第2章 问题求解与搜索1

南京信息工程大学
计算机与软件学院

应龙

2025年秋季

主要内容

1. 问题求解与形式化描述 (Problem-Solving)

2. 搜索算法框架 (Search Algorithms)

3. 搜索策略 (Search Strategies)

4. A* 搜索算法 (A* search)

Bibliography:

- ✓ 吴飞 编著, “人工智能导论：模型与算法” (2020), 高等教育出版社. Ch 3.1, 3.2
- ✓ Stuart J. Russel, Peter Norvig, “Artificial Intelligence: A Modern Approach” (4th Ed. 2020); 中译版 “人工智能 一种现代的方法” (3rd Ed., 2009), 清华大学出版社. Ch 3
- ✓ 王万良 编著, “人工智能导论” (第5版), 高等教育出版社, 2020. Ch 5
- ✓ 王万森 编著, “人工智能原理及其应用” (第4版, 2018), 电子工业出版社. Ch 4.1, 4.2

主要内容

1. 问题求解与形式化描述 (Problem-Solving)

2. 搜索算法框架 (Search Algorithms)

3. 搜索策略 (Search Strategies)

4. A* 搜索算法 (A* search)

Bibliography:

- ✓ Stuart J. Russel, Peter Norvig, “Artificial Intelligence: A Modern Approach” (4th Ed. 2020); 中译版 “人工智能 一种现代的方法” (3rd Ed., 2009), 清华大学出版社. Ch 3.1, 3.2
- ✓ 吴飞 编著, “人工智能导论：模型与算法” (2020), 高等教育出版社. Ch 3.1, 3.2
- ✓ 王万森 编著, “人工智能原理及其应用” (第4版, 2018), 电子工业出版社. Ch 4.1.1, 4.1.2
- ✓ 王万良 编著, “人工智能导论” (第5版), 高等教育出版社, 2020. Ch 5.2

Problem-Solving

AI早期的目的是想通过计算技术来求解这样一些问题：它们不存在现成的求解算法或求解方法非常复杂，而人使用其自身的智能都能较好地求解。

把问题求解定义为状态空间的搜索：在分析和研究了人运用智能求解的方法之后，人们发现许多问题的求解方法都是通过试探在某个可能的解空间内寻找一个解来求解问题，这种基于解答空间的问题表示和求解方法就是状态空间法。许多涉及智力的问题求解可看成状态空间的搜索。

When the correct action to take is not immediately obvious, an agent may need to plan ahead: to consider a sequence of actions that form a path to a goal state.

Problem-Solving Agent: 一种基于目标的Agent。使用原子(atomic) 表示：世界的状态被视为一个整体，对于问题求解算法没有可见的内部结构。使用更为先进的要素化 (factored) 或结构化 (structured) 表示的基于目标的 Agent 通常被称为 planning agent。

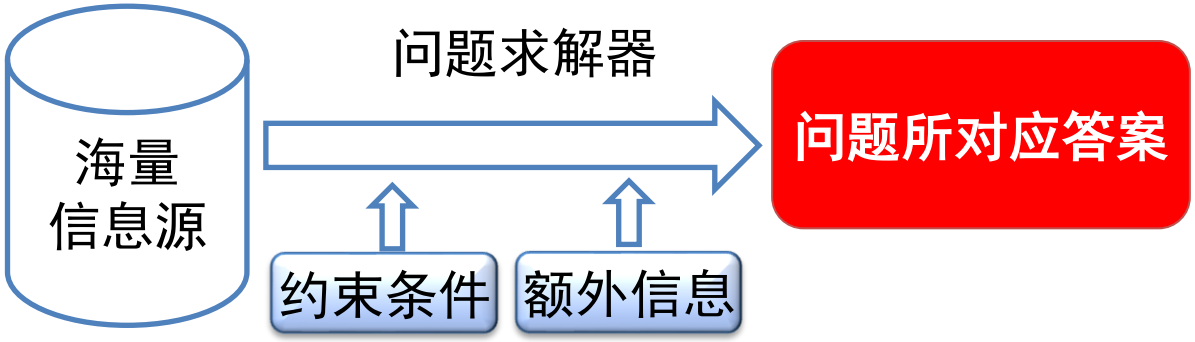
Problem-Solving

你见，或者不见我

我就在那里

不悲 不喜

---扎西拉姆多多



水壶问题

给定两个水壶，一个可装4加仑水，一个能装3加仑水。水壶上没有任何度量标记。有一水泵可用来往壶中装水。

问：怎样在能装4加仑的水壶里恰好只装2加仑水？

(1) 定义状态空间

可将问题进行抽象，用要素化(x, y)表示状态空间的任一状态。

x—表示4gallon水壶中所装的水量，x=0，1，2，3或4；

y—表示3gallon水壶中所装的水量，y=0，1，2或3；

- 初始状态为 (0,0)
- 目标状态为 (2,?)

？表示水量不限，因为问题中未规定在3加仑水壶里装多少水。

水壶问题

(2) 确定一组操作

1 (X, Y|X<4) → (4, Y)　　4加仑水壶不满时，将其装满；

2 (X, Y|Y<3) → (X, 3)　　3加仑水壶不满时，将其装满；

5 (X, Y|X>0) → (0, Y)　　把4加仑水壶中的水全部倒出；

6 (X, Y|Y>0) → (X, 0)　　把3加仑水壶中的水全部倒出；

7 (X, Y|X+Y≥4∧Y>0) → (4, Y-(4-X))

把3加仑水壶中的水往4加仑水壶里倒，直至4加仑水壶装满为止；

8 (X, Y|X+Y≥3∧X>0) → (X-(3-Y), 3)

把4加仑水壶中的水往3加仑水壶里倒，直至3加仑水壶装满为止；

9 (X, Y|X+Y≤4∧Y>0) → (X+Y, 0)

把3加仑水壶中的水全部倒进4加仑水壶里；

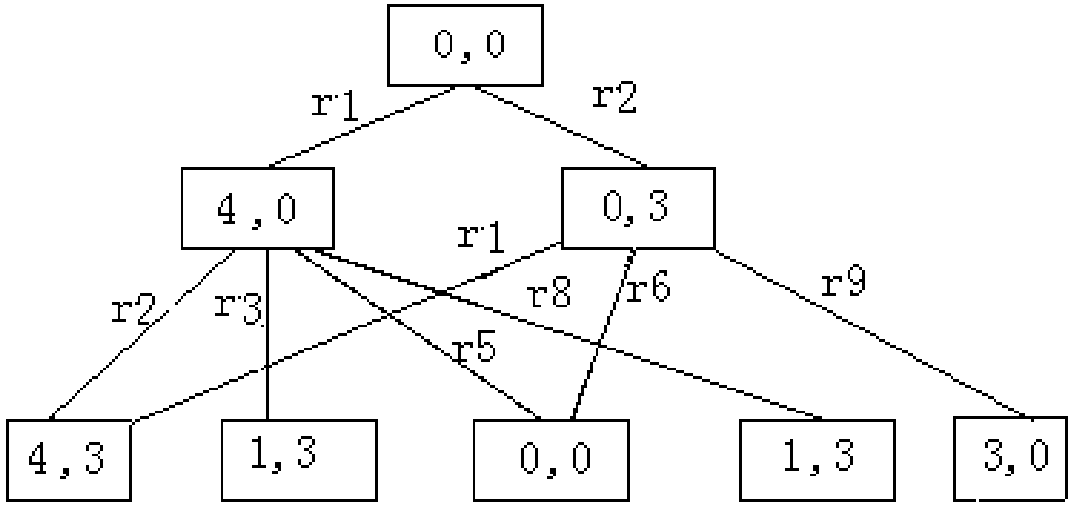
10 (X, Y|X+Y≤3∧X>0) → (0, X+Y)

把4加仑水壶中的水全部倒进3加仑水壶里；

水壶问题

(3) 选择一组搜索策略

该策略为一个简单的循环控制结构：选择其左部匹配当前状态的某条规则，并按照该规则右部的行为对此状态作适当改变，然后检查改变后的状态是否为某一目标状态，若不是，则继续该循环。



水壶问题

4加仑水壶中的水量	3加仑水壶中的水量	所应用的规则
0	0	
0	3	2
3	0	9
3	3	2
4	2	7
0	2	5
2	0	9

搜索算法的形式化描述

- 状态 (state) 是对问题在求解时某一时刻进展情况的数学描述，也可以说是一个可能解的表示。

  为描述某些不同事物（情况）间的差别而引入的一组最少变量 $q_0, q_1, q_2, \cdots, q_{n-1}$ 的有序集合，表示为：

  $Q = (q_0, q_1, q_2, \cdots, q_{n-1})$

  其中，每个元素q_i称为状态分量。给定每个分量的一组值，就得到一个具体的状态。

- 问题的状态空间 (state space) 是一个表示该问题全部可能状态及其关系的集合。

  通常以图 (graph) 的形式出现，图上的节点对应问题的状态，节点之间的边对应的是状态转移的可行性，边上的权重(weight)可以对应转移所需的代价。

- 状态空间的表示一般分为隐式图和显式图。显示图是已经把所有的状态信息都储存起来，而隐式图完全靠扩展规则来生成，也就是边搜索边生成。

搜索算法的形式化描述

- 使问题从一种状态变化为另一种状态的手段称为操作符或状态转移算子(operator)。

  操作符可能是走步（下棋）、过程、规则、数学算子、运算符号或逻辑运算符等。

- 问题的状态空间包含三种类型的集合，即该问题所有可能的
  - 初始状态集合 S,
  - 操作符集合 F,
  - 目标状态集合 G。

因此，可把状态空间记为三元组 (S, F, G)。

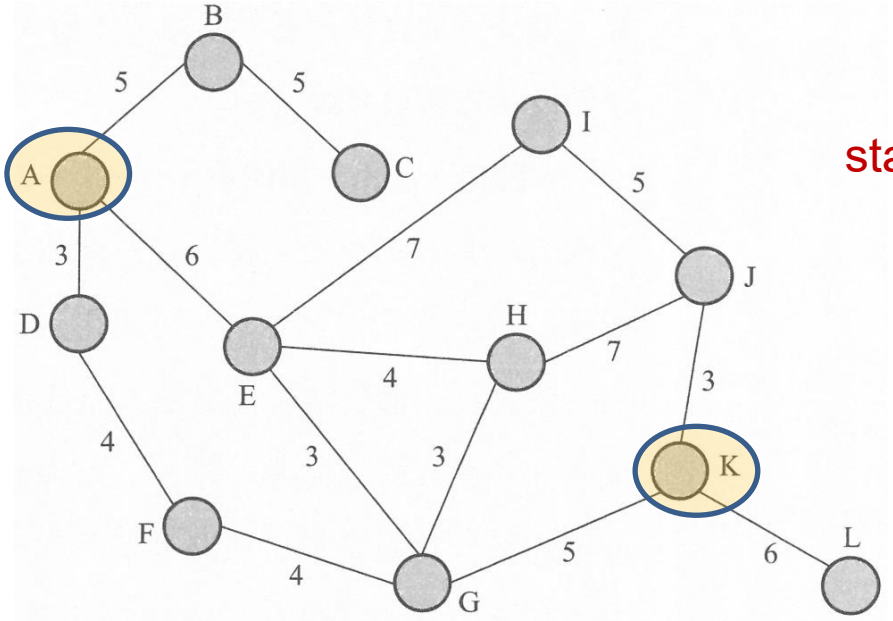
搜索算法的形式化描述

状态
state

对智能体和环境当前情形的描述。例如，在最短路径问题中，城市可作为状态。将原问题对应的状态称为初始状态。

动作
action

从当前时刻所处状态转移到下一时刻所处状态所进行操作。一般而言这些操作都是离散的。



问题：寻找从城市A到城市K之间行驶时间最短路线？

状态转移
state transition

智能体选择了一个动作之后，其所处环境状态的相应变化。

路径/代价
path/cost

一个状态序列。该状态序列被一系列操作所连接。如从A到K所形成的路径。

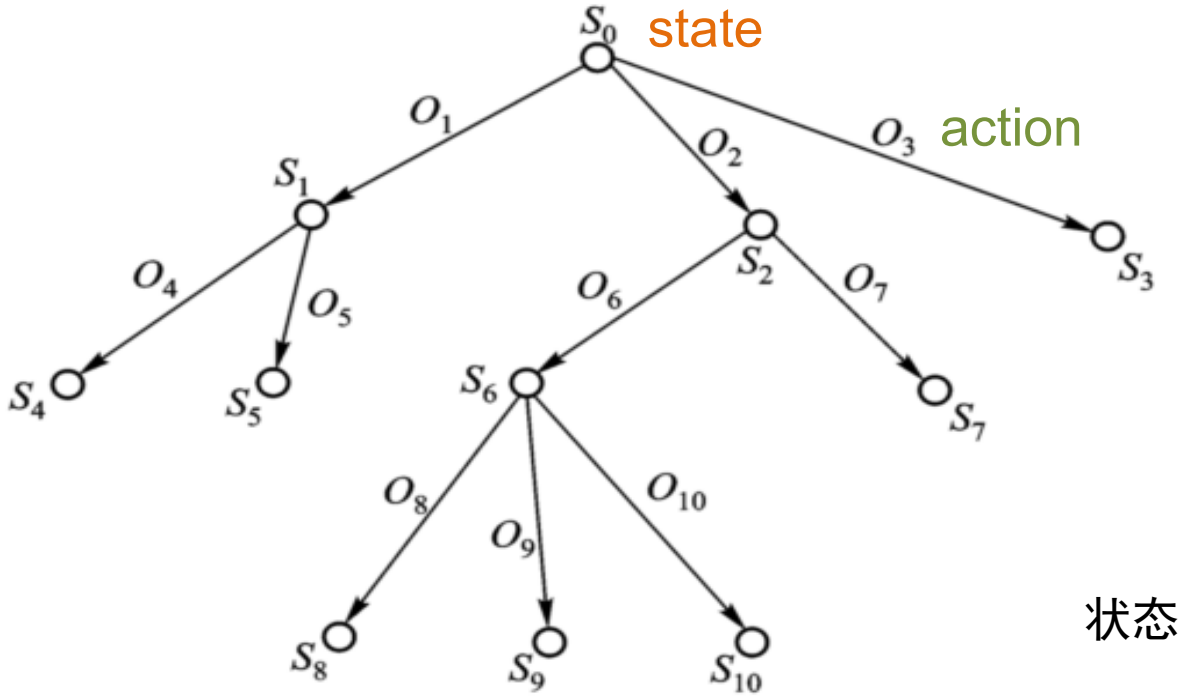
目标测试
goal test

评估当前状态是否为所求解的目标状态。

搜索算法的形式化描述

问题状态空间法的基本算法

- 根据问题，定义出相应的状态空间，确定出状态的一般表示，它含有相关对象的各种可能的排列。这里仅仅是定义这个空间的状态，而不必枚举该状态空间的所有状态，但由此可以得出问题的初始状态、目标状态，并能够表示出所有其它状态。
- 规定一组操作(算子) action，能够使状态从一个状态变为另一个状态。
- 决定一种搜索策略，使得能够从初始状态出发，沿某个路径达到目标状态。

