11 Graphs

Graph Applications

- Modeling connectivity in computer networks
- Representing maps
- Modeling flow capacities in networks
- Finding paths from start to goal (AI)
- Modeling transitions in algorithms
- Ordering tasks
- Modeling relationships (families, organizations)

Contents

11.1 Terminology and Representations

11.2 Graph Implementations

11.3 Graph Traversals

11.4 Shortest-paths Problems

11.5 Minimum-Cost Spanning Trees

Contents

11.1 Terminology and Representations

11.2 Graph Implementations

11.3 Graph Traversals

11.4 Shortest-paths Problems

11.5 Minimum-Cost Spanning Trees

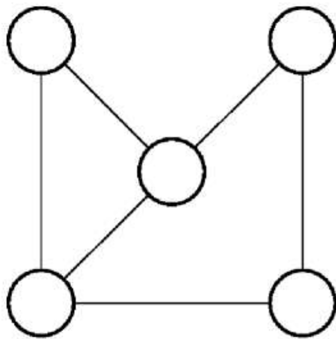
11.1 Terminology and Representations

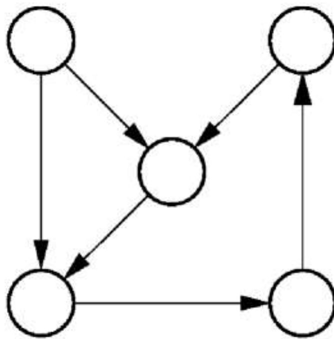
Graphs(1)

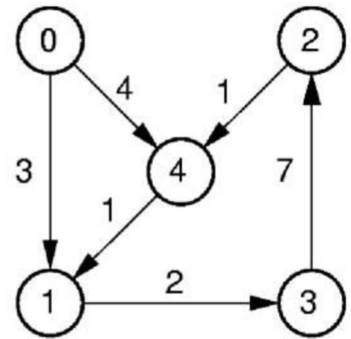
A graph **G** = (**V**, **E**) consists of a set of vertices **V**, and a set of edges **E**, such that each edge in **E** is a connection between a pair of vertices in **V**.

The number of vertices is written |**V**|, and the number edges is written |**E**|.

Graphs (2)


(a)


(b)



(c)

(a) A graph (undirected graph)

(b) A directed graph (digraph)

(c) A labeled (directed) graph with weights

Paths and Cycles

Path: A sequence of vertices $v_1, v_2, \ldots, v_n$ of length n-1 with an edge from v_i to v_{i+1} for $1 <= i < n$.

A path is simple if all vertices on the path are distinct.

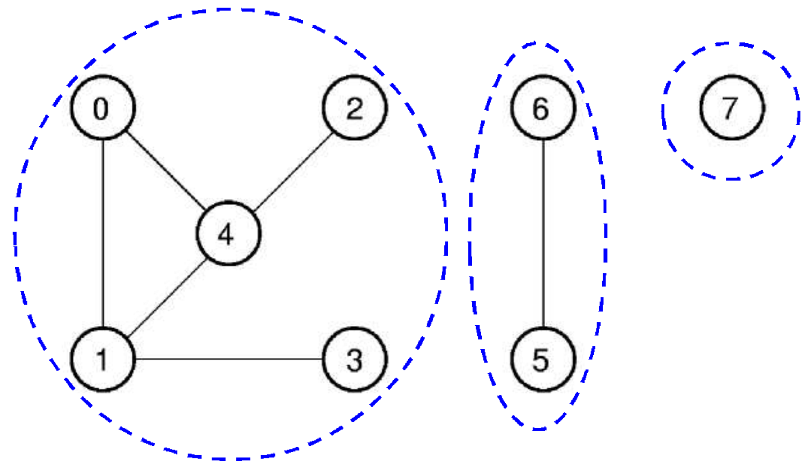
A cycle is a path of length 3 or more that connects v_i to itself.

A cycle is simple if the path is simple, except the first and last vertices are the same.

Connected Components

An undirected graph is connected if there is at least one path from any vertex to any other.

The maximum connected subgraphs of an undirected graph are called connected components(连通分量).



DAG and free tree

A graph without cycles is called acyclic.

DAG: A directed graph without cycles is called a directed acyclic graph or DAG有向无环图.

A free tree is a connected, undirected graph with no simple cycles.

(An equivalent definition is that a free tree is connected and has |V|-1 edges.)

假设图中有 n 个顶点，e 条边，则

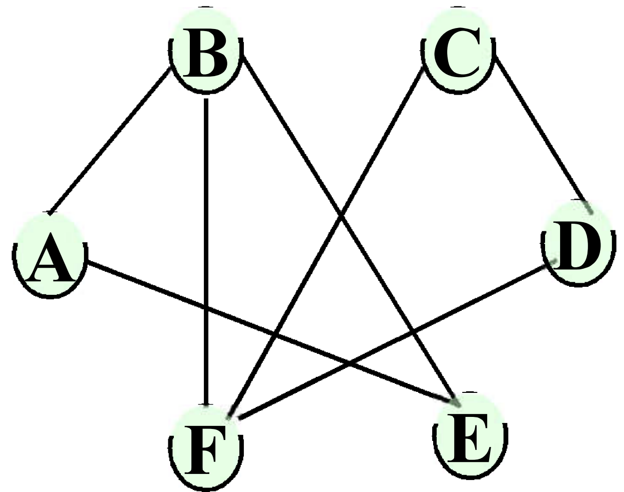
含有 $e=n(n-1)/2$ 条边的无向图称作无向完全图；

含有 $e=n(n-1)$ 条弧的有向图称作有向完全图；

若边或弧的个数 $e<n\log n$，则称作稀疏图，否则称作稠密图。

假若顶点v 和顶点w 之间存在一条边，则称顶点v和w互为邻接点，边(v, w)和顶点v和w相关联。

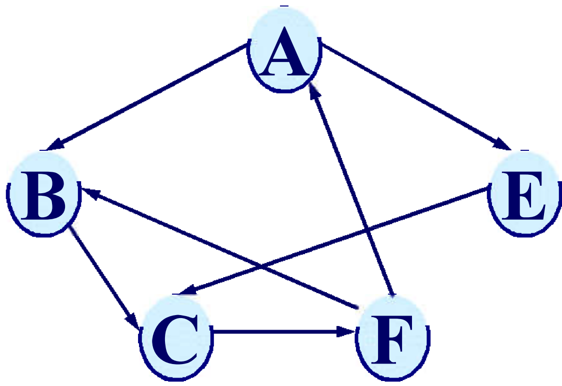
和顶点v 关联的边的数目定义为顶点的度。



TD(B) = 3

TD(A) = 2

有向图



OD(B) = 1

ID(B) = 2

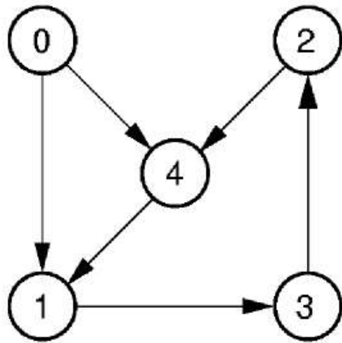
TD(B) = 3

顶点的出度:顶点v发出的边的数目;

顶点的入度:进入顶点v的边的数目。

顶点的度(TD)=出度(OD)+入度(ID)

Directed Representation

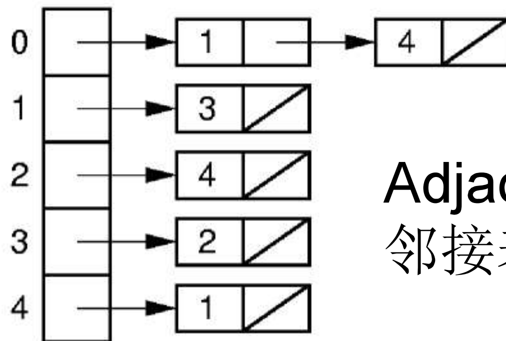


(a)

	0	1	2	3	4
0		1			1
1				1	
2					1
3			1		
4		1			

(b)

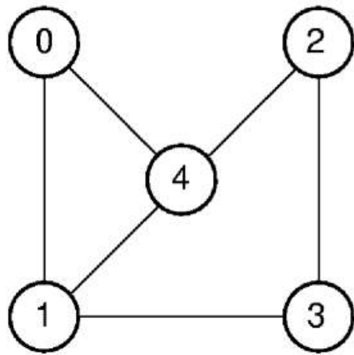
Adjacency matrix
邻接矩阵



(c)

Adjacency list
邻接表

Undirected Representation

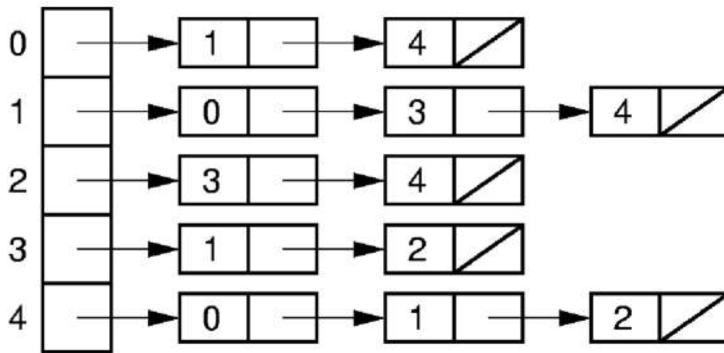


(a)

	0	1	2	3	4
0		1			1
1	1			1	1
2				1	1
3		1	1		
4	1	1	1		

(b)

Adjacency matrix
邻接矩阵



(c)

Adjacency list
邻接表

Representation Costs

- Adjacency Matrix:
  - requires $\Theta(|\mathbf{V}|^2)$ space
- Adjacency List:
  - requires $\Theta(|\mathbf{V}| + |\mathbf{E}|)$ space.
- $|\mathbf{E}|$ could be as low as 0, or as high as $\Theta(|\mathbf{V}|^2)$.
- Which representation actually requires less space depends on the number of edges.

