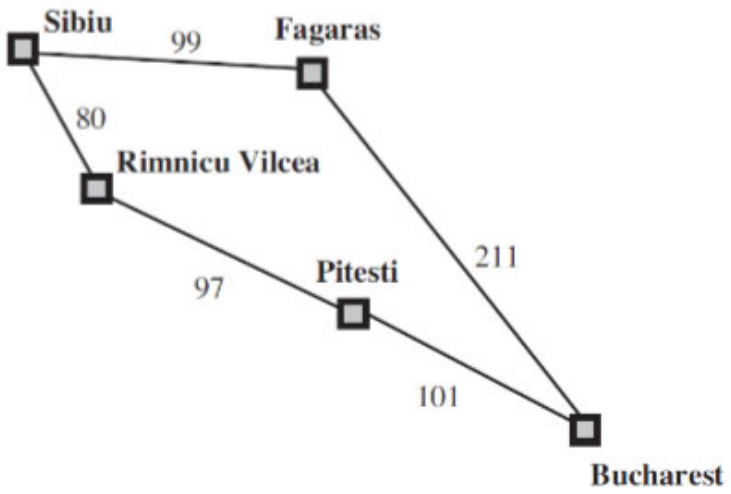
从Sibiu到Bucharest

- From Sibiu to Bucharest, least-cost node, Rimnicu Vilcea, is expanded, next, adding Pitesti with cost $80 + 97 = 177$.
  从Sibiu到Bucharest，扩展最低代价节点Rimnicu Vilcea，然后加上Pitesti的代价

- The least-cost node is now Fagaras, and adding goal node Bucharest with cost $99 + 211 = 310$.
  现在最低代价节点为Fagaras，加上目标节点Bucharest的代价

- Choosing Pitesti and adding a second path to Bucharest with cost $177 + 101 = 278$.
  选择Pitesti并加上第二条路径到Bucharest的代价

- This new path is better, so lowest path cost is 278.
  这条新路径较好，故最低路径代价为 278.



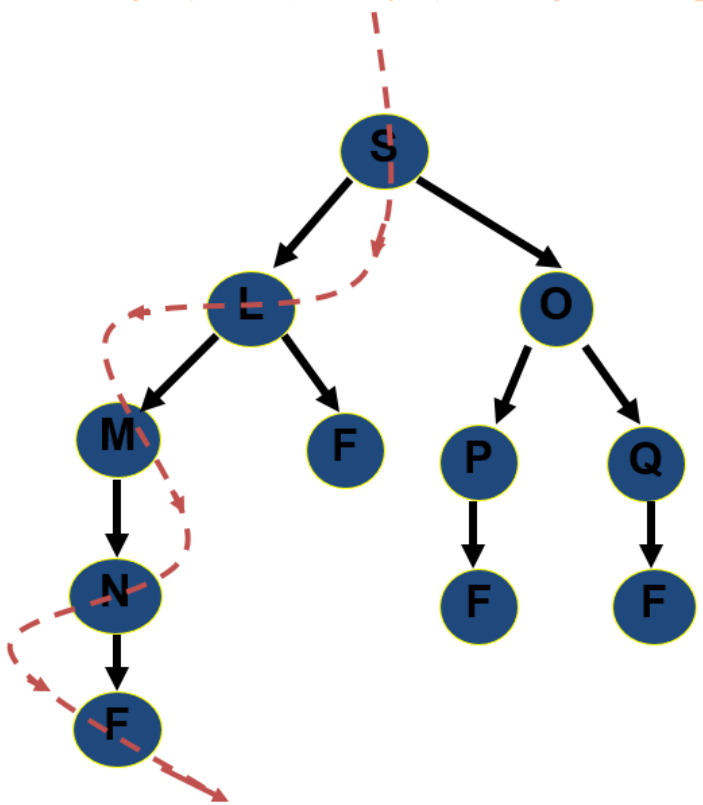
一致代价搜索的性质

- **完备性：**如果每一步的代价都大于或等于某个小的正值 ，那么该算法是完备的。
- **最优性：** 满足
- **时间复杂度：** $O(b^{1+\lfloor C^*/\epsilon \rfloor})$
- **空间复杂度：** $O(b^{1+\lfloor C^*/\epsilon \rfloor})$

- b -- the branching factor
  分支因子
- C^* -- the cost of the optimal solution
  最优解的代价
- ϵ -- every action costs at least
  至少每个动作的代价

深度优先搜索

在搜索的过程中，对open表中同一层的节点每次只选择表中一个节点进行考查和扩展，只有当这个节点是不可扩展的才选择同层的兄弟节点进行考查扩展。或者说，深度优先搜索预定的搜索方向是沿搜索树的深度方向展开的。深度相等的节点可以任意排列。



有界深度优先搜索（depth-limited search）

- 对于有界深度搜索策略，有下面几点需要说明：

  1）在有界深度搜索算法中，深度限制d_m是一个很重要的参数。

  2）深度限制d_m不能太大。

  3）有界深度搜索的主要问题是深度限制值d_m的选取。

- 问题：但是对很多问题，我们并不知道该值到底为多少，直到该问题求解完成了，才可以确定出深度限制d_m。

37

深度优先搜索算法（深度有界）

(1)初始化open表与closed表，把初始节点(序号1)放入open表中。设置搜索最大深度 d_{max}的值。

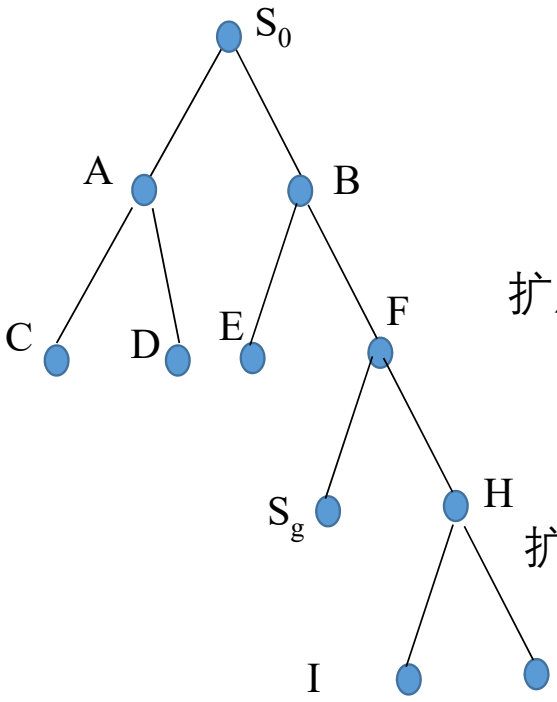
(2)若open＝()，则算法失败终止；否则，转步骤(3)。

(3)把open表的第1个节点i移出，放入closed表的表尾。

(4)若节点i的状态S_i是目标状态，则算法成功终止；否则，转步骤(5)。

(5)若节点i不可扩展(即在算符集中找不到可用算符，则转步骤(2)；否则，转步骤(6)。

(6)逐一用可用算符扩展节点i，生成i的所有子节点。若子节点j的状态S_j既不在open表中，又不在closed表中，且$d_j \leq d_{max}$，则节点j是一个允许的新节点。将所有的新子节点按节点序号从小到大依序放入open表的表首，并记载子节点各域的值。否则，不把新节点j放入open表中(放弃节点j)，转步骤(2)。



