Data Structures

Graphs

Teacher : Wang Wei

1. Ellis Horowitz,etc., Fundamentals of Data Structures in C++
2. ,
3. ,
4. http://inside.mines.edu/~dmehta/

Graphs

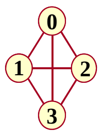
- Definition
  - Consists of two sets **V** and **E**

    Graph＝(*V*, *E*)

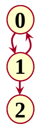
  - **vertices** ***V* = { *u* | *u* ∊ DataSet }** **, a finite, *V(G)* ≠ ∅**
  - **edges** ***E* = { (*u*, *v*) or <u,v> | *u*, *v* ∊ *V* }**

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**

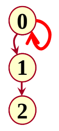


Restrictions of Graph

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

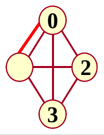
– self edges

– (v, v) or <v, v> is not legal



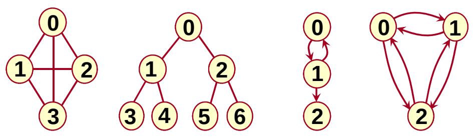
(2) may not have multiple occurrences of the same edge

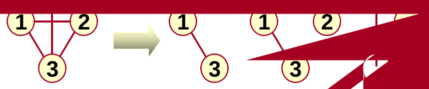
– if allowed, get a multigraph



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)





Adjacent

- if (u, v) ∈E

  u and v are adjacent

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

- if <u, v> ∈E

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

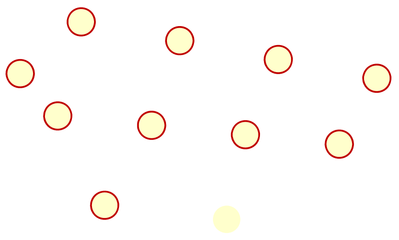
  edge<u, v> is incident to u and v

7

Vertex Degree

Number of edges incident to vertex

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



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

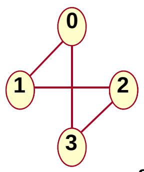
- **Network is a graph with weighted edges**
  - Driving Distance/Time Map
    -

Adjacency Matrix

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

- Such as

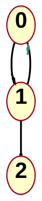
$$\text{A.edge[i][j]} = \begin{cases} 1, & \text{iff } <i,j> \;\; E \text{ or } (i,j) \;\; E \\ 0, & \text{otherwise} \end{cases}$$



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

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

- an graph is symmetric



- a digraph may not be symmetric

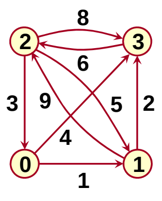
$$\text{out-d}_i = \sum_{j=0}^{n-1} a[i][j]$$

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

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

Adjacency Matrix of weighted diGraph

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



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

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

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);
                      //
  friend ostream& operator << (ostream& out, Graphmtx<T, E>& G);
                      //
```

16

```
private:
    T *VerticesList;                      //
    E **Edge;                             //

    int getVertexPos (T vertex)
   {
     //              vertex
       for (int i = 0; i < numVertices; i++)
          if (VerticesList[i] == Vertex) return i;
       return  -1;
    }
```

17

```
public:
    Graphmtx (int sz = DefaultVertices);    //
        Graphmtx ()                         //
        { delete [ ]VerticesList;  delete [ ]Edge; }

    T getValue (int i) {
      //          i        , i                  0
        return  i >= 0 && i <= numVertices ? VerticesList[i] : NULL;
    }

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

18

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

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

   VerticesList = new T[maxVertices];  //

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

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

   for (i = 0; i < maxVertices; i++)       //
      for (j = 0; j < maxVertices; j++)
         Edge[i][j] =  (i == j)    0 : maxWeight;
}
```

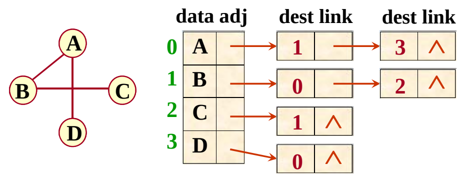
```
   template <class T, class E>
   int Graphmtx<T, E>::getFirstNeighbor (int v) {
   //                      v                                  ,
   //                 ,                 -1
      if (v != -1)
      {
         for (int col = 0; col < numVertices; col++)
            if (Edge[v][col] && Edge[v][col] < maxWeight)
               return col;
      }
      return -1;
   }
```

```
template <class T, class E>
int Graphmtx<T, E>::getNextNeighbor (int v, int w) {
//              v                    w
   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

