Data Structures

Graphs

Teacher : Wang Wei

1. Ellis Horowitz,etc., Fundamentals of Data Structures in C++
2. 金远平, 数据结构
3. 殷人昆, 数据结构
4. http://inside.mines.edu/~dmehta/

Graphs

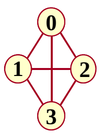
- Definition
  - Consists of two sets **V** and **E**

    Graph＝(*V*, *E*)

  - **vertices** $V = \{ u \mid u \in \text{DataSet} \}$ **, a finite,** $V(G) \neq \varnothing$
  - **edges** $E = \{ (u, v) \text{ or } <u,v> \mid u, v \in V \}$

Undirected and Directed graphs

- **Undirected graph : graph**
  - **no oriented edge**
  - **any edge is unordered**
  - **(u, v) = (v, u) , the same edge**

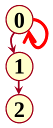


- **Directed graph : digraph**
  - **every edge has an orientation**
  - **any edge is ordered**
  - **<u, v>, u : tail, v : head**
  - **<u,v> ≠ <v,u> , two different edges**

Restrictions of Graph

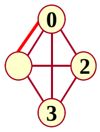
(1) may not have an edge from a vertex back to itself

– self edges
– (v, v) or <v, v> is not legal



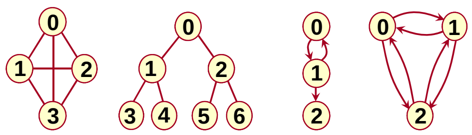
(2) may not have multiple occurrences of the same edge

– if allowed, get a multigraph



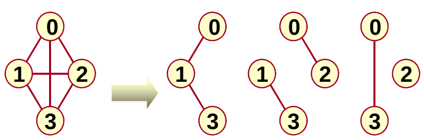
Complete Graphs with *n vertex*

- A graph
  - each edge : (u,v), u != v
  - the maximum number of edges is = $n(n-1)/2$
- A digraph
  - each edge : <u,v>, u != v
  - the maximum number of edges = $n(n-1)$



Subgraph

- G1 is a subgraph of G
  - G＝(V, E) and G 1＝(V 1, E 1)
  - $V1 \subseteq V$ and $E1 \subseteq E$



Adjacent

- if (u, v) $\in$E

  u and v are adjacent

  edge (u, v) is incident on vertices u and v

- if <u, v> $\in$E

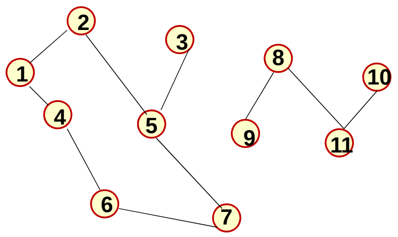
  vertex u is adjacent to v, and v is adjacent from u

  edge<u, v> is incident to u and v

Vertex Degree

Number of edges incident to vertex

degree(2) = 2, degree(5) = 3, degree(3) = 1

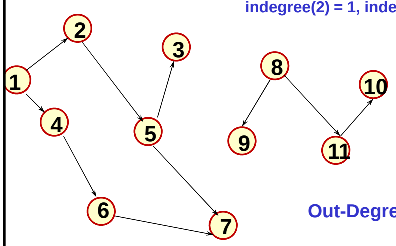


Sum of degrees = 2e (e is number of edges)

In-Degree Of A Vertex

in-degree is number of incoming edges

indegree(2) = 1, indegree(8) = 0



Out-Degree Of A Vertex

out-degree is number of outbound edges

outdegree(2) = 1, outdegree(8) = 2

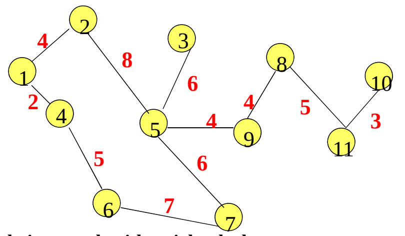
Sum Of In- And Out-Degrees

– with **n** vertices and ***e*** edges

Sum Of In-Degrees = Sum Of Out-Degrees = e

– each edge contributes 1
- *to the in-degree of some vertex*
- *to the out-degree of some other vertex*

Weighted Graphs : Network



- **Network is a graph with weighted edges**
  - Driving Distance/Time Map
    - vertex = city
    - edge weight = driving distance/time

Graph Representations

Three most commonly:

- **(1) Adjacency matrices**
- **(2) Adjacency lists**
- **(3) Adjacency multilists**

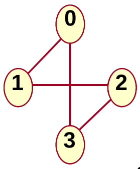
- The actual choice depends on application

