Chapter 11 Graph

Graph Applications

- Modeling computer networks
- Representing maps
- Finding paths from start to goal (AI)
- Ordering tasks
- Modeling relationships (families, organizations)

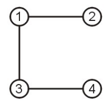
Content

11.1 Terminology and Representations

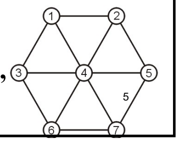
Terminology (1)

A graph G = (V, E) consists of a set of vertices(顶点) V, and a set of edges(边) 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|.



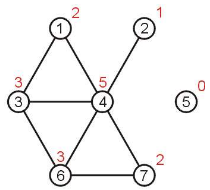
Example 1: given G_1 = (V, E), V = {*1, 2, 3, 4*}, E = {{*1, 2*}, {*1, 3*}, {*3, 4*}} |V| = 4, |E| = 3



Example 2 : given G_2 = (V, E), V = {1, 2, 3, 4, 5, 6, 7}, E = {{1, 2}, {1, 3}, {1, 4}, {2, 4}, {2, 5}, {3, 4}, {3, 6}, {4, 5}, {4, 6}, {4, 7}, {5, 7}, {6, 7}}, |V| = , |E| =

Terminology (2)

- **Adjacent(邻接):** two vertices are said to be ***adjacent*** if there exists an edge between the two vertices
  - for example: if there has a edge $\{a, b\}$ in E, then a is adjacent to b, and b is adjacent to a
  - We will assume that a vertex is not adjacent to itself, that is, each edge in E is made up of two distinct vertices
- **Degree(度):** the degree of a vertex is the number of its adjacent vertices

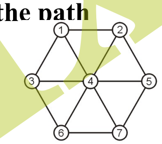


6

6

Terminology (3)

- **Path:** an ordered sequence of vertices $(v_0, v_1, v_2, ..., v_k)$ is called a path, where $\{v_{i-1}, v_i\}$ is an edge in E for $i = 1, ..., k$
- **Length of path**: the number of edges in the path



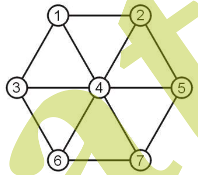
  (1, 2, 4, 3, 6, 7, 5) : 6
  (1, 4, 2, 4, 3, 4, 5, 4, 6, 4, 7): 10
  (2, 4, 1, 2, 4, 2, 1): 6
  (2, 4, 1, 2): 3
  (1): 0
- ***Simple path:*** vertices no repetitions except perhaps the first and last vertices
- ***Cycle(环):*** *a simple path that* the first and last vertices are equal

7

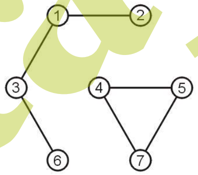
7

Terminology (4)

- **Connectedness (连通)**
  - two vertices v_i, v_j are said to be ***connected*** if there exists a path from v_i to v_j
  - A graph is connected if any two vertices are *connected.*



Connected



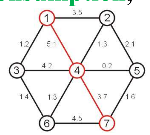
Unconnected

8

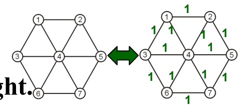
8

Terminology (5)----Weighted Graphs

- A **weight(权值)** may be **associated** with each **edge** in a graph, which could represent distance, energy consumption, cost, etc.



- **weighted graph:** each edge has a weight.
  - The *length* **of a path** within weighted graph is the sum of the weight in the path. (1,4,7), 8.8
- There may be **multiple paths** between two vertices, each with a different weighted length
  - (1,4,5,7), 6.9
- **Shortest path:** the path with the **shortest length** between two vertices.
  - (1, 3, 6, 4, 5, 7), 5.7



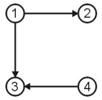
- **unweighted graph:** edge no associated weight.
  - can be regarded to be a weighted graph with all edges have weight 1

