法律声明

- 本课件包括演示文稿、示例、代码、题库、视频和声音等内容，小象学院和主讲老师拥有完全知识产权的权利；只限于善意学习者在本课程使用，不得在课程范围外向任何第三方散播。任何其他人或机构不得盗版、复制、仿造其中的创意及内容，我们保留一切通过法律手段追究违反者的权利。

- 课程详情请咨询
  - 微信公众号：小象
  - 新浪微博：ChinaHadoop

图(上)

——基本问题

小象学院 ChinaHadoop.cn 邹博

主要内容

- 并查集
- 图的存储
- 深度优先搜索
  - 计算割点/割边
- 最短路径
  - Dijkstra
  - A^*算法：启发式搜索
  - Floyd/Bellman-Ford
- 最小生成树 (MST)
  - Prim/Kruskal

例：进制推断

- 如果公式84*148=B6A8成立，则该公式采用的是________进制表示的。
  - A：15
  - B：11
  - C：12
  - D：14
  - E：16

84*148=B6A8

- 常规做法：
- 假定数值是x进制的，则写出等式：

$$(8x+4)\cdot(x^2+4x+8)=11x^3+6x^2+10x+8$$

$$\Rightarrow (3x^2+6x+2)\cdot(x-12)=0$$

$$\Rightarrow x=12$$

84*148=B6A8

- "启发式"做法：
- 在十进制体系下，左侧个位乘积4*8=32；
- 右侧个位为8，差32-8=24，从而进制必然是24的约数。只有C选项12是24的约数。

  - 以上分析，用"十进制体系"仅仅是计算习惯。
  - 这种做法可以辅助第1种解法做交互验证。

图的表述与搜索

- 图的表示
  - 邻接矩阵
    - n*n的矩阵，有边是1，无边是0
  - 邻接表
    - 为每个点建立一个链表(数组)存放与之连接的点
- 搜索
  - BFS (Breadth First Search) 广(宽)度优先
  - DFS (Depth First Search) 深度优先

图的划分

- 某国家有N个小岛组成，经过多年的基础设施积累，若干岛屿之间建立了若干桥梁。先重新完善该国的行政区划，规定只要有桥梁连接的岛屿则归属同一个城市(可以通过其他岛屿中转)，问该国一共多少个城市？
  - 求给定图G的连通分量的数目。
- 解决方案：集合划分

集合的表达

- 将全集I划分为若干子集$S_1,S_2,\ldots,S_n$，使得$S_1 \cup S_2 \cup \ldots \cup S_n=I$，且$S_1 \cap S_2 \cap \ldots \cap S_n=\emptyset$。则$S_1,S_2,\ldots,S_n$称为全集I的一个可行划分。
- 思考：用何种数据结构方便表达呢?

集合的性质

- 自反性(a)，对称性(a、b)
- 传递性：如果c是b的子结点，b是a的子结点，则a、b、c位于相同的集合中。
- 由于集合中元素是等价的，选择任意一个元素作为代表，其他元素都指向该元素，即完成一个集合的表达；称指向元素为子结点，被指向元素为父结点。

Code

```cpp
class CUnionFindSet
{
private:
	int m_nN;
	int* m_pParent;
public:
	CUnionFindSet(int n);
	~CUnionFindSet();
	void Union(int i, int j);
	int Find(int i);
	void Print() const;
};

CUnionFindSet::CUnionFindSet(int n)
{
	m_nN = n;
	m_pParent = new int[m_nN];
	for(int i = 0; i < m_nN; i++)
		m_pParent[i] = i;
}

CUnionFindSet::~CUnionFindSet()
{
	if(m_pParent != NULL)
	{
		delete[] m_pParent;
		m_pParent = NULL;
	}
}
```

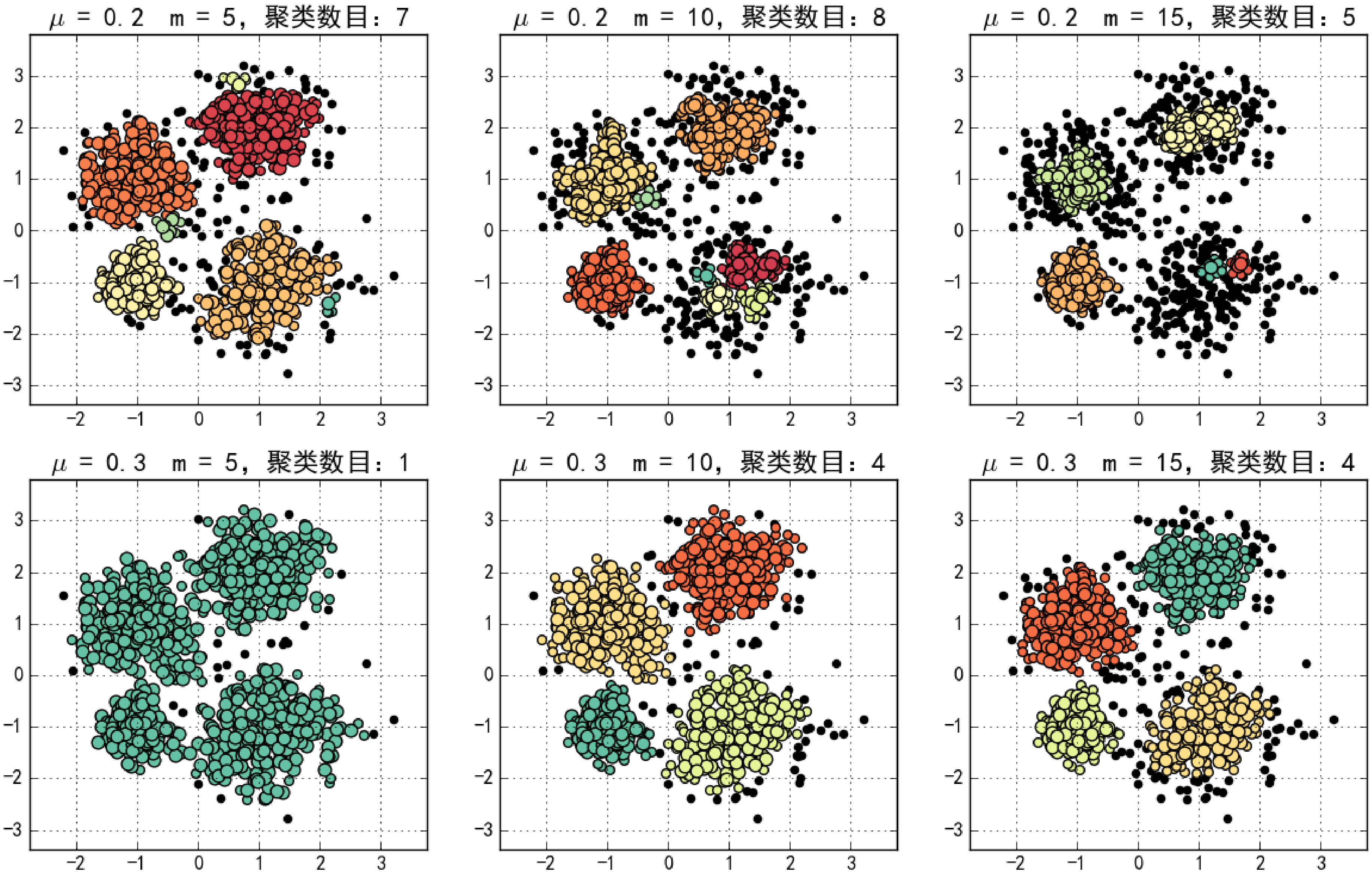
Code

```
int CUnionFindSet::Find(int i)
{
    if((i < 0) || (i >= m_nN))
        return -1;

    int root = i;
    while(root != m_pParent[root])  //尚未到根节点
    {
        root = m_pParent[root];
    }

    int t = i;
    int p;
    while(t != root)
    {
        p = m_pParent[t];   //t2的原父节点
        m_pParent[t] = root;    //t2的父节点直接赋值给根t
        t = p;  //沿着原来的父节点继续向上查找
    }

    return root;
}

void CUnionFindSet::Union(int i, int j)
{
    if((i < 0) || (i >= m_nN) || (j < 0) || (j >= m_nN))
        return;

    int ri = Find(i);
    int rj = Find(j);
    if(ri != rj)
        m_pParent[ri] = rj;
}
```

Code

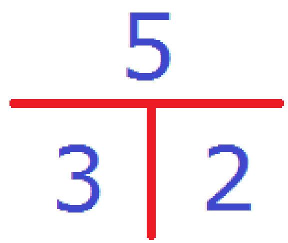
```
int CalcCompoment()
{
    const int N = 10;   //岛屿数量
    CUnionFindSet ufs(N);
    ufs.Union(2,6); //根据边初始化并查集
    ufs.Union(5,6);
    ufs.Union(1,8);
    ufs.Union(2,9);
    ufs.Union(5,3);
    ufs.Union(4,8);
    ufs.Union(4,0);

    int* component = new int[N];
    memset(component, 0, N*sizeof(int));
    int i;
    for(i = 0; i < N; i++)  //计算每个岛屿的“首府”
        component[ufs.Find(i)]++;

    int nComponent = 0; //统计“首府”的数目
    for(i = 0; i < N; i++)
    {
        if(component[i] != 0)
            nComponent++;
    }
    delete[] component;
    return nComponent;
}
```