Adjacency Matrix

- **0/1 n x n matrix A = (V, E)**
  - **n = numbers of vertices**

- Such as

$$\textbf{A.edge[i][j]} = \begin{cases} 1, & \text{iff } \langle i,j\rangle \in E \text{ or } (i,j) \in E \\ 0, & \text{otherwise} \end{cases}$$



$$\textbf{A.edge} = \begin{bmatrix} 0 & 1 & 0 & 1 \\ 1 & 0 & 1 & 0 \\ 0 & 1 & 0 & 1 \\ 1 & 0 & 1 & 0 \end{bmatrix}$$

$$d_i = \sum_{j=0}^{n-1} a[i][j]$$

- **an graph is symmetric**
- **a digraph may not be symmetric**

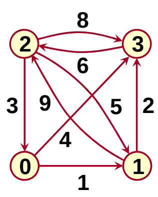
$$\textbf{A.edge} = \begin{bmatrix} 0 & 1 & 0 \\ 1 & 0 & 1 \\ 0 & 0 & 0 \end{bmatrix}$$

$$\text{out-}d_i = \sum_{j=0}^{n-1} a[i][j]$$

$$\text{in-}d_j = \sum_{i=0}^{n-1} a[i][j]$$

Adjacency Matrix of weighted diGraph

$$\textbf{A.edge}[i][j] = \begin{cases} \mathbf{W}(i,j), & i \neq j \text{ and } \langle i,j\rangle \in \mathbf{E} \text{ or } (i,j) \in \mathbf{E} \\ \infty, & i \neq j \text{ and } \langle i,j\rangle \notin \mathbf{E} \text{ or } (i,j) \notin \mathbf{E} \\ 0, & i == j \end{cases}$$



W(i,j) **is weight of edge *(i,j)***

$$\textbf{A.edge} = \begin{bmatrix} 0 & 1 & \infty & 4 \\ \infty & 0 & 9 & 2 \\ 3 & 5 & 0 & 8 \\ \infty & \infty & 6 & 0 \end{bmatrix}$$

Class definition using Adjacency Matrix

```
template <class T, class E>
class Graphmtx : public Graph<T, E>
{
  friend istream& operator >> ( istream& in,   Graphmtx<T, E>& G);
               //输入
  friend ostream& operator << (ostream& out, Graphmtx<T, E>& G);
               //输出
```

王伟, 计算机工程系, 东南大学 16

```
private:
   T *VerticesList;                    //顶点表
   E **Edge;                           //邻接矩阵

   int getVertexPos (T vertex)
   {
    //给出顶点vertex在图中的位置
      for (int i = 0; i < numVertices; i++)
         if (VerticesList[i] == Vertex) return i;
      return -1;
   }
```

王伟, 计算机工程系, 东南大学 17

```
public:
   Graphmtx (int sz = DefaultVertices);   //构造函数
   ～Graphmtx ()                          //析构函数
      { delete [ ]VerticesList; delete [ ]Edge; }

   T getValue (int i) {
    //取顶点 i 的值, i 不合理返回0
      return  i >= 0 && i <= numVertices ? VerticesList[i] : NULL;
   }

   E getWeight (int v1, int v2) {
    //取边(v1,v2)上权值
      return  v1 != -1 && v2 != -1 ? Edge[v1][v2] : 0;
   }
```

王伟, 计算机工程系, 东南大学 18

```
    int getFirstNeighbor (int v);
     //取顶点 v 的第一个邻接顶点
    int getNextNeighbor (int v, int w);
     //取 v 的邻接顶点 w 的下一邻接顶点
    bool insertVertex (const T vertex);
     //插入顶点vertex
    bool insertEdge (int v1, int v2, E cost);
     //插入边(v1, v2),权值为cost
    bool removeVertex (int v);
     //删去顶点 v 和所有与它相关联的边
    bool removeEdge (int v1, int v2);
     //在图中删去边(v1,v2)
};
```



```
template <class T, class E>
   Graphmtx<T, E>::Graphmtx (int sz) {      //构造函数
   maxVertices = sz;
   numVertices = 0;  numEdges = 0;
   int i, j;

   VerticesList = new T[maxVertices];  //创建顶点表

   Edge = (int **) new int *[maxVertices];

   for (i = 0; i < maxVertices; i++)
      Edge[i] = new int[maxVertices];   //邻接矩阵

   for (i = 0; i < maxVertices; i++)       //矩阵初始化
      for (j = 0; j < maxVertices; j++)
         Edge[i][j] =  (i == j)  ?  0 : maxWeight;
}
```