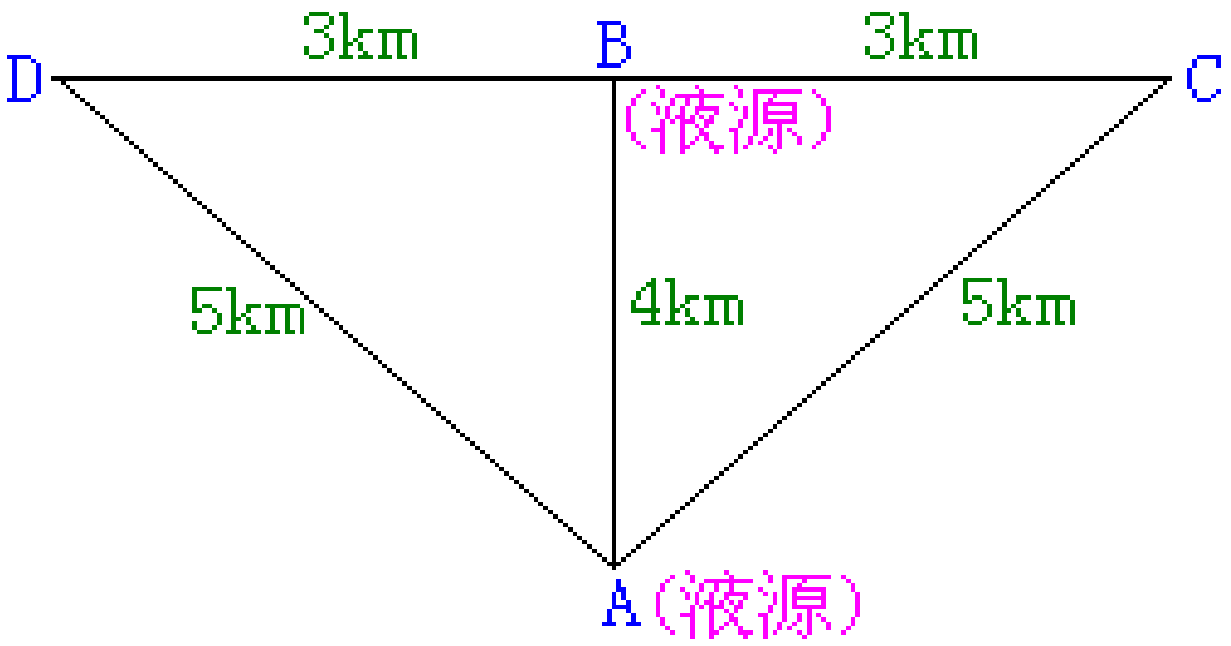


状态空间的有向图描述

分配问题

有两个液源A和B。A的流量为100L/m ,B的流量为50L/m。现要求它们以75L/m的流量分别供应两个同样的洗涤槽C，D。液体从液源经过最大输出能力为75L/m的管道进行分配，A、B、C、D的位置、距离如图2-2所示。并且要求只允许管子在液源或洗涤槽位置有接头。

问：如何连接管子使得管材最少？



分配问题

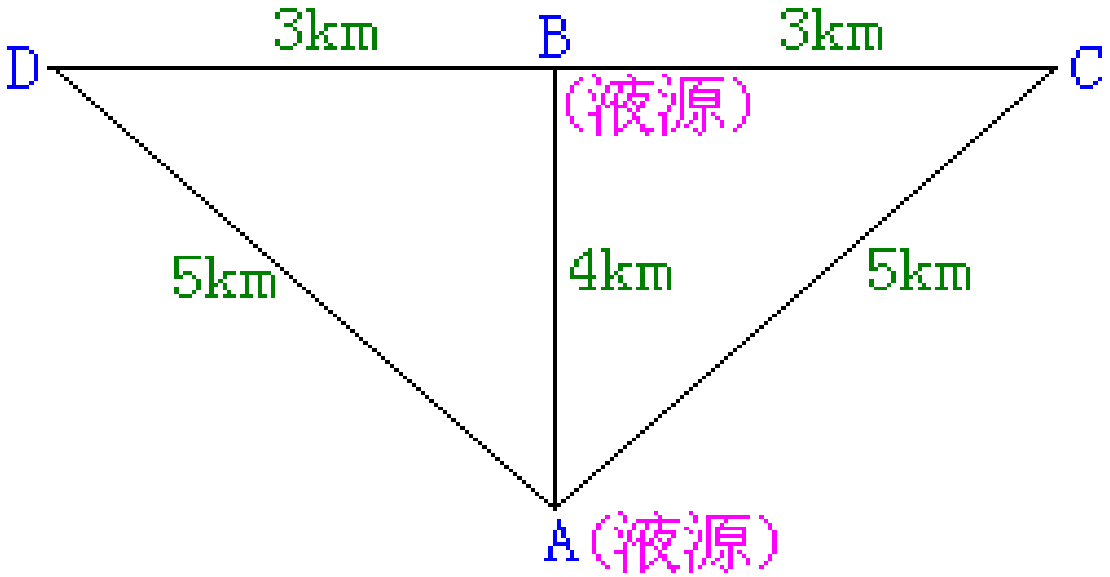
(1) 定义状态空间中的状态表示

状态以要素化的形式表示为：

(A=?　B=?　C=?　D=?)

初　　态：(A=100 B=50 C=0　D=0)

目标状态：(A=0　B=0　C=75 D=75)



分配问题

(2) 定义操作

现在取从一处到另一处流量的增量，为各点流量与各处所需流量的最大公约数(great common divisor)。100、50、75的GCD为25，作为基本分配量。

1 A→B (A≥75∧B<75)→(A-25，B+25，C,D)　4km

2 A→C (A≥25∧C<75)→(A-25，B，C+25)　5km

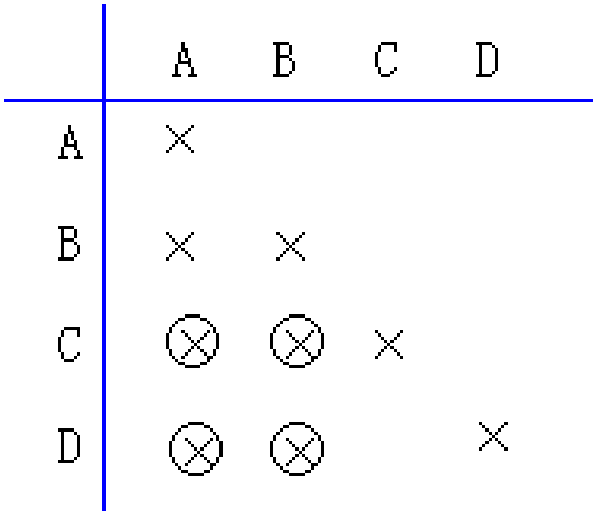
3 A→D (A≥25∧D<75)→(A-25，B，C，D+25)　5km

4 B→C (B≥25∧C<75)→(A，B-25，C+25，D)　3km

5 B→D (B≥25∧D<75)→(A，B-25，C，D+25)　3km

6 C→D (B≥25∧C<75)→(A，B-25，C-25，D+25)　6km

7 D→C (B≥25∧C<75)→(A，B，C+25，D-25)　6km



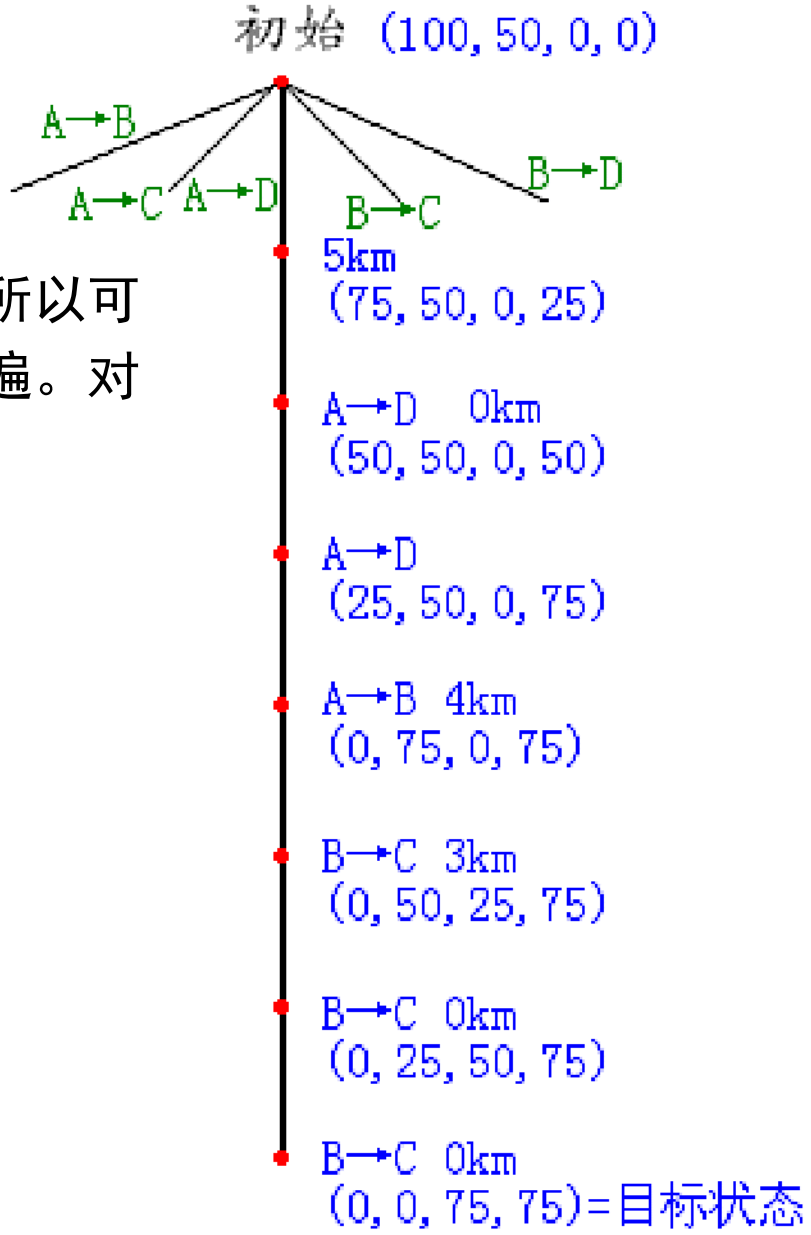
分配问题

(3) 定义搜索策略

因为现在没有给出任何知识可用来指导搜索，所以可采用耗尽式搜索，即每次却将7个操作试用一遍。对于该具体问题,搜索时要注意：

① 若操作重复时，只算一次距离;

② 边搜索边求出距离最短的管长。



主要内容

1. 问题求解与形式化描述 (Problem-Solving)

2. 搜索算法框架 (Search Algorithms)

3. 搜索策略 (Search Strategies)

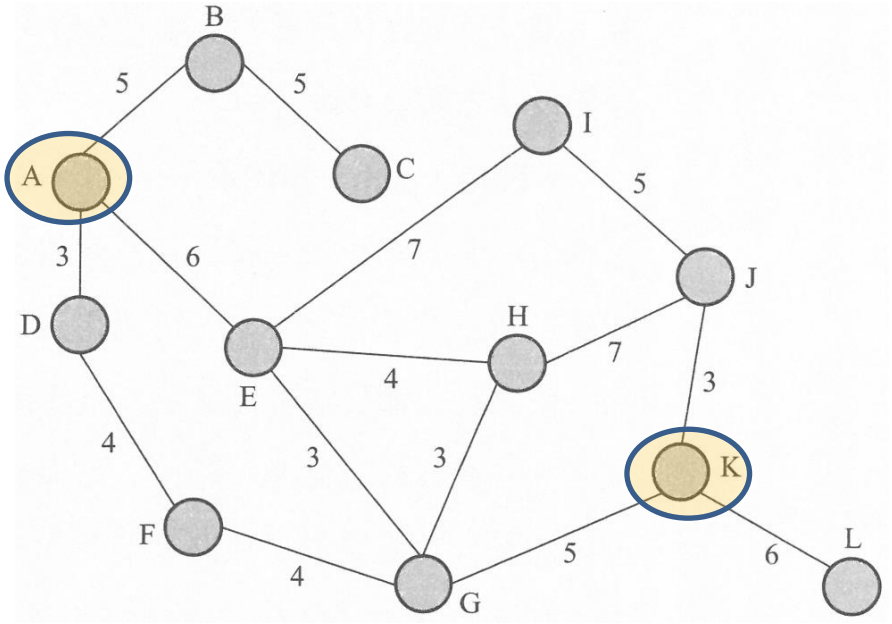
4. A* 搜索算法 (A* search)

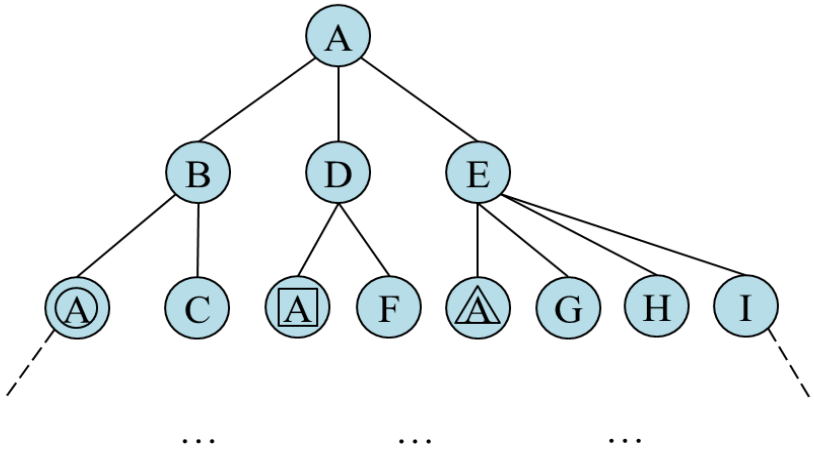
Bibliography:

- ✓ 吴飞 编著, “人工智能导论：模型与算法” (2020), 高等教育出版社. Ch 3.1.2, 3.1.3
- ✓ Stuart J. Russel, Peter Norvig, “Artificial Intelligence: A Modern Approach” (4th Ed. 2020); 中译版 “人工智能 一种现代的方法” (3rd Ed., 2009), 清华大学出版社. Ch 3.3
- ✓ 王万森 编著, “人工智能原理及其应用” (第4版, 2018), 电子工业出版社. Ch 4.1.3

搜索算法框架

搜索树：用一棵树来记录算法探索过的路径





在解决问题的过程中，搜索算法会时刻记录所有从初始结点出发已经探索过的路径，每次从中选出一条，从该路径末尾状态出发进行一次状态转移，探索一条尚未被探索过的新路径部分。

不同的结点可能包含相同的状态，对应从初始状态出发的不同路径。

搜索算法框架

搜索算法需要一个数据结构来记录搜索树的构造过程。对树中的每个结点，我们定义的数据结构包含四个元素：

- n.STATE: 对应状态空间中的状态；
- n.PARENT: 搜索树中产生该结点的结点（即父母结点）；
- n.ACTION: 父结点生成该结点时所采取的行动；
- 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代价；

```
function CHILD-NODE(problem, parent, action) returns a node
  return a node with
      STATE = problem.RESULT(parent.STATE, action),
      PARENT = parent, ACTION = action,
      PATH-COST = parent.PATH-COST + problem.STEP-COST(parent.STATE, action)
```

函数：TreeSearch（）

输入：根结点root，结点选择函数pick_from，后继结点计算函数successor_nodes

输出：从初始状态到终止状态的路径

```
1  F ← {根结点}
2  while F ≠ Ø do
3      n ← pick_from(F)
4      F ← F − {n}
5      if goal_test(n) then
6          return n.path
7      end
8      F ← F ∪ successor_nodes(n)
9  end
```

边缘(fringe)集合，也被叫做frontier 或 open list

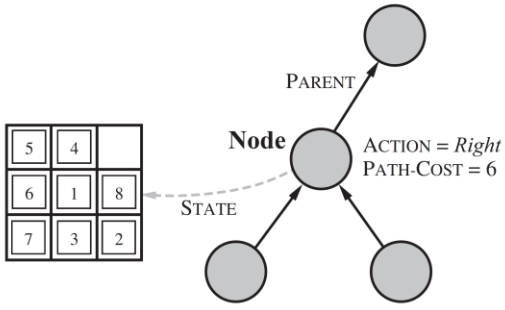


Figure 3.10 Nodes are the data structures from which the search tree is constructed. Each has a parent, a state, and various bookkeeping fields. Arrows point from child to parent.