Contents

11.1 Terminology and Representations

11.2 Graph Implementations

11.3 Graph Traversals

11.4 Shortest-paths Problems

11.5 Minimum-Cost Spanning Trees

11.2 Graph Implementations

Graph ADT

```
class Graph {  // Graph abstract class
public:
  virtual int n() =0;  // # of vertices
  virtual int e() =0;  // # of edges
  // Return index of first, next neighbor
  virtual int first(int) =0;
  virtual int next(int, int) =0;
  // Store new edge
  virtual void setEdge(int, int, int) =0;
  // Delete edge defined by two vertices
  virtual void delEdge(int, int) =0;
  // Weight of edge connecting two vertices
  virtual int weight(int, int) =0;
  virtual int getMark(int) =0;
  virtual void setMark(int, int) =0;
};
```

Adjacency Matrix Implementation(1)

```
class Graphm : public Graph {
private:
  int numVertex, numEdge;  //# of V, E
  int **matrix;  //adjacency matrix
  int *mark;  // mark array
public:
  Graphm(int numVert){  //Make graph w/ numVert V
   int i;
   numVertex = numVert;  numEdge = 0;
   mark = new int[numVert];  //Init mark array
   for(i=0;i<numVertex;i++) mark[i]=UNVISITED;
   matrix = (int**)new int*[numVertex];
   for (i=0; i<numVertex; i++)
     matrix[i] = new int[numVertex];
```

Adjacency Matrix Implementation(2)

```
    for (i=0; i< numVertex; i++) //Init 0 weights
      for(int j=0;j<numVertex;j++) matrix[i][j]=0;
}
~Graphm() { // Destructor
  delete [] mark;
  for (int i=0; i<numVertex; i++)
    delete [] matrix[i];
  delete [] matrix;
}
int n() { return numVertex; } //# of vertices
int e() { return numEdge; } // # of edges
int first(int v) { //first neighbor of "v"
  for (int i=0; i<numVertex; i++)
    if (matrix[v][i] != 0)  return i;
  return numVertex; // Return n if none
}
```

	0	1	2	3	4
0		1			1
1	1			1	1
2				1	1
3		1	1		
4	1	1	1		

Adjacency Matrix Implementation(3)

	0	1	2	3	4
0		1			1
1	1			1	1
2				1	1
3		1	1		
4	1	1	1		

```
int next(int v1,int v2){
 //next neighbor after v2
  for(int i=v2+1; i<numVertex; i++)
    if (matrix[v1][i] != 0) return i;
  return numVertex;  // Return n if none
}
void setEdge(int v1, int v2, int wgt) {
  Assert(wgt>0, "Illegal weight value");
  if (matrix[v1][v2] == 0) numEdge++;
  matrix[v1][v2] = wgt;
}
```

Adjacency Matrix Implementation(4)

```
void delEdge(int v1, int v2) {//Del edge(v1, v2)
  if (matrix[v1][v2] != 0) numEdge--;
  matrix[v1][v2] = 0;
}
int weight(int v1, int v2) {
   return matrix[v1][v2]; }
int getMark(int v) { return mark[v]; }
void setMark(int v, int val) { mark[v] = val; }
};
```

Contents

11.1 Terminology and Representations

11.2 Graph Implementations

11.3 Graph Traversals

11.4 Shortest-paths Problems

11.5 Minimum-Cost Spanning Trees

11.3 Graph Traversals

Graph Traversals

Some applications require visiting every vertex in the graph exactly once.

The application may require that vertices be visited in some special order based on graph topology.

Examples:

- Artificial Intelligence Search
- Shortest paths problems

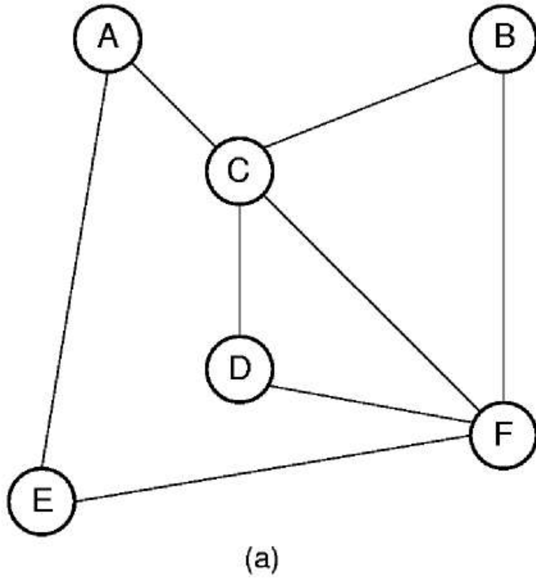
Graph Traversals (2)

To insure visiting all vertices:

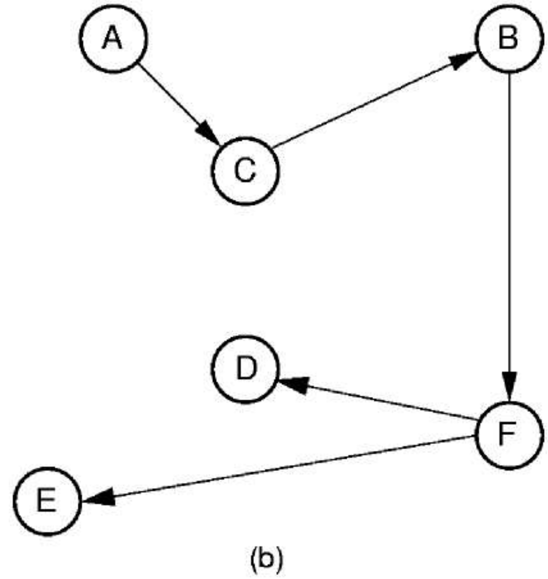
```
void graphTraverse(const Graph* G) {
  for (v=0; v<G->n(); v++)
    G->setMark(v, UNVISITED);// Initialize
  for (v=0; v<G->n(); v++)
    if (G->getMark(v) == UNVISITED)
      doTraverse(G, v);
}
```

Depth First Search (1)

深度优先遍历



(a)



(b)

Depth First Search (2)

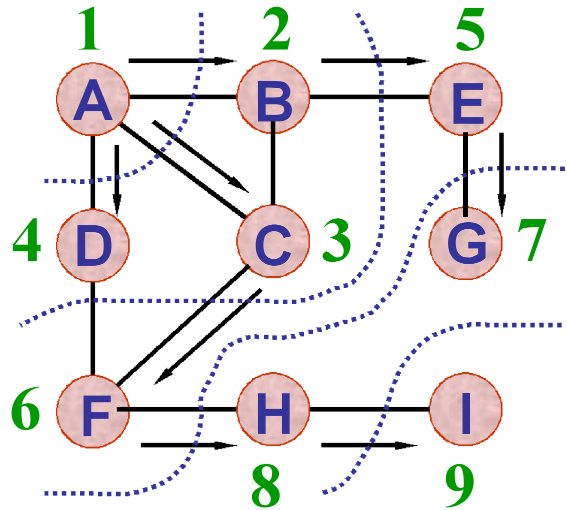
```
// Depth first search
void DFS(Graph* G, int v) {
  PreVisit(G, v);  // Take action
  G->setMark(v, VISITED);
  for (int w=G->first(v); w<G->n();
                        w = G->next(v,w))
    if (G->getMark(w) == UNVISITED)
      DFS(G, w);
  PostVisit(G, v); // Take action
}
```

Breadth First Search (1)

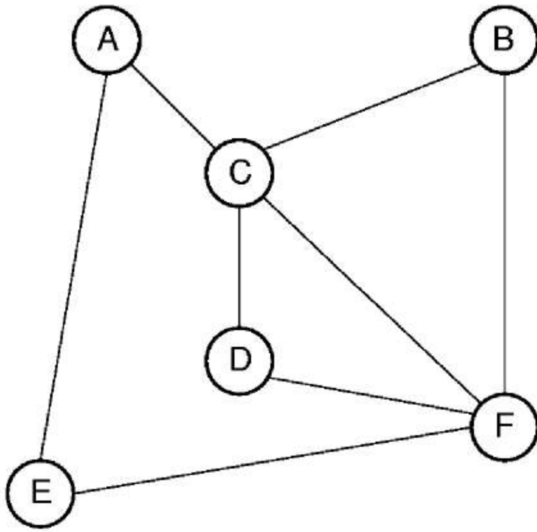
广度优先遍历

Like DFS, but replace stack with a queue.

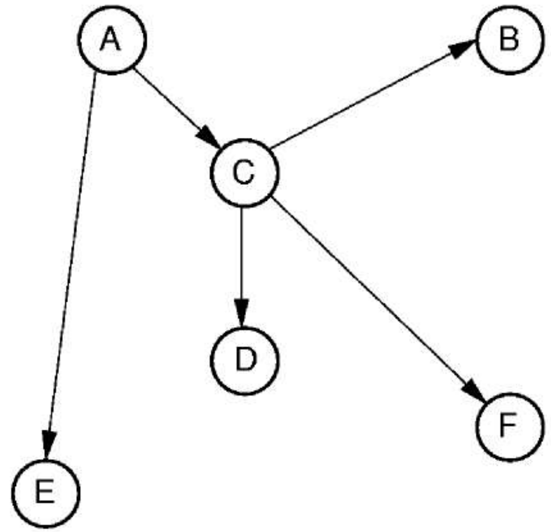
– Visit vertex's all neighbors before continuing deeper in the tree.



