第 5 章　搜索求解策略

教材：

王万良《人工智能导论》（第5版）

高等教育出版社，2020

第5章 搜索求解策略

- 在求解一个问题时，涉及到两个方面：一是该问题的表示，如果一个问题找不到一个合适的表示方法，就谈不上对它求解。另一方面则是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，搜索为一种求解问题的一般方法。
- 下面首先讨论搜索的基本概念，然后着重介绍状态空间知识表示和搜索策略，主要有回溯策略、宽度优先搜索、深度优先搜索等盲目的图搜索策略，以及A及A*搜索算法等启发式图搜索策略。

第5章 搜索求解策略

- 5.1 搜索的概念
- 5.2 状态空间的搜索策略
- 5.3 盲目的图搜索策略
- 5.4 启发式图搜索策略

第5章 搜索求解策略

- ✓ **5.1 搜索的概念**
- **5.2 状态空间的搜索策略**
- **5.3 盲目的图搜索策略**
- **5.4 启发式图搜索策略**

5.1 搜索的概念

- 问题求解：
  - 问题的表示。
  - 求解方法。
- 问题求解的基本方法：搜索法、归约法、归结法、推理法及产生式等。

5.1.1 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。

5.1.1 搜索的基本问题与主要过程

- **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。

5.1.2 搜索策略

1. 搜索方向：

(1) **数据驱动**：从初始状态出发的正向搜索。

正向搜索——从问题给出的条件出发。

(2) **目的驱动**：从目的状态出发的逆向搜索。

逆向搜索：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：从开始状态出发作正向搜索，同时又从目的状态出发作逆向搜索，直到两条路径在中间的某处汇合为止。

5.1.2 搜索策略

- **2. 盲目搜索与启发式搜索：**

（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。

第5章 搜索求解策略

- 5.1 搜索的概念
- ✓ 5.2 状态空间的搜索策略
- 5.3 盲目的图搜索策略
- 5.4 启发式图搜索策略

5.2 状态空间的搜索策略

- **5.2.1 状态空间表示法**
- **5.2.2 状态空间的图描述**

5.2.1 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：

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

- **操作：**表示引起状态变化的过程型知识的一组关系或函数：

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

5.2.1 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：若干具体状态或满足某些性质的路径信息描述。

5.2.1 状态空间表示法

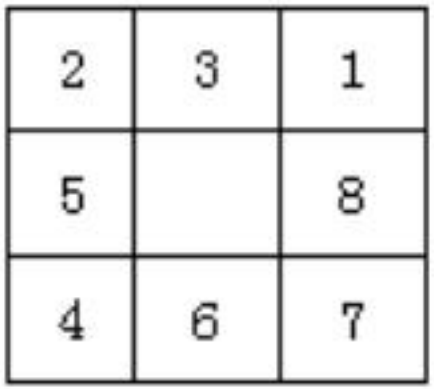
- **求解路径：** 从 S_0 结点到 G 结点的路径。
- **状态空间解：** 一个有限的操作算子序列。

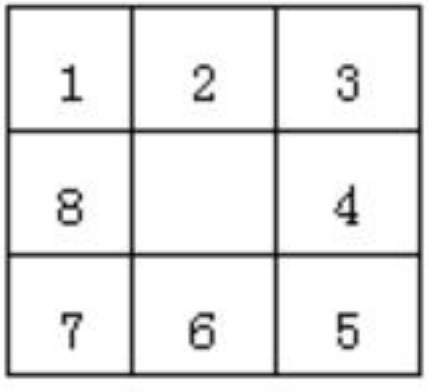
$$S_0 \xrightarrow{O_1} S_1 \xrightarrow{O_2} S_2 \xrightarrow{O_3} \cdots \xrightarrow{O_k} G$$

$O_1, \cdots, O_k$ ：状态空间的一个解。

5.2.1 状态空间表示法

- **例5.1　八数码问题的状态空间。**

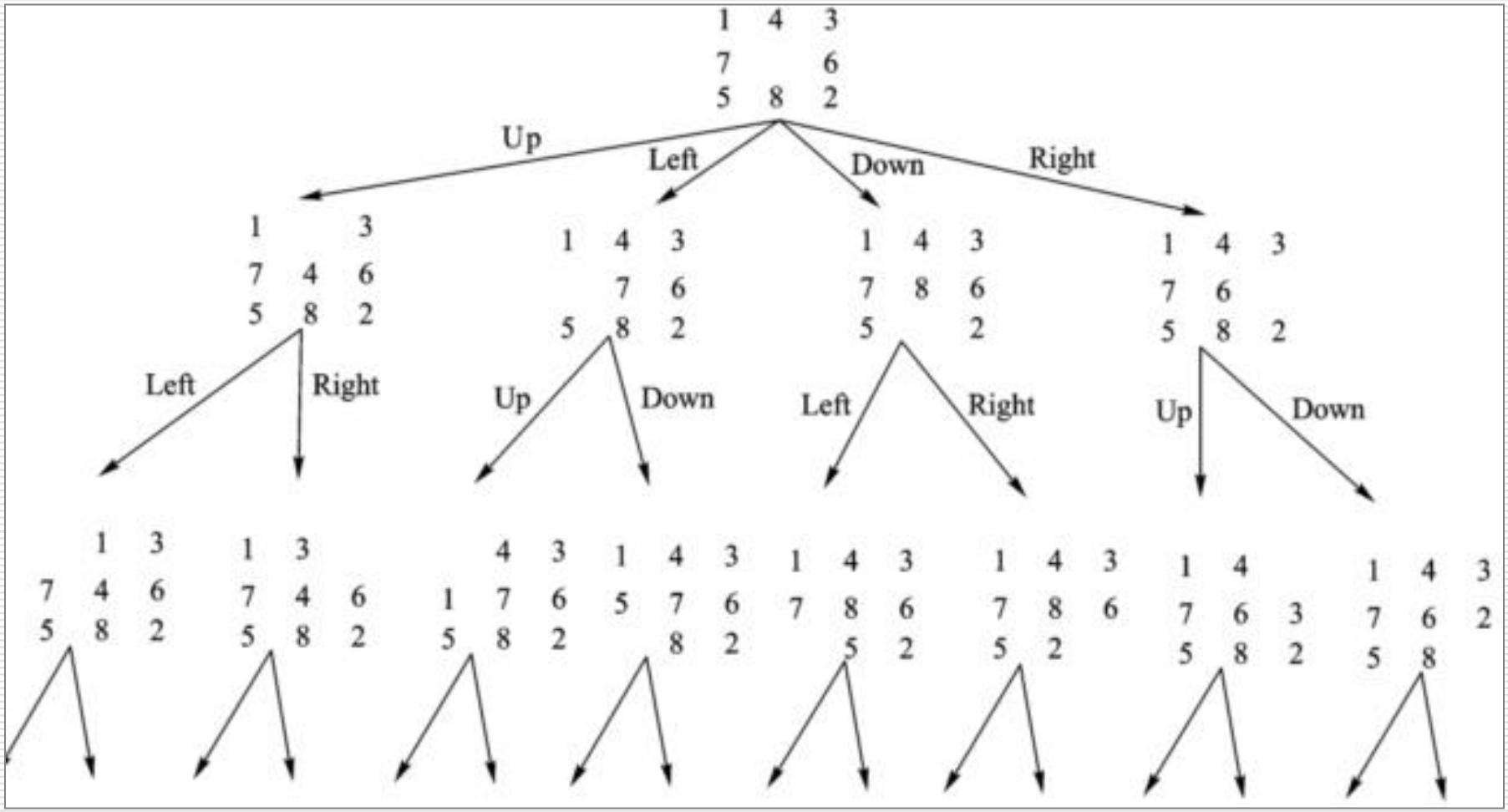




状态集 S ：所有摆法　　操作算子：

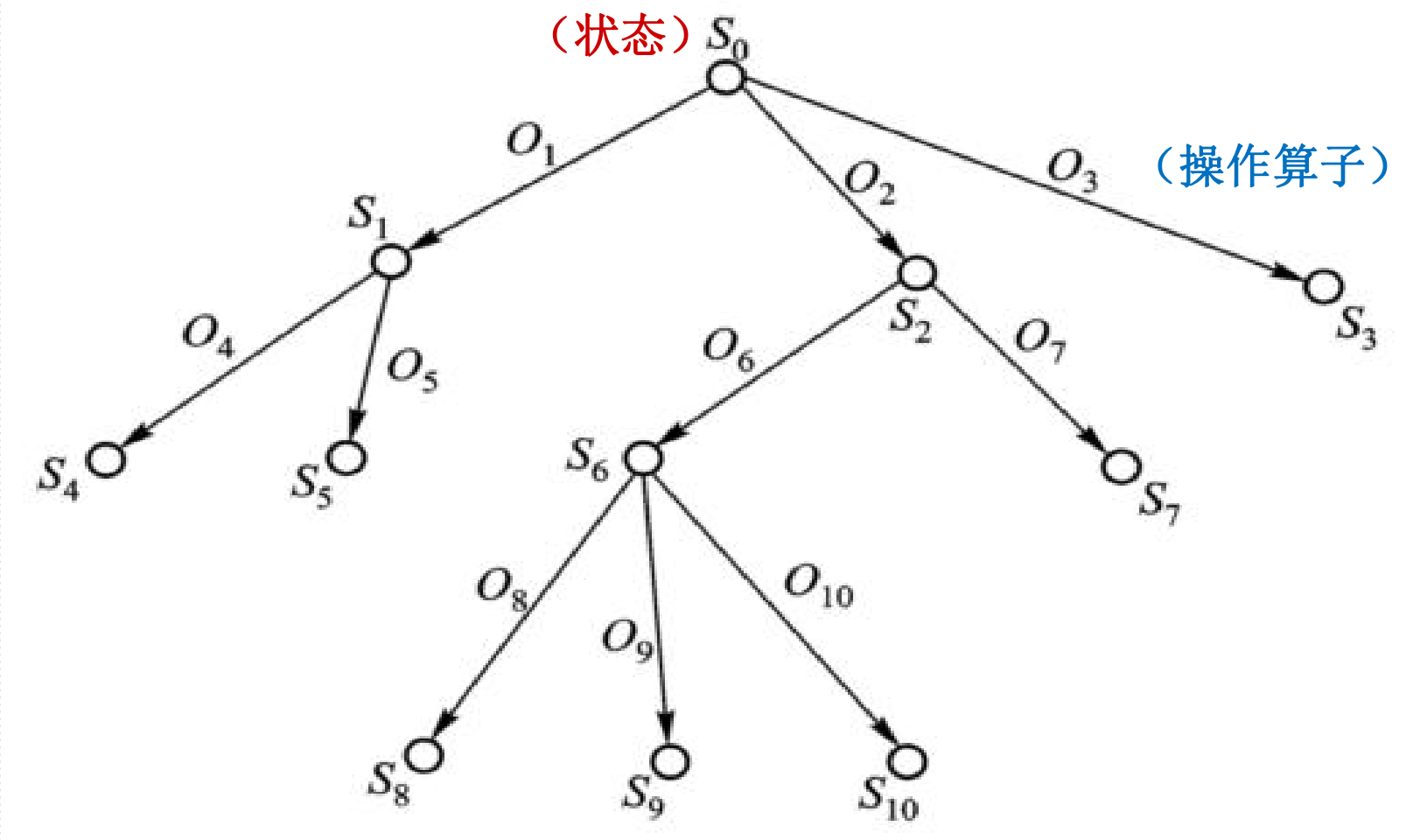
将空格向上移Up
将空格向左移Left
将空格向下移Down
将空格向右移Right

