1.Connectivity information

Def:relationships that exist between pairs of objects.

eg. In city maps:the objects are roads,in the routing tables for the internet:the objects are computers.

连接性问题也存在于二叉树中节点的子父关系中。

Graphs(图)是一种储存，表达并利用这种信息的方式。

2.Graphs

Def:a graph is a set of objects(vertices or nodes) and use edges to connect them.

Applications in:mapping,transportation,electrical engineering and computer networking.

即：图是一组对象(在图中由顶点或节点表示，用边连接)，表示方法G=(V,E),V是顶点的集合，E是来自V的边的集合。

2.1Edge types and graph types

Edge types:

1. Directed edge(有向边)：

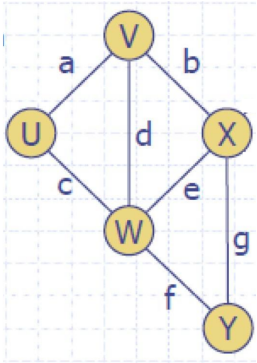
- 一组有序的顶点对(u,v)
- u是起始顶点，v是目的顶点
- 边是从u指向v的有向边 例如一次单程航班：

2.Undirected edge(无向边，与上述提到的相反) 例如一条航线：

由上述定义可以得出不同类型图的定义：directed graph以及undirected graph： 前者所有边都是有向边，后者所有边都是无向边。

2.2Terminology(相关术语)

所涉及的相关术语的定义：



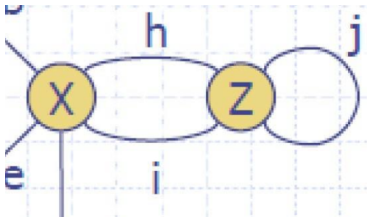
边的**End vertices(endpoints)**:连接一条边的两个顶点称为这条边的end vertices，如图中U和V是a的endpoints。

Edges **incident** on a vertex：与V incident的边有a，b和d。

Adjacent vertices：连接一条边的两个顶点，例如u和v就是adjacent的。

Degree of a vertex：与这个顶点连接的边的数量，例如V的degree是3。

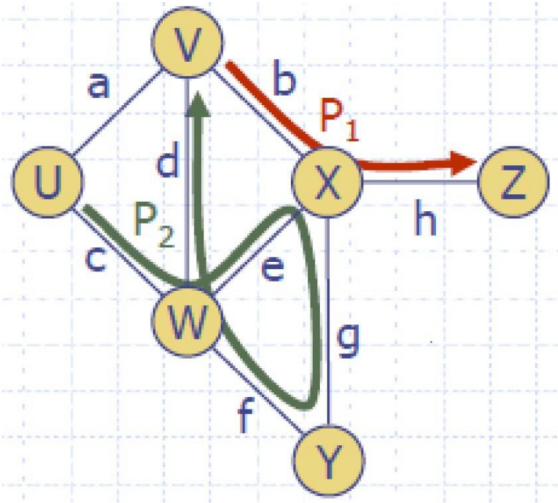
Parallel edges和Self-loop：前者如下图所示的h和i，后者是j。