Breadth First Search (2)



(a)



(b)

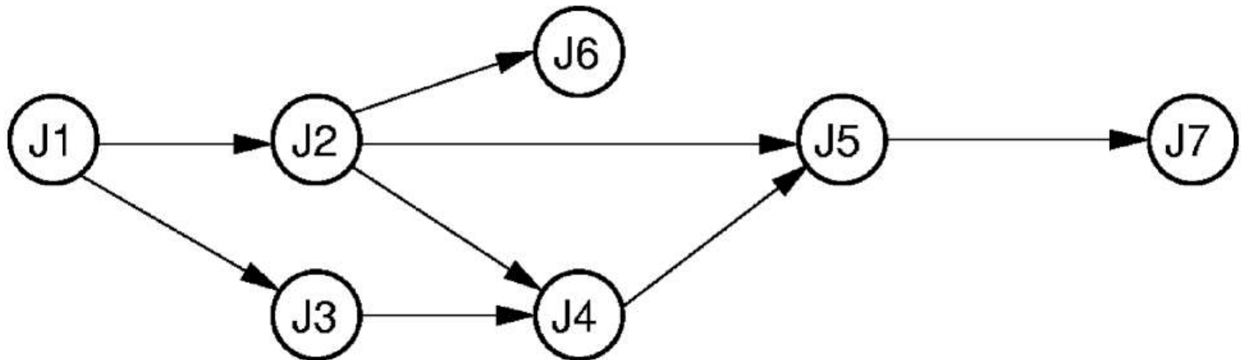
Breadth First Search (3)

```
void BFS(Graph* G, int start,Queue<int>*Q)
  {int v, w;
  Q->enqueue(start);    // Initialize Q
  G->setMark(start, VISITED);
  while (Q->length()!= 0) {// Process Q
    Q->dequeue(v);
    PreVisit(G, v);  // Take action
    for(w=G->first(v);w<G->n();
                               w=G->next(v,w))
      if (G->getMark(w) == UNVISITED) {
        G->setMark(w, VISITED);
        Q->enqueue(w);  }
   }
}
```

Topological Sort (1)拓扑排序

- Problem: Given a set of jobs, courses, etc., **with prerequisite constraints**, output the jobs in an order that does not violate any of the prerequisites.
- The problem is modeled with a DAG.
- The process of laying out the vertices of a DAG in a linear order to meet the prerequisite rules is called a **topological sort**.

Topological Sort (2)



J1 J2 J3 J6 J4 J5 J7

Or J1 J3 J2 J4 J5 J7 J6

Or ……

Queue-Based Topsort(1)

(1)从有向图中选取一个没有前驱的顶点，并输出之;

(2)从有向图中删去此顶点以及所有它发出的边;

(3)重复上述两步，直至图空，或者图不空但找不到无前驱的顶点为止。

Queue-Based Topsort(2)

a b h c d g f e

Queue-Based Topsort(3)

在算法中需要用定量的描述替代定性的概念

没有前驱的顶点 ==

入度为零的顶点

删除顶点及它发出的边 ==

边的终端顶点的入度减1

Queue-Based Topsort(4)

```
void topsort(Graph* G, Queue<int>* Q) {
  int Count[G->n()];  int v, w;
  for (v=0; v<G->n(); v++) Count[v] = 0;
  for (v=0; v<G->n(); v++) // Process edges
   for(w=G->first(v);w<G->n();w=G->next(v,w))
       Count[w]++;   // Add to w's count
  for (v=0; v<G->n(); v++) // Initialize Q
    if (Count[v] == 0) Q->enqueue(v);
  while (Q->length() != 0) {
    Q->dequeue(v);  printout(v);
    for (w=G->first(v); w<G->n();
                        w = G->next(v,w)) {
      Count[w]--;    // One less prereq
      if (Count[w] == 0) Q->enqueue(w);
   }
} }
```

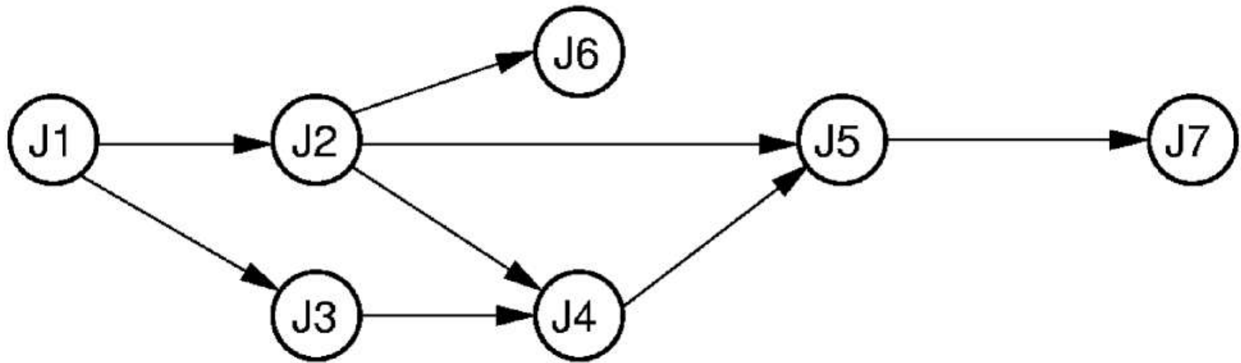
Stack-Based Topsort(1)

- A topological sort may be found by performing a DFS on the graph.
- When a vertex is visited, no action is taken.
- When the recursion pops back to that vertex, function PostVisit prints the vertex.
- This yields a topological sort in reverse order. It does not matter where the sort starts, as long as all vertices are visited in the end.

Stack-Based Topsort(2)

```
void topsort(Graph* G) {
for (i=0; i<G->n(); i++) // Initialize
    G->setMark(i, UNVISITED);
  for (i=0; i<G->n(); i++) //Do vertices
    if (G->getMark(i) == UNVISITED)
      tophelp(G, i);       //Call helper
}
void tophelp(Graph* G, int v) {
  G->setMark(v, VISITED);
  for (int w=G->first(v); w<G->n();
                          w = G->next(v,w))
    if (G->getMark(w) == UNVISITED)
      tophelp(G, w);
  printout(v);    //PostVisit for v
}
```

Stack-Based Topsort(3)



- Starting at J1 and visiting adjacent neighbors in alphabetic order
- vertices are printed: J7, J5, J4, J6, J2, J3, J1
- Reverse: the legal topological sort J1, J3, J2, J6, J4, J5, J7

Contents

11.1 Terminology and Representations

11.2 Graph Implementations

11.3 Graph Traversals

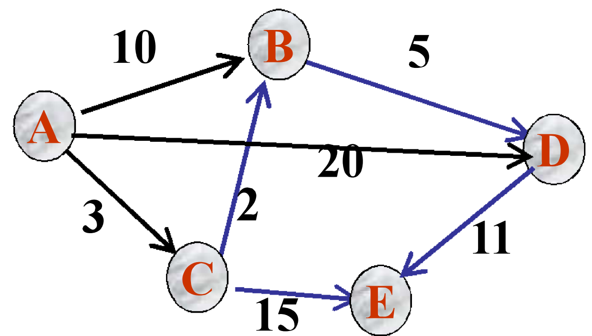
11.4 Shortest-paths Problems

11.5 Minimum-Cost Spanning Trees

11.4 Shortest-paths Problems
最短路径问题

Shortest Paths Problems

Input: A graph with <u>weights</u> or <u>costs</u> associated with each edge.



Output: The list of edges forming the shortest path.

Sample problems:

- Find shortest path between two named vertices
- Find shortest path from *S* to all other vertices
- Find shortest path between all pairs of vertices

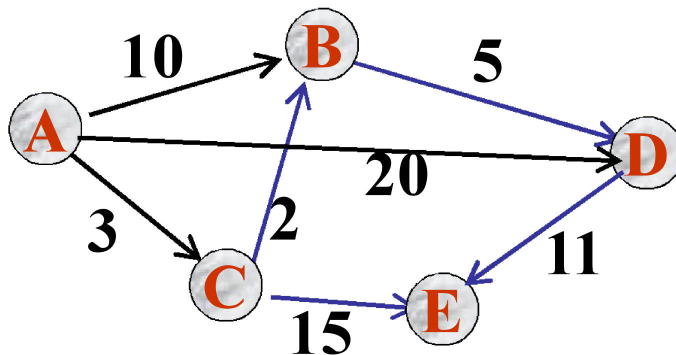
Shortest Paths Definitions

d(A, B) is the shortest distance from vertex A to B.

w(A, B) is the weight of the edge connecting A to B.

- If there is no such edge, then w(A, B) = ∞.

w(A, D) = 20; d(A, D) = 10



Single-Source Shortest Paths(1)

Given start vertex *s*, find the shortest path from *s* to all other vertices.

- Can I only find the shortest path between two vertices, *s* and *t* ?
- ☹ In the worst case, while finding the shortest path from *s* to *t*, we might find the shortest paths from *s* to every other vertex as well.
- So there is no better algorithm (in the worst case) for finding the shortest path to a single vertex than to find shortest paths to all vertices.