5.2.2 状态空间的图描述



八数码状态空间图

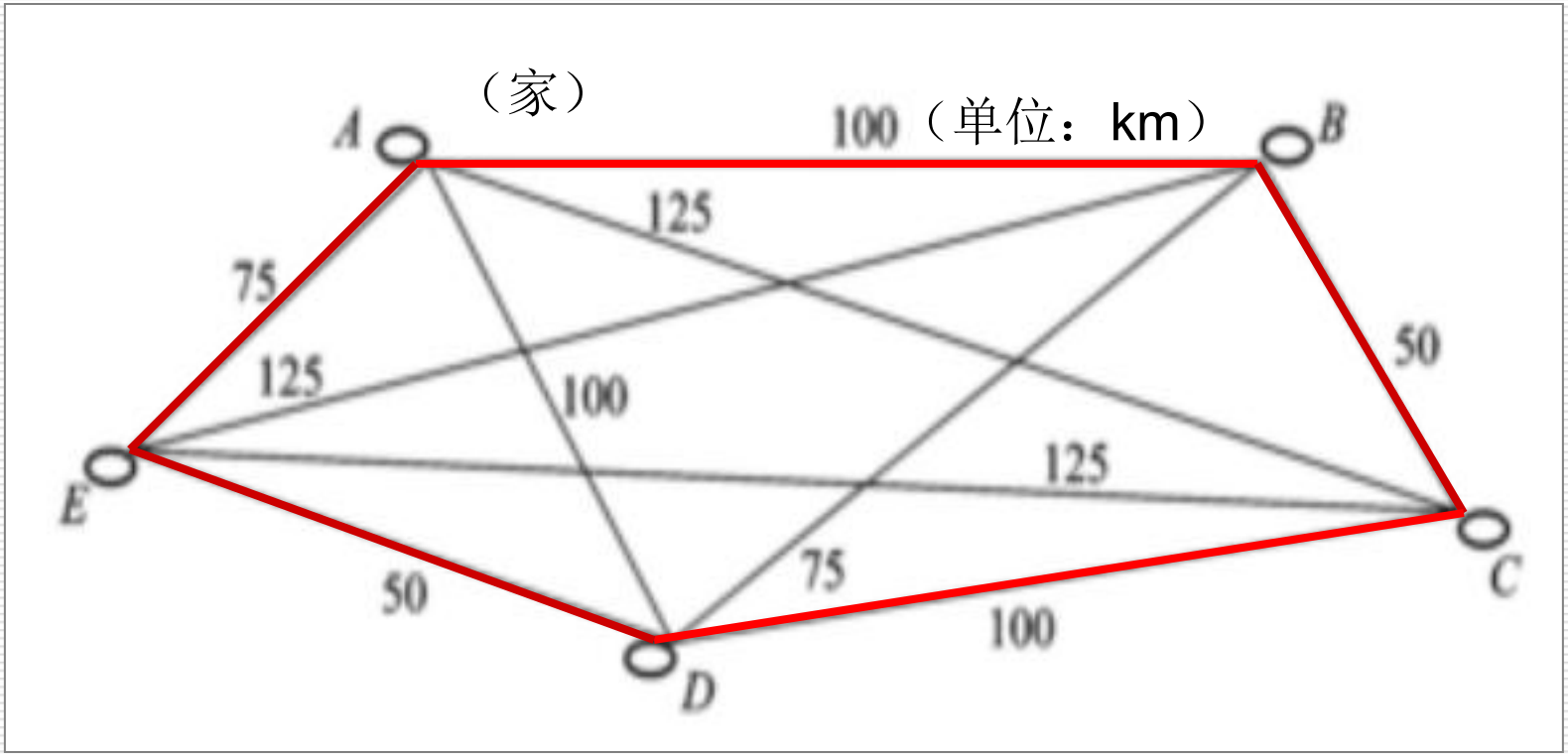
5.2.2 状态空间的图描述



状态空间的有向图描述

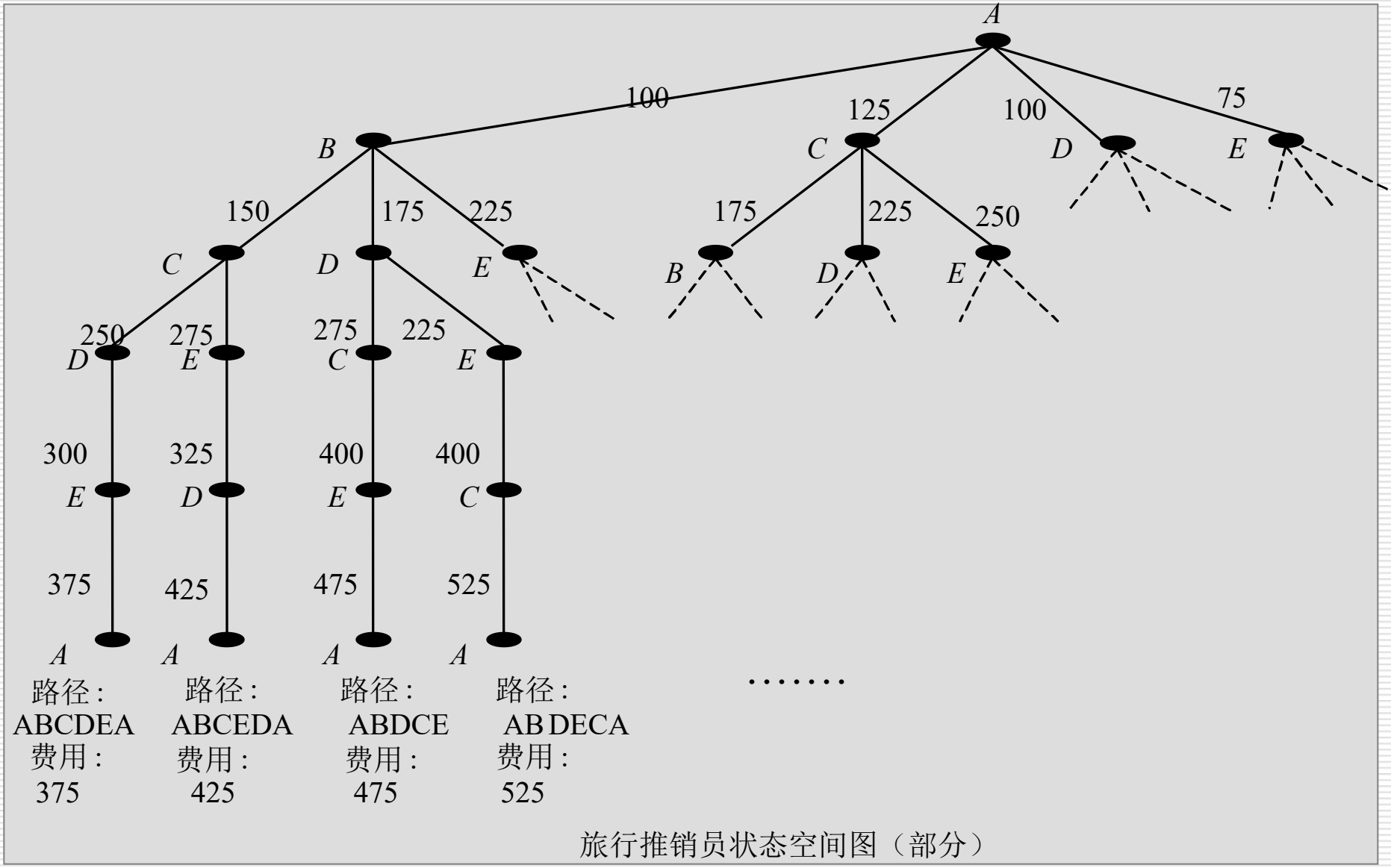
5.2.2 状态空间的图描述

例5.3 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*）

5.2.2 状态空间的图描述



旅行推销员状态空间图（部分）

第5章 搜索求解策略

- 5.1 搜索的概念
- 5.2 状态空间知识表示方法
- ✓ 5.3 盲目的图搜索策略
- 5.4 启发式图搜索策略

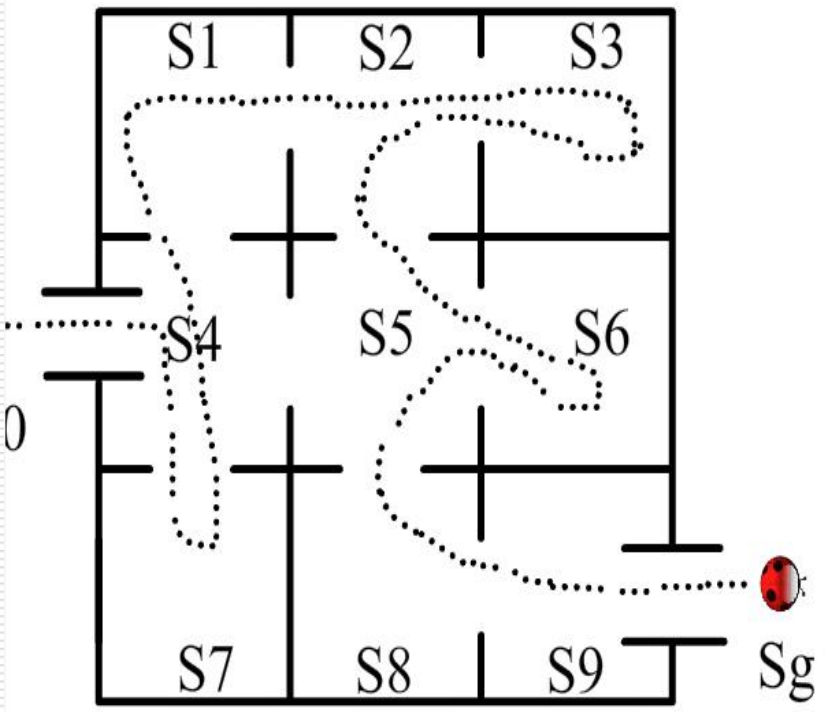
5.3 盲目的图搜索策略

- **5.3.1 回溯策略**
- **5.3.2 宽度优先搜索策略**
- **5.3.3 深度优先搜索策略**

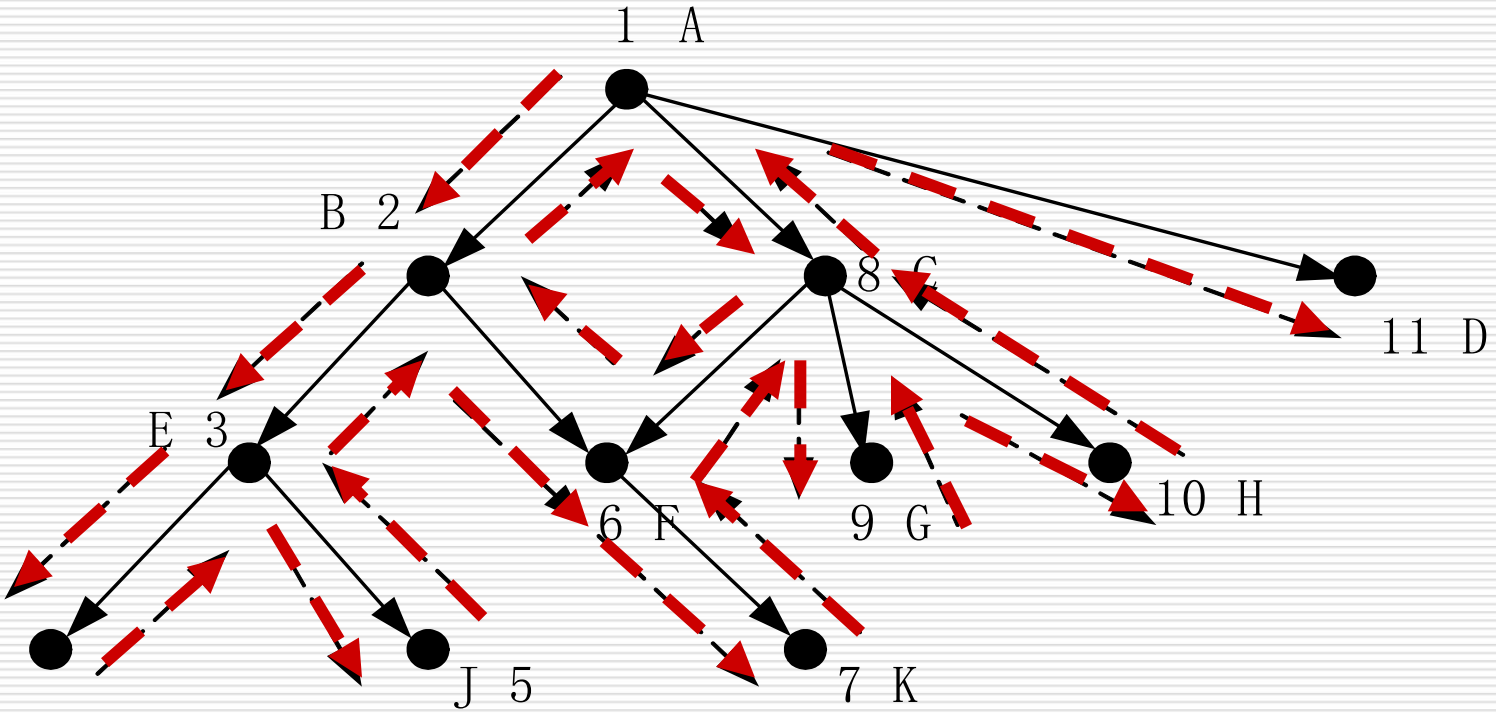
5.3.1 回溯策略

带回溯策略的搜索：

- 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。



5.3.1 回溯策略



回溯搜索示意图

5.3.1 回溯策略

回溯搜索的算法

(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。

5.3.1 回溯策略

- 图搜索算法（深度优先、宽度优先、最好优先搜索等）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 。

5.3.2 宽度优先搜索策略

- **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) 一种高代价搜索，但若有解存在，则必能找到它。

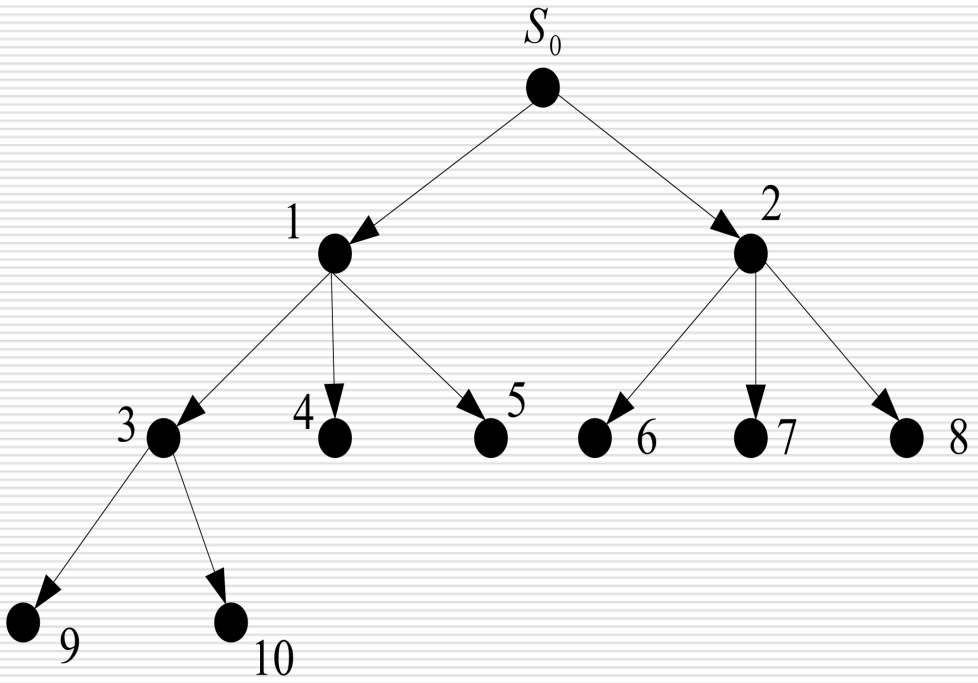
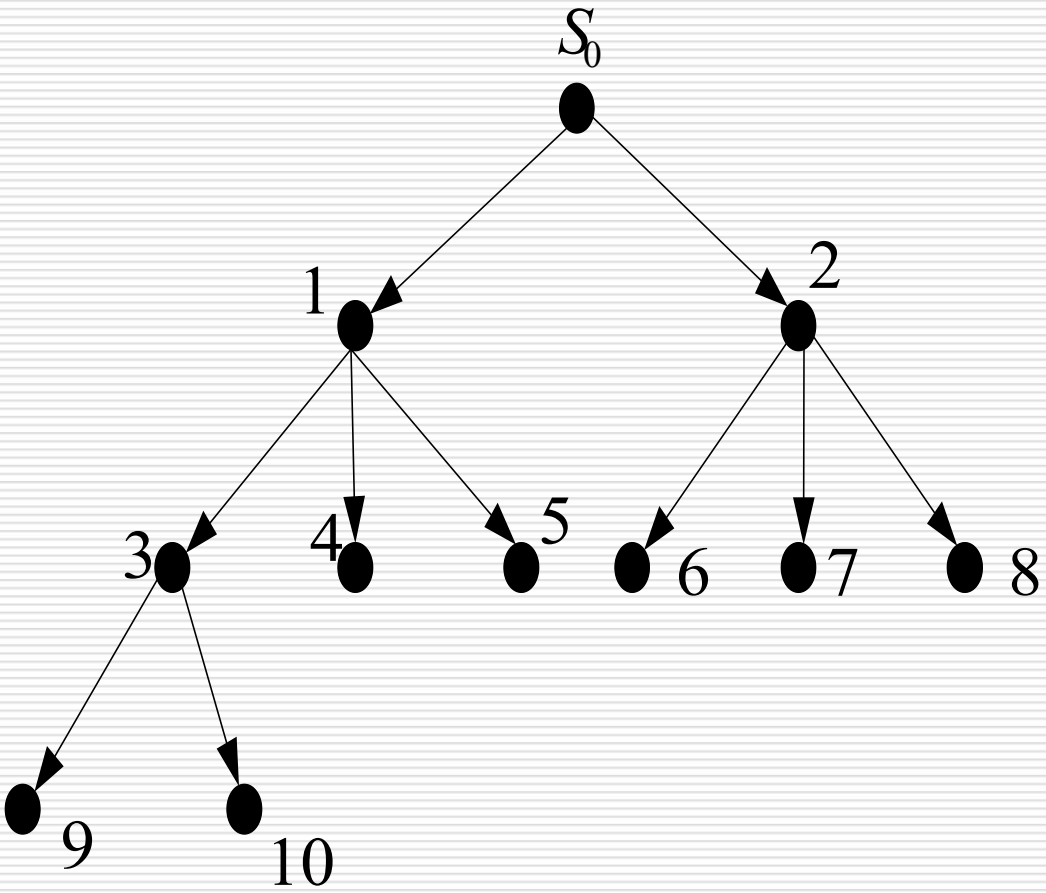