开始：S_0初始状态放在表首

Open ：s_0

closed ：s_0

扩展S_0：

Open ：B A

closed ：s_0 B

扩展B：

Open ：F E A

closed ：s_0 B F

扩展F：

open ：H s_g E A

closed s_0 B F H

。。。。。。

扩展H：

open ：J I s_g E A

closed ：s_0 B F H J

open ：s_g E A

closed ：s_0 B F H J I s_g

- 深度优先搜索算法搜索过程：

1. Open：s_0

   closed：s_0

2. Open：5 4 3 2

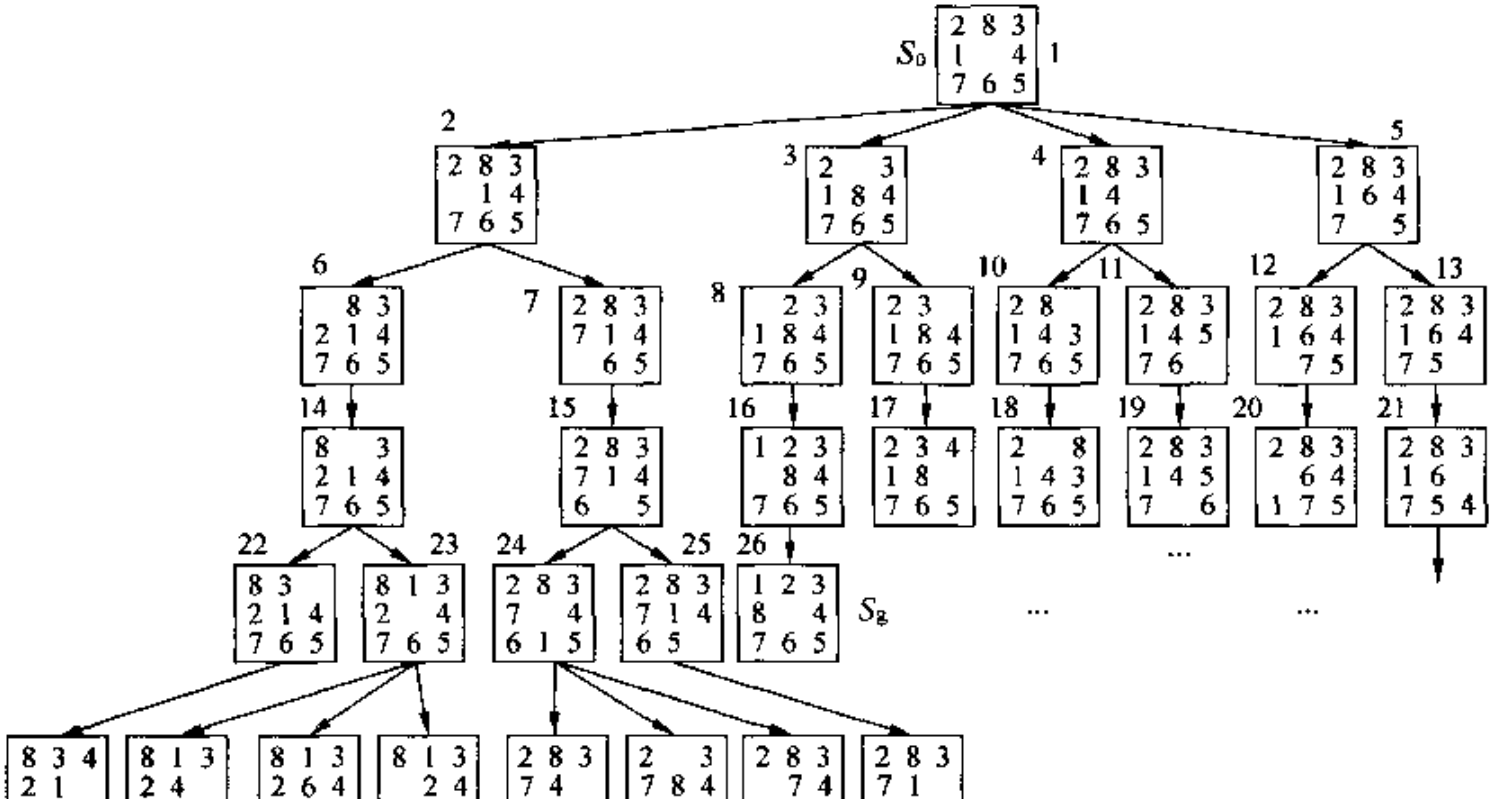
   closed：s_0 5

3. Open：13 12 4 3 2

   closed：s_0 5 13

4. Open：21 12 4 3 2

   closed：s_0 5 13 21



深度优先搜索的性质

- **完备性：** 不一定能找到解
- **最优性：** 不是最优的
- **时间复杂度：** $O(b^m)$
- **空间复杂度：** $O(bm)$

where

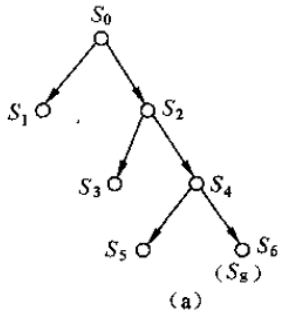
- b -- the branching factor
  分支因子
- m -- the maximum depth of any node
  任一节点的最大深度

深度优先搜索的性质

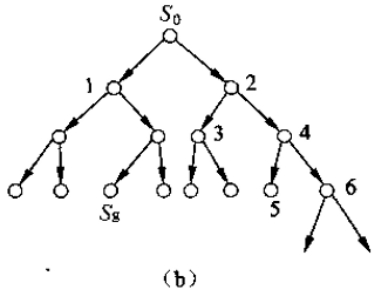
- 深度优先搜索的优点：节省大量时间和空间；
- 深度优先搜索的缺点：不一定能找到解。因为在无限搜索树的情况下，最坏的情况可能是不停机。

比较宽度优先搜索与深度优先搜索

- 宽度优先搜索生成的子节点放入open表的表尾，深度优先搜索生成的子节点放入open表的表首。
- 如果问题有解，那么宽度优先搜索总能找到路径最短的解路径，组成这条解路径的算符序列称为最优解。而深度优先搜索，如果搜索最大深度d_{max}设置合理，即目标节点的深度$d_g \leq d_{max}$，那么深度优先搜索能找到一条解路径，但是不一定是最优解的解路径。
- 一般而言，深度优先搜索成功终止时，生成的搜索树的规模(节点数量)小于宽度优先搜索生成的搜索树的规模。或者说，深度优先搜索的时空开销小于宽度优先搜索。



(a)

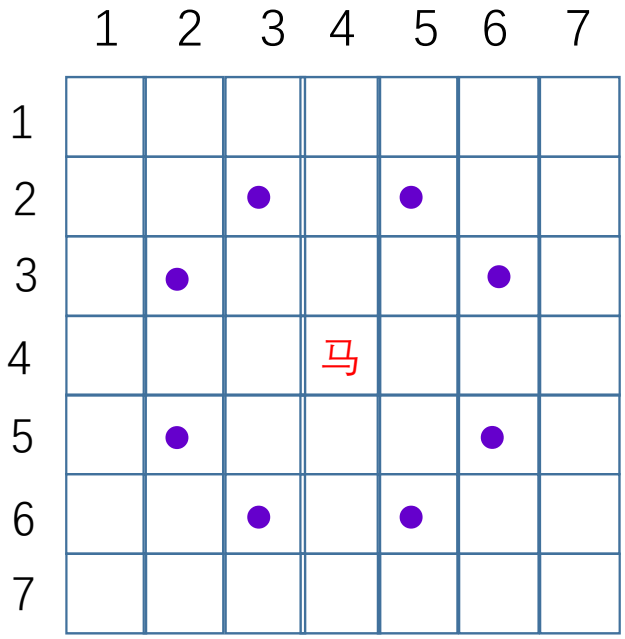


(b)

迭代加深搜索（interative deepening search）

- 迭代加深搜索是在有界深度优先搜索的基础上，对d_{max}进行迭代，即逐层加深。
- 先任意给定一个较小的数作为d_m，然后按有界深度算法搜索，若在此深度限制内找到了解，则算法结束；如在此限度内没有找到问题的解，则增大深度限制d_m，继续搜索。

一个盲目搜索问题的几种实现



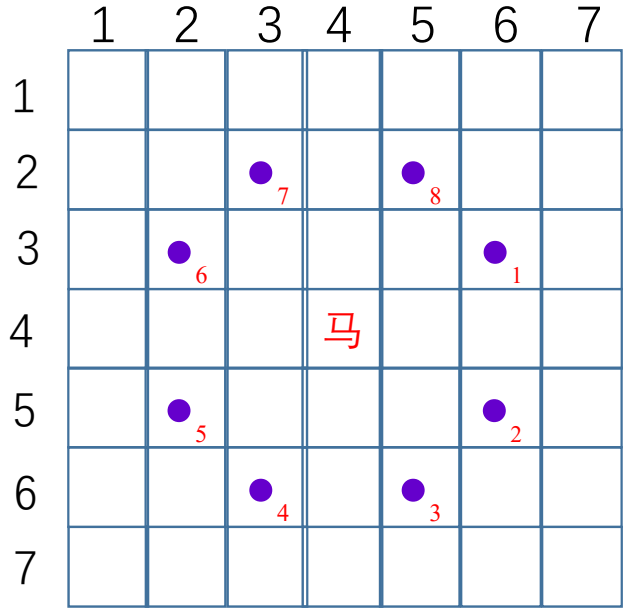
设计一个算法，对于任意给定的棋盘上的坐标位置tp，输出马从当前位置cp出发通过搜索到达的该坐标位置。

