



第 11 章 广度优先搜索


11-1 广度优先搜索

11.1.1 广度优先搜索的设计思想

 **广度优先搜索 (breadth-first search)** 以顶点 u 为起始点, 依次访问和 u 有路径相通且路径长度为 1、2、... 的顶点。广度优先搜索的基本思想是: 访问顶点 u , 然后依次访问 u 的各个未被访问的邻接点 v_1 、 v_2 、...、 v_k , 再分别从 v_1 、 v_2 、...、 v_k 出发依次访问它们未被访问的邻接点, 直至图中所有与顶点 u 有路径相通的顶点都被访问到。

 为了使 “**先被访问顶点的邻接点**” 先于 “**后被访问顶点的邻接点**” 被访问, 设置**队列**存储已被访问的顶点。

11.1.1 广度优先搜索的设计思想

 为避免重复搜索，将经过的顶点标记为已访问，显然，初始时所有顶点均未被访问。广度优先搜索的算法思想用伪代码描述如下：

算法：BFS

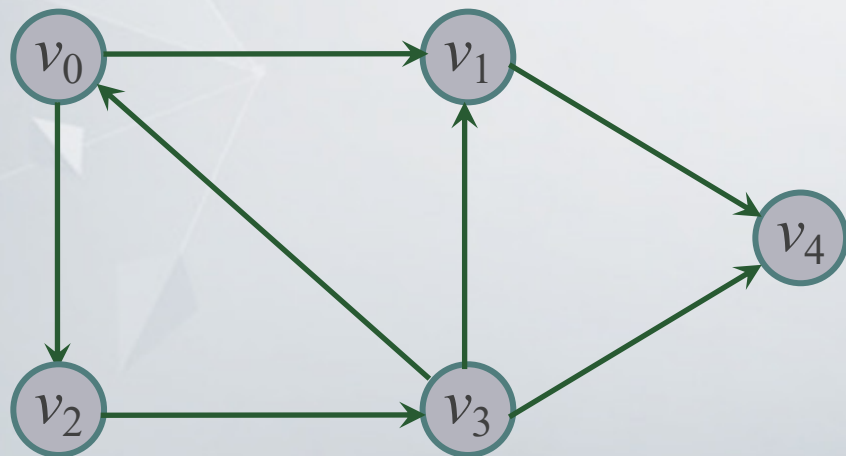
输入：起始顶点 u

输出：搜索经过的顶点序列

1. 队列 Q 初始化;
2. 访问顶点 u ; 修改标志 $\text{visited}[u]=1$; 顶点 u 入队列 Q ;
3. 当队列 Q 非空时执行下述操作:
 - 3.1 u = 队列 Q 的队头元素出队;
 - 3.2 v = 顶点 u 的邻接点;
 - 3.3 重复下述操作直至 v 不存在:
 - 3.3.1 如果 v 未被访问，则
访问顶点 v ; 修改标志 $\text{visited}[v]=1$; 顶点 v 入队列 Q ;
 - 3.3.2 v = 顶点 u 的下一个邻接点;

11.1.1 广度优先搜索的设计思想

 运行实例——深入理解操作过程

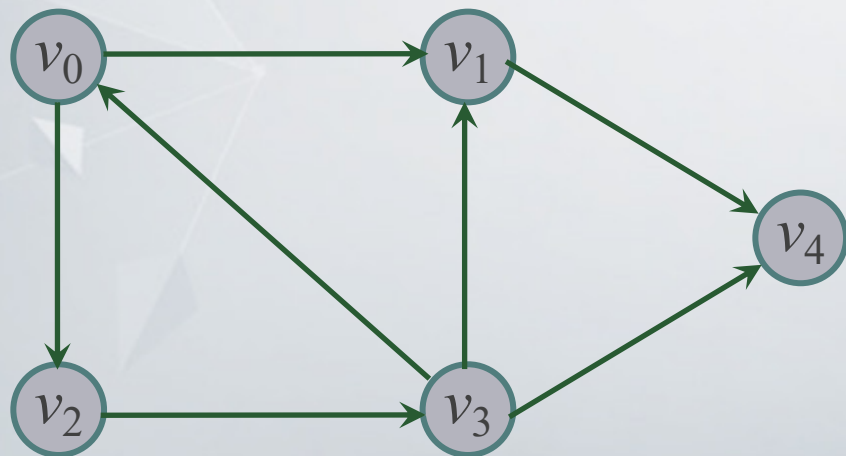


广度优先遍历序列: v_0 v_1 v_2

v_0	v_1	v_2
-------	-------	-------

11.1.1 广度优先搜索的设计思想

 运行实例——深入理解操作过程

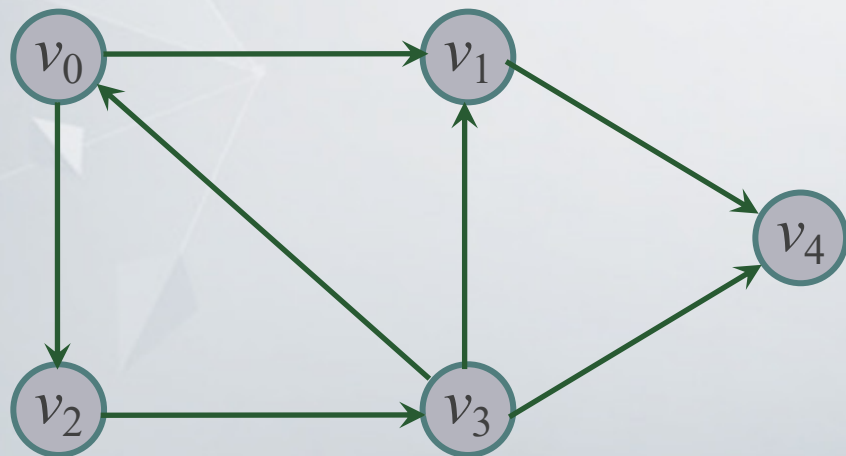


v_1 v_2 v_4 v_3

广度优先遍历序列: v_0 v_1 v_2 v_4 v_3

11.1.1 广度优先搜索的设计思想

 运行实例——深入理解操作过程



$v_4 \quad v_3$

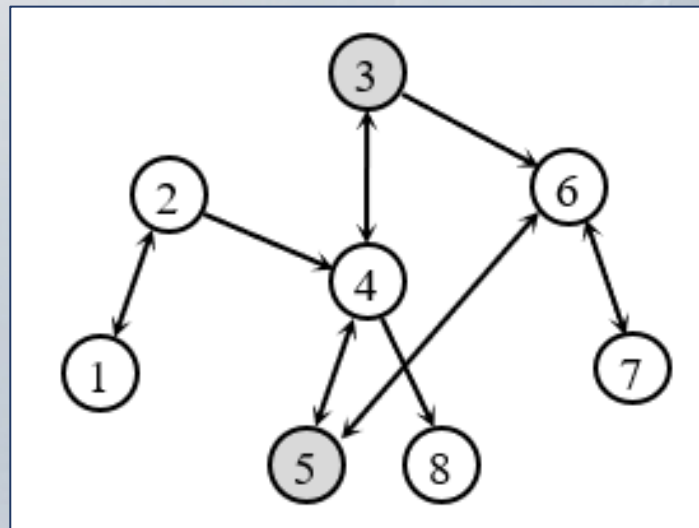
广度优先遍历序列: v_0 v_1 v_2 v_4 v_3

11.1.2 农夫抓牛

【问题】 假设农夫和牛都位于数轴上，农夫位于点 N ，牛位于点 K ($K > N$)，农夫有以下两种移动方式：（1）从点 X 移动到 $X-1$ 或 $X+1$ ，每次移动花费一分钟；（2）从点 X 移动到点 $2X$ ，每次移动花费一分钟，假设牛没有意识到农夫的行动，站在原地不动，农夫最少要花费多长时间才能抓住牛？

【想法】 这是一个最少步数问题，适合用广度优先搜索。将数轴上每个点看作是图的顶点，对于任意点 X ，有两条双向边连到点 $X-1$ 和 $X+1$ ，有一条单向边连到点 $2X$ ，则农夫抓牛问题转化为求从顶点 N 出发到顶点 K 的最短路径长度。

假设 $N=3$ ， $K=5$ ，广度优先搜索过程如右图所示，最短路径长度是 2。



【算法】 设数组 $\text{flag}[2*K]$ 表示数轴上某个点是否被搜索，设变量 right 表示每一层最后搜索的顶点，变量 rear 指向队列的队尾位置， front 指向队列队头的前一个位置， steps 表示待扩展结点距起点 N 的步数，算法如下：

