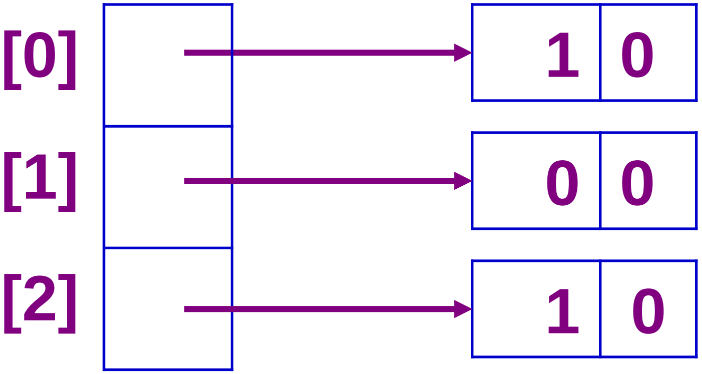
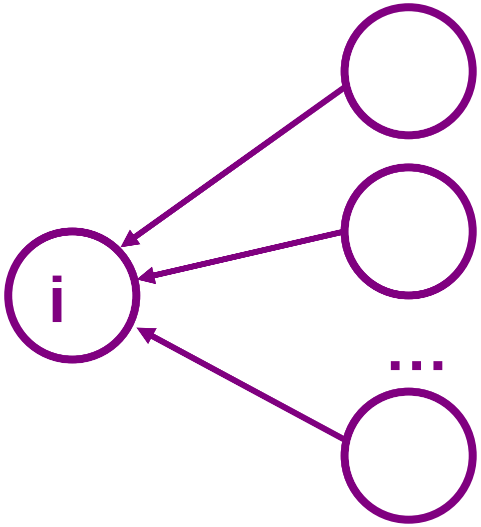
Inverse adjacency lists

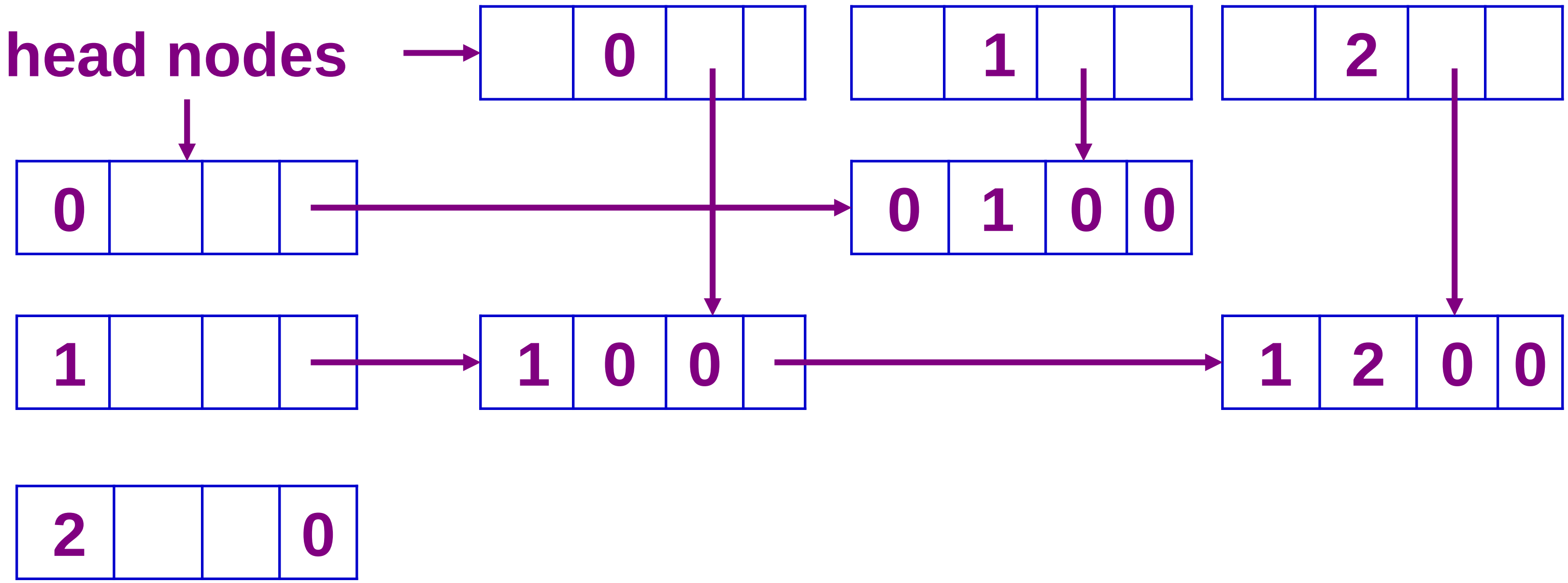
- one list for each vertex
- the nodes in list i represent the vertices adjacent to vertex i.



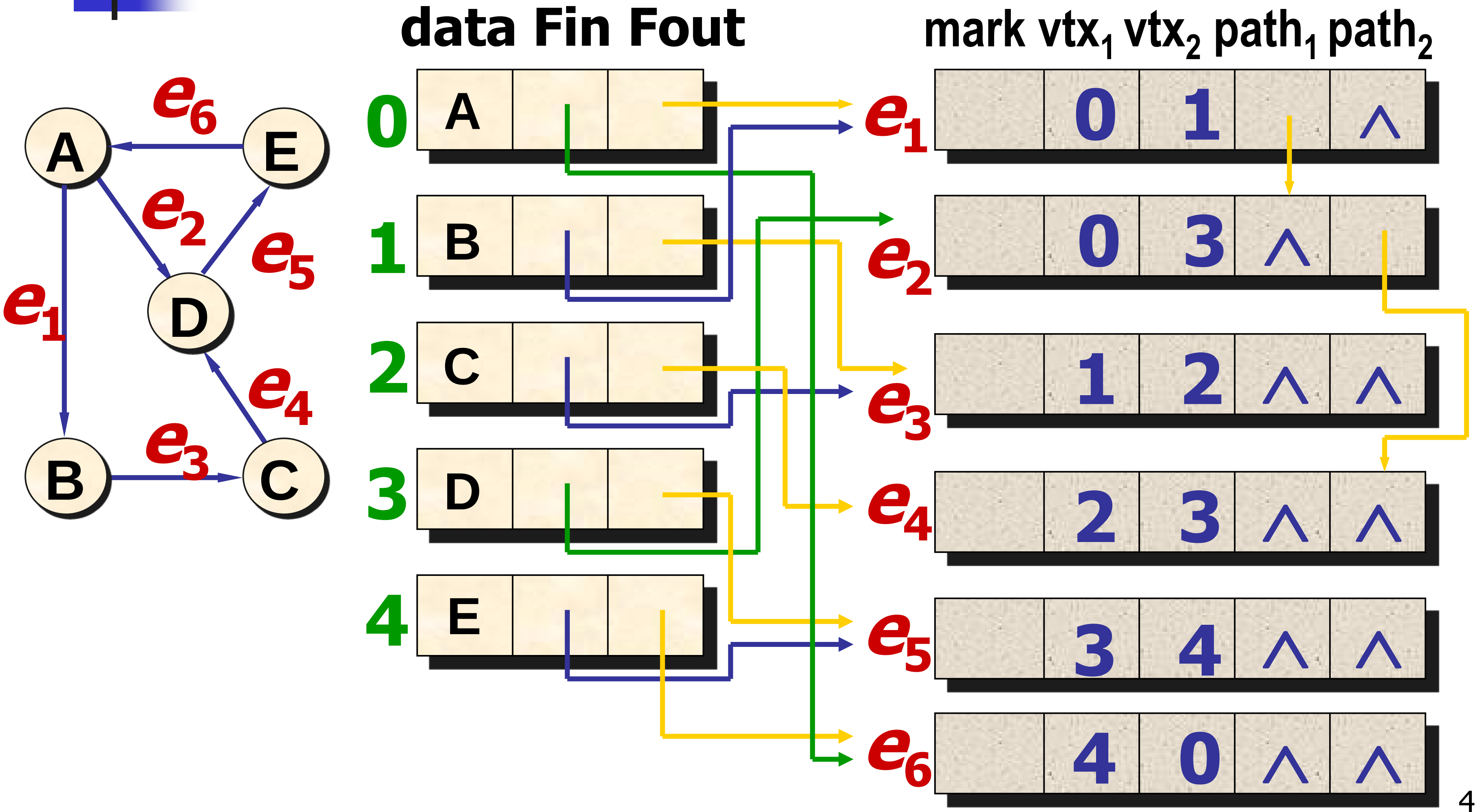


Combining nodes for adjacency lists and inverse adjacency lists together forms an orthogonal list node:

tail	**head**	**column link**	**row link**



head nodes



	data	Fin	Fout
0	A		
1	B		
2	C		
3	D		
4	E		

	mark	vtx_1	vtx_2	$path_1$	$path_2$
e_1		0	1		∧
e_2		0	3	∧	
e_3		1	2	∧	∧
e_4		2	3	∧	∧
e_5		3	4	∧	∧
e_6		4	0	∧	∧

