第二章
基础知识

搜索

搜索求解策略

- 在求解一个问题时：
  - 问题的表示，找到一个合适的表示方法
  - 选择一种相对合适的求解方法。
    - 由于绝大多数需要人工智能方法求解的问题缺乏直接求解的方法，因此，搜索不失为一种求解问题的一般方法。

- 本章概要：
  - 搜索的基本概念
  - 状态空间表示方法
  - 搜索策略
    - 盲目搜索
    - 启发式搜索

本章主要内容

- 搜索的基本概念
- 状态空间表示方法
- 盲目搜索策略
- 启发式搜索策略

搜索的基本问题与主要过程

- 搜索中需要解决的基本问题：
  （1）是否一定能找到一个解
  （2）找到的解是否是最佳解
  （3）时间与空间复杂性如何
  （4）是否终止运行或是否会陷入一个死循环
- 搜索的主要过程：
  （1）从初始或目的状态出发，并将它作为当前状态。
  （2）扫描操作算子集，将适用当前状态的一些操作算子作用于当前状态而得到新的状态，并建立指向其父结点的指针。
  （3）检查所生成的新状态是否满足结束状态，如果满足，则得到问题的一个解，并可沿着有关指针从结束状态反向到达开始状态，给出一解答路径；否则，将新状态作为当前状态，返回第(2)步再进行搜索。

搜索方向

（1）正向搜索：从初始状态出发的正向搜索

- 数据驱动——从问题给出的条件出发。

（2）逆向搜索：从目的状态出发的逆向搜索

- 目的驱动——从想达到的目的入手，看哪些操作算子能产生该目的以及应用这些操作算子产生目的时需要哪些条件。

（3）双向搜索

- 双向搜索——从开始状态出发正向搜索，同时又从目的状态出发逆向搜索，直到两条路径在中间的某处汇合为止。

搜索策略

（1）盲目搜索：在不具有对特定问题的任何有关信息的条件下，按固定的步骤（依次或随机调用操作算子）进行的搜索。

（2）启发式搜索：考虑特定问题领域可应用的知识，动态地确定调用操作算子的步骤，优先选择较适合的操作算子，尽量减少不必要的搜索，以求尽快地到达结束状态。

本章主要内容

- 搜索的基本概念
- 状态空间表示方法
- 盲目搜索策略
- 启发式搜索策略

状态空间表示法

- 状态：表示系统状态、事实等叙述型知识的一组变量或数组：

$$Q=[q_1,q_2,\cdots,q_n]$$

- 操作：表示引起状态变化的过程型知识的一组关系或函数：

$$F=\{f_1,f_2,\cdots,f_m\}$$

状态空间表示法

- 状态空间：利用状态变量和操作符号，表示系统或问题的有关知识的符号体系，状态空间是一个四元组

$$(S, O, S_0, G)$$

S：状态集合。

O：操作算子的集合。

S_0：包含问题的初始状态，是 S 的非空子集。

G：包含问题的目的状态，是若干具体状态或满足某些性质的状态信息描述。

状态空间表示法

- 例：八数码问题的状态空间

2	3	1
5		8
4	6	7

初始状态

1	2	3
8		4
7	6	5

目标状态

- 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

状态集S：所有摆法

操作算子：

- 将数字n向上移Up
- 将数字n向左移Left
- 将数字n向下移Down
- 将数字n向右移Right

状态空间表示法

- 求解路径：从 S_0 结点到 G 结点的路径。

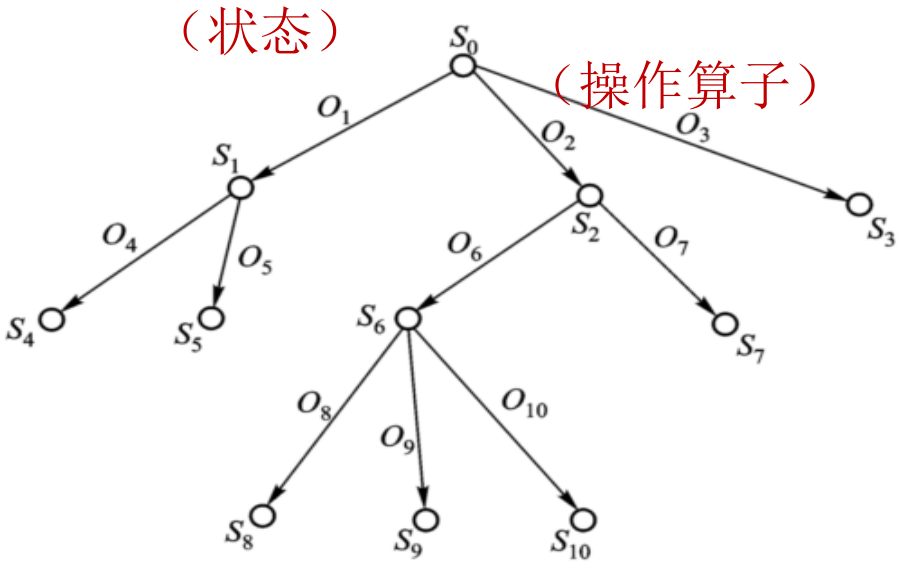
- 状态空间解：一个有限的操作算子序列。

$$S_0 \xrightarrow{O_1} S_1 \xrightarrow{O_2} S_2 \xrightarrow{O_3} \cdots \xrightarrow{O_k} G$$