算法：农夫抓牛CatchCattle

输入：农夫的位置 N ，牛位的位置 K ，

输出：最少步数

1. 队列 Q 初始化；初始化 $\text{flag}[2*K] = \{0\}$ ；
2. 将起点 N 放入队列 Q ，修改标志 $\text{flag}[N] = 1$ ， $\text{right} = \text{rear}$ ；
3. 当队列 Q 非空时执行下述操作：
 - 3.1 u = 队列 Q 的队头元素出队；
 - 3.2 如果 u 等于 K ，输出到达顶点 u 的步数 steps ，算法结束；
 - 3.3 依次扩展结点 u 的每个子结点：
 - 3.3.1 $v = u-1$ ；如果 $\text{flag}[v]$ 等于0，将 v 入队，修改标志 $\text{flag}[v] = 1$ ；
 - 3.3.2 $v = u+1$ ；如果 $\text{flag}[v]$ 等于0，将 v 入队，修改标志 $\text{flag}[v] = 1$ ；
 - 3.3.3 $v = u+u$ ；如果 $\text{flag}[v]$ 等于0，将 v 入队，修改标志 $\text{flag}[v] = 1$ ；
 - 3.4 如果 front 等于 right ，则 $\text{steps}++$ ； $\text{right} = \text{rear}$ ；
4. 队列为空，没有到达位置 K ，返回失败标志-1；

11.1.2 农夫抓牛

【算法实现】 设队列 Q 采用顺序队列，简单起见，假定队列 Q 不会发生溢出，如果当前出队结点是本层最后搜索的顶点，则步数steps加 1，并且接下来入队的结点应该是这层的最后顶点，调整变量right。程序如下：

```
int CatchCattle(int N, int K)
{
    int u, v, right = 0, steps = 0, flag[K] = {0};
    int Q[K], front, rear;
    front = rear = -1;
    Q[++rear] = N; flag[N] = 1;
    while (front != rear)
    {
        u = Q[++front];
        if (u == K) return steps;
        v = u - 1; if (flag[v] == 0) { Q[++rear] = v; flag[v] = 1; }
```

11.1.2 农夫抓牛

【算法实现】 设队列 Q 采用顺序队列，简单起见，假定队列 Q 不会发生溢出，如果当前出队结点是本层最后搜索的顶点，则步数 $steps$ 加 1，并且接下来入队的结点应该是这层的最后顶点，调整变量 $right$ 。程序如下：

```
v = u + 1; if (flag[v] == 0) { Q[++rear] = v; flag[v] = 1; }  
v = u + u; if (flag[v] == 0) { Q[++rear] = v; flag[v] = 1; }  
if (front == right) { steps++; right = rear; }  
}  
return -1;  
}
```

11.1.3 骑士旅行

【问题】 在一个国际象棋的棋盘上，给定起点 (x_1, y_1) 和终点 (x_2, y_2) ，计算骑士从起点到终点最少需要移动的步数。

【想法】 国际象棋的骑士走“日”字并且可以越子，按照骑士的走步规则，骑士每次移动的位置增量有八个方向。骑士旅行问题属于寻找最短路径问题，可以用广度优先搜索求解。

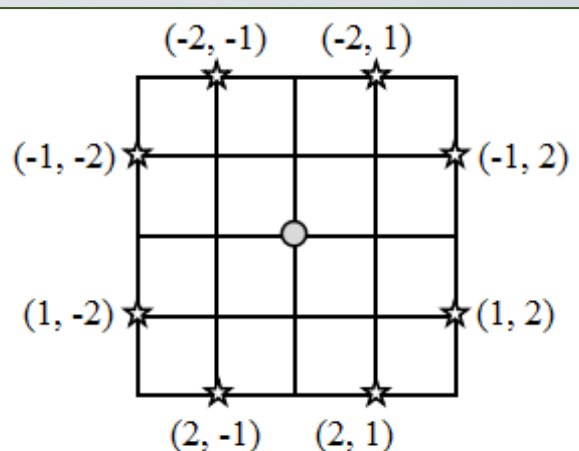
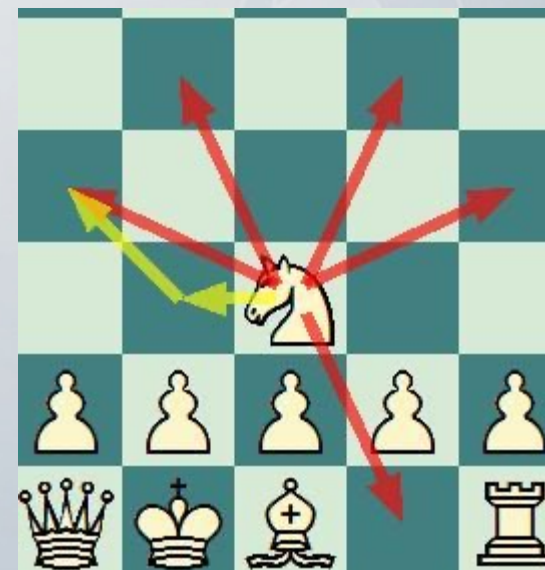


图 11-4 骑士移动的位置增量

11.1.3 骑士旅行

【算法】 设 nxt 表示下一步的位置， $\text{flag}(i)$ 表示位置 i 是否走过，算法如下：

算法：计算骑士问题BfsKnight

输入：起点的位置 $s(x_1, y_1)$ ，终点的位置 $t(x_2, y_2)$

输出：最少步数

1. 队列 Q 初始化;
2. 将起点 s 放入队列 Q ，修改标志 $\text{flag}(s) = 1$;
3. 当队列 Q 非空时执行下述操作：
 - 3.1 u = 队列 Q 的队头元素出队;
 - 3.2 如果 u 等于 t ，输出到达顶点 u 的步数，算法结束;
 - 3.3 循环变量 k 从 0~7 从 8 个方向扩展：
 - 3.3.1 $\text{nxt} = u$ 的下一步;
 - 3.3.2 如果 nxt 在棋盘内且未走过，则将 nxt 放入队列 Q ；修改标志 $\text{flag}(\text{nxt}) = 1$;
4. 队列为空，没有到达终点 t ，返回失败标志 -1;

11.1.3 骑士旅行

【算法实现】 设队列 Q 采用顺序队列，假定队列 Q 不会发生溢出，如果当前出队结点是本层最后搜索的顶点，则步数steps加 1，并且接下来入队的结点应该是这层的最后顶点，调整变量right。程序如下：

```
struct status
{
    int x, y, steps;
};
int dx[8] = {-2, -1, 1, 2, 2, 1, -1, -2};
int dy[8] = {-1, -2, -2, -1, 1, 2, 2, 1};

int BfsKnight(status s, status t)
{
    int k, front, rear, flag[8][8] = {0};
    status u, v, nxt, Q[64];
    front = rear = -1;
```





11.1.3 骑士旅行

```
Q[++rear] = s; flag[s.x][s.y] = 1;
while (front != rear)
{
    u = Q[++front];
    if ((u.x == t.x) && (u.y == t.y)) return u.steps;
    for (k = 0; k < 8; k++)
    {
        nxt.x = u.x + dx[k]; if (nxt.x < 0 || nxt.x > 7) continue;
        nxt.y = u.y + dy[k]; if (nxt.y < 0 || nxt.y > 7) continue;
        if (flag[nxt.x][nxt.y] == 0)
        {
            nxt.steps = u.steps + 1;
            Q[++rear] = nxt; flag[nxt.x][nxt.y] = 1;
        }
    }
}
return -1;
}
```

第 11 章 广度优先搜索

11-2 A*算法

11.2.1 A^* 算法的设计思想

-  A^* (A -Star) 算法是求最短路径非常有效的一种搜索方法，也是解决搜索问题常用的启发式算法。
-  所谓启发式搜索是通过启发式函数 (heuristic function, 也称估价函数) 引导算法的搜索方向，以达到减少搜索结点的目的。
-  启发式函数通常利用与问题有关的某种启发信息，对于路径搜索问题，结点就是搜索空间的状态，启发信息通常是距离，启发式函数表示为：

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

其中， $f(n)$ 是从初始状态到目标状态的估计代价；

$g(n)$ 是从初始状态到状态 n 的实际代价；

$h(n)$ 是从状态 n 到目标状态最佳路径的估计代价。

11.2.1 A*算法的设计思想

 A*算法以**优先队列**形式组织的**open表**，存储搜索过程中经过的状态，每次从open表中选取 $f(n)$ 值最小（或最大）的结点作为下一个待扩展的结点。


算法：A*算法

输入：问题模型，启发式函数

输出：最优值

1. 将起始状态加入open表中；
2. 重复下述操作直到open表为空：
 - 2.1 $n = \text{open}$ 表中第一个状态结点；
 - 2.2 如果结点 n 为终点，则返回最优值，算法结束；
 - 2.3 结点 n 不是终点，执行下述操作：
 - 2.3.1 将结点 n 从open表中删除；
 - 2.3.2 生成结点 n 的所有子结点；
 - 2.3.3 计算所有子结点的估价函数值；
 - 2.3.4 将所有子结点加入open表中，并按估价函数排序；
3. 搜索失败；

11.2.1 A^* 算法的设计思想

 A^* 算法能够尽快找到或最优解的关键在于函数 $h(n)$ 的选取，通过预估 $h(n)$ 的值，降低搜索走弯路的可能性，加快搜索速度。当 $h(n)$ 始终为 0，则由 $g(n)$ 决定结点的优先级，此时退化为Dijkstra算法。以 $d(n)$ 表示状态 n 到目标状态的最短距离，则 $h(n)$ 的选取大致有如下三种情况：