DBSCAN聚类

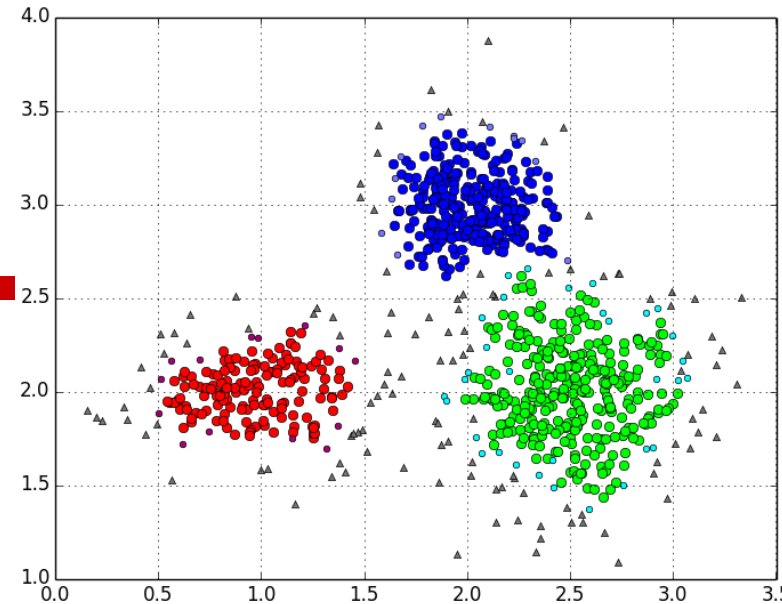


μ = 0.2 m = 5，聚类数目：7
μ = 0.2 m = 10，聚类数目：8
μ = 0.2 m = 15，聚类数目：5
μ = 0.3 m = 5，聚类数目：1
μ = 0.3 m = 10，聚类数目：4
μ = 0.3 m = 15，聚类数目：4
3
2
1
0
−1
−2
−3
−2
−1
0
1
2
3

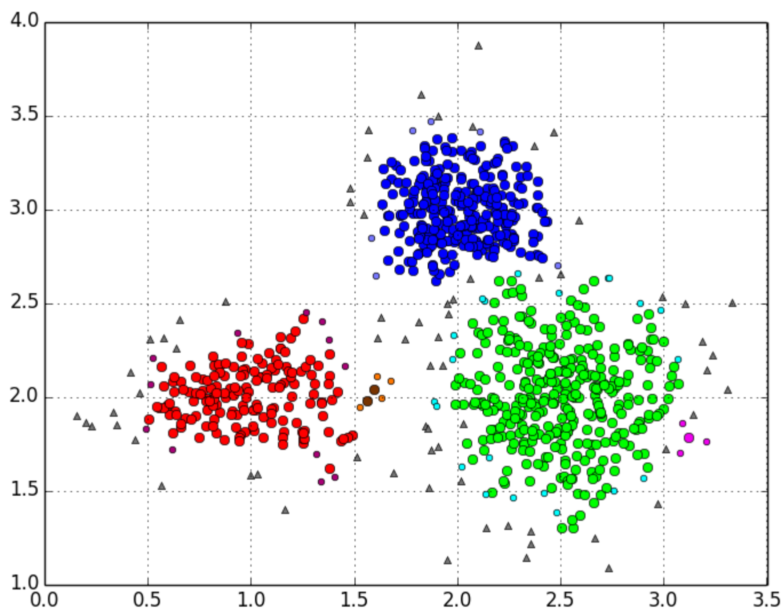
密度聚类：r=0.1



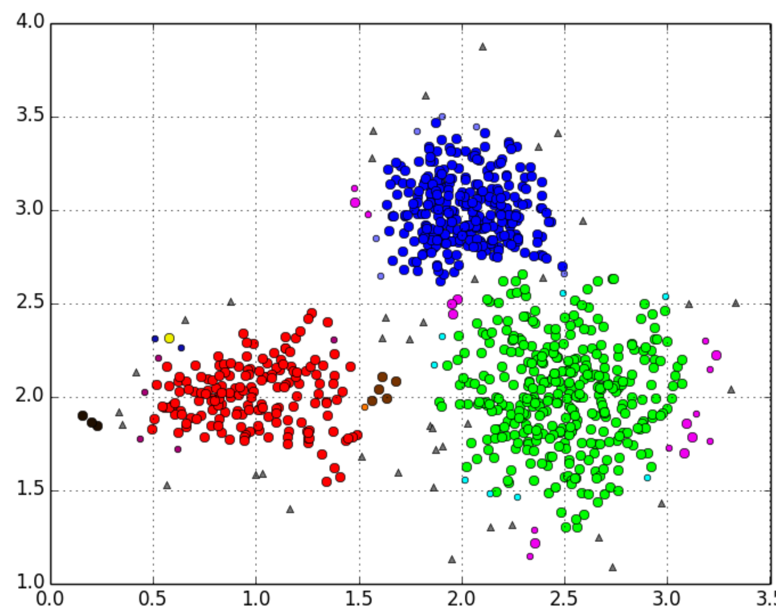
5
3
2



4.0
3.5
3.0
2.5
2.0
1.5
1.0
0.0
0.5
1.0
1.5
2.0
2.5
3.0
3.5



4.0
3.5
3.0
2.5
2.0
1.5
1.0
0.0
0.5
1.0
1.5
2.0
2.5
3.0
3.5

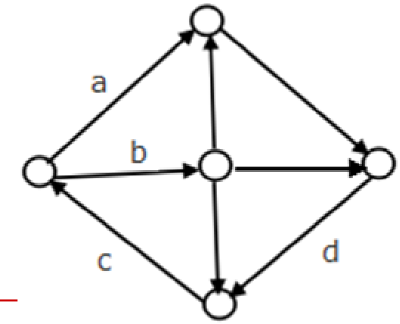


4.0
3.5
3.0
2.5
2.0
1.5
1.0
0.0
0.5
1.0
1.5
2.0
2.5
3.0
3.5

Code

```python
def density_cluster(data):
    # 计算核心对象
    m = len(data)       # 样本数目
    sim = [[] for i in range(m)] * m   # sim[i]:第i个样本的近邻
    r2 = r * r
    for i in range(m):  # 距离
        print i
        for j in range(m):
            if distance2(data[i], data[j]) < r2:
                sim[i].append(j)

    # 根据核心对象合并
    uf = uf_init(m)   # 并查集初始化
    for i in range(m):
        if len(sim[i]) > kernel_num: # i是核心对象
            for j in sim[i]:
                uf_union(uf, i, j)

    # 计算并查集的每个对象所属类型
    types = {}
    t = 0
    for i in range(m):
        c = uf_find(uf, i)
        if not types.has_key(c):
            if c != i or len(sim[i]) > kernel_num:   # 去除孤立点
                t += 1
                types[c] = t

    # 根据字典映射关系，计算每个样本所属的簇
    cluster = [0] * m
    for i in range(m):
        c = uf_find(uf, i)
        if (c != i) or (len(sim[i]) > kernel_num):
            if len(sim[i]) > kernel_num:  # 是核心对象
                cluster[i] = types[c]
            else:
                cluster[i] = k+types[c]
    return cluster
```

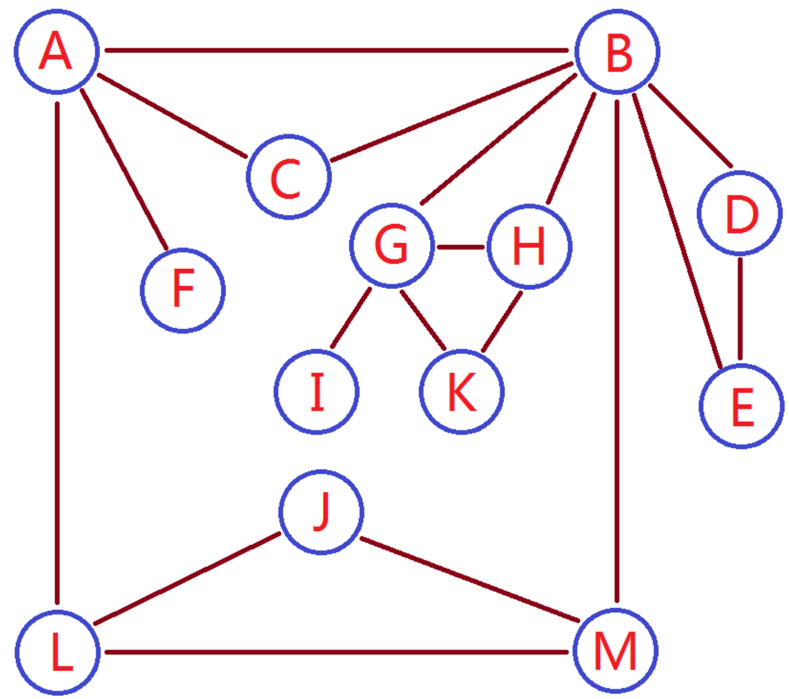
思考：有向图呢？



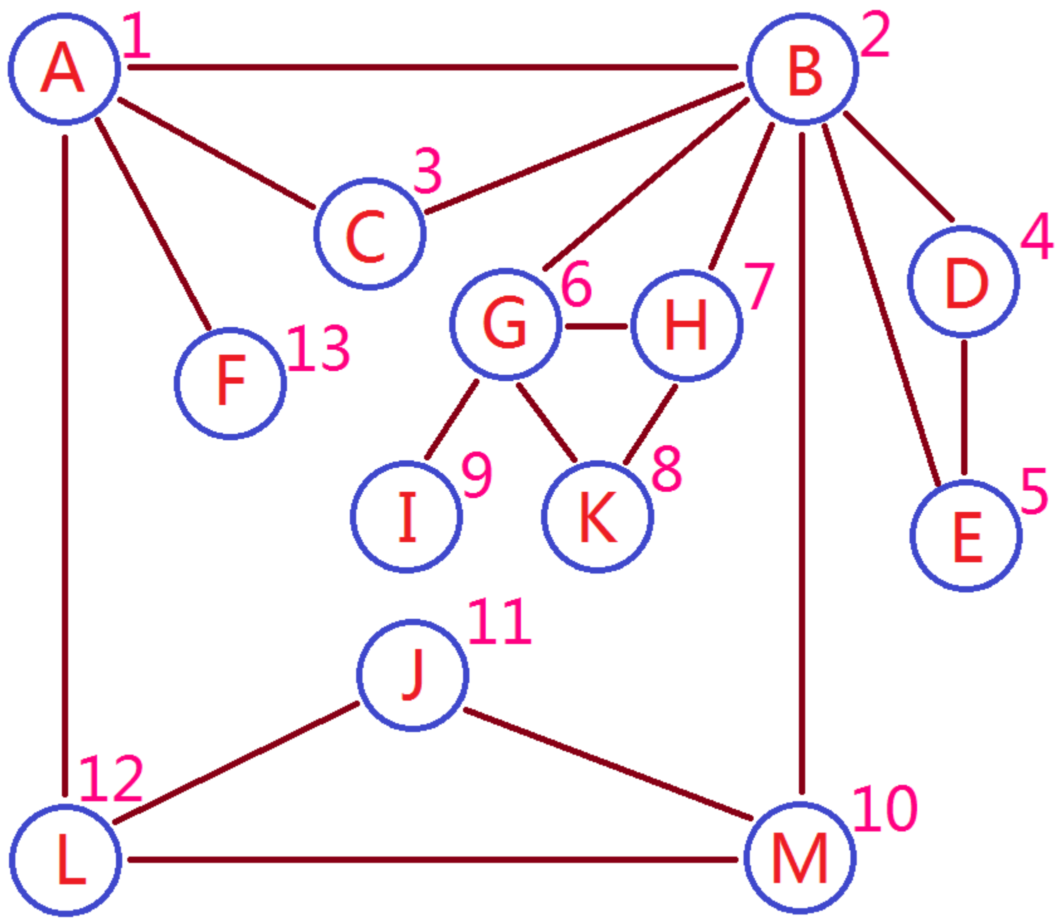
- 对一个有向图，如果每个节点都存在到达其他任何节点的路径，那么就称它是强连通的。
- 判断图是强连通图的算法：任取有向图G的某结点S，从S开始进行深度优先搜索，若可以遍历G的所有结点，则将G的所有边反向，再次从S开始进行深度优先搜索，如果再次能够遍历G的所有结点，则G是强连通图，两次搜索有一次无法遍历所有结点，则G不是强连通图。此外，上述搜索可以换成广度优先搜索等其他方案。

