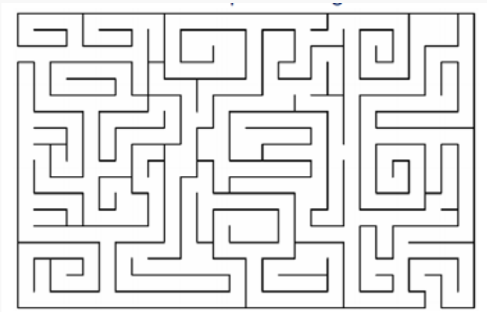
问题求解与实践
——贪心算法

主讲教师：陈雨亭、沈艳艳

贪心算法

- 贪心法建议通过一系列步骤来构造问题的解，每一步都对目前构造的部分解做一个扩展，直到获得问题的完整解为止
- 在每一步中，它“贪心”地选择最佳操作，并希望通过一系列局部的最优选择，能够产生一个整个问题的全局最优解
- 事实上，贪心算法不是对所有问题都能得到整体最优解，但对许多问题他能产生整体最优解或者是整体最优解的近似解



贪心算法举例 —— 找零钱

◆ 买东西时，售货员常常计算最少需要找多少张零钱，以便简化工作流程。比如顾客购物需要48.5元，他交给售货员100元整，则售货员最少需要找三张零钞：50元一张、1元一张、5角一张。

算法

1. 计算100-48.5=51.5
2. 为了使得找零的零钞数量较少，先按50元大面额的找零

   51.5 / 50 再取整 = **1**（张）

   51.5 % 50 -> 余额 1.5（元）
3. 为了使得零钞数量较少，再按照1元面额的找零

   1.5 / 1再取整 = **1**（张）

   1.5 % 1 -> 0.5（元）

   ……

是最优解吗？

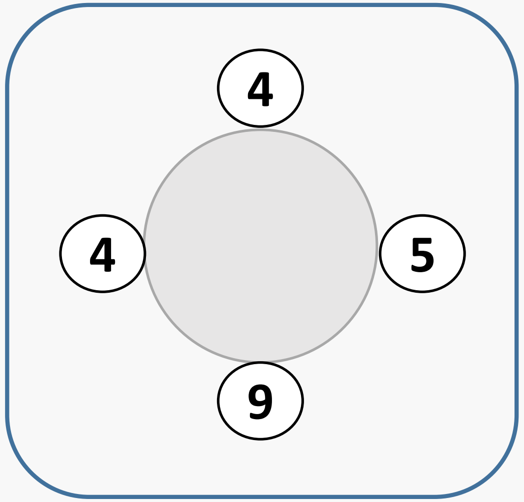
贪心算法举例 —— 付零钱

- 一位旅客坐公交车时需要支付n元。他现在拿出一个零钱袋，里面有各种类型硬币。请问，他如何投币？
  - 他需要尽可能把零钱花掉的时候?
  - 他需要投递最少数量的硬币的时候?
  - 投币机可以自动找零的时候?
  - 投币机不可以自动找零的时候?
  - 硬币中出现quarter的时候?
  - 投币机不识别某几种面值硬币的时候?

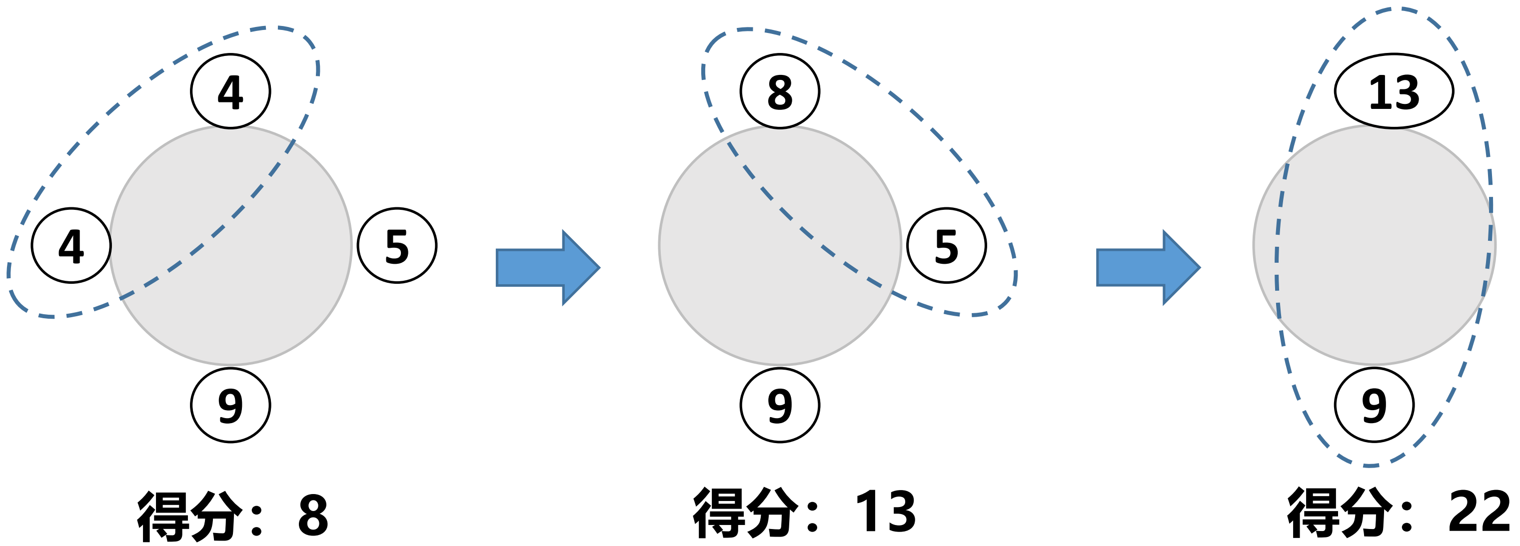
每一种情况都能用贪心算法解决吗?

贪心算法举例 —— 石子合并问题

◆ 在操场的四周摆放 N 堆石子，现要将石子有次序地合并成一堆。规定每次只能选相邻的两堆合并，并将新的一堆的石子数记为该次合并的得分。已知每堆石子数量，请选择一种合并方案，使得进行N-1次合并得分的总和最小



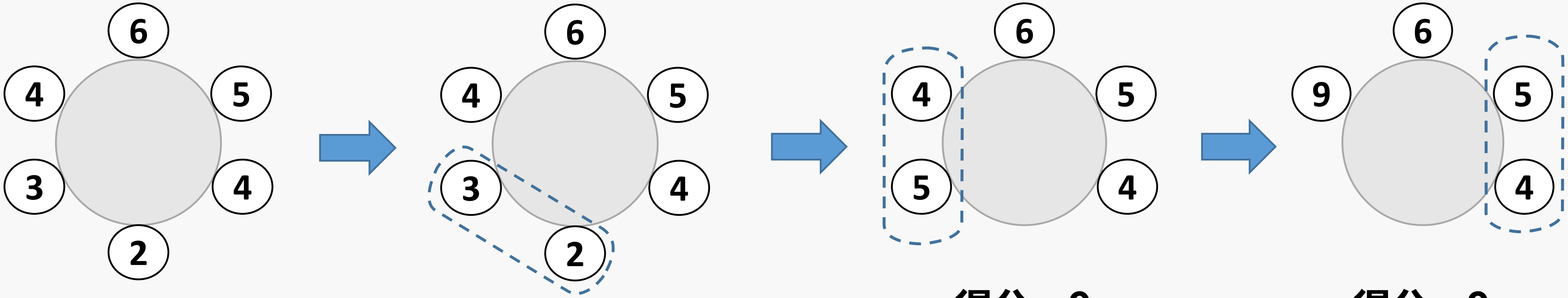
一种经验解答（贪心法）



得分总和=8+13+22=43

贪心算法举例 —— 石子合并问题

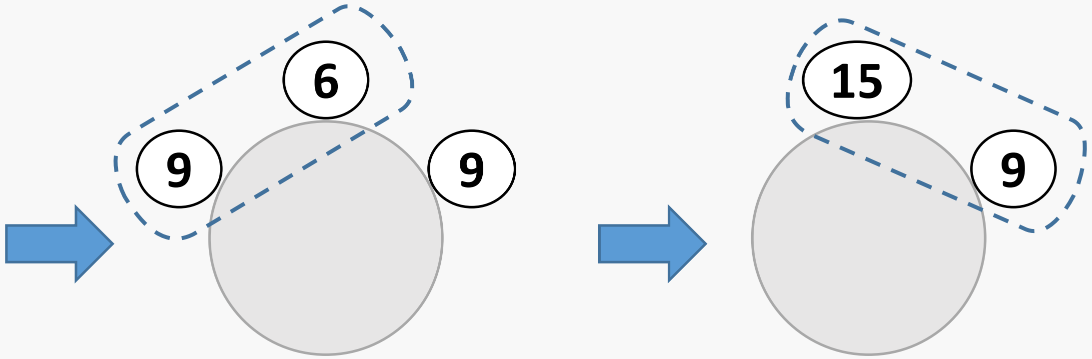
➢贪心策略：每次选相邻和最小的两堆石子合并，一定能得到最优解吗?