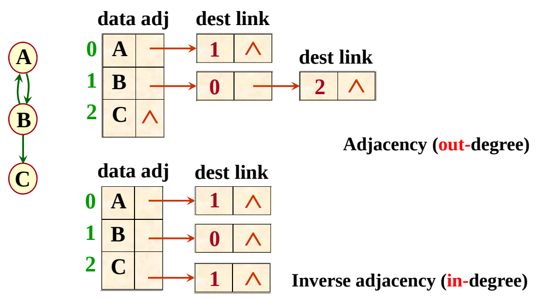
- **if explicitly represent only edges**
  - **when e << n^2**
- **n rows of Adjacency Matrix are represented as n chains**
  - an array of **n adjacency lists**
- **Each adjacency list of each vertex is a chain**
  - **chain i is a linear list** of vertices adjacent **from vertex i**

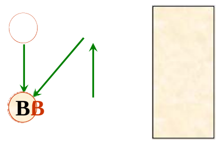
Adjacency Lists of Graph



- **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





```
   template <class T, class E>
   struct Vertex {                    //
      T data;                         //
      Edge<T, E> *adj;                //
   };


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

28

```
   private:
      Vertex<T, E> *NodeTable;
                    //        (                    )

      int getVertexPos (const T vertx)
     {
                    //          vertex
         for (int i = 0; i < numVertices; i++)
            if (NodeTable[i].data == vertx) return i;
         return  -1;
      }
```

29

```
   public:
      Graphlnk (int sz = DefaultVertices);  //
      ~Graphlnk();                          //

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

      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);
   };
```

30

```
template <class T, class E>
Graphlnk<T, E>::Graphlnk (int sz)
{
//
   maxVertices = sz;
   numVertices = 0;  numEdges = 0;
   NodeTable = new Vertex<T, E>[maxVertices];
        //

   if (NodeTable == NULL)
      { cerr << "                    " << endl;  exit(1); }

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

31

```
template <class T, class E>
Graphlnk<T, E>::   Graphlnk()
{
//
   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;               //
};
```

32

```
template <class T, class E>
int Graphlnk<T, E>::getFirstNeighbor (int v)
{
//                    v                              ,
//              ,                 -1
   if (v != -1)
   {                    //      v
   Edge<T, E> *p = NodeTable[v].adj;
       //
   if (p != NULL) return p−>dest;
       //      ,
   }
   return -1;          //
}
```

33

```
template <class T, class E>
int Graphlnk<T, E>::getNextNeighbor (int v, int w)
{
//              v                  w                                     ,
//                                   ,                    -1
   if (v != -1)
   {                                  //      v
      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;              //
   }
   return -1;                           //
}
```

34

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***

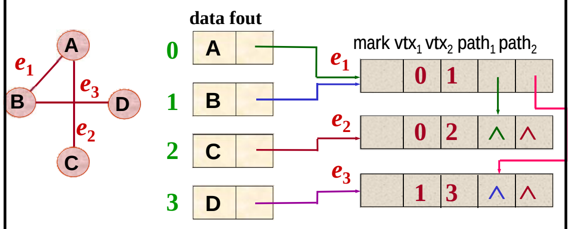
- **node** structure of **vertex**
  - **data**

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

data	firstout

35

Example : undirected graph



36

Adjacency Multilists for directed graph

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

mark	vertex1	vertex2	path1	path2

- **node** structure of **vertex**
  - **data**

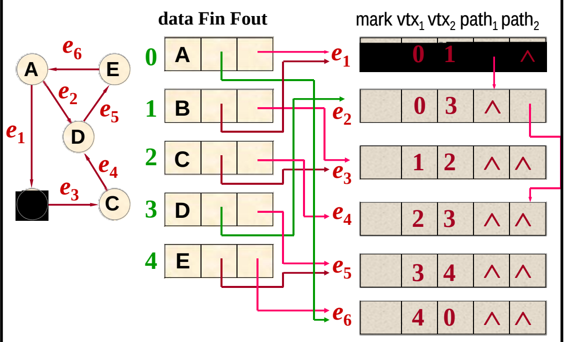
and

data	firstin	firstout

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

37

Example : digraph



38

Path

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

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

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

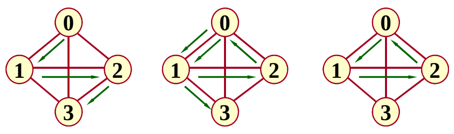
  or
  - the sum of the weights of the edges on the path

39

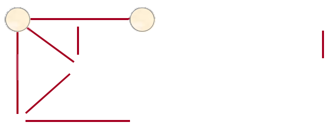
- Since a graph may have more than one path between two vertices
- May be interested in finding a path with a particular property
- For example
  - **find a path with minimum length**
  - **find a path with maximum length**