(1) $h(n) < d(n)$ ，搜索的结点数较多，搜索范围较大，效率较低，但能保证得到最优解。 $h(n)$ 值越小，搜索越多的结点，也就导致算法效率越低。

(2) $h(n) = d(n)$ ，搜索将严格沿着最短路径进行，此时的搜索效率最高。在没有达到终点之前，很难确切计算出距离终点还有多远。

(3) $h(n) > d(n)$ ，搜索的结点数较少，搜索范围较小，效率较高，但不能保证得到最优解。因为启发仅仅是下一步将要采取措施的一个猜想，这个猜想常常根据经验和直觉来判断，所以启发式搜索可能出错。

11.2.2 八数码问题

【问题】八数码问题 (8-puzzle problem) 也称重排九宫问题，在一个 3×3 的方格盘上，放有 1~8 个数码，余下一格为空格，空格四周上下左右的数码可以移动到空格。给定一个八数码问题的初始状态，要求找到一个移动序列到达目标状态。

2	8	3
1	6	4
7		5

(a) 初始状态

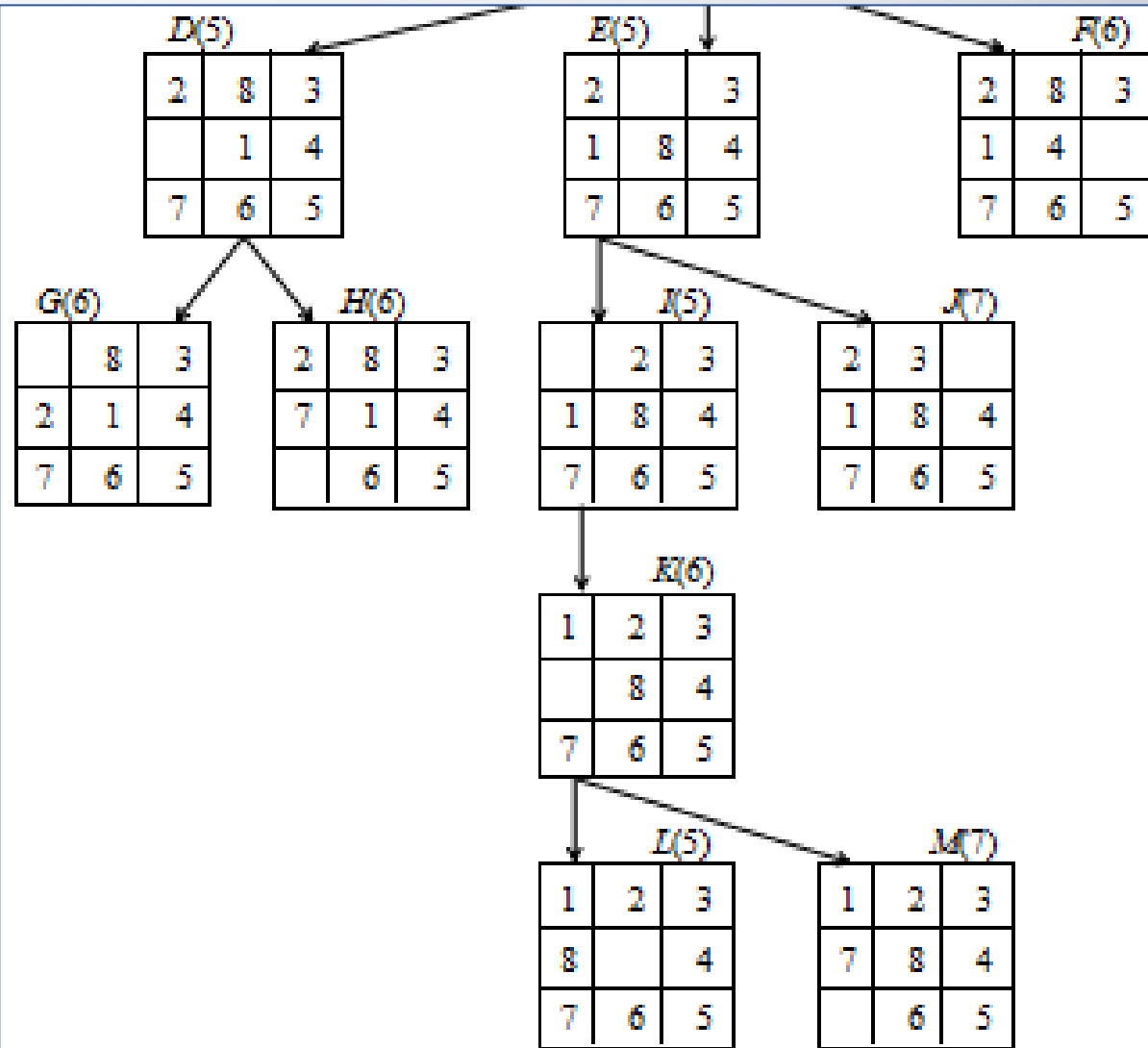
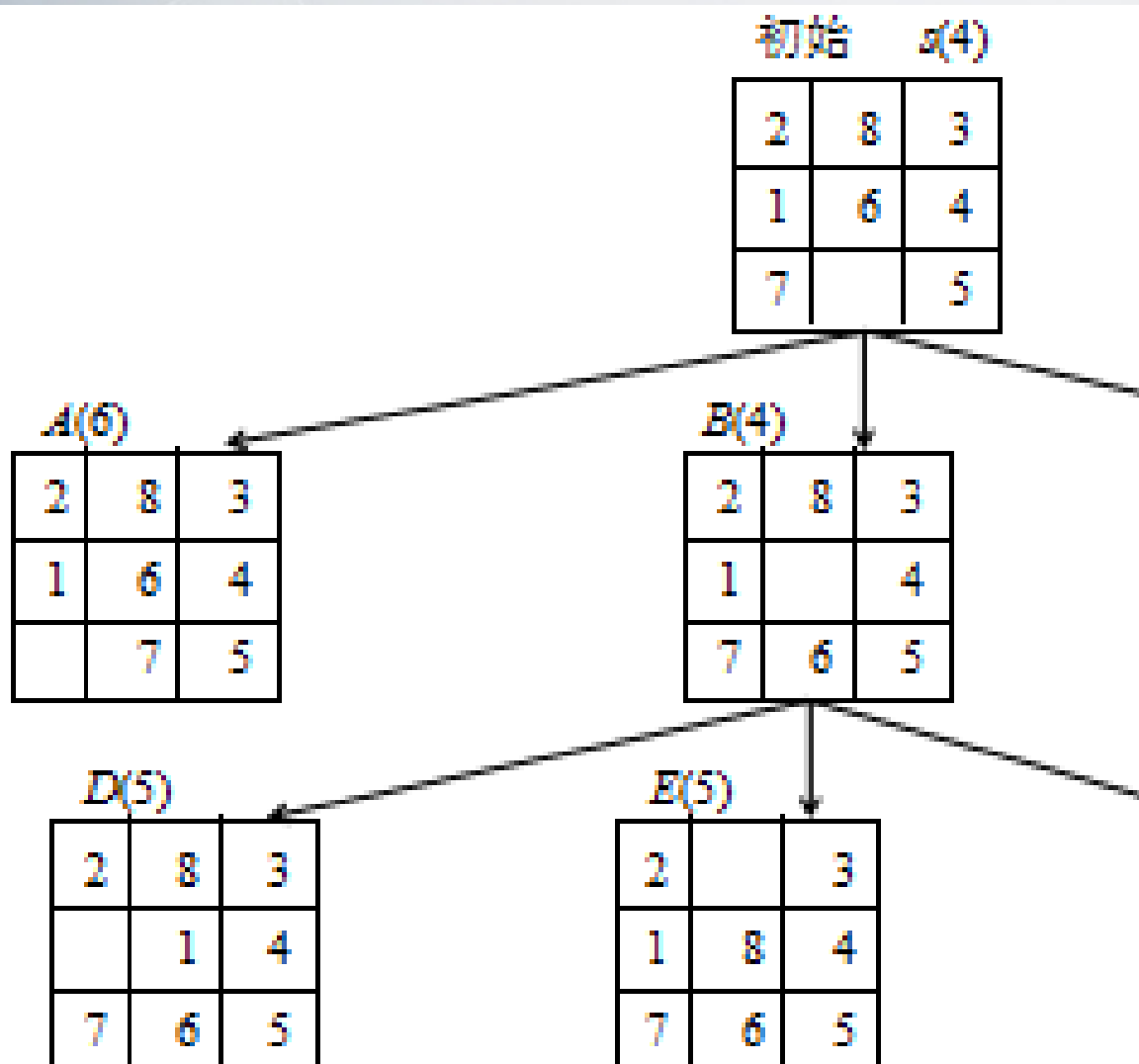
1	2	3
8		4
7	6	5

(b) 目标状态

图 11-5 八数码问题

【想法】应用 A^* 算法求解八数码问题的关键是确定启发式函数，可以将实际代价函数 $g(n)$ 定义为解空间树中从根结点到该状态的路径长度（即移动次数），估计代价函数 $h(n)$ 定义为该状态与目标状态不相符的数码个数。