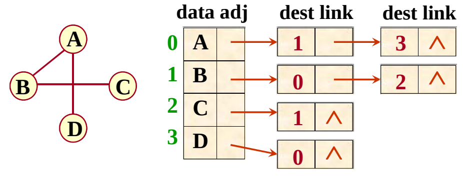
```
template <class T, class E>
int Graphmtx<T, E>::getFirstNeighbor (int v) {
//给出顶点位置为v的第一个邻接顶点的位置,
//如果找不到,则函数返回-1
   if (v != -1)
   {
      for (int col = 0; col < numVertices; col++)
         if (Edge[v][col] && Edge[v][col] < maxWeight)
            return col;
   }
   return -1;
}
```

```
template <class T, class E>
int Graphmtx<T, E>::getNextNeighbor (int v, int w) {
//给出顶点 v 的某邻接顶点 w 的下一个邻接顶点
   if (v != -1 && w != -1) {
      for (int col = w+1; col < numVertices; col++)
         if (Edge[v][col] && Edge[v][col] < maxWeight)
            return col;
   }
   return -1;
}
```


Adjacency List

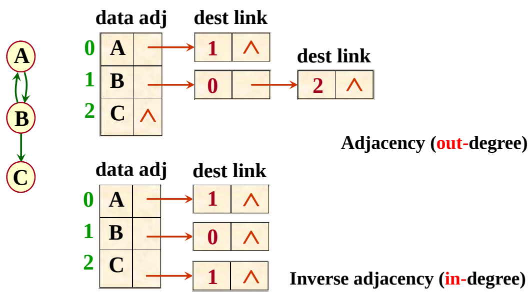
- **if explicitly represent only edges**
  - **when e << n^2**
- **n rows of Adjacency Matrix are represented as n chains**
  - an array of **n adjacency lists**
- **Each adjacency list of each vertex is a chain**
  - **chain i is a linear list** of vertices adjacent **from vertex i**

Adjacency Lists of Graph

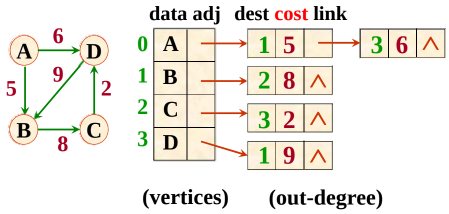


- **node** structure of **vertex** : **data** and **adj**
- **node** structure of **chain** : **dest** and **link**
- **Degree of vertex i = number of nodes in chain i**
- **edge (v_i, v_j) : vertex i** and **vertex j**

Adjacency Lists of DiGraph



Adjacency Lists of network



cost = weight of edge *(i,j)*

Class definition using Adjacency lists

```
template <class T, class E>
struct Edge {                          //边结点的定义
    int dest;                          //边的另一顶点位置
    E cost;                            //边上的权值
    Edge<T, E> *link;                  //下一条边链指针

    Edge () {}                         //构造函数
    Edge (int num, E cost)             //构造函数
          : dest (num), weight (cost), link (NULL) { }

    bool operator != (Edge<T, E>& R) const
          { return dest != R.dest; }        //判边等否
};
```

```
template <class T, class E>
struct Vertex {                    //顶点的定义
    T data;                        //顶点的名字
    Edge<T, E> *adj;               //边链表的头指针
};

template <class T, class E>
class Graphlnk : public Graph<T, E>
{   //图的类定义
friend istream& operator >> (istream& in,    Graphlnk<T, E>& G);
                 //输入
friend ostream& operator << (ostream& out, Graphlnk<T, E>& G);
                 //输出
```

王伟, 计算机工程系, 东南大学 28

```
private:
   Vertex<T, E> *NodeTable;
              //顶点表 (各边链表的头结点)

   int getVertexPos (const T vertx)
  {
              //给出顶点vertex在图中的位置
     for (int i = 0; i < numVertices; i++)
        if (NodeTable[i].data == vertx) return i;
     return -1;
  }
```

王伟, 计算机工程系, 东南大学 29

```
public:
   Graphlnk (int sz = DefaultVertices);  //构造函数
   ~Graphlnk();                          //析构函数

  T getValue (int i) {                 //取顶点 i 的值
     return (i >= 0 && i < NumVertices) ? NodeTable[i].data : 0;
  }
   E getWeight (int v1, int v2);      //取边(v1,v2)权值

   bool insertVertex (const T& vertex);
   bool removeVertex (int v);
   bool insertEdge (int v1, int v2, E cost);
   bool removeEdge (int v1, int v2);
   int getFirstNeighbor (int v);
   int getNextNeighbor (int v, int w);
};
```