Weighted Edges

- Edges may have weight
- In the case of adjacency matrix, A[i][j] may keep this information.
- In the case of adjacency lists, we need a weight field in the list node.
- A graph with weighted edges is called a network.

Graph

- Basic Concepts
  - Representation
  - Adjacency Matrics
  - Adjacency Lists
- Traversal Algorithms
  - Depth-first Search
  - Breadth-first Search
- Applications
  - Spanning Tree
  - Shortest Paths
  - Activity Networks

Elementary Graph Operations

Given G = (V, E), and v in V(G), we wish to visit all vertices in G that are reachable from v.

In the following methods, we assume the graphs are undirected, although they work on the directed as well.

Graph Traversal

- 深度优先搜索

  DFS (Depth First Search)

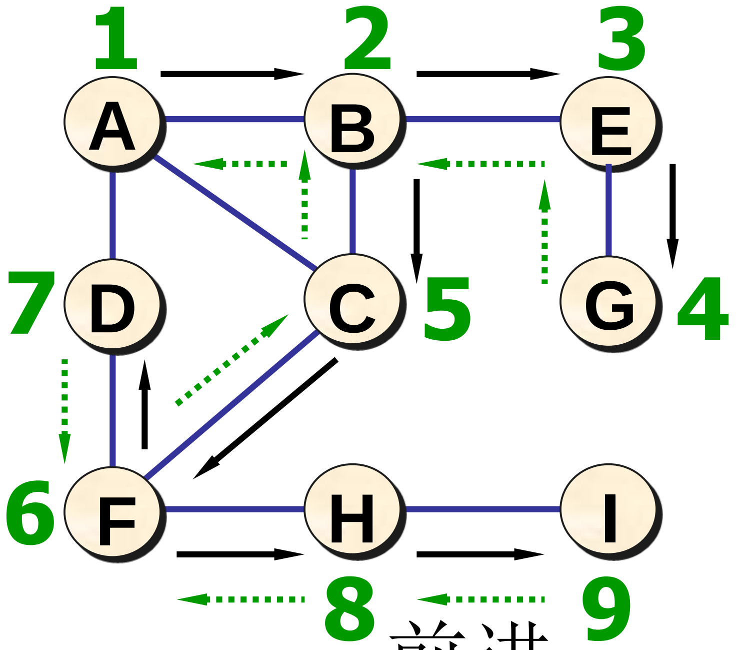
- 广度优先搜索

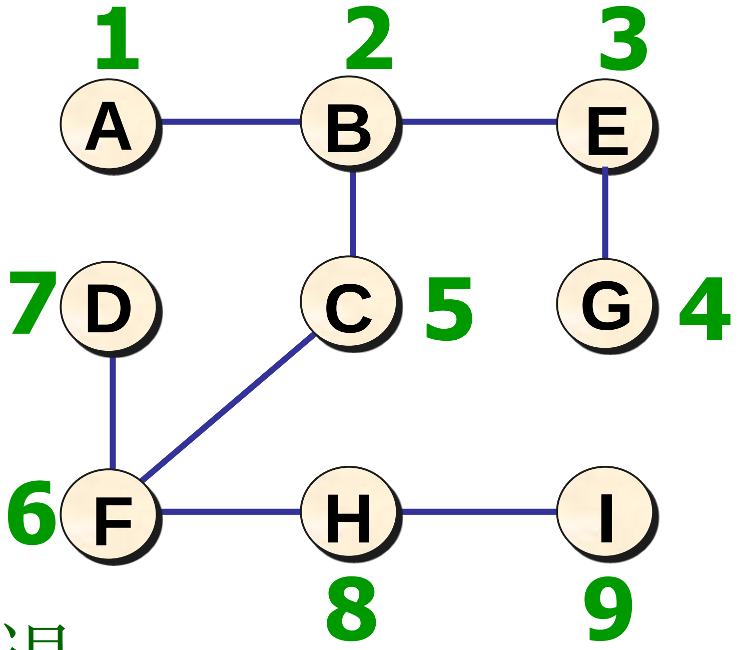
  BFS (Breadth First Search)

Graph

- Basic Concepts
  - Representation
  - Adjacency Matrics
  - Adjacency Lists
- Traversal Algorithms
  - Depth-first Search
  - Breadth-first Search
- Applications
  - Spanning Tree
  - Shortest Paths
  - Activity Networks

DFS





前进 ——→　回退 ⋯⋯→

深度优先搜索过程　　深度优先生成树

- DFS 在访问图中某一起始顶点 v 后,
- 由 v 出发, 访问它的任一邻接顶点 w_1;
- 再从 w_1 出发, 访问与 w_1邻 接但还没有访问过的顶点 w_2;
- 然后， 再从 w_2 出发, 进行类似的访问, …
- 重复, 直至到达所有的邻接顶点都被访问过的顶点 u 为止。
- 接着, 退回一步, 退到前一次刚访问过的顶点, 看是否还有其它没有被访问的邻接顶点。
  - 如果有, 则访问此顶点, 之后再从此顶点出发, 进行与前述类似的访问;
  - 如果没有, 就再退回一步进行搜索。
- 重复上述过程, 直到连通图中所有顶点都被访问过为止。