11.2.2 八数码问题



11.2.3 多段图的最短路径问题

【问题】 设 $G=(V, E)$ 是一个带权有向连通图，如果把顶点集合 V 划分成 k 个互不相交的子集 V_i ($2 \leq k \leq n, 1 \leq i \leq k$)，使得 E 中的任一条边 (u, v) ，必有 $u \in V_i, v \in V_{i+m}$ ($1 \leq i < k, 1 < i+m \leq k$)，称 $s \in V_1$ 为源点， $t \in V_k$ 为终点。

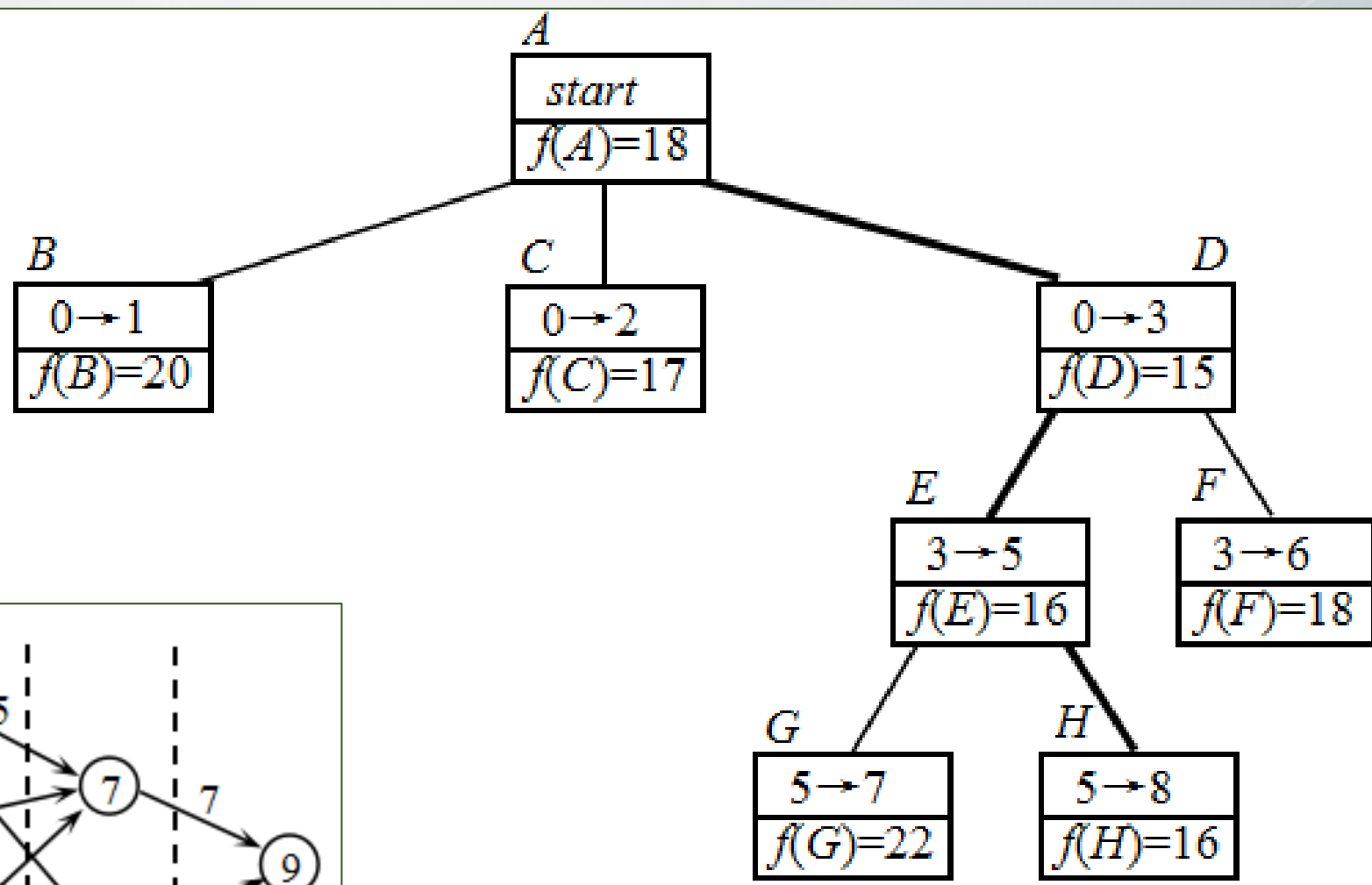
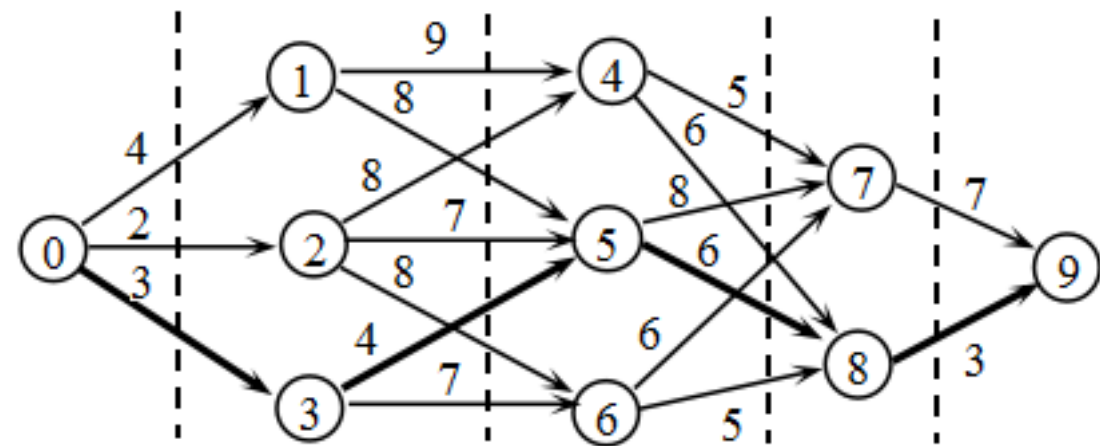
多段图的最短路径问题 (multi-segment graph shortest path problem) 是求从源点到终点的最小代价路径。

【想法】 可以将 $g(i)$ 定义为从源点到顶点 i 的实际路径长度，将 $h(i)$ 定义为从顶点 i 到终点每一段的最小代价之和。一般情况下，假设当前已经确定了前 i 段 ($1 \leq i \leq k$)，路径为 (r_1, r_2, \dots, r_i) ，启发式函数定义如下：

$$g(i) = \sum_{j=1}^i c[r_{j-1}][r_j]$$

$$h(i) = \min\{c[r_i][r_p]\} + \sum_{j=i+2}^k \text{第} j \text{段的最短边}$$

11.2.3 多段图的最短路径问题



11.2.4 任务分配问题

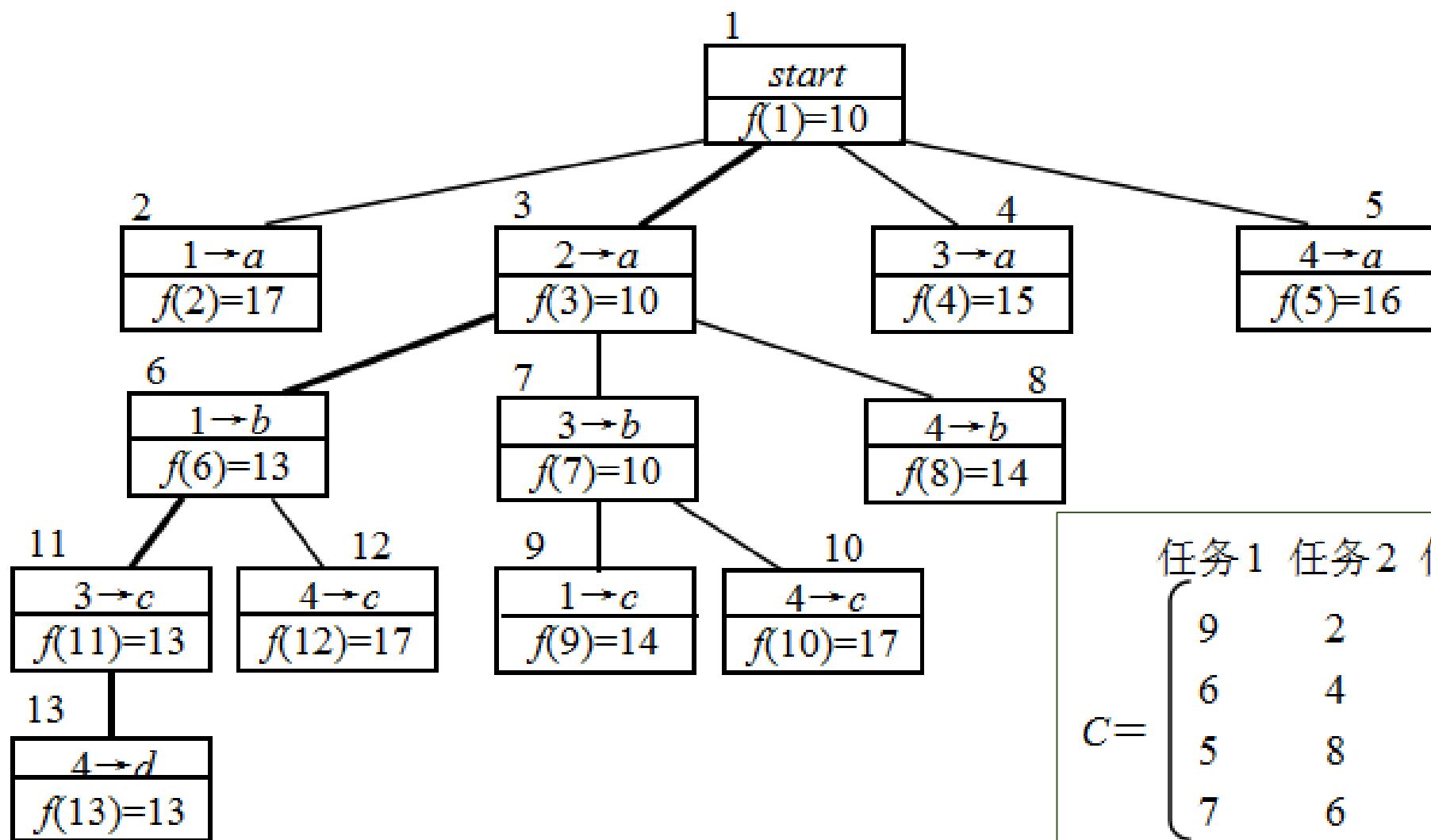
【问题】把 n 项任务分配给 n 个人，每个人完成每项任务的成本不同，**任务分配问题** (task allocation problem) 要求总成本最小的最优分配方案。

