- **Graph applications – graph-processing**

1. Topological sort

拓扑结构: 表示点和线之间位置关系. 拓扑图: 以图的形式来表示点与线之间关系.

问题的提出: 假设有一组待完成的任务, 任务之间具有优先关系或者是先决条件, 那么以何种次序合理安排这些任务呢?

⇨ 图模型: **vertex = task; directed edge = precedence constraint.**

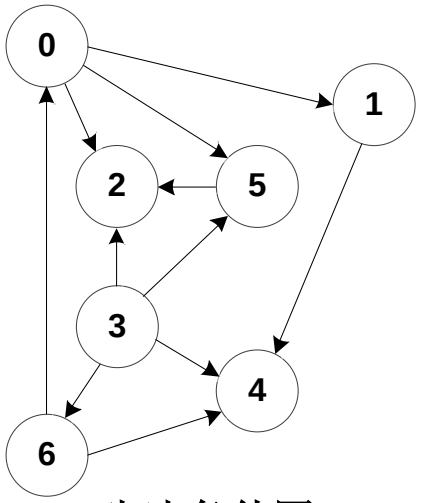
0. 算法基础
1. 计算复杂性理论
2. 人工智能
3. 计算机科学导论
4. 密码学
5. 科学计算
6. 高级程序设计

tasks or jobs

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

directed edges: v→w
↕
precedence constraint
↕
task v must be completed before task w

⇨



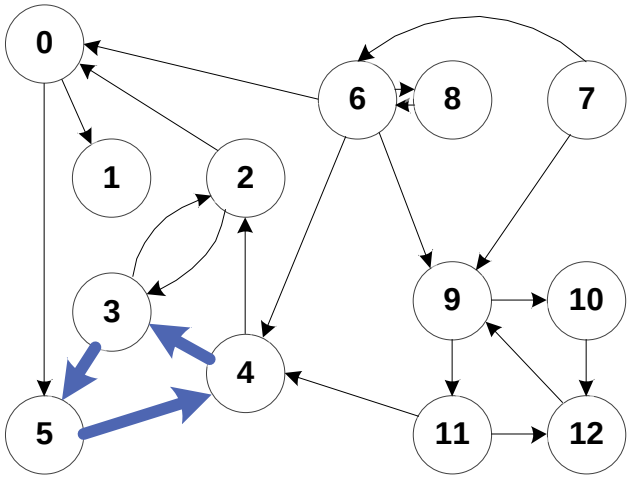
先决条件图
⇧
DAG: 有向无环图
Directed Acyclic Graph

- **Graph applications**

1. Topological sort

拓扑排序: 按照DAG顶点之间的优先关系将所有顶点排列成一个线性序列的过程.
拓扑排序的结果称之为拓扑序列(topological order).

⇨ • A digraph has a topological order iff no directed cycle.
• 一个DAG至少含有一个入度为0的顶点和一个出度为0的顶点. (WHY?见后Ex.)
• 对DAG的每条有向边v→w, 在拓扑序列中, 顶点v一定先于顶点w出现.
• 一个DAG的拓扑排序序列可能有多个. (因DAG的一些顶点之间彼此unrelated)

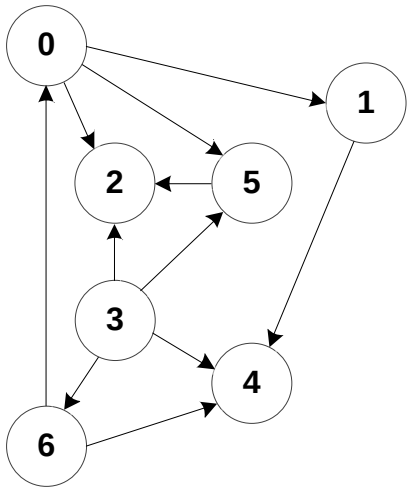


a digraph with a directed cycle

Scheduling tasks is infeasible for a directed cycle.