Single-Source Shortest Paths(2)

- For unweighted graphs (or whenever all edges have the same cost), the single-source shortest paths can be found using a simple BFS traversal.
- When weights are different, BFS will not give the correct answer.
- Solution: Process vertices in order of the shortest-path distance from *s*.
- This solution is known as Dijkstra's algorithm

Single-Source Shortest Paths(3)

Dijkstra's algorithm

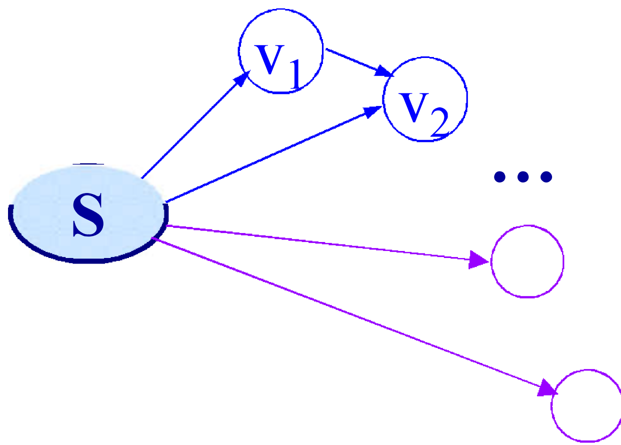
Dijkstra提出按路径长度的递增次序，逐步产生最短路径的算法。

首先求出长度最短的一条最短路径，再参照它求出长度次短的一条最短路径，依次类推。

Single-Source Shortest Paths(4)

- 路径长度最短的最短路径的特点

在这条路径上，必定只含一条边，并且这条边的权值最小。



第一条最短路径肯定是从S直接到某个顶点v_i的路径。

Single-Source Shortest Paths(3)

- 下一条路径长度次短的最短路径的特点

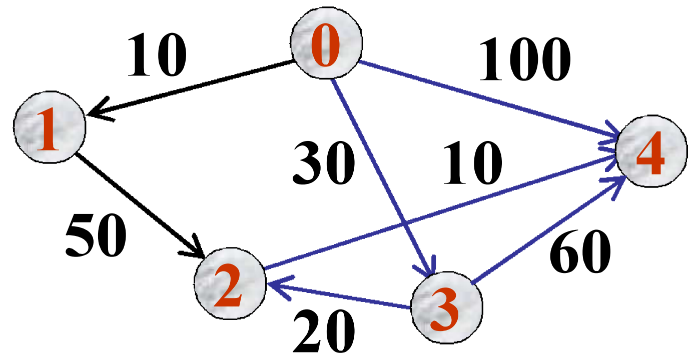
只可能有两种情况：或者是直接从S到某顶点(只含一条边)；或者是从S经过顶点v_i (具有最短路径的顶点），再到达该顶点(由两条边组成)。

Single-Source Shortest Paths(4)

- 其余最短路径的特点

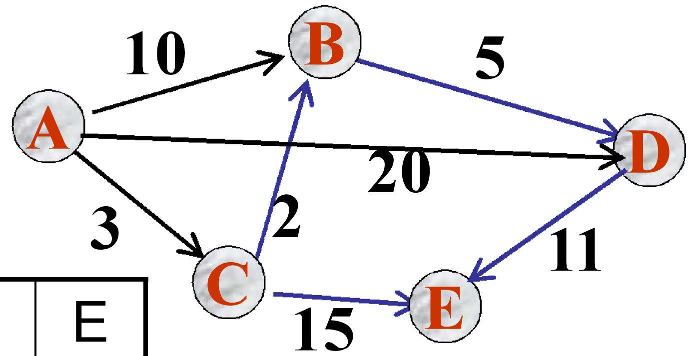
或者是直接从**S**到某顶点**(**只含一条边**)**；　或者是从**S**经过已求得最短路径的顶点，再到达该顶点。

Dijkstra's Algorithm Example(1)



Src	Dst	Shortest Path	Path Length
v_0	v_1	(v_0,v_1)	10
	v_2	(v_0,v_3,v_2)	50
	v_3	(v_0,v_3)	30
	v_4	(v_0,v_3,v_2,v_4)	60

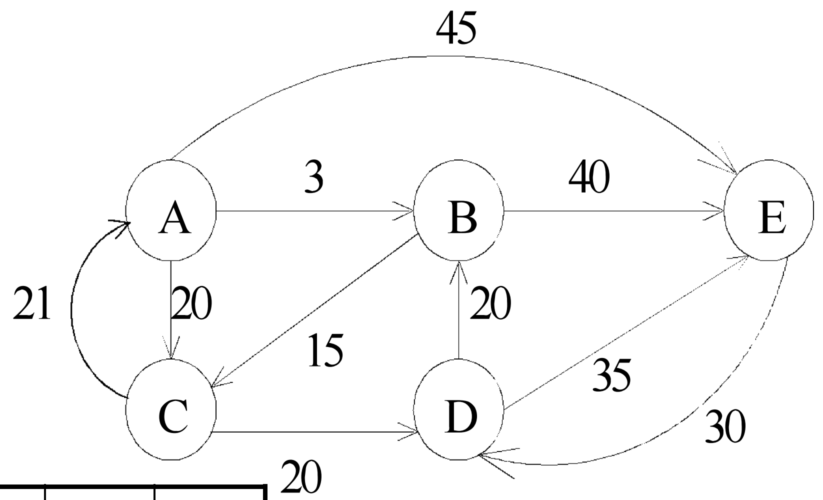
Dijkstra's Algorithm Example(2)



	A	B	C	D	E
Initial	0	∞	∞	∞	∞
Process A	0	10	3	20	∞
Process C	0	5	3	20	18
Process B	0	5	3	10	18
Process D	0	5	3	10	18
Process E	0	5	3	10	18

Can you try?

Source: A



	A	B	C	D	E
Initial	0	∞	∞	∞	∞
Process A	0	3	20	∞	45
Process B	0	3	18	∞	43
Process C	0	3	18	38	43
Process D	0	3	18	38	43
Process E	0	3	18	38	43

Dijkstra's Implementation

```
// Source is s, result in D
void Dijkstra(Graph* G, int* D, int s)
  {int i, v, w;
   for (i=0; i<G->n(); i++) {
     v=minVertex(G,D);//下一个具有最短路径的顶点
     if (D[v] == INFINITY) return;
     G->setMark(v, VISITED);
     for (w=G->first(v); w<G->n();
                        w=G->next(v,w))
       if(D[w]>(D[v]+G->weight(v, w)))
         D[w] = D[v] + G->weight(v, w);
   }
}
```

Implementing `minVertex`

Issue: How to determine the next-closest vertex? (i.e., implement `minVertex`)

Approach 1: Scan through the table of current distances.

Approach 2: Store unprocessed vertices using a min-heap to implement a priority queue ordered by D value.

Approach 1

```
// Find min cost vertex
int minVertex(Graph* G, int* D) {
  int i, v;
  // Set v to an unvisited vertex
  for (i=0; i<G->n(); i++)
    if (G->getMark(i) == UNVISITED)
      { v = i; break; }
  for(i++;i<G->n();i++)//find smallest D
    if((G->getMark(i) == UNVISITED) &&
        (D[i] < D[v]))
      v = i;
  return v;
}
```

Approach 2

```
class DijkElem {
public:
  int vertex, distance;//顶点号，到S的最短距离
  ……
};
void Dijkstra(Graph* G, int* D, int s) {
  int i, v, w;        // v is current vertex
  DijkElem temp;
  DijkElem E[G->e()]; // Heap array
  temp.distance = 0; temp.vertex = s;
  E[0] = temp;      // Initialize heap array
  minheap<DijkElem, DDComp> H(E, 1, G->e());
```

Approach 2

```
for (i=0; i<G->n(); i++) {// Get distances
   do { if(!H.removemin(temp)) return;
        v = temp.vertex;
      } while (G->getMark(v) == VISITED);
   G->setMark(v, VISITED);//找到具有最短路径的顶点
   if (D[v] == INFINITY) return;
   for(w=G->first(v); w<G->n(); w=G->next(v,w))
     if (D[w] > (D[v] + G->weight(v, w))) {
       D[w] = D[v] + G->weight(v, w);
       temp.distance = D[w]; temp.vertex = w;
       H.insert(temp); // Insert in heap
     }
  }
}
```

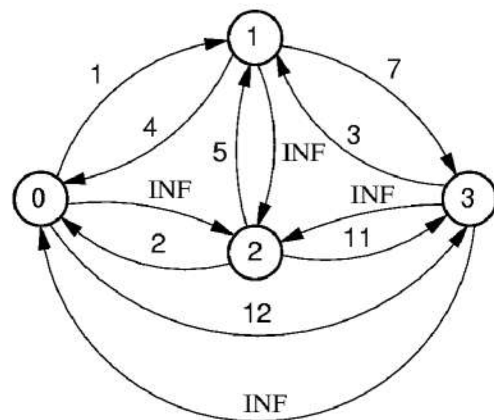
All-Pairs Shortest Paths

For every vertex u, $v \in \mathbf{V}$, calculate d(u, v).

Could run Dijkstra's Algorithm |**V**| times.

Better is Floyd's Algorithm.

Define a *k*-path from *u* to *v* to be any path whose intermediate vertices all have indices less than *k*.



Main idea:

- 若<v_i,v_j>存在，则存在路径{v_i,v_j}
  // 路径中不含其它顶点

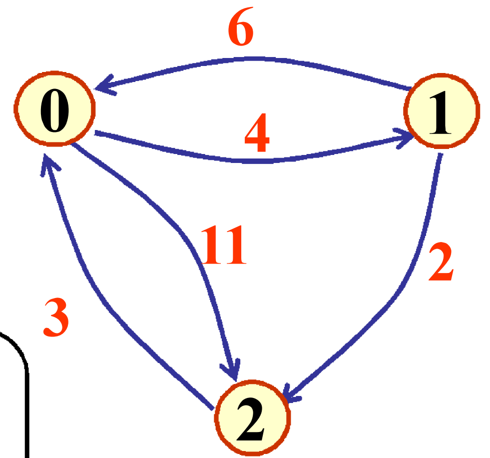
- 在路径上增加v_0，若<v_i,v_0>,<v_0,v_j>存在，则比较路径{v_i,v_0,v_j}和{v_i,v_j}长度，取长度短者为从v_i到v_j中间只经过序号不大于0的顶点的最短路径，即路径中所含顶点序号不大于0；

❖在路径上增加v_1，若$<v_i,\ldots,v_1>$,$<v_1,\ldots,v_j>$存在(它们分别是在前一步中找到的最短路径)，则将路径$\{v_i,\ldots,v_1,\ldots,v_j\}$的长度与以前找到的$v_i$到$v_j$最短路径长度比较，取长度短者为从$v_i$到$v_j$中间只经过序号不大于1的顶点的最短路径，即路径中所含顶点序号不大于1；