王伟, 计算机工程系, 东南大学 30

```
template <class T, class E>
Graphlnk<T, E>::Graphlnk (int sz)
{
//构造函数：建立一个空的邻接表
   maxVertices = sz;
   numVertices = 0;  numEdges = 0;
   NodeTable = new Vertex<T, E>[maxVertices];
        //创建顶点表数组

   if (NodeTable == NULL)
      { cerr << "存储分配错！ " << endl;  exit(1); }

   for (int i = 0; i < maxVertices; i++)
      NodeTable[i].adj = NULL;
}
```



```
template <class T, class E>
Graphlnk<T, E>::~Graphlnk()
{
//析构函数：删除一个邻接表
   for (int i = 0; i < numVertices; i++ )
   {
      Edge<T, E> *p = NodeTable[i].adj;

      while (p != NULL)
      {
         NodeTable[i].adj = p->link;
         delete p;  p = NodeTable[i].adj;
      }
   }
   delete [ ]NodeTable;              //删除顶点表数组
};
```



```
template <class T, class E>
int Graphlnk<T, E>::getFirstNeighbor (int v)
{
//给出顶点位置为 v 的第一个邻接顶点的位置,
//如果找不到, 则函数返回 -1
   if (v != -1)
   {                       //顶点v存在
   Edge<T, E> *p = NodeTable[v].adj;
        //对应边链表第一个边结点
   if (p != NULL) return p->dest;
        //存在, 返回第一个邻接顶点
   }
   return -1;         //第一个邻接顶点不存在
}
```

```
template <class T, class E>
int Graphlnk<T, E>::getNextNeighbor (int v, int w)
{
//给出顶点v的邻接顶点w的下一个邻接顶点的位置,
//若没有下一个邻接顶点, 则函数返回-1
   if (v != -1)
   {                              //顶点v存在
      Edge<T, E> *p = NodeTable[v].adj;

      while (p != NULL && p->dest != w)
         p = p->link;

      if (p != NULL && p->link != NULL)
         return p->link->dest;          //返回下一个邻接顶点
   }
   return -1;                     //下一邻接顶点不存在
}
```


Adjacency Multilists for undirected graph

- **node** structure of **edge**

mark	vertex1	vertex2	path1	path2

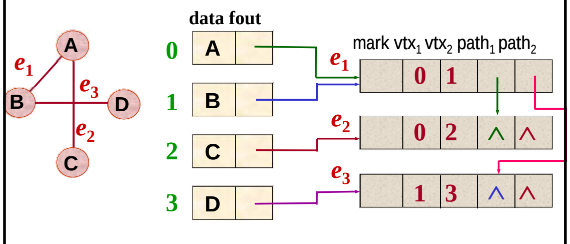
- **mark : to indicate whether or not the edge has been examined**
- **vertex1, vertex2 : two vertices of the edge**
- **path1 : to point the adjacency edge of vertex1**
- **path2 : to point the adjacency vertex of vertex2**
- ***cost* : when G is a *network***
- **node** structure of **vertex**
  - **data**

  and

  - **firstout : a pointer to point the adjacency edge of the vertex**

data	firstout

Example : undirected graph



Adjacency Multilists for directed graph

- node structure of edge

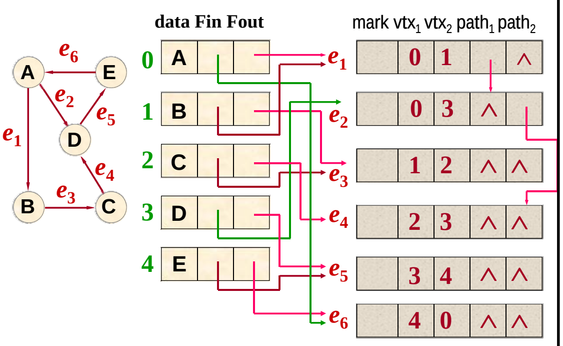
mark	vertex1	vertex2	path1	path2

- node structure of vertex
  - data and

data	firstin	firstout

  - firstout : to point the adjacency edge (out-degree)
  - firstin : to point the adjacency edge (in-degree)

Example : digraph



Path

- a path
  - from *u* to *v*
  - a sequence of vertices $u, i_1, i_2, \ldots, i_k, v$
  - *G* is undirected

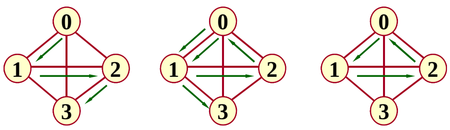
    $(u, i_1), (i_1, i_2), \ldots, (i_k, v)$ are edges in *E*
  - *G'* is directed

    $<u, i_1>, <i_1, i_2>, \ldots, <i_k, v>$ are edges in *E'*
- path length
  - the number of edges on the path

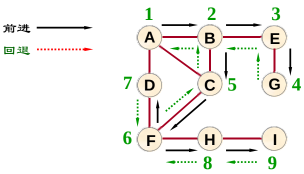
  or
  - the sum of the weights of the edges on the path

- Since a graph may have more than one path between two vertices
- May be interested in finding a path with a particular property
- For example
  - **find a path with minimum length**
  - **find a path with maximum length**

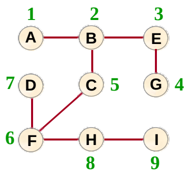
- **simple path**
  - all vertices except possibly the first and last are distinct
- **cycle**
  - the first and last vertices are the same
- for **directed** graph, **paths** and **cycles** are **directed**



DFS (Depth First Search)