9

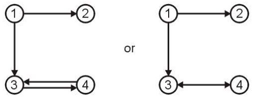
9

Terminology (6)----Directed Graphs/有向图

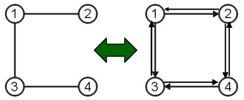
- The edges on a graph may be associated with a direction
  - all edges are ordered pairs (v_i, v_j), where this does denote a connection from v_i to v_j , does not a connection from v_j to v_i
- Such a graph is termed a *directed graph*
  - For example,
    V = {1, 2, 3, 4},
    E = {(1, 2), (1, 3), (4, 3)}



- Adjacent(邻接) in a directed graph :
  - A vertex v_i is adjacent to v_j if (v_i, v_j) is in E
  - For two vertices to be adjacent to each other, both pair must be in E.
  - For example
    V = {1, 2, 3, 4},
    E = {(1, 2), (1, 3), (3, 4), (4, 3)}



- undirected graphs
  - can be considered to be directed graphs with edges in both directions

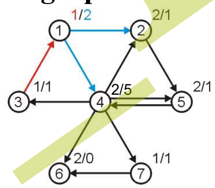


10

10

Terminology (7)----Directed Graphs

- for a vertices in a Directed Graphs
  - *out-degree*: is the number of vertices which are adjacent to the given vertex number of arrows go out
  - *in-degree*: is the number of vertices which this vertex is adjacent to, number of arrows coming in
- For example, the in/out degrees of each of the vertices in this graph are listed next to the vertex

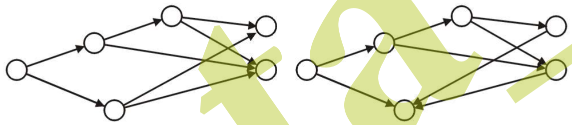


11

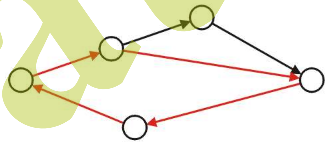
11

Terminology (8)----Directed Acyclic Graphs

Directed Acyclic Graph(有向无环图DAG): a directed graph which has no cycles



Two valid DAG



not a valid DAG

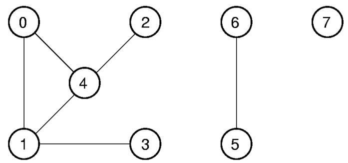
12

12

Terminology (9)---- Connected Components

Subgraph: $G_s=(V_s,E_s)$, if $V_s \subseteq V$, $E_s \subseteq E$, we say G_s is a subgraph of G=（V，E）

Connected components(连通分量): the maximally connected subgraphs of an undirected graph.



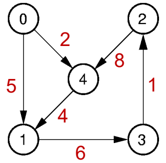
包含3个连通分量的图

13

13

Graph Representation（1）

---- Adjacency Matrix/邻接矩阵

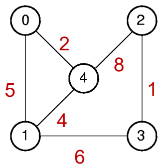


	0	1	2	3	4
0	0	5	0	0	2
1	0	0	0	6	0
2	0	0	0	0	8
3	0	0	1	0	0
4	0	4	0	0	0

space cost:

O(|V|²)

请思考： 若graph用邻接矩阵描述，如何获取各顶点的出/入度呢？

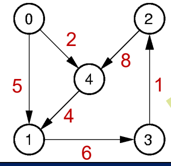


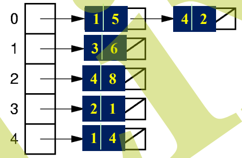
	0	1	2	3	4
0	0	5	0	0	2
1	5	0	0	6	4
2	0	0	0	1	8
3	0	6	1	0	0
4	2	4	8	0	0

非加权图的邻接矩阵呢？

Graph Representation

----Adjacency List/ /邻接链表

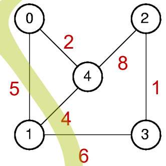


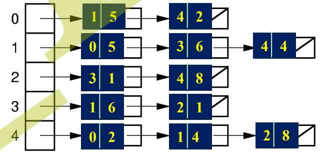


space cost:

O(|V|+|E|)
= O(|V|²)