计算割点

- 给定某无向连通图G，若删除某结点X以及与X邻接的所有边，图G变成非连通图，则结点X称为图G的割点。
- 给定某图的邻接矩阵或邻接表，如何计算该图的所有割点？



深度优先的访问顺序



算法探索

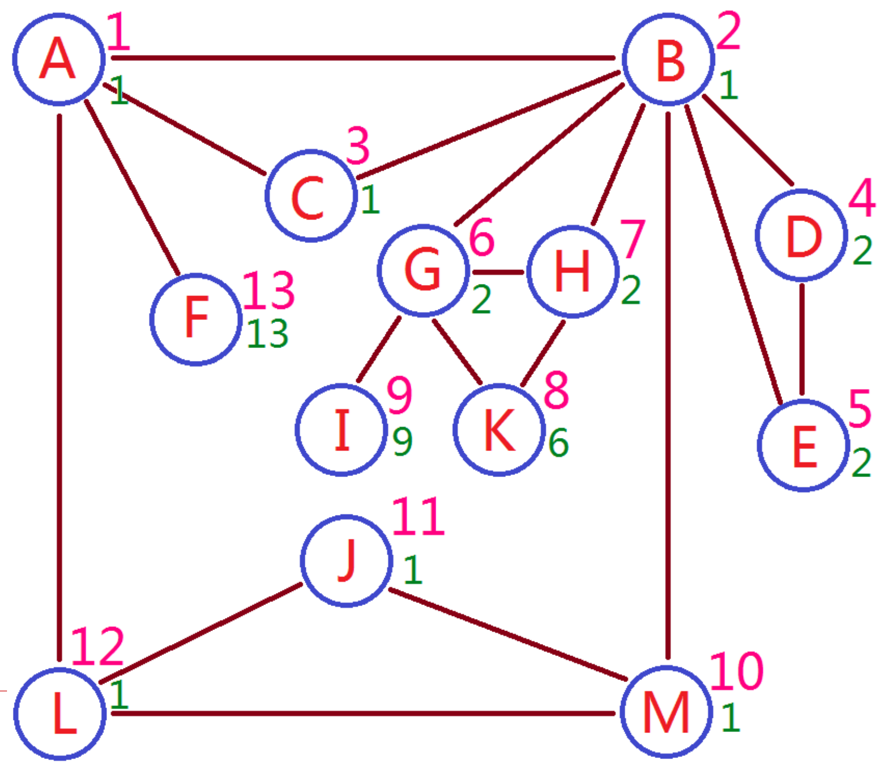
- 使用深度优先搜索，当前刚刚访问结点i，遍历结点i的相邻结点j：
- 若j尚未访问，则如果不通过结点i，结点j直接连通的最早结点假定为k；
  - 如果结点k的访问顺序比i早，说明结点j可以绕过结点i；
  - 如果结点k的访问顺序比j晚，说明要想访问结点j，必须经过结点i——即：结点i为割点。
- 特殊处理根结点。
  - 若根节点的访问分支多于两个，则根结点为割点。

割点隔边算法：记录回溯到的最早结点

A	B	C	D	E	F	G	H	I	J	K	L	M
1	2	3	4	5	13	6	7	9	11	8	12	10
1	1	1	2	2	13	2	2	9	1	6	1	1

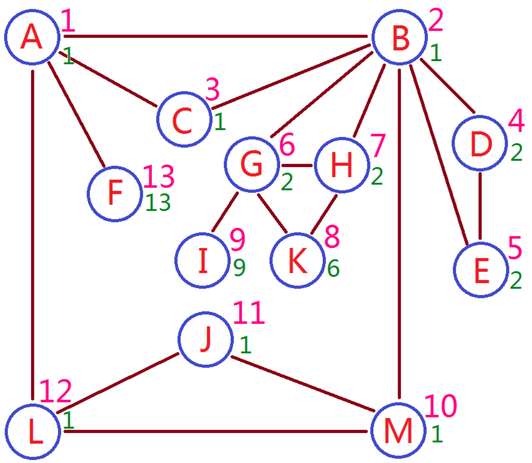
割点： A B G

隔边： (G, I) (A, F)



Code

```
int _tmain(int argc, _TCHAR* argv[])
{
    int N = 13;
    vector<vector<int> > graph(N, vector<int>(N));
    int i,j;
    ifstream iFile("Graph.txt");
    for(i = 0; i < N; i++)
        for(j = 0; j < N; j++)
            iFile >> graph[i][j];
    bool* an = new bool[N];
    fill(an, an+N, false);
    ArticulationPoint(graph, an);
    Print(an, N);
    return 0;
}
```

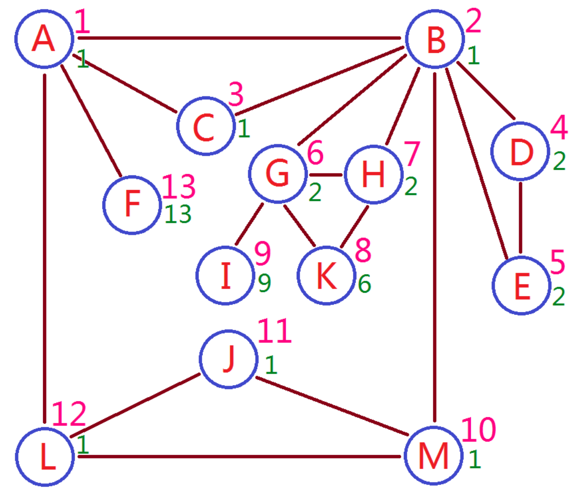



```
void ArticulationPoint(const vector<vector<int> >& graph, bool* an)
{
    int N = (int)graph.size();
    int* dfn = new int[N];  //dfn[i]=j: 第i个节点是第j个被访问的
    fill(dfn, dfn+N, -1);   //所有结点尚未访问
    int* low = new int[N];  //low[i]=j: i及其子孙回溯的最早结点为j
    int root = 0;   //任意指定一个结点作为根节点
    int n = 0;      //访问编号
    _ArticulationPoint(graph, root, -1, root, dfn, low, n, an);
    delete[] dfn;
    delete[] low;
}
```

Code

A B G

```
void _ArticulationPoint(const vector<vector<int> >& graph, int i, int parent,
    int root, int* dfn, int* low, int& n, bool* an)
{
    dfn[i] = low[i] = n;
    n++;
    int N = (int)graph.size();
    int child = 0;
    for(int j = 0; j < N; j++)
    {
        if(graph[i][j] != 0)
        {
            if(dfn[j] == -1)     //尚未访问
            {
                child++;
                _ArticulationPoint(graph, j, i, root, dfn, low, n, an);
                low[i] = min(low[i], low[j]);
                if((i != root) && (low[j] >= dfn[i]))
                    an[i] = true;
                else if((i == root) && (child == 2))//本质是child>=2
                    an[i] = true;
            }
            else if(parent != j)     //已访问过，说明j是i的父结点或祖先结点
                                     //若此时i的父结点不是j，则j是i的祖先结点
            {
                low[i] = min(low[i], dfn[j]);
            }
        }
    }
}
```