Depth-first search



Depth-first Spanning Tree

dfn **: depth-first number**
- **outside the vertices**
- **give the sequence in which the vertices are visited during the depth-first search**

DFS

- Begin by visiting the start vertex *v*
- Next an unvisited vertex w_1 adjacent to *v* is selected
- From w_1 to visit an unvisited vertex w_1 adjacent to w_2
- From w_2 to w_3, and so on
- When a vertex *u* is reached
  - all its adjacent vertices have been visited
- Back up to the last vertex visited
  - that has an unvisited vertex *w*
- Search terminates
  - When no unvisited vertex can reached from any of the visited vertices

DFS Algorithm

```
template<class T, class E>
void DFS (Graph<T, E>& G, const T& v)
{
//从顶点v出发对图G进行深度优先遍历的主过程
    int i, loc, n = G.NumberOfVertices();        //顶点个数

    bool *visited = new bool[n];               //创建辅助数组
    for (i = 0; i < n; i++) visited [i] = false;
                                      //辅助数组visited初始化

    loc = G.getVertexPos(v);
    DFS (G, loc, visited);          //从顶点0开始深度优先搜索
    delete [] visited;                            //释放visited
}
```

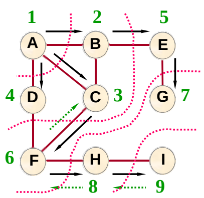


```
template<class T, class E>
void DFS (Graph<T, E>& G, int v, bool visited[])
{
   cout << G.getValue(v) << ' ';      //访问顶点v
   visited[v] = true;                 //作访问标记
   int w = G.getFirstNeighbor (v);    //第一个邻接顶点

   while (w != -1)
   {   //若邻接顶点w存在
      if ( !visited[w] ) DFS(G, w, visited);
                         //若w未访问过, 递归访问顶点w
     w = G.getNextNeighbor (v, w); //下一个邻接顶点
   }
}
```

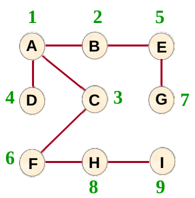
Analysis of DFS

- **Adjacency lists**
  - **T(n) is O(e)**
- **Adjacency matrix**
  - **determine all vertices adjacent to v, T(n) is O(n)**
  - **Total time: T(n) is O(n^2)**

BFS (Breadth First Search)



Breadth-first search



Breadth-first Spanning Tree

BFS

- Begin by visiting the start vertex *v*
- Next all unvisited vertices w_1, w_2, …, w_t adjacent to *v* are selected
- Unvisited vertices adjacent to these newly visited vertices are then visited
- And so on

BFS Algorithm

```
template <class T, class E>
void BFS (Graph<T, E>& G, const T& v)
{
    int i, w, n = G.NumberOfVertices();      //图中顶点个数
    bool *visited = new bool[n];
    for (i = 0; i < n; i++) visited[i] = false;

    int loc = G.getVertexPos (v);            //取顶点号
    cout << G.getValue (loc) << ' ';         //访问顶点v
    visited[loc] = true;                     //做已访问标记
    Queue<int> Q;  Q.EnQueue (loc);
                                  //顶点进队列, 实现分层访问
```



```
while (!Q.IsEmpty() ) {                      //循环, 访问所有结点
    Q.DeQueue (loc);
    w = G.getFirstNeighbor (loc);            //第一个邻接顶点
    while (w != -1) {                        //若邻接顶点w存在
        if (!visited[w]) {                   //若未访问过
            cout << G.getValue (w) << ' ';   //访问
            visited[w] = true;
            Q.EnQueue (w);                   //顶点w进队列
        }
        w = G.getNextNeighbor (loc, w);
                              //找顶点loc的下一个邻接顶点
    }
  }   //外层循环, 判队列空否
  delete [] visited;
}
```


Analysis of BFS

- Using a queue
  - each visited vertex enters it exactly once
- Adjacency lists
  - T(n) is O(e)
- Adjacency matrix
  - Loop time: T(n) is O(n)
  - Total time: T(n) is O(n^2)

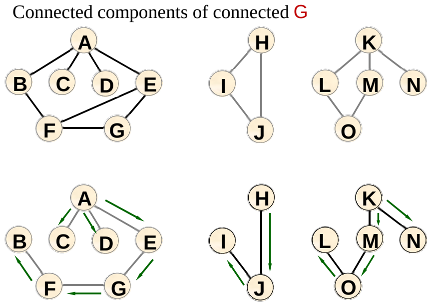
Connectedness

- u and v are connected
  - iff : a path in G from u to v (also from v to u)
- an undirected G is connected
  - iff : for every pair of distinct u and v in V, there is a path from u to v

So
- a path between every pair of vertices

- A undirected G is connected
  - can not add vertices and edges from original graph and retain connectedness
- A connected graph has exactly 1 component
  - a maximal subgraph
- A directed G' is strongly connected
  - every pair of distinct u and v
  - a directed path from u to v and also from v to u
- A strongly connected component
  - a maximal subgraph