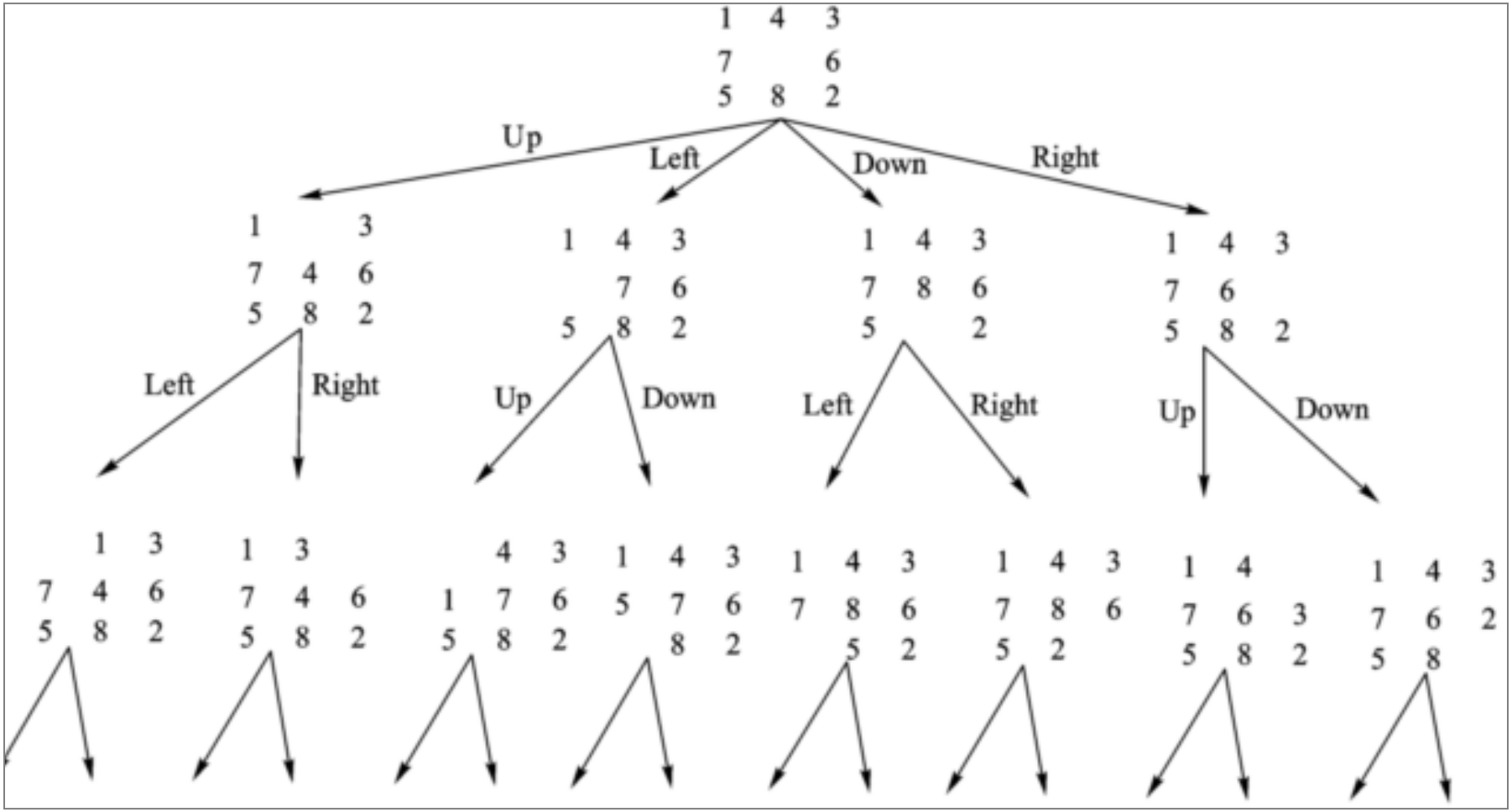
$O_1, \cdots, O_k$ ： 状态空间的一个解

状态空间的图描述

- 用有向图描述状态空间
  - 结点表示状态、边表示操作算子
  - 图中某一路径↔一种状态转换为另一种状态的操作算子序列
- 求解路径↔状态空间解



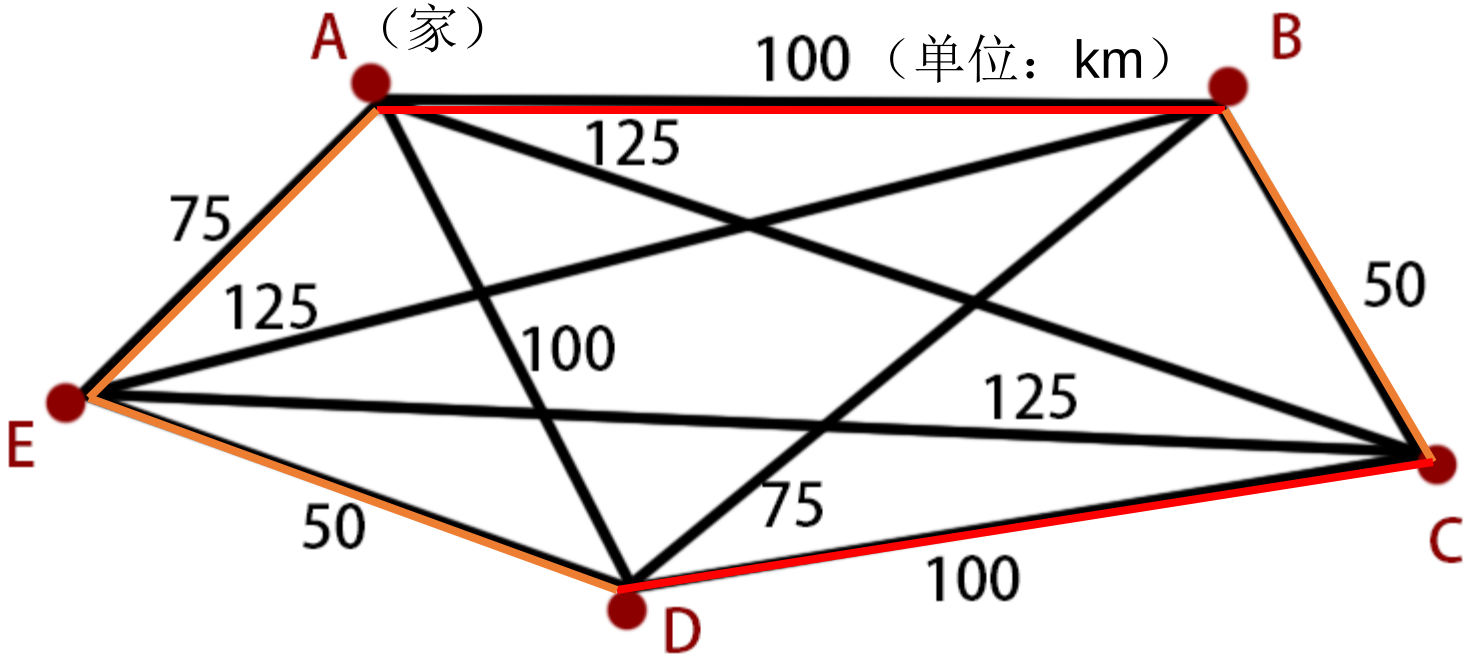
状态空间的图描述



八数码状态空间图

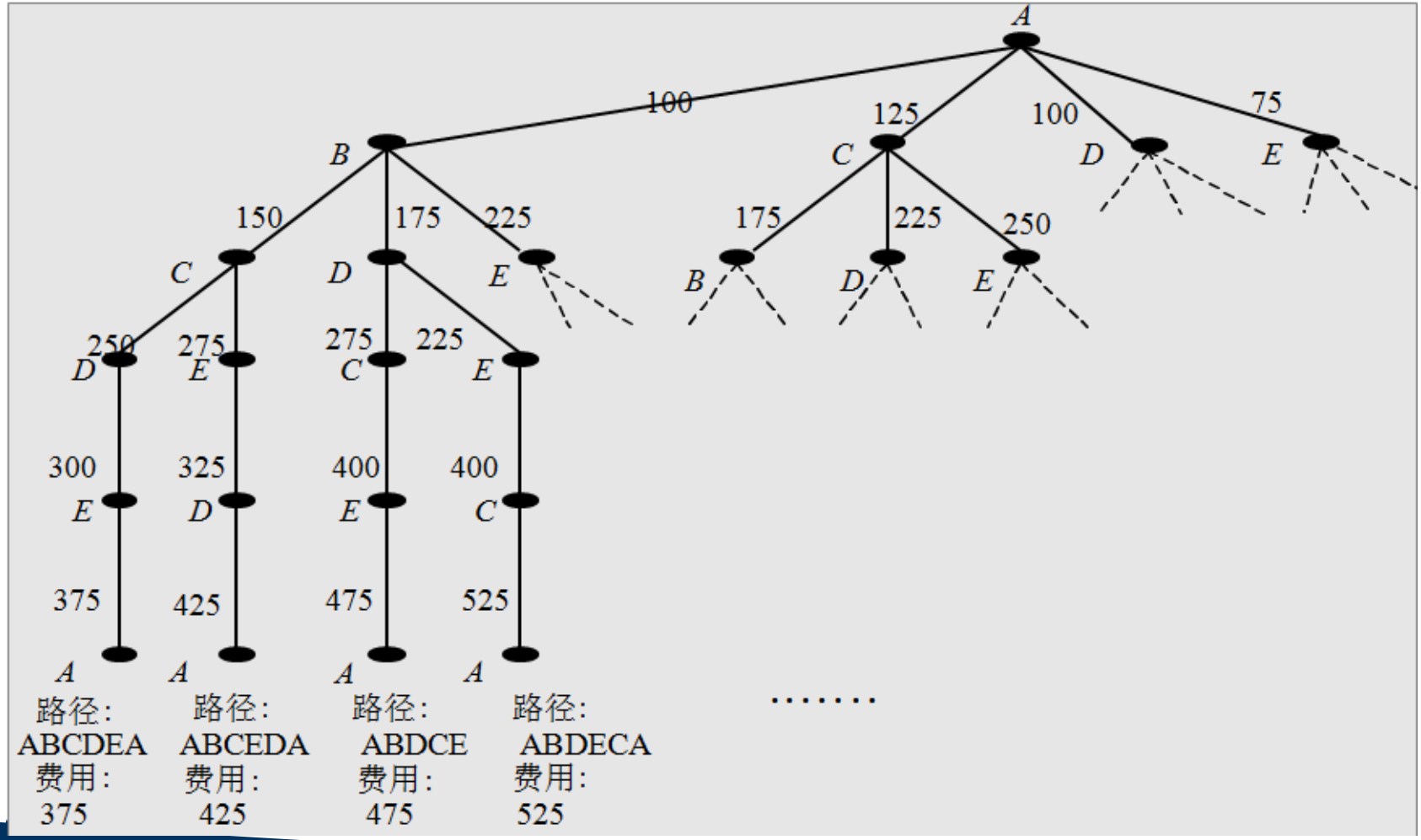
状态空间的图描述

- 例： 旅行商问题（traveling salesman problem, TSP）



可能路径：费用为375的路径（A，B，C，D，E，A）

状态空间的图描述

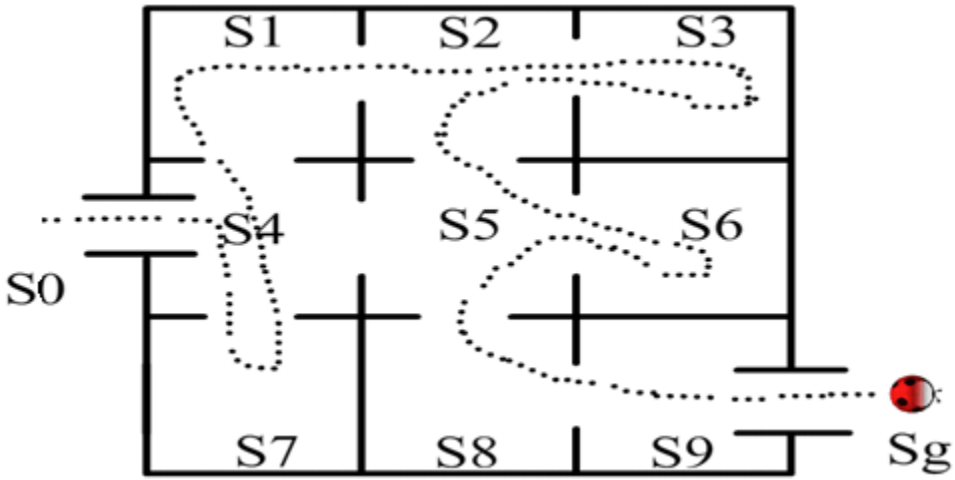

A
100
125
100
75
B
C
D
E
150
175
225
175
225
250
C
D
E
B
D
E
250
275
275
225
D
E
C
E
300
325
400
400
E
D
E
C
375
425
475
525
A
A
A
A
路径:
ABCDEA
费用:
375
路径:
ABCEDA
费用:
425
路径:
ABDCE
费用:
475
路径:
ABDECA
费用:
525
.....

本章主要内容

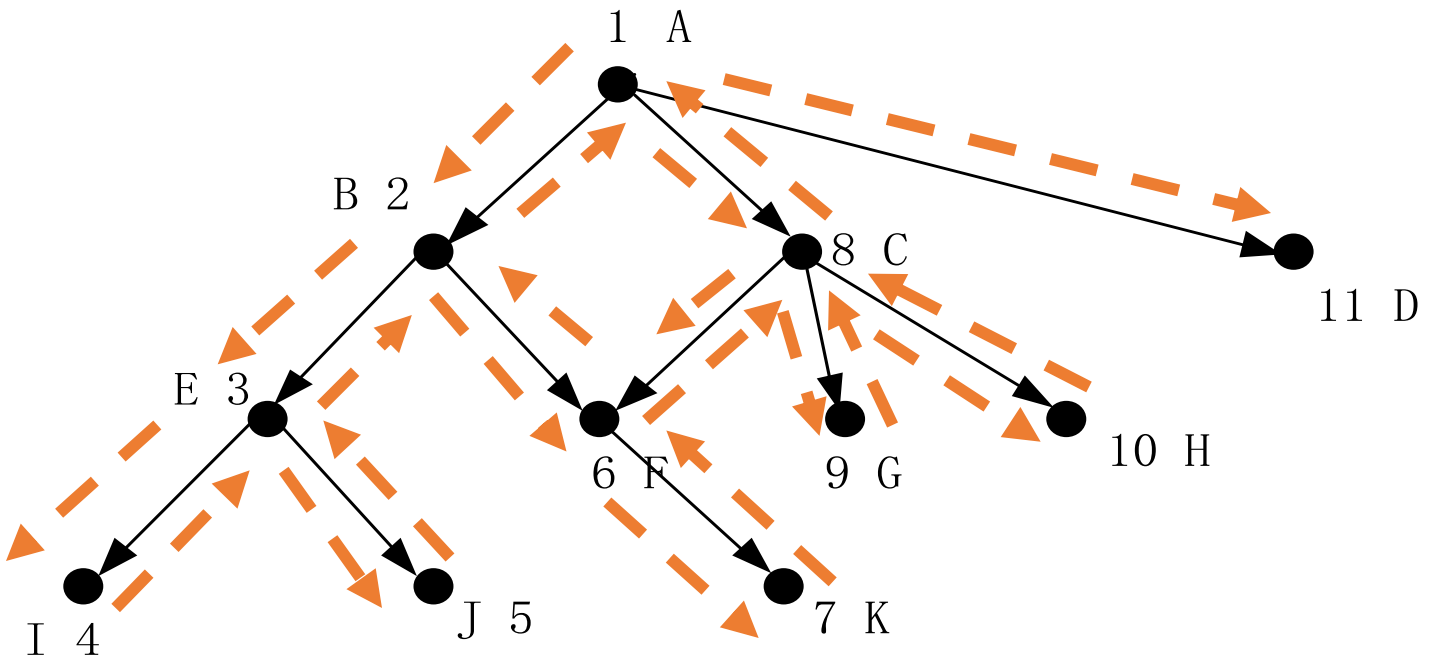
- 搜索的基本概念
- 状态空间表示方法
- 盲目搜索策略
- 启发式搜索策略

回溯策略

- 从初始状态出发，不停地、试探性地寻找路径，直到它到达目的或“不可解结点”，即“死胡同”为止。
- 若它遇到不可解结点就回溯到路径中最近的父结点上，查看该结点是否还有其他的子结点未被扩展。若有，则沿这些子结点继续搜索；如果找到目标，就成功退出搜索，返回解题路径。



回溯策略



回溯搜索示意图

回溯策略

- 回溯搜索算法用三张表保存状态空间中不同性质的结点：

(1) **PS（path states）表：**保存当前搜索路径上的状态。如果找到了目的，PS就是解路径上的状态有序集。

(2) **NPS（new path states）表：**新的路径状态表。它包含了等待搜索的状态，其后裔状态还未被搜索到，即未被生成扩展 。

(3) **NSS（no solvable states）表：**不可解状态集，列出了找不到解题路径的状态。如果在搜索中扩展出的状态是它的元素，则可立即将之排除，不必沿该状态继续搜索。