- 依次在路径上增加$v_2,\ldots,v_{n-1}$。

经过n次比较后，最后求得的必是从v_i到v_j的最短路径。

Floyd's Algorithm Example:



$$D^{-1} = \begin{pmatrix} 0 & 4 & 11 \\ 6 & 0 & 2 \\ 3 & \infty & 0 \end{pmatrix}$$

$$D^{0} = \begin{pmatrix} 0 & 4 & 11 \\ 6 & 0 & 2 \\ 3 & 7 & 0 \end{pmatrix}$$

D[2][1] = D[2][0] + D[0][1]

$$D^{1} = \begin{pmatrix} 0 & 4 & 6 \\ 6 & 0 & 2 \\ 3 & 7 & 0 \end{pmatrix}$$

D[0][2] = D[0][1] + D[1][2]

$$D^{2} = \begin{pmatrix} 0 & 4 & 6 \\ 5 & 0 & 2 \\ 3 & 7 & 0 \end{pmatrix}$$

D[1][0] = D[1][2] + D[2][0]

Floyd’s Algorithm

```
//Floyd's all-pairs shortest paths algorithm
void Floyd(Graph* G) {
  int D[G->n()][G->n()];  // Store distances
  for (int i=0; i<G->n(); i++) // Initialize
    for (int j=0; j<G->n(); j++)
      D[i][j] = G->weight(i, j);
  // Compute all k paths
  for (int k=0; k<G->n(); k++)
    for (int i=0; i<G->n(); i++)
      for (int j=0; j<G->n(); j++)
        if (D[i][j] > (D[i][k] + D[k][j]))
          D[i][j] = D[i][k] + D[k][j];
}
```

Contents

11.1 Terminology and Representations

11.2 Graph Implementations

11.3 Graph Traversals

11.4 Shortest-paths Problems

11.5 Minimum-Cost Spanning Trees

11.5 Minimum-Cost Spanning Trees
最小代价生成树

Minimum Cost Spanning Trees

Minimum-cost Spanning Tree (MST) Problem:

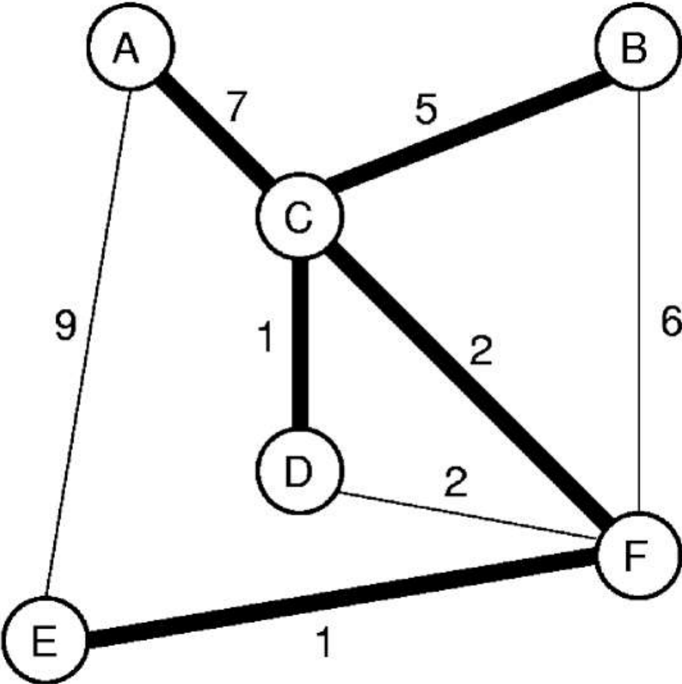
Input: An undirected, connected graph G.

Output: The subgraph of G that

1) has minimum total cost as measured by summing the values of all the edges in the subset, and

2) keeps the vertices connected.

MST Example



构造MST的准则

- *n* 个顶点的无向连通图的生成树有 *n* 个顶点、*n*-1 条边；
- 必须使用且仅使用该图中的*n*−1条边来联结图中的 *n* 个顶点；
- 不能使用产生回路的边；
- 各边上的权值的总和达到最小。

构造MST的算法

算法一： Prim算法
(普里姆)

算法二： Kruskal算法
(克鲁斯卡尔)

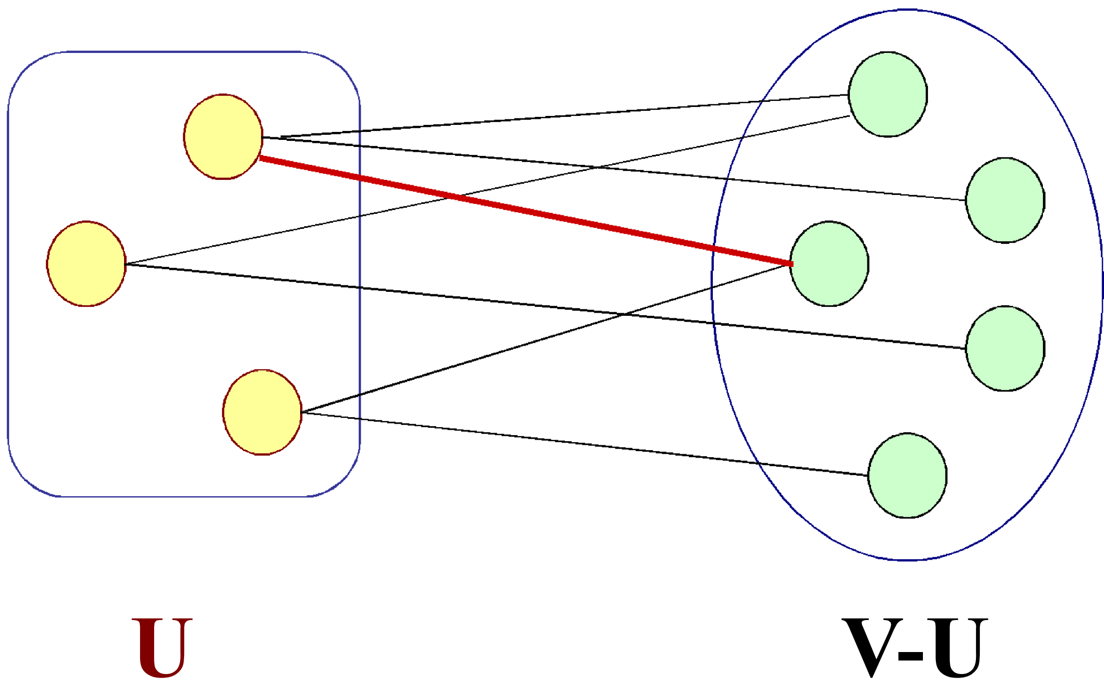
一、Prim算法的基本思想

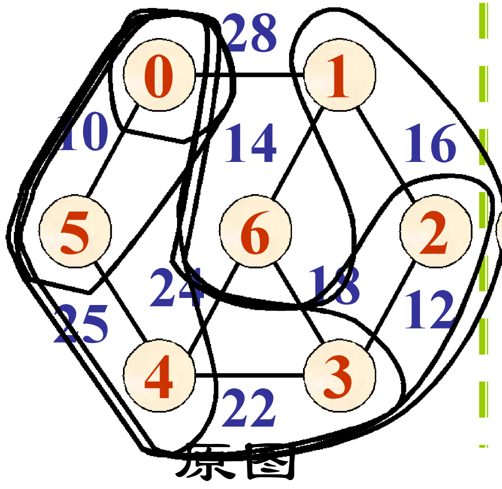
1.取图中任意一个顶点v作为生成树的根，之后往生成树上添加新的顶点w;

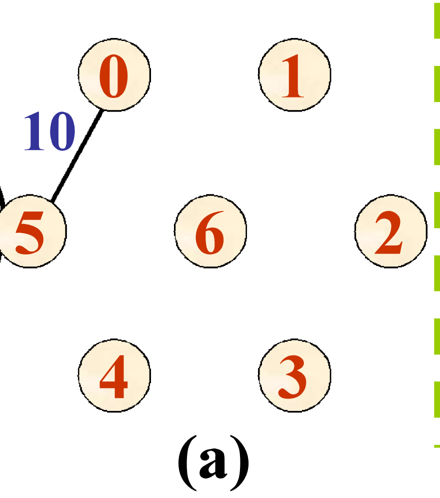
2.在添加的顶点 w 和已经在生成树上的顶点 v 之间必定存在一条边，并且该边的权值在所有连通顶点 v 和 w 之间的边中取值最小;

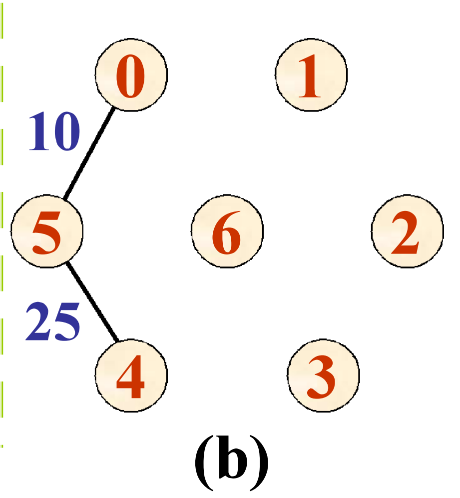
3.之后继续往生成树上添加顶点，直至生成树上含有 n-1 个顶点为止。

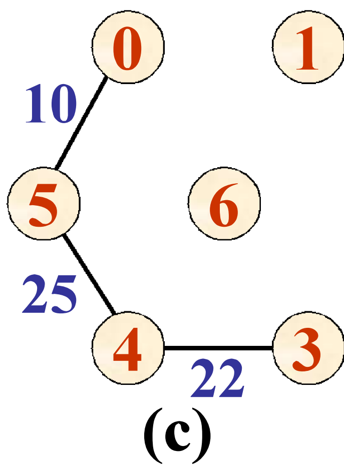
一般情况下所添加的顶点应满足下列条件：在生成树的构造过程中，图中顶点集合V分属两个集合：已落在生成树上的顶点集 U 和尚未落在生成树上的顶点集V-U，则应在所有连通U中顶点和V-U中顶点的边中选取权值最小的边。