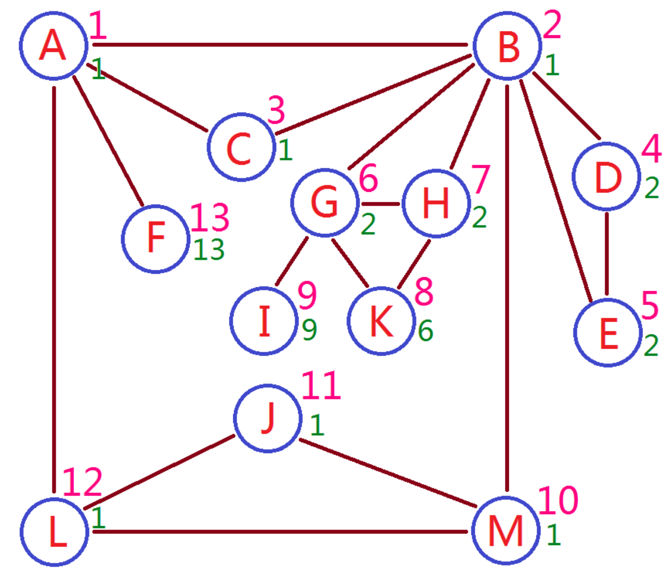
进一步思考

- 给定某无向连通图G，若删除某边E，则图G变成非连通图，则边E称为图G的割边。
- 问：给定某图的邻接矩阵或邻接表，如果计算该图的所有割边？

Code

```
void ArticulationEdge(const vector<vector<int> >& graph, list<pair<int, int> >& ae)
{
    int N = (int)graph.size();
    int* dfn = new int[N];  //dfn[i]=j：第i个节点是第j个被访问的
    fill(dfn, dfn+N, -1);   //所有结点尚未访问
    int* low = new int[N];  //low[i]=j：i及其子孙回溯的最早结点为j
    int n = 0;       //访问编号
    _ArticulationEdge(graph, 0, -1, dfn, low, n, ae);
    delete[] dfn;
    delete[] low;
}
```

```
void _ArticulationEdge(const vector<vector<int> >& graph, int i, int parent,
    int* dfn, int* low, int& n, list<pair<int, int> >& ae)
{
    dfn[i] = low[i] = n;
    n++;
    int N = (int)graph.size();
    for(int j = 0; j < N; j++)
    {
        if(graph[i][j] != 0)
        {
            if(dfn[j] == -1)    //尚未访问
            {
                _ArticulationEdge(graph, j, i, dfn, low, n, ae);
                low[i] = min(low[i], low[j]);
                if(low[j] > dfn[i])
                    ae.push_back(make_pair(i, j));
            }
            else if(parent != j)
            {
                low[i] = min(low[i], dfn[j]);
            }
        }
    }
}
```

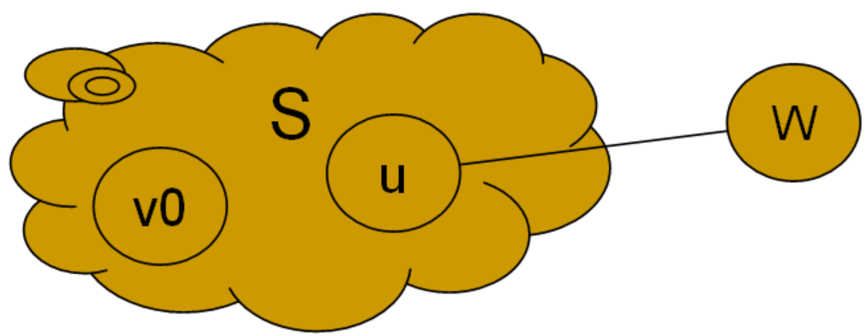


(G,I)

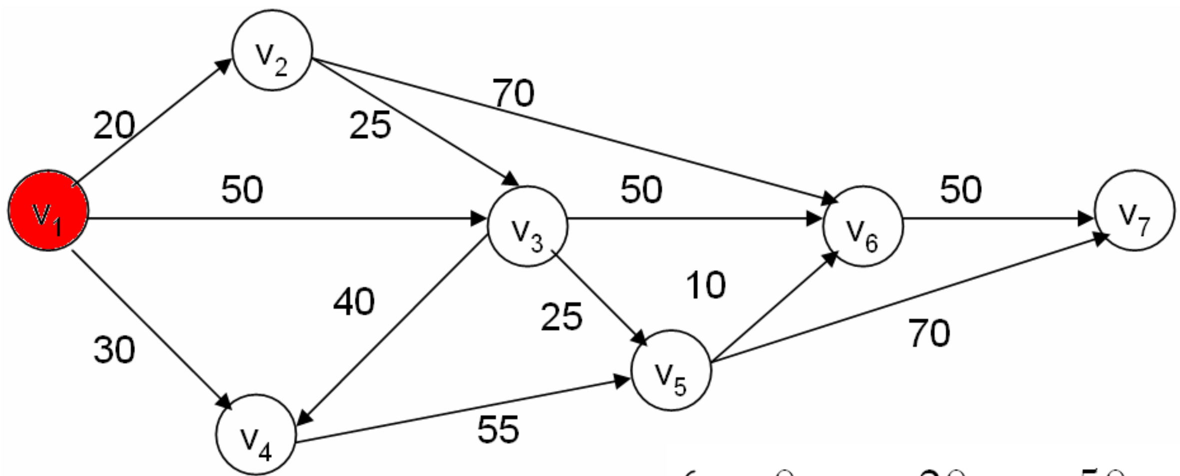
(A,F)

最短路径SPF：Shortest Path First(Dijkstra)

- 将Prim算法稍做调整即得Dijkstra算法：
  - 结点集V初始化为源点S一个元素：V={S}，到每个点的最短路径的距离初始化为dist[u]=graph[S][u]；
  - 选择最小的dist[u]：则v是当前找到的不在V中且距离S最近的结点，更新V=V∪{v}，调整dist[u]=min{dist[u],dist[v]+graph[v][u]}；
  - 重复n-1次。

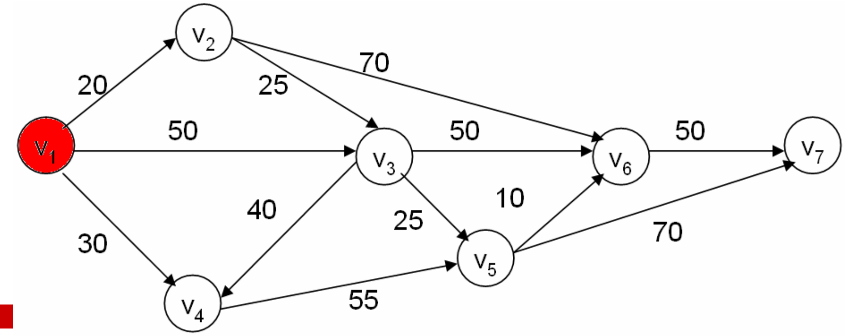


最短路径示例



$$
\begin{pmatrix}
0 & 20 & 50 & 30 & +\infty & +\infty & +\infty \\
+\infty & 0 & 25 & +\infty & +\infty & 70 & +\infty \\
+\infty & +\infty & 0 & 40 & 25 & 50 & +\infty \\
+\infty & +\infty & +\infty & 0 & 55 & +\infty & +\infty \\
+\infty & +\infty & +\infty & +\infty & 0 & 10 & 70 \\
+\infty & +\infty & +\infty & +\infty & +\infty & 0 & 50 \\
+\infty & +\infty & +\infty & +\infty & +\infty & +\infty & 0
\end{pmatrix}
$$

最短路径示例



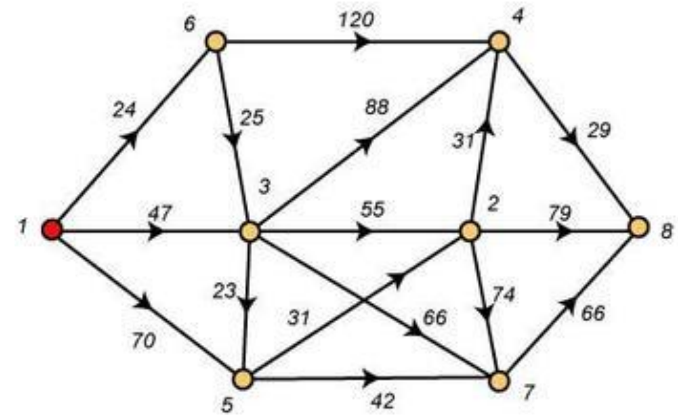
迭代	选取的结点	S	DIST						
			(1)	(2)	(3)	(4)	(5)	(6)	(7)
置初值	—	1	0	20	50	30	+∞	+∞	+∞
1	2	1,2	0	20	45	30	+∞	90	+∞
2	4	1,2,4	0	20	45	30	85	90	+∞
3	3	1,2,4,3	0	20	45	30	70	90	+∞
4	5	1,2,4,3,5	0	20	45	30	70	90	140
5	6	1,2,4,3,5,6	0	20	45	30	70	90	130

- 算法的执行在有n-1个结点加入到S中后终止，此时求出了v0至其它各结点的最短路径。
- 问题：如何求出所有这些最短路径？
  - 记录前驱

Code

```
//graph：邻接矩阵，src：源点，sp[i]：src到i的最短路径的长度
void Dijkstra(const vector<vector<int> >& graph, int src, vector<int>& sp)
{
    int N = (int)graph.size();
    vector<bool> s(N, false);   //集合S初始化为空
    sp = graph[src];            //从src到任意点的直接路径

    //将src加入集合S
    s[src] = true;
    sp[src] = 0;
    int num,k,j;
    int mp;
    for(num = 1; num < N; num++)    //将剩余的N-1个结点加入到集合S
    {
        k = -1;     //候选结点编号
        mp = -1;    //最短路径值
        for(j = 0; j < N; j++)
        {
            if(s[j])
                continue;
            if((mp < 0) || (mp > sp[j]))
            {
                mp = sp[j];
                k = j;
            }
        }
        s[k] = true;
        for(j = 0; j < N; j++)
        {
            if(s[j])
                continue;
            sp[j] = min(sp[j], sp[k]+graph[k][j]);
        }
    }
}
```

```
//graph: 邻接矩阵, src: 源点, sp[i]: src到i的最短路径的长度
void Dijkstra2(const vector<vector<int> >& graph, int src, vector<int>& sp, vector<int>& pre)
{
    int N = (int)graph.size();
    vector<bool> s(N, false);    //集合S初始化为空
    sp = graph[src];             //从src到任意点的直接路径

    //将src加入集合S
    s[src] = true;
    sp[src] = 0;
    int num,k,j;
    int mp;
    for(num = 1; num < N; num++)     //将剩余的N-1个结点加入到集合S
    {
        k = -1;     //候选结点编号
        mp = -1;    //最短路径值
        for(j = 0; j < N; j++)
        {
            if(s[j])
                continue;
            if((mp < 0) || (mp > sp[j]))
            {
                mp = sp[j];
                k = j;
            }
        }
        s[k] = true;
        for(j = 0; j < N; j++)
        {
            if(s[j])
                continue;
            if(sp[j] > sp[k]+graph[k][j])
            {
                sp[j] = sp[k]+graph[k][j];
                pre[j] = k;
            }
        }
    }
}
```




```
void MinPath(int start, int end,
    const vector<int>& pre, vector<int>& path)
{
    path.push_back(end);
    int e = end;
    while(start != e)
    {
        e = pre[e];
        path.push_back(e);
    }
    reverse(path.begin(), path.end());
}
```