得分：5

得分：9

得分：9



得分：15

得分：24

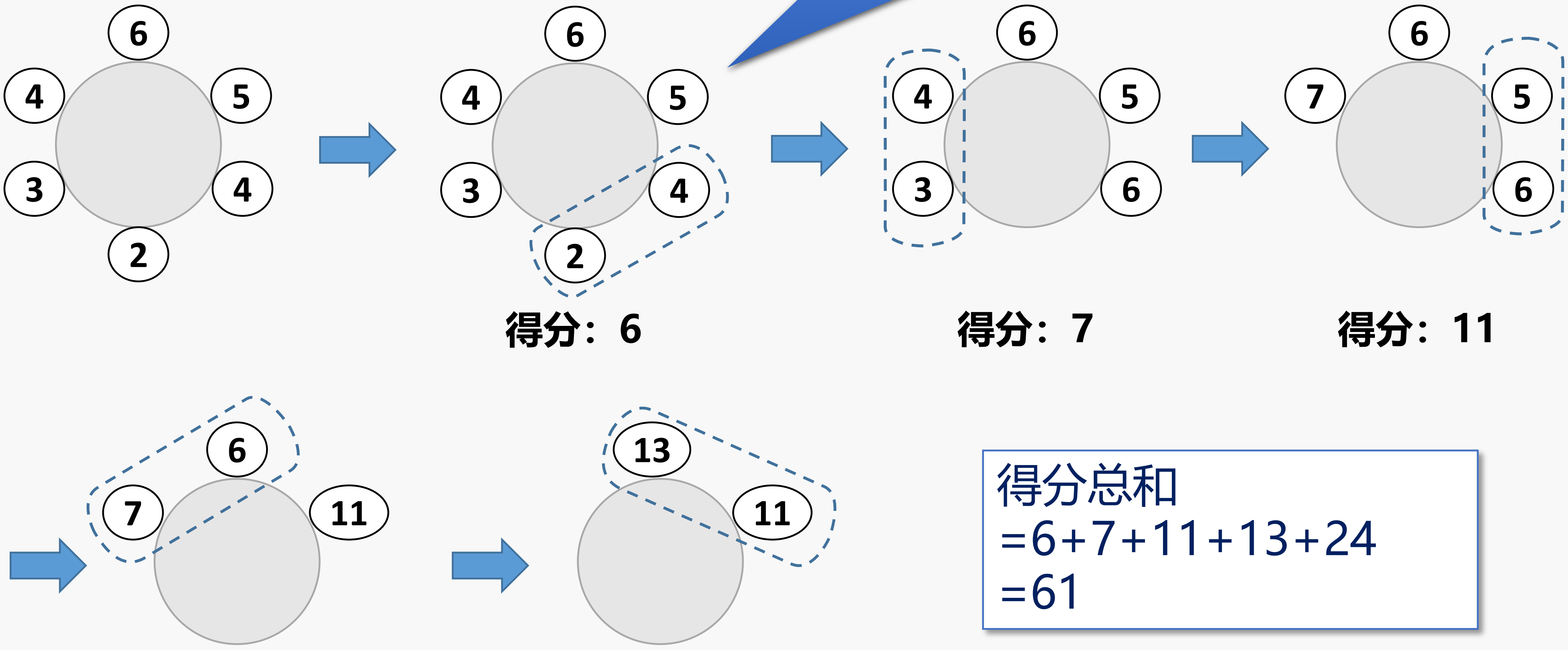
得分总和
=5+9+9+15+24
=62

是最优解?

贪心算法举例 —— 石子合并问题

➢ 不用贪心策略的另一解法

这里没有用贪心策略



得分：6

得分：7

得分：11

得分：13

得分：24

得分总和
=6+7+11+13+24
=61

问题求解与实践
——活动安排问题

主讲教师：陈雨亭、沈艳艳

问题

- 如何设计一个活动安排，保证资源使用最充分？即资源空闲时间最少。

贪心算法 —— 活动安排问题

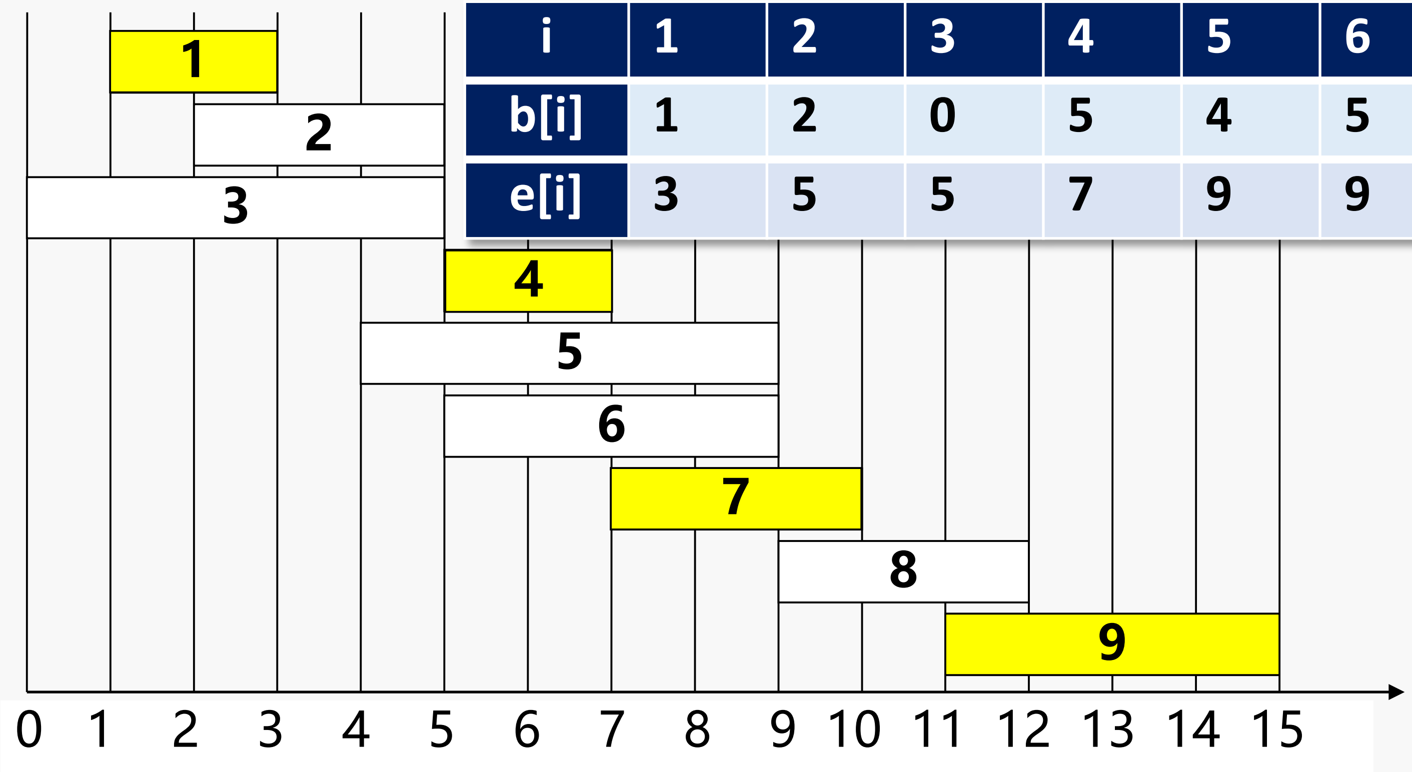
◆ 问题描述

- 设有 n 个活动的集合S={1,2,…,n}，其中每个活动都要求使用同一资源，如学校礼堂等，而在**同一时间内只有一个活动能使用这一资源**；
- 每个活动都有使用的起始时间b_i和结束时间e_i 。若$b_i \geq e_j$或$b_j \geq e_i$（即一个活动结束后另一个才开始），则活动 i 与活动 j 相容；
- 活动安排问题就是要在所给的活动集合中选出最大的相容活动子集合。

贪心算法 —— 活动安排问题

- 贪心策略：选择结束时间尽量早的活动，以便腾出更多时间留给后续活动
- 不妨假设活动已经按照结束时间从小到大排序（如下图所示）

i	1	2	3	4	5	6	7	8	9
b[i]	1	2	0	5	4	5	7	9	11
e[i]	3	5	5	7	9	9	10	12	15



1，因为其
；
1相容的活
动中，选择结束时间最早
的活动4；

- 再在与活动1和4都相容的活动中，选结束时间最早的活动7；
- 最后，在选择与活动1、4、7都相容的活动9

活动安排问题编程实现

- 在下列算法中，n为活动个数，数组b[]和e[]分别为活动开始和结束时间。假设活动已经按结束时间递增排列：e[1]≤e[2]≤ …≤e[n]
- 数组A记录所选择的活动，A[i]=1表示活动 i 被选中，A[i]=0表示活动 i 未被选中

```
GreedyActivitySelector( n， b[], e[], A[] )
{
    A[1]=1;                //选中活动1
    j=1;                   //记录最后选中的活动
    for(i 从 2 到 n )
    {
       if( b[i] ≥ e[j]) {
          A[i]←1;          //选中活动i
          j=i;             //记录最后选中的活动
       }
    }
}
```

每次选中一个活动 j 之后，i 恰好从 j 之后继续开始寻找相容活动

问题

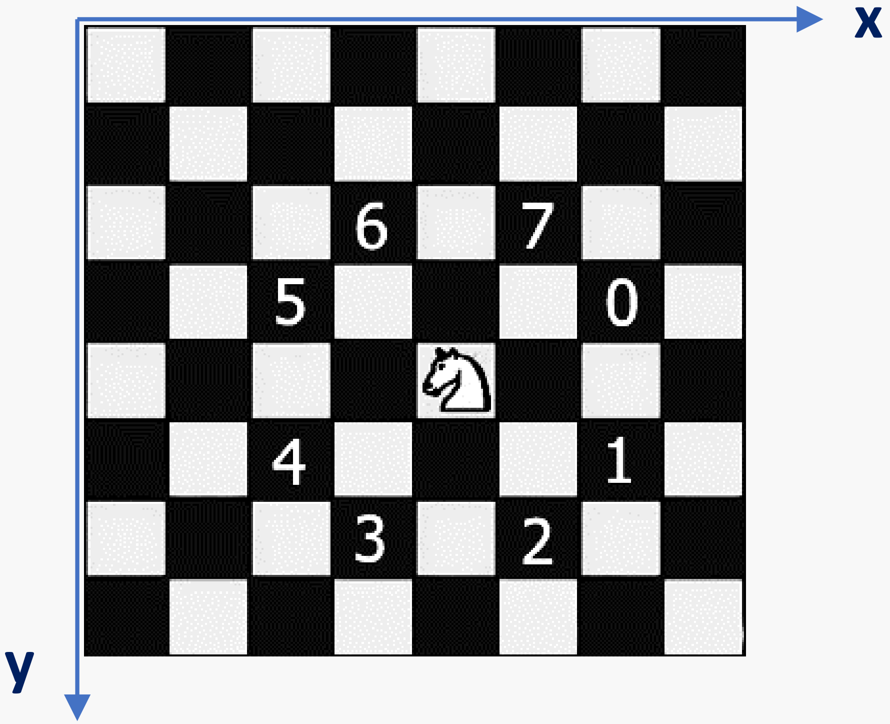
- 如何设计一个活动安排，保证资源使用最充分？即资源空闲时间最少。

问题求解与实践
——马踏棋盘(贪心法)

主讲教师：陈雨亭、沈艳艳

马踏棋盘问题

- 国际象棋棋盘是8*8的方格，现将“马”放在任意指定的方格中，按照走棋规则移动该棋子。要求每个方格只能进入一次，最终走遍棋盘64个方格
- 棋盘可用一个矩阵表示，当“马”位于棋盘上某一位置时，它就有一个唯一的坐标。根据规则，如果当前坐标是(x,y)，那么它的下一跳可能有8个位置，分别是(x+2,y-1)、(x+2,y+1)、(x+1,y+2)、(x-1,y+2)、(x-2,y+1)、(x-2,y-1)、(x-1,y-2)、(x+1,y-2) 。当然**坐标不能越界**