Path和Simple Path：

Path：一组由边以及其顶点组成的序列，P2就是一条path。

Simple path：所有的顶点和边均不重复，例如P1。

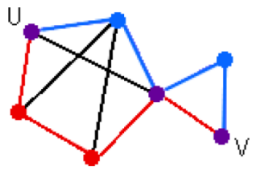


Walk：一组由顶点和边组成的序列。

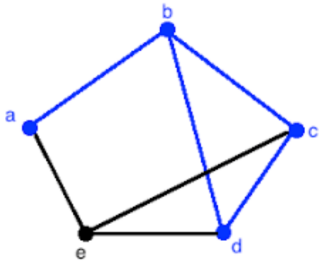
Trail：一个没有重复边的walk。

Circut：起始和终止的点相同的walk。

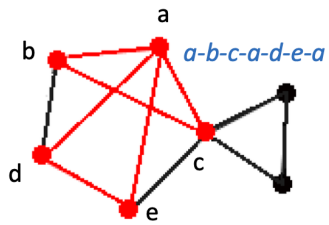
Circle：除起始和终止点没有相同点的circuit。



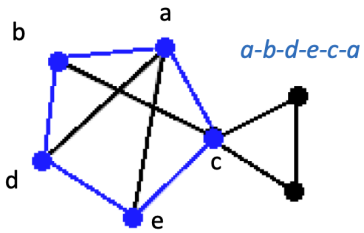
Graph 1:Two paths from U to V



Graph 2: Trail



Graph 3: circuit

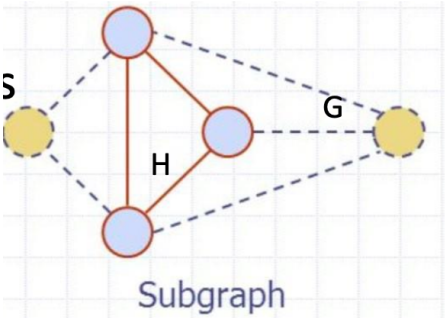


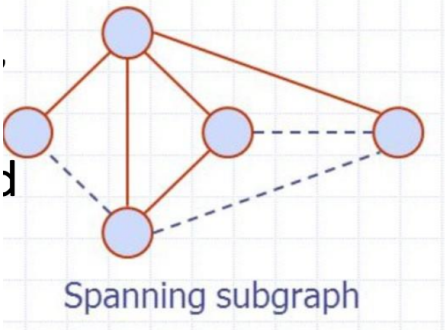
Graph 4:cycle

Directed walk：所有的边都是有向的，且在遍历时按照边的方向进行遍历，directed trail，circuit，cycle同理。

Subgraph：对于图G来说，G的子图H即为所有的边和顶点都是G的子集的图。

Spanning subgraph：G的子图但却包含G的所有顶点。



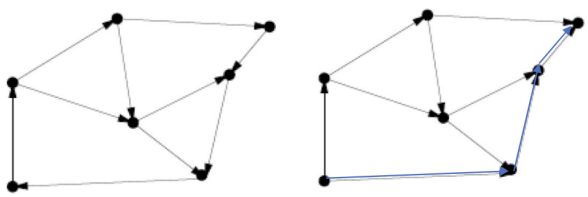


一个图被称作是**connected**的如果任意两个不同的顶点间都存在一条path。

如果一个图不是connected的，则它的最大的connected subgraphs被称作该图的**connected components**。

Acyclic graph：不包含任何cycle，树就是**connected acyclic undirected graphs**。

Directed acyclic graphs（DAGs）：定义和名字一样，而且通过遍历边的方法从一个顶点返回该顶点是不可能的。



2.3图的性质

Notation:

- deg(v):顶点v的度(degree)

- n:顶点的数量
- m:边的数量

Property 1:

$$\sum_v \deg(v) = 2m$$

Proof：由定义可以判断，与每个顶点相连的边都被计算的两遍。

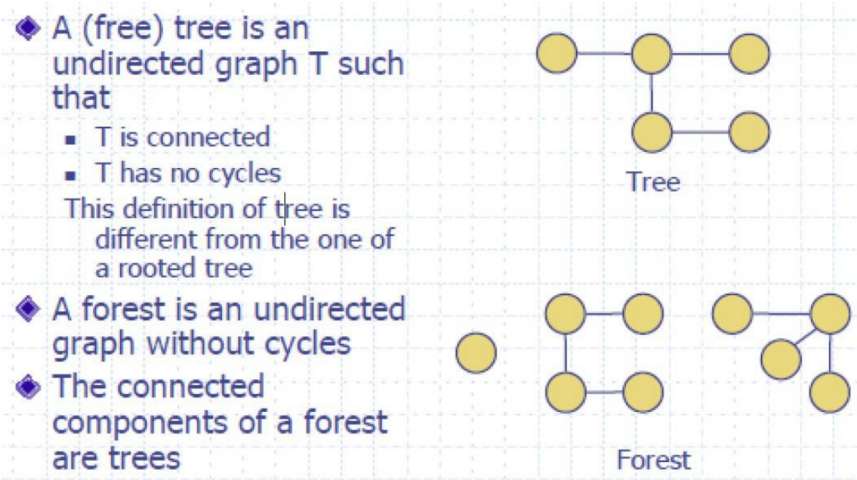
Property 2:

在无向图中，如果没有self-loop和multiple edges（即两个节点间只有一条边），那么则有：

m<=n(n-1)/2

Proof:每条边最多和n-1个顶点连接。

2.4Trees and forests



注意此处的树不同于此前的二叉树。另外：

Let *G* be an undirected graph with *n* vertices and *m* edges. We have the following observations:

If *G* is *connected*, then $m \geq n - 1$;

If *G* is a tree, then $m = n - 1$;

If *G* is a forest, then $m \leq n - 1$.

16

由此可得**Spanning trees**（生成树）：上述提到的树的子图。

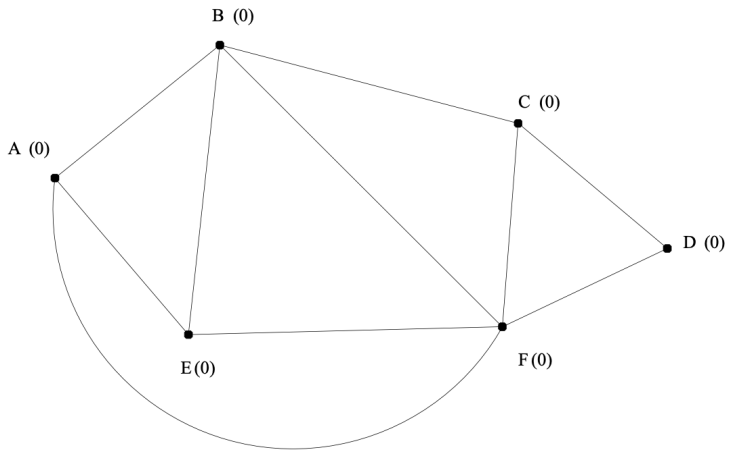
如果图不是树的话，则该图的生成树不是唯一的。同理可得到**Spanning forests**。

3.Pathfinding algorithms

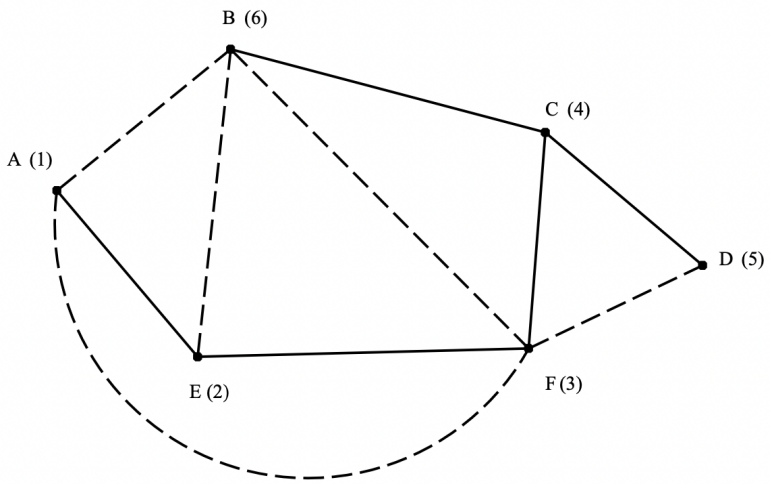
3.1Depth first search(DFS,深度优先搜索)

“在返回之前尽可能远的探索“，伪代码(递归)如下：

```
DFS(G, v)
    1>Input: A graph G and a vertex v of G.
    1>Output: A labeling of the edges of G.
1   for all edges e in G.INCIDENTEDGES(v)
2       do
3           if UNEXPLORED(e)
4             then w ← G.OPPOSITE(v,w)  //Return the endpoint
5                  if UNEXPLORED(w)
6                    then label e as discovery edge
7                         DFS(G, w)
8                  else
9                         Label e as back edge
```