一个盲目搜索问题的几种实现

1. 定义状态空间

设状态空间的一点为10*9的矩阵。

一个盲目搜索问题的几种实现



这8个方向依次编号1,2，……8

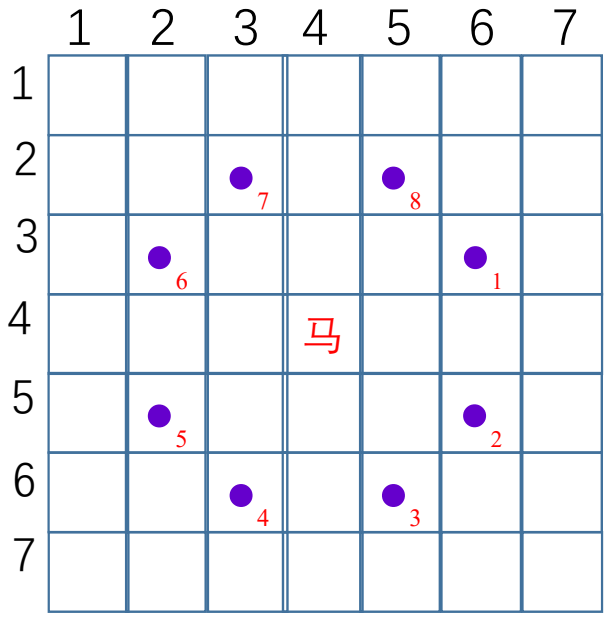
二维数组move记录第i个方向上的行下标增量move[i][0]和列下标增量move[i][1]

2. 定义操作规则

马走棋盘可以模拟为一个搜索过程：每到一处，总让它按东，东南，南，西南，西，西北、北，东北8个方向顺序试探下一个位置；如果某方向可以通过，并且在前面的搜索过程中不曾到达，则可以前进一步；在新位置上按上述方法就可以重新进行搜索，并将到达的位置标记*。

（x，y）沿第i个方向走　新位置 x1=x+move[i][0]　y1=y+move[i][1]

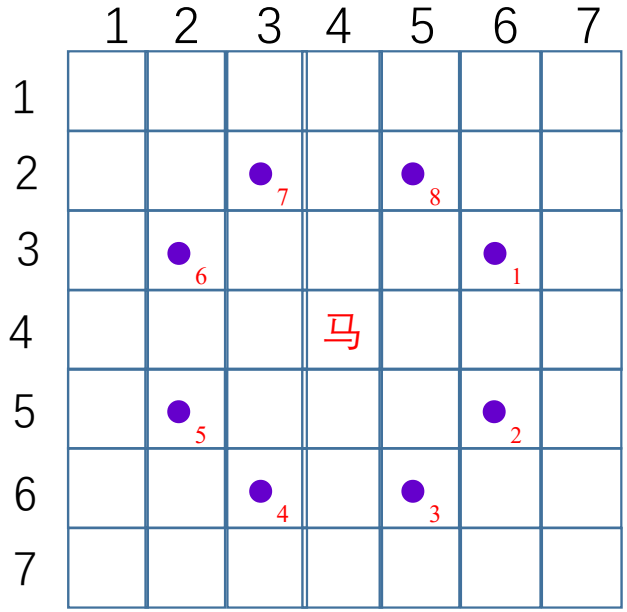
一个盲目搜索问题的几种实现



3. 定义搜索规则

（1）宽度优先搜索：沿着8个方向，如果可行，都前进一步，看是否达到位置tp，如果没有达到，则依次从新的位置为起点，沿着8个方向继续前进一步……直到搜索到目标位置tp，或找不到未搜索的位置。

一个盲目搜索问题的几种实现



3. 定义搜索规则

（2）深度优先搜索：沿着8个方向中的某一个，看是否达到位置tp，如果没有达到，则以这个位置为起点，沿着8个方向中的某一个继续前进一步……某个分支搜索不通时，再沿着当前位置8个方向的下一个方向继续搜索。直到搜到目标位置tp，或找不到未搜索的位置为止。

3.2 启发式搜索方法

如果能够找到一种方法用于排列待扩展节点的顺序，即选择最有希望的节点加以扩展，那么，搜索效率将会大大提高。启发式搜索就是基于这种想法，它是深度优先的改进。搜索时不是任取一个分枝，而是根据一些启发式信息，选择最佳一个分枝或几个分枝往下搜索。

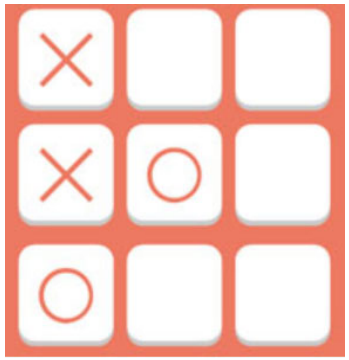
启发式信息的表示

1．启发式搜索的依据

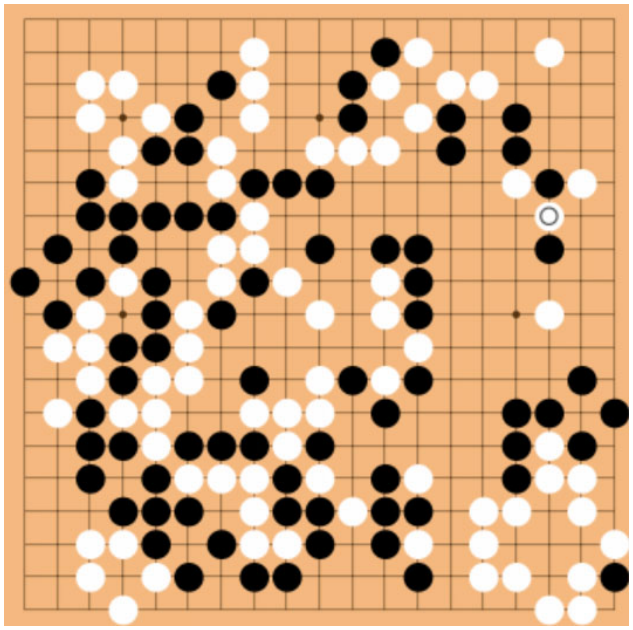
- 人们善于利用一些线索来帮助自己选择搜索方向，这些线索统称为启发式(Heuristics)信息。
- 现实问题往往只需一个解，而不要求最优解或全部解。
- 启发式信息可以避免某些领域里的组合爆炸问题。

51

一字棋：9!



国际象棋：10^{120}



围棋：10^{761}

启发式信息的分类

（1）表示为启发式函数（heuristic function）

- 确定一个启发式函数h(n), n 为被搜索的节点,它把问题状态的描述映射成问题解决的程度，通常这种程度用数值来表示，就是启发式函数的值。这个值的大小用来决定最佳搜索路径。

启发式信息的分类

（2）表示成规则

如AM的一条启发式规则为：

如果存在一个有趣的二元函数f(*x*, *y*)，那么看看两变元相同时会发生什么？

若f表示乘法：导致发现平方；

若f表示集合并运算：导致发现恒等函数；

若f表示思考：导致发现反省；

若f表示谋杀：导致发现自杀。

（3）表示成元规则

体现了状态空间的搜索策略。

启发式信息的表示

2. 启发式函数的表示方法

- 启发式函数是一种映射函数，它把对问题状态的描述映射成一种希望的程度。
- 启发式函数设计得好，对有效引导搜索过程有着重要的作用。非常简单的启发式函数搜索路径能够作出非常令人满意的估计。

如何构造启发式函数

（1）启发式函数能够根据问题的当前状态，确定用于继续求解问题的信息。

求解水壶问题：如何通过两个容量为4升、3升的水壶得到2升水。

考虑利用启发式信息。水壶容量信息：4, 3, 0，如何求得2?