马踏棋盘问题

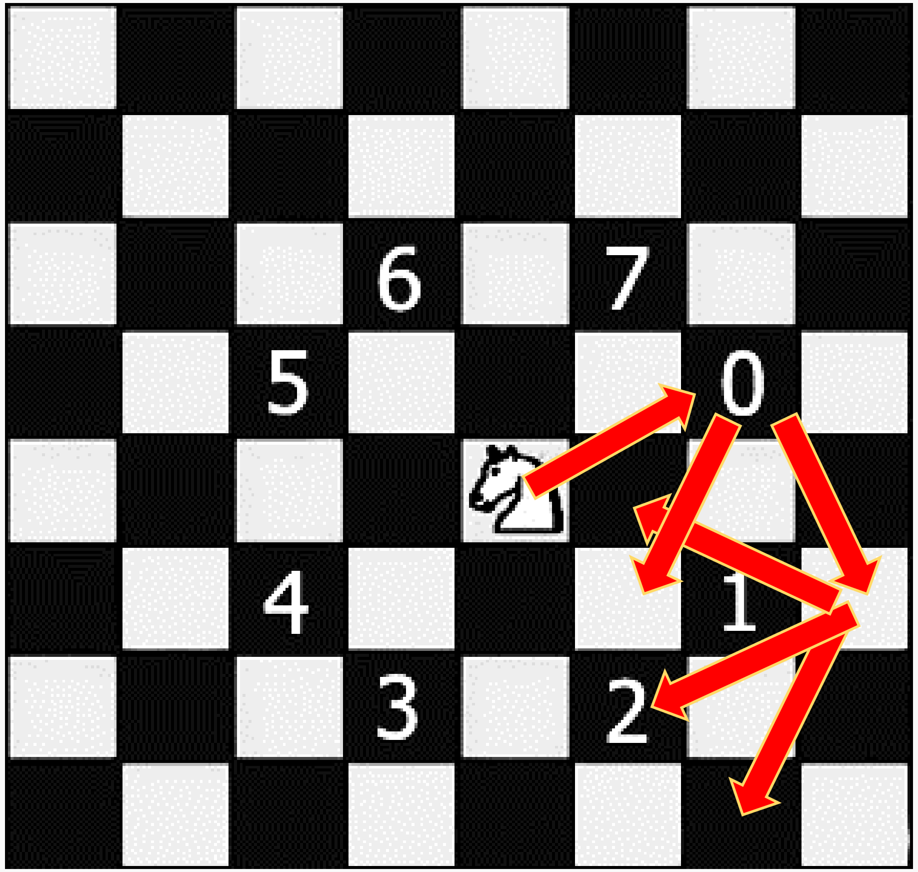
- 马最初的位置标为1，它的下一跳的位置标为2，再下一跳的位置标为3，依次类推，如果马走完棋盘，那么最后在棋盘上标的位置是64
- 要求编程解决马踏棋盘问题，最后输出一个8*8的矩阵，并用数字1-64来标注“马”的移动

27	18	29	32	25	20	3	50
30	33	26	19	2	51	24	21
17	28	31	58	55	22	49	4
34	57	46	1	52	41	54	23
45	16	63	56	59	48	5	40
64	35	60	47	42	53	8	11
15	44	37	62	13	10	39	6
36	61	14	43	38	7	12	9

马踏棋盘问题

◆ 基本解法：深度优先搜索

从某位置出发，先试探下一个可能位置；进入一个新位置后就从该位置进一步试探下一位置，若遇到不可行位置则回退一步，然后再试探其他可能位置



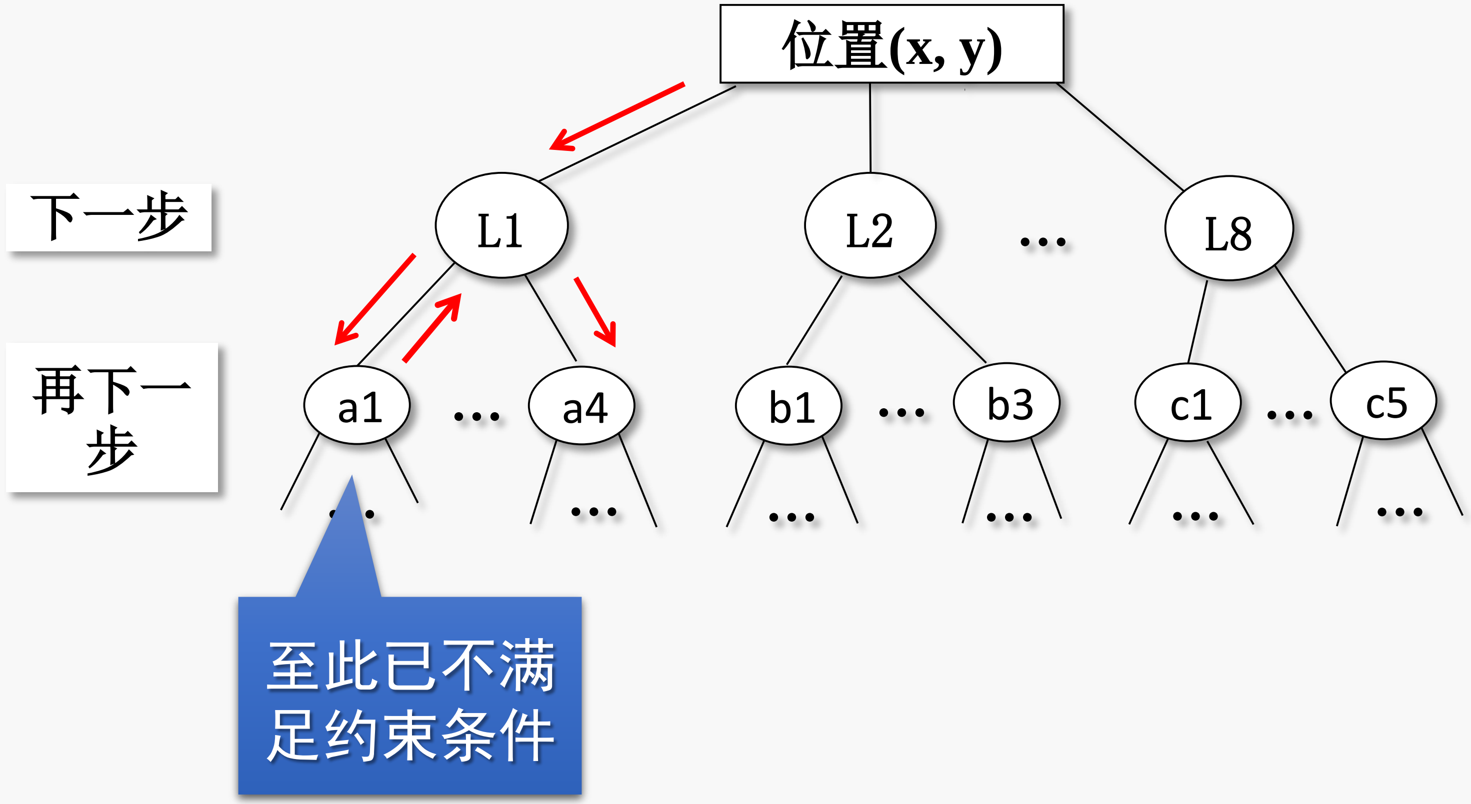
马踏棋盘——深度优先搜索求解

- 可以用8*8数组 chess[][] 存储棋子周游状态，未走过位置赋0，走过的位置依次为1、2、3、……。设(x,y)为当前位置，j 为当前走到第几步

```
// 深度优先搜索求解核心代码
bool Dfs(int chess[][8], int x, int y, int j)
{
    chess[x][y] = j;                    //将新的一步标注到矩阵中
    if (j == 64)   return true;         //成功！依次回退结束程序
    计算下一可能位置(nextX, nextY);         //不考虑是否可行
    while ( (nextX, nextY)位置是否可行 ) {
        if (Dfs(chess, nextX, nextY, j+1))    //递归调用，将进入新位置
            return true;
        计算下一可能位置(nextX, nextY);
    }
    chess[x][y] = 0;        //这一步不可走，回退
    return false;
}
```