图5.6 宽度优先搜索法中状态的搜索次序

5.3.2 宽度优先搜索策略

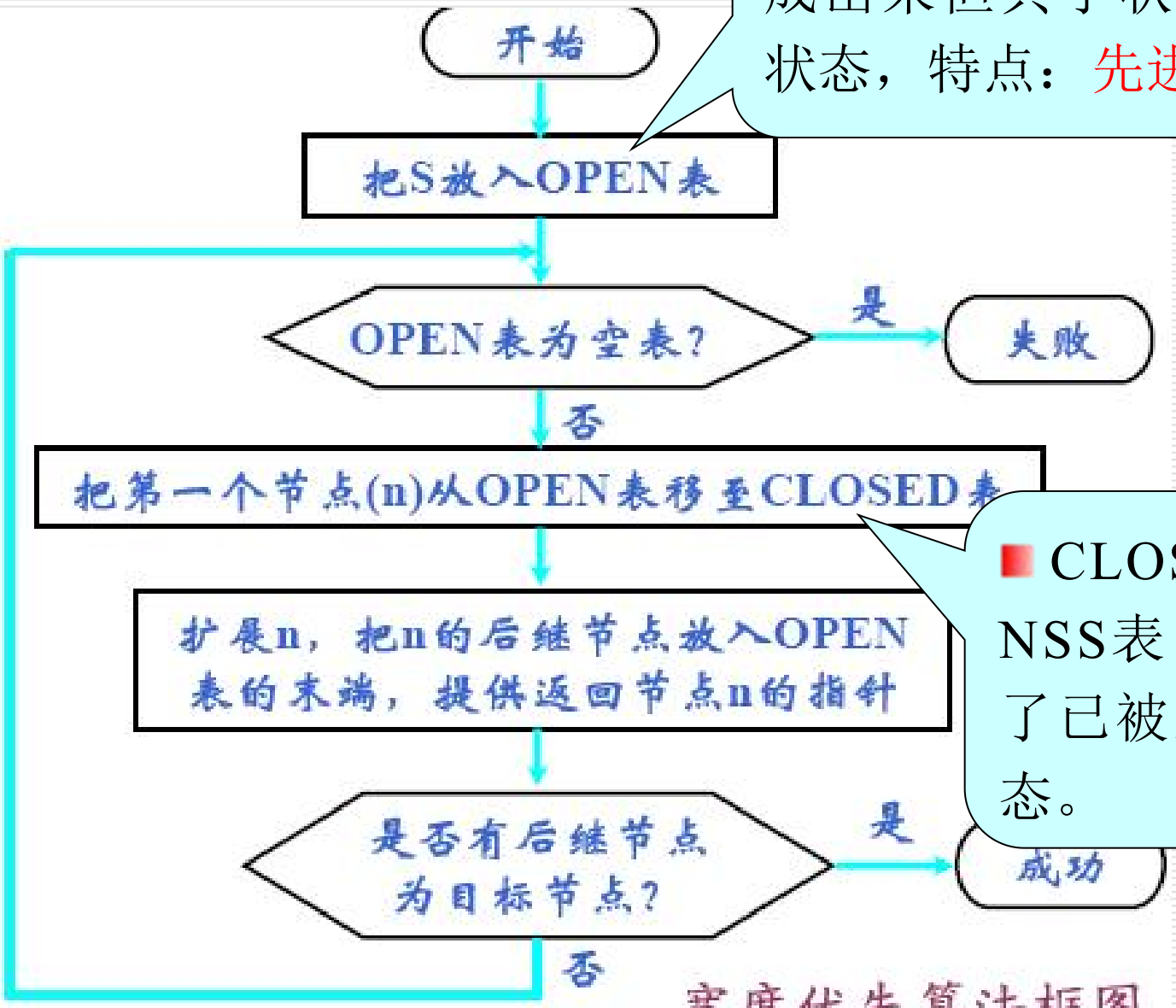


宽度优先搜索法中状态的搜索次序

- open表（NPS表）：已经生成出来但其子状态未被搜索的状态。
- closed表（ PS表和NSS表的合并）：记录了已被生成扩展过的状态。

5.3.2 宽度优先搜索策

- OPEN表（NPS表）：已经生成出来但其子状态未被扩展的状态，特点：先进先出。

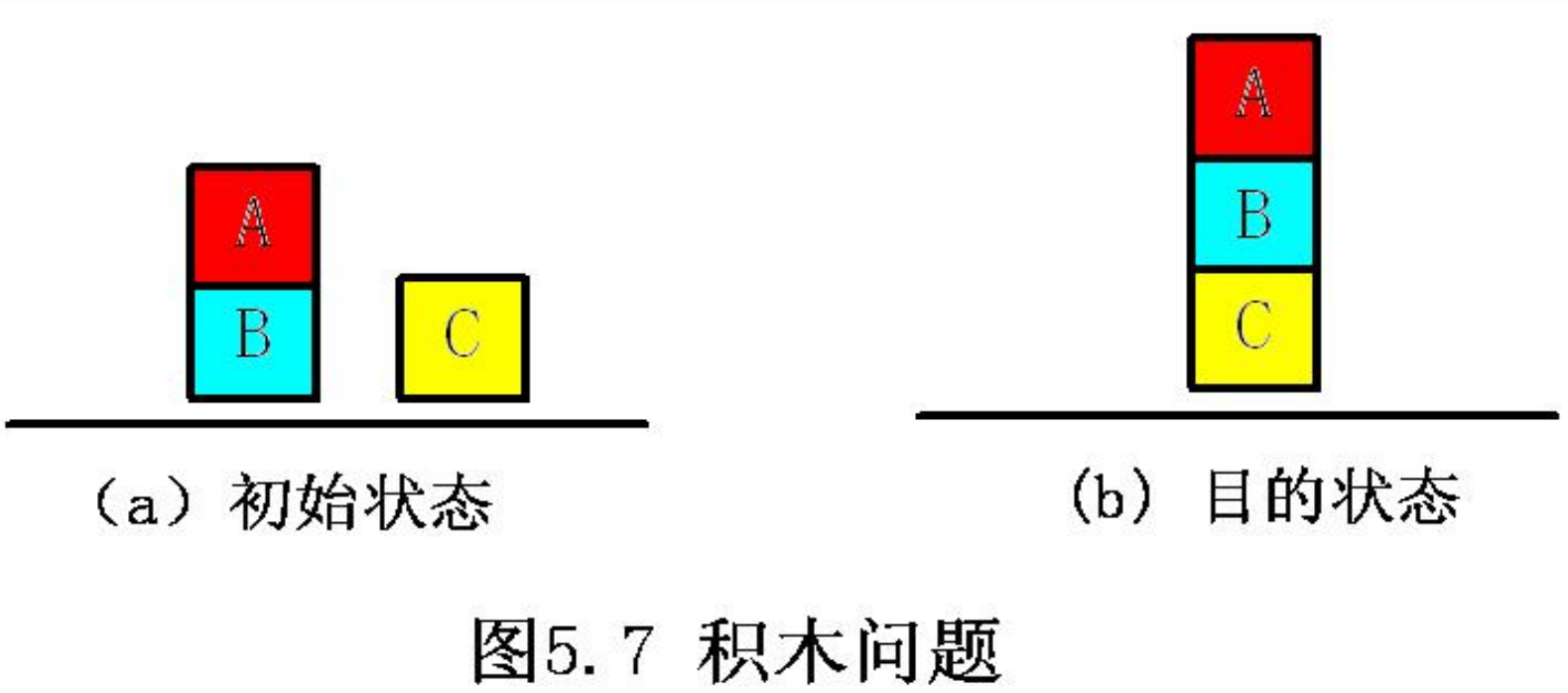


宽度优先算法框图

- CLOSED表（PS表和NSS表的合并）：记录了已被生成扩展过的状态。

5.3.2 宽度优先搜索策略

- 例5.4 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

图5.7 积木问题

5.3.2 宽度优先搜索策略

- **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。

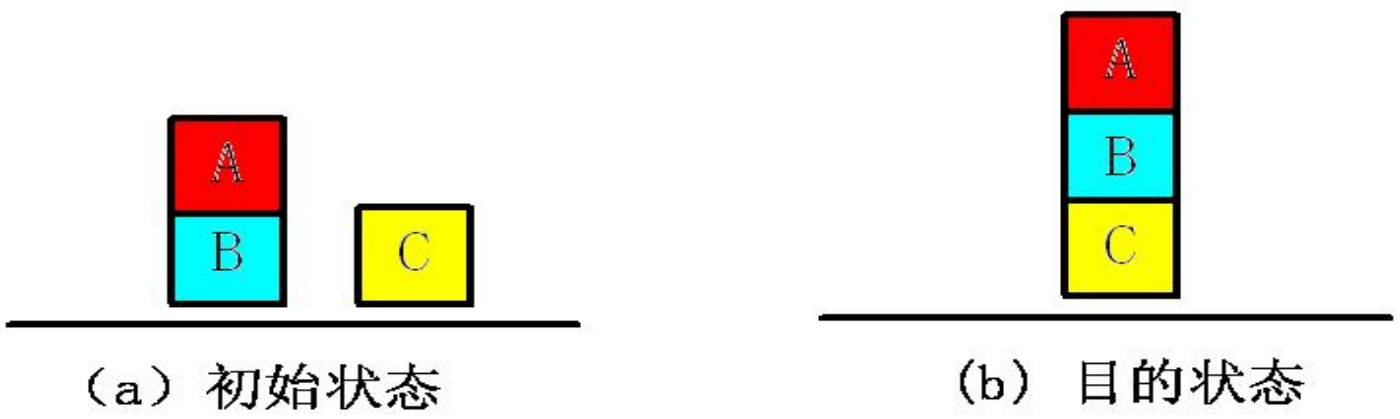
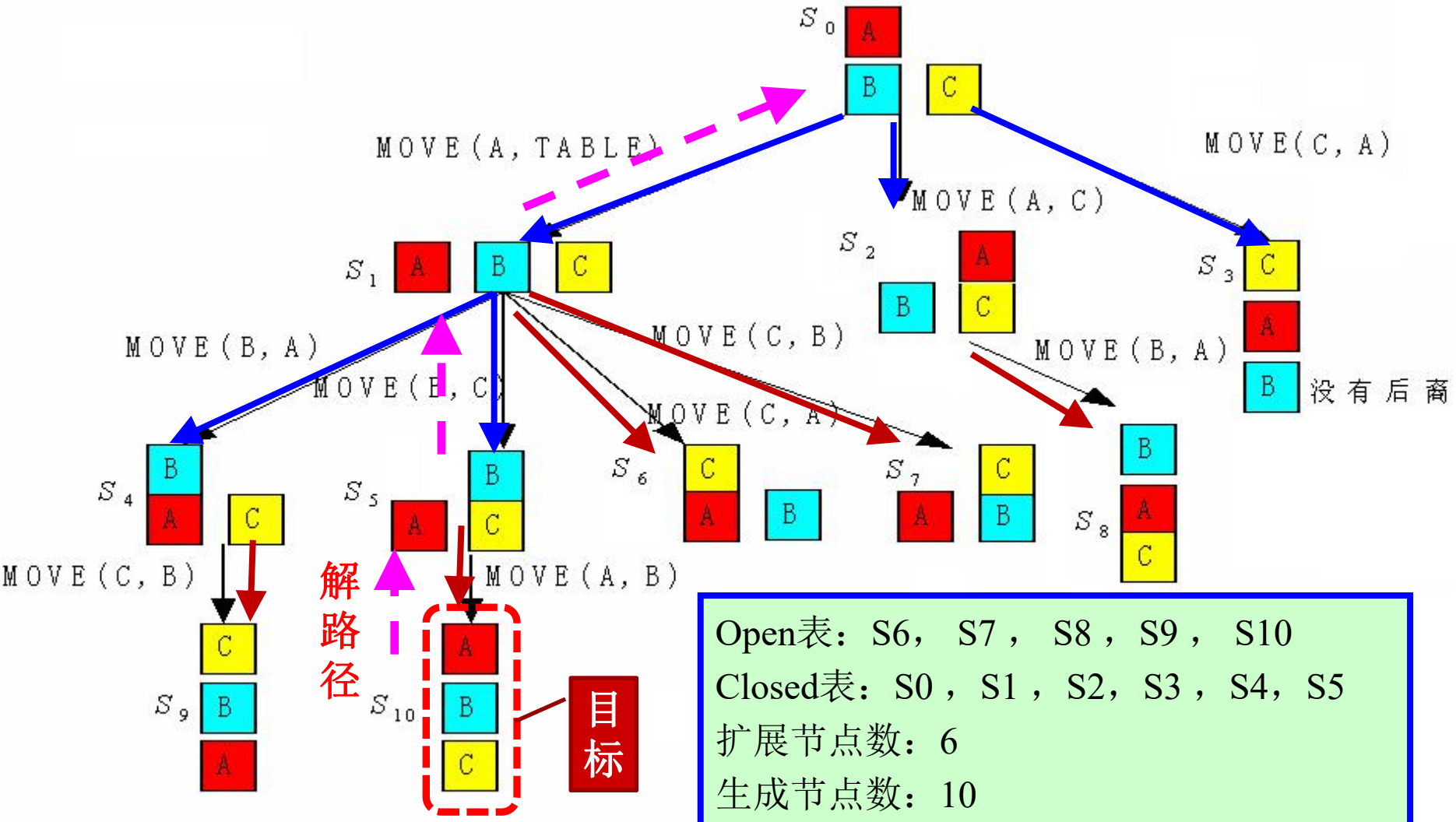