4-2=2　　3-1=2　　如何求得1？

4-3=1：可以从加满水的4升水壶里向空的3升水壶里加满，

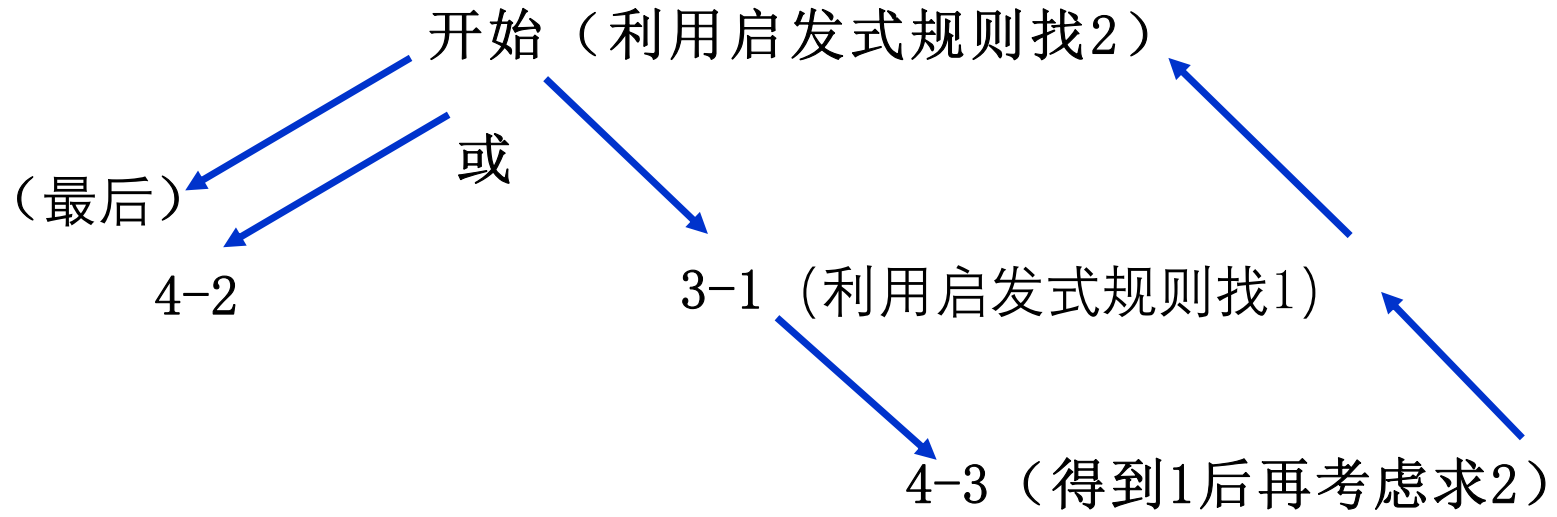
这样4升水壶里只剩下1升。得到数字1。

3-1=2：可以把3升水壶倒空，再把4升水壶的1升水倒入3升水壶中，3升水壶还缺2升水，现在有数字2了，可以考虑4-2=2。

4-2=2：将4升水壶的水加满，然后用4升水壶中的水将3升的水壶加满。这样4升水壶中就只剩下2升了。

(0, 0) ->(4, 0)->(1, 3)->(1, 0)->(0, 1)->(4, 1)->(2, 3)=目标

利用启发式信息求解水壶问题的搜索过程

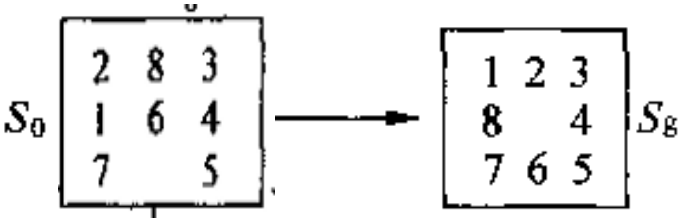


如何构造启发式函数

（2）启发式函数能够估计已找到的状态与达到目标的有利程度。

8数码问题

各状态间的转换规则为：



Pr1: 空格上移 ↑

If $\square_{i,j}$ and $i \neq 1$ then $a_{i-1,j} \longleftrightarrow \square_{i,j}$

Pr2: 空格下移 ↓

If $\square_{i,j}$ and $i \neq 3$ then $a_{i+1,j} \longleftrightarrow \square_{i,j}$

Pr3: 空格左移 ←

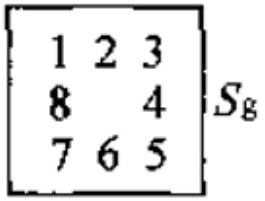
If $\square_{i,j}$ and $j \neq 1$ then $a_{i,j-1} \longleftrightarrow \square_{i,j}$

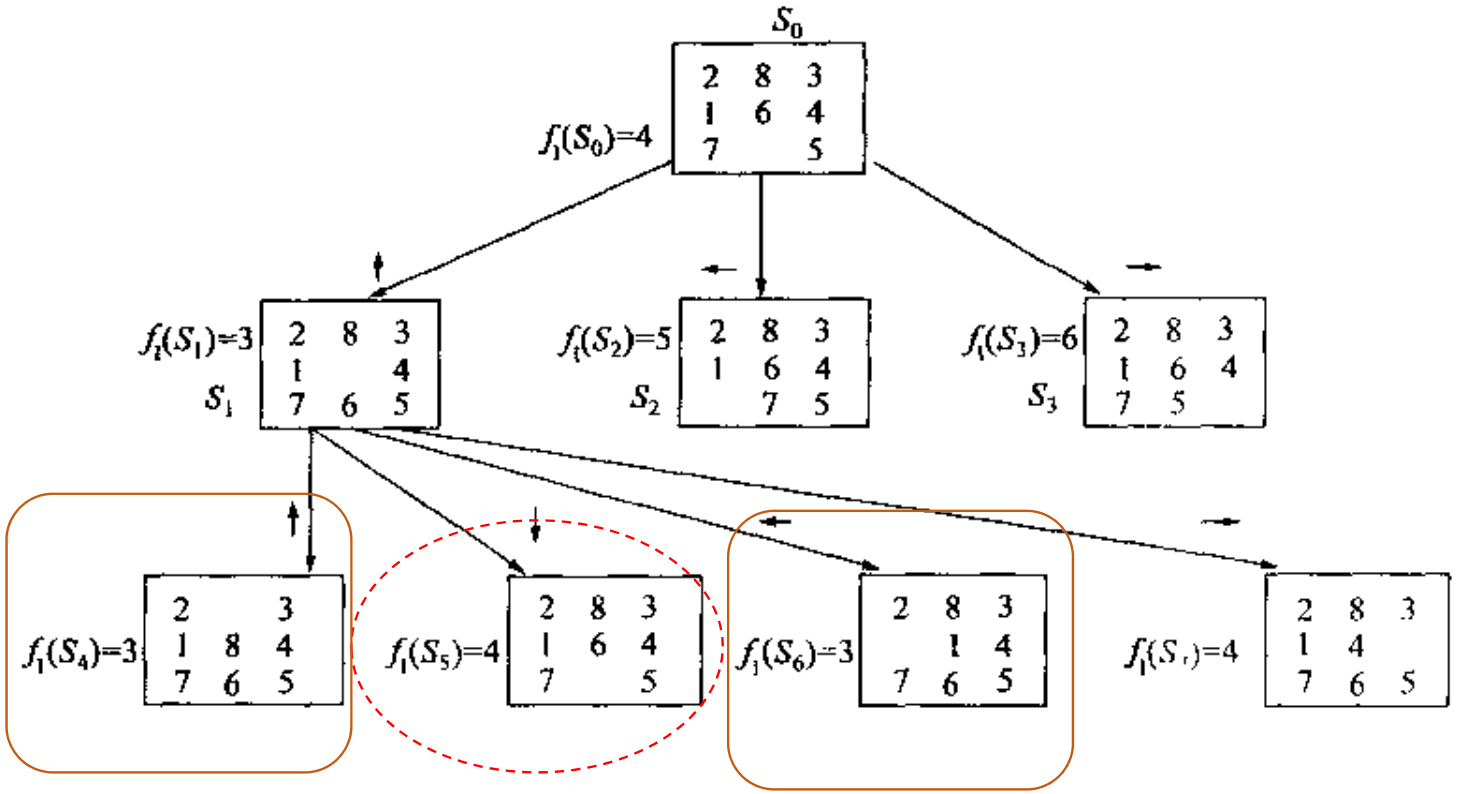
Pr4: 空格右移 →

If $\square_{i,j}$ and $j \neq 3$ then $a_{i,j+1} \longleftrightarrow \square_{i,j}$

$$\begin{matrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ a_{31} & a_{32} & a_{33} \end{matrix}$$

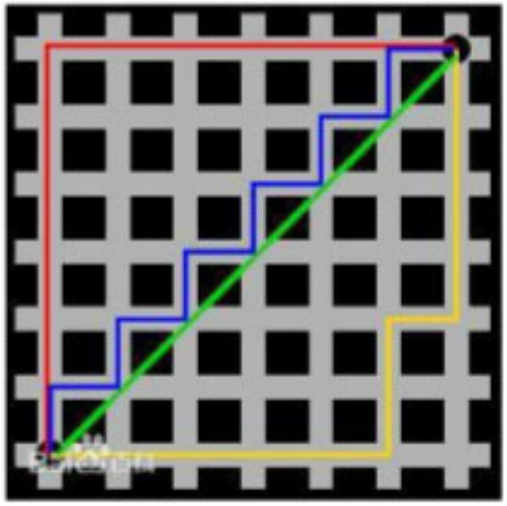
启发式函数f_1= 数字错放位置的个数，f_1=0，则达到目标。





当f_1值相同时如何决定走步?