0:	0	1
1:	101	1 → 5 → 2
2:	47	1 → 3
3:	132	1 → 5 → 2 → 4
4:	70	1 → 5
5:	24	1 → 6
6:	112	1 → 5 → 7
7:	161	1 → 5 → 2 → 4 → 8

启发式搜索：A*算法

- 给定有向图G，求从S到E的最短路径。
- 思考：从S到E的路径探索中，假定当前位于结点i，则与i相连的结点中，应选择哪个？
- 定义 $f[j]:=g[j]+h[j]$
  - g[j]：从起始结点S到当前结点j的距离
  - h[j]：从当前结点j到终止结点E的距离
- 则，应选择f[j]最小的结点作为i的后继。

城市名称	中心坐标x	中心坐标y
北京	12956000	4824875
上海	13522000	3641093
广州	12614437	2631281
深圳	12694500	2562500
成都	11585750	3567500
天津	13046000	4714250
西安	12127500	4039281
南京	13222250	3747750
重庆	11857375	3424562
香港	12705437	2539812
澳门	12640003	2518359
合肥	13055906	3722562
安庆	13029953	3550359
蚌埠	13067218	3862125
巢湖	13121625	3690312
池州	13077937	3566328
滁州	13170531	3782062
阜阳	12891343	3859500
亳州	12888750	3988093
淮北	13003281	4000531
淮南	13026219	3825375
黄山	13170781	3445859
六安	12968312	3708375
马鞍山	13190562	3698750
宿州	13021000	3957750
铜陵	13114718	3603000
芜湖	13177156	3654093
宣城	13218718	3604484
福州	13280531	2990640
龙岩	13027531	2870218
南平	13156296	3058828
宁德	13308171	3060968
莆田	13248500	2911625
泉州	13203500	2846875
三明	13093156	3011906
厦门	13147125	2794937
漳州	13097281	2798562
兰州	11554484	4283500
白银	11596093	4350625
定西	11645968	4217000
甘南	11455500	4137875
嘉峪关	10939781	4808312
金昌	11376000	4623625
酒泉	10966171	4799953

结点的类别

- 结点分成三类：活跃结点、待计算结点和计算完成结点。
- 每次从活跃结点集合中取出f值最小的结点j
  - 对于j的相邻结点k，
    - 更新g[k]=min(g[k], g[k]>g[j]+graph[j][k])
    - 更新f[k]=g[k]+h[k]
    - k归类到活跃结点
  - j归类到计算完成结点
- 注意：某结点即使是计算完成结点，仍然有可能因为g[k]的更新回到活跃结点。

Code

```
//计算start到end的最短距离，其中graph[i][j]为i-j之间的距离，p[i]为第i个城市的坐标和名称
void AStar(const vector<vector<float> >& graph, int start, int end, const vector<CPoint>& p)
{
    int N = (int)graph.size();
    int* type = new int[N];     //0: 新点, 1: 边点, 2: 内点
    float* f = new float[N];    //经过f[i]的start到end的估计值
    float* g = new float[N];    //从start到g[i]的精确值
    float* h = new float[N];    //从h[i]到end的估计值
    int* pre = new int[N];      //pre[i]:i的前驱

    //初始化
    int k;
    for(k = 0; k < N; k++)
    {
        g[k] = graph[start][k];                 //到点距离
        h[k] = CPoint::Distance(p[k], p[end]);  //离点距离
        f[k] = g[k] + h[k];                     //总距离
        pre[k] = -1;
        type[k] = 0;
    }

    //start加入边点集合
    type[start] = 1;
    g[start] = 0;
    f[start] = h[start];

    float d;
    int j = start;
    while(j != end) //尚未到达end
    {
        //选择边点集合中f[i]最小的
        j = -1;
        for(k = 0; k < N; k++)
        {
            if(type[k] != 1)    //k必须是边点
                continue;
            if((j == -1) || (d > f[k]))
            {
                d = f[k];
                j = k;
            }
        }

        //计算新到点距离g[i]+graph[i][j]是否比原到点距离小
        for(k = 0; k < N; k++)
        {
            if(type[k] == 0)
            {
                if(g[k] > g[j] + graph[j][k])
                {
                    g[k] = g[j] + graph[j][k];
                    f[k] = g[k] + h[k];
                }
                type[k] = 1;
                pre[k] = j;
            }
            else if(type[k] == 2)   //i是内点
            {
                if(g[k] > g[j] + graph[j][k])
                {
                    g[k] = g[j] + graph[j][k];
                    f[k] = g[k] + h[k];
                    type[k] = 1;
                    pre[k] = j;
                }
            }
            else if(type[k] == 1)
            {
                if(g[k] > g[j] + graph[j][k])
                {
                    g[k] = g[j] + graph[j][k];
                    f[k] = g[k] + h[k];
                    pre[k] = j;
                }
            }
        }
        type[j] = 2;    //边点变成内点
    }

    //恢复路径
    vector<int> path;
    path.push_back(end);
    while(pre[end] != -1)
    {
        end = pre[end];
        path.push_back(end);
    }
    reverse(path.begin(), path.end());

    //输出路径的结点
    float m = 0;
    for(k = 0; k < (int)path.size(); k++)
    {
        cout << p[path[k]].name << '\n';
        if(k != 0)
            m += graph[path[k-1]][path[k]];
    }
    cout << m << '\n';
    delete[] f;
    delete[] g;
    delete[] h;
    delete[] pre;
    delete[] type;
}
```

Code1：初始化

```
//计算start到end的最短距离，其中graph[i][j]为i-j之间的距离，p[i]为第i个城市的坐标和名称
void AStar(const vector<vector<float> >& graph, int start, int end, const vector<CPoint>& p)
{
    int N = (int)graph.size();
    int* type = new int[N];     //0: 新点, 1: 边点, 2: 内点
    float* f = new float[N];    //经过f[i]的start到end的估计值
    float* g = new float[N];    //从start到g[i]的精确值
    float* h = new float[N];    //从h[i]到end的估计值
    int* pre = new int[N];      //pre[i]:i的前驱

    //初始化
    int k;
    for(k = 0; k < N; k++)
    {
        g[k] = graph[start][k];                     //到点距离
        h[k] = CPoint::Distance(p[k], p[end]);      //离点距离
        f[k] = g[k] + h[k];                         //总距离
        pre[k] = -1;
        type[k] = 0;
    }

    //start加入边点集合
    type[start] = 1;
    g[start] = 0;
    f[start] = h[start];
```

Code2：迭代

```
float d;
int j = start;
while(j != end) //尚未到达end
{
    //选择边点集合中f[i]最小的
    j = -1;
    for(k = 0; k < N; k++)
    {
        if(type[k] != 1)     //k必须是边点
            continue;
        if((j == -1) || (d > f[k]))
        {
            d = f[k];
            j = k;
        }
    }
```



```
    for(k = 0; k < N; k++)
    {
        if(graph[j][k] >= INFINITY)
            continue;
        if(type[k] == 0)
        {
            if(g[k] > g[j] + graph[j][k])
            {
                g[k] = g[j] + graph[j][k];
                f[k] = g[k] + h[k];
            }
            type[k] = 1;
            pre[k] = j;
        }
        else if(type[k] == 2)    //i是内点
        {
            if(g[k] > g[j] + graph[j][k])
            {
                g[k] = g[j] + graph[j][k];
                f[k] = g[k] + h[k];
                type[k] = 1;
                pre[k] = j;
            }
        }
        else if(type[k] == 1)
        {
            if(g[k] > g[j] + graph[j][k])
            {
                g[k] = g[j] + graph[j][k];
                f[k] = g[k] + h[k];
                pre[k] = j;
            }
        }
    }
    type[j] = 2;    //边点变成内点
}
```

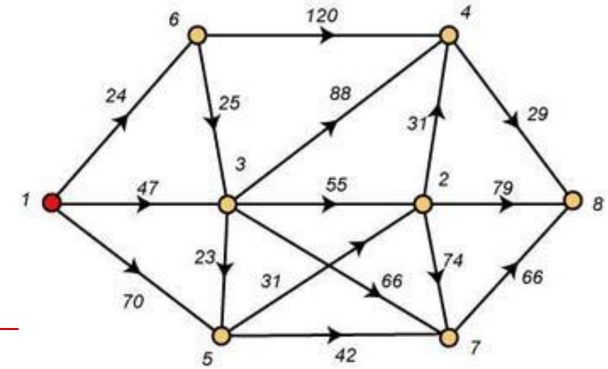
Code3：计算路径本身

北京
沧州
泰安
徐州
蚌埠
铜陵
衢州
宁德
台北
2108.69

```
//恢复路径
vector<int> path;
path.push_back(end);
while(pre[end] != -1)
{
    end = pre[end];
    path.push_back(end);
}
reverse(path.begin(), path.end());

//输出路径的结点
float m = 0;
for(k = 0; k < (int)path.size(); k++)
{
    cout << p[path[k]].name << '\n';
    if(k != 0)
        m += graph[path[k-1]][path[k]];
}
cout << m << '\n';
```

城市名称	中心坐标x	中心坐标y
北京	12956000	4824875
上海	13522000	3641093
广州	12614437	2631281
深圳	12694500	2562500
成都	11585750	3567500
天津	13046000	4714250
西安	12127500	4039281
南京	13222250	3747750
重庆	11857375	3424562
香港	12705437	2539812
澳门	12640003	2518359
合肥	13055906	3722562
安庆	13029953	3550359
蚌埠	13067218	3862125
巢湖	13121625	3690312
池州	13077937	3566328
滁州	13170531	3782062
阜阳	12891343	3859500
亳州	12888750	3988093
淮北	13003281	4000531
淮南	13026219	3825375
黄山	13170781	3445859
六安	12968312	3708375
马鞍山	13190562	3698750
宿州	13021000	3957750
铜陵	13114718	3603000
芜湖	13177156	3654093
宣城	13218718	3604484
福州	13280531	2990640
龙岩	13027531	2870218
南平	13156296	3058828
宁德	13308171	3060968
莆田	13248500	2911625
泉州	13203500	2846875
三明	13093156	3011906
厦门	13147125	2794937
漳州	13097281	2798562
兰州	11554484	4283500
白银	11596093	4350625
定西	11645968	4217000
甘南	11455500	4137875
嘉峪关	10939781	4808312
金昌	11376000	4623625
酒泉	10966171	4799953