图5.7 积木问题

5.3.2 宽度优先搜索策略



Open表：S6， S7， S8，S9， S10

Closed表：S0，S1，S2，S3，S4，S5

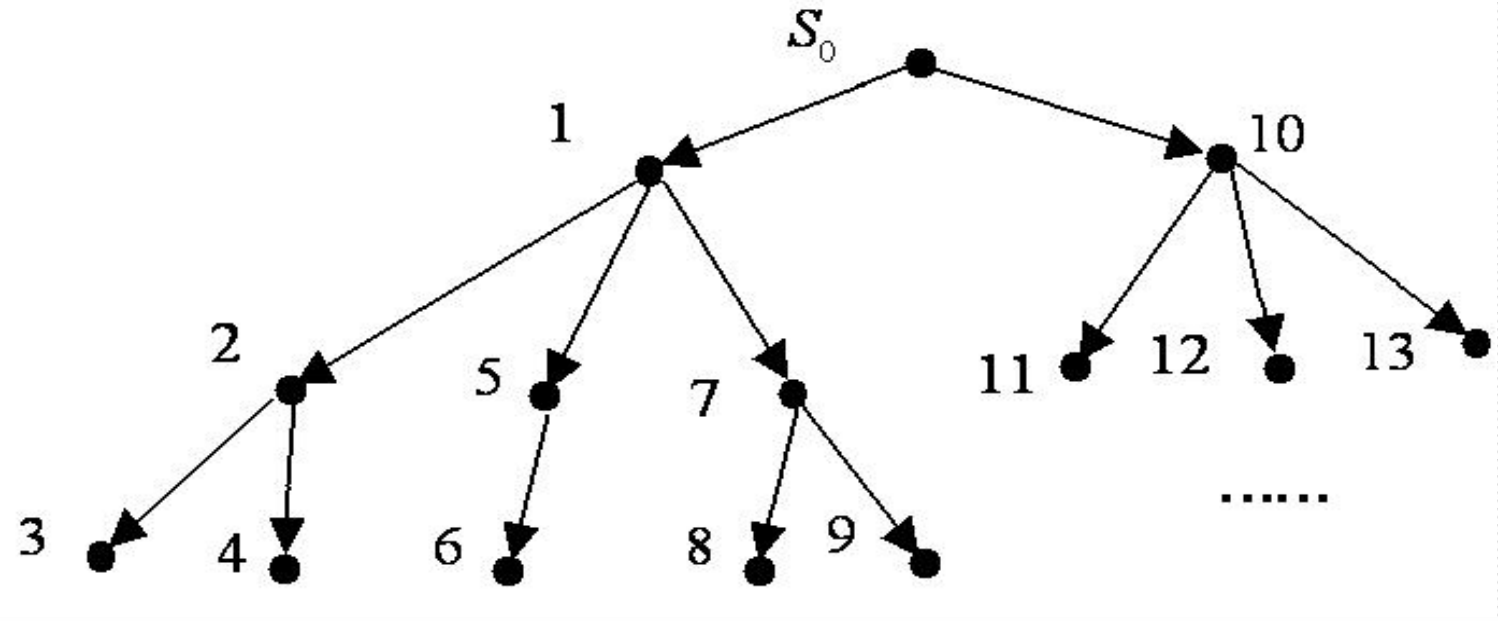
扩展节点数：6

生成节点数：10

图 5.8 积木问题的宽度优先搜索树

5.3.3 深度优先搜索策略

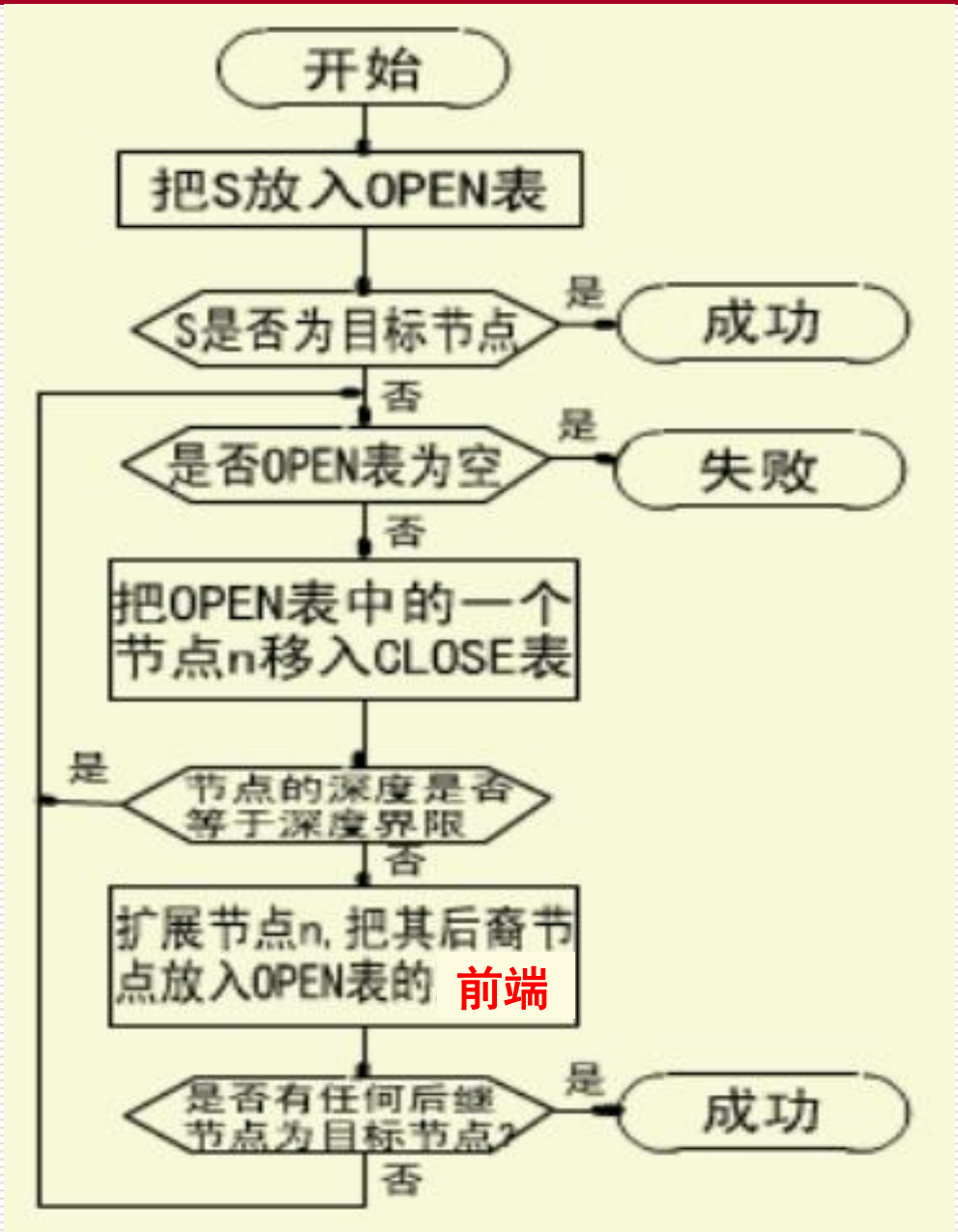
- **深度优先搜索（Depth-first Search):** 首先扩展最新产生的节点，深度相等的节点按生成次序的盲目搜索。



- 特点：扩展最深的节点的结果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下去；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。

5.3.3 深度优先搜索策略

- 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**表后进先出。



5.3.3 深度优先搜索策略

- 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。

5.3.3 深度优先搜索策略

- 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。

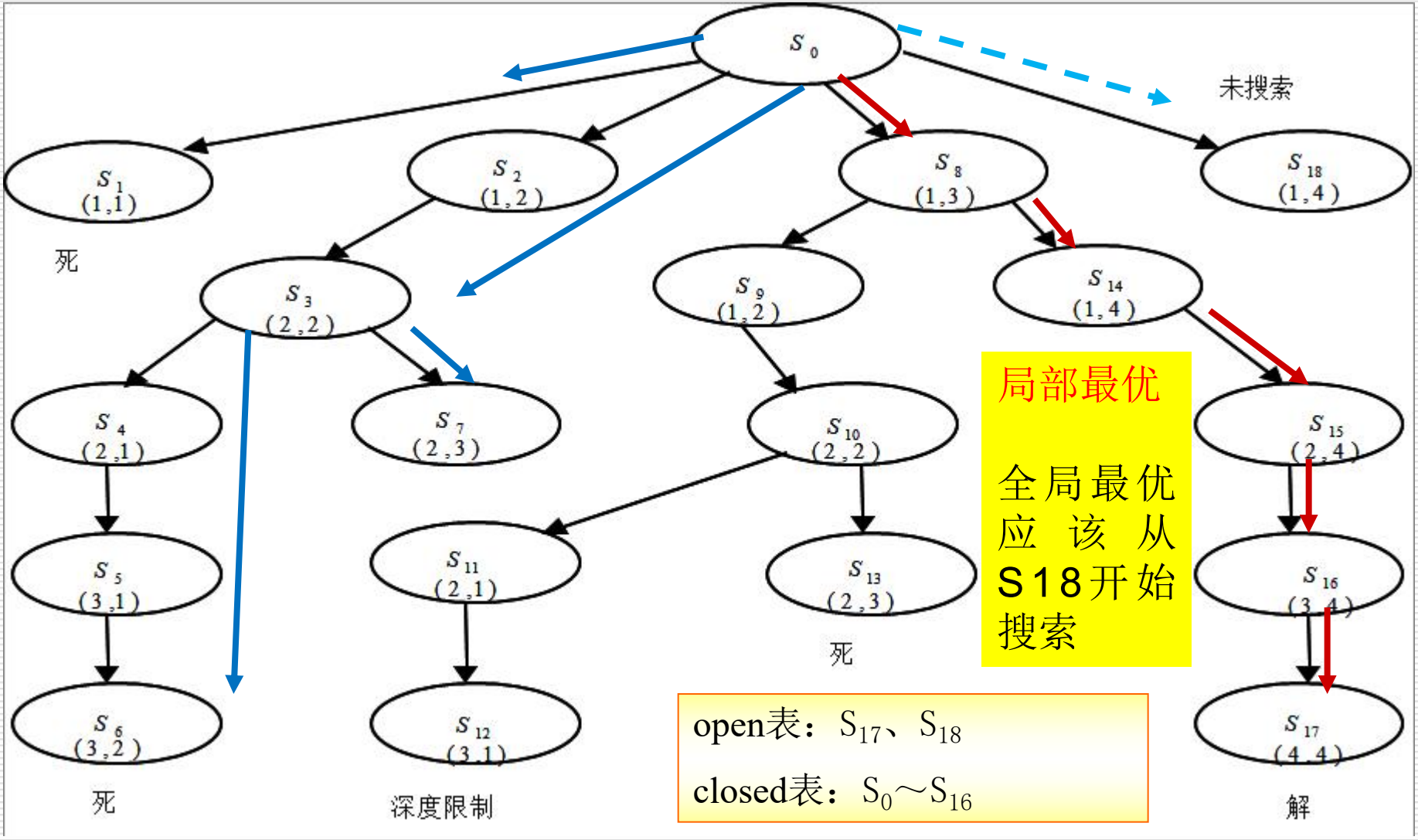
5.3.3 深度优先搜索策略

- **例5.5　卒子穿阵问题**，要求一卒子从顶部通过下图所示的阵列到达底部。卒子行进中不可进入到代表敌兵驻守的区域（标注1），并不准后退。假定深度限制值为5。

行	1	2	3	4	列
1	1	0	0	0	
2	0	0	1	0	
3	0	1	0	0	
4	1	0	0	0	

阵列图

5.3.3 深度优先搜索策略



卒子穿阵的深度优先搜索树

第5章 搜索求解策略

- 5.1 搜索的概念
- 5.2 状态空间知识表示方法
- 5.3 盲目的图搜索策略
- ✓ 5.4 启发式图搜索策略
- 5.5 与/或图搜索策略

5.4 启发式图搜索策略

- **5.4.1 启发式策略**
- **5.4.2 启发信息和估价函数**
- **5.4.3 *A*搜索算法**
- **5.4.4 *A**搜索算法及其特性分析**

5.4.1 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。
- 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。

5.4.1 启发式策略

- **运用启发式策略的两种基本情况：**

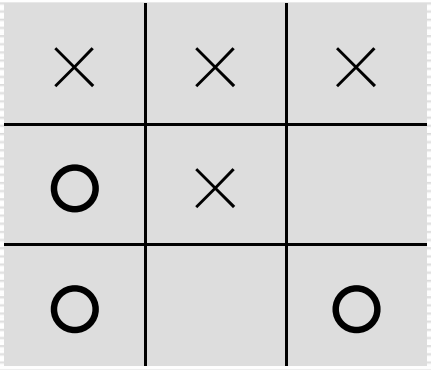
（1）一个问题由于存在问题陈述和数据获取的模糊性，可能会使它没有一个确定的解。

（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。

5.4.1 启发式策略

- **例5.6 一字棋。**

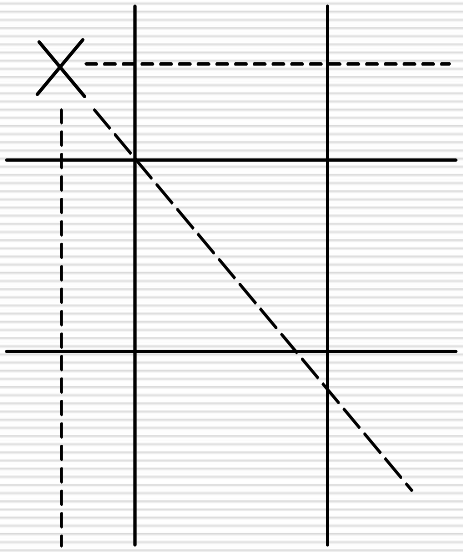
在九宫棋盘上，从空棋盘开始，双方轮流在棋盘上摆各自的棋子 × 或 ○（每次一枚），谁先取得三子一线（一行、一列或一条对角线）的结果就取胜。



- × 和 ○ 能够在棋盘中摆成的各种不同的棋局就是问题空间中的不同状态。
- 9个位置上摆放{空， × ， ○}有 3^9 种棋局。（部分状态在下棋中不会出现。）
- 可能的走法：9×8×...×1，即有意义状态的数量。（考虑下棋规则的前提下）

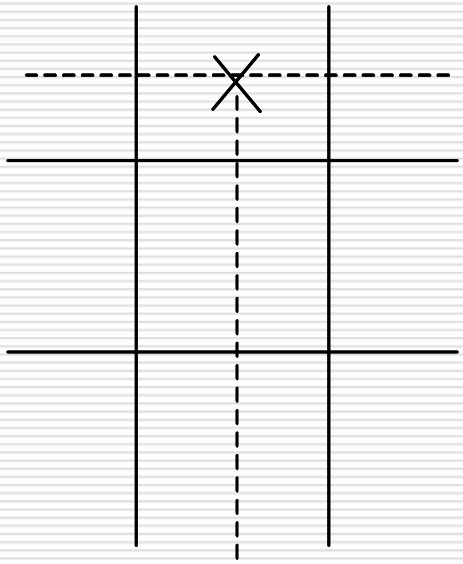
5.4.1 启发式策略

赢的几率：双方能够成一字型的摆放种类之差



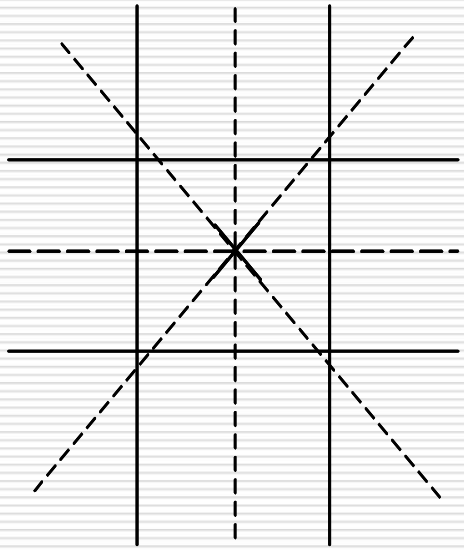
赢的几率③

8 – 5 = 3



赢的几率②

8 – 6 = 2

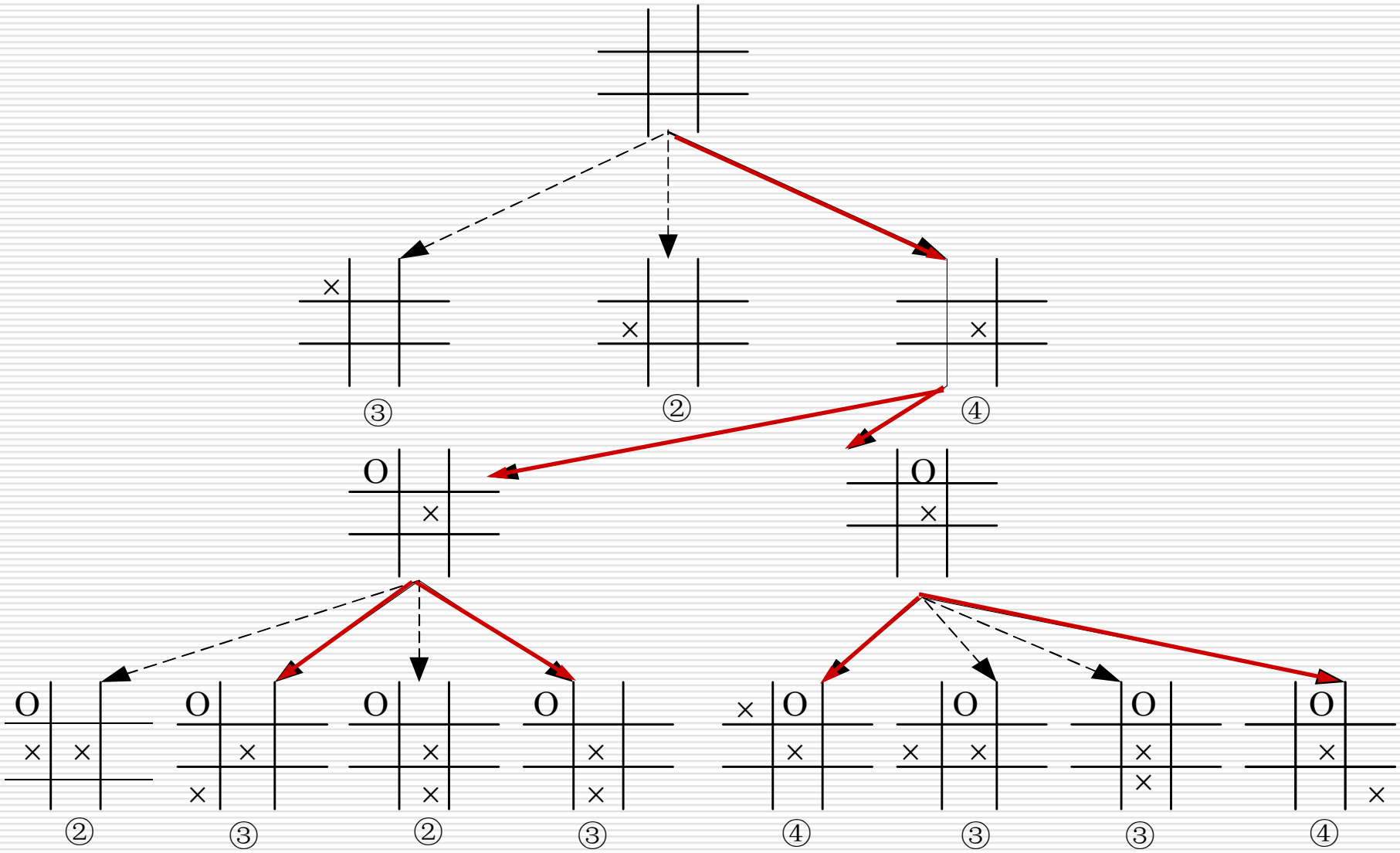


赢的几率④

8 - 4 = 4

启发式策略的运用

5.4.1 启发式策略



启发式搜索下缩减的状态空间

5.4.2 启发信息和估价函数

- 在具体求解中，能够利用与该问题有关的信息来简化搜索过程，称此类信息为**启发信息**。
- **启发式搜索：**利用启发信息的搜索过程。

5.4.2 启发信息和估价函数

- 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！

5.4.2 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
  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。