Connected components of connected G



Connected components of unconnected G

Determining Connected Components

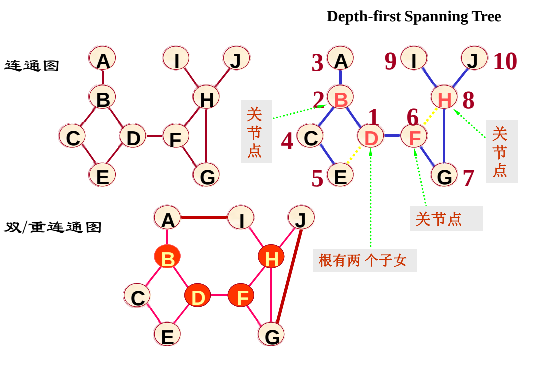
```
template <class T, class E>
void Components (Graph<T, E>& G)
{              //通过DFS，找出无向图的所有连通分量
   int i, n = G.NumberOfVertices();       //图中顶点个数
   bool *visited = new bool[n];           //访问标记数组
   for (i = 0; i < n; i++) visited[i] = false;
   for (i = 0; i < n; i++)                //扫描所有顶点
      if (!visited[i]) {                  //若没有访问过
         DFS (G, i, visited);             //访问
         OutputNewComponent();            //输出连通分量
      }
   delete [ ] visited;
}
```


Analysis of Components Algorithm

- **Adjacency lists**
  - ***for* loops time: T(n) is O(n)**
  - **DFS total time: T(n) is O(n+e)**
- **Adjacency matrix**
  - **Total time: T(n) is $O(n^2)$**

Biconnected Component

- **A vertex *v* is an articulation point(关节点)**
  - **in undirected G**
  - **iff *v* be deleted , together with the deletion of all edges incident to *v***
    - **the graph has at least two connected components**
- **Biconnected graph (双/重连通图)**
  - is a connected graph that has no articulation points
  - *任何一对顶点之间至少存在有两条路径*, 在删去某个顶点及与该顶点相关联的边时, 不破坏图的连通性
- **Biconnected component (双/重连通分量)**
  - **is a maximal biconnected subgraph**
  - **G contains no other subgraph**
  - **No edge can be in two or more biconnected components**



- No edge can be in two or more biconnected components
- Undirected graph G, the biconnected components can be found by using any depth-first spanning tree
- *root* of the depth-first spanning tree is an articulation point
  - iff it has at least two children
- *other certex u* is an articulation point
  - iff it has at least one children, such as *w*
    - it is not possible to search an ancestor of u using a path composed solely of w , descendants of w, and a single back edge
- Back edge
- Cross edge

Data Structures

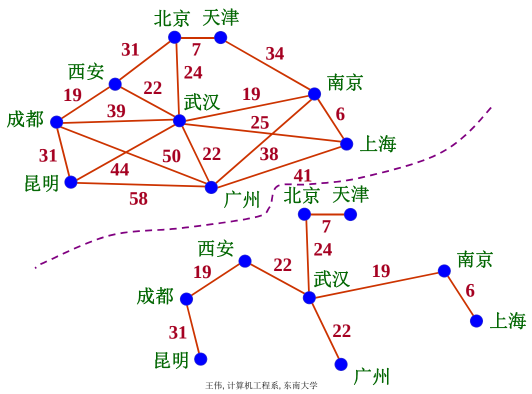
Spanning Trees

Teacher : Wang Wei

1. Ellis Horowitz,etc., Fundamentals of Data Structures in C++
2. 金远平, 数据结构
3. 殷人昆, 数据结构
4. http://inside.mines.edu/~dmehta/

spanning tree

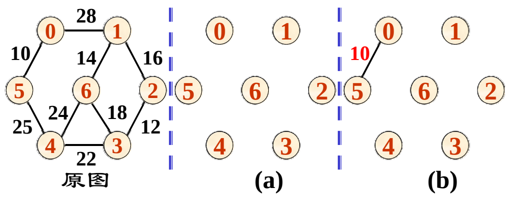
- **Minimum-Cost Spanning Tree**
  - weighted connected undirected graph
  - cost of spanning tree is sum of edge costs
  - find spanning tree that has minimum cost

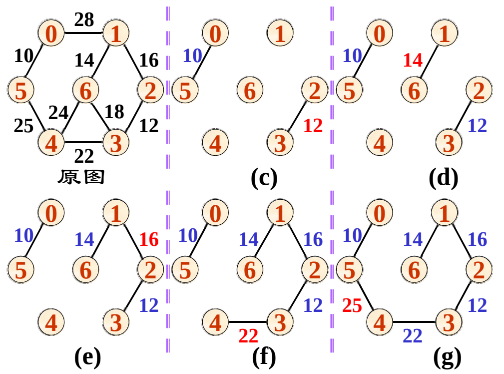


Constraints

- **To construct minimum-cost spanning tree**
  - **must use only edges within the graph**
  - **must use exactly *n-1* edges and *n* vertices**
  - **may not use edges that produce a cycle**
  - **the cost is least**