f_2 = 所有数字当前位置以最短路径走到正确位置的步数之和（**曼哈顿距离**）



Manhattan distance

5 6 7 8

0 + 1 +0 + 2 =5

0 + 0 +0 + 2 =4

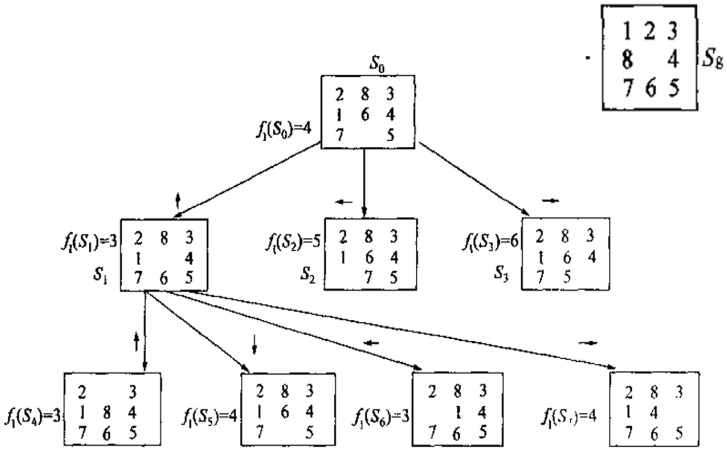
0 + 1 +1 + 2 =6

1 + 1 +0 + 2 =6

0 + 0 +0 + 1 =3

$F_2(S_5)$ = 1 + 1 +0 +0 + 0 + 1 +0 + 2 =5

$F_2(S_6)$ = 1 + 2 +0 +0 + 0 + 0 +0 + 2 =5



- 原来有$f_1(S_4) = f_1(S_6) = 3$，现在$f_2(S_4) < f_2(S_6)$，所以下一步搜索从S_4出发，产生后继节点。

如何构造启发式函数

(3) 启发式函数应能够估计出可能加速达到目标的程度

当扩展一个节点时，可以帮助确定哪些节点应删除。

启发式函数对每一节点的真正优点估计得愈精确，解题过程就愈少走弯路。

62

八皇后问题(8-Queens problem)

在8*8棋盘要求放下8个皇后，要求没有一个皇后能够攻击其它皇后，即要使得在任何一行、一列或对角线上都不存在两个或两个以上的皇后。

八皇后问题

（1）定义状态空间。

设状态空间的一点为：8*8矩阵。

（2）定义操作规则。

按如下规则放置皇后：

第一个皇后放第一行。

第二个皇后放在第二行且不与第一个皇后在同一列或对角线的空格上。

……

第i个皇后放在第i行且不与前面i －1个皇后在同一列或对角线的空格上。

……

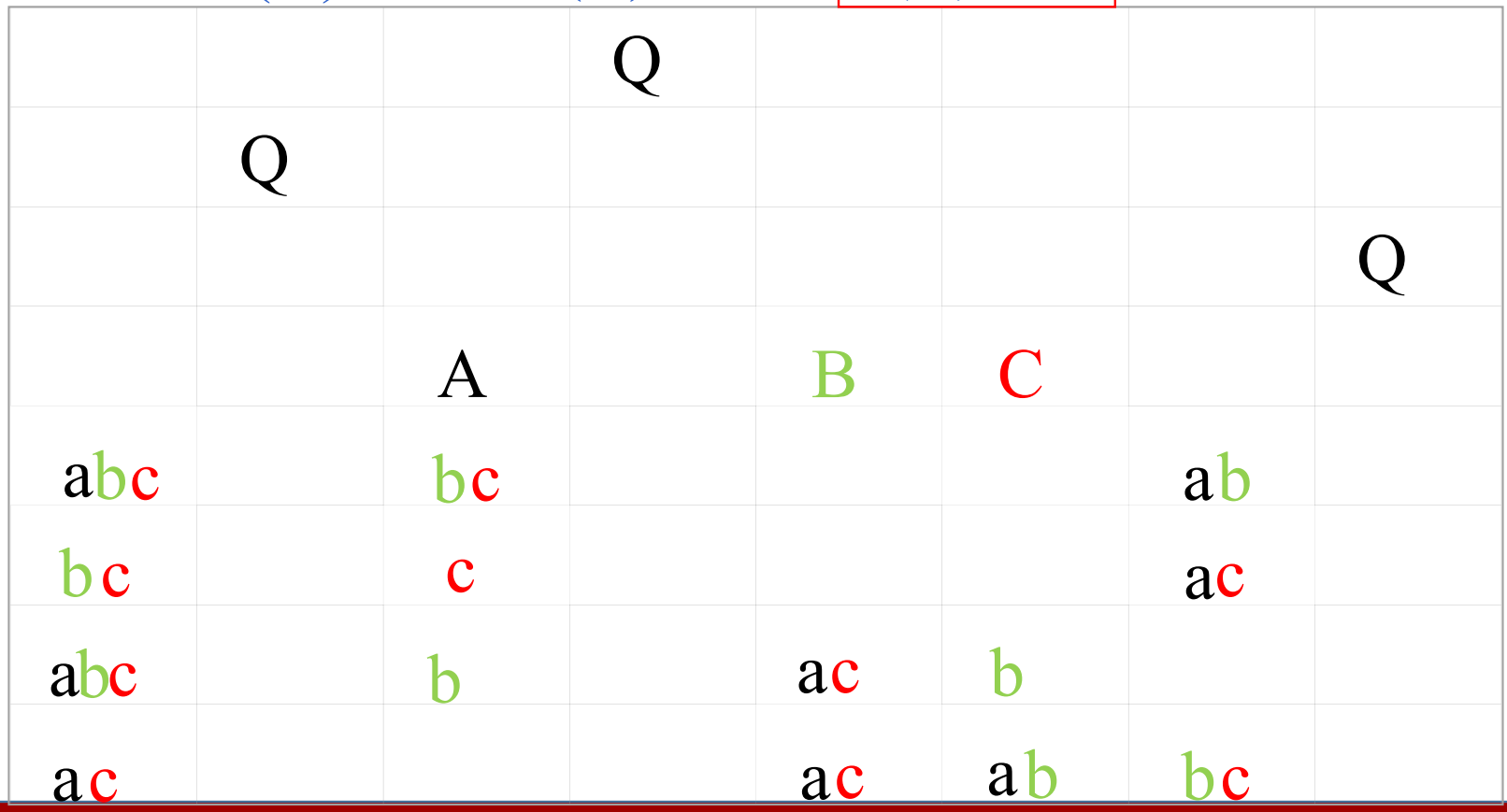
八皇后问题

可使用如下启发式函数：

- 设x为当前要放置Queen的空格
- h（x）= 剩下未放行中能够用来放Queen的空格数
- 不难看出，h（x）愈大愈好，应选择h（x）最大的空格来放置皇后。