【想法】将 $g(n)$ 定义为已经分配的任务成本， $h(n)$ 定义为其余任务的最小分配成本。任务分配问题可以采用成本矩阵表示，令 c_{ij} 表示人员 i 分配了任务 j ($1 \leq i, j \leq n$)，一般情况下，假设当前已对人员 $1 \sim i$ 分配了任务，启发式函数定义如下：

$$g(i) = \sum_{k=1}^i c_k x_k$$

$$h(i) = \sum_{k=i+1}^n \text{第} k \text{行的最小值}$$

11.2.4 任务分配问题



任务 1	任务 2	任务 3	任务 4	
9	2	7	8	人员 a
6	4	3	7	人员 b
5	8	1	8	人员 c
7	6	9	4	人员 d

$C =$


第 11 章 广度优先搜索

11-3 限界剪枝法

11.3.1 限界剪枝法的设计思想

- ✎ 限界剪枝法 (bound and pruning method) 首先确定一个合理的**限界函数** (bounding function), 并根据限界函数确定目标函数的界[down, up]。然后, 按照**广度优先**策略搜索问题的解空间树。
- ✎ 在**分支结点**上, 依次扩展该结点的所有孩子结点, 分别估算这些孩子结点的目标函数值, 如果某孩子结点的目标函数值**超出目标函数的界**, 则将其**丢弃**, 因为从这个结点生成的解不会比目前已经得到的解更好; 否则, 将其**加入open表**中。依次从open表中选取**使目标函数取得极值的结点**成为当前扩展结点, 重复上述过程, 直至找到最优解。
- ✎ 因为限界函数常常基于问题的目标函数而确定, 所以, 限界剪枝法适用于求解**最优化问题**。

11.3.1 限界剪枝法的设计思想

 实质上，限界剪枝法是在 A^* 算法的基础上加入剪枝操作，减少open表中的结点数量，从而提高搜索效率。

算法：限界剪枝法

输入：问题模型，限界函数

输出：最优值

1. 根据限界函数确定目标函数的界[down, up];
2. 估算根节点的目标函数值并加入open表;
3. 循环直到某个叶子节点的目标函数值在open表中取得极值:
 - 3.1 $n = \text{open}$ 表中具有极值的结点;
 - 3.2 对结点 n 的所有子结点 x 执行下述操作:
 - 3.2.1 估算结点 x 的目标函数值value;
 - 3.2.2 若value在[down, up]中，则将结点 x 加入open表；否则丢弃结点 x ;
4. 输出叶子结点对应的最优值，回溯求得最优解的各个分量;

11.3.2 0/1 背包问题

【问题】 给定 n 种物品和一个容量为 C 的背包，物品 i 的重量是 w_i ，其价值为 v_i ，对每种物品只有两种选择：装入背包或不装入背包，**0/1背包问题** (0/1 knapsack problem) 是如何选择装入背包的物品，使得装入背包中物品的总价值最大？

【想法】 假设 n 种物品已按单位价值由大到小排序，可以采用**贪心法**求解 0/1 背包问题的一个**下界**。

如何求得 0/1 背包问题的一个合理上界呢？考虑最好情况，背包中装入的全部是第 1 个物品且可以将背包装满，则可以得到一个非常简单的计算方法：

$$ub = C \times (v_1/w_1)。$$

11.3.2 0/1背包问题

例如，4个物品重量为(4, 7, 5, 3)，价值分别为(40, 42, 25, 12)，背包容量 $C=10$ 。

- 首先，将给定物品按单位重量价值从大到小**排序**。
- 应用**贪心法求得近似解**为(1, 0, 1, 0)，获得的价值为 65，这可以作为 0/1背包问题的下界。
- 考虑最好情况**，背包中装入的全部是第1个物品且可以将背包装满，则

$$ub = C \times (v_1/w_1) = 10 \times 10 = 100。$$

- 于是，得到了目标函数的界[65, 100]。

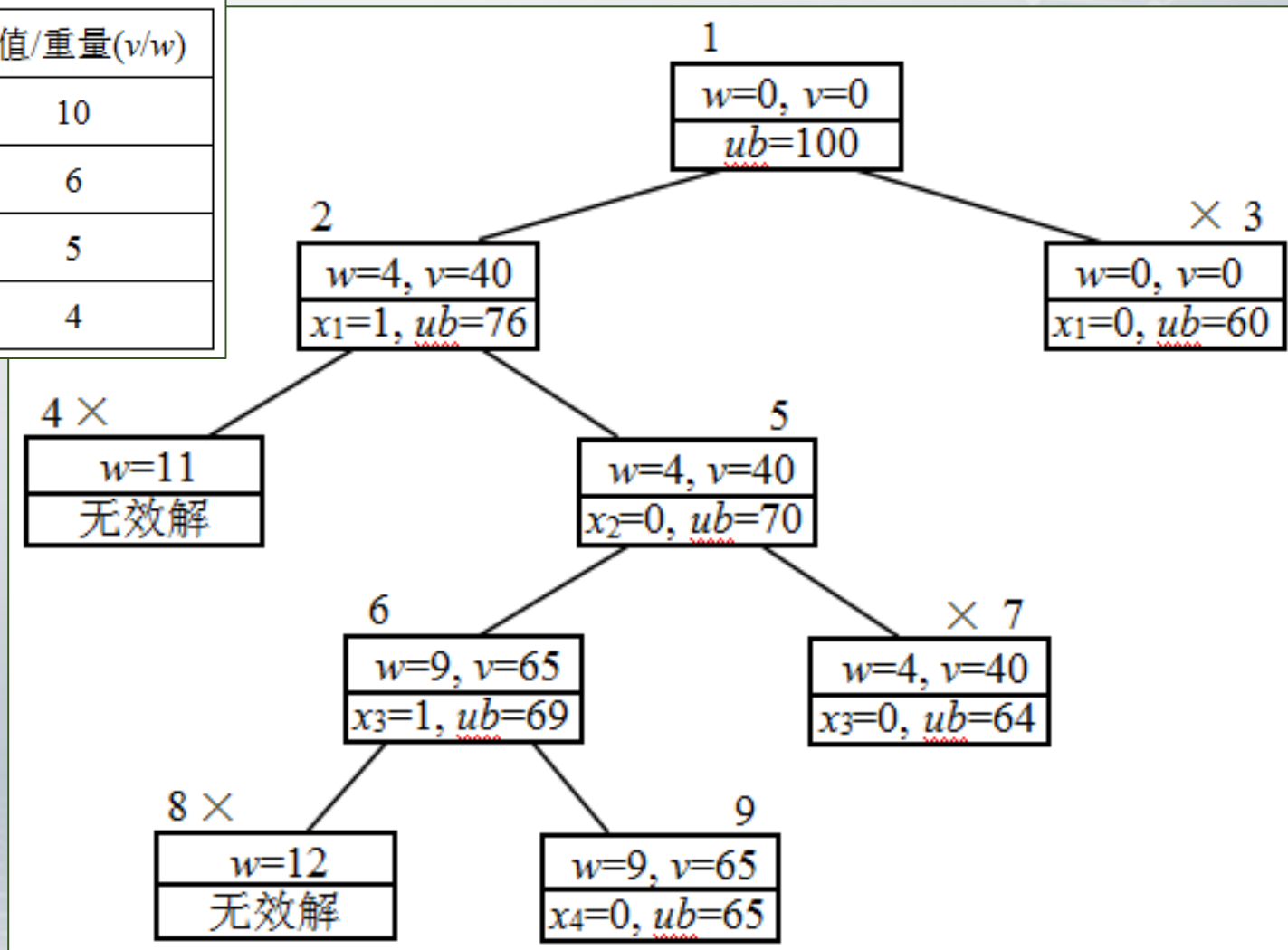
表 11-1 0/1 背包问题的价值/重量排序结果

物品	重量(w)	价值(v)	价值/重量(v/w)
1	4	40	10
2	7	42	6
3	5	25	5
4	3	12	4

11.3.2 0/1背包问题

表 11-1 0/1 背包问题的价值/重量排序结果

物品	重量(w)	价值(v)	价值/重量(v/w)
1	4	40	10
2	7	42	6
3	5	25	5
4	3	12	4



11.3.3 TSP问题

【问题】TSP问题 (traveling salesman problem) 是指旅行家要旅行 n 个城市, 要求各个城市经历且仅经历一次然后回到出发城市, 并要求所走的路程最短。