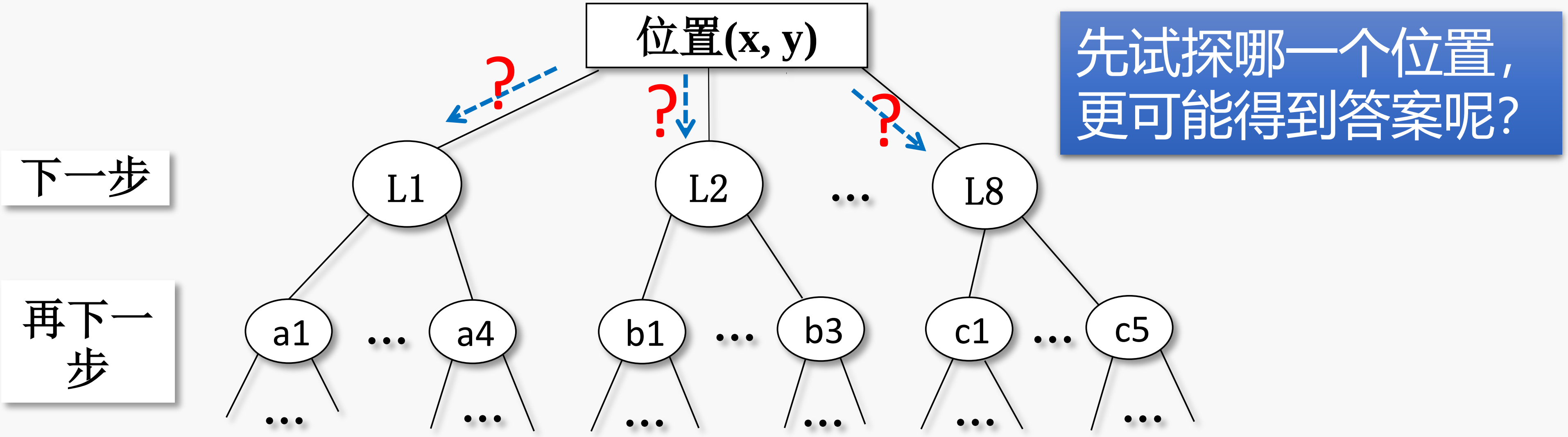
马踏棋盘——深度优先搜索

- 深度优先搜索求解，相当于在类似下面的树中查询



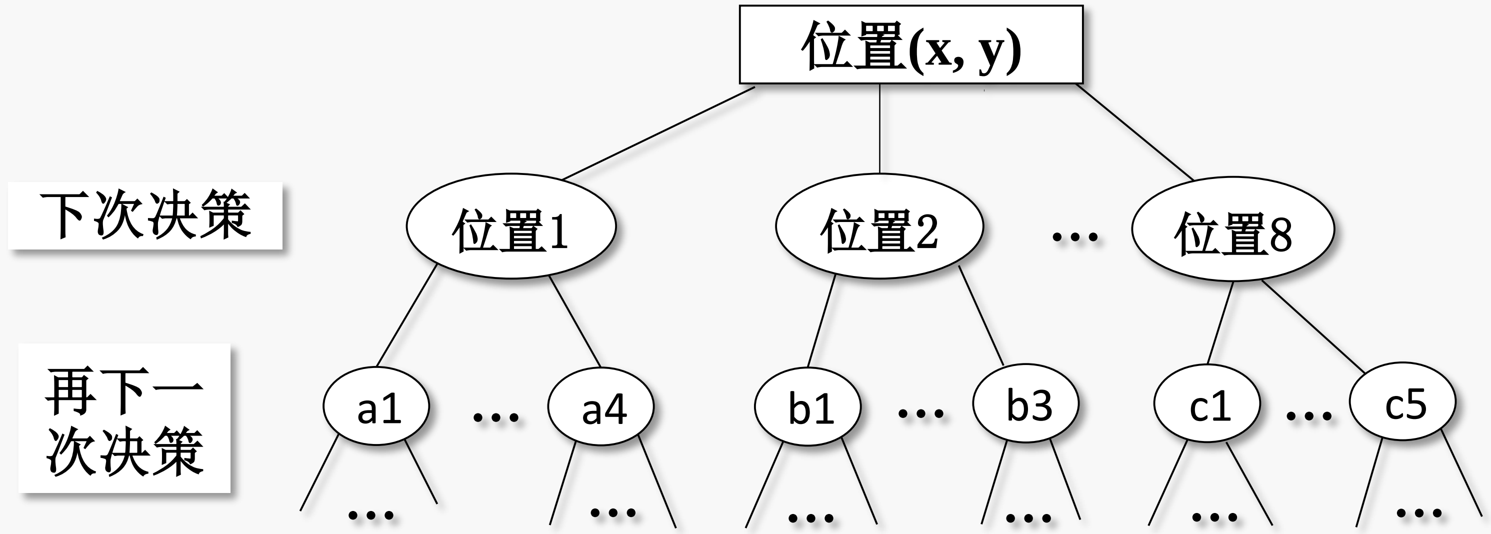
马踏棋盘——深度优先搜索 + 贪心法优化

- 从(x,y)开始，深度优先搜索会依次查询L1、L2、……、L8一共8个可能位置
- 如何加入贪心策略?



- 事实上，先选择L1~L8中哪个位置试探，可利用L1~L8位置的子结点个数确定

马踏棋盘——深度优先搜索 + 贪心法优化



这里每个位置的子结点个数就是下一跳可能位置的个数，我们称为**出口数**

贪心策略：在选择下一跳的位置时，总是先选择出口少的那个位置

?

- 如果优先选择出口多的子结点，那么**出口少的子结点就会越来越多**，很可能出现走不通的‘死’结点。这时只有回退再搜索，会浪费很多时间
- 反之如果优先选择出口少的结点，那出口少的结点就会越来越少，这样成功的机会就更大

马踏棋盘——深度优先搜索

```
// 深度优先搜索求解核心代码
bool Dfs(int chess[][8], int x, int y, int j)
{
    chess[x][y] = j;              //将新的一步标注到矩阵中
    if (j == 64)   return true;   //成功！依次回退结束程序
    计算下一可能位置(nextX, nextY);
    while ( 下一位置(nextX, nextY) 可行 ) {
        if (Dfs(chess, nextX, nextY, j+1))   //递归，进入新位置
            return true;
        计算下一可能位置(nextX, nextY);
    }
    chess[x][y] = 0;         //此路可走，回退
    return false;
}
```

将计算下一位置的工作进行修改

计算(x,y)的下一位置放入数组nextX[]、nextY[]中(不超8个)，然后将数组nextX[]、nextY[]**按出口数从小到大**排序，再依次循环试探各个位置

马踏棋盘——深度优先搜索 + 贪心法优化

- 函数NextXY计算(x,y)的下一位置。 nextX[], nextY[]最终按优先级存储下一个位置。 weight[]存储每个结点出口数。

```
void NextXY(int chess[][8], int x, int y, int nextX[], int nextY[])
{
    //nextX[],nextY[]数值取-1表示这个位置不可行（已走过或出界）
    将8个位置 (nextX[0],nextY[0]) 到 (nextX[7],nextY[7]) 都设为 -1;

    循环计算 (x,y) 的下一可行结点的出口数，并将 (nextX[i],nextY[i]) 的
    出口数放入weight[i];

    按出口数 weight[] 的值从小到大对 nextX[]、nextY[] 排序;
}
```

马踏棋盘——深度优先搜索 + 贪心法优化

```
bool Dfs(int chess[][8], int x, int y, int j)  {
    chess[x][y] = j;                    //将新的一步标注到矩阵中
    if (j == 64)    return true;        //成功！依次回退结束程序

    用NextXY(.)函数计算下一可能位置存入nextX[], nextY[];

    i=0;
    while ( nextX[i]>0 && i<8 )         // 位置可行
    {
        if (Dfs(chess, nextX[i], nextY[i], j+1))    // 进入新位置
            return true;
        i++;
    }

    chess[x][y] = 0;        //此路不可走，回退
    return false;
}
```

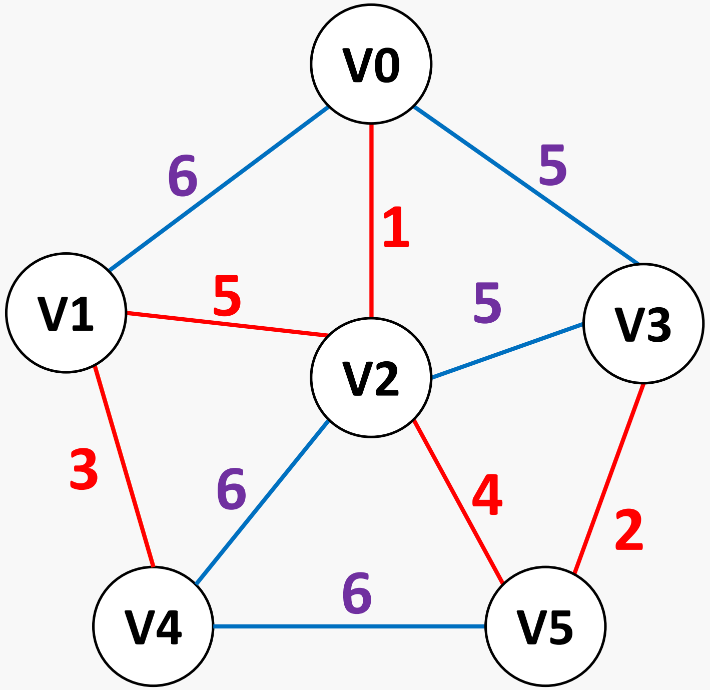
问题求解与实践
——最小生成树

主讲教师：陈雨亭、沈艳艳

最小生成树

- 最小生成树是图结构中的一个经典问题，同时也是贪心算法的典型问题
- 什么是最小生成树?

  在一个有N个点的**含权的连通图**中，找出其中的N-1条边，它们恰好连接所有的N个点，并且这N-1条边的**权值之和**是所有方案中**最小**的

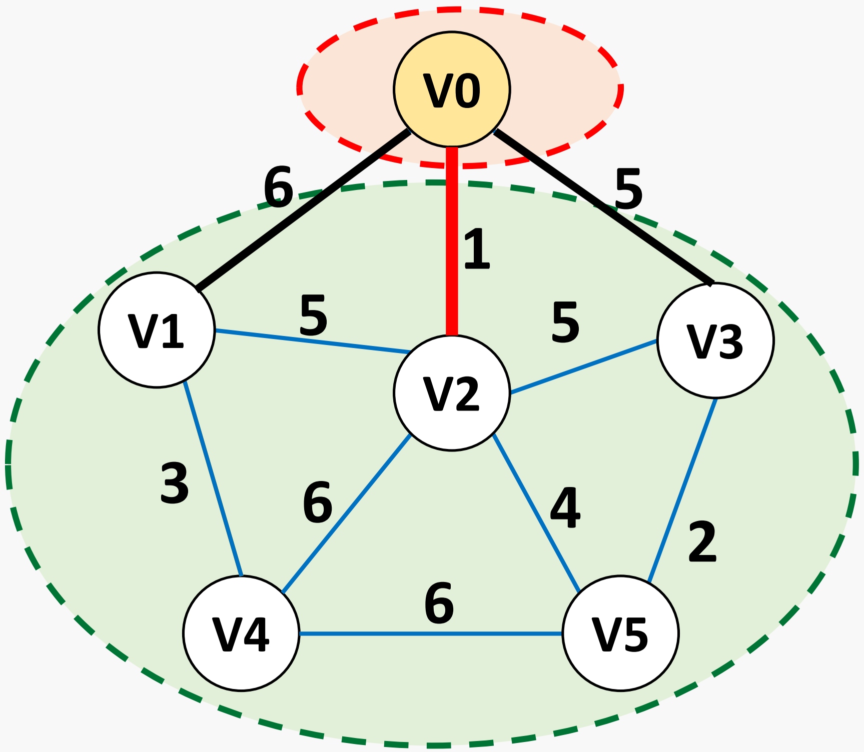


得到最小生成树的常见算法:

- 普里姆算法
- 克鲁斯卡尔算法

最小生成树——普里姆算法

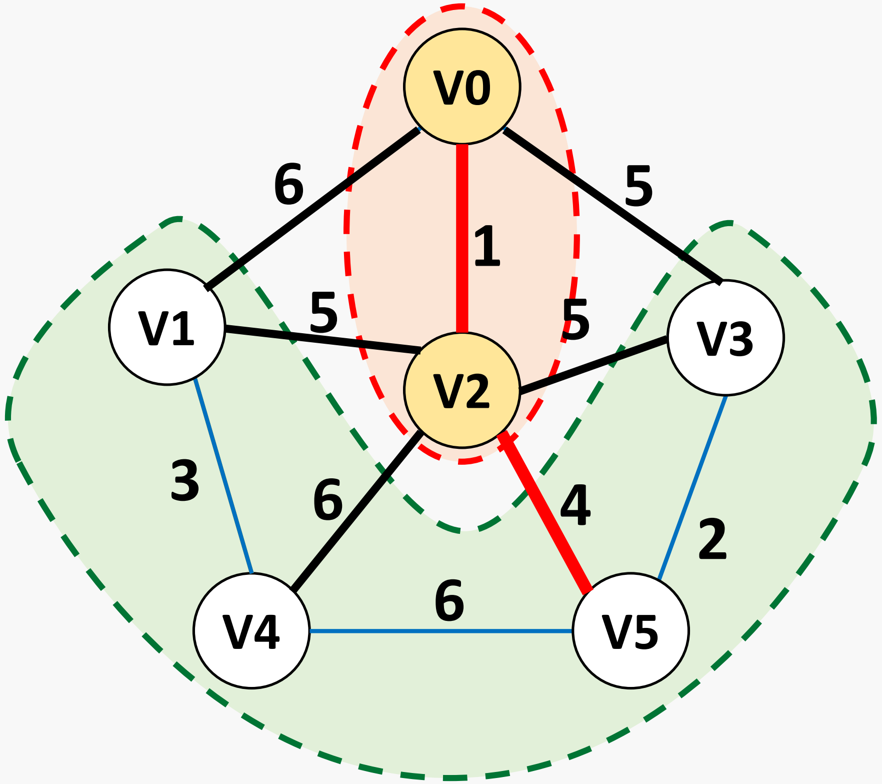
- 普里姆算法是从某个顶点出发，逐步“生长”出其他所有顶点
- 考察下图，不妨取V0为初始点，应该**取哪一条边**扩展下一个顶点呢？



- **可以将顶点分成两个集合：{V0} 和 {除去V0的其他点}**
- **显然，在生成树中上面两个集合必然连通，即三条黑色的边中必有一条在生成树中**
- **为了使生成树边上权值之和最小，根据贪心的原则，应取三条黑色的边中的(V0, V2)**

最小生成树——普里姆算法

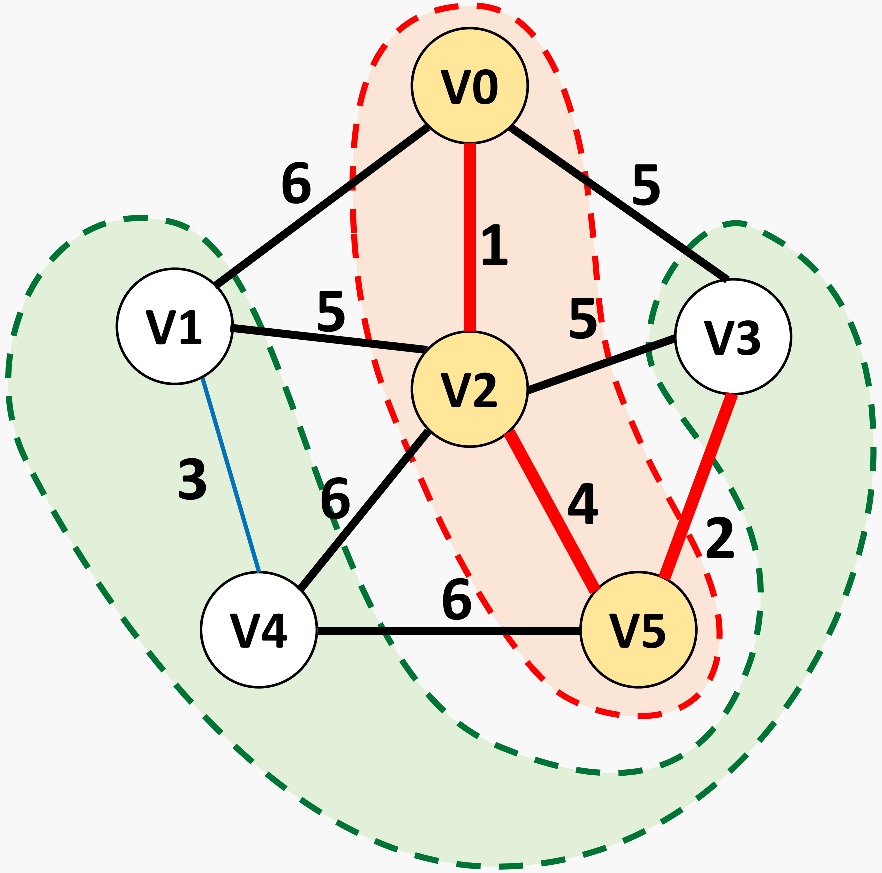
➢ 以上做法可以进一步使用，继续“生长”出其他顶点



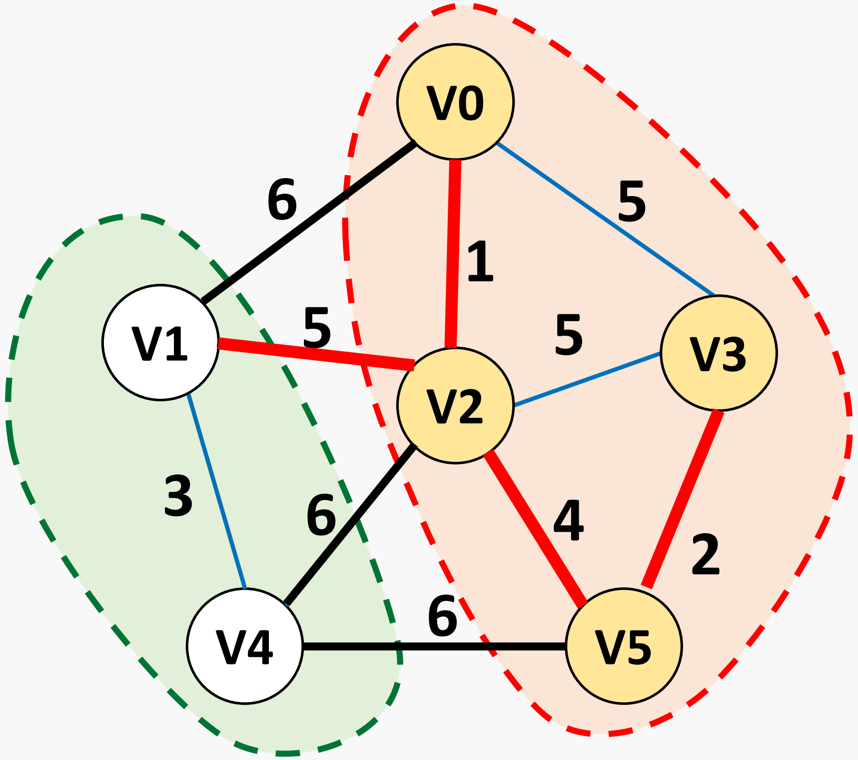
- 将顶点分成两个集合：{V0, V2} 和{除去V0、V2的其他点}
- 显然，在生成树中上面**两个集合必然连通**，即连接两个集合的黑色的边中必有一条在生成树中
- 为了使生成树边上权值之和最小，根据贪心的原则，应取黑色的边中的(V2, V5)

最小生成树——普里姆算法

➢ 继续“生长”出其他顶点



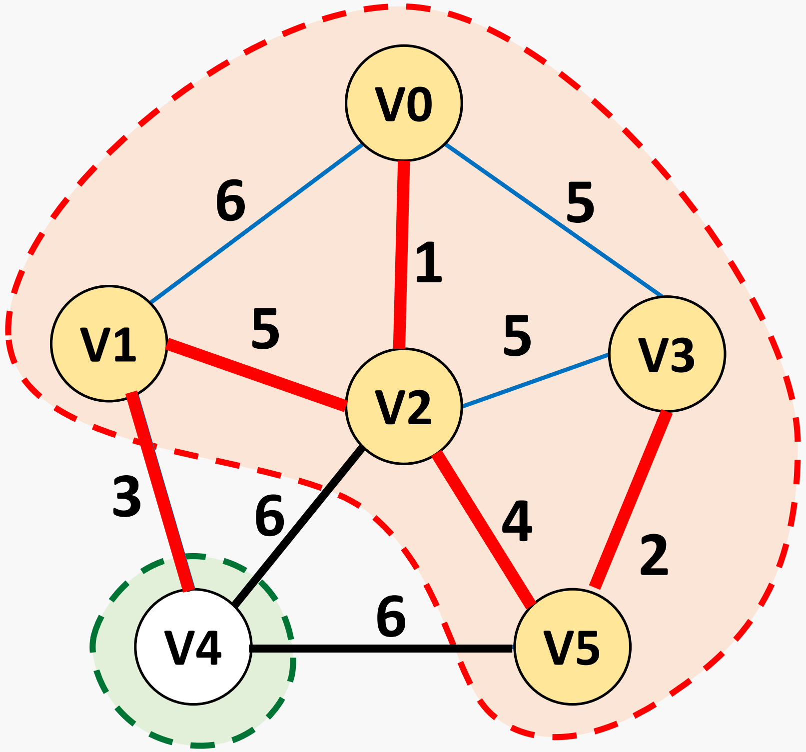
为了使生成树边上权值之和最小，
应取黑色的边中的(V3, V5)

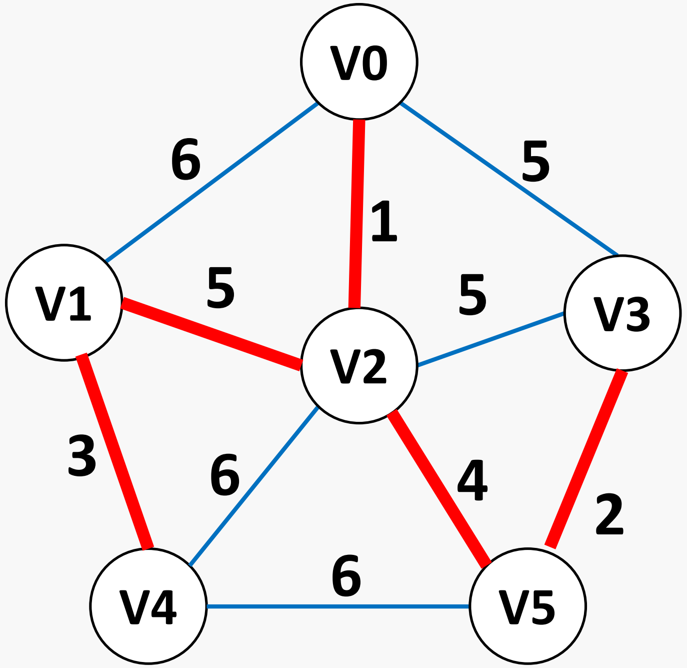


为了使生成树边上权值之和最小，
应取黑色的边中的(V1, V2)

最小生成树——普里姆算法

- 继续“生长”出其他顶点





为了使生成树边上权值之和最小，应取黑色的边中的(V1, V4)