例如，在放置了三个Q后，第4个Q可放在第4行的A，B，C三个位置。

h(A)=8　　h(B)=9　　h(C)=10



八皇后问题

（3）定义搜索策略。

第i个皇后放到第i行中的那个与前面i-l个皇后不在同一列或对角线上且h（x）值最大的空格中。

启发式信息是某些领域里的知识信息，它能使计算机系统在这些知识信息提示以后可能采取的某些可能的动作或避免某些不可能的动作。

启发式方法适用范围

（1）适用于特定的领域，如果这个领域较窄，这种方法就变成了类似于计算方法中的算法。

（2）适用于较广泛领域的通用启发式搜索算法。这就是后面要讨论的**弱法**。

搜索方向的选择

搜索过程：在问题空间中找出从开始状态到目标状态的一条最好的或较好的路径。

这种搜索可按两个方向进行：

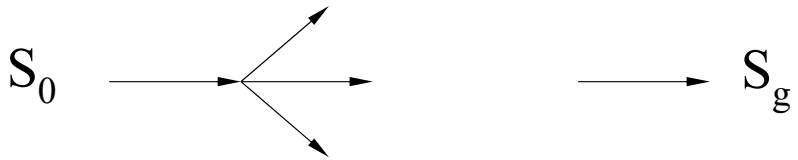
正向搜索

逆向搜索

双向搜索

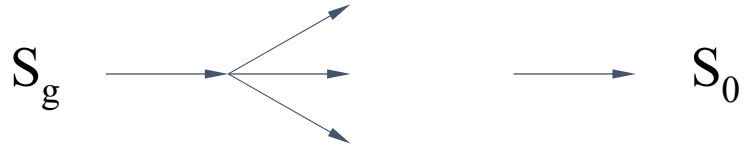
正向搜索

从初始状态朝着目标状态方向搜索

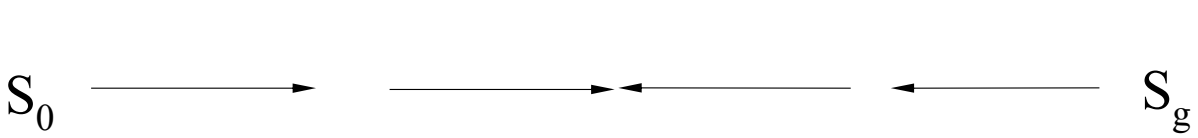


逆向搜索

从目标状态朝着初始状态方向搜索



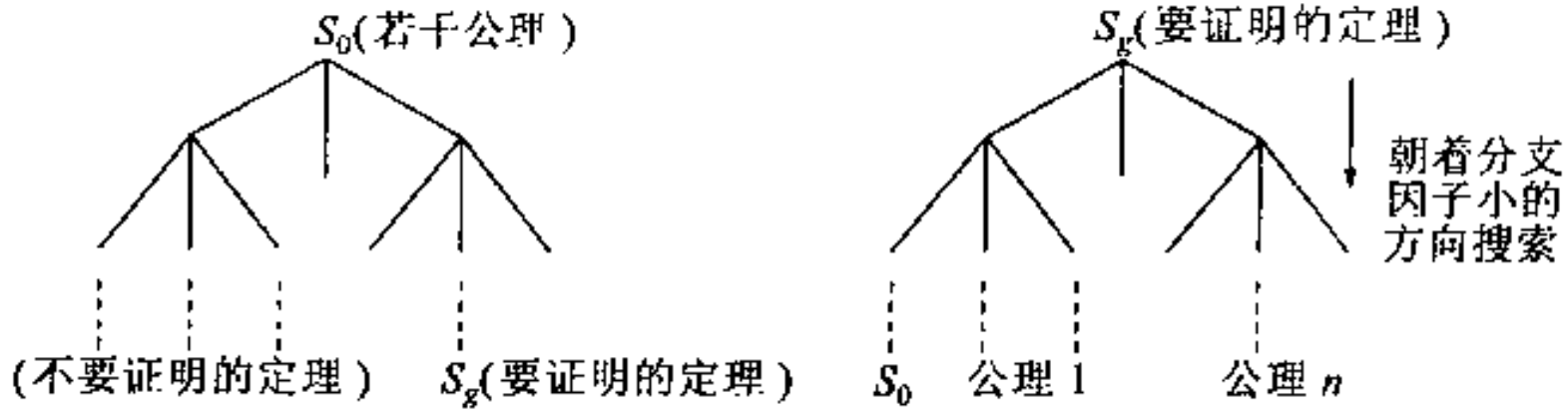
将以上两种搜索方法结合起来，就产生了**双向搜索**



搜索方向的选择

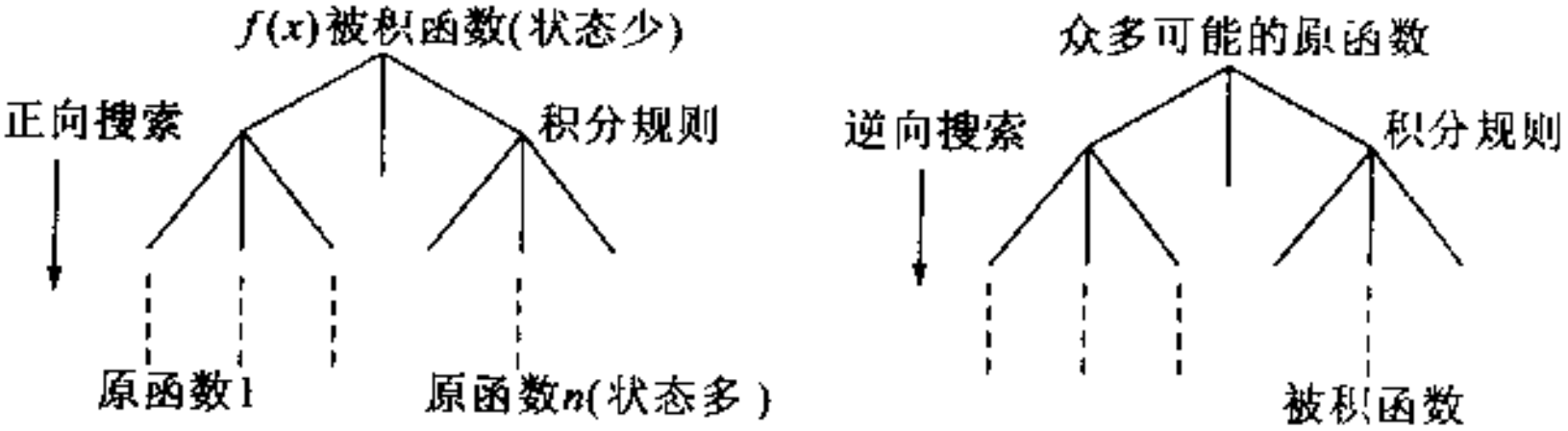
（1） 朝分支因子低的方向更有效

- 分支因子指从一点出发可以直接到达的平均结点数。朝着分支因子低的方向搜索意味着朝着“收敛”的方向搜索.



搜索方向的选择

(2) 由状态少的一方出发，朝着大量的可识别的状态的方向搜索.



积分问题的两个方向搜索

搜索方向的选择

(3) 依据用户可接受的方向

特别是需要向用户解释推理过程时，顺应用户的心理，选择搜索方向会使系统显得更自然一些。在建造专家系统时，向用户解释为什么系统会得出某个结论，这一步骤是必不可少的，所以尤其要考虑这个问题。

几种最基本的搜索策略

- 下面主要介绍几个基本搜索策略，这些方法构成了许多AI系统的构架，其效率取决于问题所在领域知识的利用与开发。由于这些方法的通用性，并且难于克服搜索过程的组合爆炸问题，所以又称为弱法(Weak method）。