【想法】 首先确定目标函数的界[down, up], 采用贪心法确定TSP问题的一个上界。如何求得TSP问题的一个合理的**下界**呢? 对于无向图的代价矩阵, 把矩阵中每一行最小的元素相加, 可以得到一个简单的下界。

但是还有一个信息量更大的**下界**: 考虑TSP问题的一个完整解, 路径上每个城市都有两条邻接边, 一条是进入这个城市的, 另一条是离开这个城市的, 那么, **把矩阵中每一行最小的两个元素相加再除以 2**, 就得到了一个合理的下界。

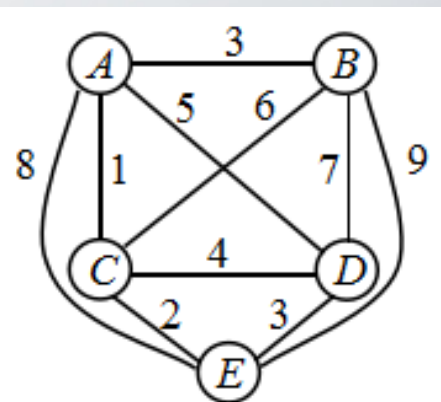
这个下界对应的解可能不是一个可行解 (没有构成哈密顿回路), 但给出了一个**参考下界**。

11.3.3 TSP问题

例如，对于下图，采用贪心法求得近似解为 $A \rightarrow C \rightarrow E \rightarrow D \rightarrow B \rightarrow A$ ，路径长度为 $1+2+3+7+3=16$ ，这可以作为TSP问题的上界。把矩阵中每一行最小的两个元素相加再除以 2，得到TSP问题的下界： $[(1+3)+(3+6)+(1+2)+(3+4)+(2+3)]/2=14$ 。于是，得到了目标函数的界 $[14, 16]$ 。

一般情况下，假设当前已确定的路径为 $U=(r_1, r_2, \dots, r_k)$ ，即路径上已确定了 k 个顶点，此时，该部分分解对应目标函数值的计算方法（即界限函数）如下：

$$lb = \sum_{i=1}^{k-1} c[r_i][r_{i+1}] + \left(\sum_{i=1, k} r_i \right) + \sum_{r_j \notin U} r_j \quad) / 2$$

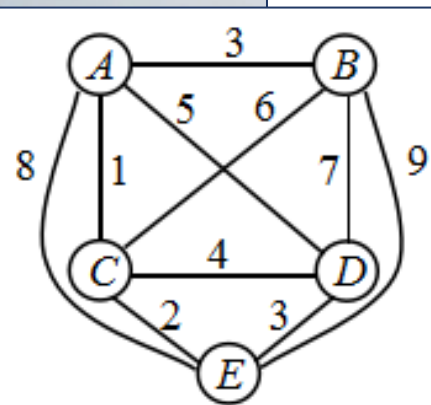
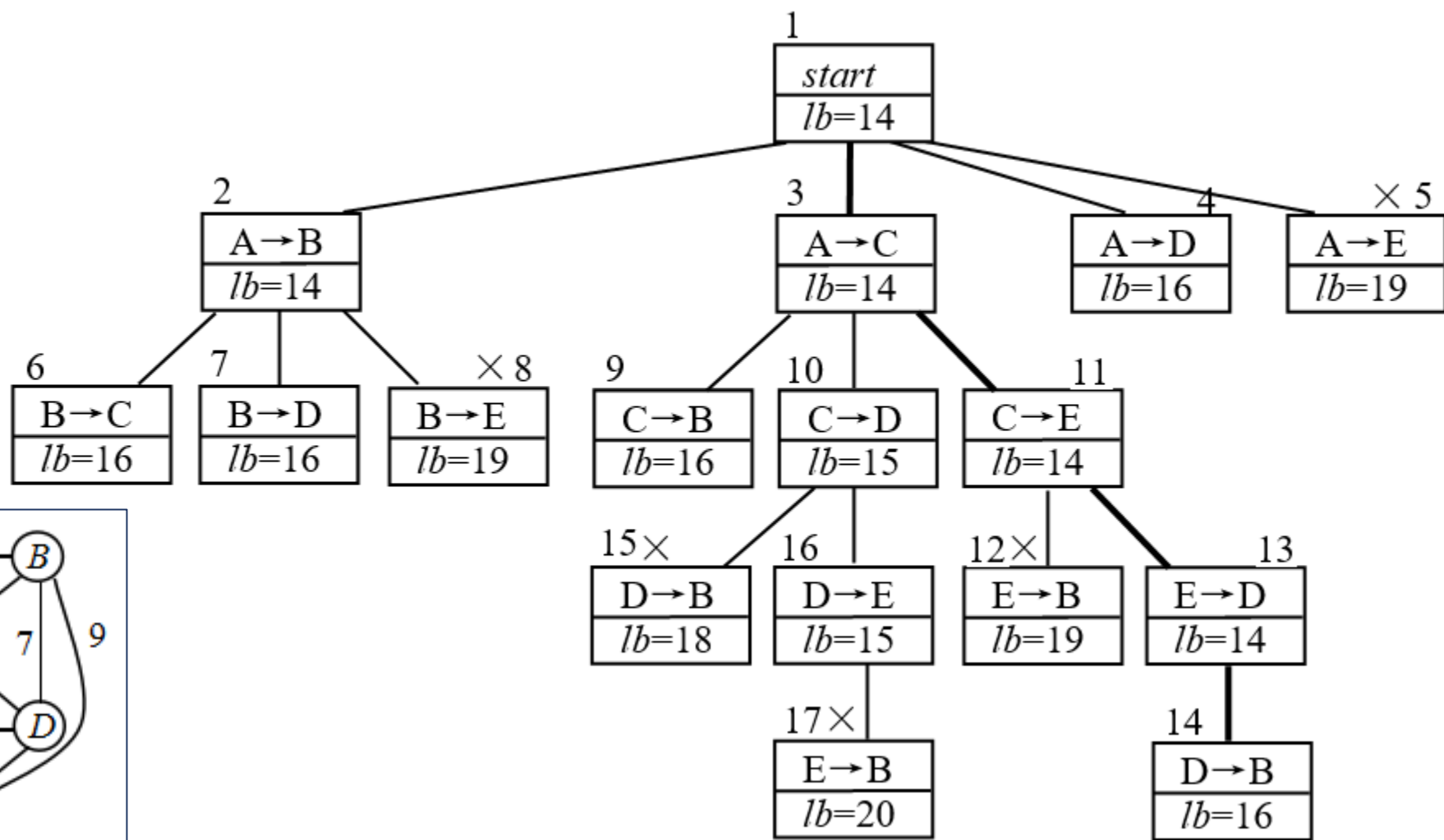


(a) 一个无向图

$$C = \begin{bmatrix} \infty & 3 & 1 & 5 & 8 \\ 3 & \infty & 6 & 7 & 9 \\ 1 & 6 & \infty & 4 & 2 \\ 5 & 7 & 4 & \infty & 3 \\ 8 & 9 & 2 & 3 & \infty \end{bmatrix}$$