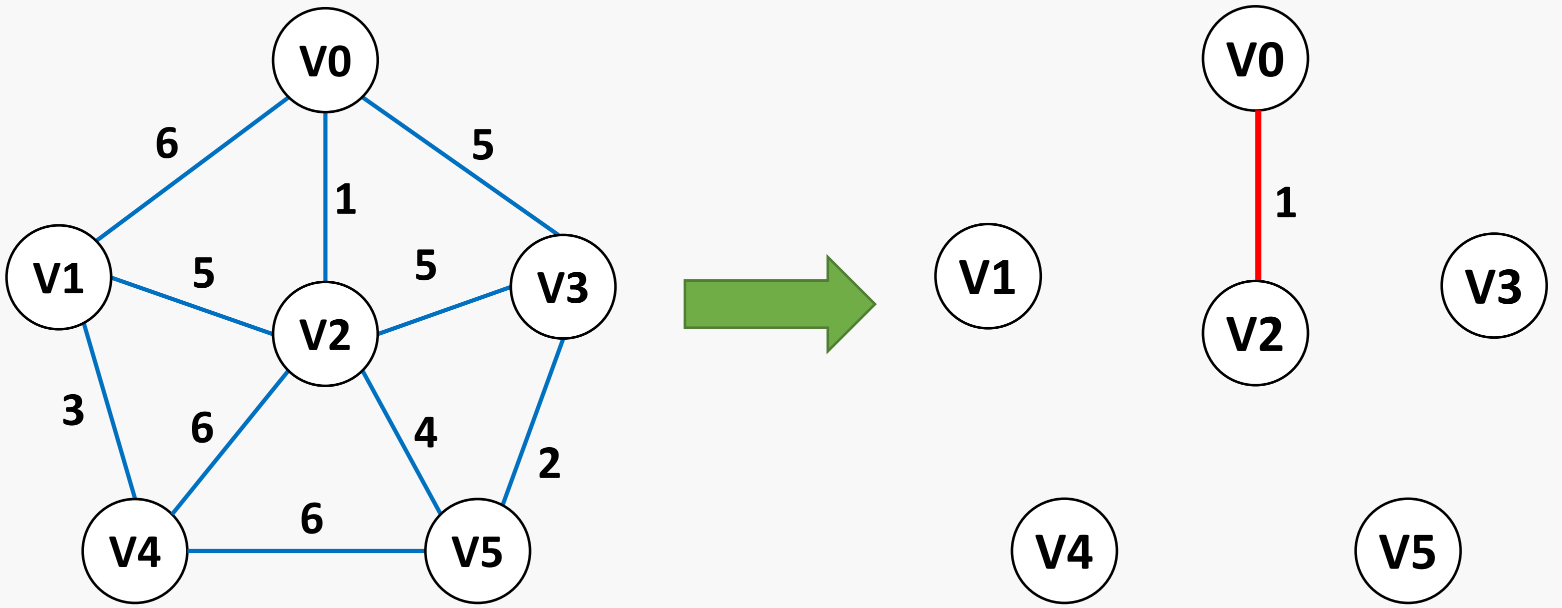
最小生成树——普里姆算法描述

设U为生成树顶点的集合，初始状态仅包含起点u_0

- 从含权连通图的某一顶点 u_0 出发，选择与它关联的具有最小权值的边（u_0,v），将其顶点 v 加入到生成树的顶点集合U中，同时记录边（u_0,v）为生成树的一部分
- 从一个顶点在U中（记作u），而另一个顶点不在U中（记作v）的各条边中选择权值最小的边（u,v），把顶点 v 加入集合U，同时记录边（u,v）为生成树的一部分
- 将上面步骤反复作下去，直到图中的所有顶点都加入到生成树顶点集合U中

最小生成树——克鲁斯卡尔算法

- 克鲁斯卡尔算法从另一途径求最小生成树，该算法着眼于边考虑解法

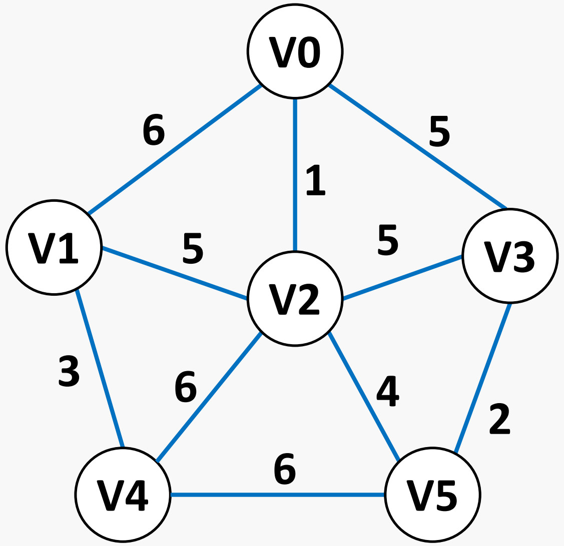


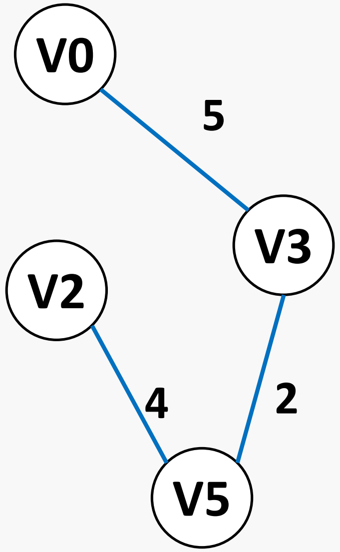
可以证明，最小生成树必然包含边(V0,V2)

- 将原图拆解为互不相连的N个连通分量(单结点树)，考察所有**连接不同分量的边**
- 为了使生成树边上权值之和最小，**根据贪心的原则**，这里**取所有连接不同分量的边中权值最小的边** **(V0,V2)**作为生成树的一部分

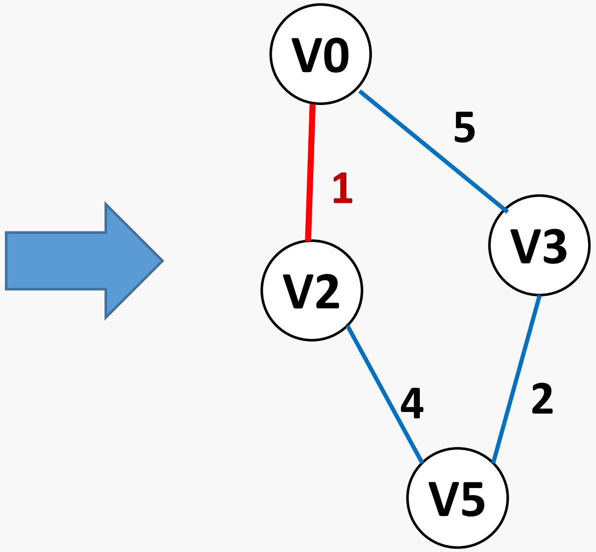
最小生成树——克鲁斯卡尔算法

- 为什么最小生成树必然包含边(V0,V2)?
  - 假设最小生成树**不包含**(V0,V2)，那么在生成树中V0必然通过其他路径与V2连通。取出**生成树中V0→V2的路径**，假设如图所示
  - 删除V0到其他点的边，添加边(V0,V2)