回溯策略

- 图搜索算法（深度优先、宽度优先、最佳优先搜索等）的回溯思想：

（1）用未处理状态表（NPS）使算法能返回（回溯）到其中任一状态。

（2）用一张“死胡同”状态表（NSS）来避免算法重新搜索无解的路径。

（3）在PS 表中记录当前搜索路径的状态，当满足目的时可以将它作为结果返回。

（4）为避免陷入死循环必须对新生成的子状态进行检查，看它是否在该三张表中 。

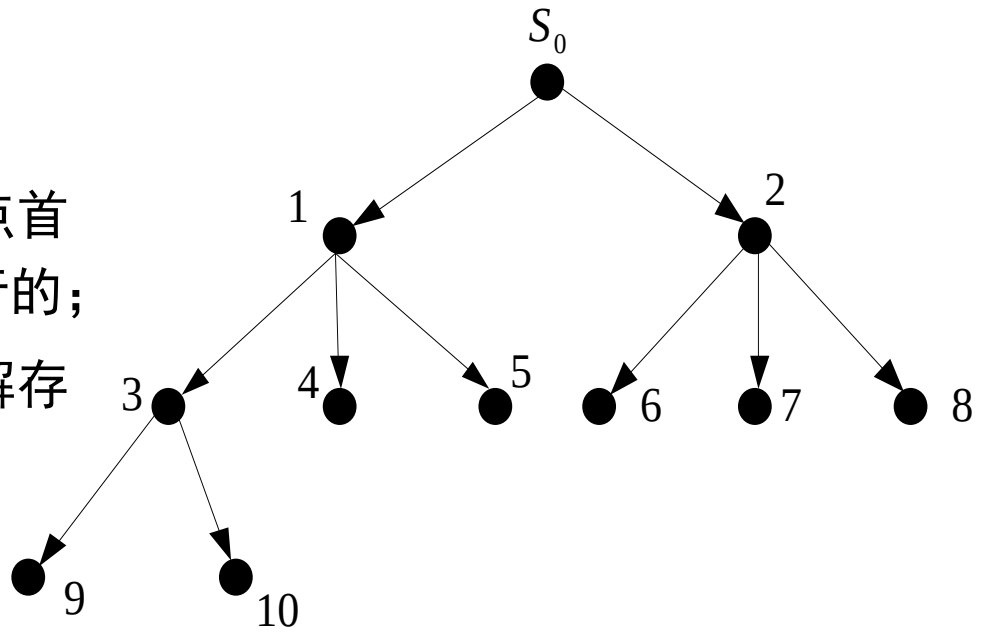
宽度优先搜索

- 宽度优先搜索（breadth-first search，广度优先搜索）：以接近起始节点的程度（深度）为依据，进行逐层扩展的节点搜索方法。

特点：

（1）每次选择深度最浅的节点首先扩展，搜索是逐层进行的；

（2）一种高价搜索，但若有解存在，则必能找到它。



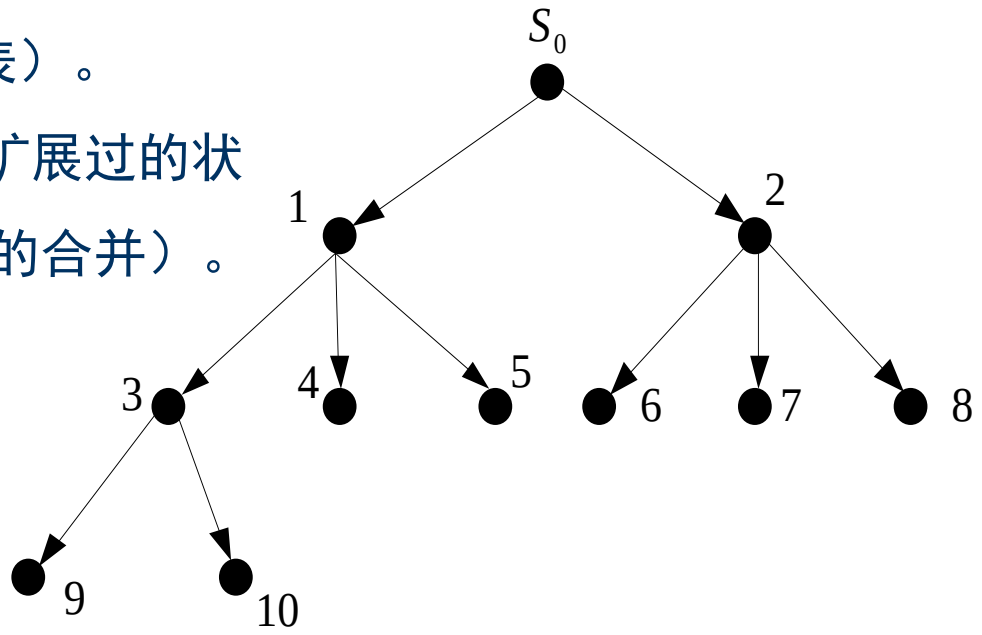
宽度优先搜索法中状态的搜索次序

宽度优先搜索

- 宽度优先搜索中保存状态空间搜索轨迹用到的表：
  - Open：已经生成出来但其子状态未被搜索的状态（类似NPS表）。
  - Closed：记录了已被生成扩展过的状态（相当于 PS表和NSS表的合并）。