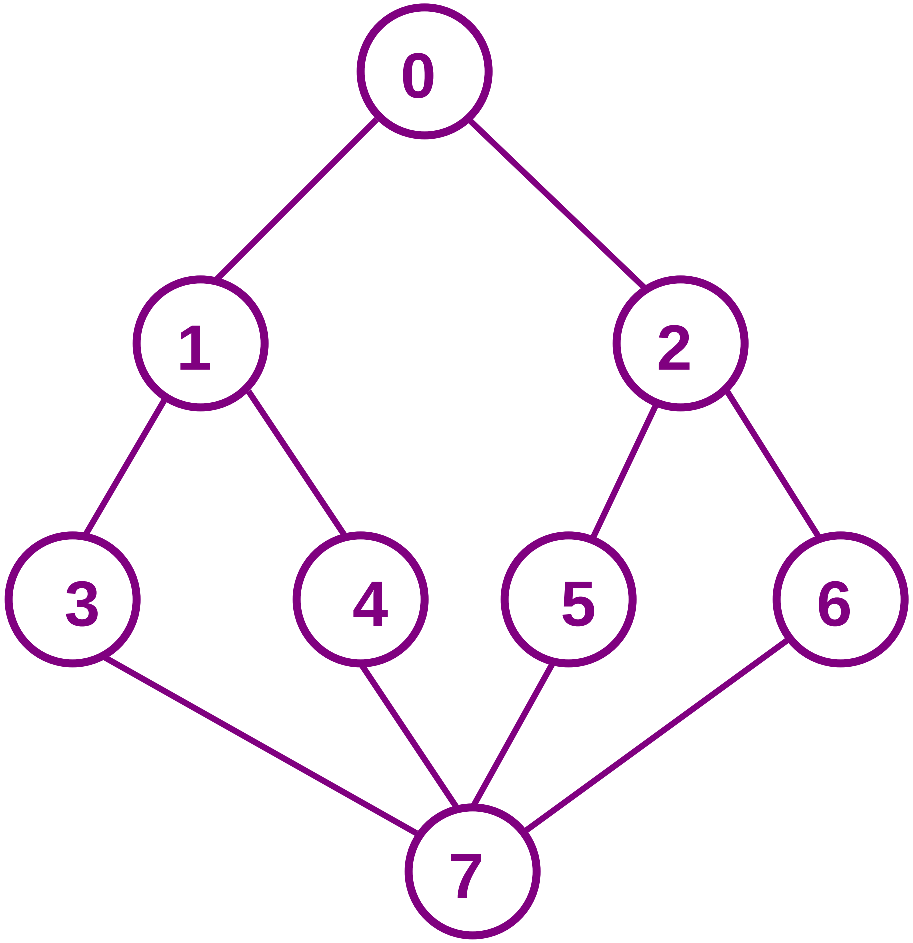
DFS Algorithm

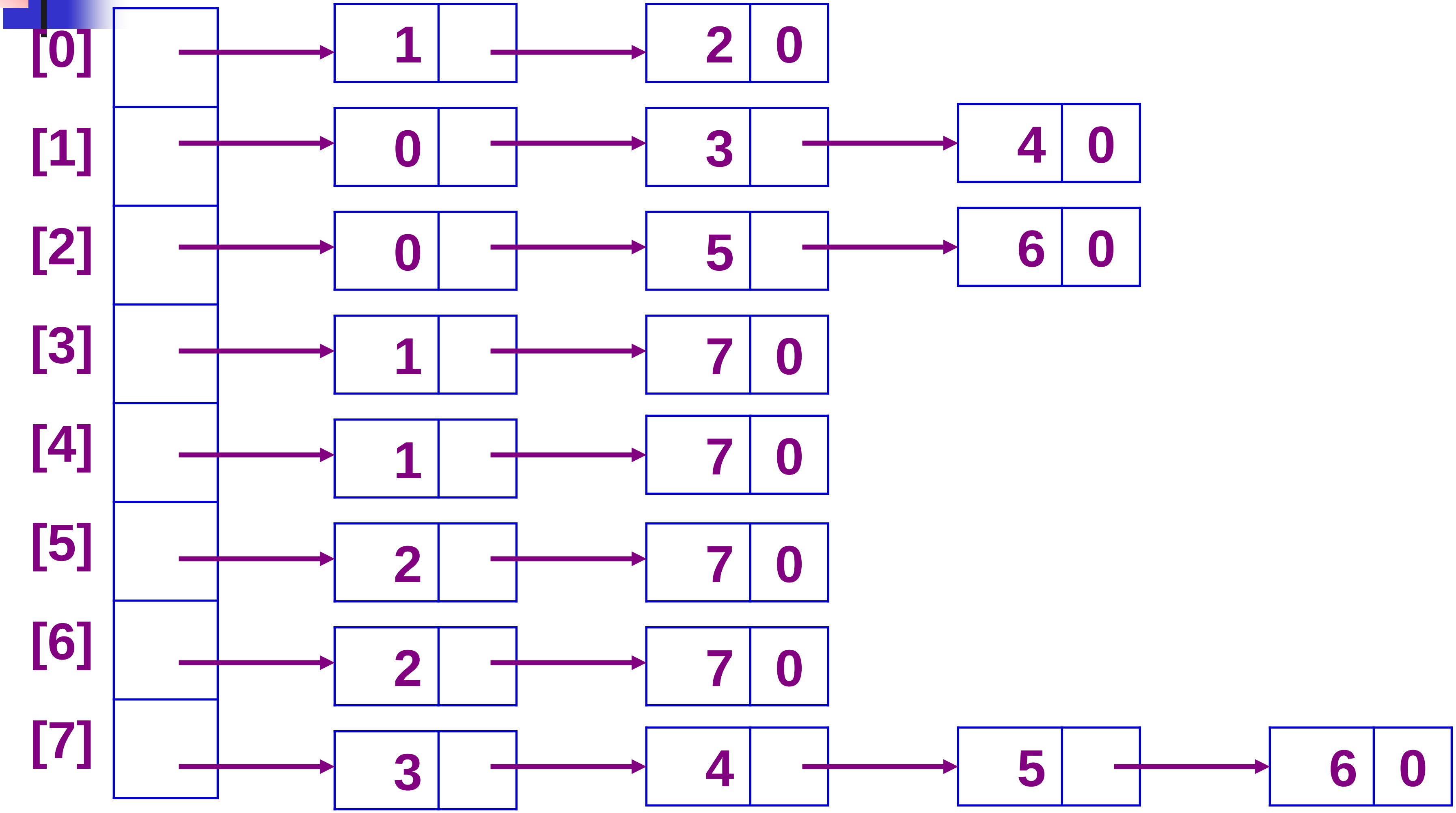
```
template<class T, class E>
void DFS (Graph<T, E>& G, const T& v)
{
//从顶点v出发对图G进行深度优先遍历的主过程
    int i, loc, n = G.NumberOfVertices();    //顶点个数

    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
    }
}
```


0
1
2
3
4
5
6
7



[0] 1 2 0
[1] 0 3 4 0
[2] 0 5 6 0
[3] 1 7 0
[4] 1 7 0
[5] 2 7 0
[6] 2 7 0
[7] 3 4 5 6 0

Started from vertex 0, the vertices in G are visited in the order:

0, 1, (0), 3, (1), 7, (3), 4, (1), (7), (back to 7), 5, 2, (0), (5), 6, (2), (7), (back to 2), (back to 5), (7), (back to 7), (6), (back to 3), (back to 1), (4), (back to 0), (2).

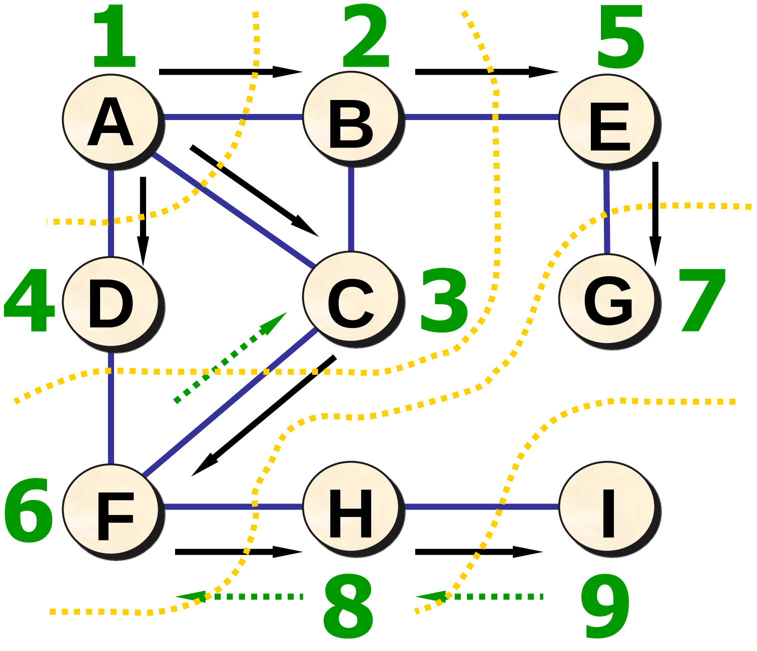
Analysis of DFS

- Initiating visited needs O(n). When G is represented by its adjacency lists, DFS examines each node at most once, so the time is O(e+n).
- If adjacency matrix is used, the time to determine all vertices adjacent to v is O(n), there are n vertices, the total time is O(n^2).