Floyd算法



- Floyd算法又称插点法，用于计算图中任意两点的最短路径。
- 记dist[i,j]为结点i到j的最短路径的距离，则

$$dist[i,j]=\min_{0\le k<n}\{dist[i,k]+dist[k,j],dist[i,j]\}$$

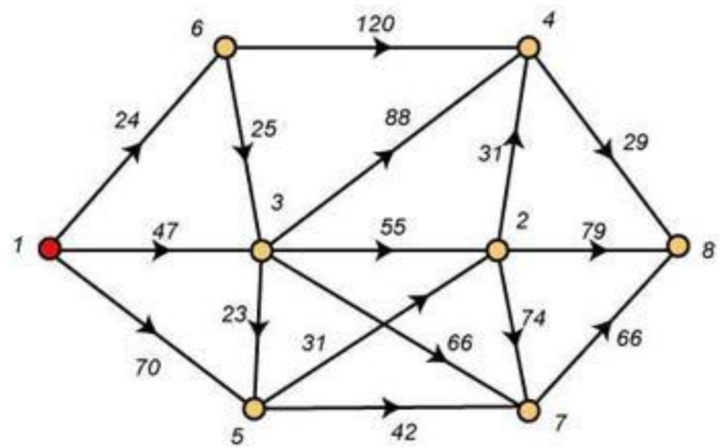
  - i,j,k取值范围为0到N-1，时间复杂度为$O(n^3)$
- 如果图中存在负的权值，算法也是适用的。
  - 思考：Dijkstra算法允许边存在负权吗？

Floyd Code

```
void Floyd(const vector<vector<int> >& graph, vector<vector<int> >& sp)
{
    sp = graph;
    int size = (int)graph.size();
    int k, i, j;
    for(k = 0; k < size; k++)
    {
        for(i = 0; i < size; i++)
        {
            for(j = 0; j < size; j++)
            {
                if(sp[i][j] > sp[i][k] + sp[k][j])
                {
                    sp[i][j] = sp[i][k] + sp[k][j];
                }
            }
        }
    }
}
```

Code

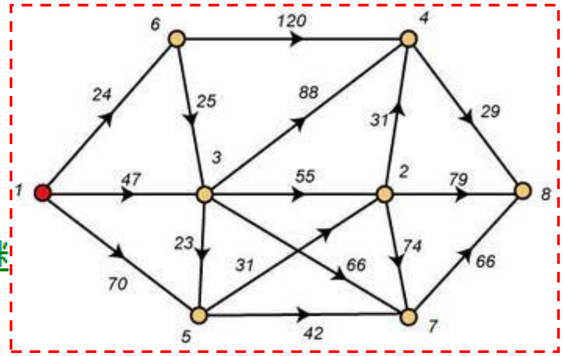
```
int _tmain(int argc, _TCHAR* argv[])
{
    const int N = 8;
    vector<vector<int> > graph(N, vector<int>(N, INFINITY));
    graph[0][5] = 24;
    graph[0][2] = 47;
    graph[0][4] = 70;
    graph[1][3] = 31;
    graph[1][6] = 74;
    graph[1][7] = 79;
    graph[2][1] = 55;
    graph[2][3] = 88;
    graph[2][4] = 23;
    graph[2][6] = 66;
    graph[3][7] = 29;
    graph[4][1] = 31;
    graph[4][6] = 42;
    graph[5][2] = 25;
    graph[5][3] = 120;
    graph[6][7] = 66;
    vector<vector<int> > sp(N, vector<int>(N, INT_MAX));     //最短路径的值
    vector<vector<int> > next(N, vector<int>(N, -1));        //直接后继
    Floyd2(graph, sp, next);

    //输出所有结点间的最短路径
    vector<int> path;
    int i,j;
    for(i = 0; i < N; i++)
        for(j = 0; j < N; j++)
        {
            path.clear();
            MinPath(i, j, next, path);
            Print(i, j, sp[i][j], path);
        }
    return 0;
}
```




```
void MinPath(int start, int end,
    const vector<vector<int> >& next, vector<int>& path)
{
    path.push_back(start);
    int s = start;
    while(s != end)
    {
        s = next[s][end];
        path.push_back(s);
    }
}
```

Code2

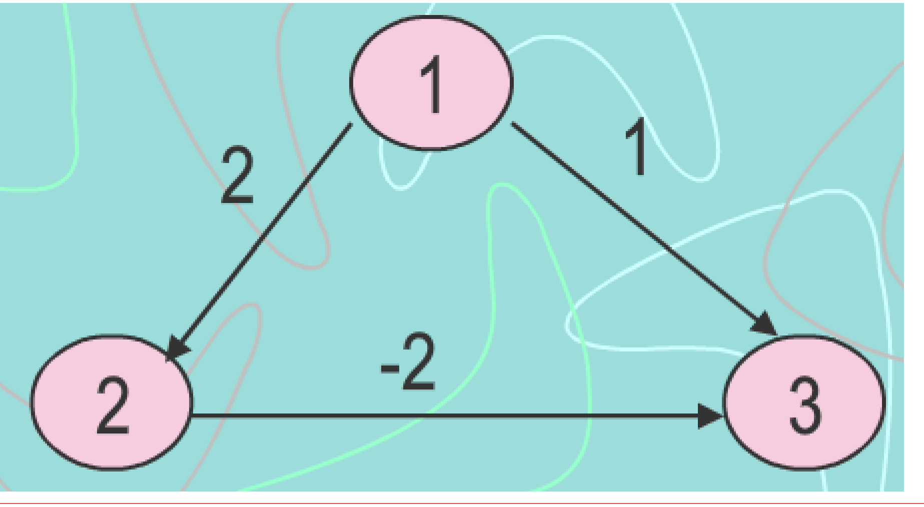
```
void Floyd2(const vector<vector<int> >& graph,
    vector<vector<int> >& sp, vector<vector<int> >& next)
{
    sp = graph;
    int size = (int)graph.size();
    int k, i, j;
    for(i = 0; i < size; i++)
        for(j = 0; j < size; j++)
            next[i][j] = j; //直接后继

    for(k = 0; k < size; k++)
    {
        for(i = 0; i < size; i++)
        {
            for(j = 0; j < size; j++)
            {
                if(sp[i][j] > sp[i][k] + sp[k][j])
                {
                    sp[i][j] = sp[i][k] + sp[k][j];
                    next[i][j] = next[i][k];
                }
            }
        }
    }
}
```



(1 , 2):	101	1 → 5 → 2
(1 , 3):	47	1 → 3
(1 , 4):	132	1 → 5 → 2 → 4
(1 , 5):	70	1 → 5
(1 , 6):	24	1 → 6
(1 , 7):	112	1 → 5 → 7
(1 , 8):	161	1 → 5 → 2 → 4 → 8
(2 , 4):	31	2 → 4
(2 , 7):	74	2 → 7
(2 , 8):	60	2 → 4 → 8
(3 , 2):	54	3 → 5 → 2
(3 , 4):	85	3 → 5 → 2 → 4
(3 , 5):	23	3 → 5
(3 , 7):	65	3 → 5 → 7
(3 , 8):	114	3 → 5 → 2 → 4 → 8
(4 , 8):	29	4 → 8
(5 , 2):	31	5 → 2
(5 , 4):	62	5 → 2 → 4
(5 , 7):	42	5 → 7
(5 , 8):	91	5 → 2 → 4 → 8
(6 , 2):	79	6 → 3 → 5 → 2
(6 , 3):	25	6 → 3
(6 , 4):	110	6 → 3 → 5 → 2 → 4
(6 , 5):	48	6 → 3 → 5
(6 , 7):	90	6 → 3 → 5 → 7
(6 , 8):	139	6 → 3 → 5 → 2 → 4 → 8
(7 , 8):	66	7 → 8

带负权的最短路径

- 计算图中最小的权值，若该权值大于0，按照Dijkstra算法正常计算；若小于0，则所有权值都加上该权值的绝对值+1，则修改后的图不再含有负权。使用Dijkstra算法计算最小路径。
- 是否可行？



带负权的最短路径：Bellman-ford算法

- 本质：动态规划
- 适用：单源结点到其他所有结点的最短路径
- 若u→v是有向边，则$d[v] \le d[u] + dis(u,v)$
- 这个操作被成为松弛操作。
- 优点：对边权无要求，可以发现负环。

Code

```
//邻接矩阵为graph，结点数目为N，计算start到其他所有点的最短路径
void BellmanFord(int graph[N][N], int N, int* path, int start)
{
    int i,j,k;
    for(i = 0; i < N; i++)
        path[i] = 1000000;

    path[start] = 0;
    for(i = 0; i < N; i++)
    {
        //对于每条边
        for(j = 0; j < N; j++)
        {
            for(k = 0; k < N; k++)
            {
                if(graph[j][k] != 0)    //说明有边
                {
                    path[k] = min(path[k], path[j] + graph[j][k]);
                }
            }
        }
    }
}
```

SPFA

```
//邻接矩阵为graph，结点数目为N，计算start到其他所有点的最短路径
void SPFA(int graph[N][N], int N, int* path, int start)
{
    int i;
    for(i = 0; i < N; i++)
        path[i] = 1000000;
    path[start] = 0;

    queue<int> q;
    q.push(0);
    int t;
    int count = 0;
    while(!q.empty())
    {
        t = q.front();
        q.pop();
        for(i = 0; i < N; i++)
        {
            if(graph[t][i] != 0)
            {
                if(path[i] > path[t] + graph[t][i]) //经过t的新路径能够更短
                {
                    path[i] = path[t] + graph[t][i];
                    q.push(i);
                }
            }
        }
    }
}
```