[illegible]

- **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)



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

43

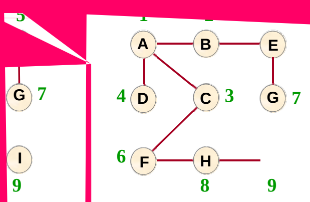
DFS Algorithm

```
template<class T, class E>
void DFS (Graph<T, E>& G, const T& v)
{
//          v               G
   int i, loc, n = G.NumberOfVertices();        //

   bool *visited = new bool[n];                //
    for (i = 0; i < n; i++) visited [i] = false;
                                       //            visited

    loc = G.getVertexPos(v);
   DFS (G, loc, visited);           //          0
    delete [] visited;                                   //       visited
}
```

44

```
template<class T, class E>
void DFS (Graph<T, E>& G, int v, bool visited[])
{
   cout << G.getValue(v) << ' ';      //              v
   visited[v] = true;                    //
   int w = G.getFirstNeighbor (v);    //

    while (w != -1)
   {     //                w
      if ( !visited[w] ) DFS(G, w, visited);
                                //   w              ,                  w
      w = G.getNextNeighbor (v, w); //
   }
}
```

45



47

BFS Algorithm

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

   int loc = G.getVertexPos (v);            //
   cout << G.getValue (loc) << ' ';         //         v
   visited[loc] = true;                     //
   Queue<int> Q;  Q.EnQueue (loc);
                                       //         ,
```

```
while (!Q.IsEmpty() ) {                      //      ,
      Q.DeQueue (loc);
      w = G.getFirstNeighbor (loc);          //
      while (w != -1) {                      //            w
         if (!visited[w]) {                  //
            cout << G.getValue (w) << ' ';      //
            visited[w] = true;
            Q.EnQueue (w);                   //      w
         }
         w = G.getNextNeighbor (loc, w);
                                       //       loc
      }
   }    //
   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**

52

- **A undirected G is connected**
  - can **not add vertices** and **edges** from original graph and retain connectedness

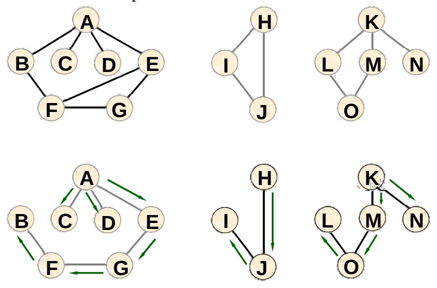
- A **connected graph has exactly 1 component**
  - **a maximal subgraph**

- 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**

53

Connected components of connected G



Connected components of unconnected G

54

Determining Connected Components

```
template <class T, class E>
void Components (Graph<T, E>& G)
{                    //      DFS
    int i, n = G.NumberOfVertices();       //
    bool *visited = new bool[n];           //
    for (i = 0; i < n; i++) visited[i] = false;
    for (i = 0; i < n; i++)                //
        if (!visited[i]) {                 //
            DFS (G, i, visited);           //
            OutputNewComponent();          //
        }
    delete [ ] visited;
}
```

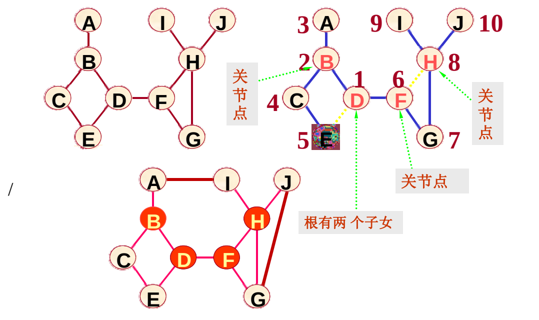
Analysis of Components Algorithm

- **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　的边时, 不破坏图的连通性
- **Biconnected component (双/重连通分　)**
  - **is a maximal biconnected subgraph**
  - **G contains no other subgraph**
  - **No edge can be in two or more biconnected components**

Depth-first Spanning Tree



- No edge can be in two or more biconnected components
- Undirected graph G, the biconnected componments 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