(b) 无向图的代价矩阵

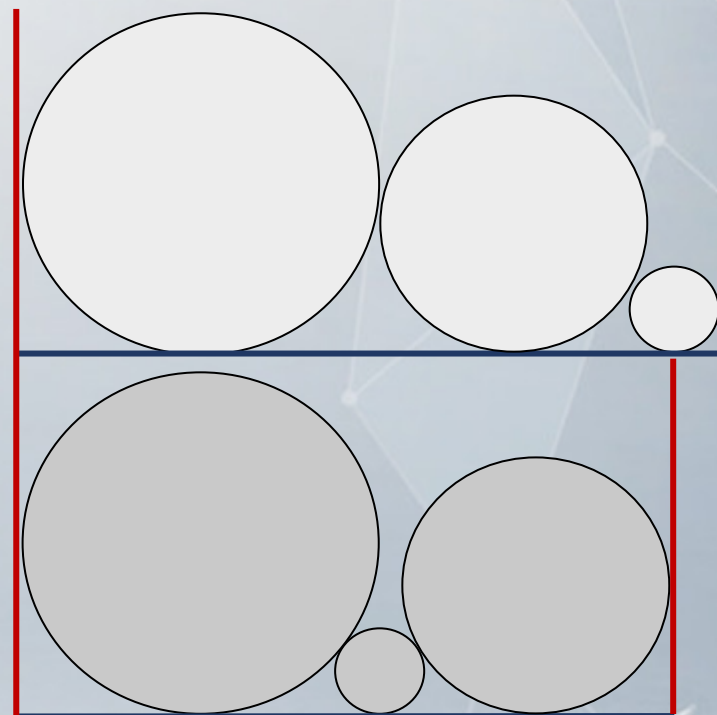
11.3.3 TSP问题



11.3.4 圆排列问题

【问题】 给定 n 个圆的半径序列，将这些圆放到一个矩形框中，各圆与矩形框的底边相切，则圆的不同排列会得到不同的排列长度。要求找出具有最小长度的圆排列。

【想法】 设各圆的编号为 $\{1, 2, \dots, n\}$ ，半径分别为 $\{r_1, r_2, \dots, r_n\}$ ，可行解为向量 (i_1, i_2, \dots, i_n) ，且解向量为 $1, 2, \dots, n$ 的全排列，表示第 k 个位置放置编号为 i_k 的圆。采用限界剪枝法求解圆排列问题，**关键是设计限界函数**，以便在搜索过程中选择使目标函数取得极值的结点优先进行扩充。



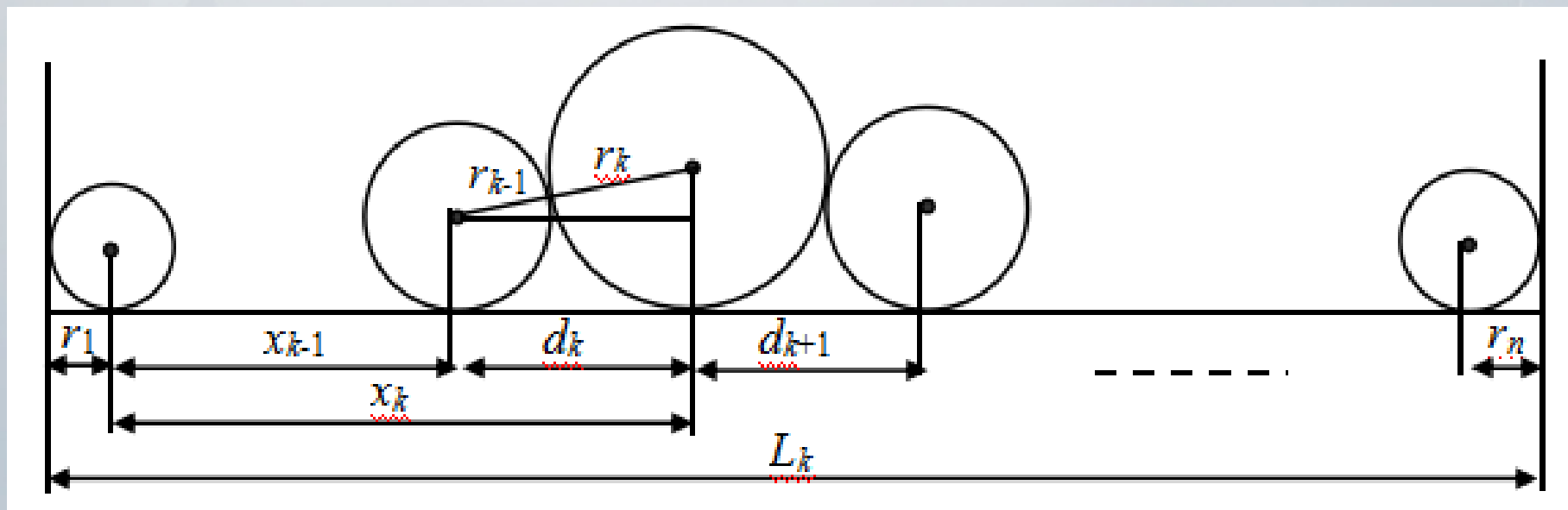
11.3.4 圆排列问题

【想法】假设已排列了圆(i_1, i_2, \dots, i_{k-1}), 则在排列圆 i_k 时, 目标函数可能取得的极小值 L_k 的计算公式如下, 其中各记号定义如下:

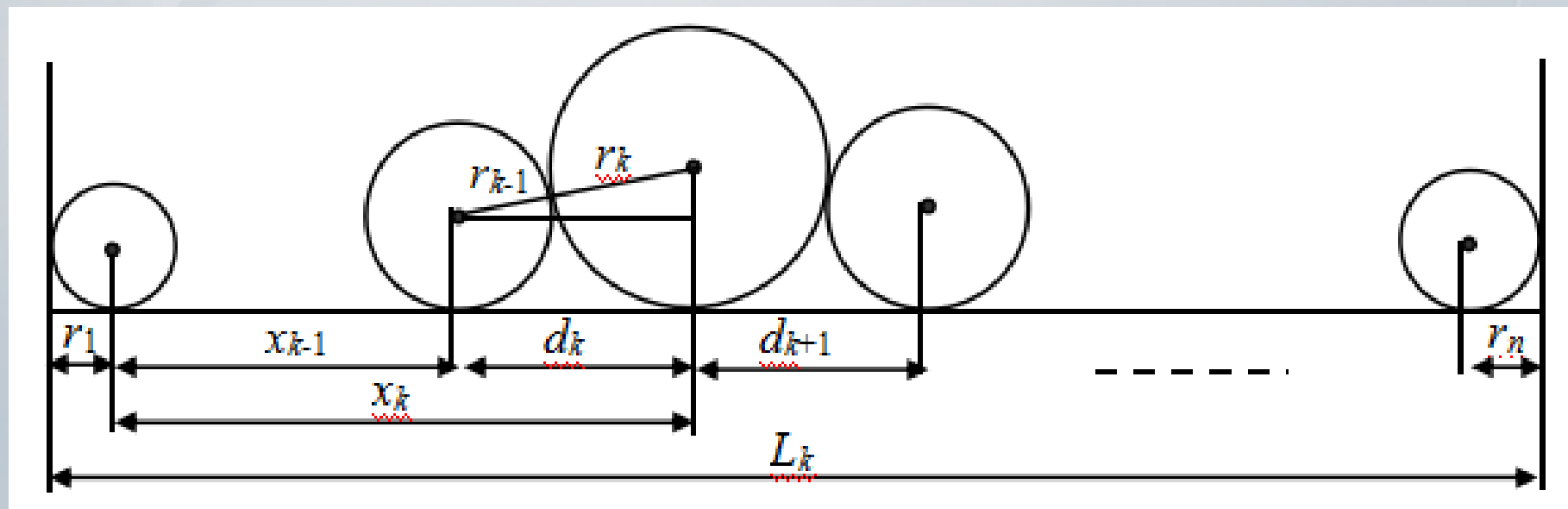
x_k : 第 k 个位置所放圆的圆心坐标, 规定第 1 个圆的圆心为坐标原点, 即 $x_1=0$ 。