5.4.2 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

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

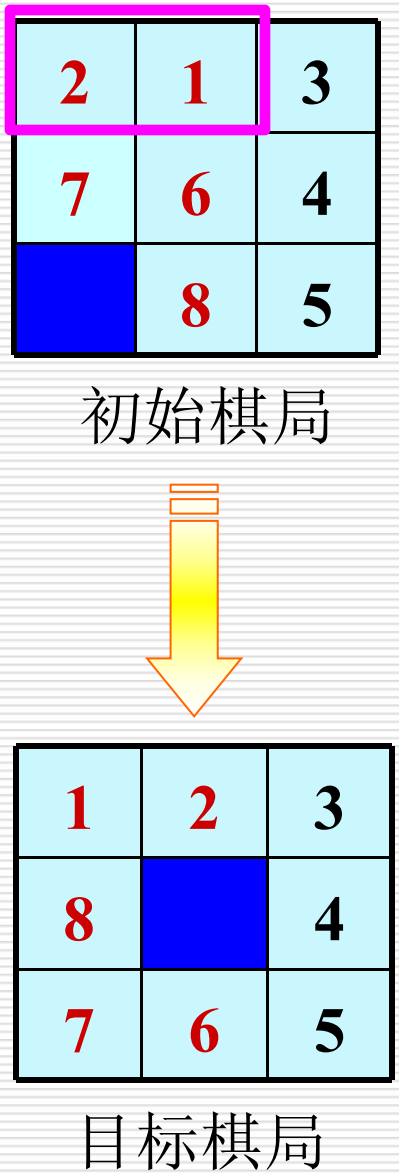
- $g(n)$：从初始节点 S_0 到节点 n 的实际代价；
- $h(n)$：从节点 n 到目标节点 S_g 的最优路径的估计代价，称为启发函数。

$h(n)$ 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。

5.4.2 启发信息和估价函数

- 例5.7　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 $h(S_0) = 8$。



5.4.3 A搜索算法

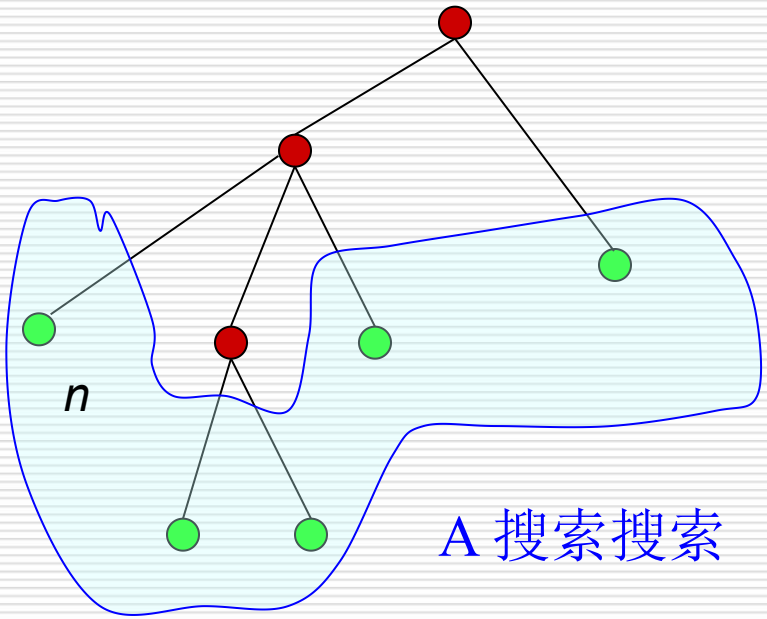
A 搜索算法：使用了估价函数 f 的最佳优先搜索。

- 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。

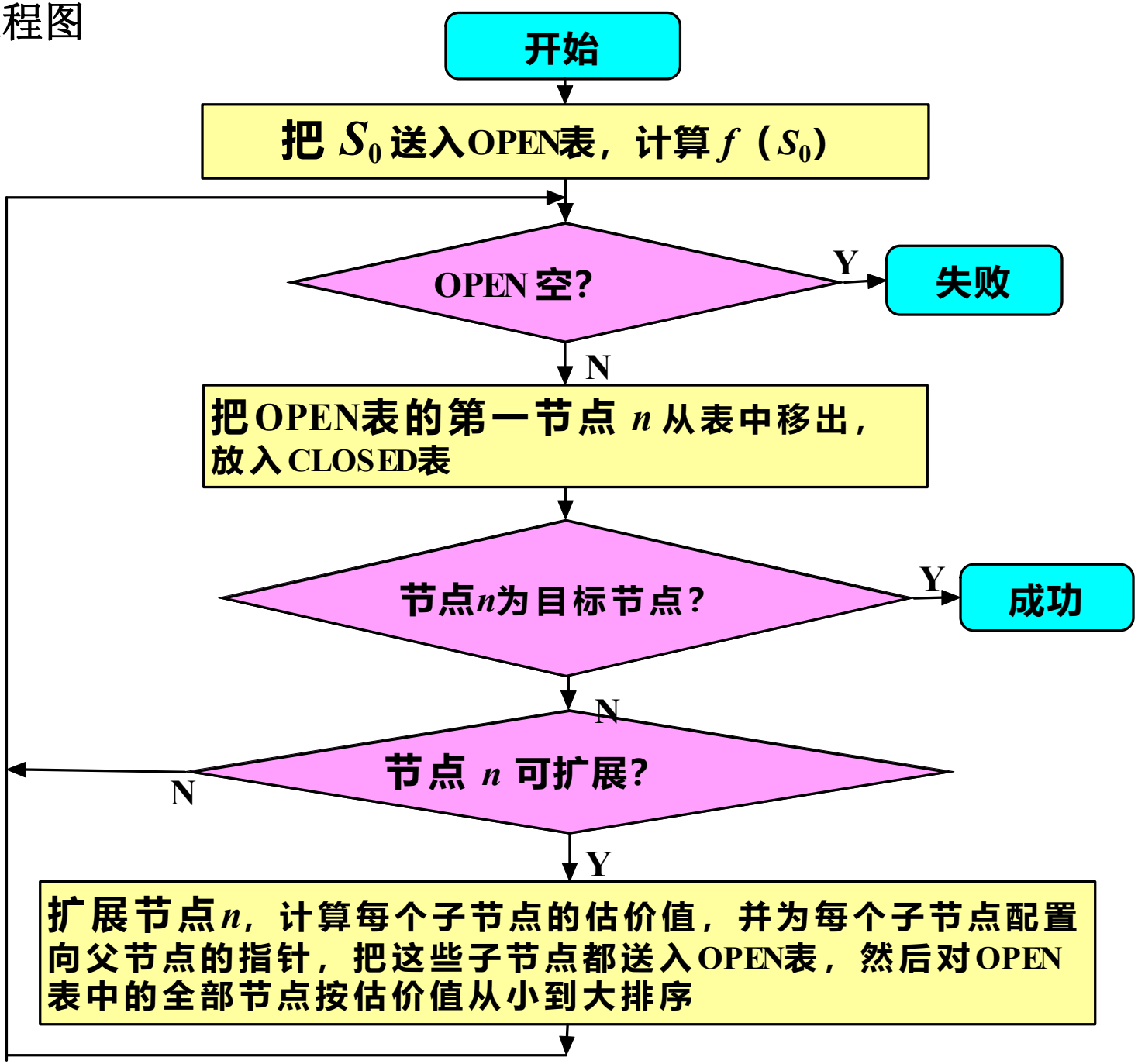
- 估价函数 $f(n)=g(n)+h(n)$
- 如何寻找并设计启发函数 $h(n)$，然后以 $f(n)$ 的大小来排列OPEN表中待扩展状态的次序，每次选择 $f(n)$ 值最小者进行扩展。

$g(n)$：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。

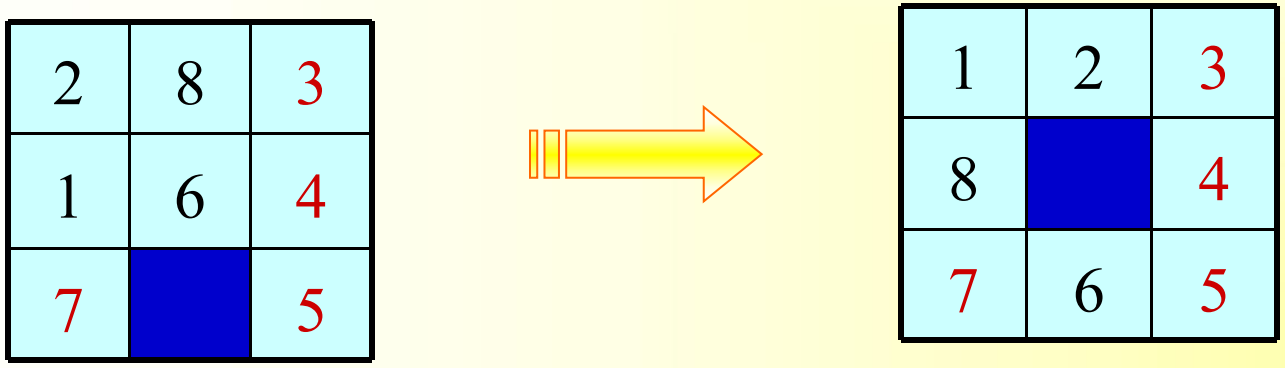


A 算法流程图

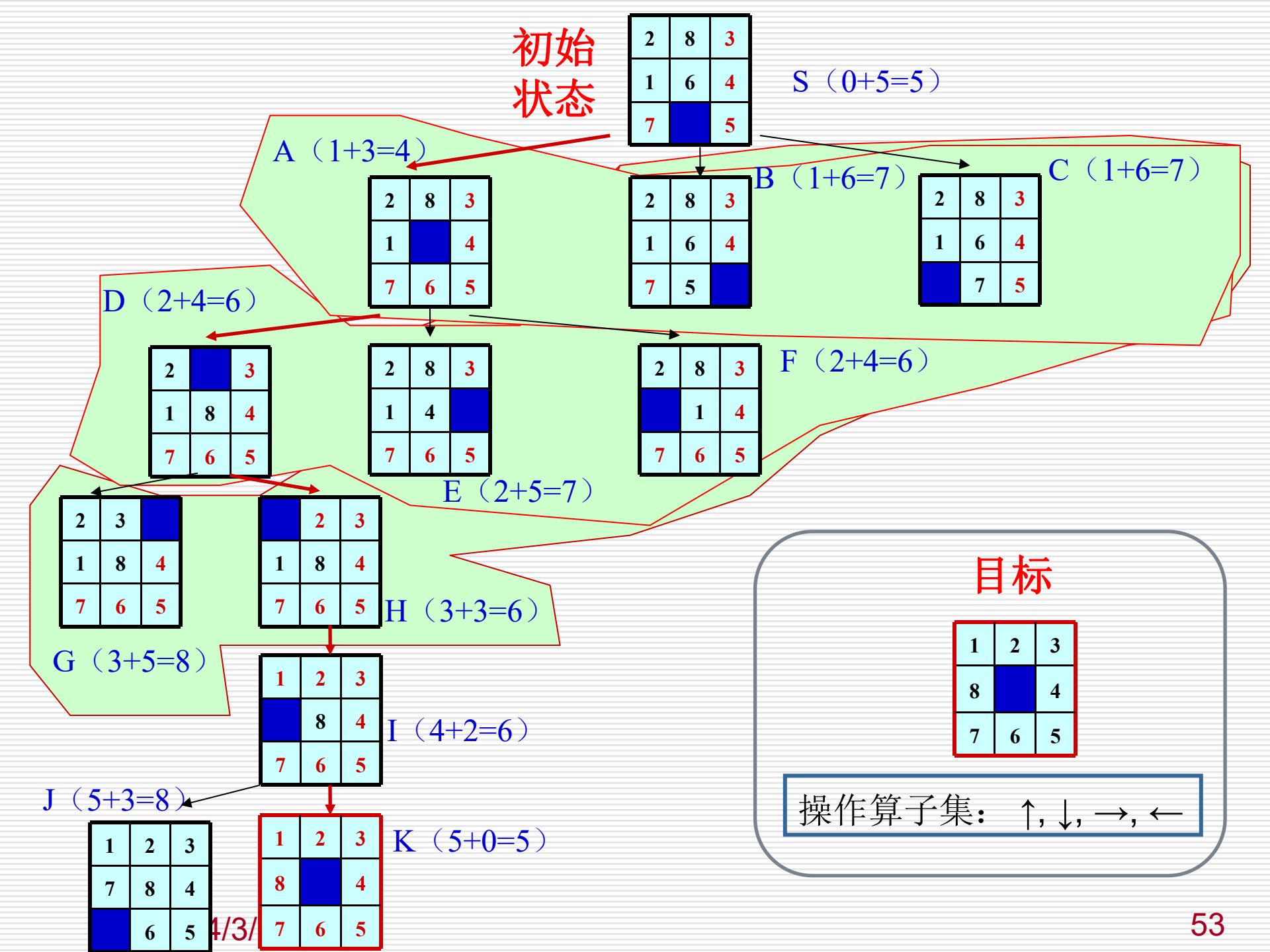

开始
把 S_0 送入OPEN表，计算 $f(S_0)$
OPEN 空?
Y
失败
N
把OPEN表的第一节点 n 从表中移出，放入CLOSED表
节点n为目标节点?
Y
成功
N
节点 n 可扩展?
N
Y
扩展节点n，计算每个子节点的估价值，并为每个子节点配置向父节点的指针，把这些子节点都送入OPEN表，然后对OPEN表中的全部节点按估价值从小到大排序