再思考： 若graph用邻接链表描述，如何获取各顶点的出/入度呢？





非加权图的邻接链表呢？

11.2 Graph implementations

- Graph ADT Class
- Adjacency Matrix implementation
- Adjacency List implementation

Graph ADT

- **Data:** 顶点集
- **Relation:** 边集
- **Basic Operation:**
  - 赋值类： setEdge(i,j,w)
  - 获得信息类： n( ), e( ), first(i)， next (i,j), weight(i,j)
  - 其他： delEdge(i,j)

Graph ADT Class

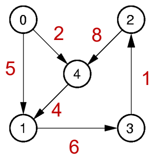
图的基本操作

```
class Graph { // Graph abstract class
public:
  virtual int n() =0; // get number of vertices
  virtual int e() =0;  // get number  of edges
  virtual int first(int v) =0;  // Return index of first neighbor of vth vertex
  virtual int next(int v, int w) =0;  // Return index of next neighbor of vth vertex
  virtual void setEdge(int v, int w, int) =0;   // Set new edge between vth and wth
     vertices
  virtual void delEdge(int v, int w) =0;   // Delete edge connecting vth and wth  vertice
  virtual int weight(int v, int w) =0;  // return weight of edge connecting vth and wth
     vertices
  virtual int  getMark(int v) =0;    //Get the mark value of the vth vertex
  void setMark(int v, int) =0;    //Set the mark value of the vth vertexa
};
```

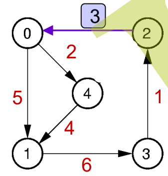
18

18

Garph Basic Operation



给出对右图做下列操作后i的值及
图的变化（若有的话）

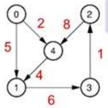


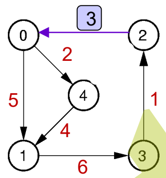

```
i = n();          5
i = e();          6
i = first(0);     1
i = next(0, 1);   4
i = next(1, 3);   5
setEdge(2,0,3);
delEdge(2,4);
i = weight(4, 2); 0
```

19

19

Adjacency Matrix implementation(1)





int **matrix;

	0	1	2	3	4
0	0	5	0	0	2
1	0	0	0	6	0
2	3	0	0	0	0
3	0	6	0	0	0
4	2	0	0	0	0

```
i = n();          5
i = e();          6
i = first(0);     1
i = next(0, 1);   4
i = next(1, 3);   5
setEdge(2,0,3);
delEdge(2,4);
i = weight(4, 2); 0
```

① 1个2D数组描述图的邻接矩阵

② 1个整形变量numV保存顶点个数

③ 1个整形变量numE保存边的条数

20

20

Adjacency Matrix implementation(2)

```
Class GraphM: public Graph {    //implement adjacency matrix
  int  numVertex, numEdge;     // number of vertuces and edges
  int  **matrix;
  int  *mark;     //pointer to a mark array
public:
  GraphM(int n){
     int i,j;
     numVertex = n;
     numEdge = 0;
     mark = new int[numVertex];
     for (i=0;i<n;i++)  mark[i]=0;
     matrix = (int **) new  int * [numVertex];
     for(i=0;i<n;i++)    matrix[i]= new int[numVertex];
     for(i=0;i<n;i++)
          for(j=0;j<n;j++)   matrix[i][j]= 0;
  }
```

int **matrix;

0	0	0	0	0
0	0	0	0	0
0	0	0	0	0
0	0	0	0	0
0	0	0	0	0

int *mark;

0	0	0	0	0

21

21

Adjacency Matrix implementation(3)

```
~GraphM ( ){
   delete []mark;
   for(i=0;i<n;i++)    delete []matrix[i];
   delete []matrix;
}
int n()  { return numVertex; }
int e()  { return numEdge;  }
//int setN(int n)  { numVertex = n ; }
int first(int v) {
   int i;
   for( i=0; (i < numVertex) && (matrix(v,i) ==0); i++) ;
   return i;
}
int next(int v, int w) {
    int i;
    for( i =w+1;(i<numVertex) && (matrix(v,i) ==0); i++) ;
    return i;
}
```