Tree-like search

搜索算法框架

搜索算法的评价指标

常见的评判标准有如下四个:

- 完备性：当问题存在解时，算法是否能保证找到一个解。当然，这个解可能不是最优解。
- 最优性：搜索算法是否能保证找到的第一个解是最优解。
- 时间复杂度：找到一个解所需时间。
- 空间复杂度：在算法的运行过程中需要消耗的内存量。

完备性和最优性刻画了算法找到解的能力以及所求的解的质量，时间复杂度和空间复杂度衡量了算法的资源消耗， 它们通常用O符号(big O notation) 来描述。

在本章中， 时间复杂度将通过扩展的结点数量， 空间复杂度将通过算法同时记录的结点数量来衡量。

表3.1　搜索树中用于估计复杂度的变量含义

符号	含义
b	分支因子，即搜索树中每个结点最大的分支数目
d	根结点到最浅的目标结点的路径长度
m	搜索树中路径的最大可能长度
n	状态空间中状态的数量

搜索算法框架

剪枝搜索 – 并不是其所有的后继节点都值得被探索

在某些情况下，主动放弃一些后继结点能够提高搜索效率而不会影响最终搜索结果，甚至能解决无限循环（即算法不停机）问题。

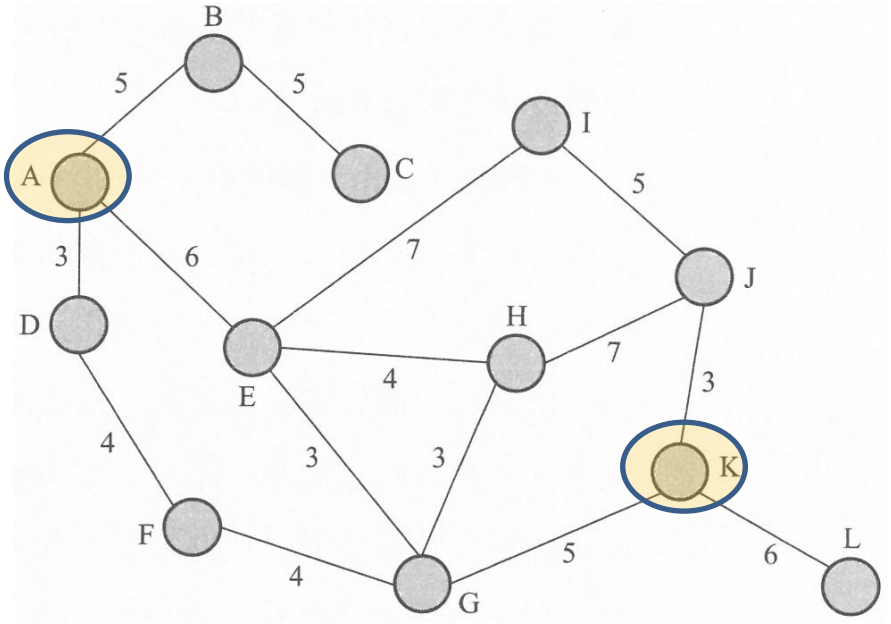
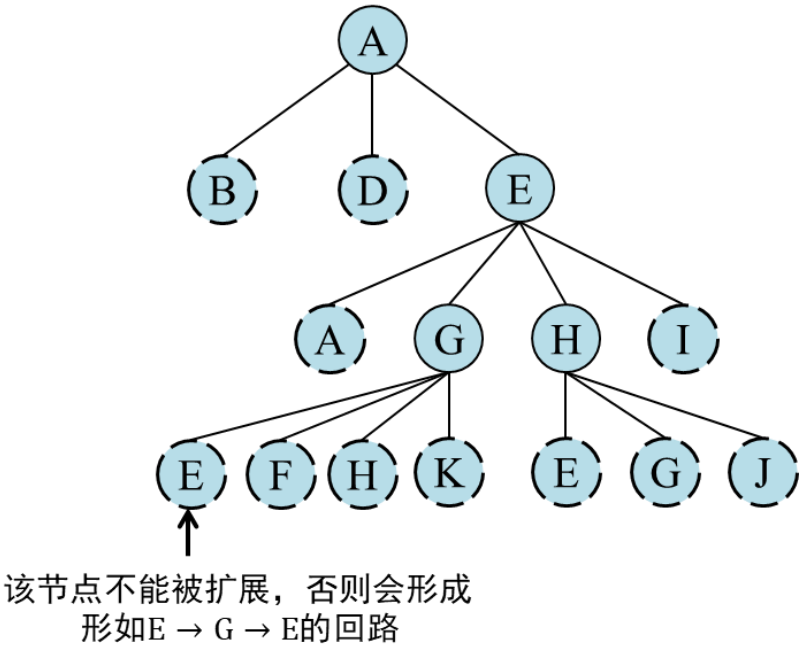


图 3.3 正在构建中的搜索树



注意到图3.3中右侧的路径为A → E → G → E → G → E → …，这意味着在某些搜索策略下（例如深度优先搜索），算法可能会沿着搜索树的右侧路径在状态E和状态G之间陷入无限循环，即出现环路或回路，搜索算法无法终止，此时算法不具有完备性。

搜索算法框架

- Tree search and graph search

算法 3.1　简单的搜索算法框架

```
函数：TreeSearch（）
输入：根结点root，结点选择函数pick_from，
后继结点计算函数successor_nodes
输出：从初始状态到终止状态的路径
1  F←{根结点}
2  while F ≠ Ø do
3      n←pick_from(F)
4      F←F−{n}
5      if goal_test(n) then
6          return n.path
7      end
8      F←F ∪ successor_nodes(n)
9  end
```

排除环路的树搜索

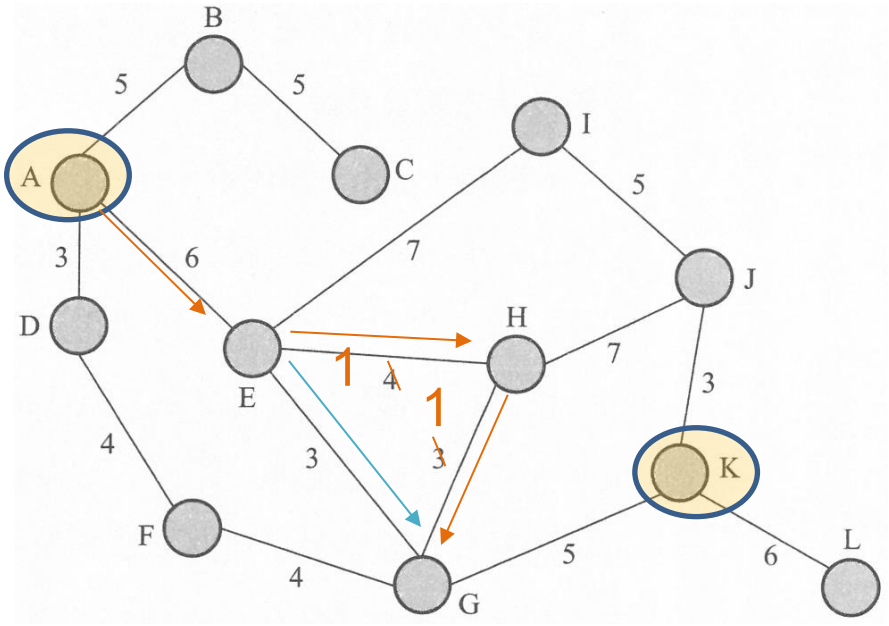
在图搜索策略下，在边缘集合中所有产生环路的节点都要被剪枝

算法3.2　图搜索算法

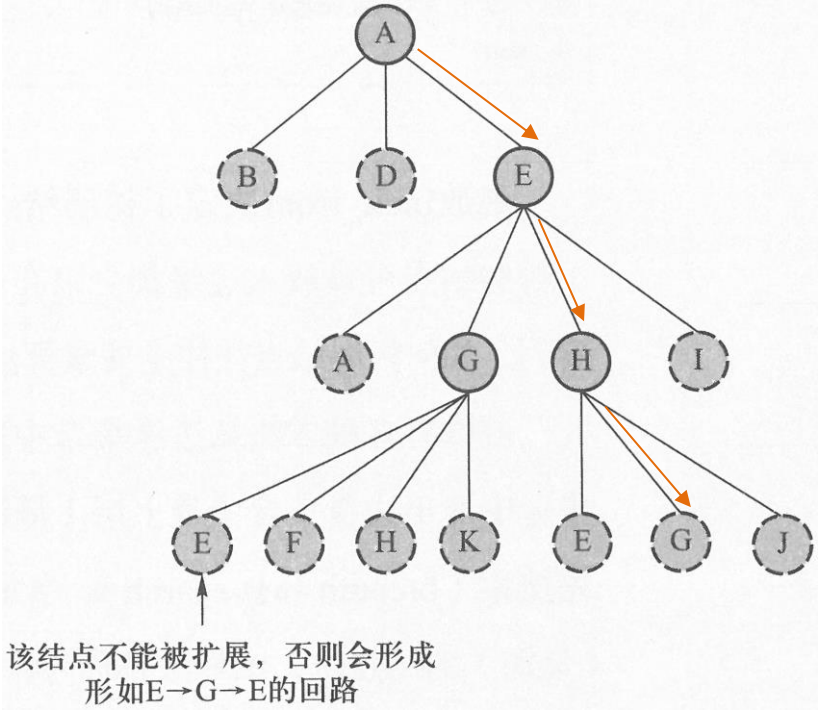
```
函数：GraphSearch（）
输入：根结点root，结点选择函数pick_from，
后继结点计算函数successor_nodes
输出：从初始状态到终止状态的路径
1  F←{根结点}
2  C←Ø
3  while F ≠ Ø do
4      n←pick_from(F)
5      F←F−{n}
6      if goal_test(n) then
7          return n.path
8      end
9      if n.state ∉ C then
10         C ← C ∪ {n.state}
11         F←F ∪ successor_nodes(n)
12     end
13 end
```

搜索算法框架

- Tree search and graph search



问题：寻找从城市A到城市K之间行驶时间最短路线？



E — H 和 H — G 之间的单步代价变为1，那么此时从A到K的最短路径为 A → E → H → G → K

在采用图搜索时，可能要针对具体算法添加更严格的条件限制或调整算法的流程来保证其最优性。

搜索算法框架

- Tree search and graph search

算法 3.1　简单的搜索算法框架

函数：TreeSearch（ ）

输入：根结点root，结点选择函数pick_from，后继结点计算函数successor_nodes
输出：从初始状态到终止状态的路径

```
1  F ← {根结点}
2  while F ≠ ∅ do
3      n ← pick_from(F)
4      F ← F − {n}
5      if goal_test(n) then
6          return n.path
7      end
8      F ← F ∪ successor_nodes(n)
9  end
```

排除环路的树搜索

- 不必担心对过去的重复，在一些问题形式化中，很少或不可能出现两条路径到达相同状态。
- 记住之前到达的状态（就像最佳优先搜索），这样能够检测所有冗余路径，并只保留每个状态的最优路径。
- 可以选择折中方法，检查循环，但通常不检查冗余路径。由于每个节点都有一个父指针链，可以通过跟踪父指针链，查看路径末端的状态之前是否在路径中出现过，从而不需要额外的内存即可检查是否存在循环。

问题特征分析

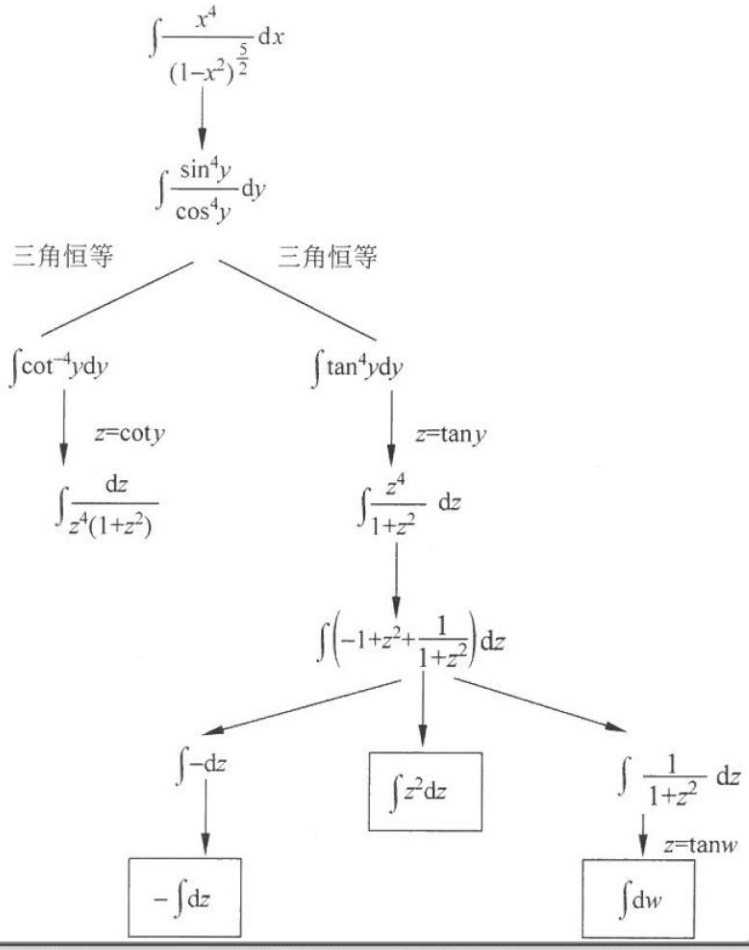
对问题的几个关键指标或特征加以分析。一般要考虑：

- 问题可分解成为一组独立的、更小、更容易解决的子问题吗？
- 当结果表明解题步骤不合适的时侯，能忽略或撤回吗？
- 问题的全域可预测吗？
- 在未与所有其它可能解作比较之前，能说当前的解是最好的吗？
- 用于求解问题的知识库是相容的吗？
- 求解问题一定需要大量的知识吗？或者说，有大量知识时候，搜索时应加以限制吗？
- 只把问题交给电脑，电脑就能返回答案吗？或者说，为得到问题的解，需要人机交互吗？