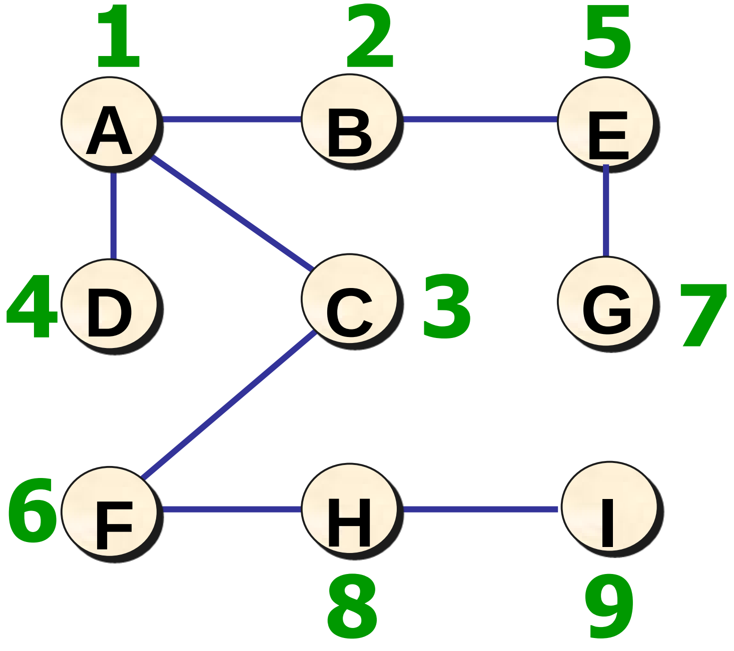
Graph

- Basic Concepts
  - Representation
  - Adjacency Matrics
  - Adjacency Lists
- Traversal Algorithms
  - Depth-first Search
  - Breadth-first Search
- Applications
  - Spanning Tree
  - Shortest Paths
  - Activity Networks

BFS



广度优先搜索过程



广度优先生成树

- BFS在访问了起始顶点 v 之后,
- 由 v 出发, 依次访问 v 的各个未被访问过的邻接顶点 $w_1, w_2, \ldots, w_t$,
- 然后，再顺序访问 $w_1, w_2, \ldots, w_t$ 的所有还未被访问过的邻接顶点。
- 再从这些访问过的顶点出发,
- 再访问它们的所有还未被访问过的邻接顶点，…
- 如此做下去，直到图中所有顶点都被访问到为止。

- 广度优先搜索是一种分层的搜索过程, 每向前走一步可能访问一批顶点, 不像深度优先搜索那样有往回退的情况。
- 因此, 广度优先搜索不是一个递归的过程。
- 为了实现逐层访问, 算法中使用了一个队列, 以记录正在访问的这一层和上一层的顶点, 以便于向下一层访问。
- 为避免重复访问, 需要一个辅助数组 visited [ ]，给被访问过的顶点加标记。

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

- Each visited vertex enters the queue exactly once, the while loop is iterated at most n times.
- If adjacency matrix is used, the loop takes O(n) for each node visited, the total time is $O(n^2)$.
- If adjacency lists are used, the loop has a total cost of d_0+,…,+ d_{n-1} = O(e), the total time is O(n+e).

Graph

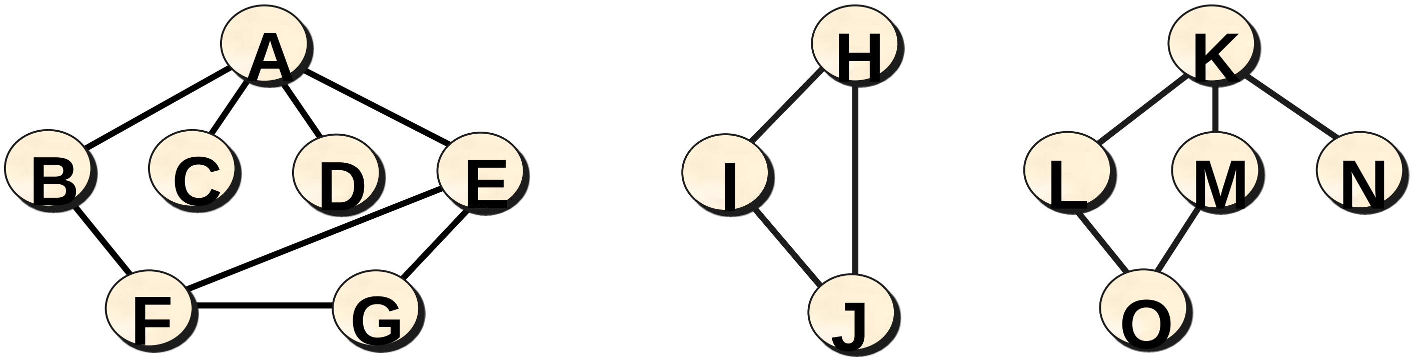
- Basic Concepts
  - Representation
  - Adjacency Matrics
  - Adjacency Lists
- Traversal Algorithms
  - Depth-first Search
  - Breadth-first Search
- Applications
  - Spanning Tree
  - Shortest Paths
  - Activity Networks

Connected Components

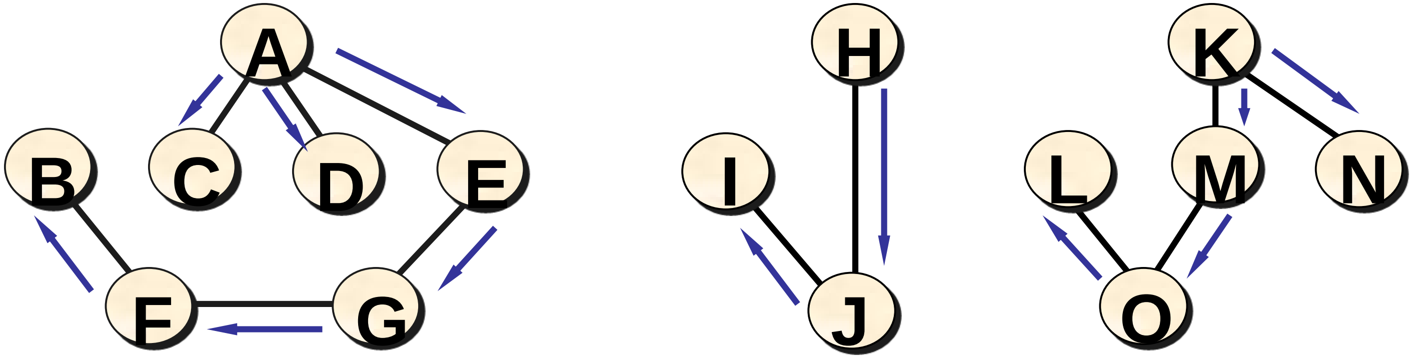
- To obtain all the connected components of an undirected graph, we can make repeated calls to either DFS(v) or BFS(v) for unvisited v.
- This leads to function Components.
- Function OutputNewComponent output all vertices visited in the most recent invocation of DFS, together with all edges incident on them.

连通分量

- 当无向图为非连通图时，从图中某一顶点出发，利用深度优先搜索算法或广度优先搜索算法不可能遍历到图中的所有顶点，只能访问到该顶点所在最大连通子图（连通分量）的所有顶点。
- 若从无向图每一连通分量中的一个顶点出发进行遍历， 可求得无向图的所有连通分量。
- 例如，对于非连通的无向图，所有连通分量的生成树组成了非连通图的生成森林。



非连通无向图



非连通图的连通分量

```
virtual void Graph::Components()
{ // Determine the connected components of the graph.
    visited = new bool[n];
    fill(visited, visited+n, false);
    for (int i=0; i<n; i++)
       if (!visited[i]) {
           DFS (i); // find a component
           OutputNewComponent();
       }
    delete [ ] visited;
}
```

Analysis of Components

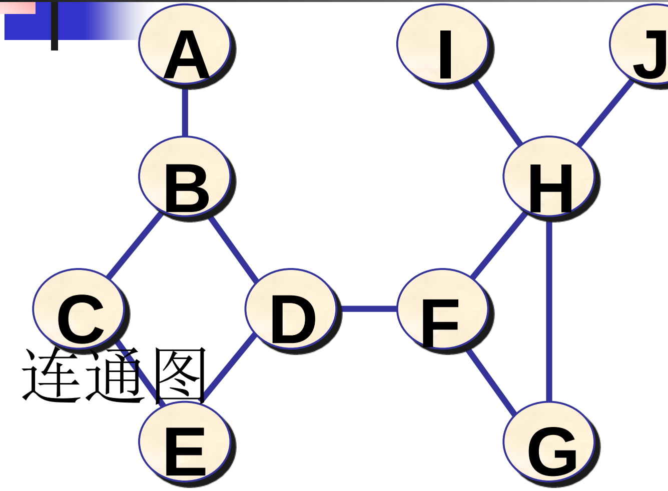
If adjacency lists are used, the total time taken by DFS is O(e). The output can be completed in O(e) if DFS keeps a list of all newly visited vertices. The for loops take o(n). The total time is O(e+n).