思考：若没找到，返回值为？

22

22

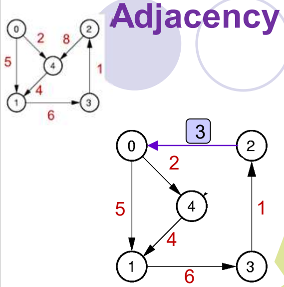
Adjacency Matrix implementation(4)

```
void setEdge(int v, int w, int wgt) {
    Assert(wgt > 0, "Illegal weight value");
    if (matrix[v][w]==0)   numEdge++;
    matrix[v][w]==wgt;
 }
void delEdge(int v, int w) {
    if (matrix[v][w]!=0)    {numEdge --; matrix[v][w]=0;}
}
 int weight(int v, int w)   {  return matrix[v][w];  }
 int getMark(int v)     {  return mark[v]; }
 void setMark (int v, int val)   {  mark[v]=val;  }
};
```

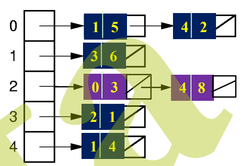
23

23

Adjacency List implementation(1)



LList<Edge>   ** vertex;



```
Class Edge {
Public:
  int vertexIndex
  int weight;
  ……
}
```

① 1个数组（元素为链表）描述图的邻接链表

② 1个整形变量numV保存顶点个数

③ 1个整形变量numE保存边的条数

```
i = first(0);        1
i = next(0, 1);      4
i = next(1, 3);      5
setEdge(2,0,3);
delEdge(2,4);
i = weight(4, 2);    0
```

24

24

Adjacency List implementation(2)

```
Class Edge {
Public:
  int vertexIndex
  int weight;
  Edge( )  { vertexIndex= -1; weight = -1;}
  Edge(int v, int w )  { vertexIndex = v; weight = w;}
  Edge operator = (Edge e1)  {
       vertexIndex=e1.vertexIndex;
       weight=e1.yweight;
       return *this;   }
};

Class GraphL: public Graph {
private:
   int  numVertex, numEdge;     // number of vertices and edges
   LList<Edge>  ** vertex;  // linked list header;
   int  *mark;      //pointer to a mark array
```

25

25

Adjacency List implementation(3)

```
public:
  GraphL(int n){
   int i, j;
   numVertex=n;   numEdge=0;
   mark = new int[numVertex];   for (i=0;i<n;i++)  mark[i]=0;
   vertex = (LList<Edge>**) new LList<Edge> *[numVertex];
   for(i=0;i<n;i++)    vertex[i]= new LList<Edge> [numVertex];
  }

~GraphL( ){
     delete [ ]mark;
     for(i=0;i<n;i++)  vertex[i]->clear;
     delete  [ ]vertex;
}
```

26

26

Adjacency List implementation(4)

```
int n( )  { return numVertex;  }
int e( )  { return numEdgee;   }
// int setN(int n)  { numVertex = n ; }
int first(int v) {     //若没有，返回numVertex
  if (vertex[v] -> length()==0)  return  numVertex;
  vertex[v]->moveToStart();
  return   (vertex[v]->getValue( ) ).vertexIndex;
}
int weight(int v, int w)  {    //若v,w之间没有弧，返回0
   Edge curr;  int  l=vertex[v] -> length();
   curr=vertex[v]->getValue( );
   if (curr.vertexIndex != w)
     for (vertex[v]->moveToStart();vertex[v]->currPos()<l; vertex[v]->next())
       { curr=vertex[v]->getValue() ;  if (curr.vertexIndex>=w)    break; }
   if (curr.vertexIndex == w)     return curr.weight;
   else return 0;    }
```