拓扑序列:
3, 6, 0, 5, 2, 1, 4
或 3, 6, 0, 1, 4, 5, 2
或 …

一个逆拓扑序列:
4, 1, 2, 5, 0, 6, 3



a directed acyclic graph

拓扑序列 & 逆拓扑序列 (reverse Topological order):
• 前者入度为0的顶点优先, 而后者则是出度为0的顶点优先.
• 对 逆拓扑序列 进行逆置(reverse)处理, 则得到拓扑序列; 反之亦然.

- **Graph applications**

1. Topological sort

拓扑排序方法1: DFS based to find a Topological order

回顾: DFS每个顶点仅被访问1次, 遍历有两个结果序列
①先序序列preorder ⇦ 在调用执行DFS()时产生
②后序序列postorder ⇦ 在DFS()执行结束时产生

```
void DFS(Graph *G, int v) { //Do DFS recursively from a given vertex v
   visited[v] = TRUE;
   EnQueue(preorder, v); //当前顶点v入队列preorder
   for (all w adjacent to v)  if (!visited[w]) DFS(G, w);
   EnQueue(postorder, v); //当前顶点v入队列postorder
   Push(Topoorder, v);  //当前顶点v入栈Topoorder, 即相当于逆置后序序列
}
```

⇨ 对DAG的DFS后序序列来说, 该图中的每一个顶点只有在所有依赖于它的顶点都被访问过之后才会被访问.

- 对于DAG来说, DFS后序序列中的第一个顶点, 其出度一定是0
- 对DAG的任意一条边v→w, 递归调用时, DFS(w)一定先于DFS(v)执行完成, 也就是说, DFS后序序列中, 顶点w一定先于顶点v出现.

⇨ 这样对DAG深度优先遍历所得到的后序序列, 逆置后就是该DAG的一个拓扑序列.

- **Graph applications**

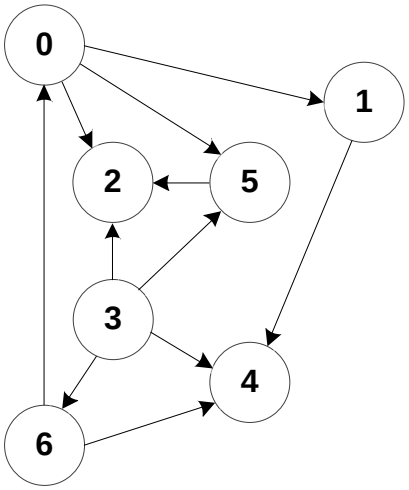
1. Topological sort

拓扑排序方法1: DFS based to find a Topological order

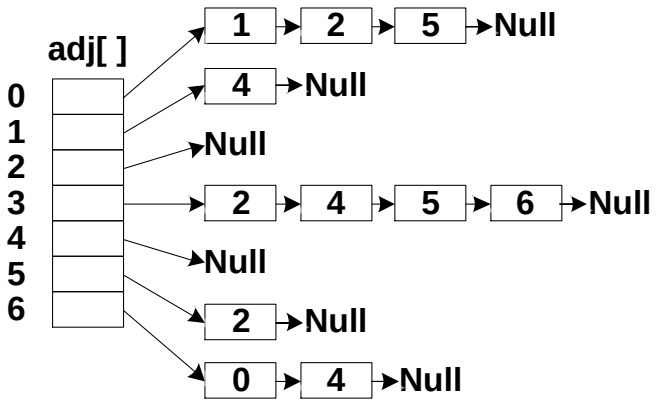
思路: 利用DFS所得到DAG的后序序列进行拓扑排序.