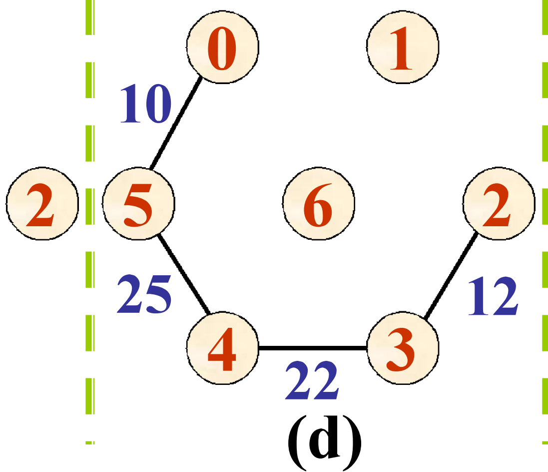

U
V-U

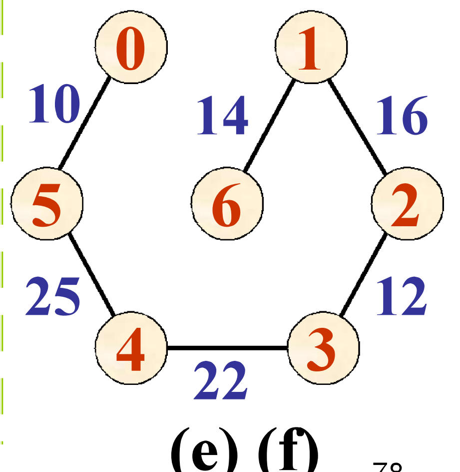

28
0
1
10
14
16
5
6
2
24
18
25
12
4
3
22
原图


0
1
10
5
6
2
4
3
(a)


0
1
10
5
6
2
25
4
3
(b)


0
1
10
5
6
2
25
4
3
22
(c)


0
1
10
5
6
2
25
12
4
3
22
(d)

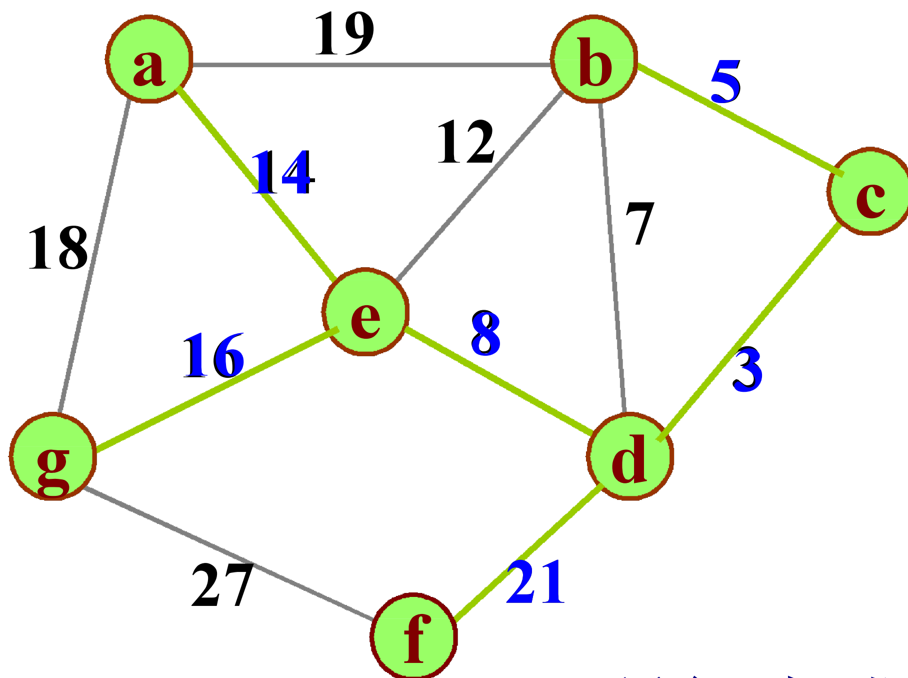

0
1
10
14
16
5
6
2
25
12
4
3
22
(e) (f)

Prim's MST Algorithm

- Prim's algorithm is quite similar to Dijkstra's algorithm for finding the single-source shortest paths.
- The primary difference is that we are seeking not the next closest vertex to the start vertex, but rather the next closest vertex to any vertex currently in the MST.

Prim’s MST Algorithm

```
void Prim(Graph* G, int* D, int s) {
  int MSTV[G->n()];  // Who's closest
  int i, w;
  for (i=0; i<G->n(); i++) {// Do vertices
    int v = minVertex(G, D);
    G->setMark(v, VISITED);
    if (v != s) AddEdgetoMST(MSTV[v], v);
    if (D[v] == INFINITY) return;
    for (w=G->first(v); w<G->n();
                        w = G->next(v,w))
      if (D[w] > G->weight(v,w)) {
        D[w] = G->weight(v,w);// Update dist
        MSTV[w]=v; //Update who it came from
      }
  }
}
```



所得生成树权值和

= 14+8+3+5+16+21 = 67

二、Kruskal算法的基本思想

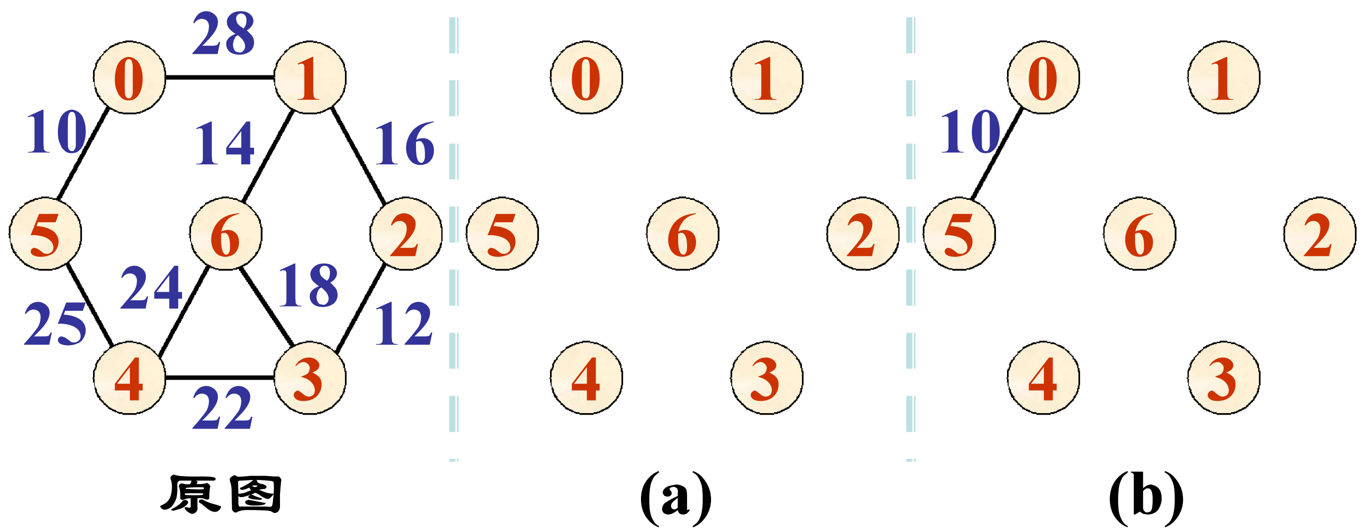
- 基本实现思路：为使生成树上边的权值之和达到最小，则应使生成树中每一条边的权值尽可能地小。

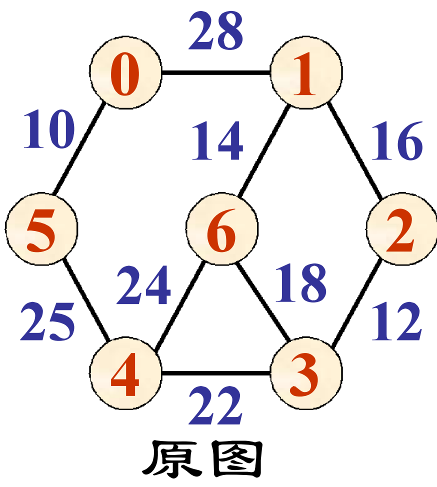
- 具体做法: 先构造一个只含 n 个顶点的子图 SG，然后从权值最小的边开始，若它的添加不使 SG 中产生回路，则在 SG 上加上这条边，如此重复，直至加上 n-1 条边为止。

Kruskal's MST Algorithm

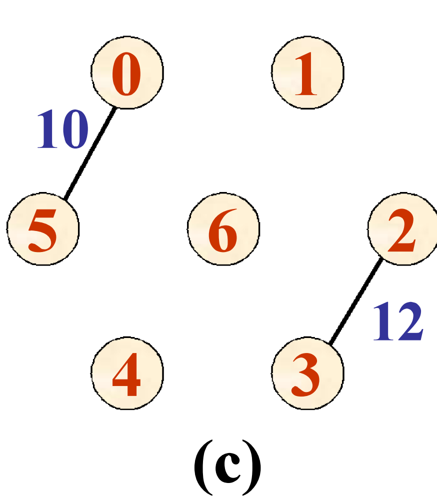
- Initially, each vertex is in its own MST.
- Merge two MST's that have the shortest edge between them.
  - Use a priority queue to order the unprocessed edges. Grab next one at each step.
- How to tell if an edge connects two vertices already in the same MST?
  - Use the UNION/FIND algorithm with parent-pointer representation.