在线性表 vertex[v]中寻找合适的结点

27

27

Adjacency List implementation(5)

```
int next(int v, int w) {    //若没有，返回numVertex
    Edge curr = vertex[v]->getValue();
    int  l=vertex[v] -> length();
    if(curr.vertexIndex != w)
    {
          vertex[v]->moveToStart( );
          for (; vertex[v]->currPos() < l ; vertex[v]->next())
         {          curr=vertex[v]->getValue();
                     if (curr.vertexIndex >=w)     break;      }
    }
    if ( curr.vertexIndex ==w)
    {    vertex[v].next();    return  (vertex[v]->getValue()).vertexIndex;  }
    else  return  numVertex;
  }
```

在线性表 vertex[v]中寻找合适的结点

28

28

Adjacency List implementation(6)

```
void setEdge(int v, int w, int wgt) {
     Assert(wgt > 0, "Illegal weight value");
     int  l=vertex[v] -> length();
     Edge  it(w,wgt);
     Edge  curr=vertex[v]->getValue();
     if (curr.vertexIndex != w)  {
       vertex[v]->moveToStart( );
       for (; vertex[v]->currPos() < l ;  vertex[v]->next())
          { curr=vertex[v]->getValue();
             if (curr.vertexIndex>=w)    break;    } }
     if (curr.vertexIndex== w)  vertex[v]->remove(curr);
     else   numEdge++;
     vertex[v]->insert(it);
}
```

在线性表 vertex[v]中寻找合适的结点

29

29

Adjacency List implementation(7)

```
void delEdge(int v, int w) {
    Edge curr;
    curr=vertex[v]->getValue( );
    if (curr.vertexIndex != w)  {
        vertex[v]->moveToStart( );
        for (; vertex[v]->currPos() < l ; vertex[v]->next())
        {  curr= vertex[v]->getValue(); if (curr.vertexIndex>=w)    break;
    }
    if (curr.vertexIndex == w)  { vertex[v]->remove( );  numEdge--;  }
}
int getMark(int v){  return mark[v];  }
void setMark(int v, int val){    mark[v] = val;  }
};
```

在线性表 vertex[v]
中寻找合适的结点

30

30

11.3 Graph Traversals

11.3.1 Graph Traversal

11.3.2 Depth First Search

11.3.3 Breadth First Search

11.3.4 **Topological Sort**

31

31

11.3.1 Graph Traversal

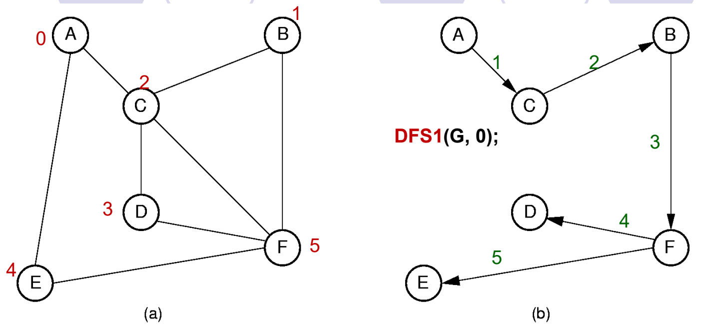
- Some applications require visiting every vertex in the graph exactly once. （无条件）
  - **Depth First Search: DFS**
  - **Breadth-First Search: BFS**
- The application may require that vertices be visited in some special order based on graph topology.(有条件)
  - **Topological Sort**
- Application Examples:
  - **Connected components analysis**
  - **Shortest paths problems**
  - **Artificial Intelligence Search**

To insure visiting all vertices once and only exactly once

32

32

11.3.2 Depth First Search of connected graph (1)



Visit order: A C B F D E

上述**DFS(root-Cs)**类似 与树的前根遍历

33

33

Depth First Search of connected graph(2)