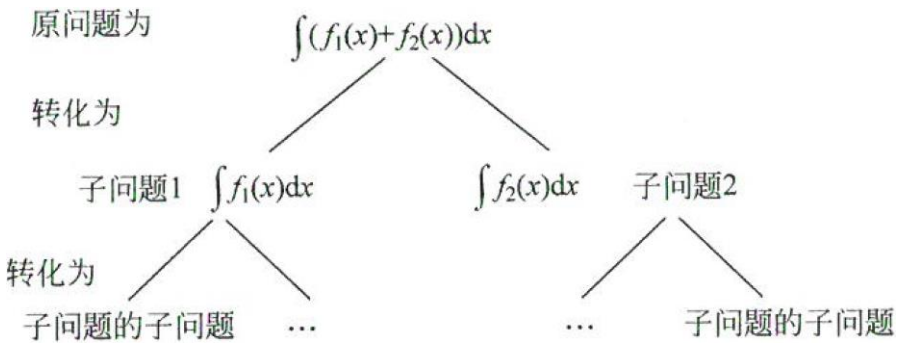
问题特征分析

◆ 问题是否可分解？

如果问题能分解成若干子问题，则将子问题解出后，原问题的解也就求出来了。称这种解决问题的方法为问题的归约 (reduction)。



符号积分



问题特征分析

- 问题是否可分解？

分解

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。

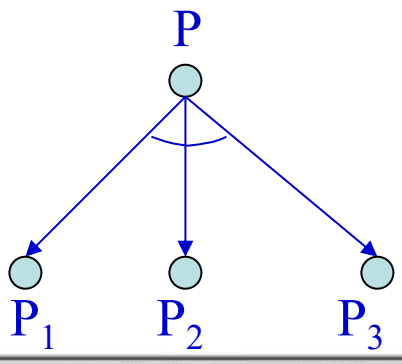
即分解所得到的子问题的“与”与原问题P等价。

等价变换

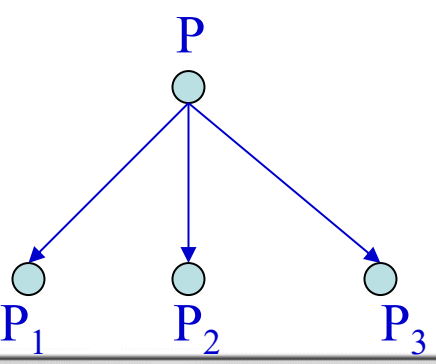
如果一个问题P可以归约为一组子问题$P_1,P_2,\ldots,P_n$，并且子问题P_i中只要有一个有解则原问题P就有解，只有当所有子问题P_i都无解时原问题P才无解，称此种归约为问题的等价变换，简称变换。

即变换所得到的子问题的“或”与原问题P等价。

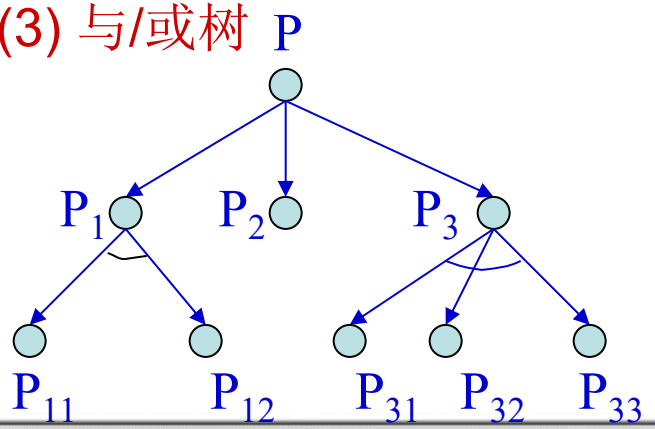
(1)与树　分解



(2) 或树　等价变换



(3) 与/或树



问题特征分析

- 问题是否可分解？

例4.4　三阶梵塔问题。要求把1号钢针上的3个金片全部移到3号钢针上，如下图所示。

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。

为了能够解决这一问题，首先需要定义该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“→”表示状态的转换。上述三元组中

i 代表金片A所在的钢针号

j 代表金片B所在的钢针号

k 代表金片C所在的钢针号

问题特征分析

利用问题归约方法，原问题可分解为以下三个子问题：

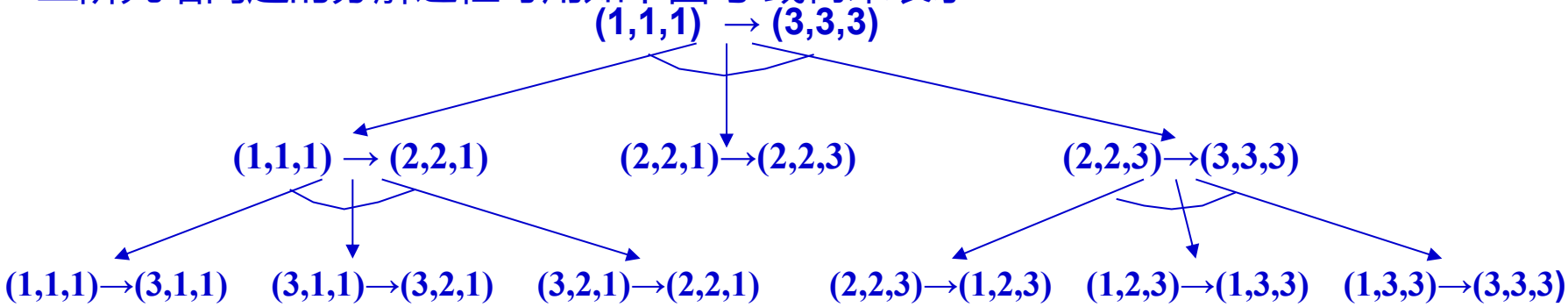
(1) 把金片A及B移到2号钢针上的双金片移动问题。即(1, 1, 1)→(2, 2, 1)

(2) 把金片C移到3号钢针上的单金片移动问题。即 (2, 2, 1)→(2, 2, 3)

(3) 把金片A及B移到3号钢针的双金片移动问题。即(2, 2, 3)→((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)→(3, 1, 1)　(3, 1, 1)→(3, 2, 1)　(3, 2, 1)→(2, 2, 1)　(2, 2, 1)→(2, 2, 3)

(2, 2, 3)→(1, 2, 3)　(1, 2, 3)→(1, 3, 3)　(1, 3, 3)→(3, 3, 3)

问题特征分析

◆ 问题求解步骤是否可撤回？

在问题求解的每一步骤完成后，分析一下它的路径，可分为3类：

（1）求解步骤可忽略

如定理证明，证明定理的每一件事情都为真或者为假，且总是保存知识库里，它是怎样推出来的对下一步并不重要，因而控制结构不需要带回溯。

（2）可复原

如走迷宫，实在走不通，可退回一步重来。这种搜索需用回溯技术，例如:

- 需用一定的控制结构；
- 需采用堆栈技术。

（3）不可复原

如下棋、决策等问题，要提前分析每走一步后会导致的结果。不可回头重来，这需要使用规划技术。

问题特征分析

- 问题全域可预测否？

有些问题的全域可预测，该问题空间有哪些状态是可以预测的，如水壶问题、定理证明，这些问题结局肯定，可只用开环控制结构。

有些问题的全域不可预测，如变化环境下机器人的控制，特别是危险环境下工作的机器人随时可能出意外，必须利用反馈信息，应使用闭环控制结构。

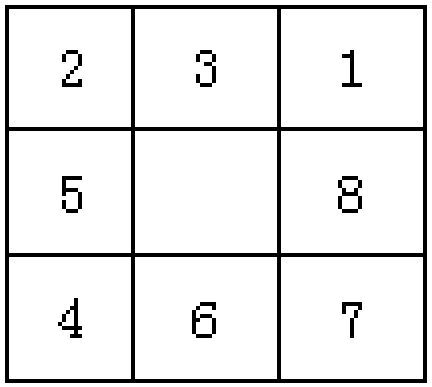
问题特征分析

- 问题要求的是最优解还是较满意解？

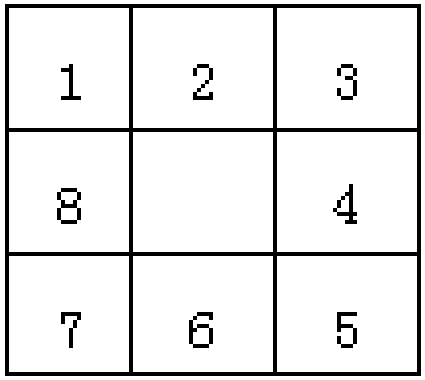
一般说来，最佳路径问题的计算较任意路径问题的计算要困难。如果使用的启发式方法不理想，那么对这个解的搜索就不可能很顺利。有些问题要求找出真正的最佳路径，可能任何启发式法都不能适用。因此，得进行穷举式搜索。