Kruskal's MST Example (1)

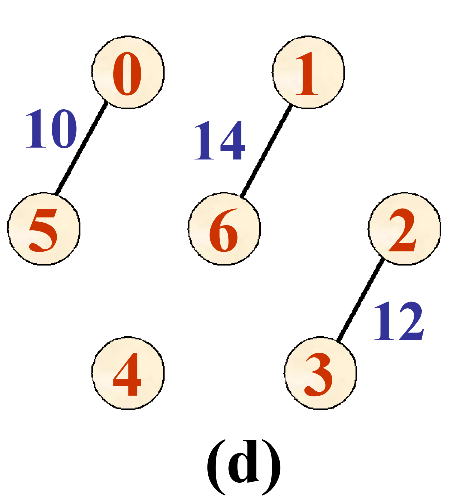




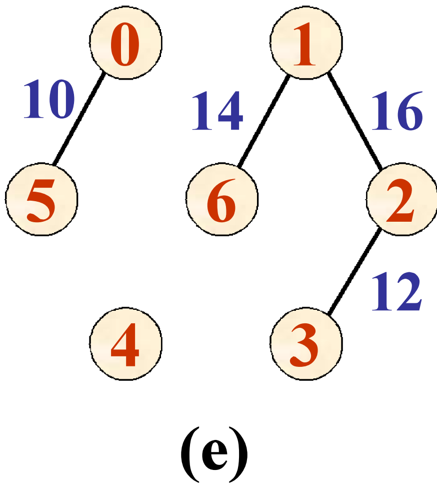
原图



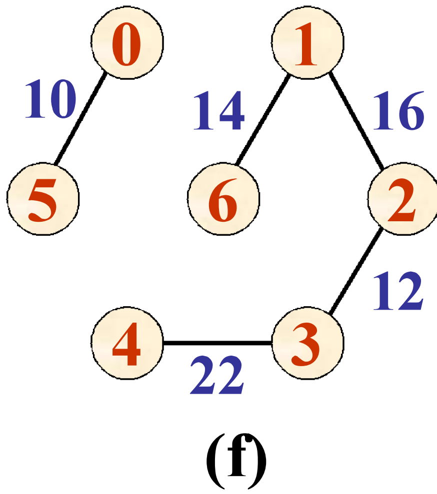
(c)



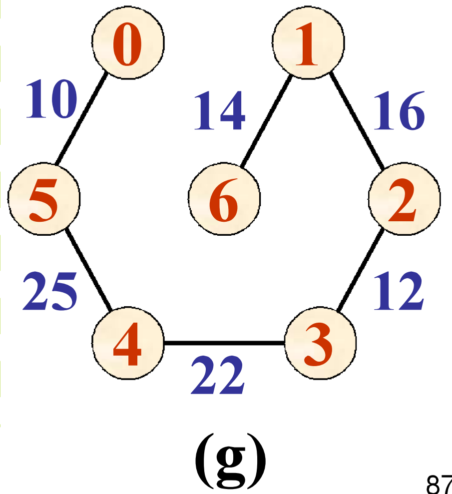
(d)



(e)



(f)



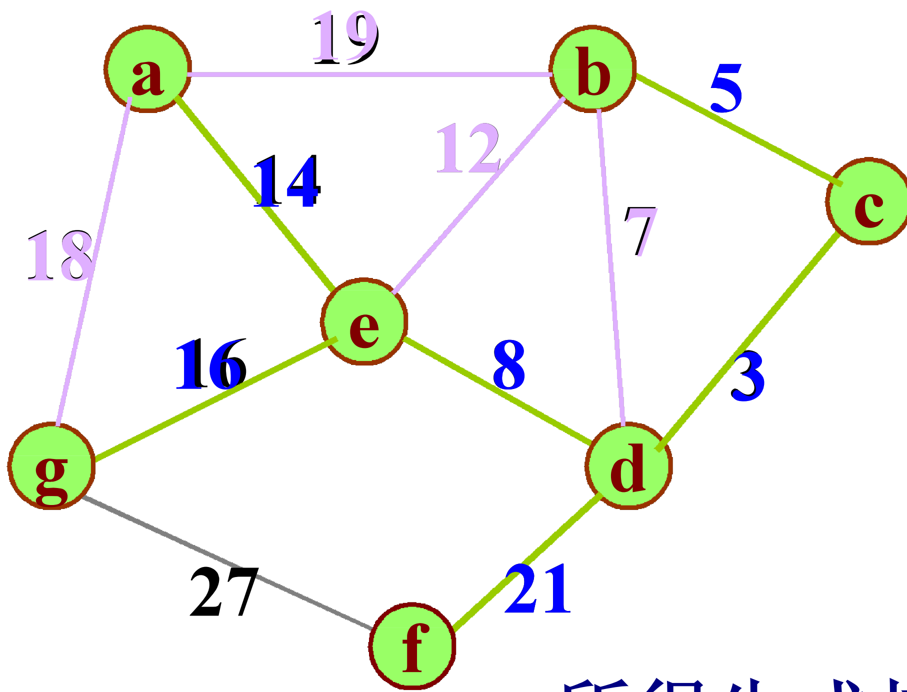
(g)

Kruskal's Algorithm Implementation(1)

```
class KruskElem { // An element for the heap
public:
  int from, to, distance; // The edge
  ……
};
void Kruskal(Graph* G) {// Kruskal's algorithm
  Gentree A(G->n()); // Equivalence class array
  KruskElem E[G->e()];//Edges Array for min-heap
  int i,w,edgecnt = 0;
  for (i=0; i<G->n(); i++) //Put edges on array
    for(w=G->first(i); w<G->n(); w=G->next(i,w))
    {
      E[edgecnt].distance = G->weight(i, w);
      E[edgecnt].from = i;
      E[edgecnt++].to = w;
    }
```

Kruskal's Algorithm Implementation(2)

```
// Heapify the edges
minheap<KruskElem, KKComp> H(E,edgecnt,edgecnt);
int numMST = G->n(); //Initially n equiv classes
for (i=0; numMST>1; i++){//Combine equiv classes
  KruskElem temp;
  H.removemin(temp); // Get next cheapest edge
  int v = temp.from; int u = temp.to;
  if(A.differ(v, u)) {//If in diff equiv classes
     A.UNION(v, u); // Combine equiv classes
     AddEdgetoMST(temp.from, temp.to);
     numMST--; // One less MST
    }
  }
}
```



所得生成树权值和

= 14+8+3+5+16+21 = 67

Kruskal's MST Algorithm Cost

Cost is dominated by the time to remove edges from the heap.

- Can stop processing edges once all vertices are in the same MST

Total cost: $\Theta(|\mathbf{V}| + |\mathbf{E}| \log |\mathbf{E}|)$.

Summary

11.1 Terminology and Representations

11.2 Graph Implementations

11.3 Graph Traversals

11.4 Shortest-paths Problems

11.5 Minimum-Cost Spanning Trees

Homework

11.3

11.10

11.17

11.18

补充材料：关键路径

关键路径(Critical Path)

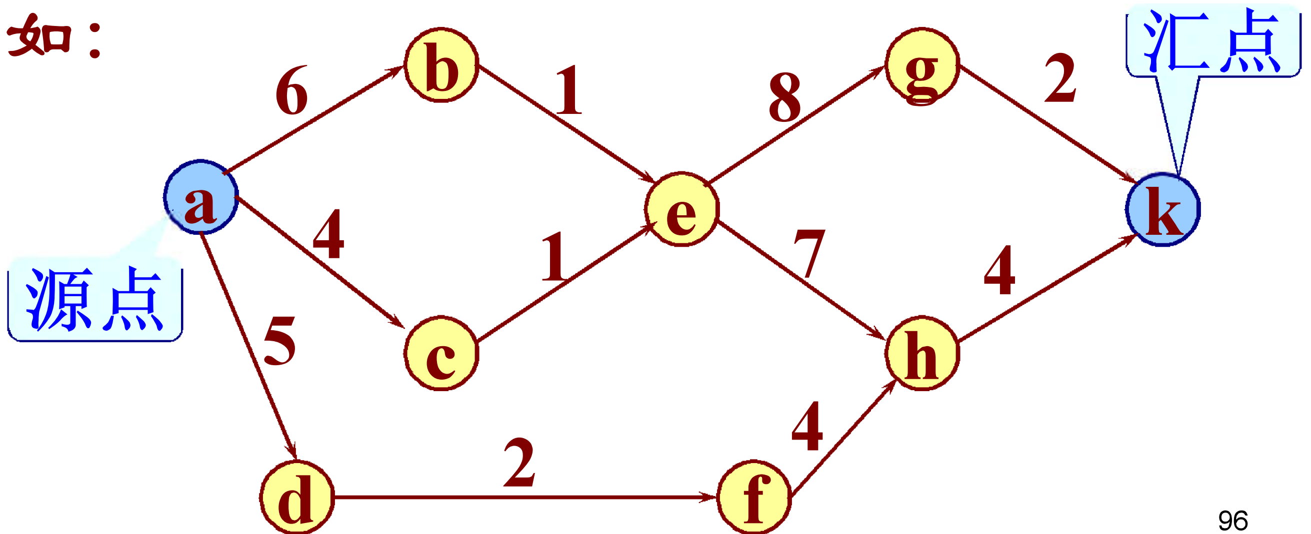
问题: 假设以有向网表示一个施工流程图，弧及其权值表示各项子工程及完成该项子工程所需时间。

问：哪些子工程项是“关键工程”？

即：哪些子工程项将影响整个工程的完成期限?