如上图，A–E–F–C–D–B（此处遍历顺序不唯一）：



非递归的伪代码：

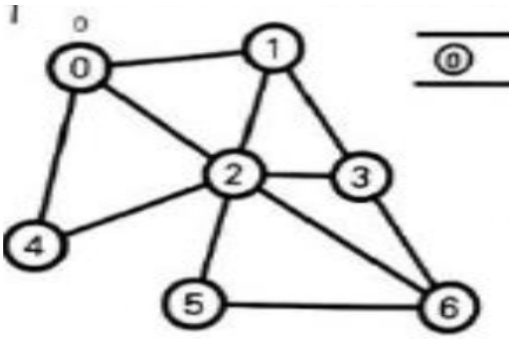
```
DFS(G, v)
    ▷ Input: A graph G and a vertex v of G.
    ▷ Output: A labeling of the discovery and back edges of G.
1   S.push(v)
2   VISITED(v) ← true
3   while not S.isEmpty() do
4           u ← S.top()
5           if there is an edge (u, w) and not VISITED(w) then
6                   S.push(w)
7                   VISITED(w) ← true
8                   parent(w) ← u
9                   Label the edge (u, w) as a discovery edge
10          elseif there is an edge (u, w) and VISITED(w) then
11                  Label the edge (u, w) as a back edge
12          else
13                  S.pop()
```

3.2Breadth First Search(BFS,广度优先搜索)

和DFS相反，BFS优先访问节点的所有邻居。伪代码如下：

```
BFS(G, v)
    ▷ Input: A graph G and a vertex v of G.
    ▷ Output: A labeling of the edges of G.
1   Q.enqueue(v)
2   VISITED(v) ← true
3   while not Q.isEmpty() do
4           u ← Q.dequeue()
5           for each edge (u, w) incident to u do
6                   if not VISITED(w) then
7                           Label (u, w) as a discovery edge
8                           VISITED(w) ← true
9                           Q.enqueue(w)
10                    else
11                          Label (u, w) as a cross edge
```

例子如下，遍历顺序为0124356:



DFS遍历：

- 更适用于复杂的连接性问题。
- 可以构成生成树（Spanning trees）即所有的不是树的边都是back edges。

BFS遍历：

- 寻找图中的最短路径。
- 可以构成生成树且所有的不是树的边都是cross edges。

Back edge：

Back edge: It is an edge (u, v) such that v is ancestor of node u but not part of DFS tree.

Cross edge：

It is a edge which connects two node such that they do not have any ancestor and a descendant relationship between them.

4.Weighted graphs

加权图即每条边上都有一个表示权重的数字标签。

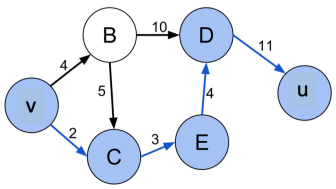
一条路径的长度（或权重）是这条路径上所有的权重相加起来。d(u,v)表示从u到v的最短路径。

单源最短路径 Single-Source Shortest Paths SSSP

Often in the case of weighted graphs, we want to consider the following problem:

For some fixed vertex v, find the shortest path from v to all other vertices $u \neq v$ in G (viewing weights on edges as distances).

This problem is known as the *Single-Source Shortest Path* problem (SSSP for short).



计算SSSP（没有带有负权值的边）时，可以用到贪心算法（即带有权值的广度优先搜索），即

迪杰斯特拉算法 Dijkstra's algorithm

Let v be a *source vertex* and let $D[u]$ represent the *temporary distance* in G from v to u. Initially we take $D[v]=0$ and $D[u]=+\infty$ for all $u \neq v$

At the start of the algorithm, all entries in the array D are temporary, but after each round of the algorithm one entry in D becomes *fixed*.

然后进行**边的松弛（Edge relaxation）**

For any vertex z for which $D[z]$ is temporary, perform the *edge relaxation*:

- If $D[u]+w(\{u,z\}) < D[z]$ then $D[z] \leftarrow D[u]+w(\{u,z\})$.