生成树中
V0→V2的路径

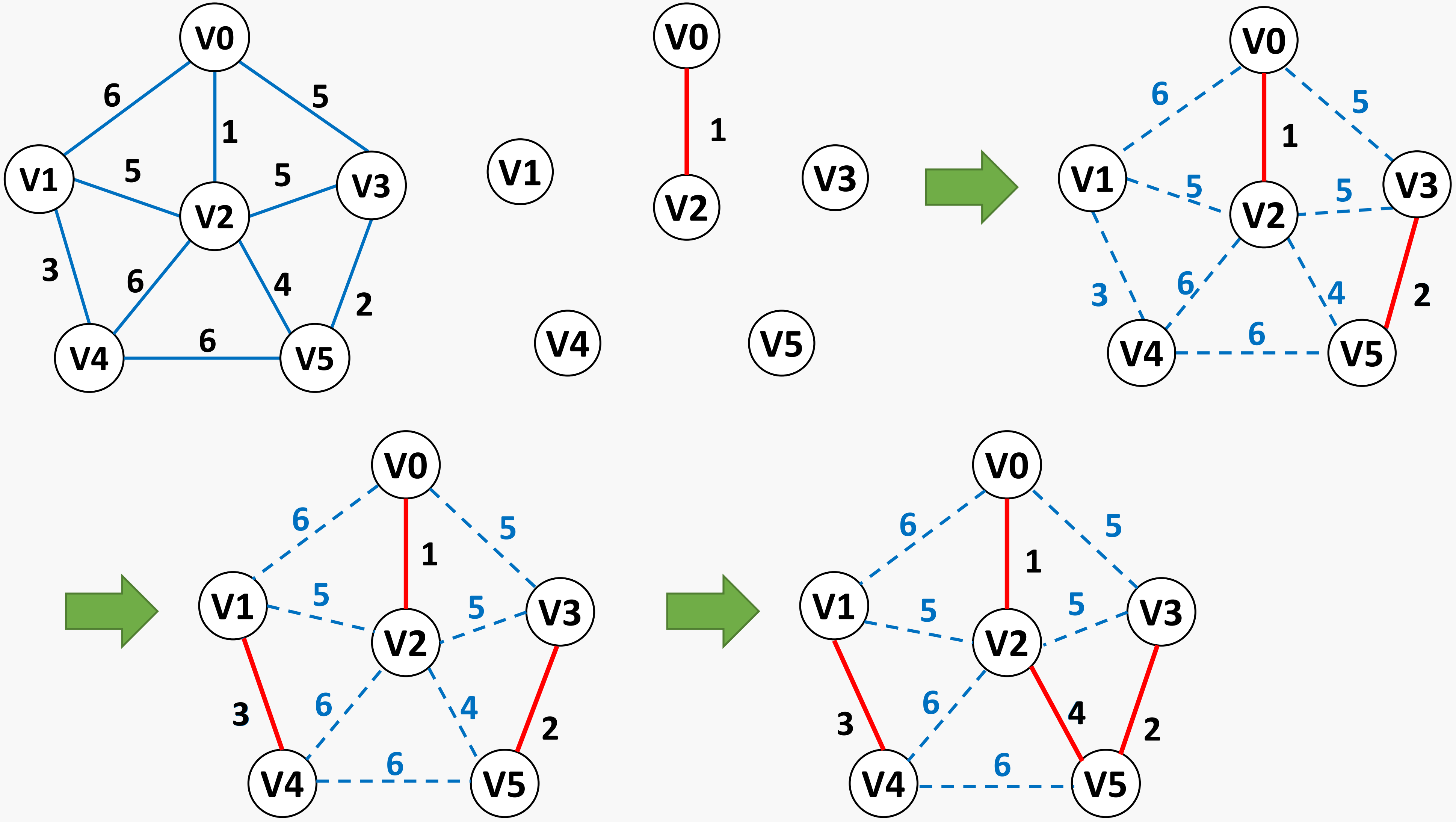


修改生成树中
V0→V2路径

修改后的生成树权值之和更小，假设不成立

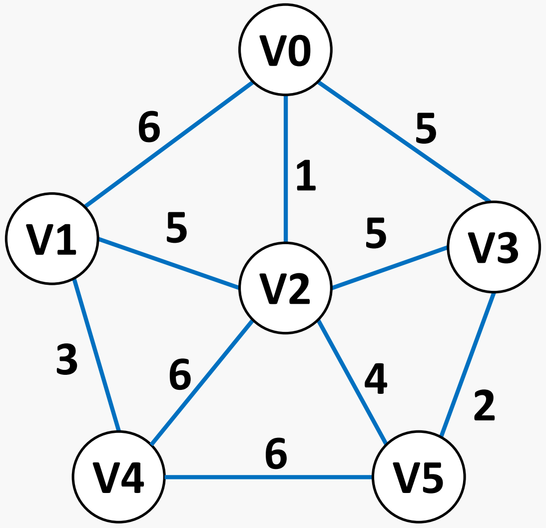
最小生成树——克鲁斯卡尔算法

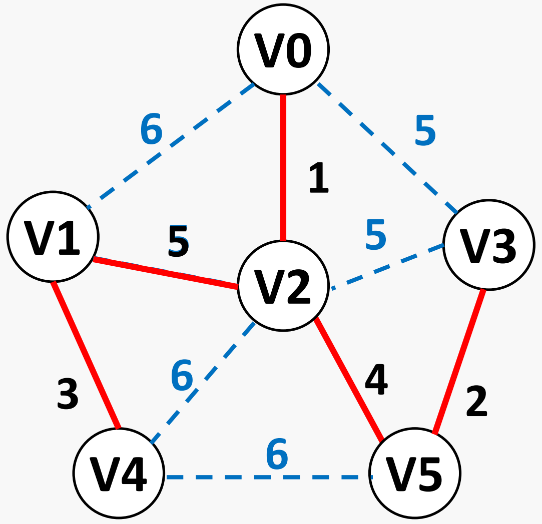
- 按克鲁斯卡尔算法，继续**选择连接两个不同分量的权值最小的边**



最小生成树——克鲁斯卡尔算法

◆ 按克鲁斯卡尔算法，继续**选择连接两个不同分量的权值最小的边**





最小生成树——克鲁斯卡尔算法总结

- 假设含权连通N=(V,{E})，则令最小生成树的初始状态为只有n个顶点而无边的非连通图T=(V,{})，图中每个顶点自成一个连通分量
- 在E中选择代价最小的边，若该边依附的顶点落在T中不同的连通分量上，则将此边加入到T中，否则舍去此边而选择下一条代价最小的边
- 以此类推，直至T中所有顶点都在同一连通分量上为止

问题求解与实践
——最小生成树编程要点

主讲教师：陈雨亭、沈艳艳

最小生成树编程问题

最小生成树算法

- 普里姆算法
- 克鲁斯卡尔算法 ✓

存储结构

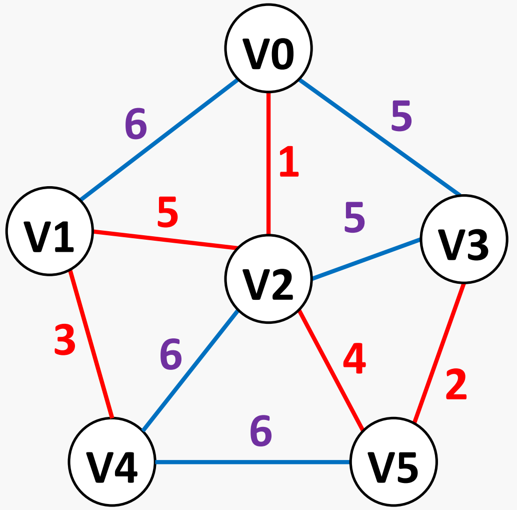
- 邻接表
- 邻接矩阵 ✓
- ……

图的存储方式

- 首先定义邻接矩阵表示图

```
typedef  struct
{
    int arc[MAXVEX][MAXVEX];      //邻接矩阵
    int numVertexes, numEdges;    //顶点数，边数
}MGraph;
```

其中，MAXVEX为预定义常数，arc[][] 存储边的权重



邻接矩阵

0	6	1	5	M	M
6	0	5	M	3	M
1	5	0	5	6	4
5	M	5	0	M	2
M	3	6	M	0	6
M	M	4	2	6	0

M为大整数
表示不直连

存储边的结构体

- 克鲁斯卡尔算法的要点是找出连接不同分量且权值最小的边，这将涉及到将所有的边按照权重大小排序
- 在矩阵中以上排序操作不方便编程，所以定义下面结构体存储边的信息

```
typedef  struct
{
        int begin;
        int end;
        int weight;
}Edge;

// 定义存放边的数组
Edge edges[MAXEDGE];
```

$$\begin{matrix} 0 & 6 & 1 & 5 & M & M \\ 6 & 0 & 5 & M & 3 & M \\ 1 & 5 & 0 & 5 & 6 & 4 \\ 5 & M & 5 & 0 & M & 2 \\ M & 3 & 6 & M & 0 & 6 \\ M & M & 4 & 2 & 6 & 0 \end{matrix}$$

下标	起点	终点	权重
0	0	1	6
1	0	2	1
2	0	3	5
3	1	2	5
4	1	4	3
5	2	3	5
6	2	4	6
7	2	5	4
8	3	5	2
9	4	5	6

- 程序运行时，需要将邻接矩阵中边的信息读入edges[]

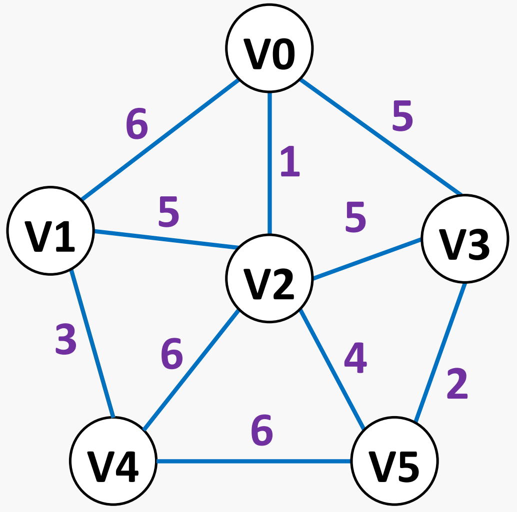
模拟程序执行过程的

(1) 将邻接矩阵中边的信息读入edges[]

(2) 将数组中的边按权重从小到大排序

(3) 依次读取数组中的边，并做如下操作：

- 若该边连接不同分量则边加入生成树
- 否则，跳过此边，处理下一条边



edges 数组

下标	起点	终点	权重
0	0	2	1
1	3	5	2
2	1	4	3
3	2	5	4
4	0	3	5
5	1	2	5
6	2	3	5
7	0	1	6
8	2	4	6
9	4	5	6

是否连接不同分量?

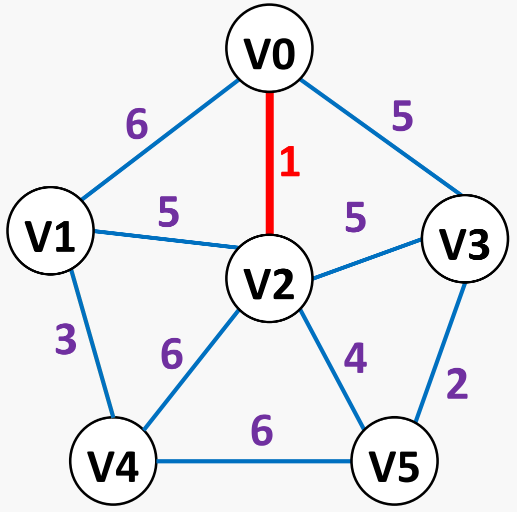
模拟程序执行过程的

(1) 将邻接矩阵中边的信息读入edges[]

(2) 将数组中的边按权重从小到大排序

(3) 依次读取数组中的边，并做如下操作：

- 若该边连接不同分量则边加入生成树
- 否则，跳过此边，处理下一条边



edges 数组

下标	起点	终点	权重
0	0	2	1
1	3	5	2
2	1	4	3
3	2	5	4
4	0	3	5
5	1	2	5
6	2	3	5
7	0	1	6
8	2	4	6
9	4	5	6

是否连接不同分量?

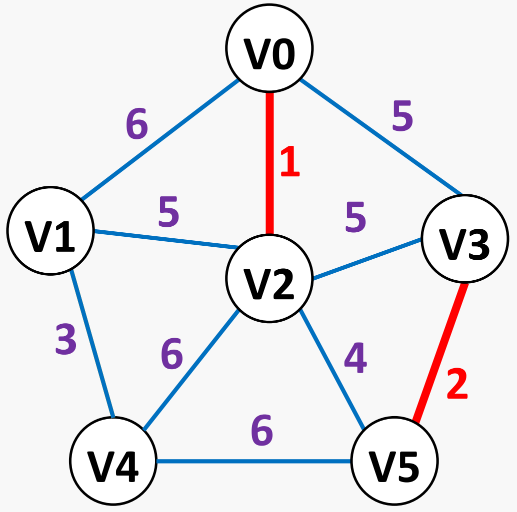
模拟程序执行过程的

(1) 将邻接矩阵中边的信息读入edges[]

(2) 将数组中的边按权重从小到大排序

(3) 依次读取数组中的边，并做如下操作：

- ➢ 若该边连接不同分量则边加入生成树
- ➢ 否则，跳过此边，处理下一条边



edges 数组

下标	起点	终点	权重
0	0	2	1
1	3	5	2
2	1	4	3
3	2	5	4
4	0	3	5
5	1	2	5
6	2	3	5
7	0	1	6
8	2	4	6
9	4	5	6

是否连接不同分量?