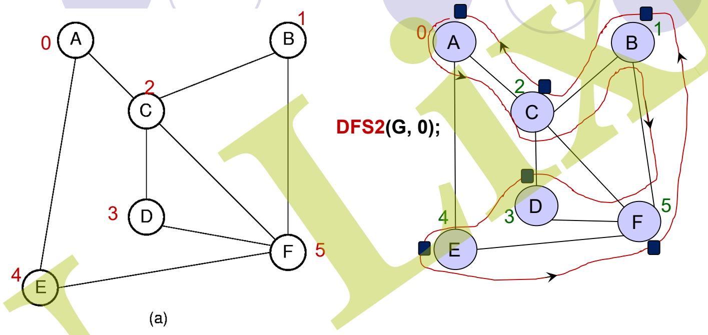
```
// Depth first search---root-Cs
void DFS-RootCs(Graph* G, int v) {
   G->setMark(v, VISITED);
   printf("%d\n", v) ;  // print visited vertex
   for (int w=G->first(v); w<G->n(); w = G->next(v,w))
       if (G->getMark(w) == UNVISITED)
             DFS-RootCs(G, w);
}
```

Cost: $\Theta(|\mathbf{V}| + |\mathbf{E}|)$.

34

34

11.3.2 Depth First Search of connected graph (3)



Cs-root visit order: D E F B C A

上述DFS类似 与树的后根遍历

35

35

Depth First Search of connected graph(4)

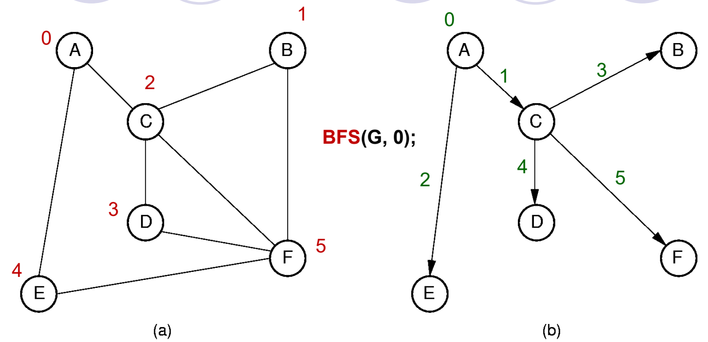
```
// Depth first search--- Cs--root
void DFS-CsRoot(Graph* G, int v) {
   G->setMark(v, VISITED);
   for (int w=G->first(v); w<G->n(); w = G->next(v,w))
       if (G->getMark(w) == UNVISITED)
             DFS-CsRoot(G, w);
   printf("%d\n", v) ;  // print visited vertex
}
```

Cost: $\Theta(|\mathbf{V}| + |\mathbf{E}|)$.

36

36

11.3.3 Breadth-First Search of connected graph (1)



Visit order: A C E B D F

37

37

Breadth-First Search of connected grpah (2)

```
void BFS(Graph* G, int start) {
  int v, w;   LQueue<int>  Q;
 Q.enqueue(start);
 G->setMark(start, VISITED);
  printf("%d\n", start) ;  // print visited vertex
  while (Q.length() != 0) {    // Process Q
      v=Q.dequeue();
      for(w=G->first(v); w<G->n(); w=G->next(v,w))
         if (G->getMark(w) == UNVISITED) {
             Q.enqueue(w);
             G->setMark(w, VISITED);
             printf("%d\n", w) ;  // print visited vertex
         }
  }
}
```

Visit vertex's neighbors before continuing deeper in the graph.

38

38

Connected Graph Traversal conclusion

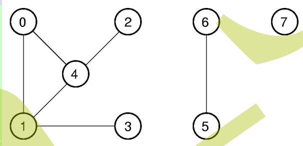
- Depth First Search of graph
  - based on 栈/递归的方式实现
- Breadth-First Search
  - based on 队列的方式实现

39

39

General Graph Traversal (无条件)

```
void graphTraverse(const Graph* G) {
 int v;
 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);
}
```



BFS(G,v);
or DFS-RootCs(G,v);
or DFS-CsRoot(G,v);

若G为无向图。doTraverse的执行次数等于图中连通分量的个数。

40

40

课后思考

- 前述无条件遍历函数（P34，P36,P38，P40）针对的是无向图，若G为有向图，要实现无条件遍历可以直接利用前述这些函数吗？
- 若你认为可以，请分析 doTraverse的执行次数除了与图本身有关，还取决于什么？
- 如果想要doTraverse的执行次数与结点编号顺序无关，你有那些修正方案？