最小生成树MST

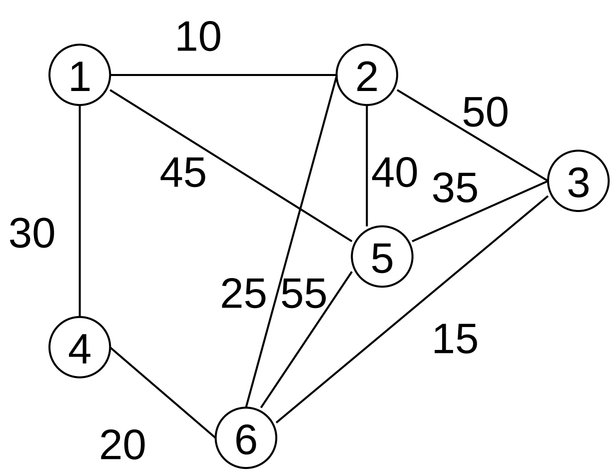
- 最小生成树要求从一个带权无向连通图中选择n－1条边并使这个图仍然连通(也即得到了一棵生成树)，同时还要考虑使树的权最小。MST最著名的是Prim算法和Kruskal算法，它们都是贪心算法。
  - Prim算法：从某个(任意一个)结点出发，选择与该结点邻接的权值最小的边；随着结点的不断加入，每次都选择这些结点发出的边中权值最小的；重复n-1次。
  - Kruskal算法：将边按照权值递增排序，每次选择权值最小并且不构成环的边，重复n-1次。

Prim算法

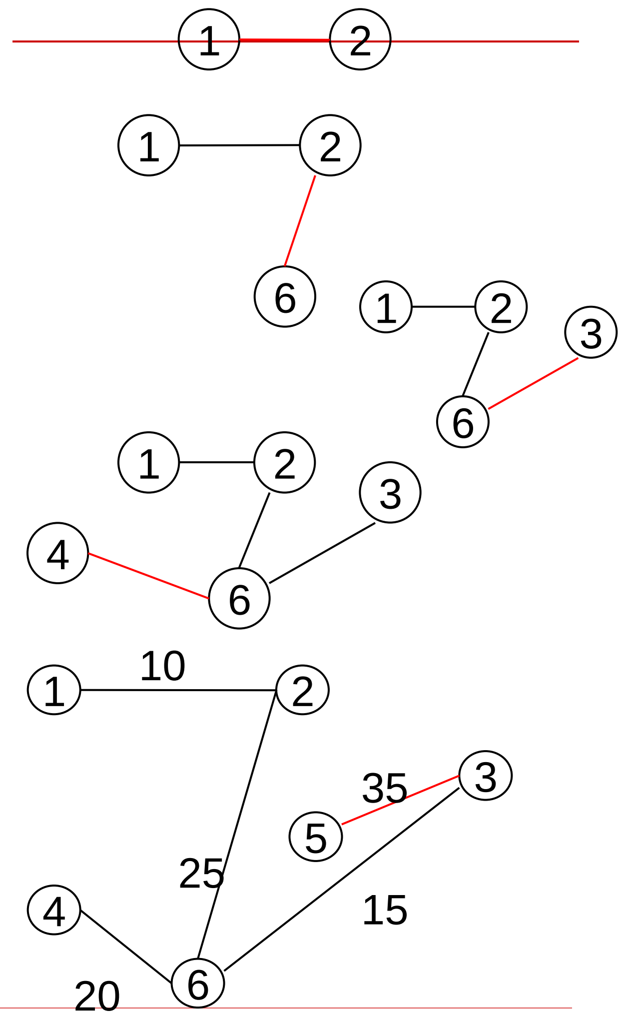
- 首先以一个结点作为最小生成树的初始结点，然后以迭代的方式找出与最小生成树中各结点权重最小边，并加入到最小生成树中。加入之后如果产生回路则跳过这条边，选择下一个结点。当所有结点都加入到最小生成树中之后，就找出了连通图中的最小生成树了。

Prim算法

使得所选择的边的集合A构成一棵树；将要计入到A中的下一条边(u,v)，应是E中一条当前不在A中且使得A∪{(u,v)}是一棵树的最小成本边。



边	成本
(1,2)	10
(2,6)	25
(3,6)	15
(6,4)	20
(3,5)	35



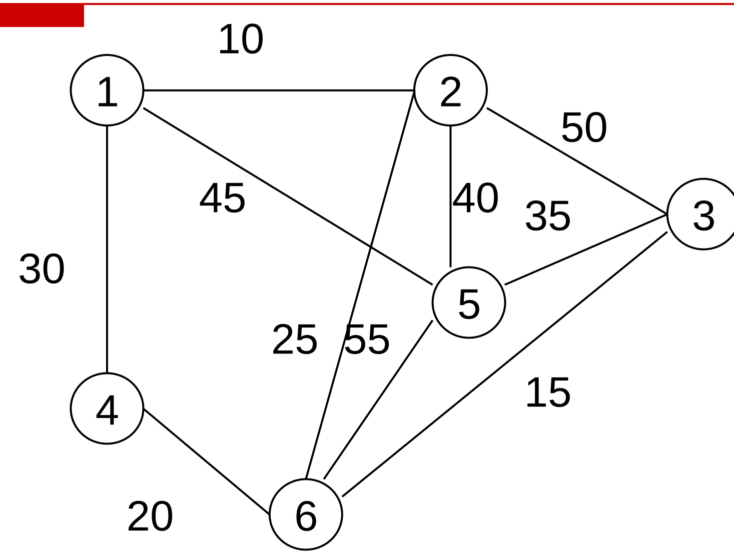
Code

```
void AddEdge(list<SEdge>& e, int from, const vector<vector<int> >& graph, const vector<bool>& s)
{
    for(int j = 0; j < (int)graph.size(); j++)
    {
        if((graph[from][j] != INFINITY) && !s[j])   //边(from,j)存在，且节点j尚未加入集合s中
            e.push_back(SEdge(from, j, graph[from][j]));
    }
}
```

```
bool Prim(const vector<vector<int> >& graph, vector<SEdge>& result)
{
    int N = (int)graph.size();
    vector<bool> s(N, false);   //当前加入的节点
    list<SEdge> e;              //s与~s的边
    s[0] = true;                //任选节点(如节点0)加入s中
    AddEdge(e, 0, graph, s);    //添加以节点0为起点的边
    SEdge m;
    for(int t = 1; t < N; t++)
    {
        if(e.empty())           //无法找到边，则表示图不连通
            return false;
        m = MinEdge(e);         //查找e中权值最小的边
        result[t-1] = m;        //添加到mst中
        s[m.j] = true;          //将顶点n加入到集合中
        DeleteEdge(e, m.j);     //删除以m.j为终点的边
        AddEdge(e, m.j, graph, s);  //添加以m.j为起点的边
    }
    return true;
}
```

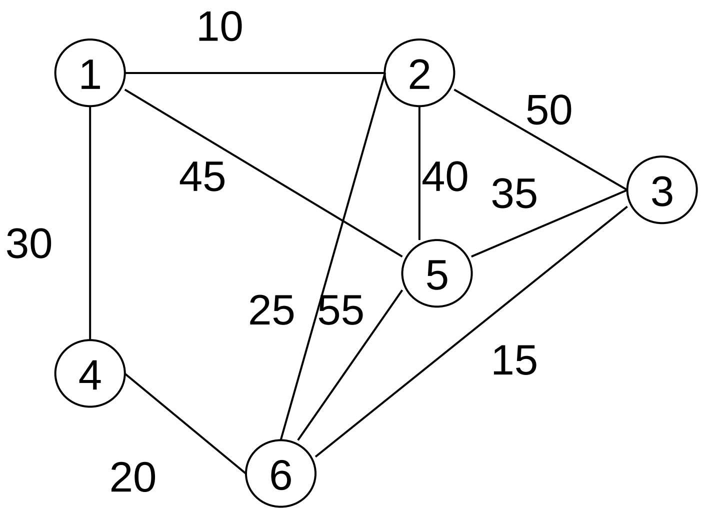
Aux Code

```
void DeleteEdge(list<SEdge>& e, int to)
{
    list<SEdge>::iterator i = e.begin();
    while(i != e.end())
    {
        if(i->j == to)
            i = e.erase(i);
        else
            i++;
    }
}

SEdge MinEdge(const list<SEdge>& e)
{
    int minEdge = e.front().value;
    list<SEdge>::const_iterator j = e.begin();
    for(list<SEdge>::const_iterator i = j; i != e.end(); i++)
    {
        if(i->value < minEdge)
        {
            minEdge = i->value;
            j = i;
        }
    }
    return *j;
}
```



(1, 2) : 10
(2, 6) : 25
(6, 3) : 15
(6, 4) : 20
(3, 5) : 35

Code2



$$
\begin{aligned}
&(1,\ \ 2):\ \ 10\\
&(2,\ \ 6):\ \ 25\\
&(6,\ \ 3):\ \ 15\\
&(6,\ \ 4):\ \ 20\\
&(3,\ \ 5):\ \ 35
\end{aligned}
$$

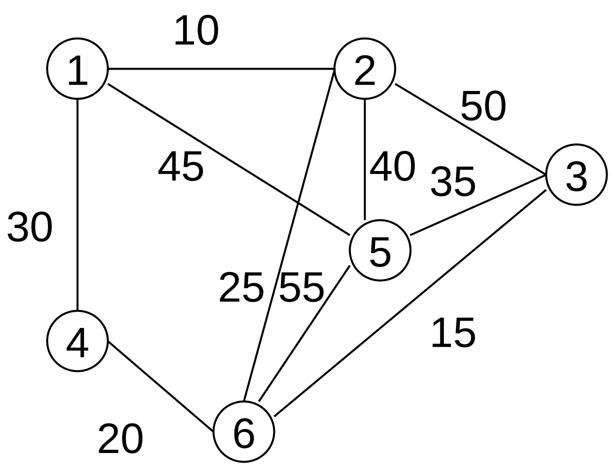
```cpp
bool Prim2(const vector<vector<int> >& graph, vector<SEdge>& mst)
{
    int N = (int)graph.size();   //节点总数
    vector<bool> s(N, false);    //已加入s中的节点，初始化为空
    vector<int> dis(N);          //dis[i]=d: s中结点到i的最短距离
    vector<int> from(N, 0);      //from[i]=j: s中距离结点i最近的结点为j
    s[0] = true;                 //s中加入任意结点(如结点0)
    dis = graph[0];              //0到其他结点的最短距离
    int t,i,k;
    int d;
    for(t = 1; t < N; t++)
    {
        //计算dis[0...N-1]的最近距离
        k = -1; //待加入节点
        for(i = 0; i < N; i++)
        {
            if(s[i])
                continue;
            if((k == -1) || (d > dis[i]))
            {
                d = dis[i];
                k = i;
            }
        }
        mst[t-1] = SEdge(from[k], k, dis[k]);   //(from,to,value)
        s[k] = true;                            //集合s加入结点k

        //使用结点k更新dis[0...N-1]
        for(i = 0; i < N; i++)
        {
            if(!s[i] && (dis[i] > graph[k][i]))
            {
                dis[i] = graph[k][i];
                from[i] = k;
            }
        }
    }
    return true;
}
```