Open表是一个队列：

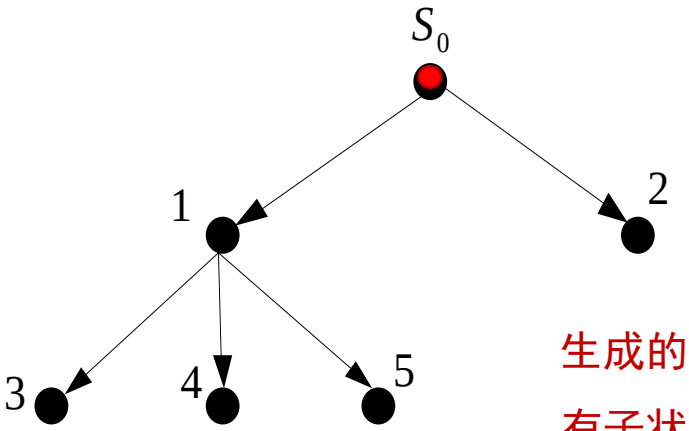
先进先出（FIFO）



宽度优先搜索法中状态的搜索次序

宽度优先搜索

- 宽度优先搜索中保存状态空间搜索轨迹用到的表：



生成的S_0所有子状态

Open

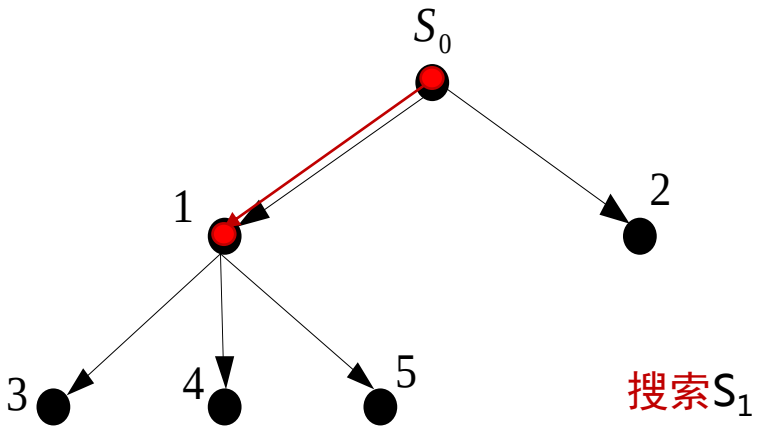
Closed

Open	Closed
~~S_0~~	S_0
S_2	

初始化

宽度优先搜索

- 宽度优先搜索中保存状态空间搜索轨迹用到的表：



搜索S_1

Open

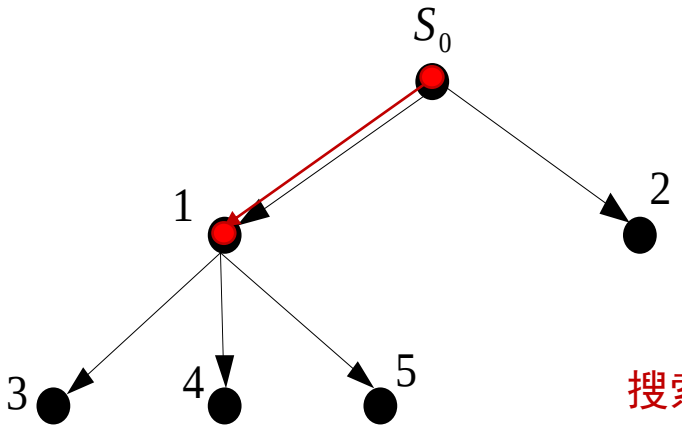
S_1
S_2

Closed

S_0

宽度优先搜索

- 宽度优先搜索中保存状态空间搜索轨迹用到的表：



搜索S_1

生成S_1所有子状态

Open

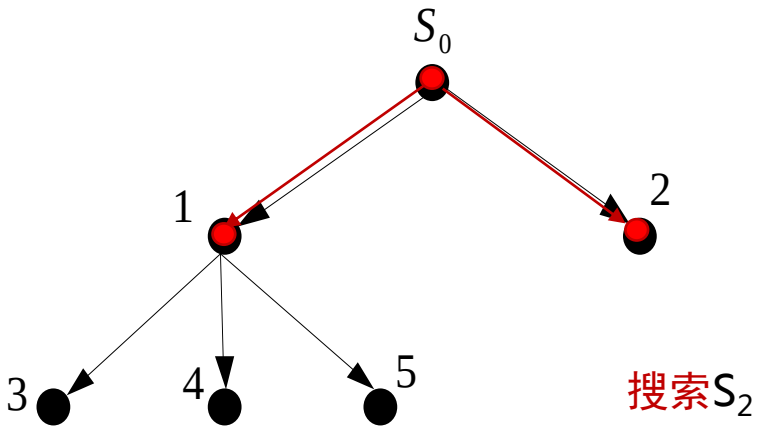
S_2

Closed

S_0
S_1

宽度优先搜索

- 宽度优先搜索中保存状态空间搜索轨迹用到的表：



搜索S_2

Open

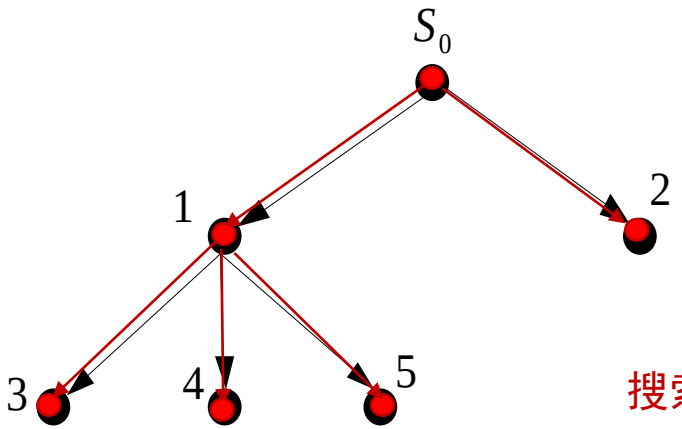
S_2
S_3
S_4
S_5

Closed

S_0
S_1

宽度优先搜索

- 宽度优先搜索中保存状态空间搜索轨迹用到的表：



搜索S_3

搜索S_4

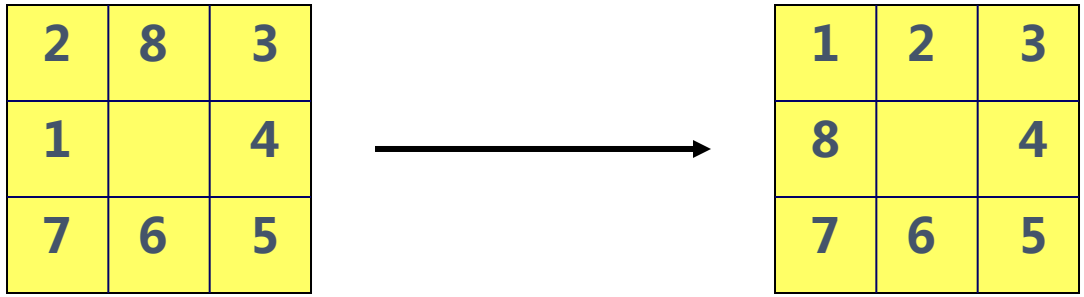
搜索S_5

Open
S_3
S_4
S_5

Closed
S_0
S_1
S_2

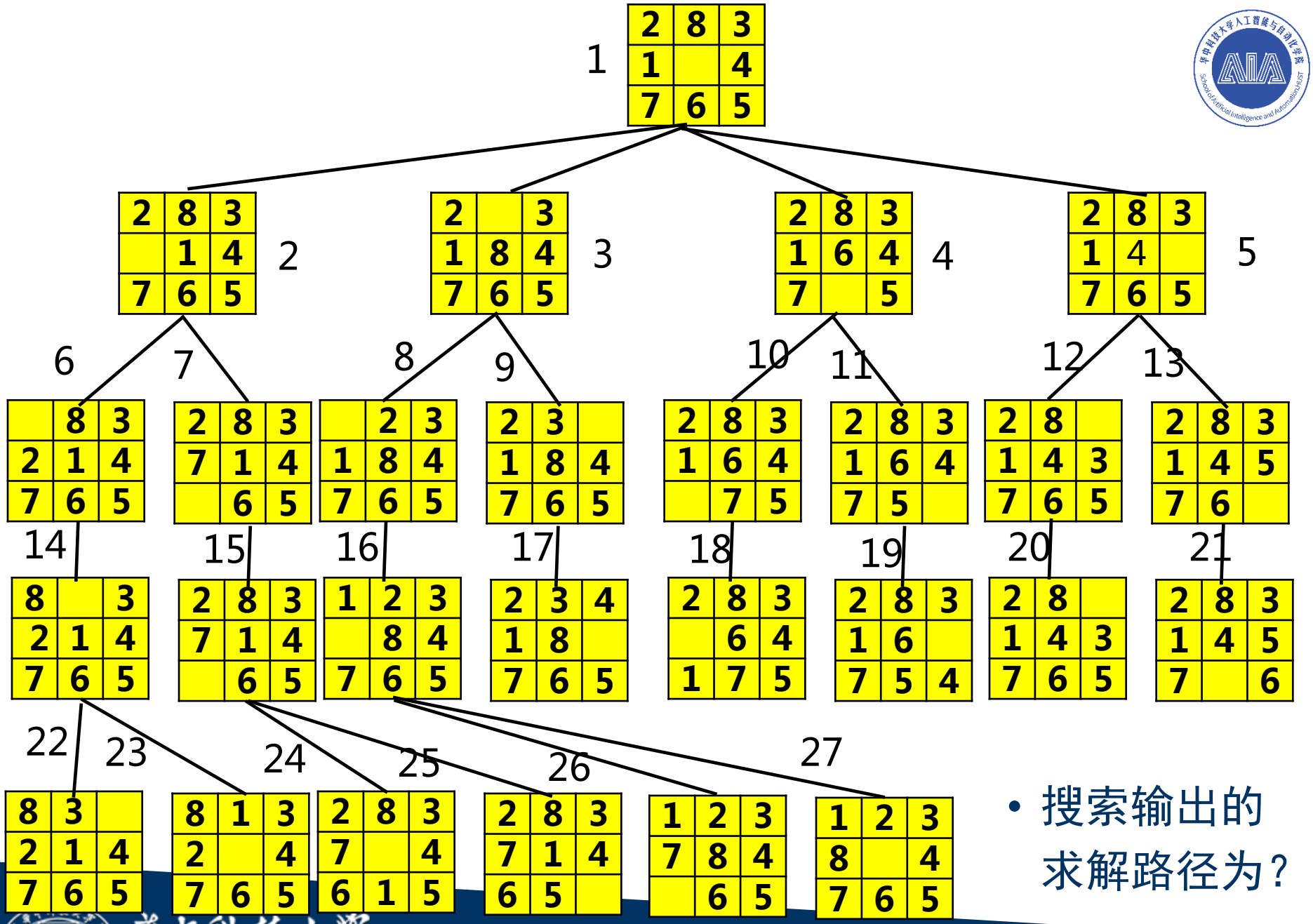
- 假设5为目标状态，则搜索输出的求解路径为：S_0->1->5

- 例子
  八数码问题（8-puzzle problem）



（初始状态）　　（目标状态）

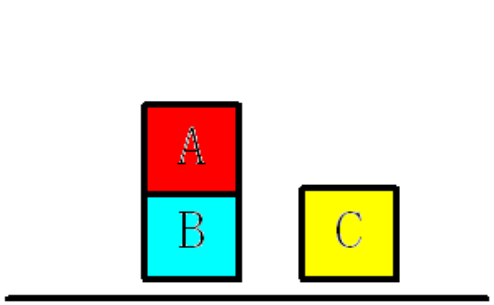
规则：
（1）不许返回先辈节点；（2）不许斜向移动；
（3）将牌移入空格的顺序为：从空格左边开始顺时针旋转。（可产生唯一的解路径，但非必须）