41

41

11.3.4 Topological Sort

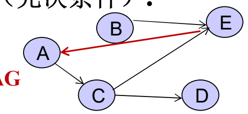
- **Given: a number of tasks, there are often a number of constraints between the tasks:**

  For example: tasks: A,B,C,D,E 非DAG/有环图？

  先决条件

  - ***task A*** **must be completed before tasks C can start**
  - ***tasks C, B*** **must be completed before task E can start**
  - ***tasks C*** **must be completed before task D can start**

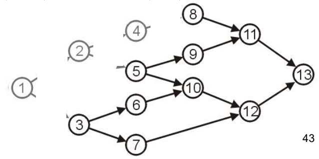
- **Problem: Output the tasks in an order that does not violate any of the prerequisites（先决条件）.**
- **Sulution:**
  - **modeling the problem using A DAG**
  - **Topological Sort the DAG**

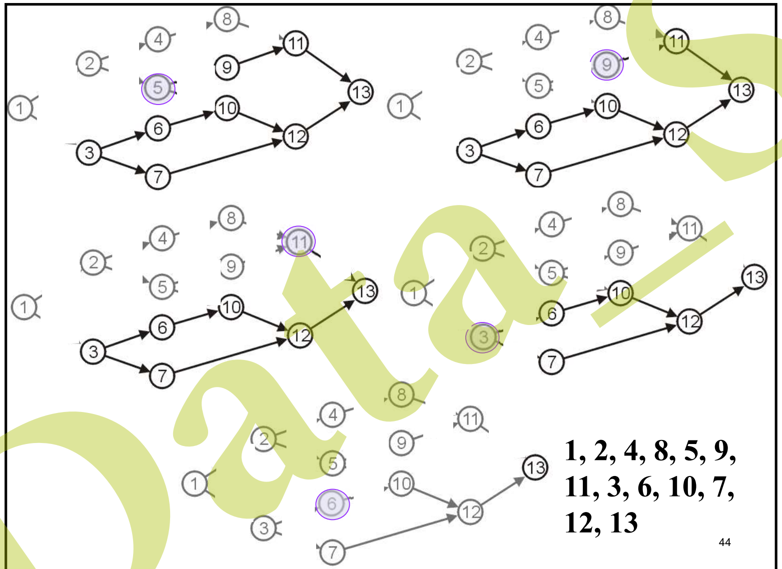


The process of laying out the vertices of a DAG in a linear order to meet the prerequisites rules is called Topological Sort

Topological Sort

- **To generate a topological sort, we must start with a vertex with an in-degree of zero: for example:1**
- **Then, we may ignore/delete those edges which connect vertex 1 to other vertices, and choose a vertex with an in-degree of zero such as 2 or 3: 1, 2**
- **then ignore/delete all edges which extend from 2, and chose a vertex with an in-degree of zero, we may choose from vertices 4, 5, or 3: 1, 2, 4**
- **and then….. 1, 2, 4, 8**



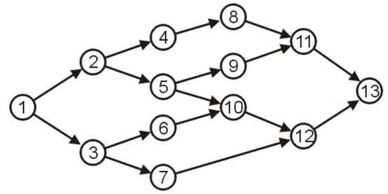


1, 2, 4, 8, 5, 9, 11, 3, 6, 10, 7, 12, 13

Topological Sort

Topological Sort result of DAG isn't unique:

- **one topological sort is 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13**
- **another topological sort is: 1, 2, 4, 8, 5, 9, 11, 3, 6, 10, 7, 12, 13**
- **another topological sort is: 1, 2, 4, 8, 5, 9, 11, 3, 7, 6, 10, 12, 13**

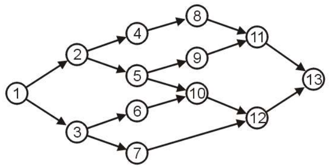


Queue-Based(BFS-based) Topsort Implementation

BFS_based Topological Sort result of DAG is unique

步骤：

1. 建一个空队列Q
2. 将所有入度为0的顶点入Q (按顶点的序号)
3. 从Q中出队一顶点v，按下列步骤处理v
   1） 访问(即输出)v;
   2） 对v的每一邻接顶点，将其入度减1， 若入度变为0则将其入队Q
4. 重复步骤3，直到Q为空