5.4.3 *A*搜索算法

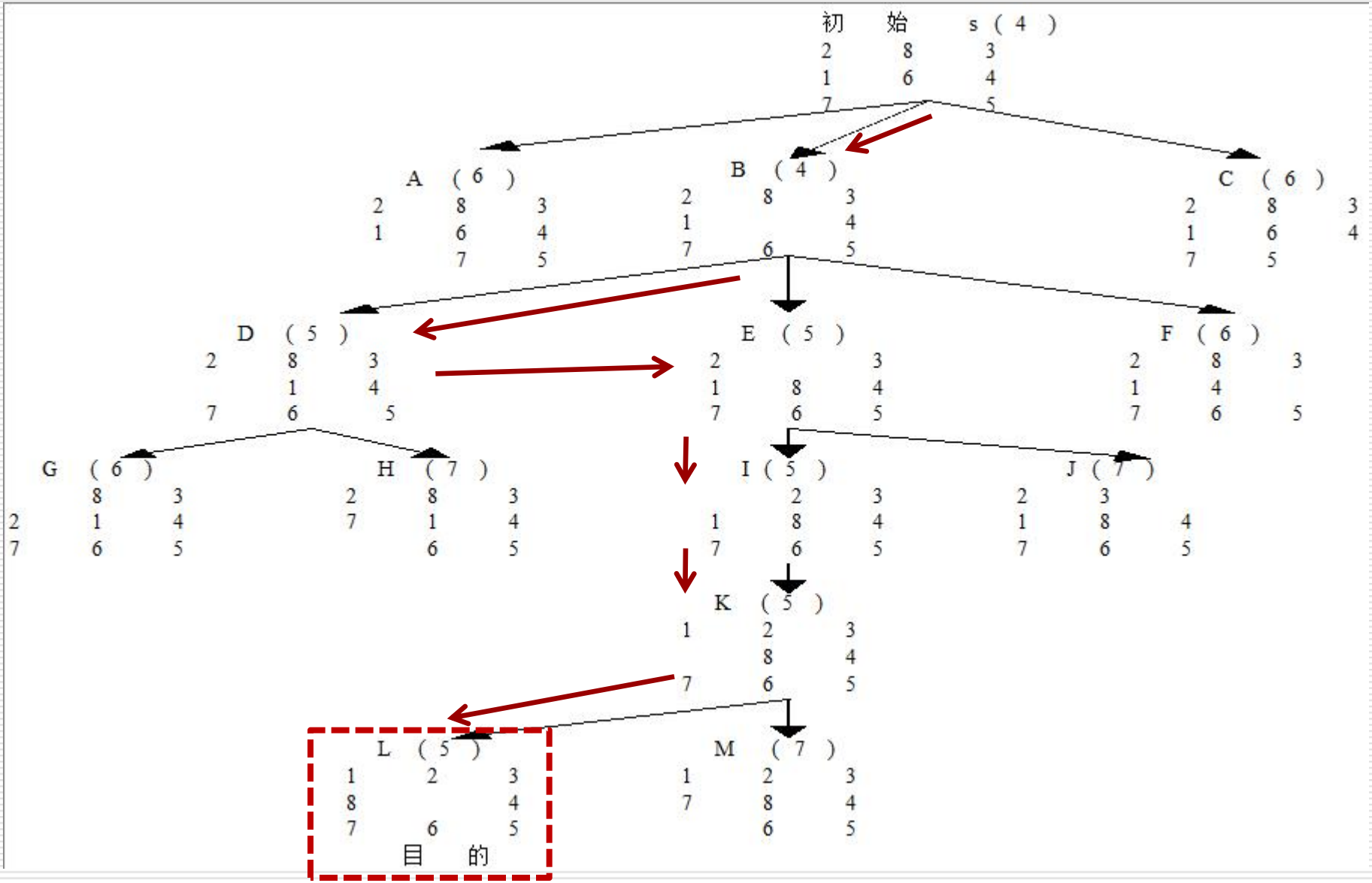
- 例5.8 利用 *A* 搜索算法求解八数码问题，问最少移动多少次就可达到目标状态？
- 估价函数定义为 $f(n)=g(n)+h(n)$
- $g(n)$：节点 n 的深度，如 $g(S_0)=0$。
- $h(n)$：节点 n 与目标棋局不相同的位数（包括空格），简称“不在位数”，如 $h(S_0)=5$。



初始状态 S_0　　目标状态


初始状态
S（0+5=5）
A（1+3=4）
B（1+6=7）
C（1+6=7）
D（2+4=6）
F（2+4=6）
E（2+5=7）
H（3+3=6）
G（3+5=8）
I（4+2=6）
J（5+3=8）
K（5+0=5）
目标
操作算子集：↑, ↓, →, ←

$h(n)$：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 $h(S_0)=4$。



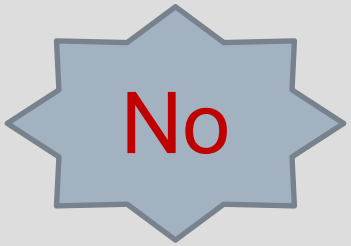
5.4.3 *A*搜索算法

- open表和closed表内状态排列的变化情况

Open 表	Closed 表
初始化：(s(4))	(_)
一次循环后：	
(B(4) A(6) C(6)	(s(4))
二次循环后：	
(D(5) E(5) A(6) C(6) F(6))	(s(4) B(4))
三次循环后：	
(E(5) A(6) C(6) F(6) G(6) H(7))	(s(4) B(4) D(5))
四次循环后：	
(I(5) A(6) C(6) F(6) G(6) H(7) J(7))	(s(4) B(4) D(5) E(5))
五次循环后：	
(K(5) A(6) C(6) F(6) G(6) H(7) J(7))	(s(4) B(4) D(5) E(5) I(5))
六次循环后：	
(L(5) A(6) C(6) F(6) G(6) H(7) J(7) M(7))	(s(4) B(4) D(5) E(5) I(5) K(5))
七次循环后：	
L为目的状态，则成功推出，结束搜索	(s(4) B(4) D(5) E(5) I(5) K(5) L(5))

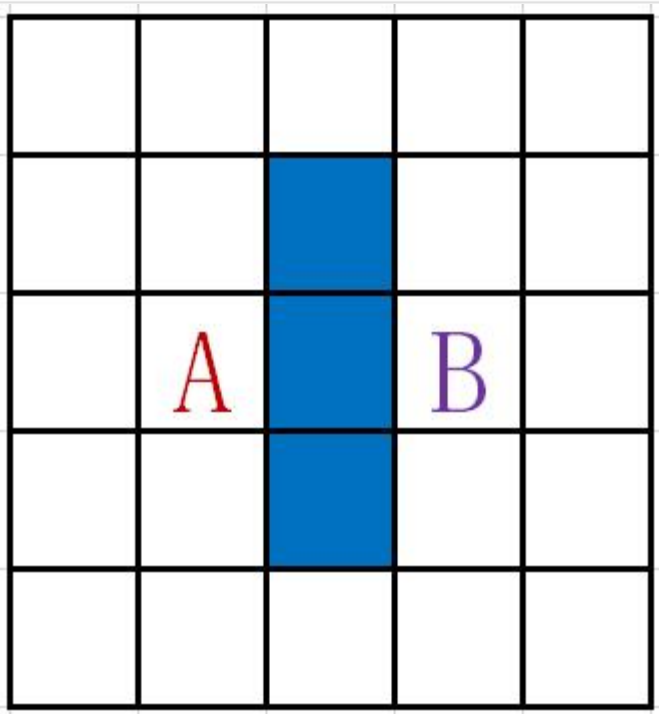
问题：A搜索算法能不能保证找到最优解（路径最短的解）？

- $g(n)$：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



5.4.4 *A**搜索算法及其特性分析

- A*算法：定义h*(n)为状态n到目的状态的最优路径的代价，则当A搜索算法的启发函数h(n)小于等于h*(n)时，即h(n)≤h*(n)，则被称为A*算法。



问题：搜索从A到B点的最优路径。

- 策略：

（1）定义h(n)为当前位置到目标点的距离。

（2）定义每个网格的距离为1。

- 启发式函数验证：

h(0)=2, h*(0)=6。

5.4.4 *A**搜索算法及其特性分析

- 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（*S*，*B*，*E*，*I*，*K*，*L*）为最优解路，其步数为状态*L*（5）上所标注的5 。

5.4.4 A*搜索算法及其特性分析

1. 可采纳性

- 当一个搜索算法在最短路径存在时能保证找到它，就称该算法是可采纳的。
- ***A***搜索算法是可采纳的*。
- ***A***搜索算法*($h(n)=0$)

宽度优先搜索算法

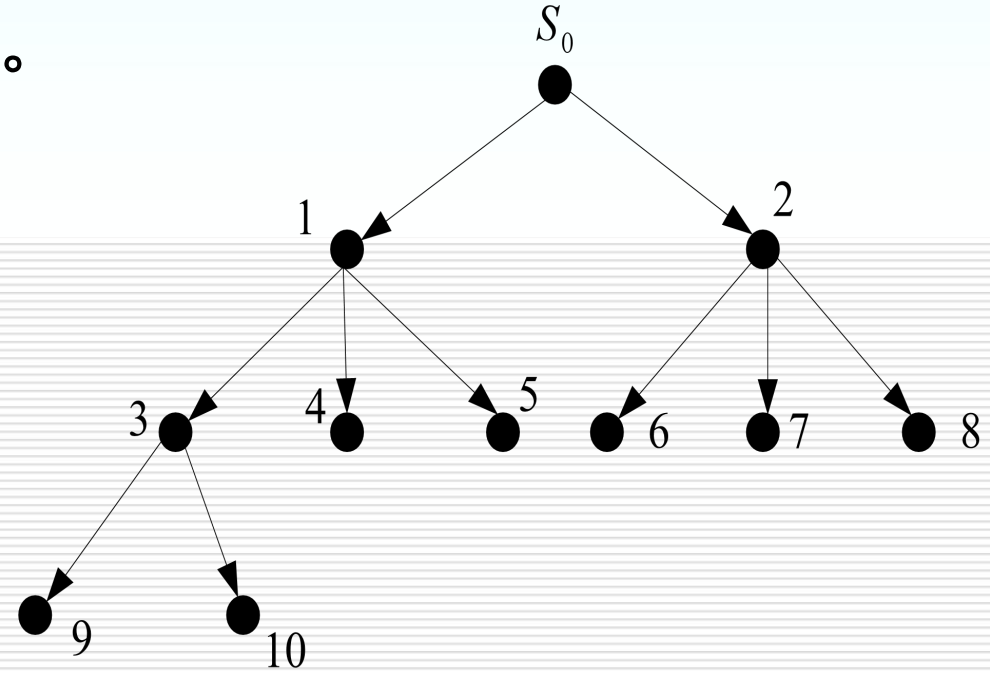


图5.6 宽度优先搜索法中状态的搜索次序

5.4.4 A*搜索算法及其特性分析

2. 单调性

- *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。
- 如果对启发函数$h(n)$加上单调性的限制，就可以总从祖先状态沿着最佳路径到达任一状态。
- 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$h(n_i)-h(n_j)<=\text{cost}(n_i, n_j)$，其中$\text{cost}(n_i, n_j)$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。

- 则称$h(n)$是单调的。
- *A**搜索算法中采用单调性启发函数，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。

5.4.4 A*搜索算法及其特性分析

3. 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$h_1(n) \le h_2(n)$，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。

5.4.5 *A**搜索算法实例

- A*算法的核心步骤
  - 构造启发式函数h(n)，并满足h(n)≤h*(n)。
  - 对当前状态计算f(n)=g(n)+h(n)，从候选项中取最小值的选项继续迭代。
  - 若有相同的最小值选项，取h(n)较小者。若h(n)也相等，则随机选择一个。

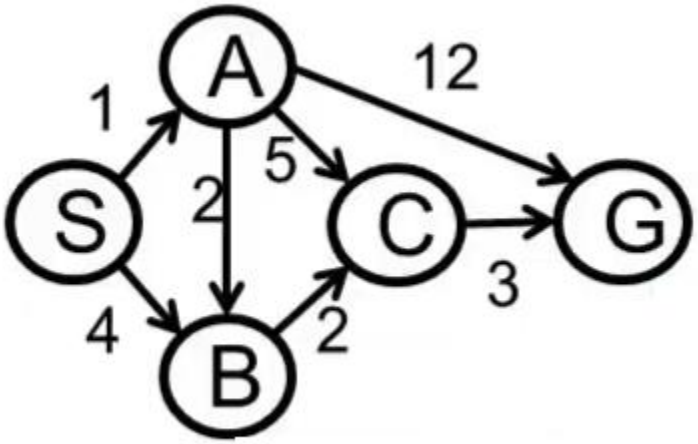
- 关键问题
  - 如何构造满足要求的h(n)

5.4.5 *A**搜索算法实例

h(n)构造示例

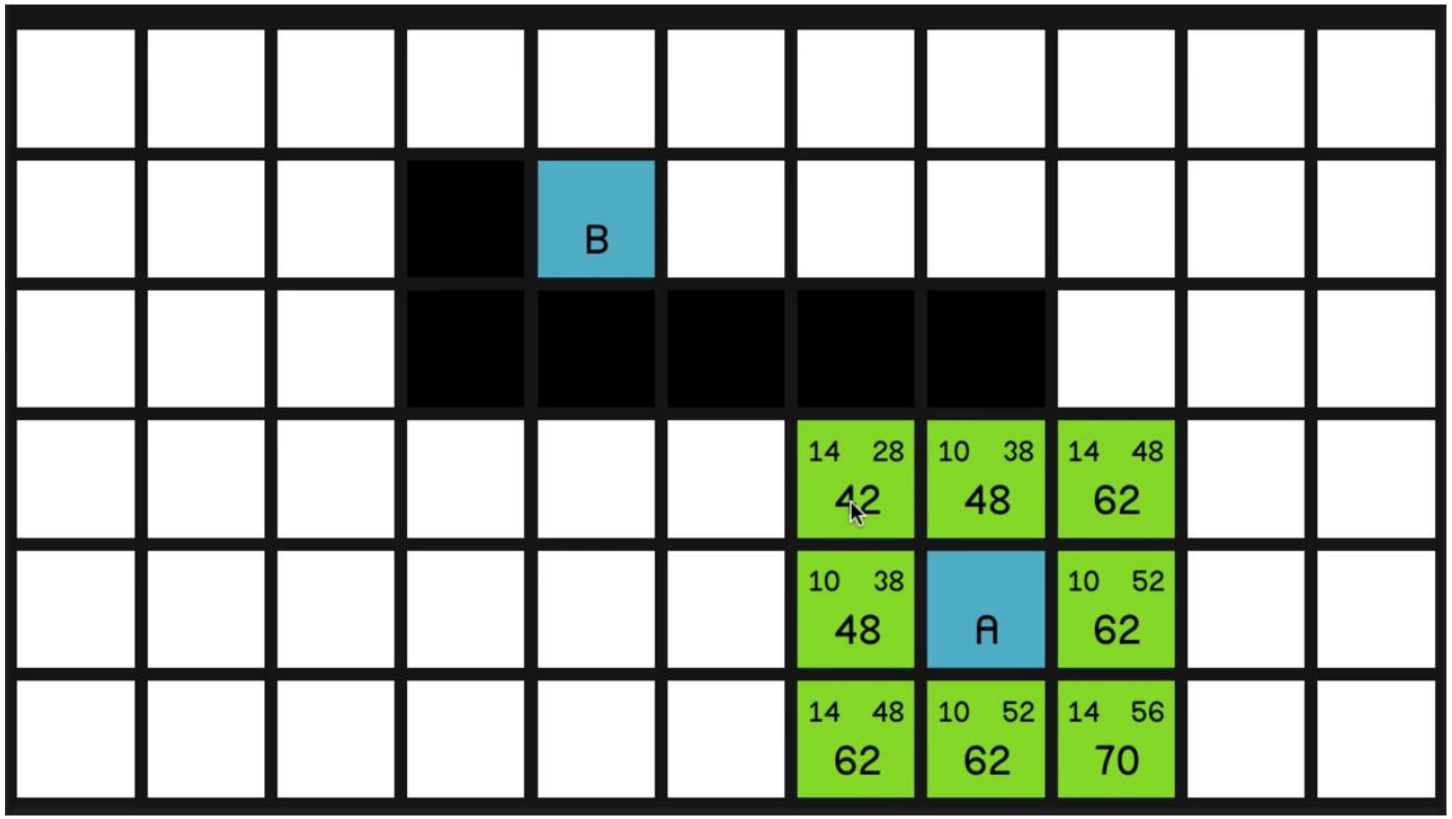
- 每个节点（或者状态）State 都有一个H值，这个值相当于对“从该点出发到终点的距离”的一个大概的估计。
- 这个启发值不能够大于实际值。