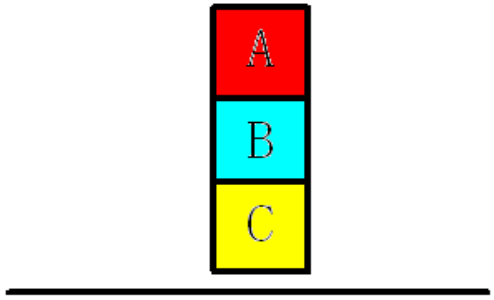
- 搜索输出的求解路径为？

宽度优先搜索

- 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

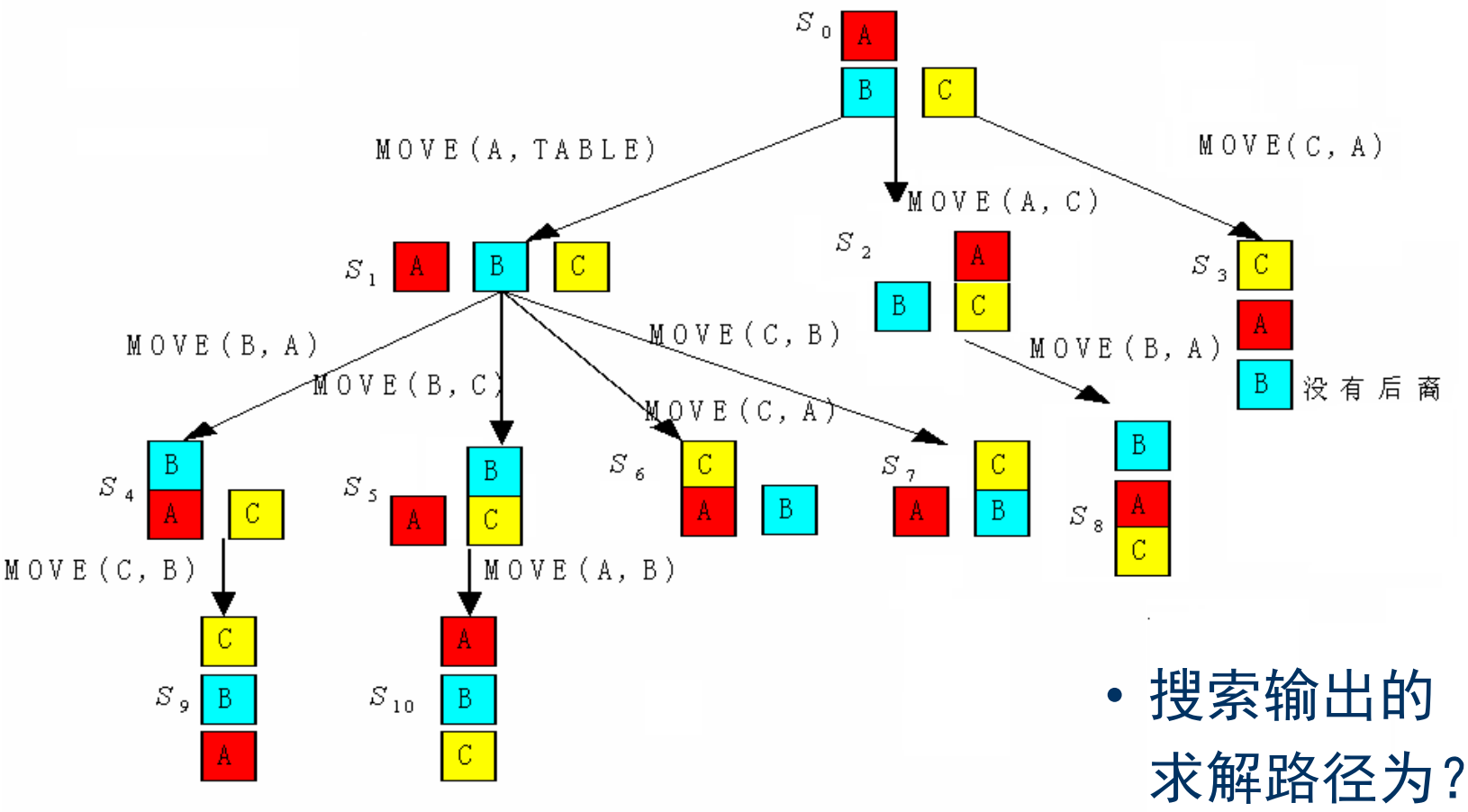


（b） 目的状态

操作算子为MOVE（X，Y）：把积木X搬到Y（积木或桌面）上面

操作算子可运用的先决条件：
（1）被搬动积木X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

宽度优先搜索



积木问题宽度优先搜索树

- 搜索输出的求解路径为？

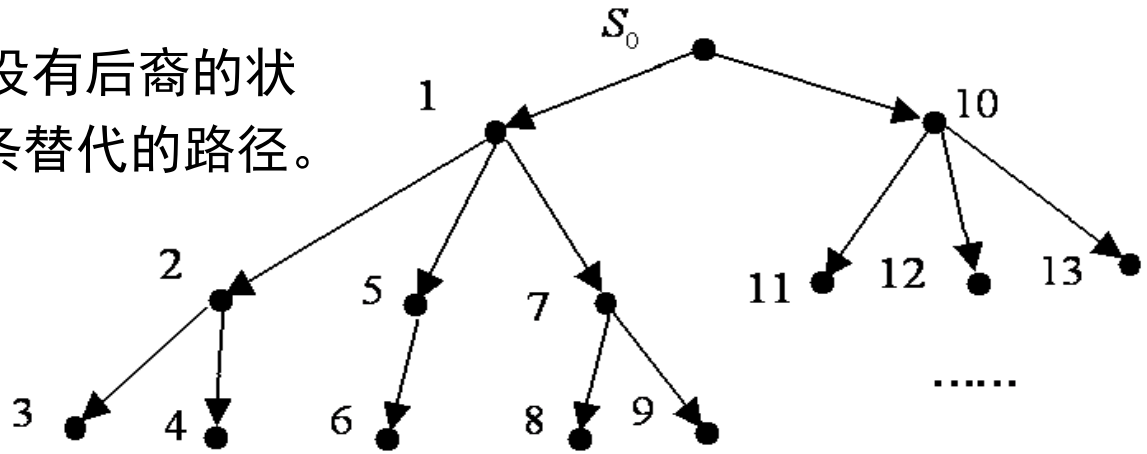
深度优先搜索

- 深度优先搜索（Depth-first Search）：首先扩展最新产生的节点，深度相等的节点按生成次序的盲目搜索。

特点：

（1）扩展最深的节点使得搜索沿着状态空间某条单一的路径从起始节点向下进行；

（2）仅当搜索到达一个没有后裔的状态时，才考虑另一条替代的路径。



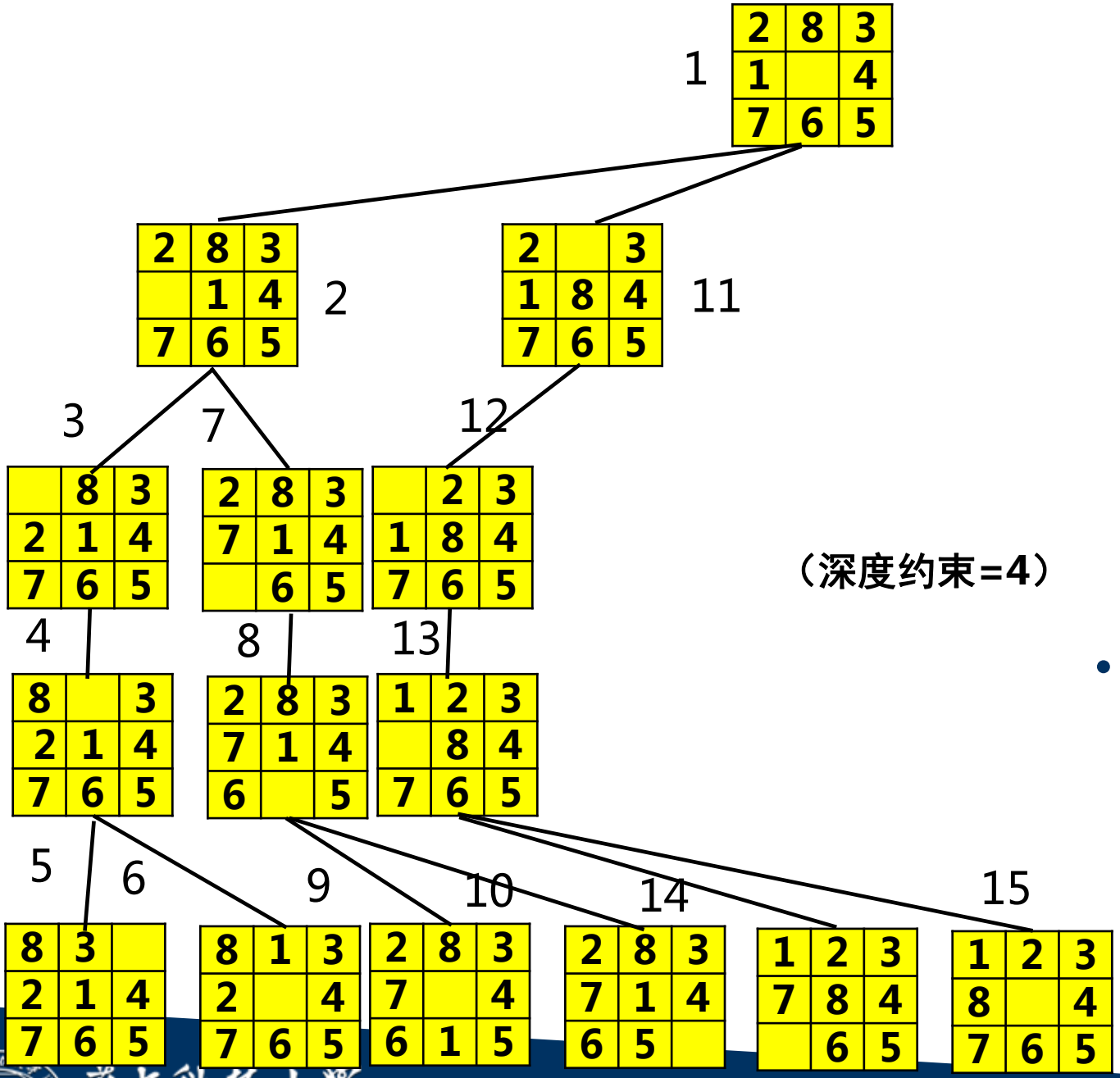
深度优先搜索

算法：

- 防止搜索过程沿着无益的路径扩展下去，往往给出一个节点扩展的最大深度——深度界限；
- 与宽度优先搜索算法最根本的不同：将扩展的后继节点放在OPEN表的前端；
- 深度优先搜索算法的Open表是堆栈，先进后出（FILO）。

深度优先搜索

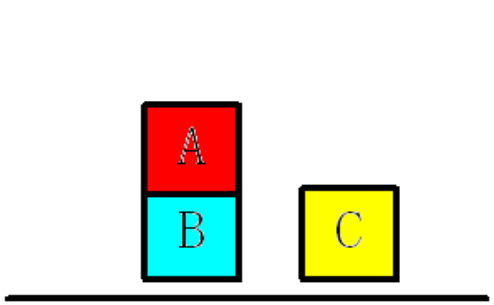
- 在深度优先搜索中，当搜索到某一个状态时，它所有的子状态以及子状态的后裔状态都必须先于该状态的兄弟状态被搜索。
- 为了保证找到解，应选择合适的深度限制值，或采取不断加大深度限制值的办法，反复搜索，直到找到解。
- 深度优先搜索并不能保证第一次搜索到的某个状态时的路径是到这个状态的最短路径。
- 对任何状态而言，以后的搜索有可能找到另一条通向它的路径。如果路径的长度对解题很关键的话，当算法多次搜索到同一个状态时，它应该保留最短路径。



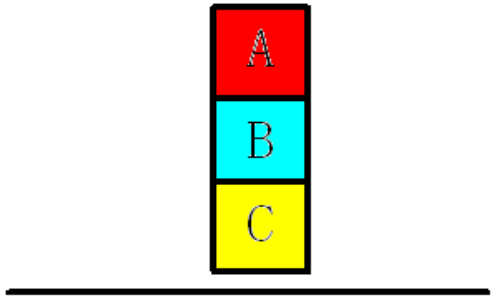
- 搜索输出的求解路径为？

深度优先搜索

- 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

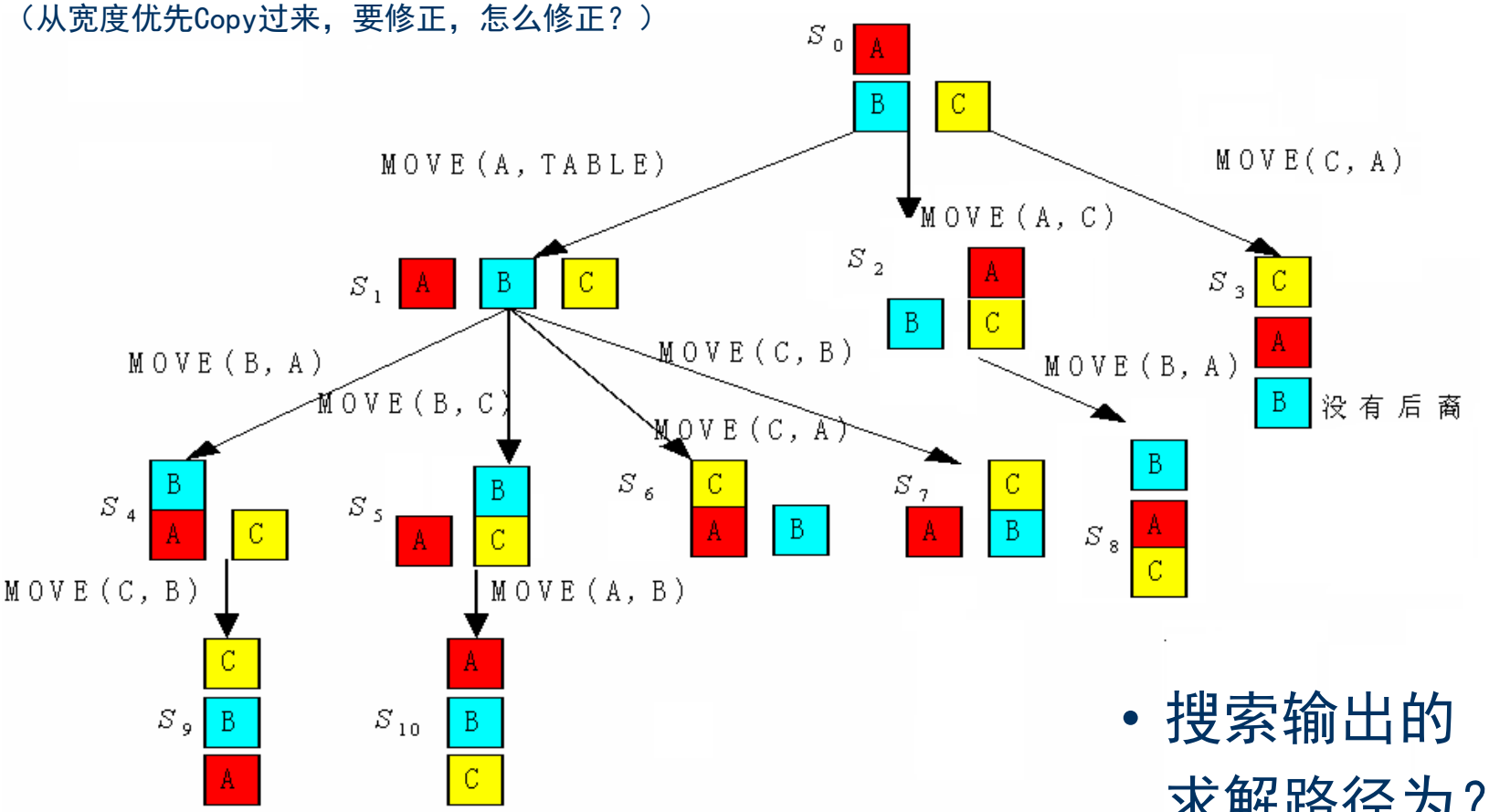
操作算子为MOVE（X，Y）：把积木X搬到Y（积木或桌面）上面

操作算子可运用的先决条件：
（1）被搬动积木X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

思考？

深度优先搜索

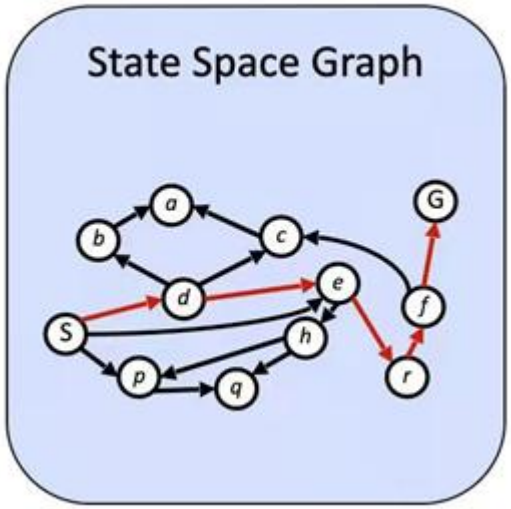
（从宽度优先Copy过来，要修正，怎么修正？）

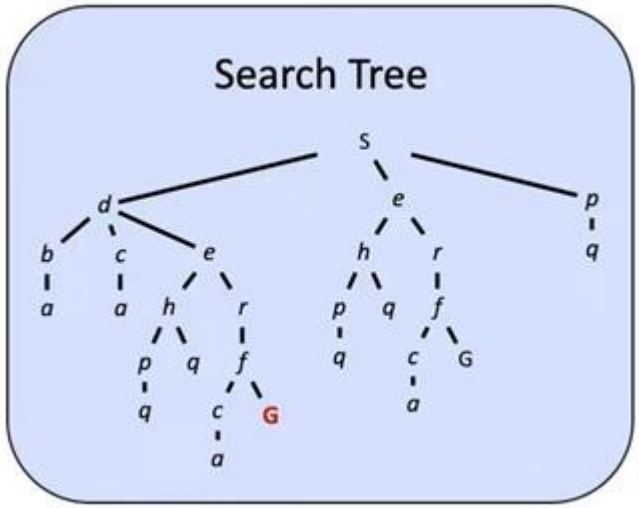


积木问题深度优先搜索树

- 搜索输出的求解路径为？

补充1：状态空间图vs搜索树



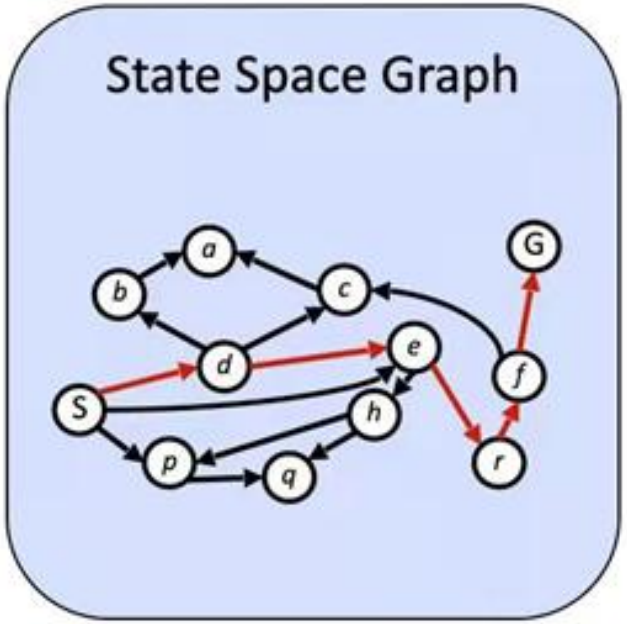


状态空间图

- 状态空间图: 一种有向图，搜索问题的数学表达
  - 节点 ： 表示状态
  - 边：表示状态转移的动作
  - 每个状态只出现一次
  - 图中寻找某一路径↔寻找操作算子序列
  - 每个节点可由多个父节点
  - 从一个节点到另外一个节点的路径不唯一
  - 实际上很难完整画出状态空间图