- 对于DAG来说, DFS后序序列中的第一个顶点, 其出度一定是0
- 对DAG的任意一条边v→w, 递归调用时, DFS(w)一定先于DFS(v)执行完成, 也就是说, DFS后序序列中, 顶点w一定先于顶点v出现

⇨ 逆置DFS的后序结果序列, 就得到一个DAG的拓扑序列.



a directed acyclic graph



postorder: 4 1 2 5 0 6 3

Topoorder: 3 6 0 5 2 1 4

例如 v = 3, 有3→2, 3→4, 3→5, 3→6

DFS(2) DFS(4) DFS(5) DFS(6)均先于DFS(3)执行完成

all vertices pointing from 3 are done, so they appear after 3 in Topoorder.

```
DFS(0)
  DFS(1)
    DFS(4)
    4 done
  1 done
  DFS(2)
  2 done
  DFS(5)
    check 2
  5 done
0 done
check 1
check 2
DFS(3)
  check 2
  check 4
  check 5
  DFS(6)
    check 0
    check 4
  6 done
3 done
check 4
check 5
check 6
done
```

- **Graph applications**

1. Topological sort

拓扑排序方法1: **DFS based to find a Topological order**

算法描述:

```
//Pre: The DAG G has been created.
void TopologicalSort(Graph *G) { //Do DFS for all unvisited vertices
    boolean visited[G->n];
    Stack T;  //be used to store Topoorder
    for (v = 0; v < G->n; v++) visited[v] = FALSE;
    for (v = 0; v < G->n; v++)  if (!visited[v])  DFS(G, v);
    while ( !StackEmpty(T) ) printf(" %d ", Pop(T));
} //Time O(V+E), Auxiliary space O(V)
```

```
void DFS(Graph *G, int v) {
    visited[v] = TRUE;
   for (all w adjacent to v)  if (!visited[w]) DFS(G, w);
    Push(T, v);  //push current vertex v  to stack T which stores results
}
```

算法中的栈也可用数组替换, 只是数组内容的逆序才是拓扑序列.

- **Graph applications**

1. Topological sort

拓扑排序方法2: **Kahn's algorithm to find a Topological order**

思路: 当前入度为0的顶点总是优先加入拓扑序列.

步骤:
1. 在有向图中选择一个入度为0的顶点v且输出之; //若没有, 则一定存在回路
2. 删除该顶点v以及所有以之为尾的弧v→w; //即相当于形成了一个新的有向子图
3. 重复上述两个步骤, 直至全部顶点均已输出, 或者当前图中不存在度为0的顶点为止.

⇨ 所需的辅助数据结构:

- Integer array indegree[ ] to store / maintain indegree of each vertex.
  //Initialize each indegree as 0
- Queue S [or Stack S] //push all vertices with indegree 0 into / onto S
- Integer array T[ ] to store result (a Topological order of all vertices)

其中, 计算图G中每个顶点v的入度的方法: 以邻接表用作图G的存储结构为例

```
for (v = 0; v < G->n; v++) indegree[v] = 0; //每个顶点v入度初始化为0
for (v = 0; v < G->n; v++)  {
    p = G->adjlist[v].head;
    while ( p != Null ) {
        indegree[p->adjvex]++; //v→w, 顶点v的每个邻接点w的入度加1
        p = p->next; } }
```

- **Graph applications**

1. Topological sort

拓扑排序方法2: **Kahn's algorithm to find a Topological order**

算法描述:

```
//Generates the topological order of graph G in the array T.
void TopologicalSort(Graph *G) {  //The graph G has been created.
    int indegree[G->n], T[G->n];
    …; //计算每个顶点v的入度indegree[v]
    InitQueue(Q);
    for (v = 0; v < G->n; ++v) if (!indegree[v]) EnQueue(Q, v);//入度为0的顶点入队
    count = 0;  //对输出顶点计数
    while ( !QueueEmpty(Q) ) {
        v = DeQueue(Q); T[count++] = v;  //输出顶点v至结果数组T并计数
       for ( p = G->adjlist[v].head; p; p = p->next) {
            w = p->adjvex;  //对顶点v的每个邻接点w的入度减1
            if ( !(--indegree[w]) ) EnQueue(Q, w); //若顶点w入度减为0, 则入队  } }
    if ( count < G->n) ERROR("该有向图含有环! 不能够产生拓扑序列.");
    else for ( v = 0; v < G->n; ++v ) printf(T[v]); //输出拓扑序列
} //Time O(V+E). 算法中辅助结构队列Q, 也可用栈S或数组替代, 只是顶点输出次序不同.
```