Construct Stages for **Kruskal's Method**





Pseudocode for Kruskal

```
T = ∅;        //T是最小生成树的边集合
              //E是带权无向图的边集合

while ( T contains less than n-1 edges && E not empty)
{
    choose an edge (v, w) form E of lowest cost;
    delete (v, w) from E;
    if( (v, w) does not create a cycle in T ) add (v, w) to T;
     else discard (v, w);
}
if ( T contains fewer than n-1 edges)
   cout << “no spanning tree" << endl;
```

- using **Min-Heap** to store edges

vertrx1	vertex2	weight
u	v	cost

- using **UFS** to determine if **v** and **w** is or not already connected by the earlier selection of edges


```
const float maxValue = FLOAT_MAX
//机器可表示的、问题中不可能出现的大数
//树边结点的类定义
template <class T, class E>
struct MSTEdgeNode
{
    int tail, head;                                  //两顶点位置
    E cost;                                          //边上的权值
    MSTEdgeNode() : tail(-1), head(-1), cost(0) { }
                                                     //构造函数
};
```



```
//MST类定义
template <class T, class E>
class MinSpanTree
 {
protected:
   MSTEdgeNode<T, E> *edgevalue;                 //边值数组
   int maxSize, n;                               //最大元素个数和当前个数

public:
   MinSpanTree (int sz = DefaultSize-1) : MaxSize (sz), n (0)
   {
      edgevalue = new MSTEdgeNode<T, E>[sz];
   }
   int Insert (MSTEdgeNode& item);
};
```

Implementation of Kruskal

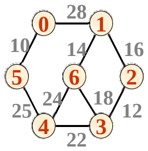
```
#include "heap.h"
#include "UFSets.h"
template <class T, class E>
void Kruskal (Graph<T, E>& G,
              MinSpanTree<T, E>& MST)
{
   MSTEdgeNode<T, E> ed;                   //边结点辅助单元
   int u, v, count;
   int n = G.NumberOfVertices();           //顶点数
   int m = G.NumberOfEdges();              //边数
   MinHeap <MSTEdgeNode<T, E>> H(m);       //最小堆
   UFSets F(n);                            //并查集
```



```
for (u = 0; u < n; u++)
    for (v = u+1; v < n; v++)
       if (G.getWeight(u,v) != maxValue)
       {                                   //插入并构造堆
          ed.tail = u;  ed.head = v;
          ed.cost = G.getWeight (u, v);
          H.Insert(ed);
       }
```



```
   count = 1;                  //最小生成树边数计数
   //反复执行, 取n-1条边
   while (count < n)
   {  H.Remove(ed);             //退出具最小权值的边
      u = F.Find(ed.tail);  v = F.Find(ed.head);
                          //取两顶点所在集合的根u与v
      if (u != v)
      {                         //不是同一集合, 不连通
         F.Union(u, v);         //合并, 连通它们
         MST.Insert(ed);        //该边存入MST
         count++;
      }
   }
}
```



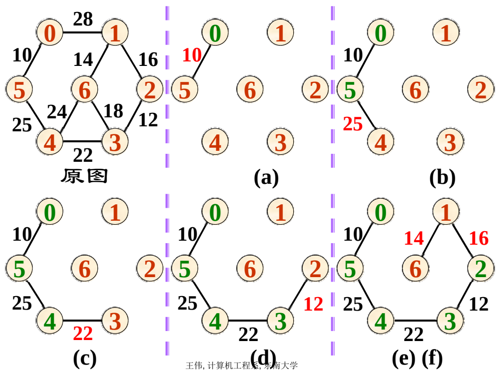
UFS

order:

	(0,5,10)	选中
	(2,3,12)	选中
	(1,6,14)	选中
	(1,2,16)	选中
	(3,6,18)	舍弃
1并4	(3,4,22)	选中
	(4,6,24)	舍弃
1并5	(4,5,25)	选中

	0	1	2	3	4	5	6	
F	-1	-1	-1	-1	-1	-1	-1	
	-2	-1	-1	-1	-1	0	-1	(0,5,10)
	-2	-1	-2	2	-1	0	-1	(2,3,12)
	-2	-2	-2	2	-1	0	1	(1,6,14)
	-2	-4	1	2	-1	0	1	(1,2,16)
	-2	-5	1	2	1	0	1	(3,4,22)
	1	-7	1	2	1	0	1	(4,5,25)