    （1）占用过多存储空间

    （2）思想非常有用！

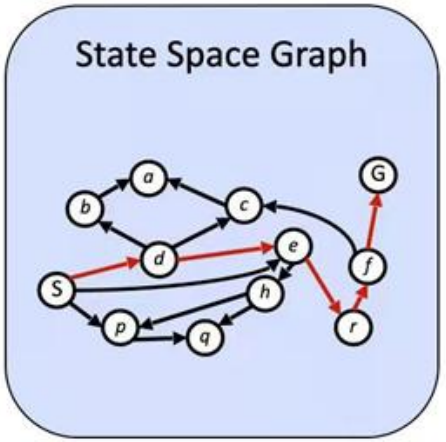


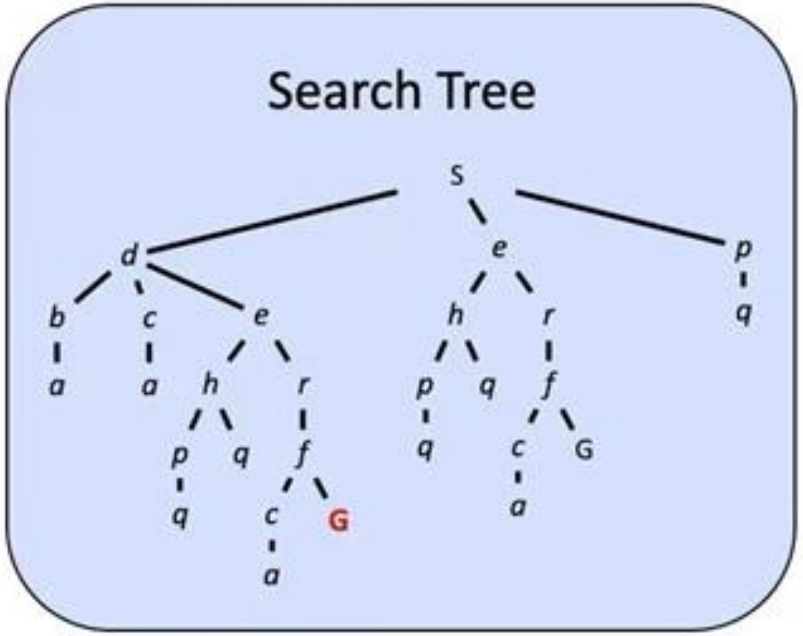
搜索树

- 一种特殊的有向图
  - 分层的模型结构
  - 结点：表示状态
  - 边：表示表示状态转移的动作
  - 初始状态为根结点
  - 同一状态可出现在多个节点
  - 每个节点，只有一个唯一的父节点
  - 从一个节点到另外一个节点的路径唯一
  - 实际上很难完整画出状态空间图

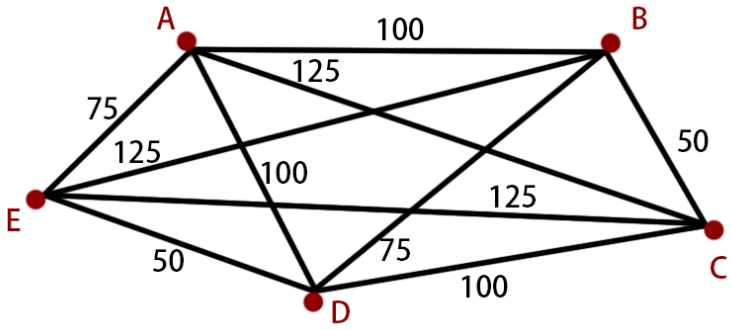
    （1）占用更多存储空间

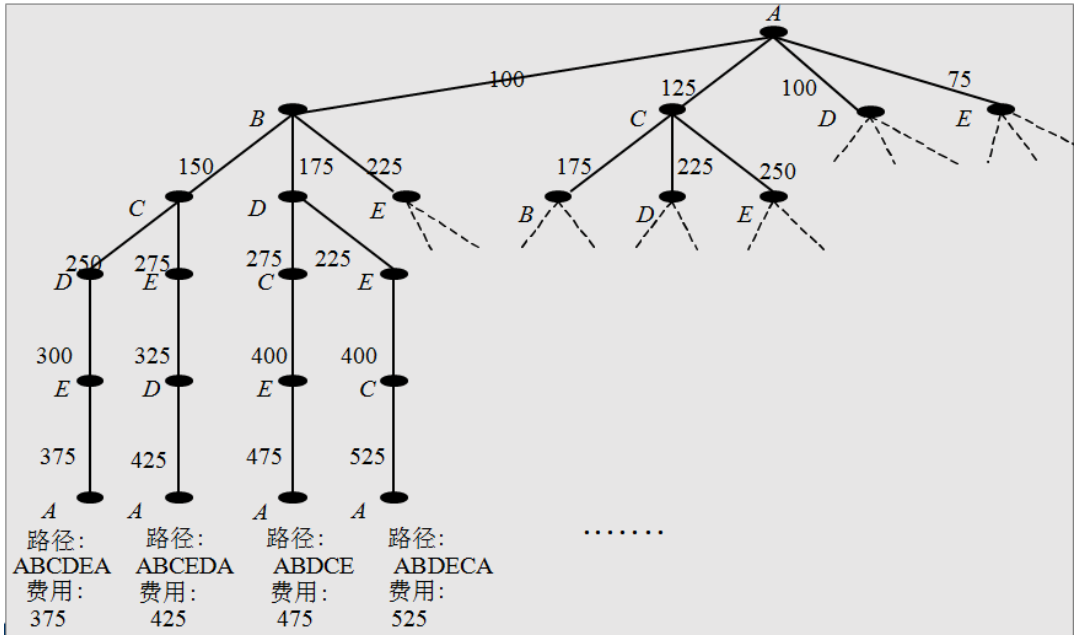
    （2）思想非常有用！





例子：旅行商问题


A
B
C
D
E
100
125
75
125
100
50
125
50
75
100


A
100
125
100
75
B
C
D
E
150
175
225
175
225
250
C
D
E
B
D
E
250
275
275
225
D
E
C
E
300
325
400
400
E
D
E
C
375
425
475
525
A
A
A
A
路径:
ABCDEA
费用:
375
路径:
ABCEDA
费用:
425
路径:
ABDCE
费用:
475
路径:
ABDECA
费用:
525
.....

补充2：生成&测试范式

```
{While no solution is found and more candidates remain
      [Generate a possible solution
      Test if all problem constraints are satisfied]
End While}
If a solution has been found, announce success and output it.
Else announce no solution found.
```

本章主要内容

- 搜索的基本概念
- 状态空间表示方法
- 盲目搜索策略
- 启发式搜索策略

启发式策略

- 启发式信息：用来简化搜索过程的有关具体问题领域的特性信息叫做启发信息；
- 启发式搜索：利用启发信息的搜索方法
  - 特点：重排OPEN表，选择最有希望的节点加以扩展
- 运用启发式策略的两种基本情况：

（1）无法知道问题的全部状态空间，只能依赖部分状态空间和有关经验信息来求解其中的问题；

（2）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，利用启发式信息进行更高效的搜索。譬如围棋的状态空间数为10^{761}，无法用前面的盲目搜索算法进行搜索。

启发信息和估价函数

- 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，使问题的状态空间缩小；

  （2）过程性启发信息：用于构造操作算子，使操作算子少而精；

  （3）控制性启发信息：表示控制策略的知识，包括协调整个问题求解过程中所使用的各种处理方法、搜索策略、控制结构等。

- 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，以免盲目选择扩展节点；

  （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。

启发信息和估价函数

- 估价函数（evaluation function）：估算节点“希望”程度的量度。
- 估价函数值 $f(n)$：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。

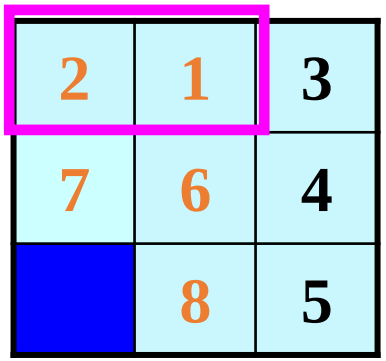
$h(n)$比重大：降低搜索工作量，但可能导致找不到最优解；

$h(n)$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。

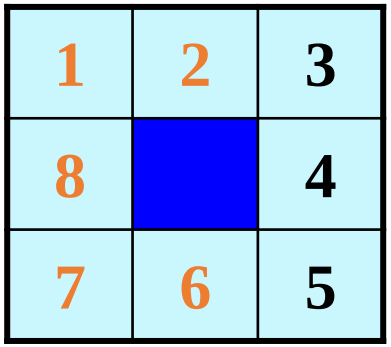
启发信息和估价函数

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（不含空格），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$h(S_0) = 8$。



初始棋局



目标棋局

最佳优先搜索

- 最佳优先搜索算法（Best-First-Search）是一种启发式搜索算法，基于宽度优先搜索，用启发估价函数对将要被遍历到的点进行估价，然后选择代价小的进行遍历，直到找到目标节点或者遍历完所有点，算法结束。
- BFS算法不能保证找到的路径是一条最短路径。