例如，G点本来就是终点，所以它对应的H就是0。A点到G点的最短路径应该是8，所以启发值就设为一个不大于8的值，例如6。



State	H
S	7
A	6
B	2
C	1
G	0

5.4.5 *A**搜索算法实例

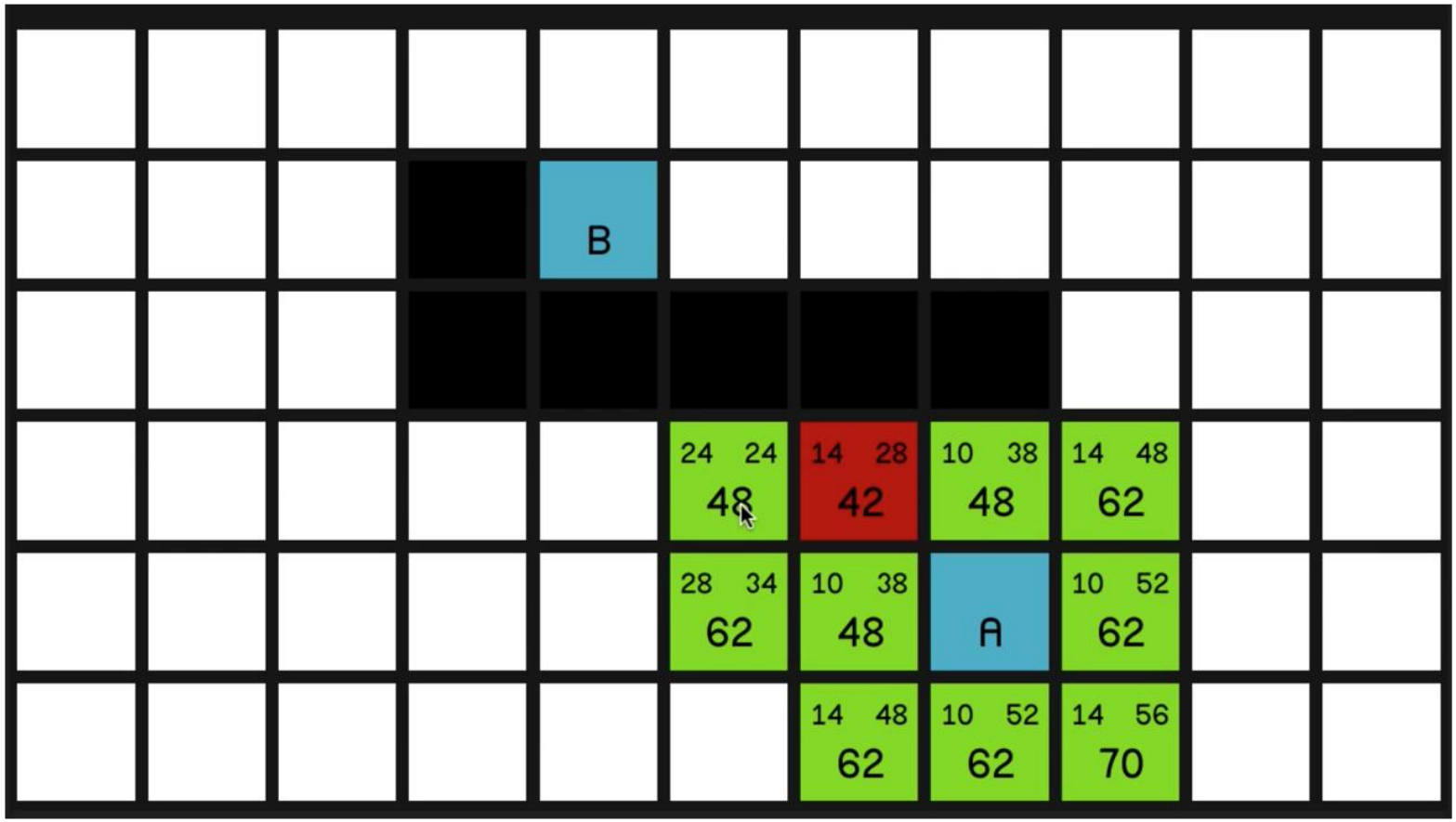


1、A表示起点，B表示终点，黑色的实心方块表示障碍物。

2、假设水平或垂直方向上相邻的两个方块之间距离是10，那么对角线方向上相邻的两个方块距离就约是14。

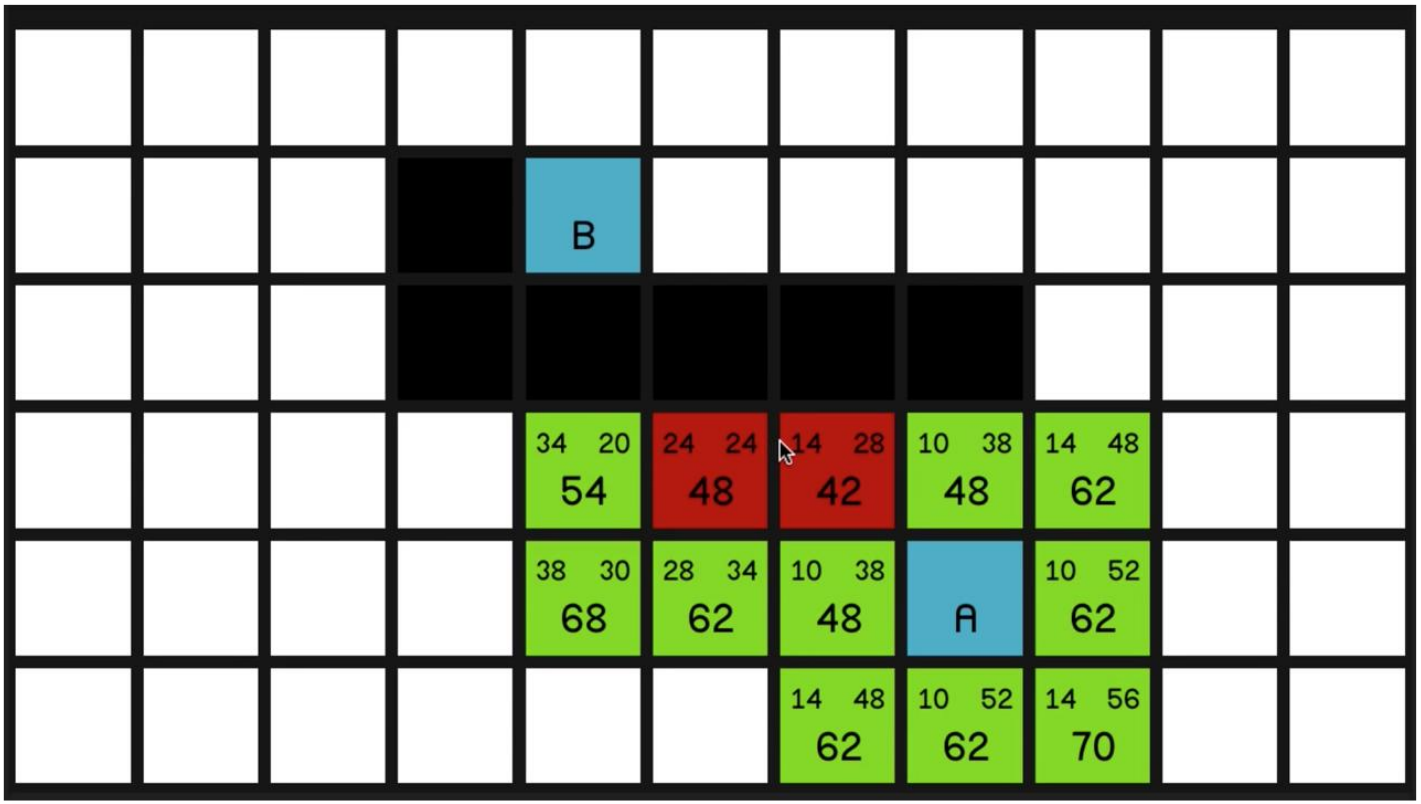
- 首先搜索A相邻的所有可能的移动位置（图中的绿色方块）。
- 每个方块左上角的值G表示该点到A的距离，右上角值为H，注意H不能大于该点到B的距离，所以这里的H就取其到B的距离。
- 最后，还要计算一个F值，F=H+G。

5.4.5 *A**搜索算法实例



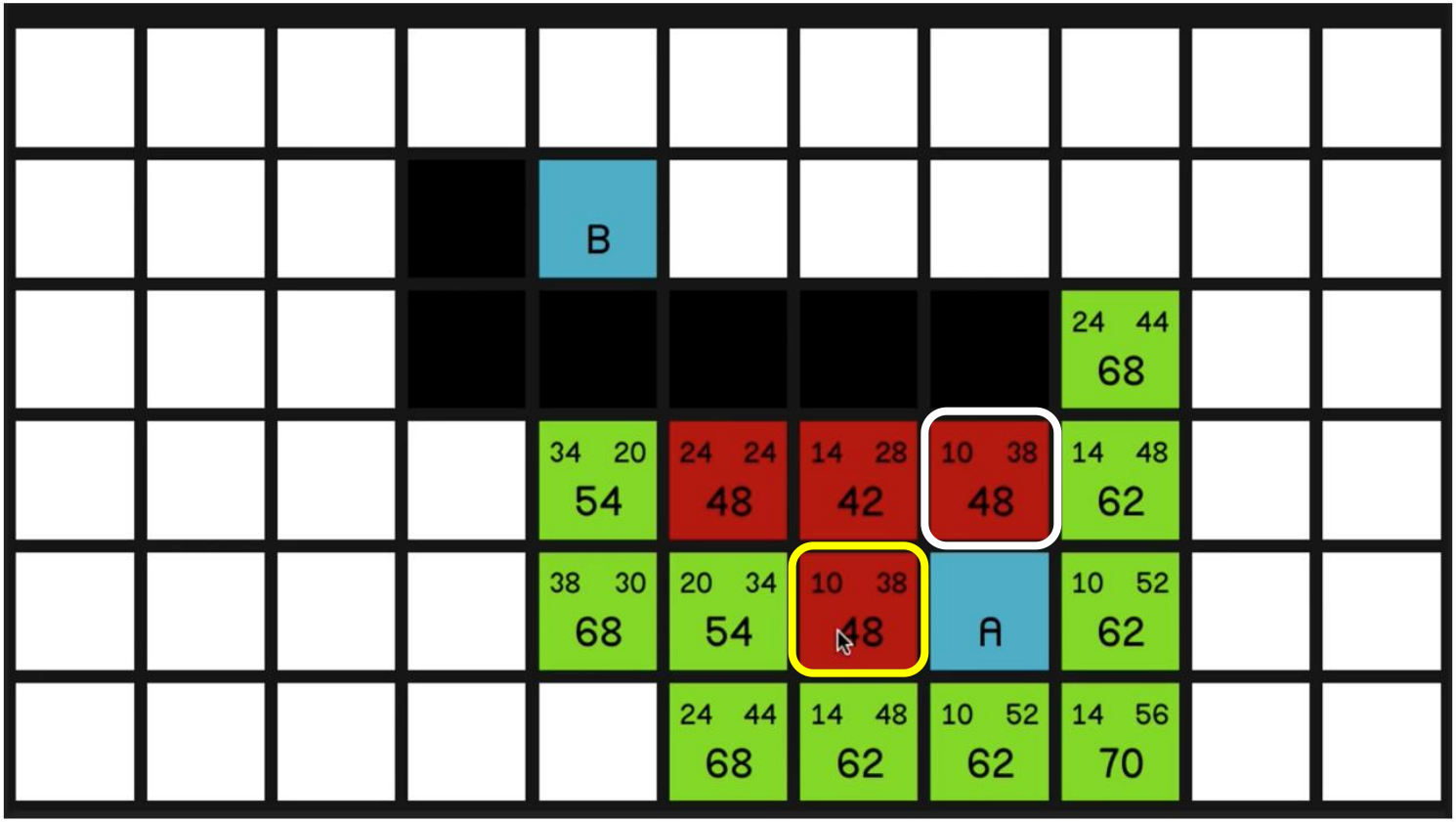
- 图中A的邻域中位于左上角的值（F=42）是最小值。
- 选中该节点，并更新该节点的邻域值。

5.4.5 *A**搜索算法实例



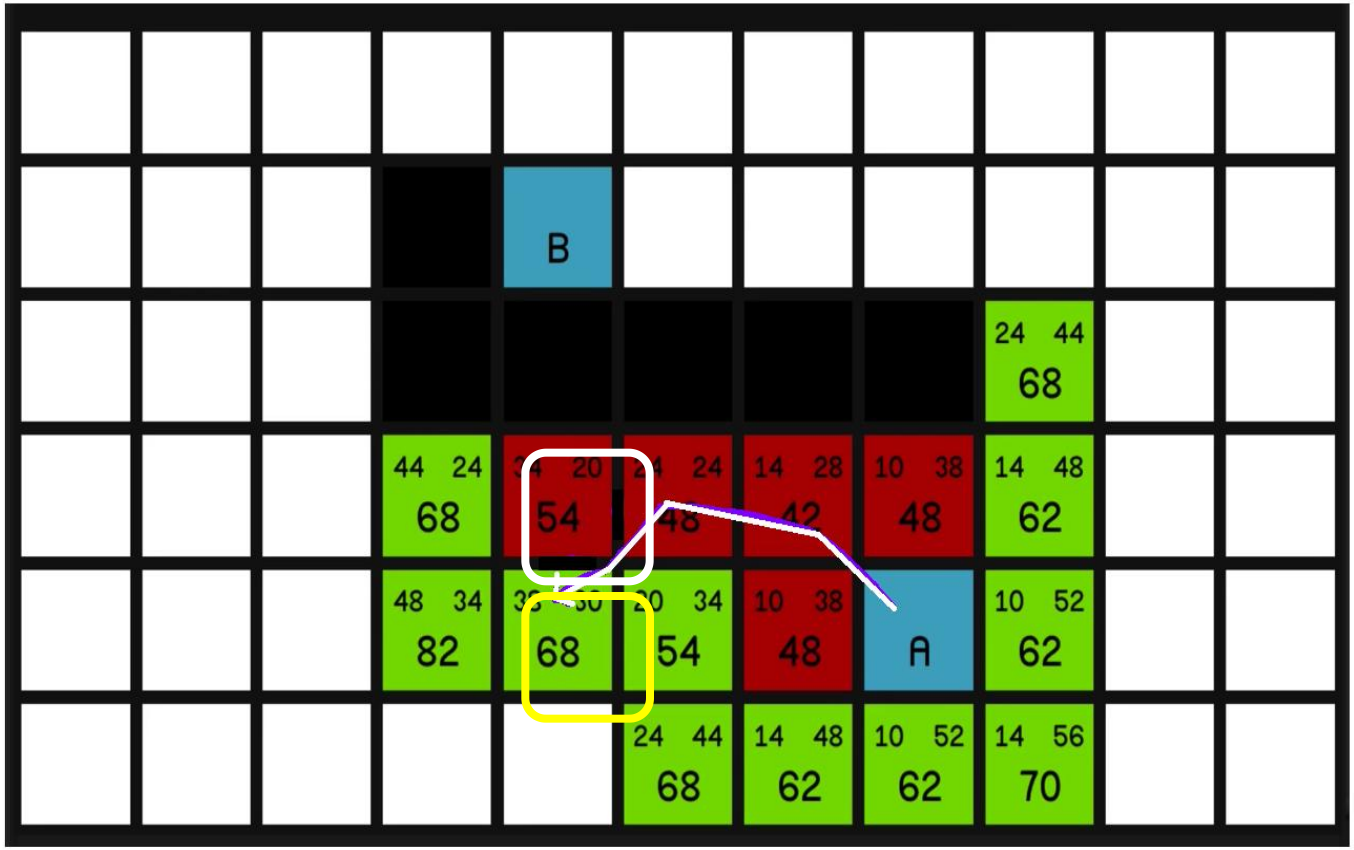
- 出现了三个F值都等于48的节点。到底应该选择哪一个来继续接下来的搜索呢？
- 这时需要考察它们中的那个H值最小，结果发现H=24是最小的，所以下面就要从该点出发继续搜索。
- 更新该节点的邻域方块中的值。