问题实例

例 八数码问题



初始状态



目标状态

状态集 S: 所有摆法

操作算子：

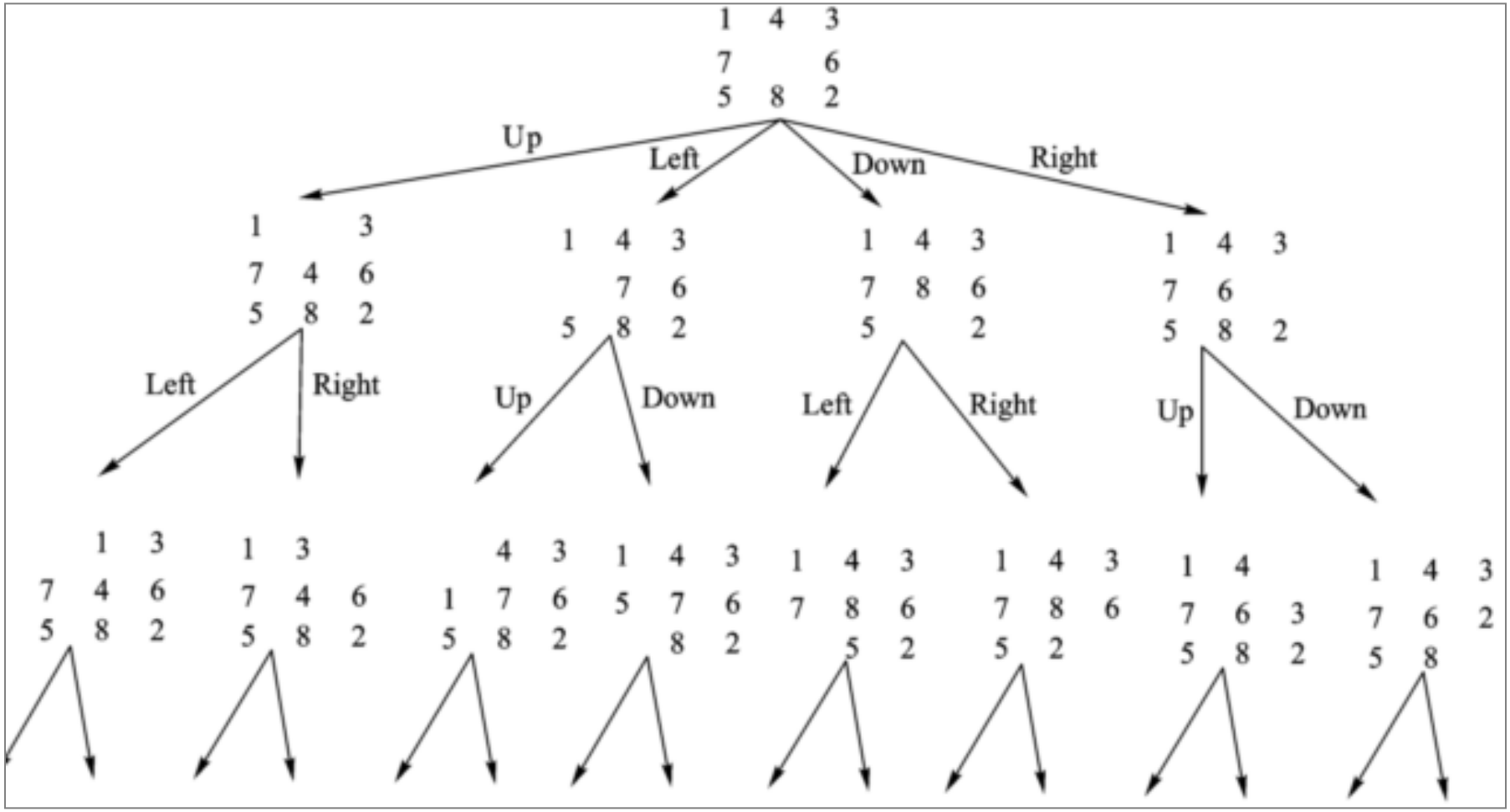
将空格向上移Up

将空格向左移Left

将空格向下移Down

将空格向右移Right

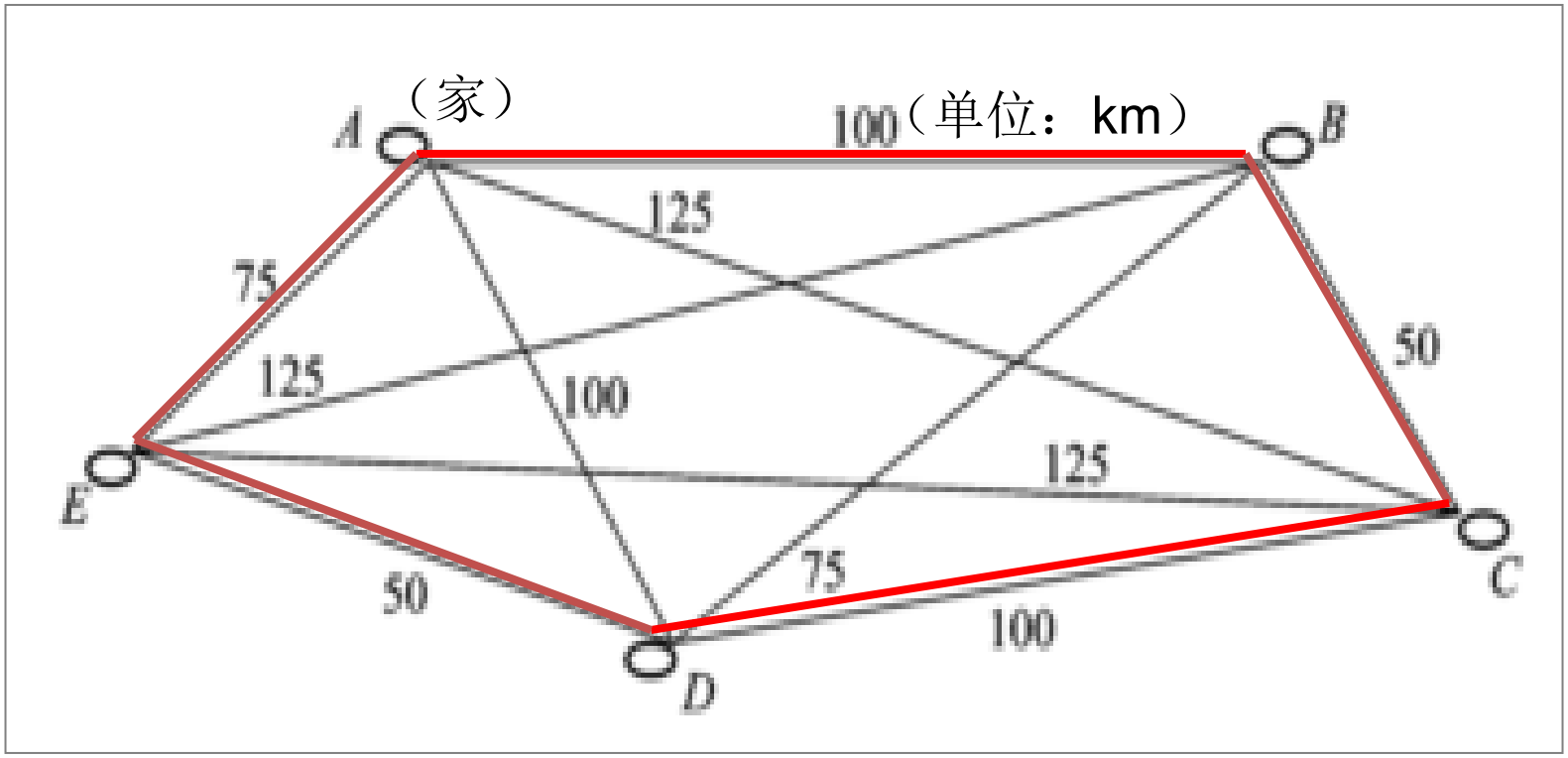
问题实例



八数码状态空间图

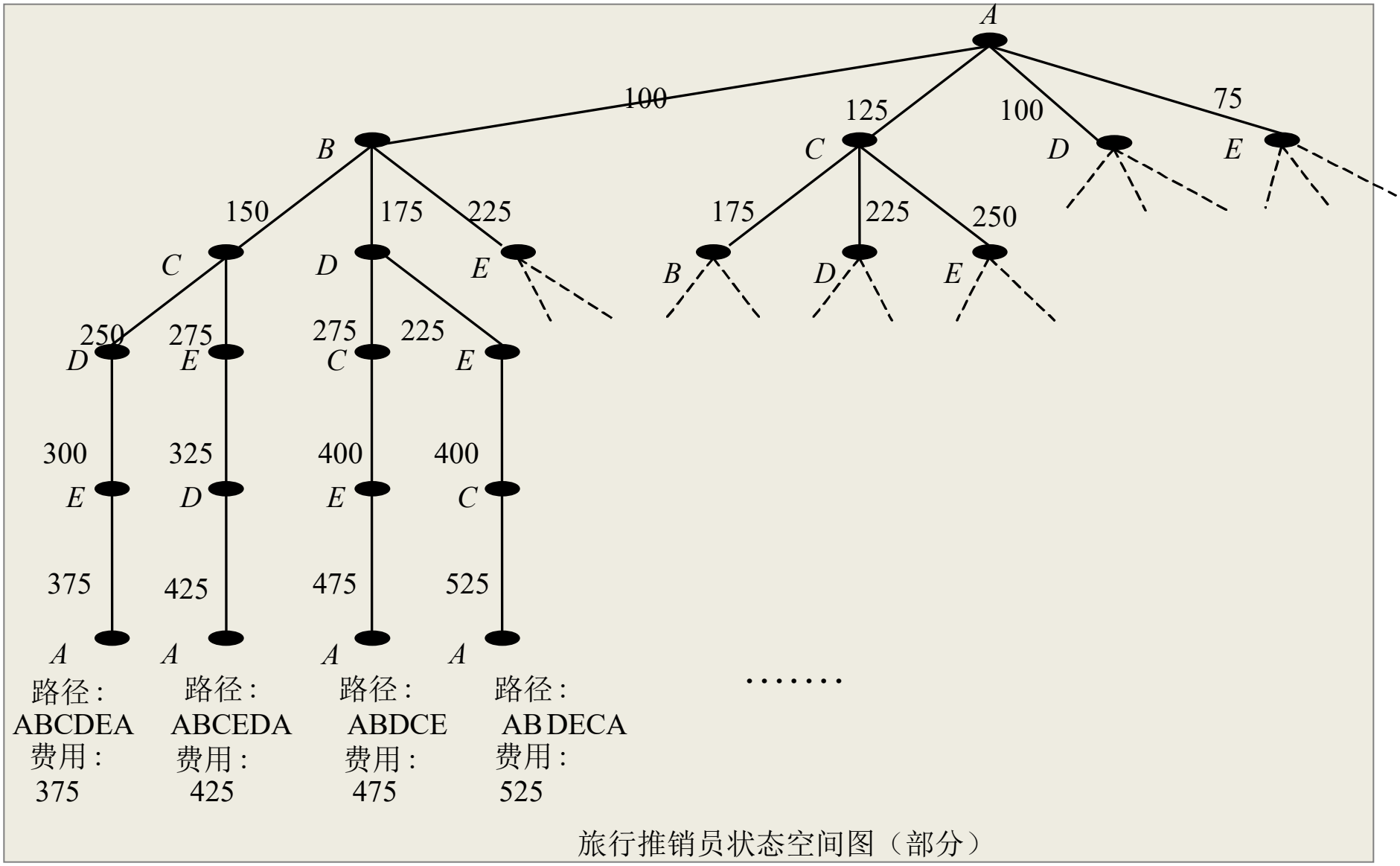
问题实例

例 旅行商问题 (traveling salesman problem, TSP) 或邮递员路径问题。



可能路径：费用为375的路径 (A, B, C, D, E, A)

问题实例



旅行推销员状态空间图（部分）

主要内容

1. 问题求解与形式化描述 (Problem-Solving)

2. 搜索算法框架 (Search Algorithms)

3. 搜索策略 (Search Strategies)

4. A* 搜索算法 (A* search)

Bibliography:

- ✓ Stuart J. Russel, Peter Norvig, “Artificial Intelligence: A Modern Approach” (4th Ed. 2020); 中译版 “人工智能 一种现代的方法” (3rd Ed., 2009), 清华大学出版社. Ch 3.4, 3.5
- ✓ 吴飞 编著, “人工智能导论：模型与算法” (2020), 高等教育出版社. Ch 3.2
- ✓ 王万良 编著, “人工智能导论” (第5版), 高等教育出版社, 2020. Ch 5.3, 5.4.1, 5.4.2

无信息搜索策略

- 宽度优先搜索 breadth-first search
- 深度优先搜索 depth-first search
- 回溯搜索 backtracking search
- 一致代价搜索 uniform-cost search
- 深度受限搜索 depth-limited search, 迭代加深搜索 iterative deepening search
- 双向搜索 Bidirectional search

Bibliography:

✓ Stuart J. Russel, Peter Norvig, “Artificial Intelligence: A Modern Approach” (4th Ed. 2020); 中译版 “人工智能 一种现代的方法” (3rd Ed., 2009), 清华大学出版社. Ch 3.4

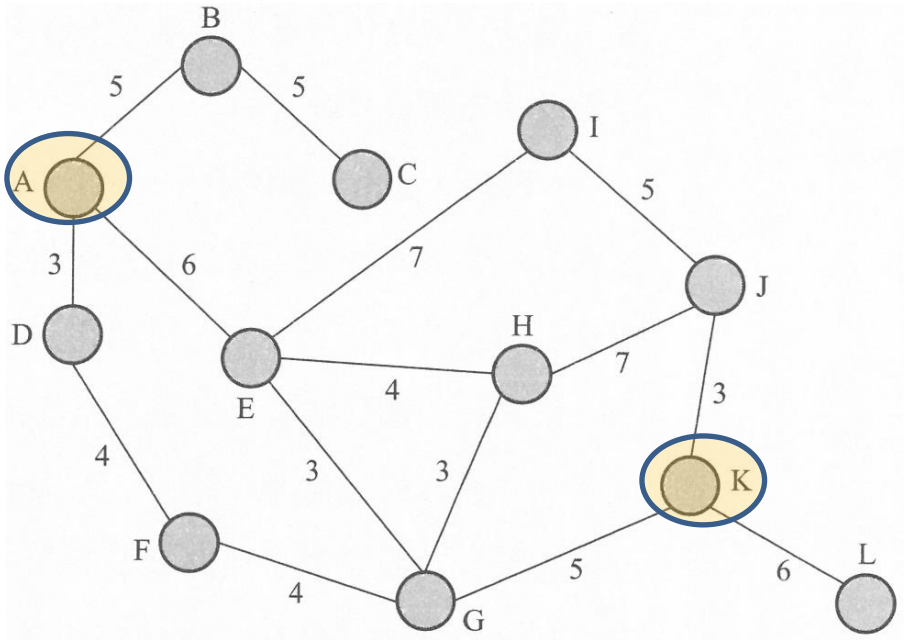
Bidirectional search

```
function BIBF-SEARCH(problem_F, f_F, problem_B, f_B) returns a solution node, or failure
  node_F ← NODE(problem_F.INITIAL)                    // Node for a start state
  node_B ← NODE(problem_B.INITIAL)                    // Node for a goal state
  frontier_F ← a priority queue ordered by f_F, with node_F as an element
  frontier_B ← a priority queue ordered by f_B, with node_B as an element
  reached_F ← a lookup table, with one key node_F.STATE and value node_F
  reached_B ← a lookup table, with one key node_B.STATE and value node_B
  solution ← failure
  while not TERMINATED(solution, frontier_F, frontier_B) do
    if f_F(TOP(frontier_F)) < f_B(TOP(frontier_B)) then
      solution ← PROCEED(F, problem_F, frontier_F, reached_F, reached_B, solution)
    else solution ← PROCEED(B, problem_B, frontier_B, reached_B, reached_F, solution)
  return solution

function PROCEED(dir, problem, frontier, reached, reached_2, solution) returns a solution
        // Expand node on frontier; check against the other frontier in reached_2.
        // The variable "dir" is the direction: either F for forward or B for backward.
  node ← POP(frontier)
  for each child in EXPAND(problem, node) do
    s ← child.STATE
    if s not in reached or PATH-COST(child) < PATH-COST(reached[s]) then
      reached[s] ← child
      add child to frontier
      if s is in reached_2 then
        solution_2 ← JOIN-NODES(dir, child, reached_2[s]))
        if PATH-COST(solution_2) < PATH-COST(solution) then
          solution ← solution_2
  return solution
```

启发式搜索策略 (有信息搜索)

Heuristic Search Strategies 在搜索的过程中利用与所求解问题相关的辅助信息，其代表算法为**贪婪最佳优先搜索**（Greedy best-first search）和**A*搜索**。



问题：寻找从城市A到城市K之间行驶时间最短路线？

启发式策略

- 什么是启发式信息？

  用来简化搜索过程有关具体问题领域的特性的信息叫做启发信息（Heuristic Information）。

- 启发式图搜索策略（利用启发信息的搜索方法）的特点：重排OPEN表，选择最有希望的节点加以扩展。

- 种类：*A*、*A**算法等。

- 运用启发式策略的两种基本情况：

（1）一个问题由于存在问题陈述和数据获取的模糊性，可能会使它没有一个确定的解。

（2）虽然一个问题可能有确定解，但是其状态空间特别大，搜索中生成扩展的状态数会随着搜索的深度呈指数级增长。

启发信息和评估函数

- 求解问题中能利用的大多是非完备的启发信息：

（1）求解问题系统不可能知道与实际问题有关的全部信息，因而无法知道该问题的全部状态空间，也不可能用一套算法来求解所有的问题。

（2）有些问题在理论上虽然存在着求解算法，但是在工程实践中，这些算法不是效率太低，就是根本无法实现。

一字棋：9！，西洋跳棋：10^{78}，国际象棋：10^{120}，围棋：10^{761}。

假设每步可以搜索一个棋局，用极限并行速度（10^{-104}年/步）来处理，搜索一遍国际象棋的全部棋局也得10^{16}年即1亿亿年才可以算完！

启发信息和评估函数

- 按运用的方法分类：
  1) 陈述性启发信息：用于更准确、更精炼地描述状态
  2) 过程性启发信息：用于构造操作算子
  3) 控制性启发信息：表示控制策略的知识
- 按作用分类：
  1) 用于扩展节点的选择，即用于决定应先扩展哪一个节点，以免盲目扩展。
  2) 用于生成节点的选择，即用于决定要生成哪些后继节点，以免盲目生成过多无用的节点。
  3) 用于删除节点的选择，即用于决定删除哪些无用节点，以免造成进一步的时空浪费。

启发信息和评估函数

- 评估函数（evaluation function）：估算节点“希望”程度的量度。

- 评估函数值 $f(n)$：从初始节点经过 n 节点到达目标节点的路径的最小代价估计值，其一般形式是

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

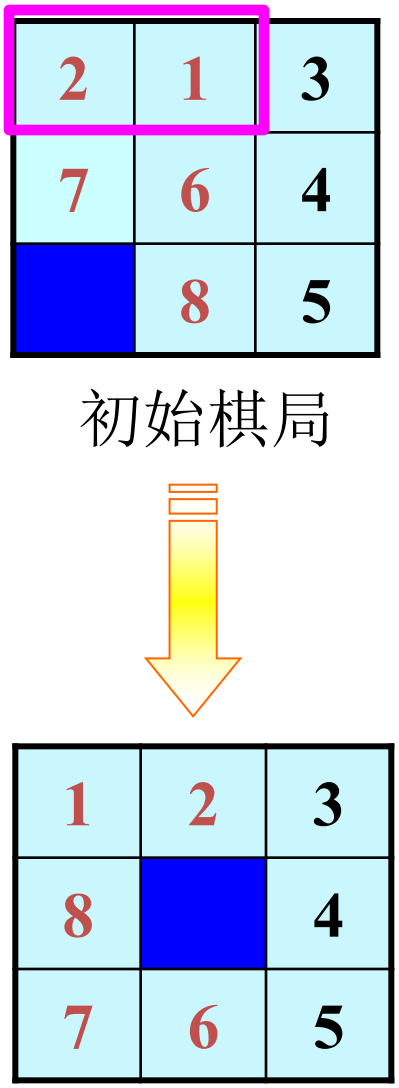
- $g(n)$：从初始节点 S_0 到节点 n 的实际代价；
- $h(n)$：从节点 n 到目标节点 S_g 的最优路径的估计代价，称为启发函数。

$h(n)$ 比重大：降低搜索工作量，但可能导致找不到最优解；

$h(n)$ 比重小：一般导致工作量加大，极限情况下变为盲目搜索，但可能可以找到最优解。

启发信息和评估函数

- 例 八数码问题的启发函数：
- 启发函数1: 取一棋局与目标棋局相比，其位置不符的数码数目，例如 $h(S_0) = 5$;
- 启发函数2: 各数码移到目标位置所需移动的距离的总和,例如 $h(S_0) = 6$;
- 启发函数3：对每一对逆转数码乘以一个倍数, 例如3倍，则$h(S_0) = 3$;
- 启发函数4：将位置不符数码数目的总和与3倍数码逆转数目相加，例如 $h(S_0) = 8$。



主要内容

1. 问题求解与形式化描述 (Problem-Solving)

2. 搜索算法框架 (Search Algorithms)

3. 搜索策略 (Search Strategies)

4. A* 搜索算法 (A* search)

Bibliography:

- ✓ 吴飞 编著, “人工智能导论：模型与算法” (2020), 高等教育出版社. Ch 3.2
- ✓ 王万良 编著, “人工智能导论” (第5版), 高等教育出版社, 2020. Ch 5.4
- ✓ Stuart J. Russel, Peter Norvig, “Artificial Intelligence: A Modern Approach” (4th Ed. 2020); 中译版 “人工智能 一种现代方法” (3rd Ed., 2009), 清华大学出版社. Ch 3.5
- ✓ 王万森 编著, “人工智能原理及其应用” (第4版，2018), 电子工业出版社. Ch 4.2

贪婪最佳优先搜索

贪婪最佳优先搜索(Greedy best-first search)：评价函数$f(n)$=启发函数$h(n)$

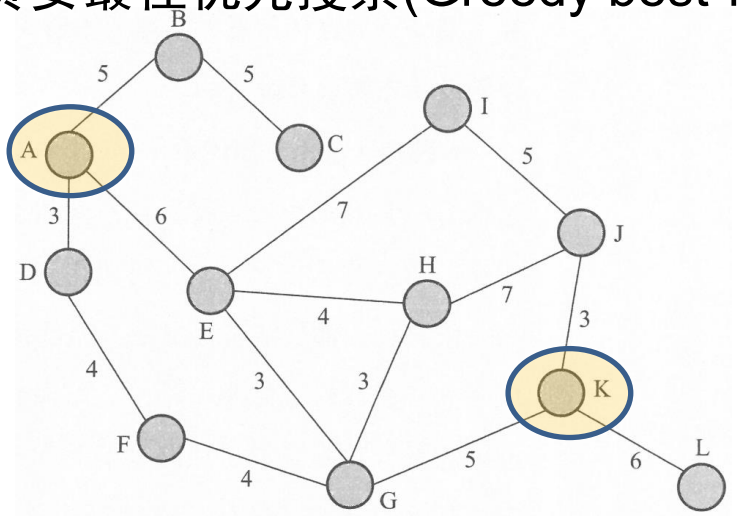
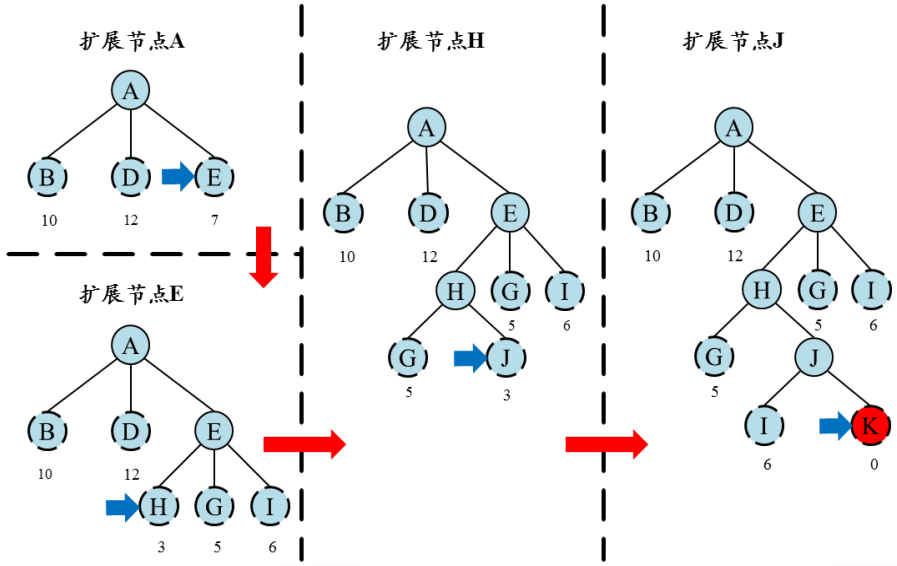