If adjacency matrix is used, the total time is $O(n^2)$.

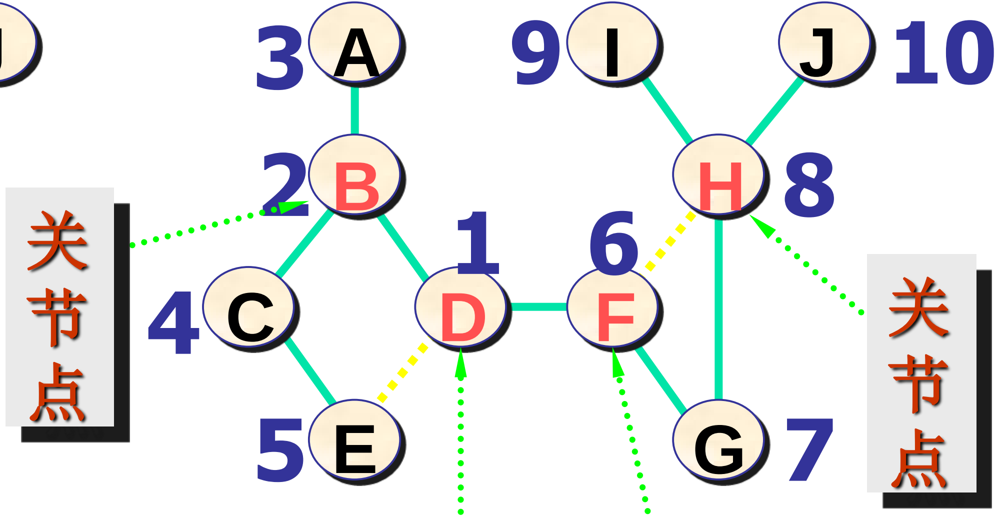
重连通分量

- 在无向连通图G中, 当且仅当删去G中的顶点*v*及所有依附于*v*的所有边后, 可将图分割成两个或两个以上的连通分量，则称顶点*v*为关节点。
- *没有关节点*的连通图叫做重连通图。
- 在重连通图上, 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, 也不破坏图的连通性。

深度优先生成树



连通图

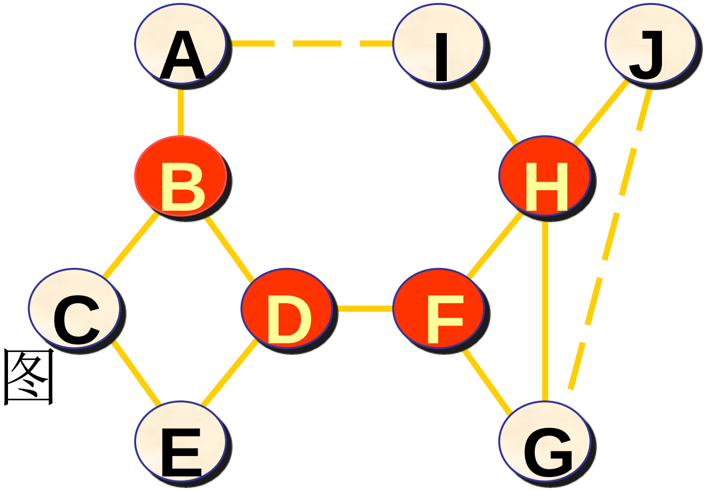


关节点

关节点

关节点

根有两个子女



重连通图

Spanning Trees

If G is connected, in DFS or BFS, all vertices are visited, the edges of G are partitioned into 2 sets:

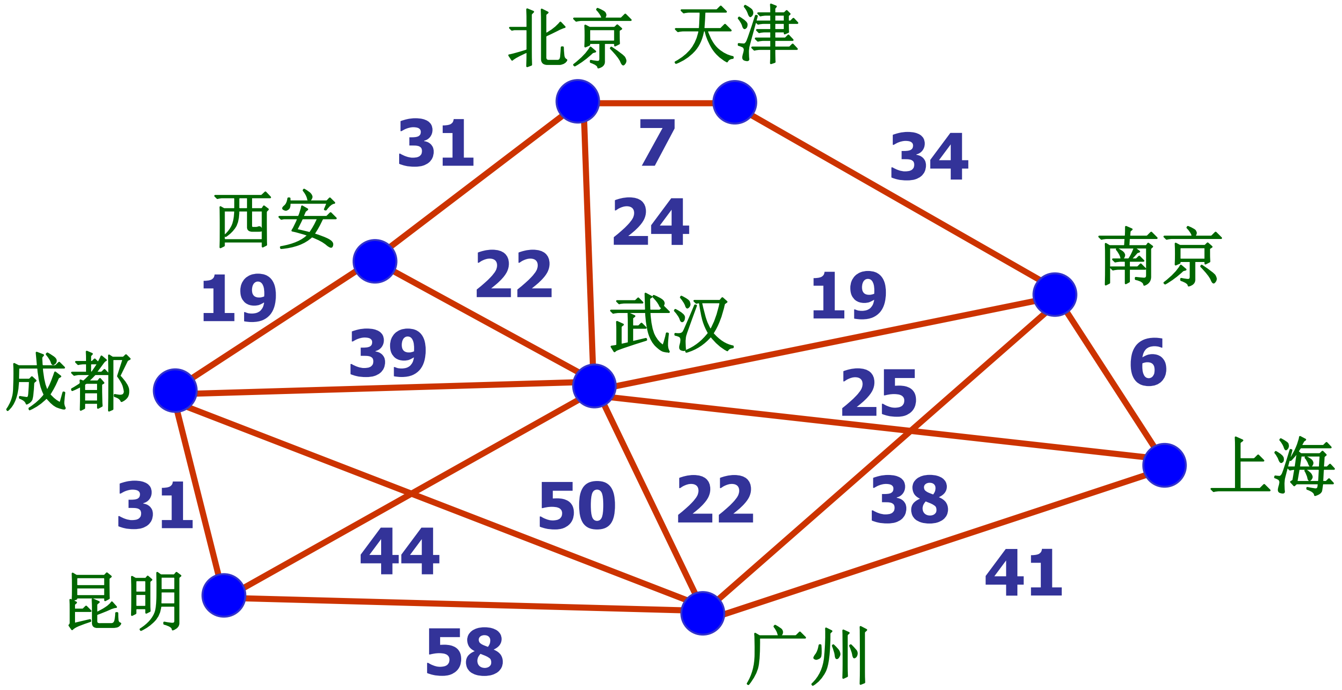
- T --- Tree edges
- N --- Nontree edges