5.4.5 *A**搜索算法实例



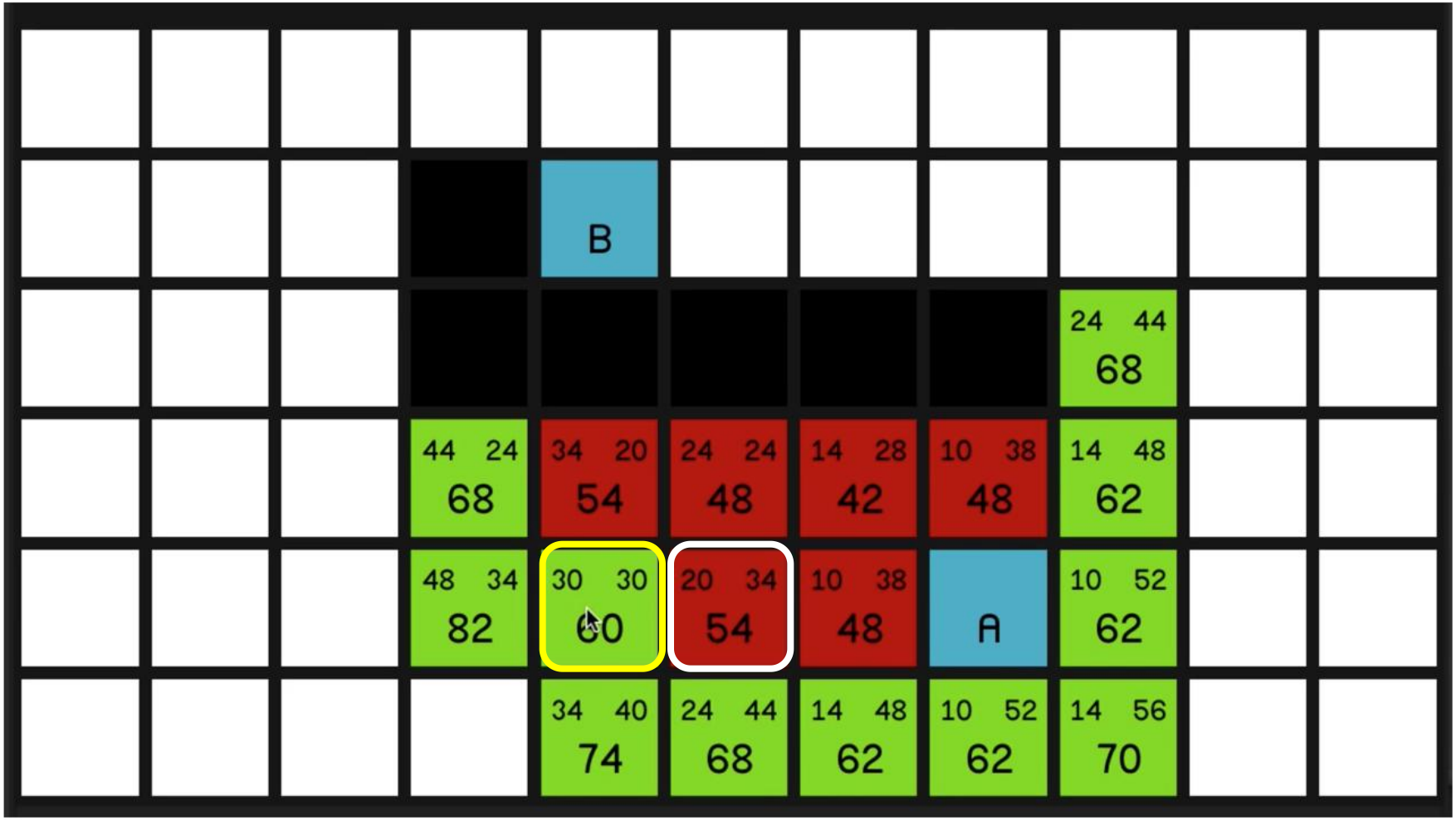
- 找出全局F值最小的点，结果发现有两个为48（G和H均相等）。随机选取一个，例如选择右上方的方块。
- 更新其邻域值。
- 再找出全局F值最小的点（位于下方的F=48的节点），更新邻域值。

5.4.5 *A**搜索算法实例



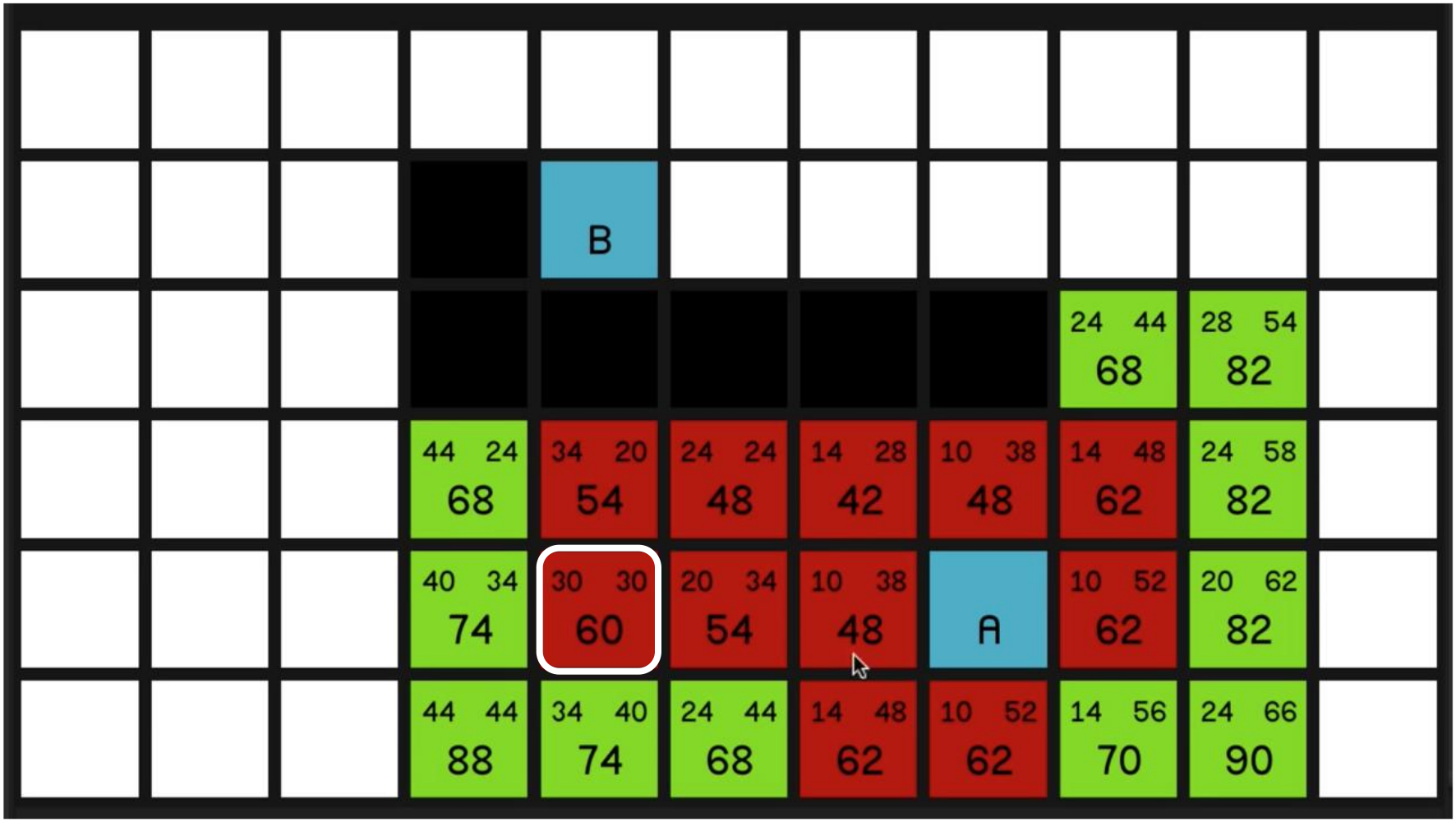
- 此时全局F最小的值为54，而且F=54的节点有两个，所以我们还是选择其中H值最小的来做更新。
- 更新该节点邻域方块中的值。
- F=54的红色节点下方邻域（F=68）的方块中，G=38（14→24→38）。但是，从A到该节点的最短路径应该是30。咋处理？后面会自动纠正的！

5.4.5 *A**搜索算法实例



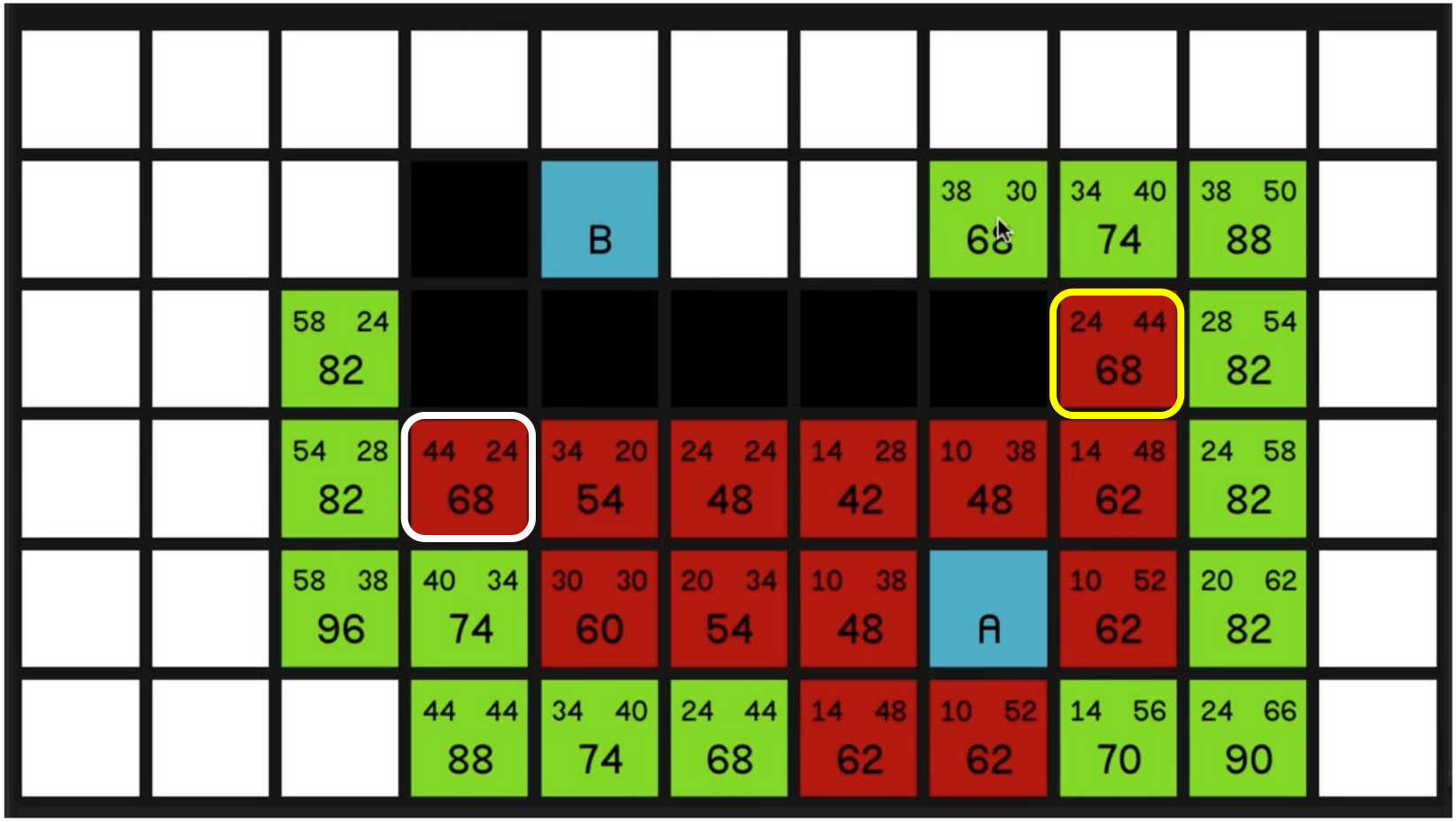
- 更新全局最小节点（**F=54**）的方块。
- 前面提到的邻节点（黄色框）**G**值更新为正确的值。

5.4.5 *A**搜索算法实例



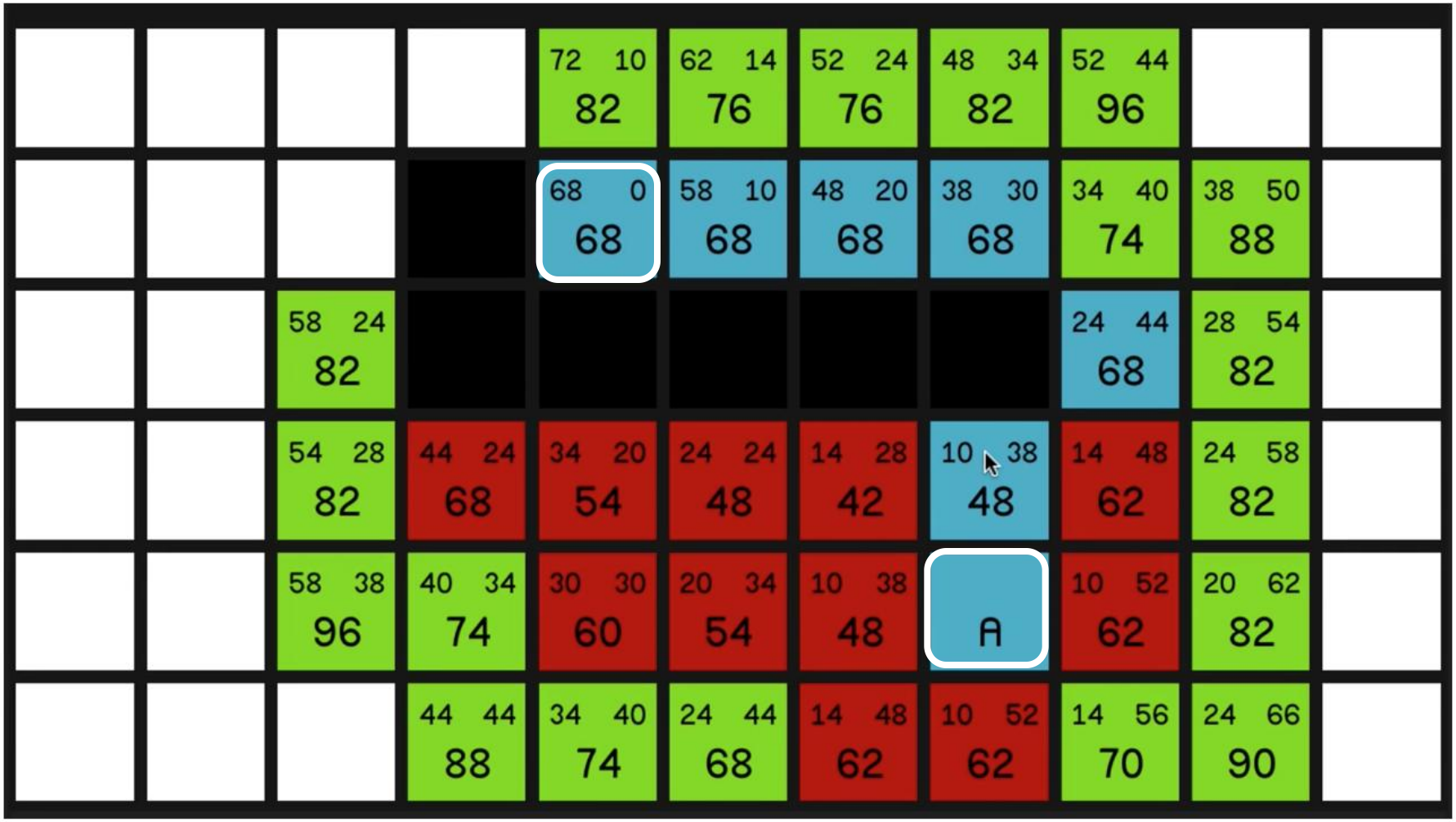
- 更新全局最小节点（**F=60**）的方块。

5.4.5 *A**搜索算法实例



- 全局最小值F=68，此时先更新H最小的。
- 继续搜索，转向右侧F=68的节点。

5.4.5 *A**搜索算法实例



- 不停反复迭代搜索
- 蓝色方块构成了最终的路径

THE END