⇨ 可见, Kahn算法也是判断一个有向图是否存在回路的一种方法. (作业: 其它方法)

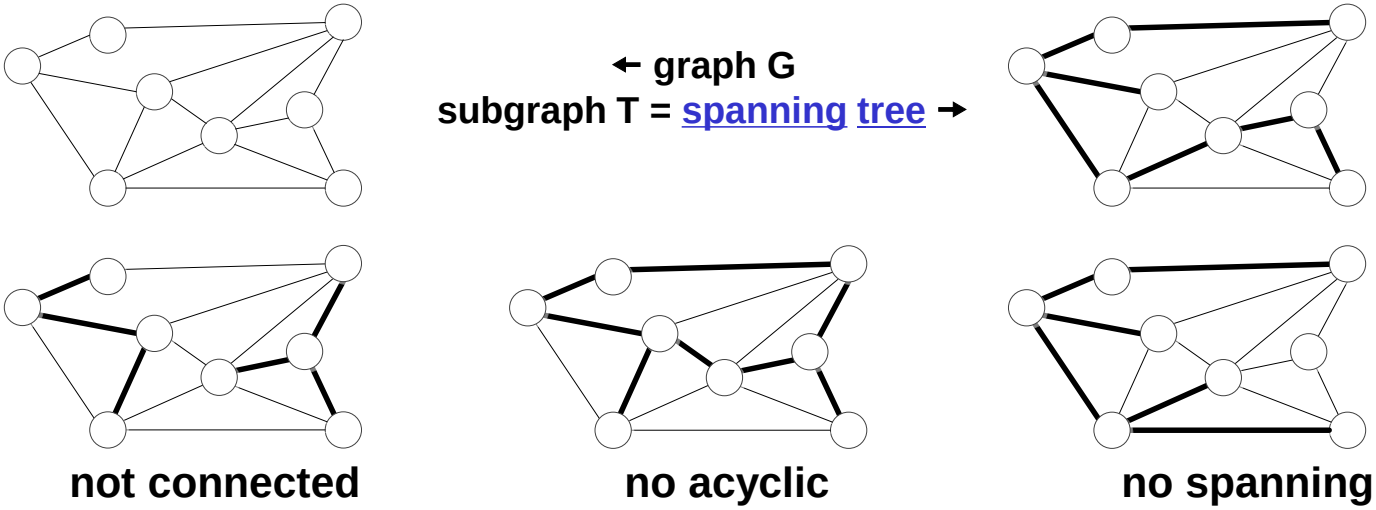
- **Graph applications**

2. Minimum Spanning Tree / 最小生成树 (MST)

生成树(spanning tree)的定义 **P159, P170**

在图论中, 通常将一个不带回路的连通的无向图定义为树.
一个无向图G本身不一定是树, 但它的子图是树. 其中很重要的一类子图是生成树.

⇨ **A *spanning tree* of undirected graph G is a subgraph T that is:**
①Connected; ②Acyclic; ③Includes all of the vertices.



⇨ 连通图的生成树就是包含该图所有顶点的极小连通子图. 所谓极小是指边数最少, 即若在生成树中去掉任何一条边则会使之变为非连通图, 若在生成树上再任意添加一条边则必定出现回路. → 连通图的生成树就是连接该图所有顶点的最小边集, 或是该图中不含有回路的最大边集.

极小连通子图(连通图, 边→极小) vs. 极大连通子图(连通分量, 顶点→极大)

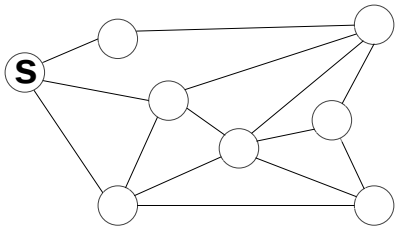
- **Graph applications**

2. Minimum Spanning Tree / 最小生成树 (MST)

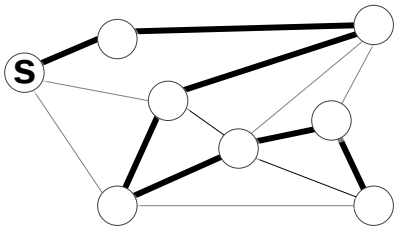
生成树(spanning tree)的生成过程 / 构造

生成树的生成过程也就是一个图的子图的创建过程. 对于一个图来说, 只要能够连通所有顶点而且不形成回路的任何一个子图都是它的生成树.