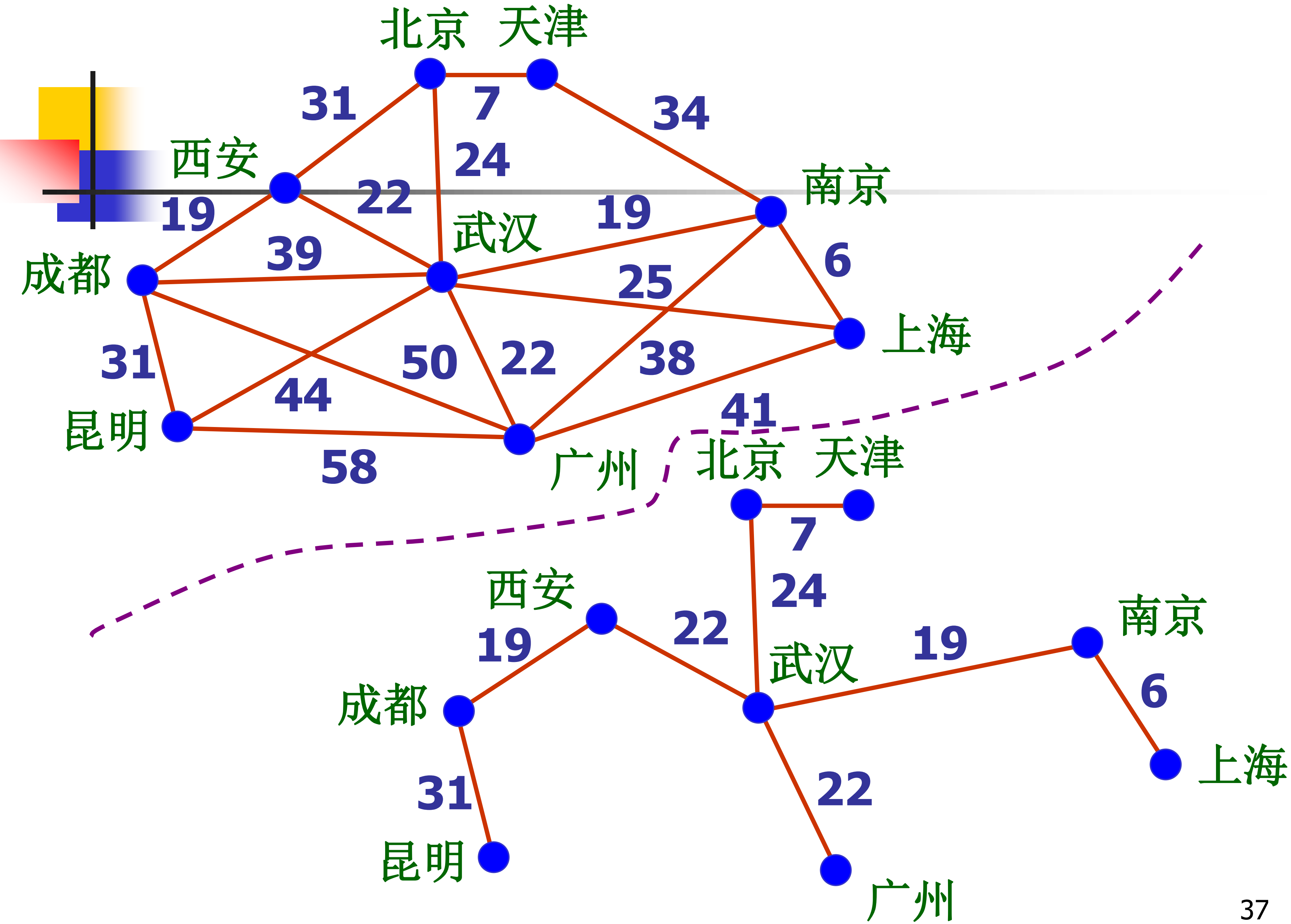
T may be obtained by inserting "$T = T \cup \{(v, w)\}$" using DFS or BFS.

- Any tree consisting solely of edges in G and including all vertices in G is called a spanning tree.
- Depth-first spanning tree --- the spanning tree resulting from a depth-first search
- Breadth-first spanning tree --- the spanning tree resulting from a breadth-first search

Minimum-cost Spanning Tree

- 使用不同的遍历图的方法，可以得到不同的生成树
- 从不同的顶点出发，也可能得到不同的生成树
- 按照生成树的定义，n 个顶点的连通网络的生成树有 n 个顶点、n-1条边。
- 构造最小生成树
  - 假设有一个网络，用以表示 *n* 个城市之间架设通信线路，边上的权值代表架设通信线路的成本。
  - 如何架设才能使线路架设的成本达到最小？


北京
天津
31
7
34
24
西安
22
南京
19
19
武汉
39
成都
6
25
上海
31
50
22
38
44
41
昆明
58
广州


北京
天津
31
7
34
西安
24
22
南京
19
武汉
19
成都
39
25
6
上海
31
50
22
38
44
昆明
41
58
广州
北京
天津
7
24
西安
22
19
南京
武汉
19
成都
6
上海
31
22
昆明
广州

If a nontree edge (v, w) is put into any spanning tree T, then a cycle is formed.

Minimum-Cost Spanning Trees

The cost of a spanning tree of a weighted, undirected graph is the sum of the costs (weights) of the edges in the spanning tree.

A minimum-cost spanning tree is a spanning tree of least cost.

- ✓ A design strategy called greedy method can be used to obtain a minimum-cost spanning tree.
- ✓ In the greedy method,
- construct an optimal solution in stages;
- at each stage, make a decision (using some criterion) that appears to be the best (local optimum);
- since the decision can’t be changed later, make sure it will result in a feasible solution, i.e., satisfying the constraints.

- At the end, if the local optimum is equal to the global optimum, then the algorithm is correct; otherwise, it has produced a suboptimal solution.
- Fortunately, in the case of constructing minimum-cost spanning tree, the method is correct.

To construct minimum-cost spanning tree, we use a least-cost criterion and the constraints are:

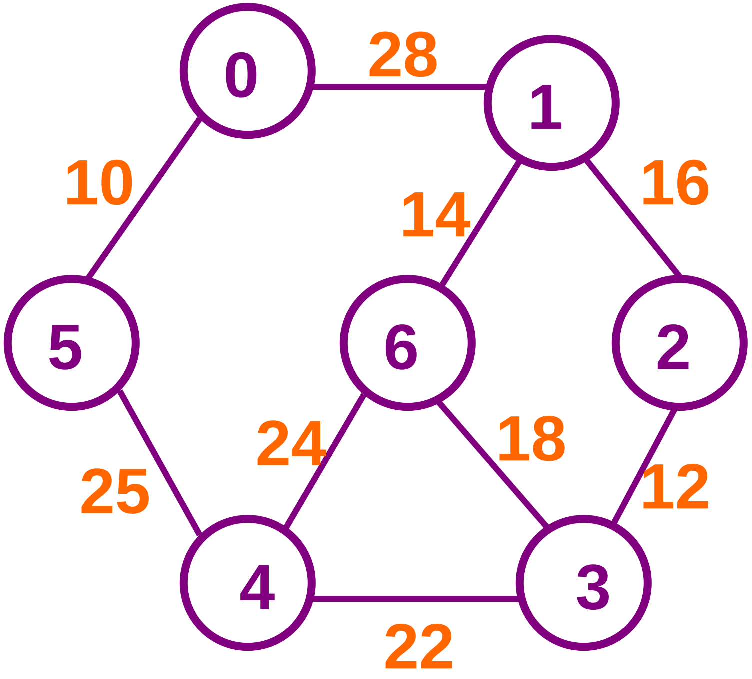
(1) must use only edges within G.

(2) must use exactly n-1 edges.

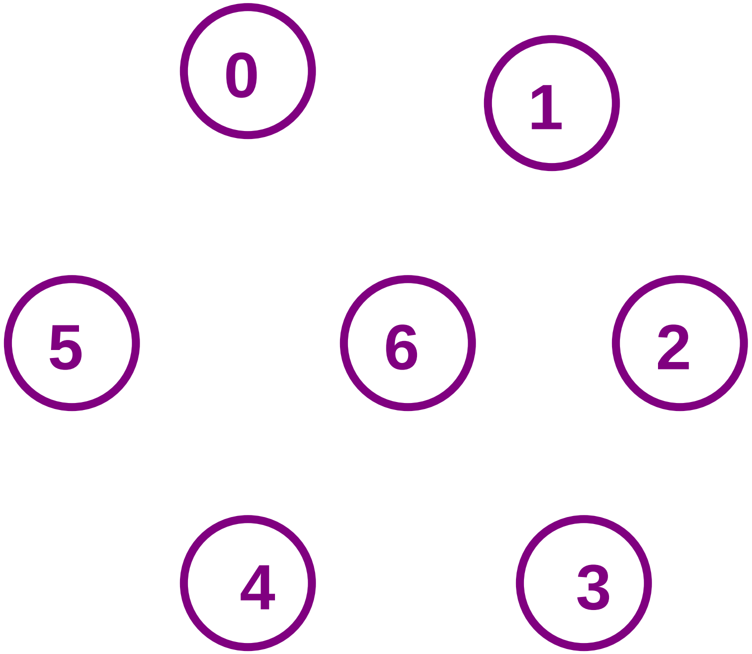
(3) may not use edges that produce a cycle.