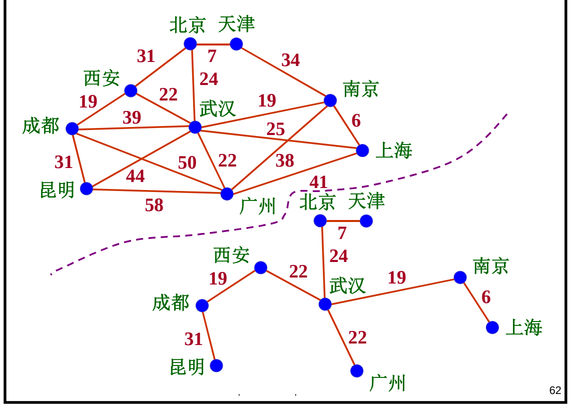
Teacher : Wang Wei

1. Ellis Horowitz,etc., Fundamentals of Data Structures in C++
2. ,
3. ,
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

61



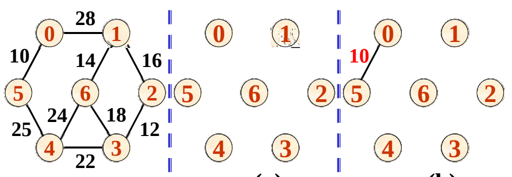
62

Constraints

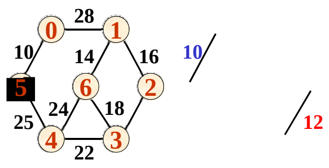
- To construct minimum-cost spanning tree
  - must use only edges within the graph
  - must use exactly *n-1* edges and *n* vertices
  - may not use edges that produce a cycle
  - the cost is least

63

Construct Stages **for Kruskal's Method**



(a) (b)



- 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**

67

```
const float maxValue = FLOAT_MAX
//
//
template <class T, class E>
struct MSTEdgeNode
{
   int tail, head;                                   //
    E cost;                                          //
    MSTEdgeNode() : tail(-1), head(-1), cost(0) { }
                                                       //
};
```

68

```
//MST
template <class T, class E>
class MinSpanTree
 {
protected:
   MSTEdgeNode<T, E> *edgevalue;                   //
   int maxSize, n;                         //

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

69

```
#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;                 //
    int u, v, count;
    int n = G.NumberOfVertices();                   //
    int m = G.NumberOfEdges();                        //
    MinHeap <MSTEdgeNode<T, E>> H(m);              //
    UFSets F(n);                                   //
```

70

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

71

```
  count = 1;                        //
  //              ,    n-1
  while (count < n)
  {  H.Remove(ed);            //
     u = F.Find(ed.tail);  v = F.Find(ed.head);
                         //                             u    v
     if (u != v)
     {                         //                  ,
        F.Union(u, v);        //       ,
        MST.Insert(ed);      //            MST
        count++;
     }
  }
}
```

72

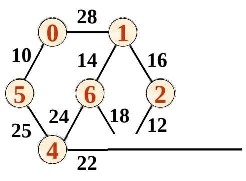
[redacted]

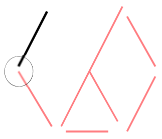
```
#include "heap.h"
template <class T, class E>
void Prim (Graph<T, E>& G, const T u0,
      MinSpanTree<T, E>& MST)
{
   MSTEdgeNode<T, E> ed;                          //
   int i, u, v, count;
   int n = G.NumberOfVertices();                  //
   int m = G.NumberOfEdges();                     //
   int u = G.getVertexPos(u0);                    //
   MinHeap <MSTEdgeNode<T, E>> H(m);              //
   bool Vmst = new bool[n];              //
```

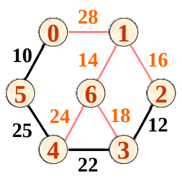
```
  MinHeap <MSTEdgeNode<T, E>> H(m);          //

  bool Vmst = new bool[n];              //
  for (i = 0; i < n; i++)
    Vmst[i] = false;

  Vmst[u] = true;                            //u
```

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

       while (v != -1)
       {                                        //      u
          if (!Vmst[v])
          {                                          //v       mst
             ed.tail = u;  ed.head = v;
             ed.cost = G.getWeight(u, v);
             H.Insert(ed);                           //(u,v)
          } //              u   mst   , v       mst
          v = G.getNextNeighbor(u, v);
       }
```

```
        while (!H.IsEmpty() && count < n)
        {
            H.Remove(ed);                   //
            if (!Vmst[ed.head])
            {
                MST.Insert(ed);                   //
                u = ed.head;  Vmst[u] = true;
                                            //u
                count++;
                break;
            }
        }

    } while (count < n);

}  // end of prim
```







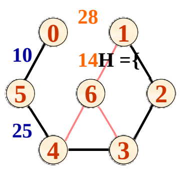
H = {(3,2,12), (3,6,18), (4,6,24),
(0,1,28)}

ed = (3, 2, 12)

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

⇩

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



82