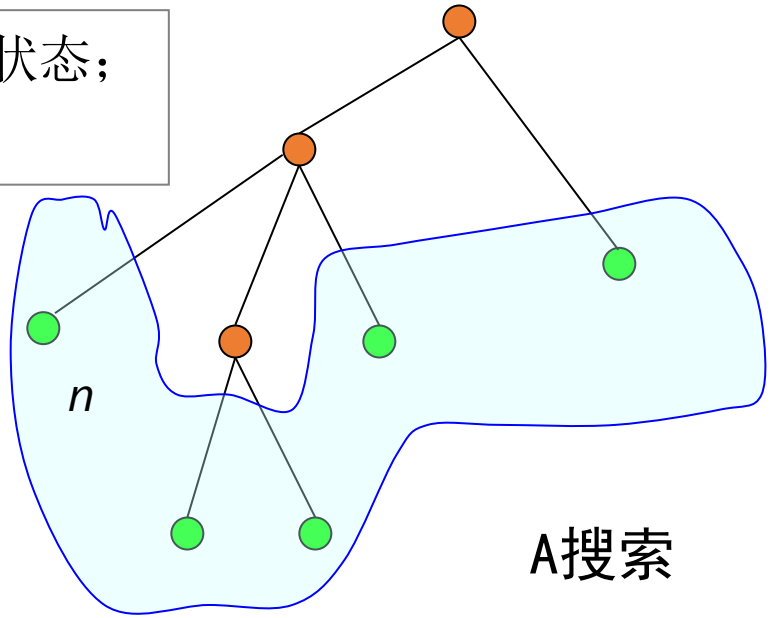
A搜索

- 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。

- 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。

$g(n)$：状态n的实际代价，例如搜索的深度；

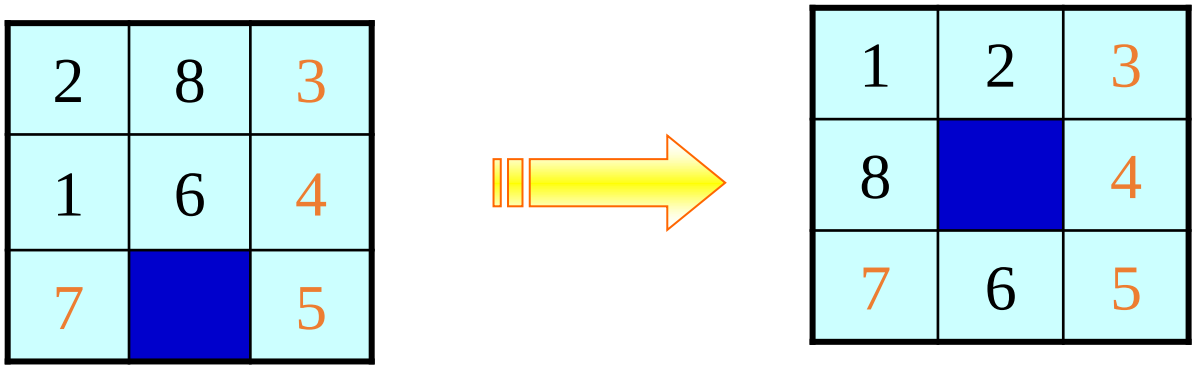
$h(n)$：对状态n与目标“接近程度”的某种启发式估计。



A搜索

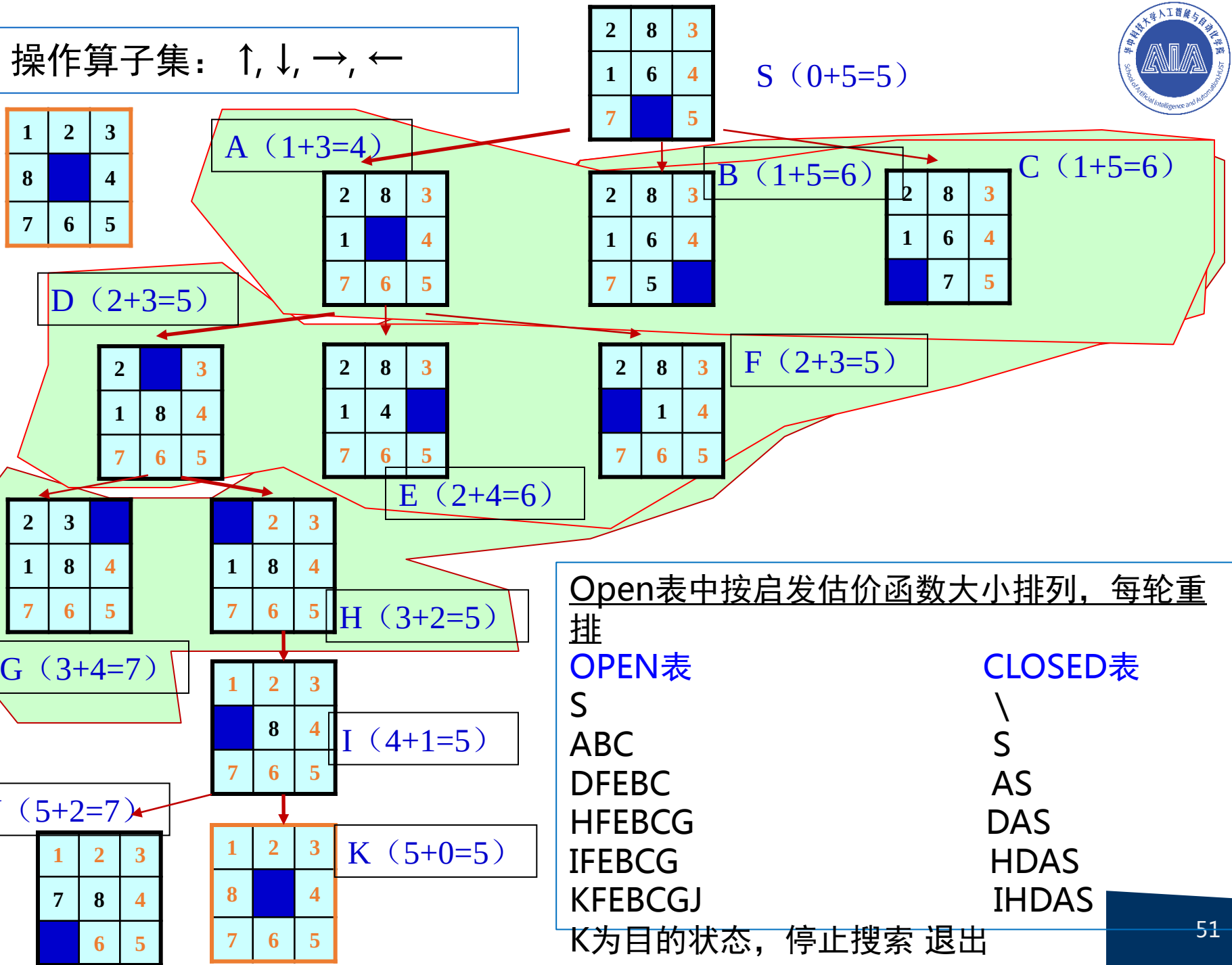
例：利用A搜索算法求解八数码问题，问最少移动多少次就可达到目标状态？

- 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（不包括空格），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：↑, ↓, →, ←



Open表中按启发估价函数大小排列，每轮重排

OPEN表	CLOSED表
S	\
ABC	S
DFEBC	AS
HFEBCG	DAS
IFEBCG	HDAS
KFEBCGJ	IHDAS

K为目的状态，停止搜索 退出

A搜索

- 问题：A搜索算法能不能保证找到最优解（路径最短的解）？
- 估价函数 $f(n) = g(n) + h(n)$

  $g(n)$：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

- A*搜索算法（最佳图搜索算法）：

定义$h^*(n)$为状态n到目的状态的最优路径代价，当A搜索算法的启发函数$h(n)$小于等于$h^*(n)$，即满足

$$h(n) \le h^*(n), \quad \text{对所有结点n}$$

时，被称为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。

- 上例中的八数码问题中定义的$h(n)$ 表示了“不在位”的数码数，满足上述条件，因此其A搜索树也是A*搜索树，所得解路径为最优解路径。

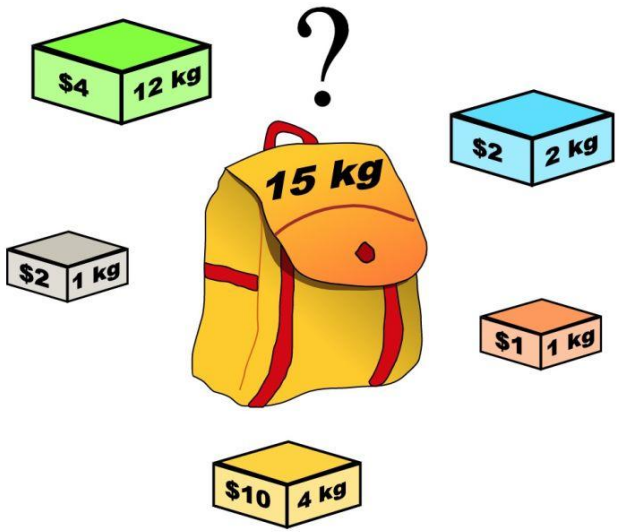
贪心算法

- 贪心（贪婪）算法：在对问题求解时，总是做出在当前看来是最好的选择。也就是说，不从整体最优上加以考虑，它所做出的仅仅是在某种意义上的局部最优解。
  - 贪心的本质是通过每一步的局部最优，期望实现全局最优的一种算法思想。
  - 贪心算法不是对所有问题都能得到整体最优解，关键是贪心策略的选择；选择的贪心策略必须具备无后效性，即某个状态以前的过程不会影响以后的状态，只与当前状态有关。

贪心算法

- 例：背包问题

问题描述：有N件物品和一个最多能被重量为W 的背包。第i件物品的重量是weight[i]，得到的价值是value[i] 。**每件物品只能用一次**，求解将哪些物品装入背包里物品价值总和最大。



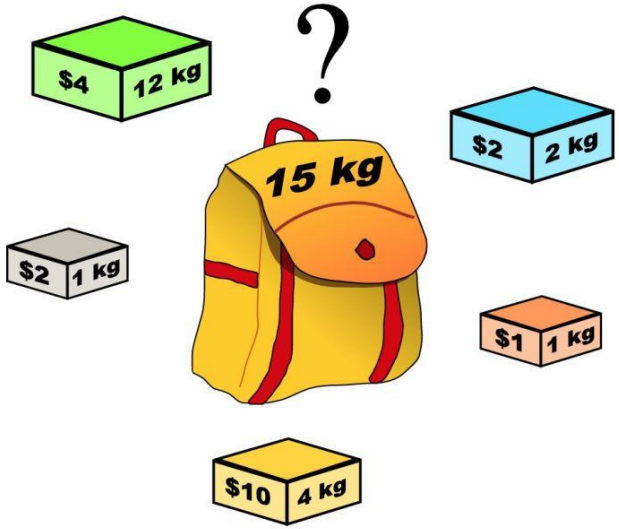
	物品1	物品2	物品3	物品4	物品5
价值$	4	2	10	1	2
重量kg	12	1	4	1	2

每一件物品其实只有两个状态，取或者不取，所以可以使用回溯法搜索出所有的情况，那么时间复杂度就是O(2^n)，这里的n表示物品数量

贪心算法

- 例：背包问题

	物品1	物品2	物品3	物品4	物品5
价值$	4	2	10	1	2
重量kg	12	1	4	1	2



贪心的三种策略：

1、从物品中选取价值最大的放入。 3 1 2 5 4

2、从物品中选区重量最小的放入。 2 4 5 3

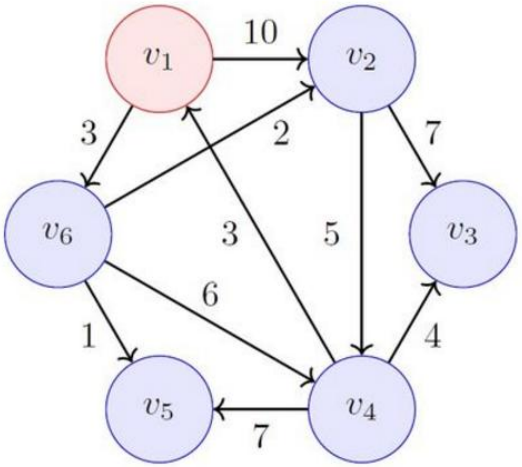
3、选取物品的价值与重量的比值大的放入。 3 2 4 5

第一第二种不能保证得到最优解，第三种也不能保证得到最优解，但是能得到最优的近似解。

Dijkstra算法

- Dijkstra算法是贪心算法的一个典型案例：

求解单源最短路径问题：在赋权有向图 G=(V, E, W) 中，假设每条边 E[i] 的长度为 w[i]，**找到由顶点 V1到其余各点的最短路径。**

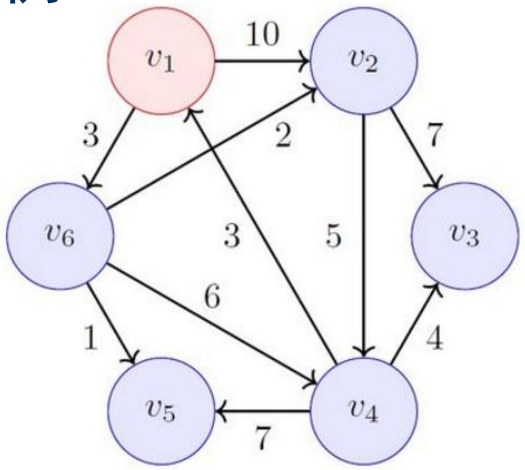


Dijkstra算法

- Dijkstra算法是贪心算法的一个典型案例：

把图中顶点集合V分成两组：

- 第一组为**已求出最短路径的顶点集合**（S）
- 第二组为其余**未确定最短路径的顶点集合**（V）



1. 初始 $S=\{v_1\}$ ；
2. 对于 $v_i \in V-S$ ，计算 $\mathrm{dist}[s,v_i]$ ；
3. 选择 $\min\limits_{v_j \in V} \mathrm{dist}[s,v_i]$ ，并将这个 v_j 放进集合 S 中，更新 $V-S$ 中的顶点的 dist 值；
4. 重复 1 ，直到 $S=V$ 。