Kruskal Algorithm

- The minimum-cost spanning tree T is built by adding edges to T one at a time.
- Edges are selected for inclusion in T in non-decreasing order of their cost.
- An edge is added to T if it does not form a cycle with the edges already in T.
- Exactly n-1 edges will be selected into T.

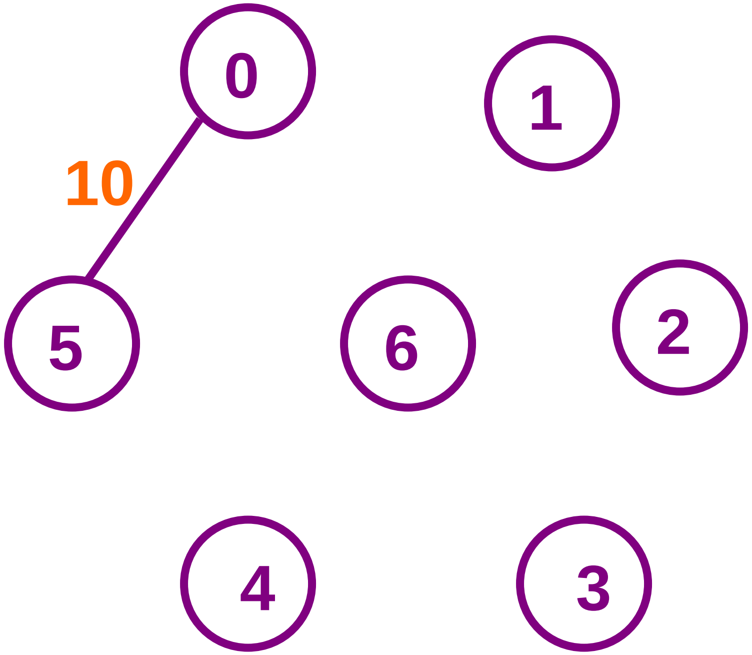


(a) G

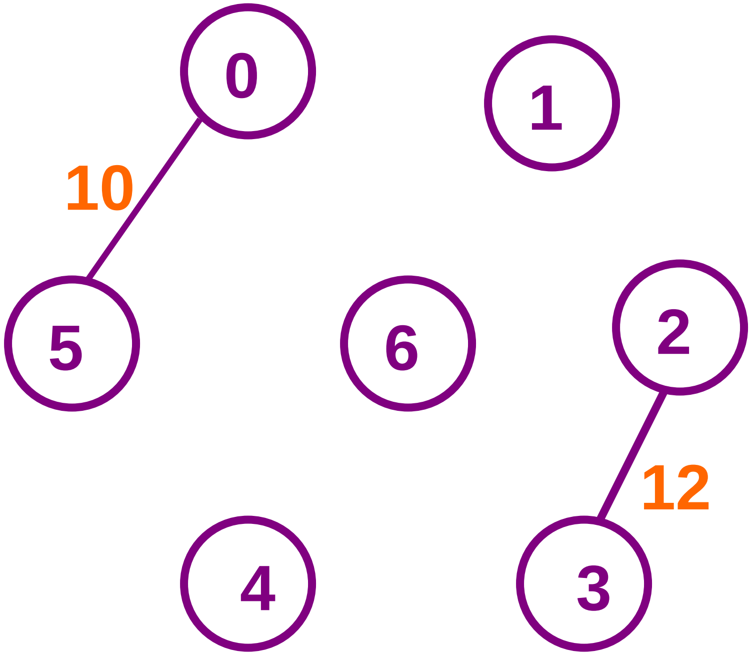


(b)

The cost of remaining edges: 10, 12, 14, 16, 18, 22, 24, 25, 28

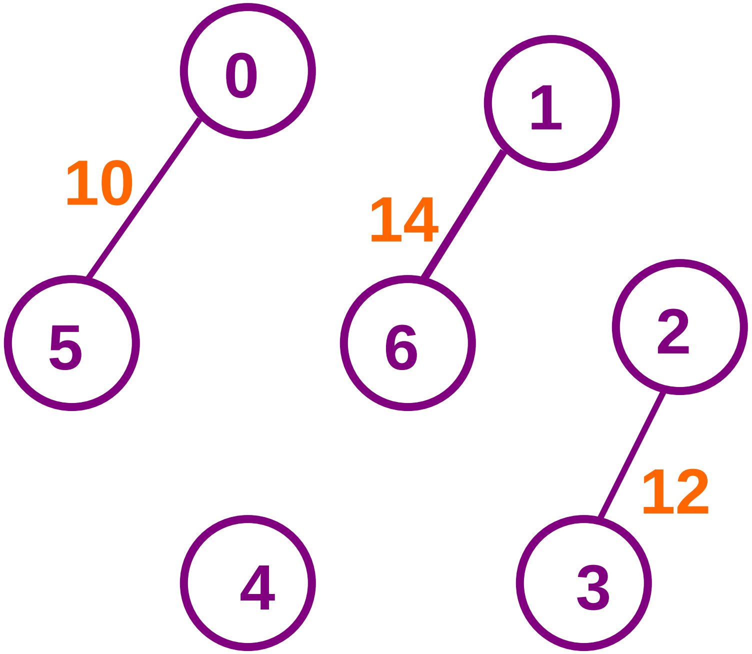


(c)

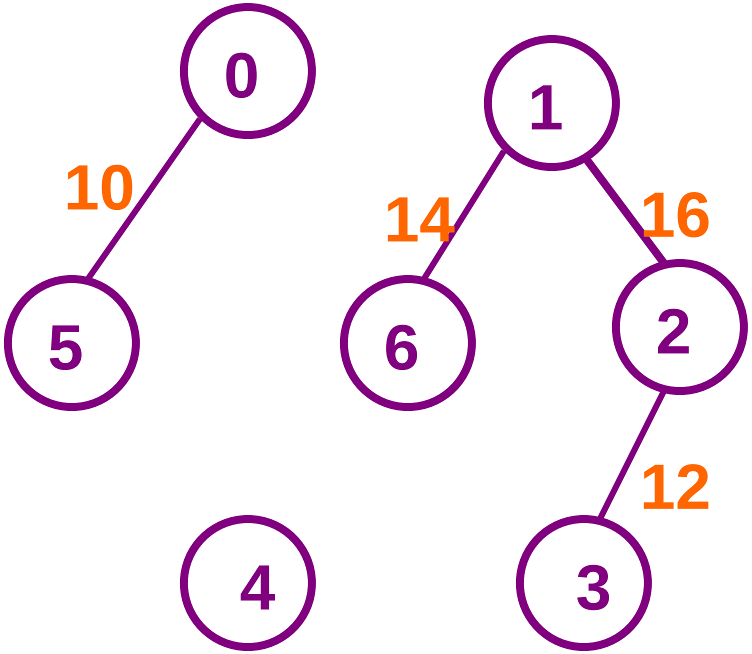


(d)

The cost of remaining edges: 14, 16, 18, 22, 24, 25, 28

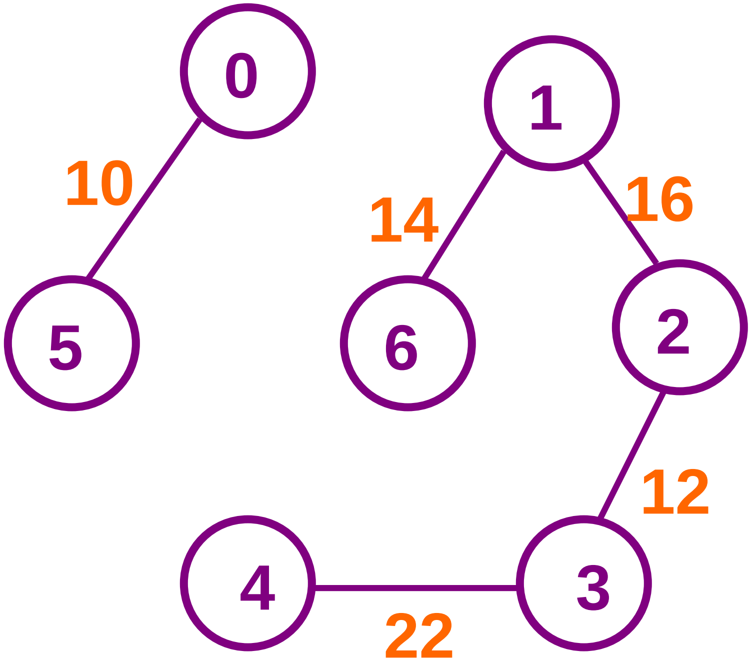


(e)