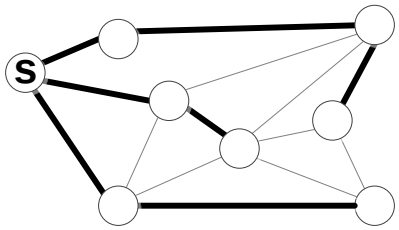
⇨ ①n阶图的生成树有且仅有n-1条边: 用图的表示方法表示生成树(子图: n顶点、n-1边).
②生成树的生成过程就是子图的创建过程. 如, 从某顶点出发的DFS/BFS生成树.



undirected graph G



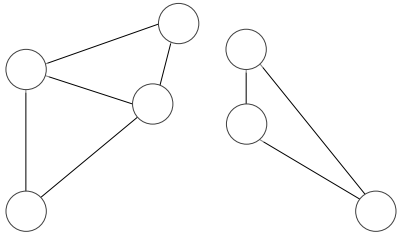
DFS生成树



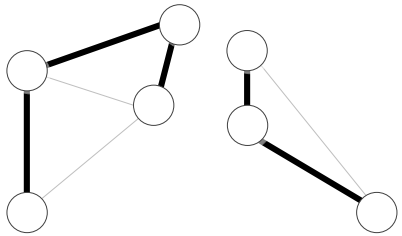
BFS生成树

➔ 还有其它生成生成树的方法: 最小生成树 (MST) & 最短路径树 (SPT).

③对非连通图, 则分别生成每个连通分量的生成树, 从而得到非连通图的生成森林.



spanning forest =
spanning tree of each component



- **Graph applications**

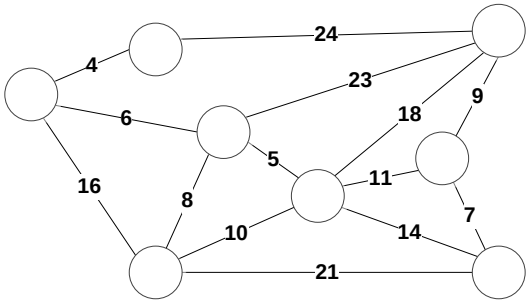
2. Minimum Spanning Tree / 最小生成树 (MST)

最小生成树的定义

最小生成树(最小代价生成树)也是生成树, 是各边权重之和最小的一种生成树.

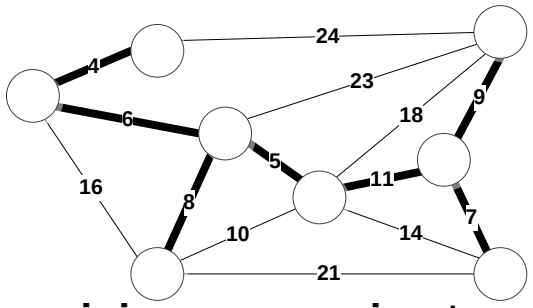
A minimum spanning tree(MST) or minimum cost spanning tree for a weighted, connected, undirected graph is a spanning tree having a cost less than or equal to the cost of every other possible spanning tree.