即每一轮应用算法都会确定下来一个点，此点到上一个点的距离最近。然后

When the edge relaxation step is completed for all temporarily labeled vertices we then

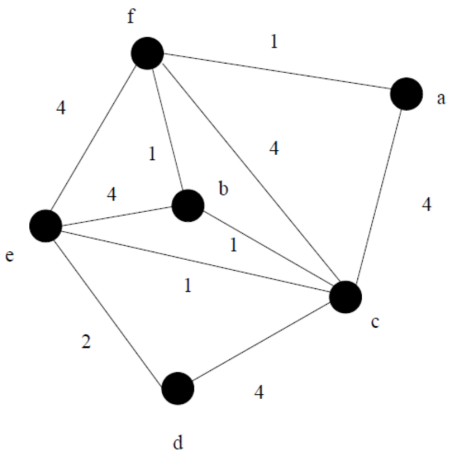
- Fix one entry in G (among vertices still outside C) with the smallest weight currently available, and add that new vertex to C.
- Proceed to the next stage of edge relaxation based on this extended set of vertices C.

伪代码如下：

```
DIJKSTRA(G, v)
1   D[v] ← 0
2   for each u ≠ v do
3           D[u] ← +∞
4   Let Q be a priority queue (heap) having all vertices of G
    using the D labels as keys.
5   while NOTEMPTY(Q) do
6           u ← REMOVEMIN(Q)
7           for each z s.t. (u, z) ∈ E do
8                   if D[u] + w({u, z}) < D[z]
9                     then D[z] ← D[u] + w({u, z})
10                         key(z) ← D[z]
11  return D
```

迪杰斯特拉算法即在每一步都应用贪心算法，对于该点进行广度优先搜索，并取权值最短的边。

例子如下：



D	[a	b	c	d	e	f]	C
	∞	∞	∞	0	∞	∞	ϕ
	∞	∞	4	**0**	2	∞	$\{d\}$
	∞	6	3	**0**	**2**	6	$\{d, e\}$
	7	4	**3**	**0**	**2**	6	$\{c, d, e\}$
	7	**4**	**3**	**0**	**2**	5	$\{b, c, d, e\}$
	6	**4**	**3**	**0**	**2**	**5**	$\{b, c, d, e, f\}$
	6	**4**	**3**	**0**	**2**	**5**	$\{a, b, c, d, e, f\}$

求d到其他节点的最短路径。

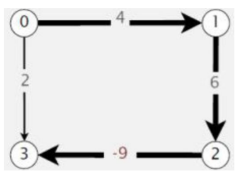
迪杰斯特拉算法综上一共只重复两件事：

1. 不断运行广度优先搜索寻找可见点，计算可见点到源点到距离长度。
2. 从当前已知的路径中选择长度最短的将其顶点加入S（记录已找出最短路径度顶点）作为确定找到的最短路径的顶点。

当存在负权值时不可以用迪杰斯特拉算法和在所有权值上加一个相同的数的方法：

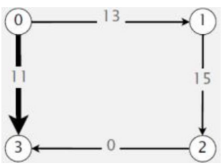
Shortest paths with negative weights: failed attempts

Dijkstra. Doesn't work with negative edge weights.



Dijkstra selects vertex 3 immediately after 0
But shortest path from 0 to 3 is $0 \to 1 \to 2 \to 3$

Re-weighting. Add a constant to every edge weight doesn't work.



Adding 9 to each edge weight changes the
shortest path from $0 \to 1 \to 2 \to 3$ to $0 \to 3$

因此需要一个新算法，即：

贝尔曼–福特算法 Bellman–ford algorithm

The *Bellman-Ford* algorithm can find shortest paths in graphs that have *negative weight edges*.

However, the assumption in this case is that the graph G is *directed*.

The Bellman-Ford algorithm also uses the notion of edge relaxation, but it does not use it with the greedy method.

Instead it performs a relaxation of every edge *exactly* $(V - 1)$ times.

This relies on the fact that, if there are no negative cycles, then there is a shortest path from v to any other vertex in G with at most $V - 1$ edges.

这种算法要求G是有向的，且每条边都要进行松弛操作(V−1)次。