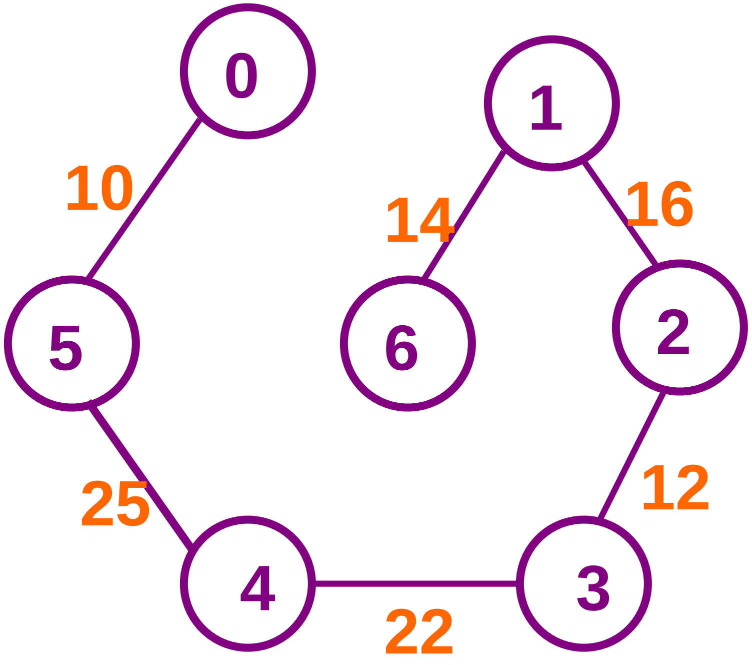


(f)

Remaining edges: 18—(3,6), 22, 24—(4,6), 25, 28



(g) (3,6) discarded



(h) (4, 6) discarded

Given G = (V, E), we have the algorithm:

```
T = ∅;
while ((T contains less than n-1 edges) && (E not Empty)) {
    choose an edge (v, w) from 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;
```

Analysis

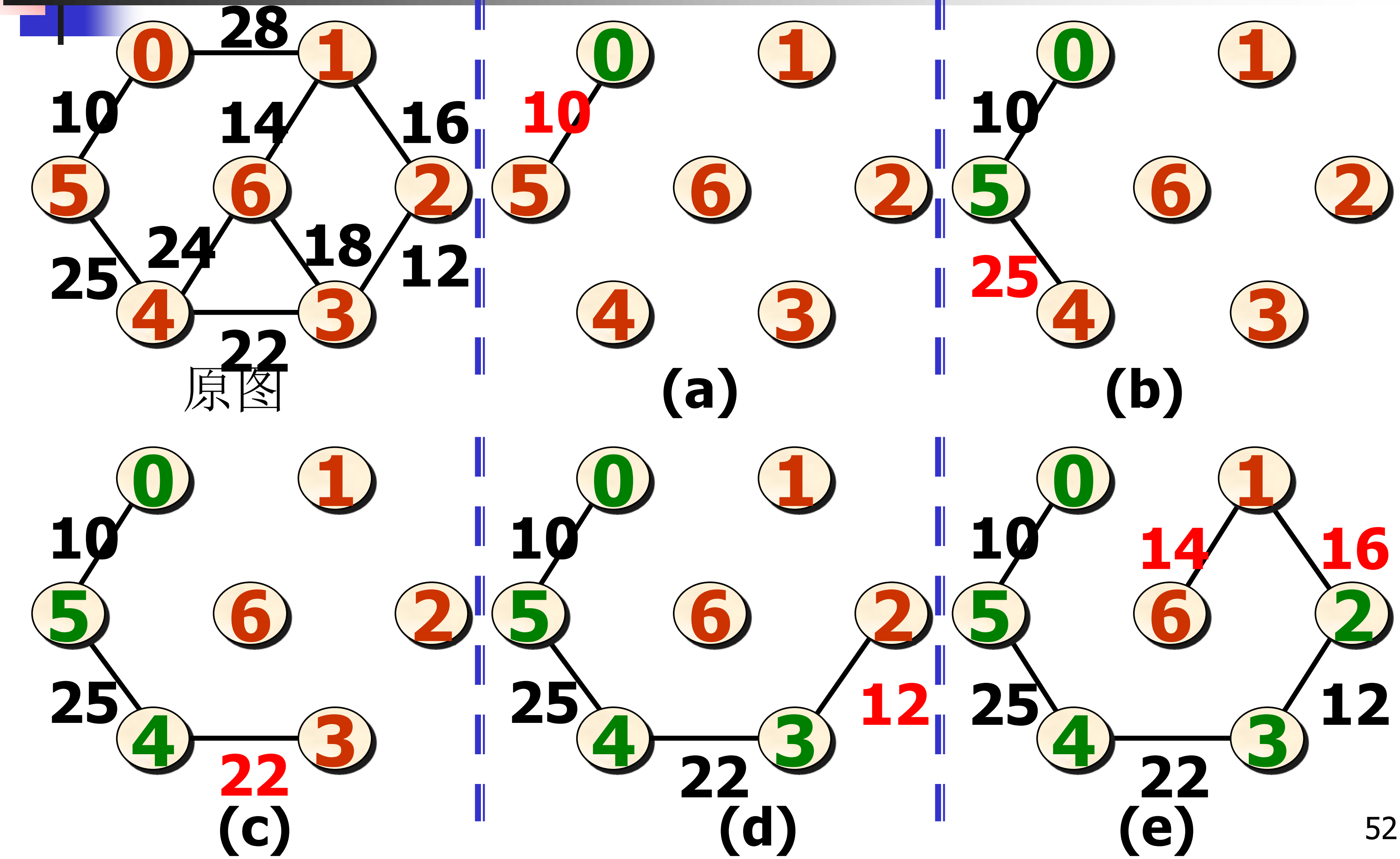
- To perform lines 3 and 4, E can be organized as a min heap, so the next edge can be chosen and deleted in O(log e). Initialization of the heap takes O(e)
- The total cost: O(e log e)

普里姆(Prim)算法

- 从连通网络 $N = \{V, E\}$中的某一顶点 u_0 出发,
- 选择与它关联的具有最小权值的边 (u_0, v), 将其顶点加入到生成树顶点集合U中。
- 每一步从一个顶点在集合U中。
- 另一个顶点不在集合U中的各条边中，选择权值最小的边(u, v), 把它的顶点加入到集合U中。
- 如此继续下去, 直到网络中的所有顶点都加入到生成树顶点集合U中为止。

普里姆(Prim)的伪代码描述

```
选定构造最小生成树的出发顶点$u_0$;
$V_{mst} = \{u_0\}$，$E_{mst} = \varnothing$;
while ($V_{mst}$包含的顶点少于$n$ && $E$不空)
{
      从$E$中选一条边$(u, v)$,
        $u \in V_{mst} \cap v \in V - V_{mst}$, 且具有最小代价(cost);
令$V_{mst} = V_{mst} \cup \{v\}$，$E_{mst} = E_{mst} \cup \{(u, v)\}$;
      将新选出的边从$E$中剔除：$E = E - \{(u, v)\}$;
}
if ($V_{mst}$包含的顶点少于$n$)
   cout << "不是最小生成树" << endl;
```


0
1
2
3
4
5
6
28
10
14
16
25
24
18
12
22
原图
(a)
(b)
(c)
(d)
(e)