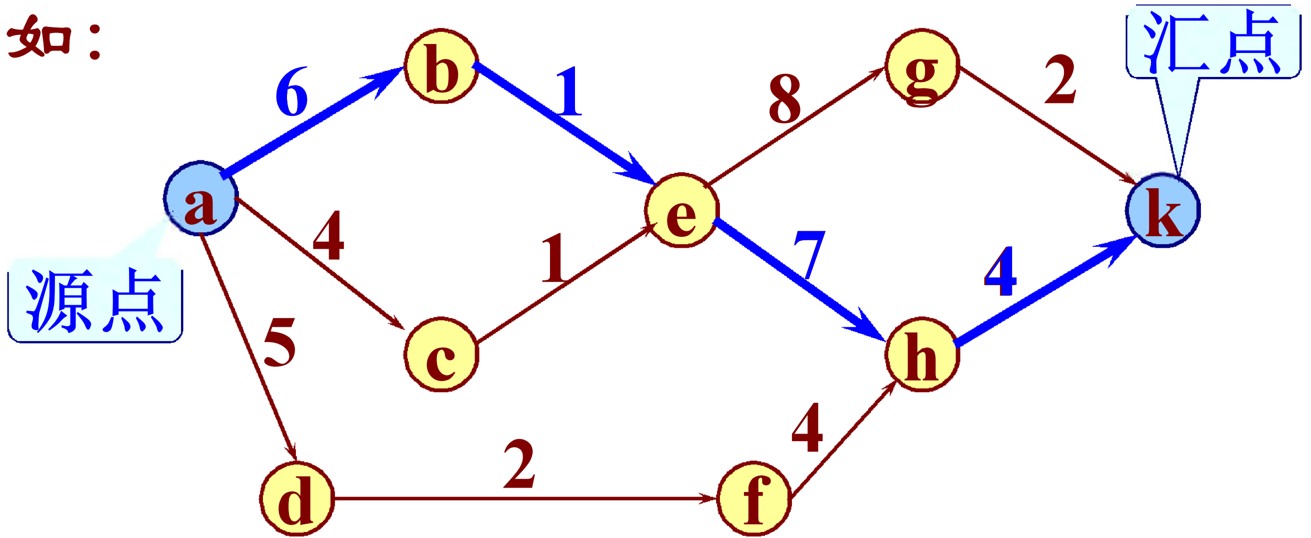
用有向边表示一个工程中的活动(Activity), 用边上权值表示活动持续时间, 用顶点表示事件(Event)，为AOE (Activity On Edges)网。

例如：



整个工程完成的时间为：从有向图的源点到汇点的最长路径。

例如：



“关键活动”指的是： 该弧上的权值增加将使有向图上的最长路径的长度增加。

完成整个工程所需的时间取决于从源点到汇点的最长路径长度, 即在这条路径上所有活动的持续时间之和。

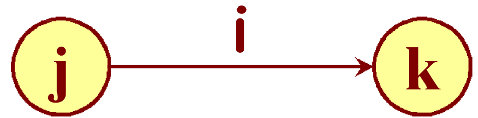
这条路径长度最长的路径就叫做关键路径。

要找出关键路径，必须找出关键活动，即不按期完成就会影响整个工程完成的活动。关键路径上的所有活动都是关键活动。

因此，只要找到了关键活动，就可以找到关键路径。

如何求关键活动？

假设第i条弧为〈j, k〉



“事件(顶点)” 的最早发生时间 *ve*(j)

***ve*(j) = 从源点到顶点j的最长路径长度；**

“事件(顶点)” 的最迟发生时间 *vl*(k)

***vl*(k) =从源点到汇点的最长路径长度－**

从顶点k到汇点的最长路径长度。

对第 i 项活动

“活动(弧)”的 最早开始时间 *ee*(i)

$ee(i) = ve(j)$;

“活动(弧)”的 最迟开始时间 *el*(i)

$el(i) = vl(k) - dut(<j,k>)$;

ve(源点) = *vl*(源点) = 0

vl(汇点) = *ve*(汇点)

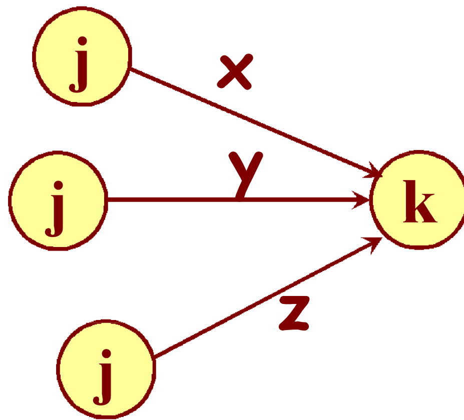
关键活动: *el*(i) = *ee*(i)

事件发生时间的计算公式：

ve(源点) = 0;

ve(k) = Max{ve(j) + dut(<j, k>)}

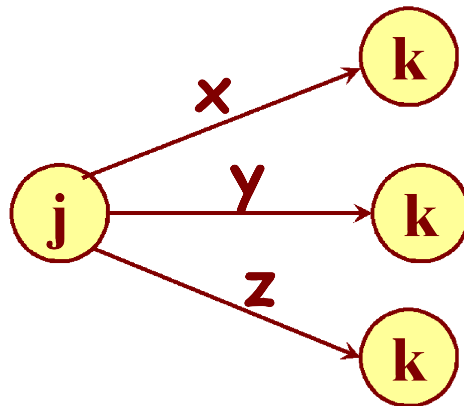
(2<=k<=n, j属于所有以k为弧头的弧尾顶点的集合)



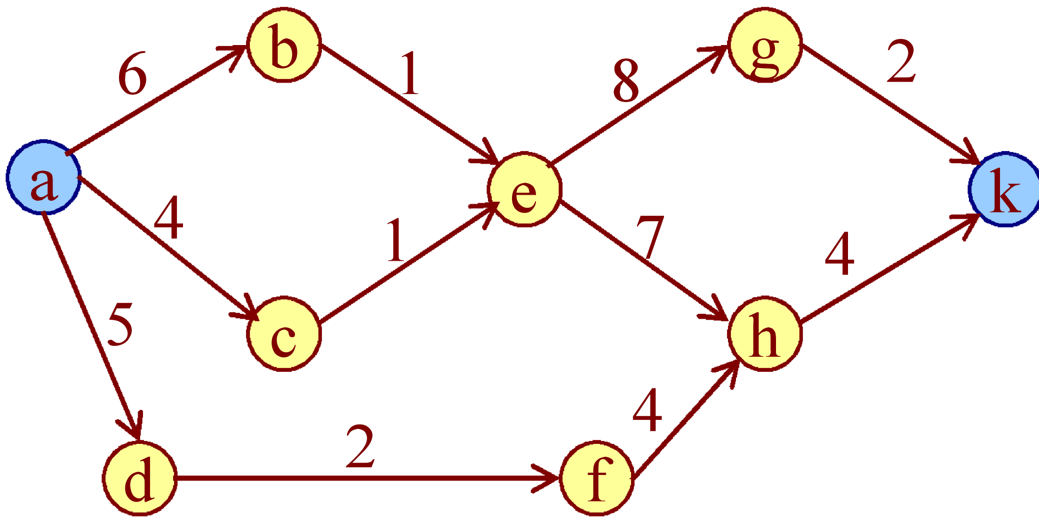
vl(汇点) = ve(汇点);

vl(j) = $Min\{vl(k) - dut(<j, k>)\}$

(1<=j<=n-1, k属于所有以j为弧尾的弧头顶点的集合)



这两个递推公式的计算必须分别在拓扑有序及逆拓扑有序的前提下进行



拓扑有序序列: a - d - f - c - b - e - h - g - k

	a	b	c	d	e	f	g	h	k
ve	0	6	4	5	7	7	15	14	18
vl	0	6	6	8	7	10	16	14	18

	a	b	c	d	e	f	g	h	k
ve	0	6	4	5	7	7	15	14	18
vl	0	6	6	8	7	10	16	14	18

$$ee(i) = ve(j)$$

$$el(i) = vl(k) - dut(\langle j,k \rangle)$$

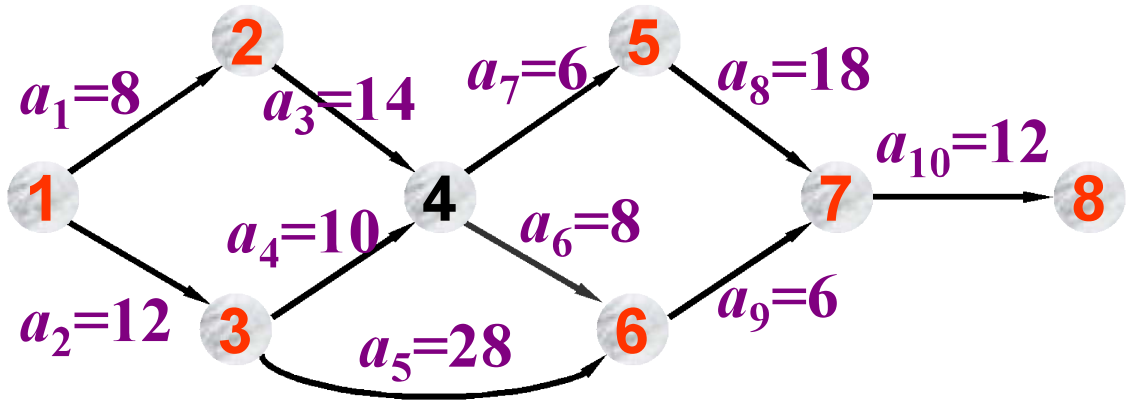
	ab	ac	ad	be	ce	df	eg	eh	fh	gk	hk
权	6	4	5	1	1	2	8	7	4	2	4
ee	0	0	0	6	4	5	7	7	7	15	14
el	0	2	3	6	6	8	8	7	10	16	14
	✓			✓				✓			✓

显然 求ve的顺序应该是按拓扑有序的次序；

而 求vl的顺序应该是按拓扑逆序的次序；

因为 拓扑逆序序列即为拓扑有序序列的逆序列；

因此 应该在拓扑排序的过程中，另设一个“栈”记下拓扑有序序列。



	1	2	3	4	5	6	7	8
ve	0	8	12	22	28	40	46	58
vl	0	8	12	22	28	40	46	58

	a_1	a_2	a_3	a_4	a_5	a_6	a_7	a_8	a_9	a_{10}
ee	0	0	8	12	12	22	22	28	40	46
el	0	0	8	12	12	32	22	28	40	46

注　意

- 仅计算 *ve*[*i*] 和 *vl*[*i*] 是不够的，还须计算 *ee*[*k*] 和 *el*[*k*]
- 不是任一关键活动加速一定能使整个工程提前
- 想使整个工程提前，要考虑各个关键路径上所有关键活动