表3.2　每个状态（城市）所对应启发函数取值

状态	A	B	C	D	E	F	G	H	I	J	K	L
$h(n)$	13	10	6	12	7	8	5	3	6	3	0	6

辅助信息（启发函数）：
任意一个城市与终点城市K之间的直线距离

问题：寻找从城市A到城市K之间行驶时间最短路线？



算法找到了一条从起始节点到终点节点的路径A → E → H → J → K，但这条路径并不是最短路径，实际上最短路径为A → E → G → K。

A搜索算法

- OPEN表：保留所有已生成而未扩展的状态；
- CLOSED表：记录已扩展过的状态。

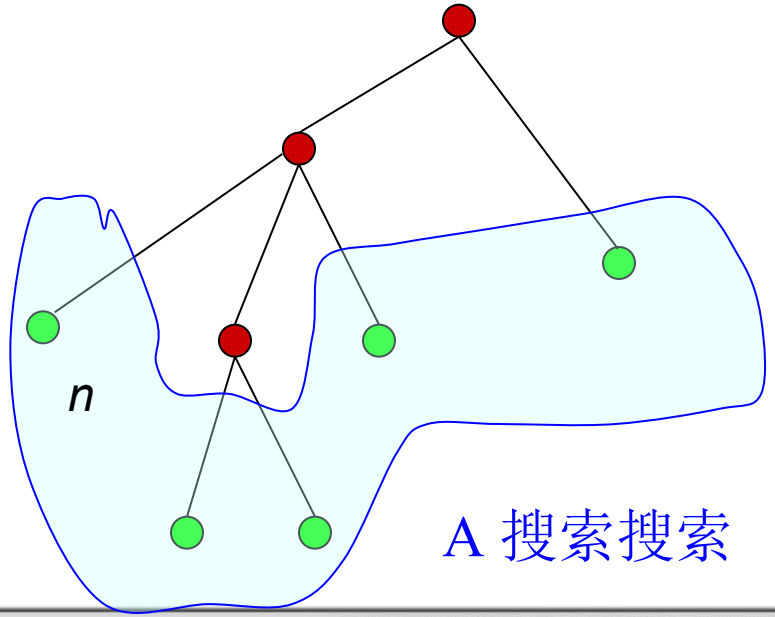
A 搜索算法：使用了估价函数 f 的最佳优先搜索。

估价函数 $f(n) = g(n) + h(n)$

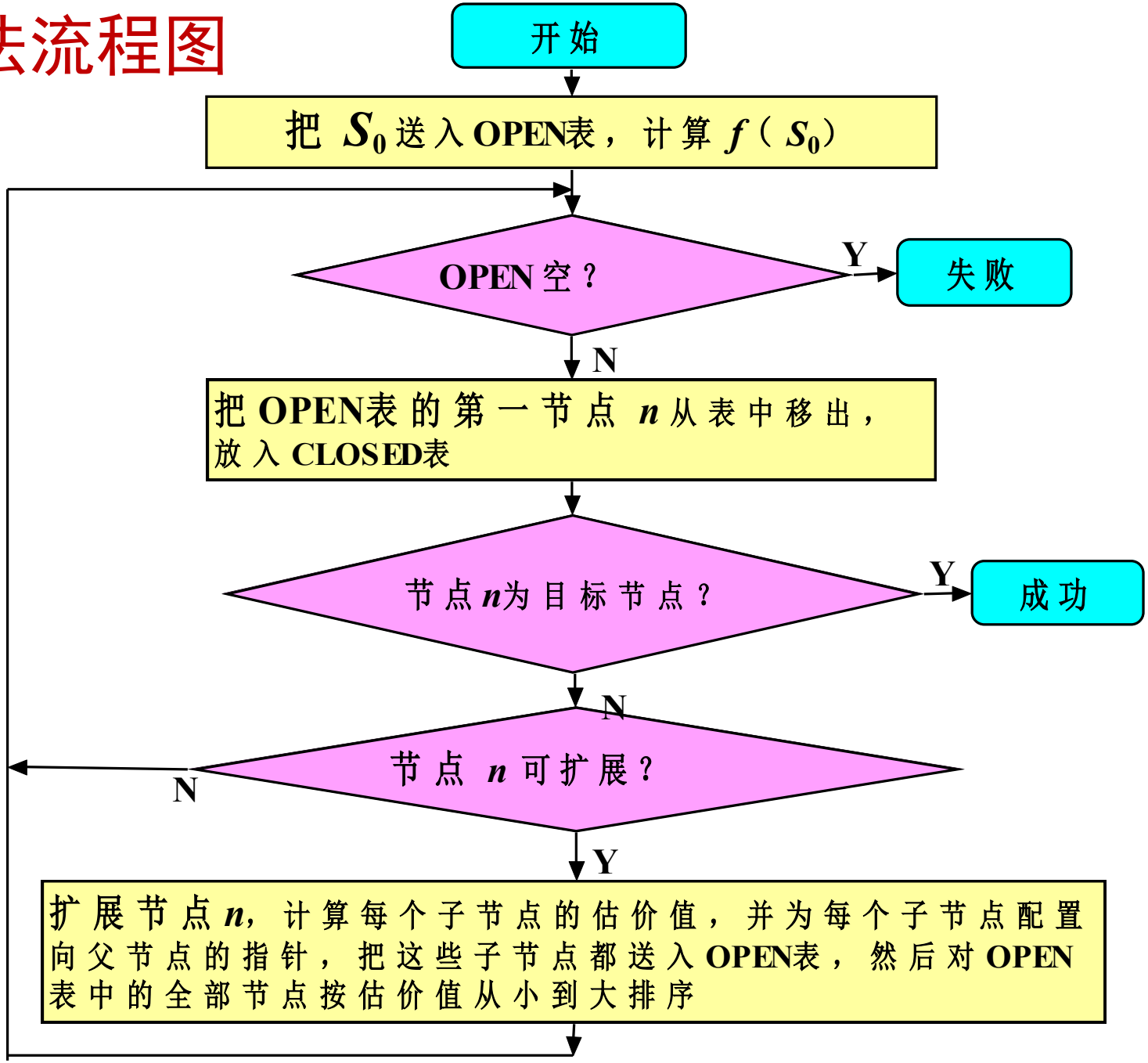
如何寻找并设计启发函数 $h(n)$，然后以 $f(n)$ 的大小来排列OPEN表中待扩展状态的次序，每次选择 $f(n)$值最小者进行扩展。

$g(n)$：状态n的实际代价，例如搜索的深度；

$h(n)$：对状态n与目标“接近程度”的某种启发式估计。



A 算法流程图


开始
把 S_0 送入OPEN表，计算 $f(S_0)$
OPEN空？
Y
失败
N
把OPEN表的第一节点 n 从表中移出，放入CLOSED表
节点 n 为目标节点？
Y
成功
N
节点 n 可扩展？
N
Y
扩展节点 n，计算每个子节点的估价值，并为每个子节点配置向父节点的指针，把这些子节点都送入OPEN表，然后对OPEN表中的全部节点按估价值从小到大排序