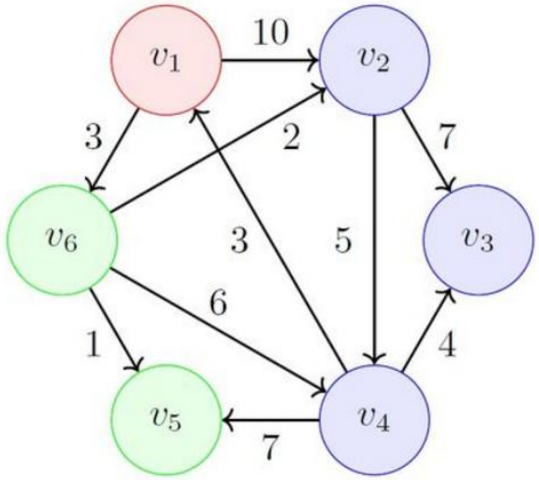
贪心

从源点到顶点的相对于集合S的最短路径

即从源点到顶点的路径中间只能经过已经包含在集合S中的顶点，不能直接到达的dist=∞

Dijkstra算法

- Dijkstra算法是贪心算法的一个典型案例：



$S = \{v_1\}$

$\text{dist}[v_1, v_2] = w_{1,2} = 10 \quad \text{dist}[v_1, v_3] = \infty$

$\text{dist}[v_1, v_4] = \infty \quad \text{dist}[v_1, v_5] = \infty$

$\text{dist}[v_1, v_6] = w_{1,6} = 3$ √ $\quad \text{dist}[v_1, v_1] = 0$

$S = \{v_1, v_6\}$

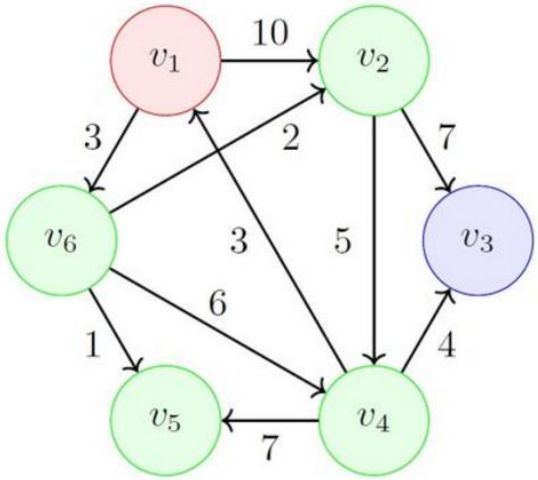
$\text{dist}[v_1, v_2] = \text{dist}[v_1, v_6] + w_{6,2} = 5 \quad \text{dist}[v_1, v_3] = \infty$

$\text{dist}[v_1, v_4] = \text{dist}[v_1, v_6] + w_{6,4} = 9 \quad \text{dist}[v_1, v_5] = \text{dist}[v_1, v_6] + w_{6,5} = 4$ √

$S = \{v_1, v_6, v_5\}$

Dijkstra算法

- Dijkstra算法是贪心算法的一个典型案例：



$$S = \{v_1, v_6, v_5\}$$

$$\text{dist}[v_1, v_2] = \text{dist}[v_1, v_6] + w_{6,2} = 5 \quad \text{dist}[v_1, v_3] = \infty$$

$$\text{dist}[v_1, v_4] = \text{dist}[v_1, v_6] + w_{6,4} = 9$$

$$S = \{v_1, v_6, v_5, v_2\}$$

$$\text{dist}[v_1, v_3] = \text{dist}[v_1, v_6] + w_{6,2} + w_{2,3} = 12$$

$$\text{dist}[v_1, v_4] = \text{dist}[v_1, v_6] + w_{6,4} = 9$$

$$S = \{v_1, v_6, v_5, v_2, v_4\}$$

$$\text{dist}[v_1, v_3] = \text{dist}[v_1, v_6] + w_{6,2} + w_{2,3} = 12 \Rightarrow S = \{v_1, v_6, v_5, v_2, v_4, v_3\}$$

Dijkstra算法

- Dijkstra算法是贪心算法的一个典型案例：

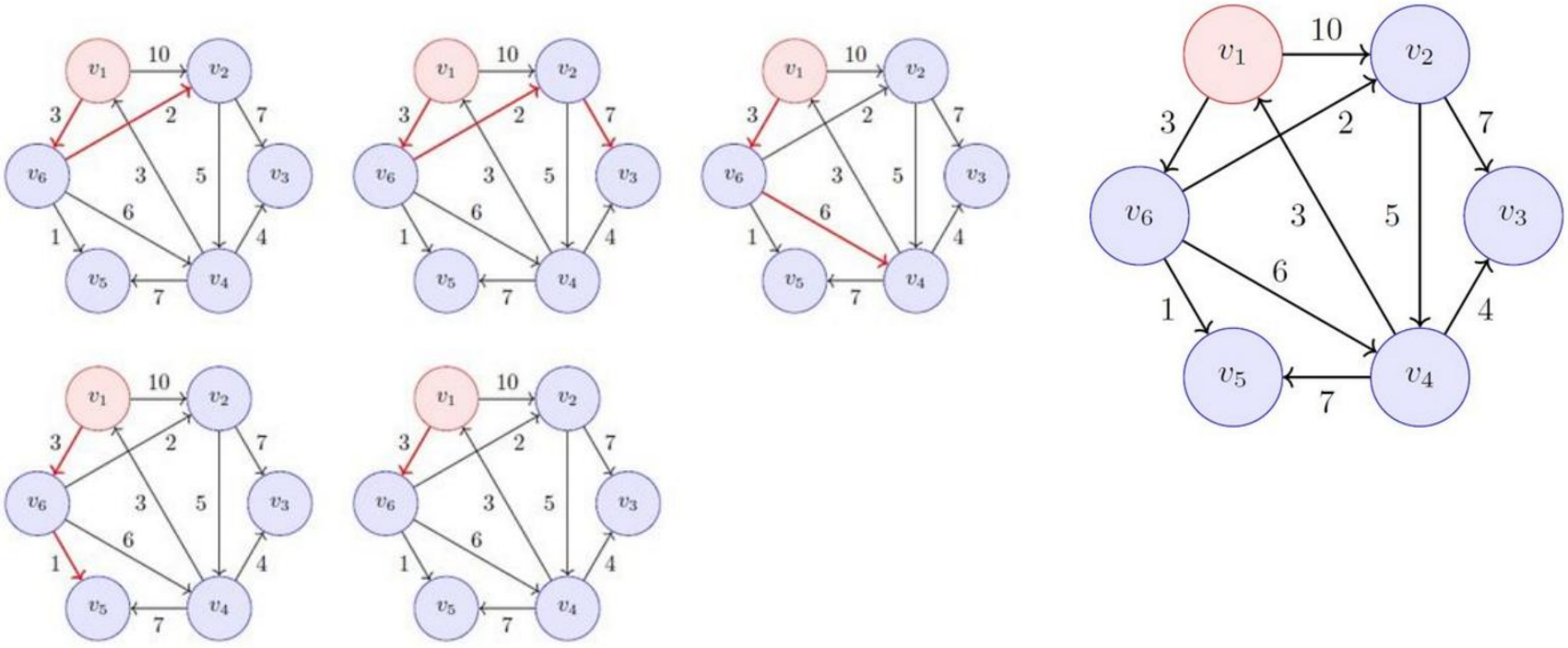
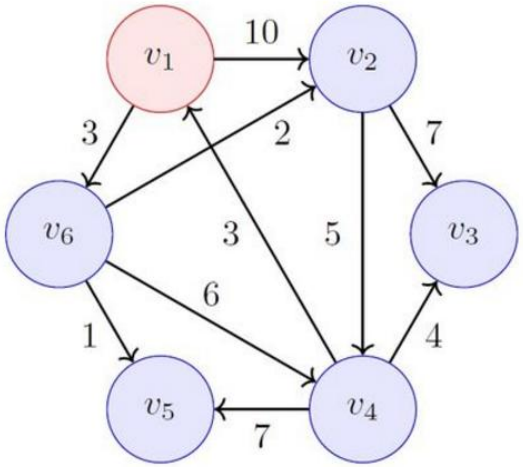
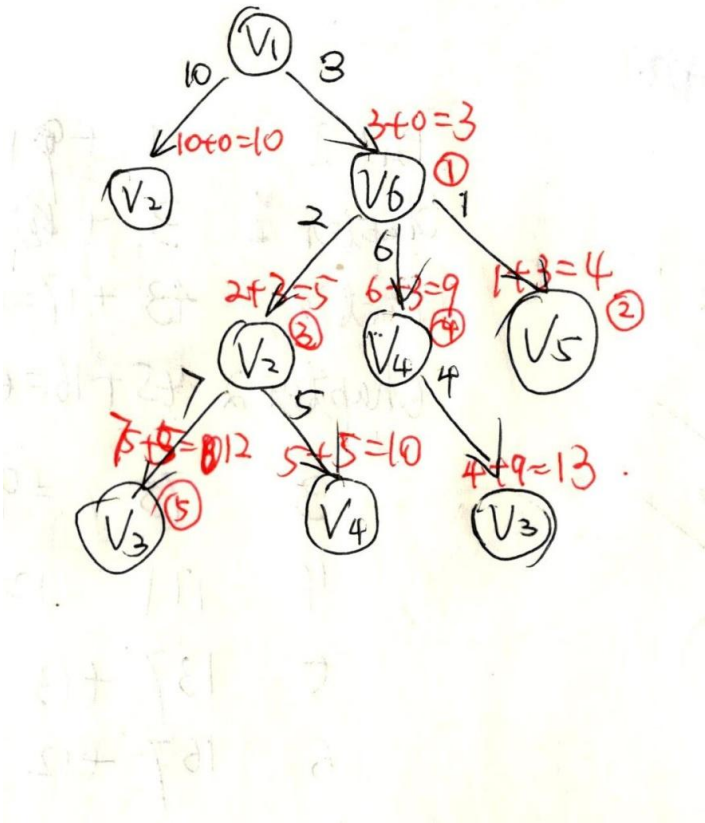


图　V中的源点s(v_1)到V中其余点的最短路径。

Dijkstra算法

- Dijkstra算法是贪心算法的一个典型案例：





$$S = \{v_1, v_6, v_5, v_2, v_4, v_3\}$$

集束搜索

- 对贪心策略简单改进。思路：稍微放宽一些考察的范围，在每一个时间步，不再只保留当前最佳的1个输出，而是保留num_beams个。当num_beams=1时，集束搜索就退化成了贪心搜索。

beam search在每一步需要考察的候选数量是贪心搜索的num_beams倍，因此是一种牺牲时间换性能的方法。