几种最基本的搜索策略

最佳优先法

生成测试法

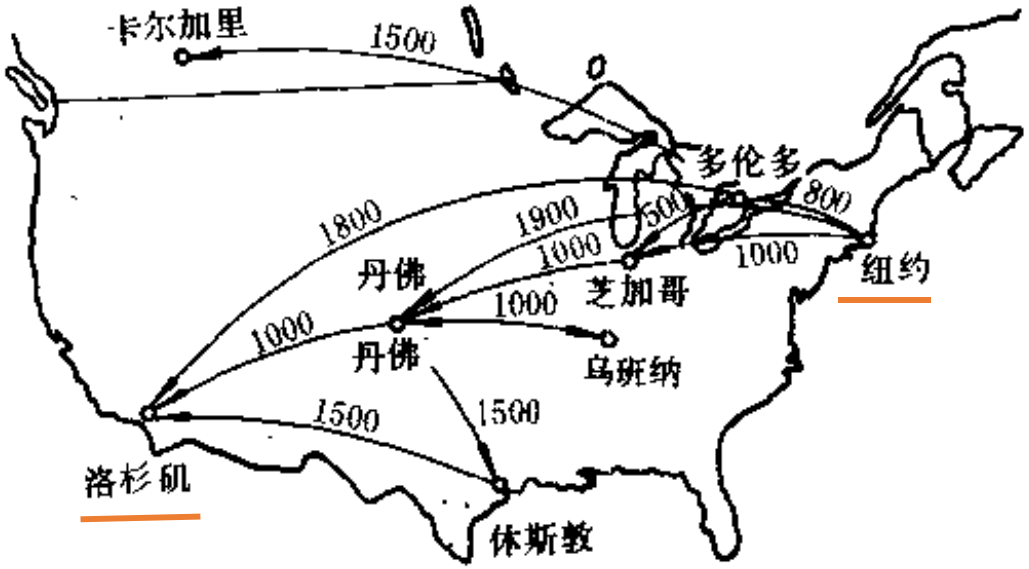
爬山法

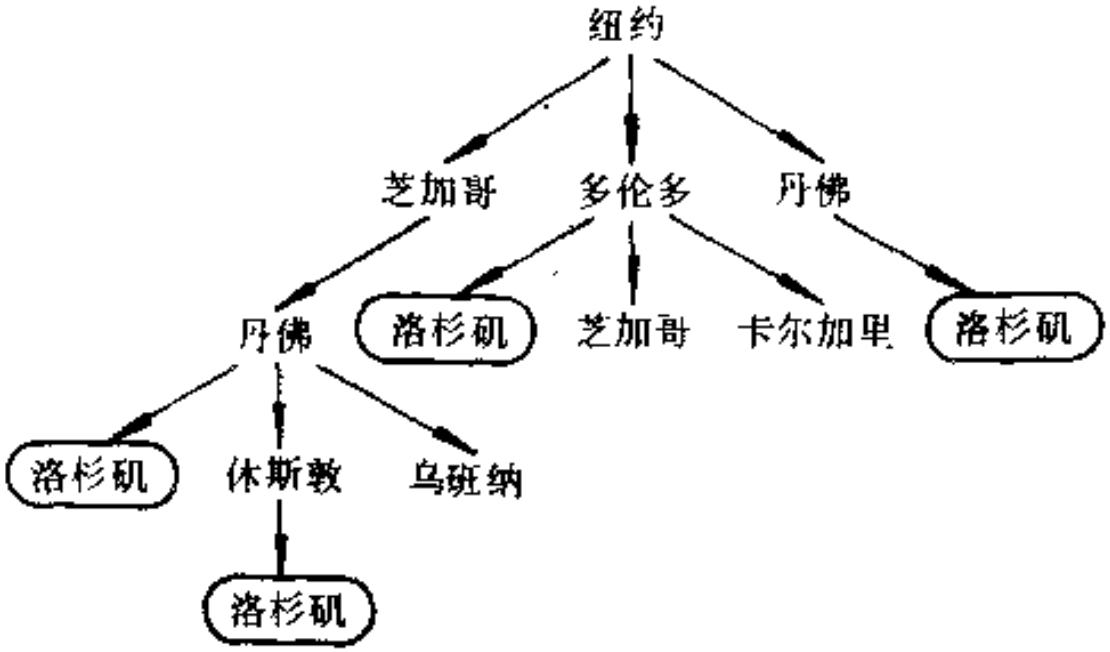
1．生成测试法(Generate-and-test）

基本步骤为：

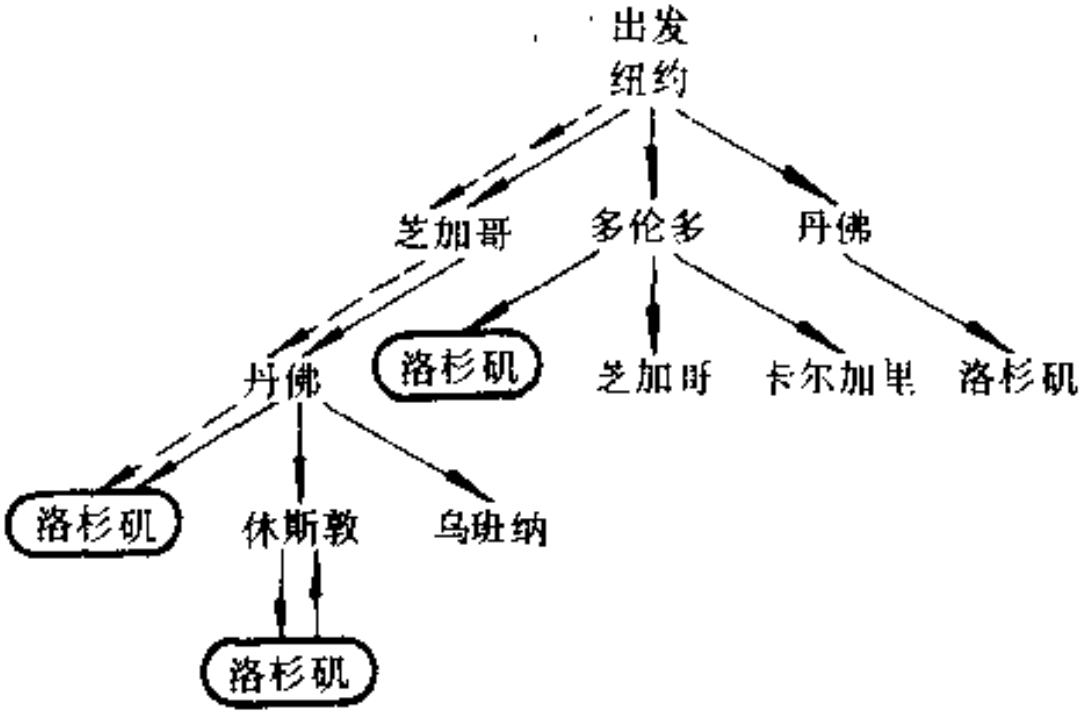
- 生成一个可能的解，此解是状态空间一个点，或一条始于S_0的路径。
- 用生成的“解”与目标比较。
- 达到目标则停止，否则转第一步。

通过中途转机，找出XY航空公司由纽约到洛杉矶的路线，给旅客订机票。





1．生成测试法



- 此方法属于深度优先搜索（depth-first-search ），因为要产生一个完全的解后再判断，若不是目标又要生成下一个“解”。这种方法几乎接近耗尽式搜索，因而效率低。

最佳优先搜索(Best-first search)

- 搜索策略

  一个节点被选择进行扩展是基于一个评价函数f（n）。

  大多数的最佳优先算法还包含一个启发式函数h（n）。

- 实现方法

  与一致代价搜索相同，但是使用f（n）代替g（n）来整理优先队列。

- 启发式函数

  h（n）：从节点n到目标状态的最低路径估计代价

最佳优先搜索的算法步骤

（1）生成第一个可能的解。若是目标，则停止；否则转下一步。

（2）从该可能的解出发，生成新的可能解集。

2.1 用测试函数测试新的可能解集中的元素，若是解，则停止；否则转2.2。

2.2 若不是解，则将新生成的“解”集加入到原可能“解”集中。

（3）从解集中挑选最好的元素作为起点，再转（2）

最佳优先搜索(Best-first search)

- 特例

  贪婪搜索（Greedy Search）

  A*搜索（A*Search）

- 贪婪搜索

  搜索策略是试图扩展最接近目标的节点。

  评价函数f(n)=h(n)

  仅仅使用启发式函数对节点进行评价。

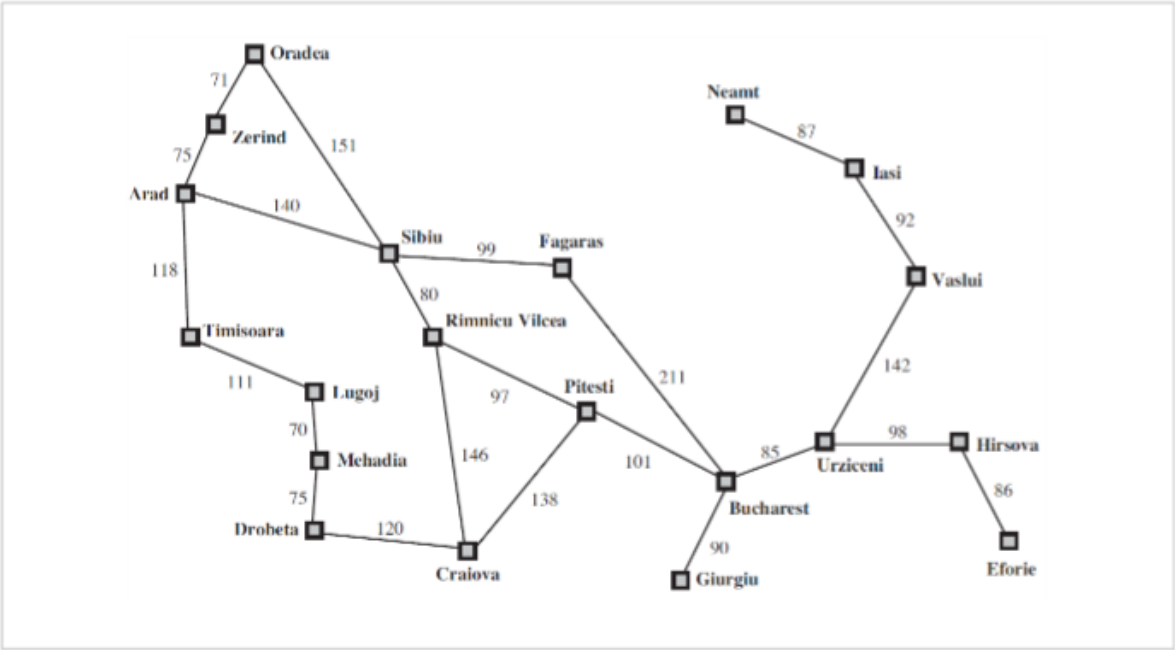
  h（n）：从n到目标的估计代价。

  WHY?