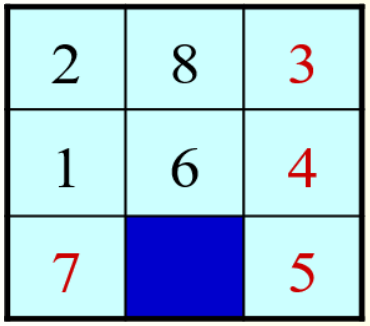
A搜索算法

例 利用 A 搜索算法求解八数码问题，问最少移动多少次就可达到目标状态？

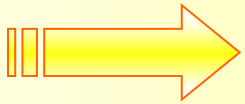
估价函数定义为 $f(n)=g(n)+h(n)$

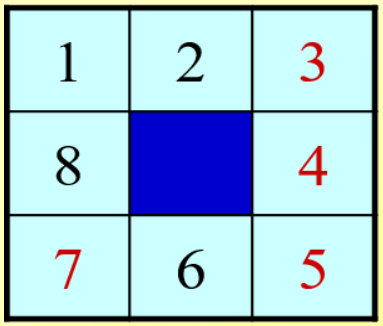
$g(n)$：节点 n 的深度，如 $g(S_0)=0$。

$h(n)$：节点 n 与目标棋局不相同的位数（包括空格），简称“不在位数”，如 $h(S_0)=5$。



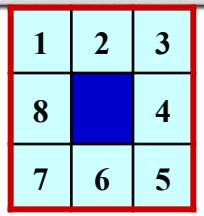
初始状态 S_0

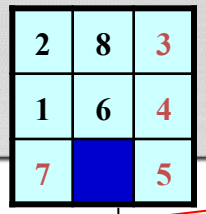




目标状态

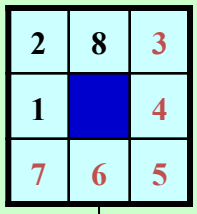
操作算子集：↑, ↓, →, ←



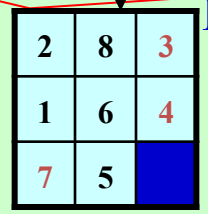


S（0+5=5）

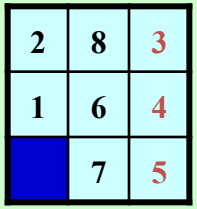
A（1+3=4）



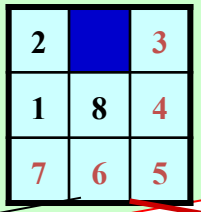
B（1+6=7）

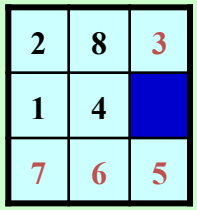


C（1+6=7）

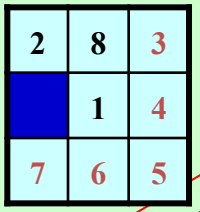


D（2+4=6）

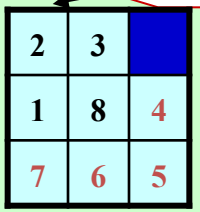


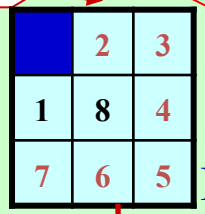


E（2+5=7）



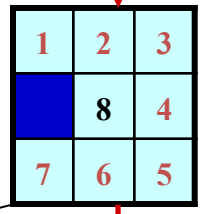
F（2+4=6）





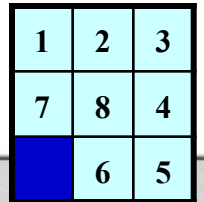
H（3+3=6）

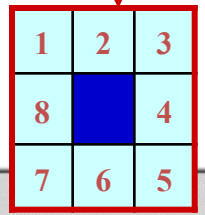
G（3+5=8）



I（4+2=6）

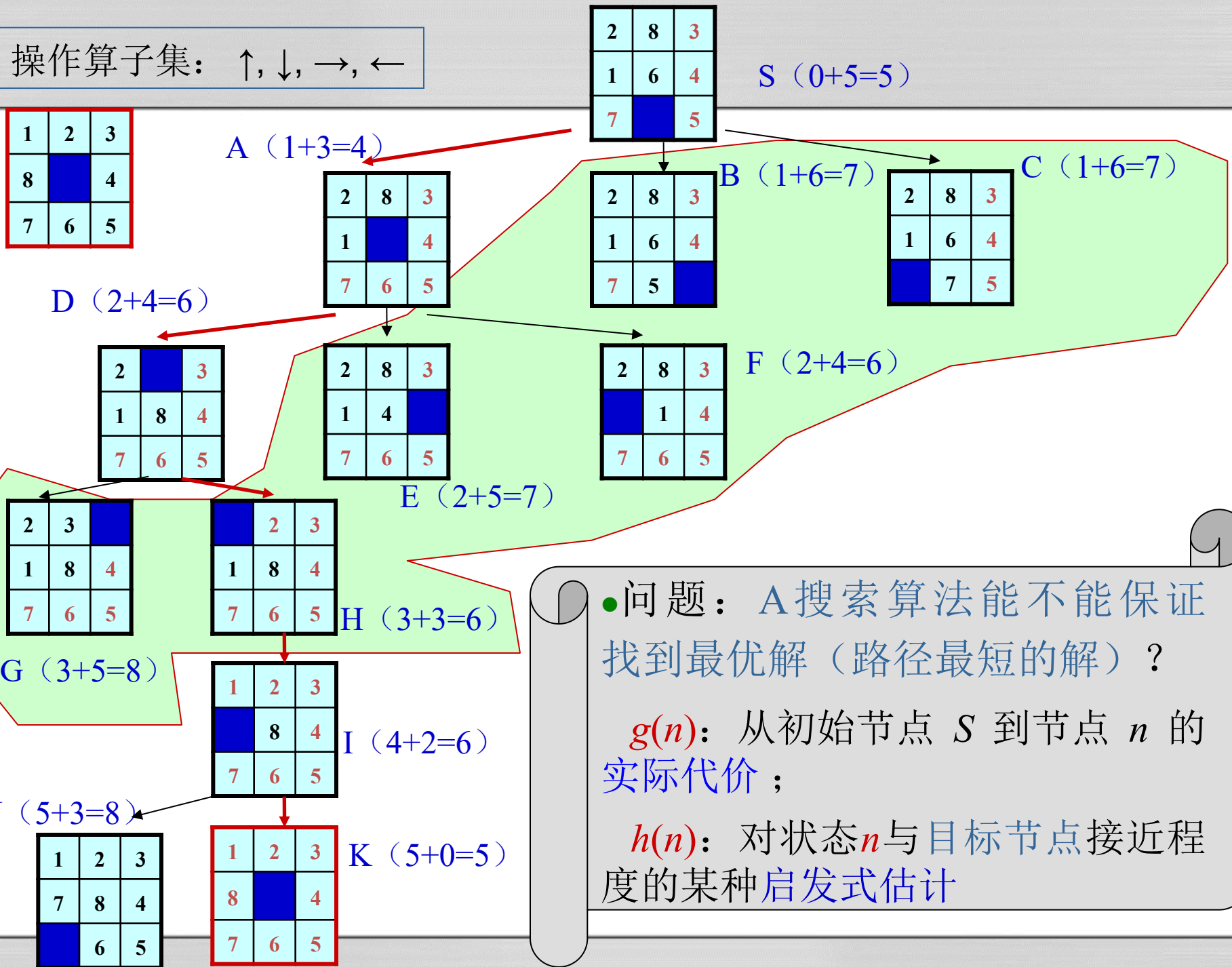
J（5+3=8）





K（5+0=5）

操作算子集：↑, ↓, →, ←



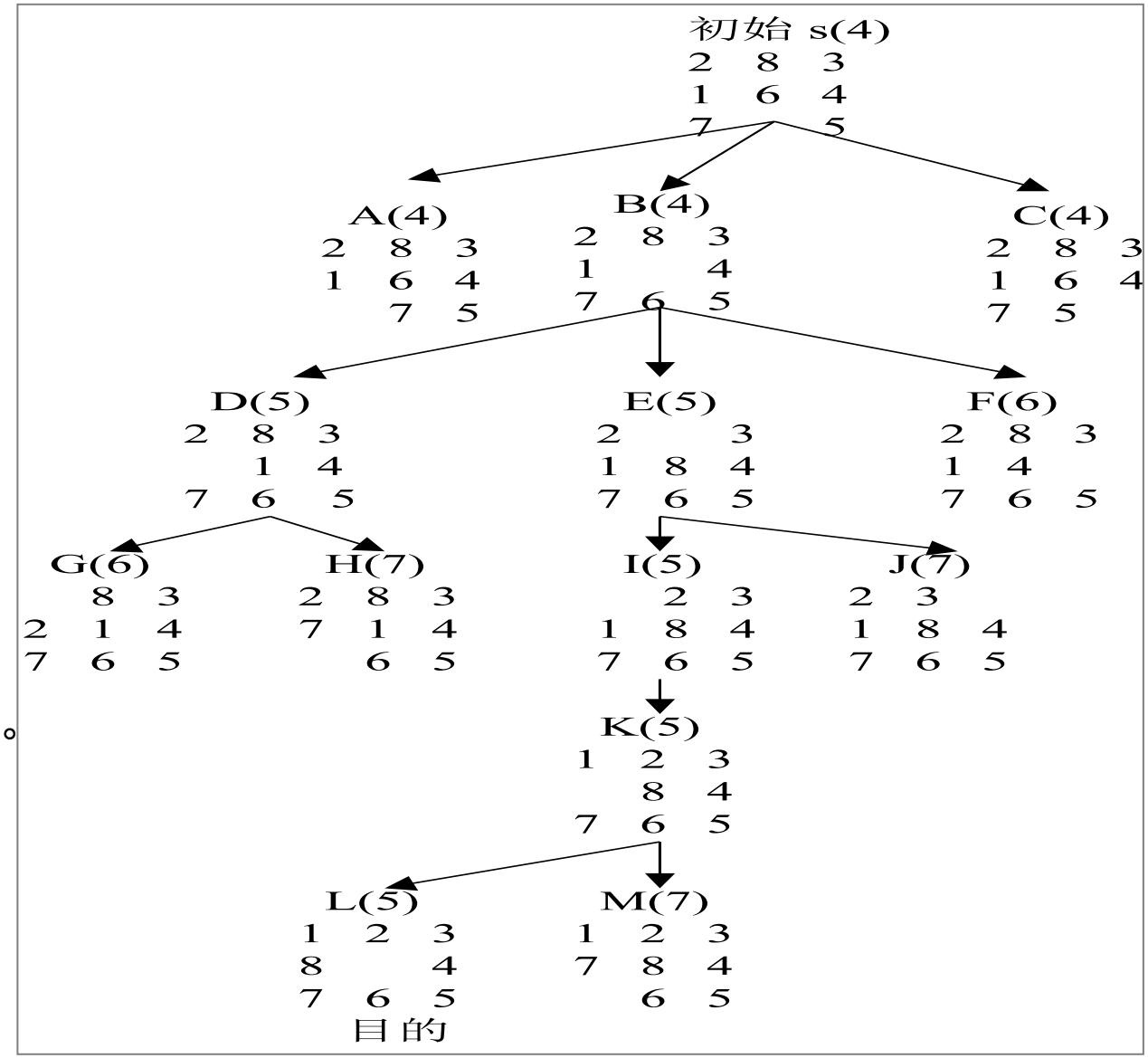
- 问题：A搜索算法能不能保证找到最优解（路径最短的解）？

$g(n)$：从初始节点 S 到节点 n 的实际代价；

$h(n)$：对状态n与目标节点接近程度的某种启发式估计

A*搜索算法

- 如果某一问题有解，那么利用A*搜索算法对该问题进行搜索则一定能搜索到解，并且一定能搜索到最优的解而结束。
- 上例中的八数码A搜索树也是A*搜索树，所得的解路径 (s,B,E,I,K,L) 为最优解路径，其步数为状态 $L(5)$ 上所标注的5 。



A*搜索算法性能分析

◆ 可容性 admissible

对任意结点n，有$h(n) \le h^*(n)$，如果n是目标结点，则有$h(n) = 0$。

满足可容性的启发函数不会高估 (over-estimate) 从结点n到终止结点的代价，是结点n到终止结点实际代价的下界。

- A^*搜索算法 $h(n) = 0 \Rightarrow f(n) = g(n)$

⇕

宽度优先搜索算法

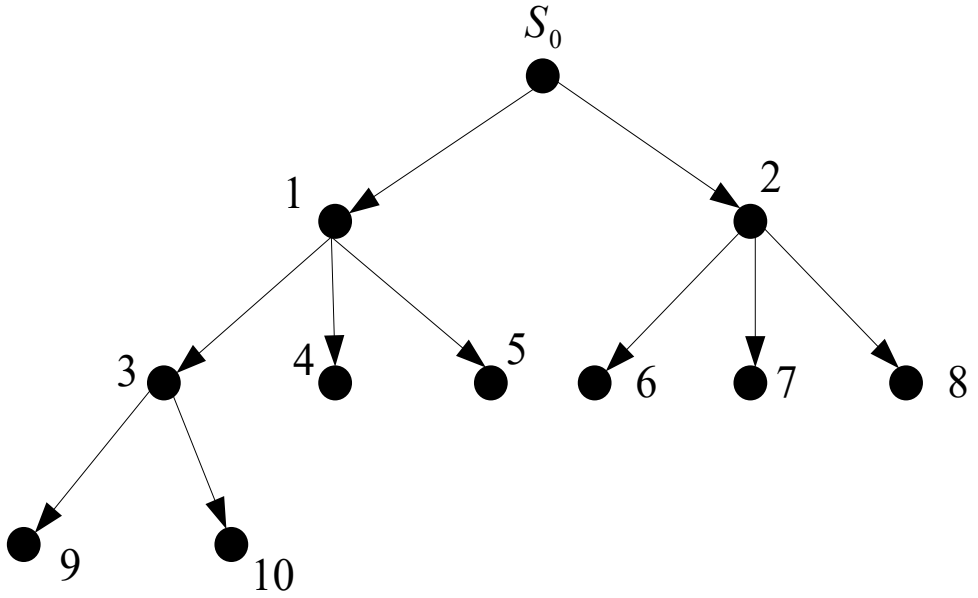


图5.6 宽度优先搜索法中状态的搜索次序

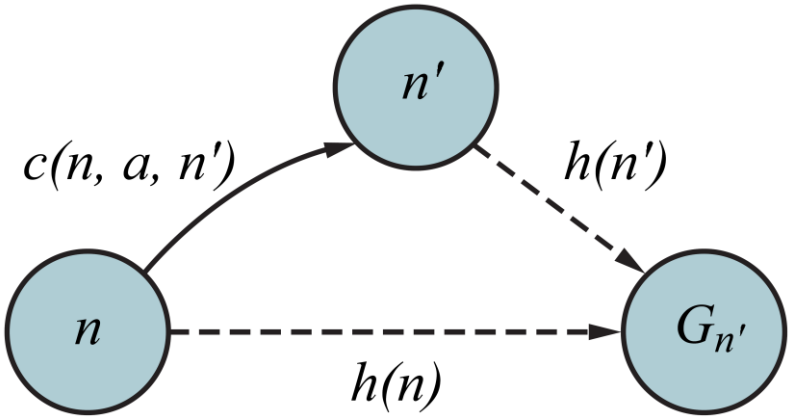
A*搜索算法性能分析

◆ 一致性 consistency

如果某一启发函数$h(n)$满足：

- 1) 对所有状态n_i和n_j，其中n_j是n_i的后裔，满足$h(n_i) - h(n_j) \leq c(n_i, a, n_j)$，其中$c(n_i, a, n_j)$是从$n_i$到$n_j$的实际代价。
- 2) 目的状态的启发函数值为0。

称$h(n)$满足一致则性条件。

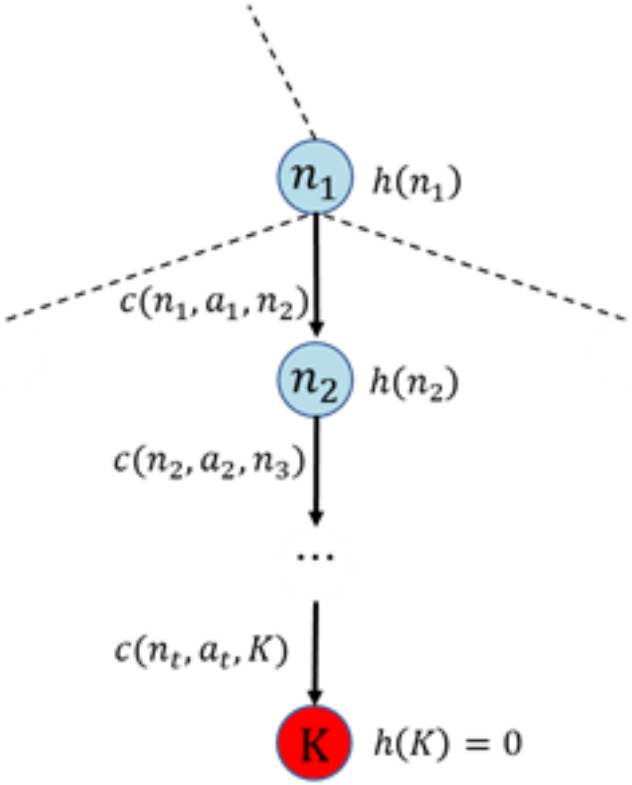


A^*搜索算法并不要求 $g(n) = g^*(n)$，则可能会沿着一条非最佳路径搜索到某一中间状态n。

A*搜索算法中采用一致性启发函数，可以减少比较代价和调整路径的工作量，从而减少搜索代价。

A*搜索算法性能分析

可容性与一致性的关系：
满足一致性条件的启发函数 → 一定满足可容性条件



若不存在终止节点，可以认为 $h^*(n_1)$ 的取值无穷大，启发函数在 n_1 上满足可容性。

以n_1为根结点的子树，假设从该节点到达终止结点代价最小的路径为$n_1 \rightarrow n_2 \rightarrow \ldots \rightarrow$ K。由于h(K) = 0且启发函数满足一致性条件，故可以推导可容性条件如下：

$$
\begin{aligned}
h(n_1) &\le h(n_2) + c(n_1, a_1, n_2) \\
&\le h(n_3) + c(n_2, a_2, n_3) + c(n_1, a_1, n_2) \\
&\le \cdots \\
&\le c(n_1, a_1, n_2) + c(n_2, a_2, n_3) + \cdots + c(n_t, a_t, \mathrm{K}) \\
&= h^*(n_1)
\end{aligned}
$$

A*搜索算法性能分析

◆ 信息性

在两个A*启发策略的h_1和h_2中，如果对搜索空间中的任一状态n都有$h_1(n) \leq h_2(n)$，就称策略h_2比h_1具有更多的信息性。

如果某一搜索策略的$h(n)$越大，则A*算法搜索的信息性越多，所搜索的状态越少。

但更多的信息性需要更多的计算时间，可能抵消减少搜索空间所带来的益处。

A*搜索算法性能分析

◆ A*算法的完备性：如果在起始点和目标点间有路径解存在，那么一定可以得到解，如果得不到解那么一定说明没有解存在。

在一些常见的搜索问题中，状态数量是有限的，此时图搜索A*算法和排除环路的树搜索A*均是完备的，即一定能够找到一个解。

在更普遍的情况下，如果所求解问题和启发函数满足以下条件，则A*算法是完备的：

- 搜索树中分支数量是有限的，即每个节点的后继结点数量是有限的。
- 单步代价的下界是一个正数。
- 启发函数有下界。

证明见文献[4]

✓ Hart, Peter E., Nils J. Nilsson and Bertram Raphael. “A Formal Basis for the Heuristic Determination of Minimum Cost Paths.” IEEE Trans. Syst. Sci. Cybern. 4 (1968): 100-107.

A*搜索算法性能分析

◆ 树搜索A*算法的最优性

如果启发函数是可容的 (admissible)，即 $h(n) \le h^*(n)$，那么树搜索的A*算法满足最优性。

证明：启发函数满足可容性时，假设树搜索的A*算法找到的第一个终止结点为n。对于此时边缘集合中任意结点n'，根据算法每次扩展时的策略，即选择评价函数取值最小的边缘节点，有$f(n) \le f(n')$.

由于A*算法对评价函数定义为$f(n) = g(n) + h(n)$, 且$h(n) = 0$ (终止结点),
有$f(n) = g(n) + h(n) = g(n)$，$f(n') = g(n') + h(n')$,
综合可得 $g(n) \le g(n') + h(n') \le g(n') + h^*(n')$ (后续添加结点的最小代价).