Kruskal算法

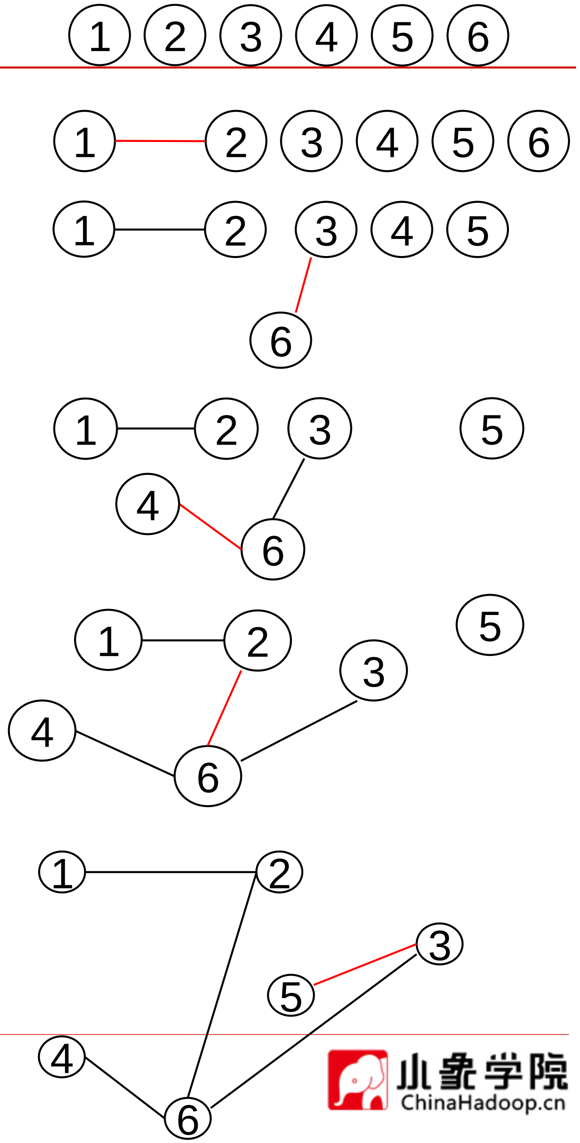
- Kruskal将权值递增的边依次加入到最小生成树中，若加入时产生回路则跳过该边，继续尝试加入下一条边。当加入N-1条边时，则得到了最小生成树。
  - Prim算法在得到最小生成树的过程中，始终保持是一颗树；而Kruskal算法最开始是森林，直到最后一条边加入，才得到树。

Kruskal算法

图G的所有边按成本非降次序排列，下一条生成树T中的边是尚未加入树的边中具有最小成本、且和T中现有的边不会构成环路的边。



边	成本
(1,2)	10
(3,6)	15
(4,6)	20
(2,6)	25
(3,5)	35

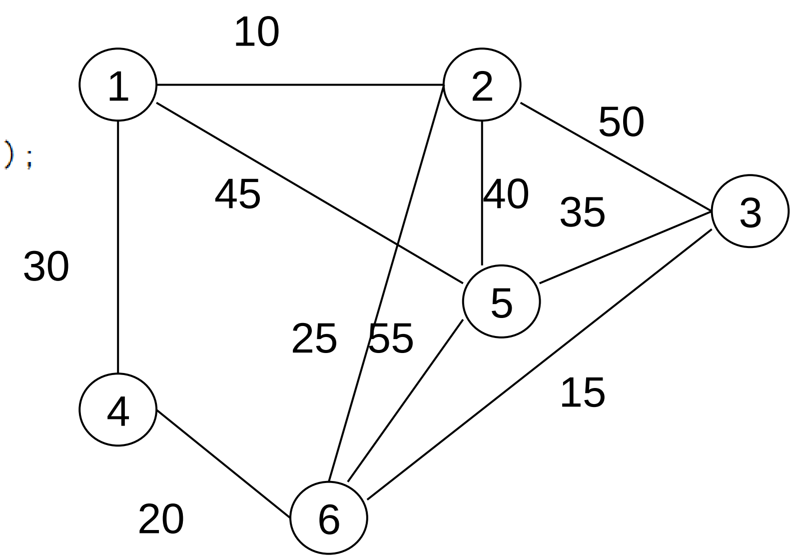


Kruskal算法几点说明

- 边集E以小顶堆的形式保存，一条当前最小成本边可以在O(loge)的时间内找到；
  - 当然，也可以用其他排序方法对边完全排序。
- 为了快速判断候选边e的加入是否会形成环，可考虑用并查集的方法：把当前状态的每个连通子图保存在各自的集合中；候选边是否可以加入，转化成边的两个顶点是否位于同一集合中；
- 算法的计算时间是O(eloge)。

Code

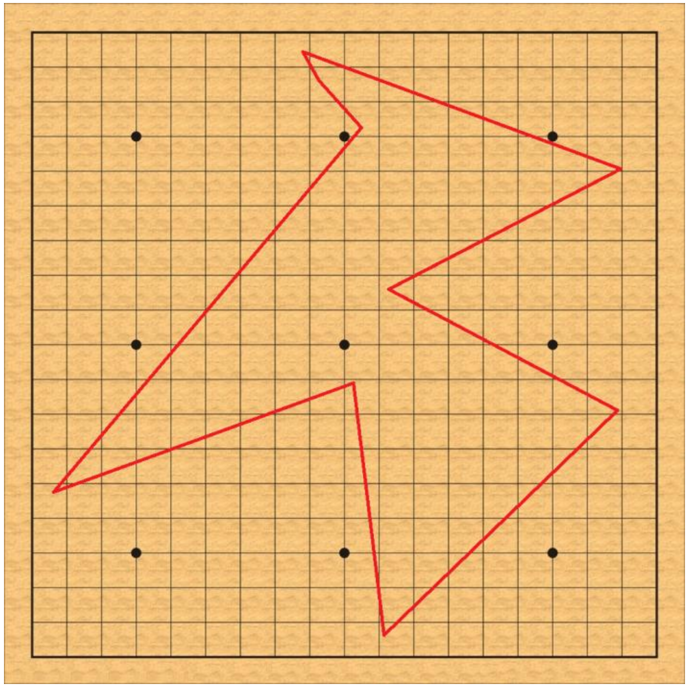
```cpp
void Kruskal(const vector<vector<int> >& graph, vector<SEdge>& mst)
{
    vector<SEdge> edge;
    int N = (int)graph.size();
    int i,j;
    for(i = 0; i < N; i++)
    {
        for(j = i+1; j < N; j++)
        {
            if(graph[i][j] != INFINITY)
                edge.push_back(SEdge(i,j,graph[i][j]));
        }
    }
    sort(edge.begin(), edge.end());
    CUnionFindSet s(N);
    int ri,rj;
    int k = 0;
    for(i = 0; i < (int)edge.size(); i++)
    {
        ri = s.Find(edge[i].i);
        rj = s.Find(edge[i].j);
        if(ri == rj)
            continue;
        s.Union(edge[i].i, edge[i].j);
        mst[k] = edge[i];
        k++;
        if(k == N-1)
            break;
    }
}
```

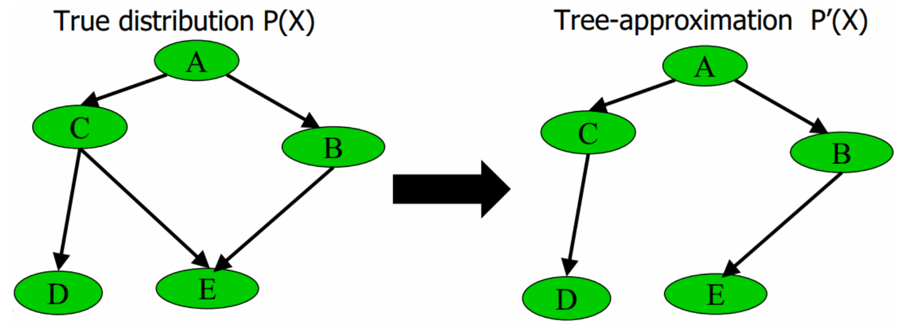


(1, 2): 10
(3, 6): 15
(4, 6): 20
(2, 6): 25
(3, 5): 35

思考题

- 在围棋棋盘上任意画一个多边形，请计算有多少个1*1的单位正方形被完全包含在该多边形内部。





总结

- 图论问题并没有想象的那么难，重视隐式图的(深度/广度优先)搜索。
- 掌握基本算法，并根据它继续扩展。
  - 最短路径、MST、BFS、DFS
- 思考：
  - 广度优先搜索和最短路径的关系。
  - Chow-Liu算法和Kruskal算法。

$$D(p\|q)=\sum_x p(x)\log\frac{p(x)}{q(x)}=E_{p(x)}\log\frac{p(x)}{q(x)}$$

我们在这里

- http://wenda.ChinaHadoop.cn
  - 视频/课程/社区
- 微博
  - @ChinaHadoop
  - @邹博_机器学习
- 微信公众号
  - 小象学院
  - 大数据分析挖掘

感谢大家！

恳请大家批评指正！