思考：

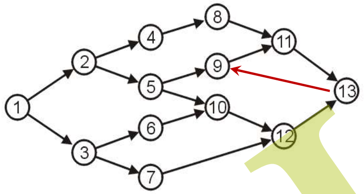
什么情况下会出现直到Q为空，依然有顶点没输出？

46

46

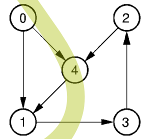
Queue-Based Topsort ---implementation

若为非DAG，会怎样？



	Q	output
初始:		
1入队:	1	
1出队，2,3入队:	2 3	1
2出队，4,5入队:	3 4 5	1, 2
3出队，6,7入队:	4 5 6 7	1, 2, 3
4出队，8入队:	5 6 7 8	1, 2, 3, 4
5出队，9入队:	6 7 8 9	1, 2, 3, 4, 5
6出队，10入队:	7 8 9 10	
7出队:	8 9 10	...
8出队:	9 10	...
9出队，11入队:	10 11	
10出队，12入队:	11 12	
11出队，:	12	
12出队，13入队：	13	
13出队：		1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13

当输入graph为有环图时，会出现直到Q为空，依然有顶点没输出的现象。



47

47

Queue-Based Topsort Implementation (3)

```
void topsort(Graph* G, Queue<int>* Q) {
  int  v, w, *inDegree;      //  Indegree 用来存放每个顶点的入度
  inDegree= new int[G->n()];
  for (v=0; v<G->n(); v++)    inDegree[v] = 0;
  for (v=0; v<G->n(); v++)    // set  inDegree[]  according edges
      for (w=G->first(v); w<G->n();  w = G->next(v,w))
          inDegree[w]++;     // increase  w's indegree/入度
  for (v=0; v<G->n(); v++)       // Initialize Q : 将入度为0的顶点入队Q
     if (inDegree[v] == 0)     Q->enqueue(v);      // 入度为0， No prereqs
  while (Q->length() != 0) {
     v=Q->dequeue();   cout<<v<<endl;      // Process for V such as print
     for (w=G->first(v); w<G->n();  w = G->next(v,w)) {
         inDegree[w]--;    // w入度（prereqs）减1
         if (inDegree[w] == 0)   Q->enqueue(w);   }   //w入度为0，入队
  }
}
```

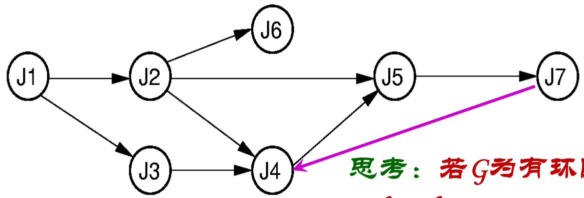
48

48

不建议使用

DFS-based TopSort Implementation

S1 从入度为0的节点开始，进行Cs-root遍历
S2 Reversed order

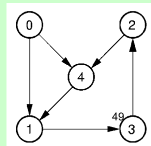


思考：若G为有环图，DFS-based 的TopSort 结果可信吗？

Cs-root result: J7,J5,J4,J6,J2,J3, J1

Reversed, we get the Topological Sort :

J1,J3,J2,J6,J4,J5, J7



49

49

Conclusion of two Topsort methods

- Queue-Based(BFS-based) Topsort(可以测定输入Directed Graph是否为有环图，从而可提醒输入的非DAG不适合Topsort: 当Q为空时，依然有顶点没输出，说明该 directed graph为有环图，即图中有环，不合适Topsort。

- DFS-based TopSort 不管输入directed graph是否为DAG，总会输出所有顶点，无输入不当提醒功能：当输入graph为DAG时，输出结果可信，但当其为非DAG时，结果不可信。

不建议使用

50

50