  每一步都试图得到最接近目标的节点。

最佳优先搜索举例

- h_{SLD} : straight-line distance 直线距离



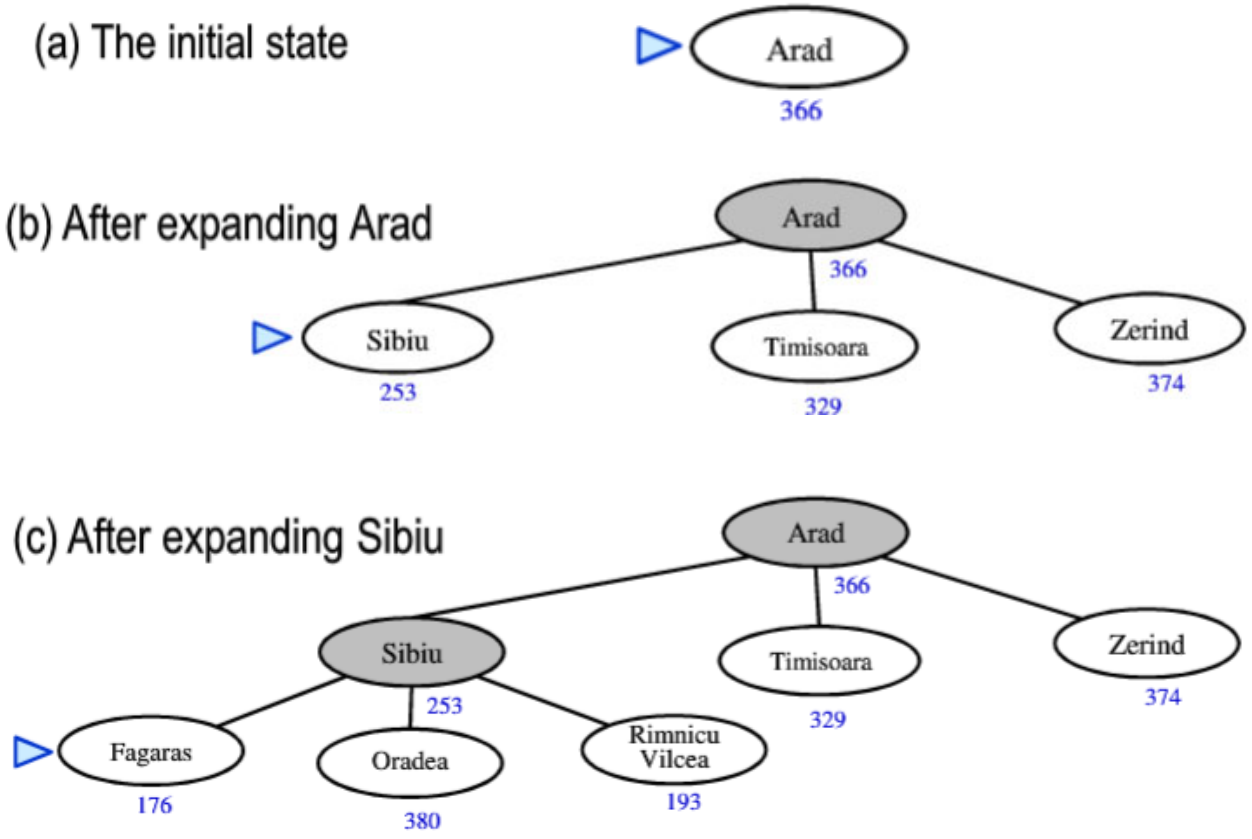
注意：h_{SLD}的值无法从问题描述本身来计算。此外，它要积累一定的经验才能知道，h_{SLD}与实际道路的距离相关，因此是一个有用的启发。

h_{SLD} Values

Arad	366	Mehadia	241
Bucharest	0	Neamt	234
Craiova	160	Oradea	380
Drobeta	242	Pitesti	100
Eforie	161	Rimnicu Vilcea	193
Fagaras	176	Sibiu	253
Giurgiu	77	Timisoara	329
Hirsova	151	Urziceni	80
Iasi	226	Vaslui	199
Lugoj	244	Zerind	374

Notice: the values of h_{SLD} cannot be computed from the problem description itself. Moreover, it takes a certain amount of experience to know that h_{SLD} is correlated with actual road distances and therefore is a useful heuristic.

罗马尼亚问题

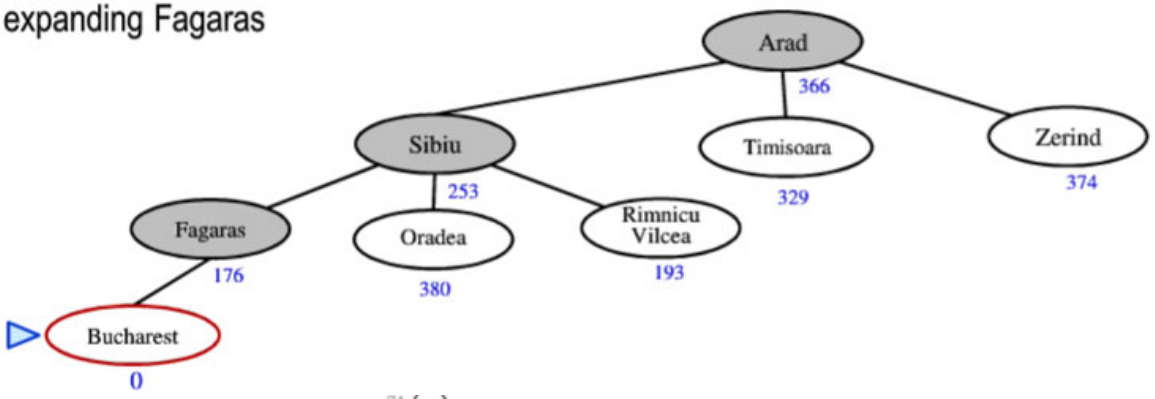


h_{SLD} Values

Arad	366
Bucharest	0
Craiova	160
Drobeta	242
Eforie	161
Fagaras	176
Giurgiu	77
Hirsova	151
Iasi	226
Lugoj	244
Mehadia	241
Neamt	234
Oradea	380
Pitesti	100
Rimnicu Vilcea	193
Sibiu	253
Timisoara	329
Urziceni	80
Vaslui	199
Zerind	374

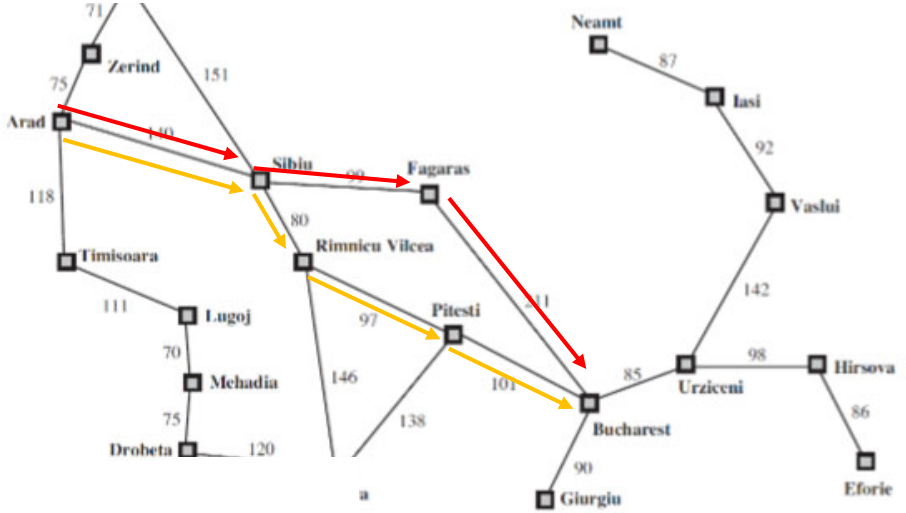
罗马尼亚问题

(d) After expanding Fagaras



注意：对这个具体问题，它采用h_{SLD}找到解，因此搜索代价是最小的。然而它不是最优的：如果计算路径代价的话，这条经由Sibiu和Fagaras到Bucharest的路径比经过Rimnicu Vilcea 和Pitesti远32公里。

$(140+99+211) - (140+80+97+101) = 32$



a

h_{SLD} Values

Arad	366
Bucharest	0
Craiova	160
Drobeta	242
Eforie	161
Fagaras	176
Giurgiu	77
Hirsova	151
Iasi	226
Lugoj	244
Mehadia	241
Neamt	234
Oradea	380
Pitesti	100
Rimnicu Vilcea	193
Sibiu	253
Timisoara	329
Urziceni	80
Vaslui	199
Zerind	374

启发函数代价最小化这一目标会对错误的起点比较敏感。考虑从Iasi到Fagaras的问题，由启发式建议须先扩展Neamt，因为其离Fagaras最近，但是这是一条存在死循环路径。

最佳优先搜索的不足之处

- 贪婪最佳优先搜索不是最优的。经过Sibiu到Fagaras到Bucharest的路径比经过Rimnicu Vilcea到Pitesti到Bucharest的路径要长32公里。
- 启发函数代价最小化这一目标会对错误的起点比较敏感。考虑从Iasi到Fagaras的问题，由启发式建议须先扩展Neamt，因为其离Fagaras最近，但是这是一条存在死循环路径。
- 贪婪最佳优先搜索也是不完备的。所谓不完备即它可能沿着一条无限的路径走下去而不回来做其他的选择尝试，因此无法找到最佳路径这一答案。
- 在最坏的情况下，贪婪最佳优先搜索的时间复杂度和空间复杂度都是$O(b^m)$，其中b是节点的分支因子数目、m是搜索空间的最大深度。

因此，需要设计一个良好的启发函数