d_k : 第 k 个位置所放圆的圆心坐标与第 $k-1$ 个位置所放圆的圆心坐标的差。

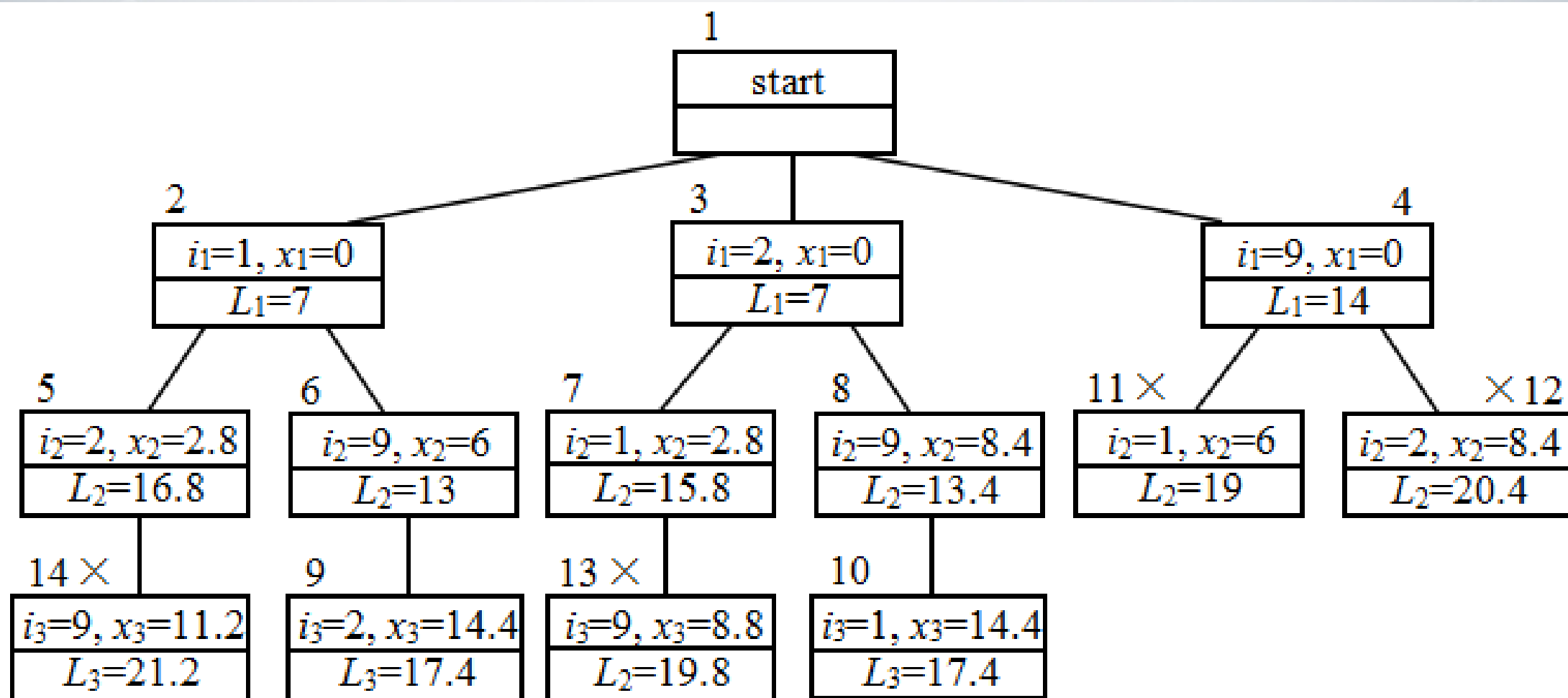
L_k : 第 1~ k 个位置放置圆后, 可能得到的目标函数的极小值。



11.3.4 圆排列问题




11.3.4 圆排列问题



第 11 章 广度优先搜索

11-4 拓展与演练

11.4.1 限界剪枝法的关键问题

 限界剪枝法的较高效率是以付出一定计算代价为基础的，其工作方式也造成了算法设计的复杂性，应用限界剪枝法的关键问题如下。

(1) **确定合适的限界函数**。好的限界函数要求：① 计算简单；② 保证最优解在搜索空间中；③ **能在搜索的早期对超出目标函数界的结点进行丢弃**，减少搜索空间，从而尽快找到问题的最优解。

(2) **open表的存储结构**。为了能快速在open表中选取使目标函数取得极值的结点，通常采用优先队列存储open表。如果open表的结点数不是很多，也可以简单的采用数组存储。

(3) **确定最优解的各个分量**。限界剪枝法跳跃式处理搜索空间中的结点，需要对每个扩展结点保存该结点到根结点的路径，或者在搜索过程中构建搜索经过的树结构。

11.4.2 批处理作业调度问题

【问题】 给定 n 个作业的集合 $J=\{J_1, J_2, \dots, J_n\}$, 每个作业都有 3 项任务分别在 3 台机器上完成, 作业 J_i 需要机器 j 的处理时间为 t_{ij} ($1 \leq i \leq n, 1 \leq j \leq 3$), 每个作业必须先由机器 1 处理, 再由机器 2 处理, 最后由机器 3 处理。**批处理作业调度问题 (batch-job scheduling problem)** 要求确定这 n 个作业的最优处理顺序, 使得从第 1 个作业在机器 1 上处理开始, 到最后一个作业在机器 3 上处理结束所需的时间最少。

【想法】 批处理作业的一个最优调度应使机器 1 没有空闲时间, 且机器 2 和机器 3 的空闲时间最小。可以证明, 存在一个最优作业调度使得在机器 1、机器 2 和机器 3 上作业以相同次序完成。

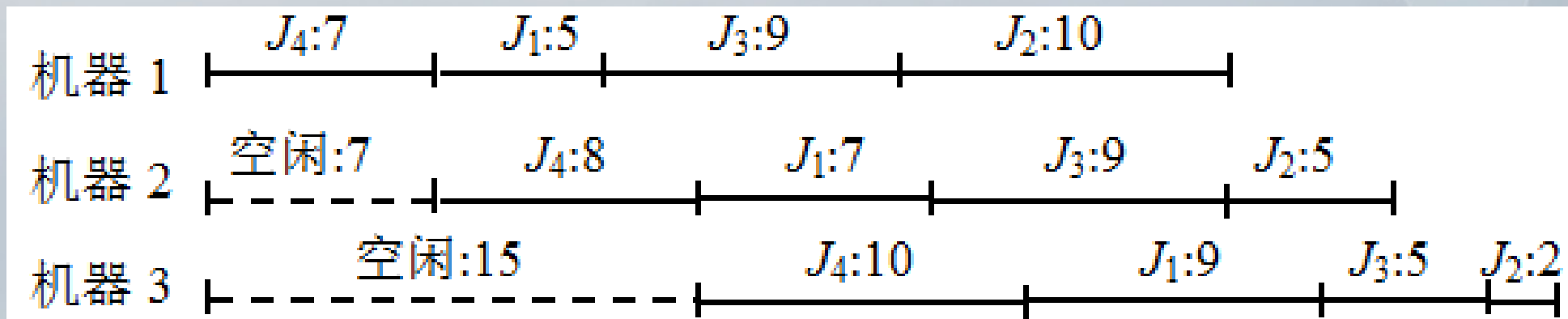
	机器 1	机器 2	机器 3
J_1	5	7	9
J_2	10	5	2
J_3	9	9	5
J_4	7	8	10

11.4.2 批处理作业调度问题

【想法】 如何在不实际求解问题的情况下得到一个近似解呢？可以随机产生几个调度方案，从中选取具有最短完成时间的调度方案作为近似最优解。

例如，设 $J = \{J_1, J_2, J_3, J_4\}$ 是 4 个待处理的作业，需要的处理时间如图所示。若处理顺序为 (J_4, J_1, J_3, J_2) ，则从作业 4 在机器 1 处理到作业 2 在机器 3 处理完成的时间 41，这可以作为批处理作业调度问题的上界。

	机器 1	机器 2	机器 3
J_1	5	7	9
J_2	10	5	2
J_3	9	9	5
J_4	7	8	10

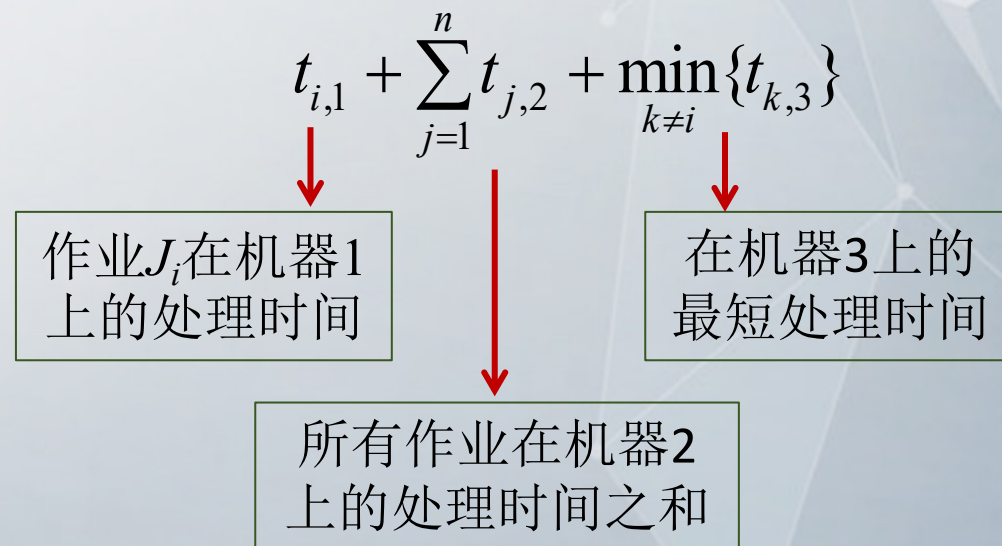


11.4.2 批处理作业调度问题

【想法】 如何估算部分分解的下界呢？考虑理想情况，机器 1 和机器 2 无空闲，最后处理的恰好是在机器 3 上处理时间最短的作业。例如，以作业 J_i 开始的处理顺序，估算处理所需的最短时间是：

一般情况下，假设 M 是当前已安排了 k 个作业的集合，设 sum1 表示机器 1 完成 k 个作业的处理时间， sum2 表示机器 2 完成 k 个作业的处理时间，现在要处理作业 $k+1$ ，该部分分解的目标函数值的下界计算方法如下：

- (1) $\text{sum1} = \text{sum1} + t_{k+1,1}$
- (2) $lb = \max\{\text{sum1}, \text{sum2}\} + \sum_{i \notin M} t_{i,2} + \min\left\{\sum_{j \neq k+1, j \notin M} t_{j,3}\right\}$
- (3) $\text{sum2} = \max\{\text{sum1}, \text{sum2}\} + t_{k+1,2}$



11.4.2 批处理作业调度问题