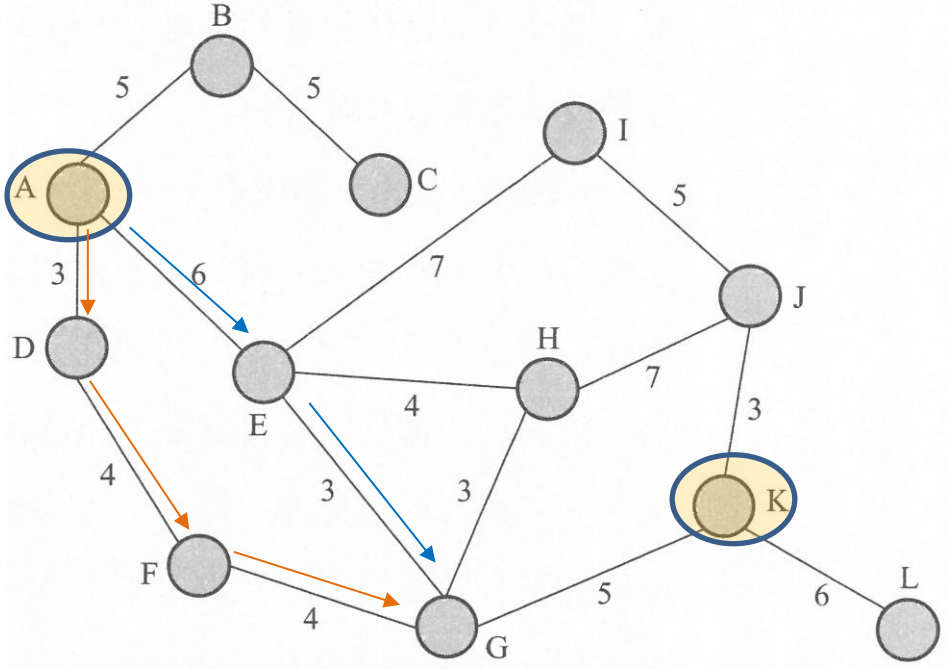
可容性 (admissible)

此时扩展其他任何一个边缘结点都不可能找到比结点n所对应路径代价更小的路径，因此结点n对应的是一条最短路径，即算法满足最优性。

A*搜索算法性能分析

◆ 图搜索A*算法的最优性

如果A*算法采用的是图搜索策略，那么即使启发函数满足可容性条件，也不能保证算法最优性。



例子：
如果有多个结点对应同一个状态（即存在多条到达同一个城市的路径），图搜索算法只会扩展其中的一个结点，而剪枝其余的结点，然而最短路径有可能经过其中某些被剪枝结点，导致算法无法找到这些最短路径。

每个状态（城市）所对应启发函数取值

状态	A	B	C	D	E	F	G	H	I	J	K	L
$h(n)$	13	10	6	12→0	7	8→0	5→0	3	6	3	0	6

此时启发函数仍然满足可容性，但图搜索的A*算法会优先扩展结点A → D → F → G，而放弃最短路径经过的节点A → E → G。

A*搜索算法性能分析

图搜索A*算法满足最优性（方法1）

修改图搜索的A*算法优先扩展结点A → D → F → G，而放弃最短路径经过的结点A → E → G。当算法从不同路径扩展同一结点时，不保留最先探索的那条路径，而是保留代价最小的那条路径。在实际操作中，算法通过为每个状态（城市）记录一个前驱状态（城市），来记录当前到达每个状态（城市）的最短路径。在处理通向相同状态的不同路径时，算法会更新当前的前驱状态。

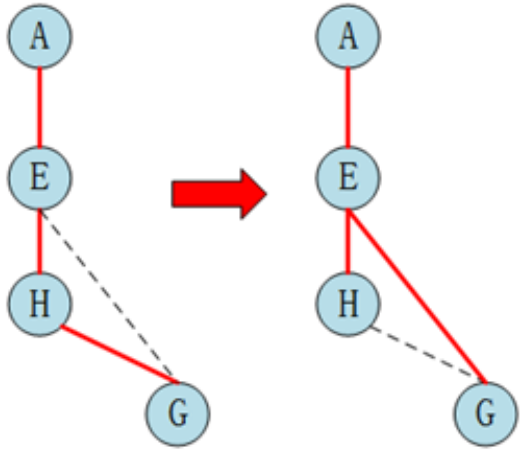


图3.7 修改后图搜索A*算法扩展A→E→G结点，红色实线表示当前搜索树中的边，虚线表示不在搜索树中的边

A*搜索算法性能分析

图搜索A*算法满足最优性（方法2）：

要求启发函数满足一致性

引理3.1：启发函数满足一致性条件时，给定一个从搜索树中得到的结点序列，每个结点是前一个结点的后继，那么评价函数对这个序列中的结点取值按照结点出现的先后次序而依次单调非减。

引理3.2：假设状态t加入搜索树时对应的结点为n，在结点n被加入搜索树后，若对应状态t的任何结点n'出现在边缘集合中，那么必然有$g(n) \leq g(n')$。

算法的最优性：对于任意一个状态t，它第一次被加入搜索树时的路径必然是最短路径。

如果对启发函数$h(n)$加上一致性的限制，就可以总从祖先状态沿着最佳路径到达任一状态。

A*搜索算法性能分析

引理3.1：启发函数满足一致性条件时，给定一个从搜索树中得到的结点序列，每个结点是前一个结点的后继，那么评价函数对这个序列中的节点取值按照节点出现的先后次序而依次单调非减。

证明：考虑序列中结点n，对结点n采取动作a而得到的后继节点n'。由于启发函数满足一致性条件，因此有 $h(n) \leq c(n,a,n')+h(n')$。于是有：

根据定义得到

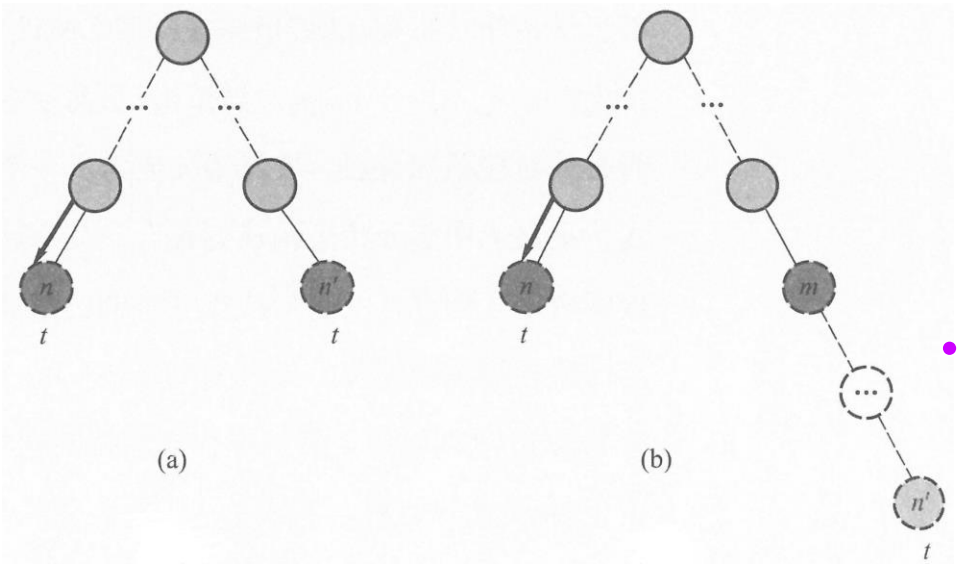
$$f(n') = g(n')+h(n') = g(n)+c(n,a,n')+h(n') \geq g(n)+h(n) = f(n)$$

$$\Rightarrow f(n') \geq f(n)$$

A*搜索算法性能分析

引理3.2：假设状态t加入搜索树时对应的结点为n，在结点n被加入搜索树后，若对应状态t的任何结点n'出现在边缘集合中，那么必然有$g(n) \leq g(n')$。

这个性质说明，每个状态被加入搜索树时，其路径代价必然小于等于任何将来出现在边缘集合中通向该状态的路径的代价。



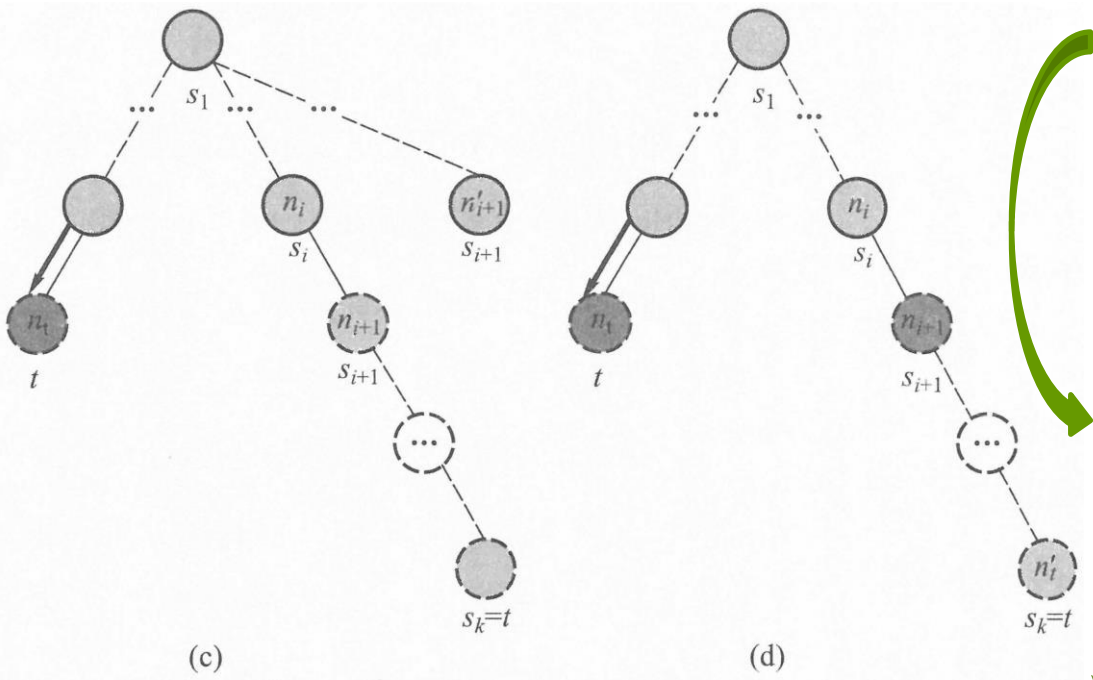
- 假设在当前搜索树中，结点n将是下一个被加入搜索树的结点，(即结点n目前是一个可扩展的边缘结点)，可证明从根结点到结点n'的路径上此时必然存在一个可扩展的边缘结点m。

- 根据A^*算法选择扩展结点的顺序，有$f(n) \leq f(m)$，又根据引理3.1，由于m和n'在同一条路径中，有$f(m) \leq f(n')$，因此$f(n) \leq f(n')$。因为n和n'对应同一个状态t，有$h(n) = h(n')$，因此得出结论$g(n) \leq g(n')$。

A*搜索算法性能分析

算法最优性：

对于任意一个状态t，它第一次被加入搜索树时的路径必然是最短路径。



- 假设t第一次被加入搜索树时对应的结点为n_t，如果n_t对应的路径不是从初始状态到t的最短路径，那么假设另有从初始状态到t的最短路径为$P = (s_1, s_2, \ldots, s_k = t)$ ✗

- 定义集合Δ为路径P中部分状态构成的集合，其中每个状态满足：该状态在当前搜索树中(①)且搜索树中通向该状态的路径是最短路径(②)。显然根结点$s_1 \in \Delta$，状态$s_k = t$不在当前搜索树中，即$s_k \notin \Delta$

  ⇒ 存在状态$s_i \in \Delta$, 且$s_{i+1} \notin \Delta$ ✗

$s_i \in \Delta$说明在搜索树中存在一个结点n_i，其路径为通向状态s_i的最短路径。

而$s_{i+1} \notin \Delta$可能对应两种情况：

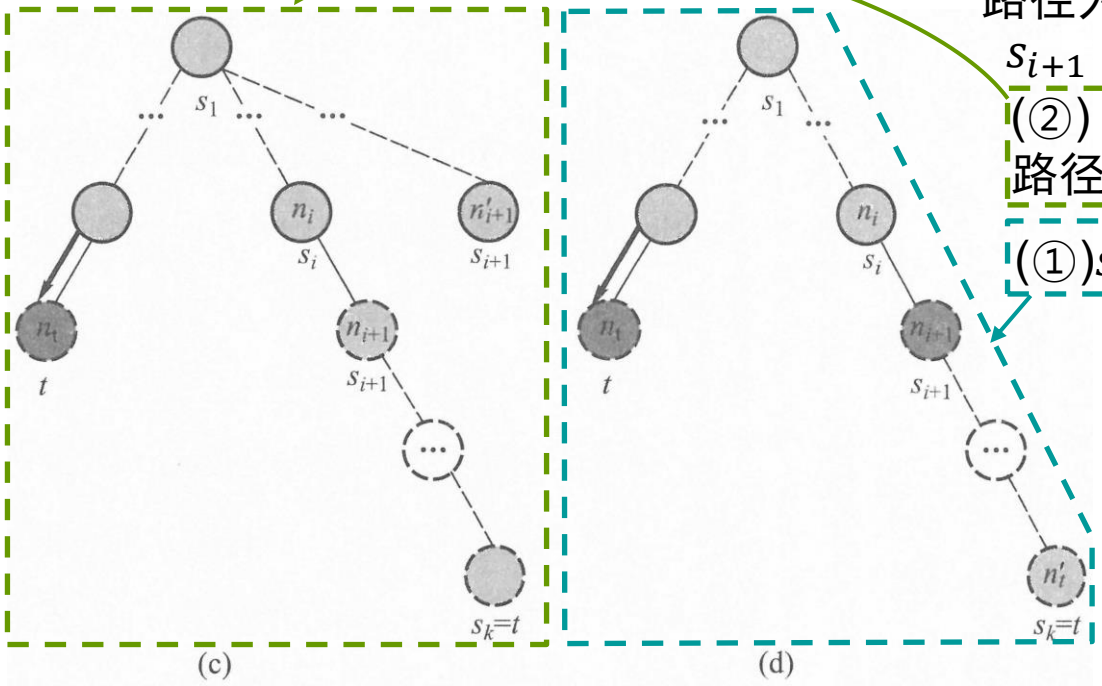
(①)s_{i+1}不在当前搜索树中；✗ 如图d 使用引理3.1 $g(n_t) = g(n'_t)$

(②) s_{i+1}在当前搜索树中，但其路径不是最短。✗ 如图c 使用引理3.2 $g(n'_{i+1}) = g(n_{i+1})$

A*搜索算法性能分析

算法最优性：

对于任意一个状态t，它第一次被加入搜索树时的路径必然是最短路径。



$s_i \in \Delta$说明在搜索树中存在一个结点n_i，其路径为通向状态s_i的最短路径。

$s_{i+1} \notin \Delta$可能对应两种情况：

(②) s_{i+1}在当前搜索树中 (结点 n'_i)，但其路径不是最短。 ✗

(①)s_{i+1}不在当前搜索树中； ✗

后继节点中状态为 s_{i+1} 的结点 n_{i+1} 不在搜索树中，且是一个边缘节点。⇒ $f(n_t) \le f(n_{i+1})$

从 n_{i+1} 开始沿着路径 $(s_{i+1}, \cdots, s_k = t)$ 到结点 n'_t 由引理3.1 $f(n_{i+1}) \le f(n'_t)$

$\Rightarrow f(n_t) \le f(n'_t) \Rightarrow g(n_t) \le g(n'_t)$

n'_t 对应到状态 t 的最短路径$\Rightarrow g(n_t) \ge g(n'_t)$

$\Rightarrow g(n_t) = g(n'_t)$

后继节点 n_{i+1} 对应通向 s_{i+1} 的更短路径(⑤)，不在搜索树中，且是一个边缘节点。

由引理3.2 $g(n'_{i+1}) \le g(n_{i+1})$

(⑤)→ $g(n'_{i+1}) \ge g(n_{i+1})$

$\Rightarrow g(n'_{i+1}) = g(n_{i+1})$

A*搜索算法性能分析

A*算法在最坏情况下扩展的结点数量为$O(b^{d+1})$，考虑到从边缘集合找出评价函数最小结点还需要额外的时间成本，其时间复杂度甚至大于广度优先搜索。

为了让A*算法发挥优势，需要设计一个好的启发函数。当树搜索算法中启发函数满足可容性时，或图搜索算法中启发函数满足一致性时，算法扩展任意结点时，该结点所对应评价函数取值必然不会超过找到最优解结点的评价函数值。设最优解的评价函数值为C^*，那么算法将剪枝评价函数值大于C^*的结点，相当于降低了分支系数b, 从而带来了更低的时间复杂度。

实际上，如果一个搜索算法剪枝任何一个评价函数值小于等于C^*的结点，那么它都可能找不到最优解，从这个意义上来说，A* 搜索是在已知信息下同类搜索策略中最优的。

Bidirectional heuristic search

We use $f_F(n) = g_F(n) + h_F(n)$ for nodes going in the forward direction (with the initial state as root) and $f_B(n) = g_B(n) + h_B(n)$ for nodes in the backward direction (with a goal state as root).

$$lb(m,n) = \max(g_F(m) + g_B(n), f_F(m), f_B(n))$$

$$f_2(n) = \max(2g(n), g(n) + h(n))$$

The node to expand next will be the one that minimizes this f_2 value; the node can comefrom either frontier.

This f_2 function guarantees that we will never expand a node (from either frontier) with $g(n) > {}^{C^*}/_2$. We say the two halves of the search "meet in the middle" in the sense that when the two frontiers touch, no node inside of either frontier has a path cost greater than the bound ${}^{C^*}/_2$.