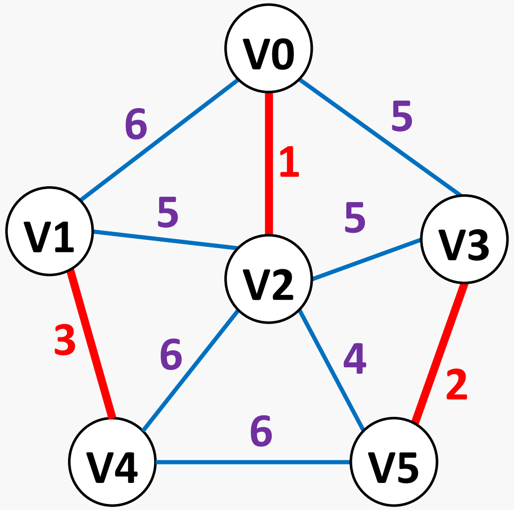
模拟程序执行过程的

(1) 将邻接矩阵中边的信息读入edges[]

(2) 将数组中的边按权重从小到大排序

(3) 依次读取数组中的边，并做如下操作：

- 若该边连接不同分量则边加入生成树
- 否则，跳过此边，处理下一条边



edges 数组

下标	起点	终点	权重
0	0	2	1
1	3	5	2
2	1	4	3
3	2	5	4
4	0	3	5
5	1	2	5
6	2	3	5
7	0	1	6
8	2	4	6
9	4	5	6

是否连接不同分量?

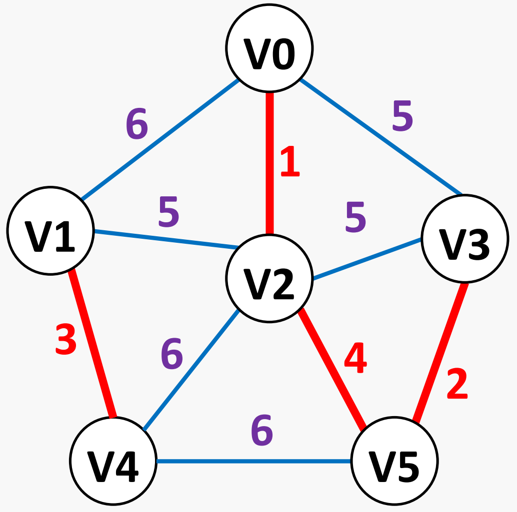
模拟程序执行过程的

(1) 将邻接矩阵中边的信息读入edges[]

(2) 将数组中的边按权重从小到大排序

(3) 依次读取数组中的边，并做如下操作：

- 若该边连接不同分量则边加入生成树
- 否则，跳过此边，处理下一条边



edges 数组

下标	起点	终点	权重
0	0	2	1
1	3	5	2
2	1	4	3
3	2	5	4
4	0	3	5
5	1	2	5
6	2	3	5
7	0	1	6
8	2	4	6
9	4	5	6

是否连接不同分量?

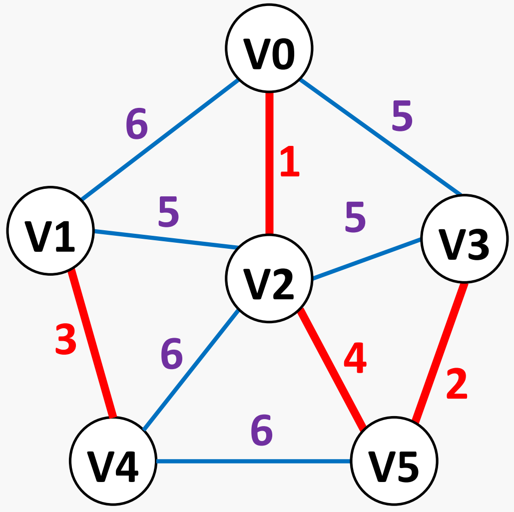
模拟程序执行过程的

(1) 将邻接矩阵中边的信息读入edges[]

(2) 将数组中的边按权重从小到大排序

(3) 依次读取数组中的边，并做如下操作：

➢ 若该边连接不同分量则边加入生成树
➢ 否则，跳过此边，处理下一条边



edges 数组

下标	起点	终点	权重
0	0	2	1
1	3	5	2
2	1	4	3
3	2	5	4
4	0	3	5
5	1	2	5
6	2	3	5
7	0	1	6
8	2	4	6
9	4	5	6

是否连接不同分量?

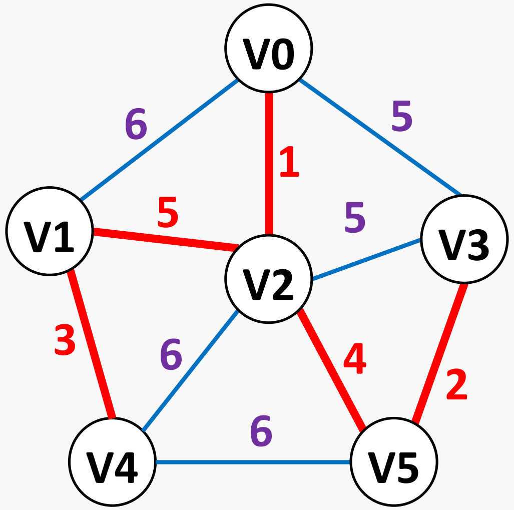
模拟程序执行过程的

(1) 将邻接矩阵中边的信息读入edges[]

(2) 将数组中的边按权重从小到大排序

(3) 依次读取数组中的边，并做如下操作：

- 若该边连接不同分量则边加入生成树
- 否则，跳过此边，处理下一条边



edges 数组

下标	起点	终点	权重
0	0	2	1
1	3	5	2
2	1	4	3
3	2	5	4
4	0	3	5
5	1	2	5
6	2	3	5
7	0	1	6
8	2	4	6
9	4	5	6

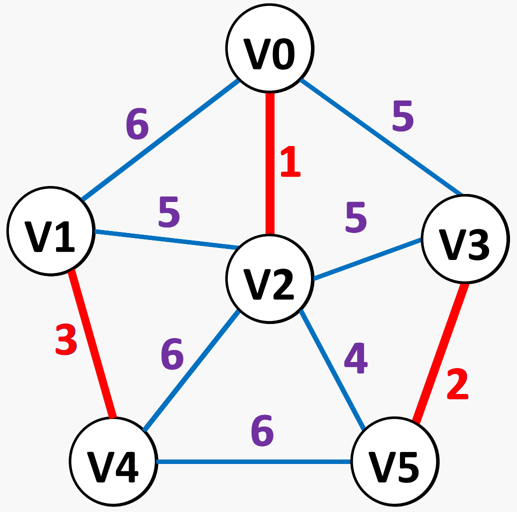
已选择
N-1条边

如何判断一条边连接两个不同分量？

- 可以在每个分量中设一个根。对一条边而言，如果其起点和终点所在分量的根不同，则这条边可以加入生成树
- **编程方案**：定义一个数组parent[]，其中parent[i]存储 i 号点的父结点或根结点 。初始状态parent[]值全为-1，表示 i 号点的根就是自己

0	1	2	3	4	5
-1	-1	0	-1	1	3

生成树每加入一条边后，执行 **parent**[终点]=起点

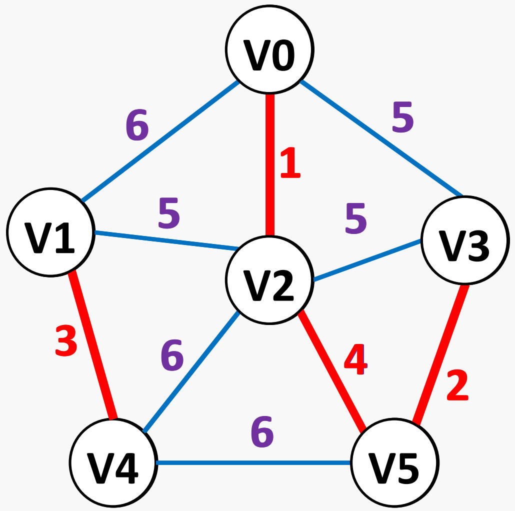


- 加入第1条边(V0,V2)，执行parent[2]=0
- 加入第2条边(V3,V5)，执行parent[5]=3
- 加入第3条边(V1,V4)，执行parent[4]=1

如何判断一条边连接两个不同分量?

0	1	2	3	4	5
-1	-1	0	0	1	3

生成树每加入一条边后，
执行 **parent[终点]=起点**

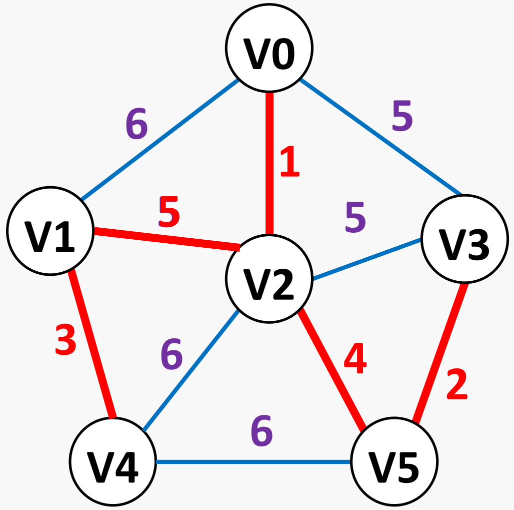


- 加入第4条边(V2,V5)
  查V2的根：parent[2]为0 -> parent[0]为-1，根为0
  查V5的根：parent[5]为3 -> parent[3]为-1，根为3
  边(V2,V5)连接不同分量，可加入，执行parent[3]=0
  【注】执行parent[终点的根]=起点的根
- **可以加入边(V0,V3)吗?**
  因parent[0]为-1，V0根为0
  查V3的根：parent[3]为0 -> parent[0]为-1，根为0
  所以V0和V3的根相同，**舍弃**

如何判断一条边连接两个不同分量？

0	1	2	3	4	5
1	-1	0	0	1	3

生成树每加入一条边后，
执行 **parent[终点]=起点**



- 加入第5条边(V1,V2)
  查V1的根：parent[1]为-1，根为1
  查V2的根：parent[2]为0 -> parent[0]为-1，根为0
  边(V1,V2)连接不同分量，可加入，执行parent[0]=1

```
// 找到根的函数
int Find(int *parent, int f) {
    while (parent[f] >= 0) {
        f = parent[f];
    }
    return f;
}
```

注意 parent[] 并未存储树结构

克鲁斯卡尔算法核心代码

```
int parent[MAXVEX];          //用于寻找根节点的数组，初始化为 -1
Edge edges[MAXEDGE];         //定义存储边的数组
// 初始化 edges 数组
for (i = 0; i < 顶点数-1; i++)
        for (j = i + 1; j <顶点数-1; j++)
            读入邻接矩阵边的信息到 edges[]
Sort(edges，边数);      // Sort函数为边数组Edge排序
for (i = 0; i < 边数; i++) {
        n = Find(parent, edges[i].begin);    //寻找边的起点所在树的根
        m = Find(parent, edges[i].end);      //寻找边的终点所在树的根
        // 两个顶点不在一棵子树内
        if (n != m)  {   parent[m] = n;     记录生成树的边(n, m);    }
}
```