Construct Stages **for Prim's Method**



Pseudocode for Prim

// Start with any single vertex

$V_{mst} = \{u_0\}$, $E_{mst} = \varnothing$;

while (V_{mst} **contains less than** n **vertices &&** E ***not empty*)**

{

choose an edge (v, w) **form** E **of lowest cost,** $u \in V_{mst} \cap v \in V - V_{mst}$;

let $V_{mst} = V_{mst} \cup \{v\}$, $E_{mst} = E_{mst} \cup \{(u, v)\}$;

discard (v, w), $E = E - \{(u, v)\}$;

}

if (V_{mst} **contains fewer than** n **vertices)**

cout << " no spanning tree " << endl;

Implementation of prim

```
#include "heap.h"
template <class T, class E>
void Prim (Graph<T, E>& G, const T u0,
      MinSpanTree<T, E>& MST)
{
   MSTEdgeNode<T, E> ed;                    //边结点辅助单元
   int i, u, v, count;
   int n = G.NumberOfVertices();            //顶点数
   int m = G.NumberOfEdges();               //边数
   int u = G.getVertexPos(u0);              //起始顶点号
   MinHeap <MSTEdgeNode<T, E>> H(m);        //最小堆
   bool Vmst = new bool[n];                 //最小生成树顶点集合
```

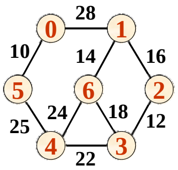


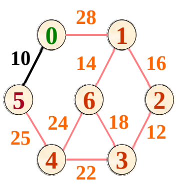
```
   MinHeap <MSTEdgeNode<T, E>> H(m);        //最小堆

   bool Vmst = new bool[n];                 //最小生成树顶点集合
   for (i = 0; i < n; i++)
     Vmst[i] = false;

   Vmst[u] = true;                          //u 加入生成树
```



```
   count = 1;
   do {  //迭代
        v = G.getFirstNeighbor(u);

       while (v != -1)
       {                                    //检测u所有邻接顶点
          if (!Vmst[v])
          {                                 //v不在mst中
             ed.tail = u;  ed.head = v;
             ed.cost = G.getWeight(u, v);
             H.Insert(ed);                  //(u,v)加入堆
          } //堆中存所有u在mst中, v不在mst中的边
         v = G.getNextNeighbor(u, v);
       }
```

```
        while (!H.IsEmpty() && count < n)
        {
            H.Remove(ed);              //选堆中具最小权的边
            if (!Vmst[ed.head])
            {
                MST.Insert(ed);             //加入最小生成树
                u = ed.head;  Vmst[u] = true;
                                       //u加入生成树顶点集合
                count++;
                break;
            }
        }

    } while (count < n);

}  // end of prim
```



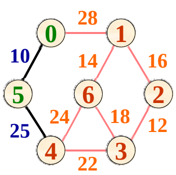
H = {(0,5,10), (0,1,28)}

ed = (0, 5, 10)

V_{mst} = {t, f, f, f, f, f, f}

⇩

V_{mst} = {t, f, f, f, f, t, f}



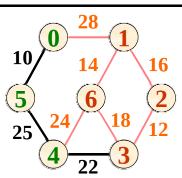
H = {(5,4,25), (0,1,28)}

ed = (5, 4, 25)

V_{mst} = {t, f, f, f, f, t, f}

⇩

V_{mst} = {t, f, f, f, t, t, f}



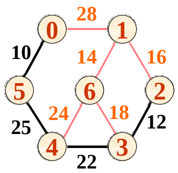
H = {(4,3,22), (4,6,24), (0,1,28)}

ed = (4, 3, 22)

V_{mst} = {t, f, f, f, t, t, f}

⇩

V_{mst} = {t, f, f, t, t, t, f}



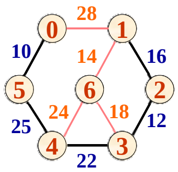
H = {(3,2,12), (3,6,18), (4,6,24), (0,1,28)}

ed = (3, 2, 12)

V_{mst} = {t, f, f, t, t, t, f}

⇩

V_{mst} = {t, f, t, t, t, t, f}



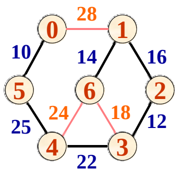
H = {(2,1,16), (3,6,18), (4,6,24), (0,1,28)}

ed = (2, 1, 16)

V_{mst} = {t, f, t, t, t, t, f}

⇩

V_{mst} = {t, t, t, t, t, t, f}



H = { (1,6,14), (3,6,18), (4,6,24), (0,1,28) }

ed = (1, 6, 14)

V_{mst} = { t, t, t, t, t, t, f }

⇩

V_{mst} = { t, t, t, t, t, t, t }

Edges in MST:

(0, 5, 10) , (5, 4, 25) , (4, 3, 22) , (3, 2, 12) , (2, 1, 16) , (1, 6, 14)