The sum of weights given to each edge of the spanning tree.



an edge-weighted graph

→ **MST**



minimum spanning tree T
(cost = 50 = 4+6+8+5+11+9+7)

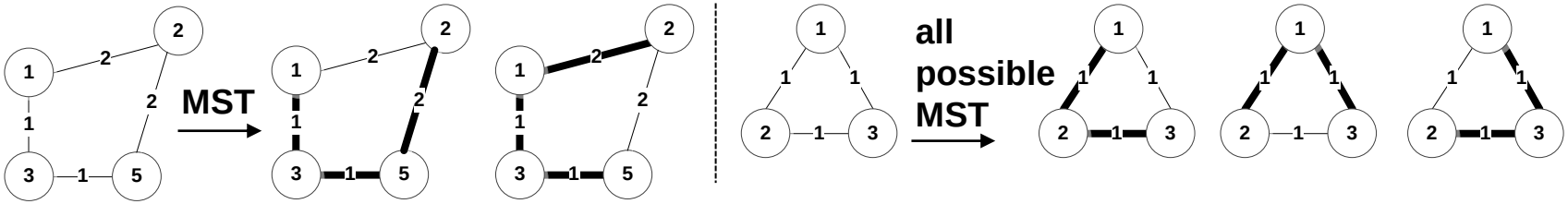
A kind of optimization problems on spanning trees. But how?

- **Graph applications**

2. Minimum Spanning Tree / 最小生成树 (MST)

最小生成树的性质

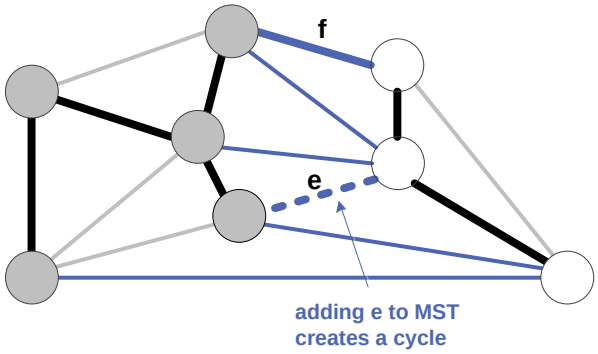
①可能的多重性: • 若存在权重相同的边, 则可能有几个权重之和相同的MST;
• 若所有边权重都相同, 则其每个MST的权重之和均相等.



②唯一性: 若每条边权重都不相同, 则存在唯一的一个MST.

③割(cut in the graph)的性质: [严. 教材. MST的基本性质]

割是将连通图的所有顶点划分为两个不相交的非空子集. 跨边(crossing edge)是图中的一条边且其所依附的两个顶点分别位于割的两个子集cut-set.
⇨ 对图的任意一个割, 权重最小的跨边一定在MST中. (用反证法容易证之)



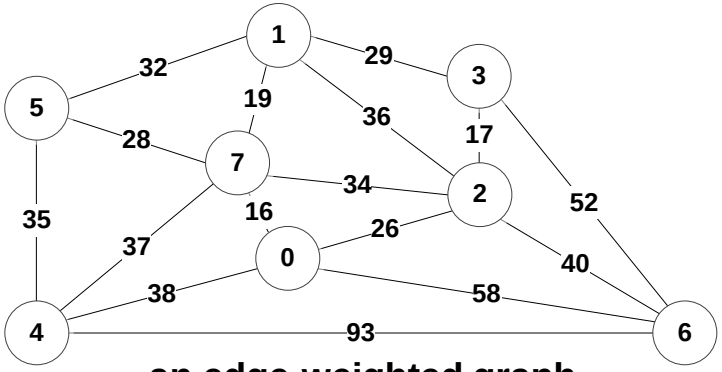
证明: 假设最小权重边e=(v, w)不在MST中, 则:
向MST添加边e, 就会形成一个环/回路, 这个环中的某条其它边f=(v', w')一定满足v'∈U, w'∈V-U (即f一定是一条跨边), 这时如果把边f删除掉, 那么依然是一棵生成树. 因为边e的权重<边f的权重, 所以这棵生成树是MST.

- **Graph applications**

2. Minimum Spanning Tree / 最小生成树 (MST)

最小生成树的表示 – 子图: n-1条边(以及每条边上的权重)

```
typedef struct edge { int v; int w; int weight;} Edge;
//加权边的类型定义, 以便处理和表示生成树上的边
Edge e; //表示图G或生成树T中的一条边
```



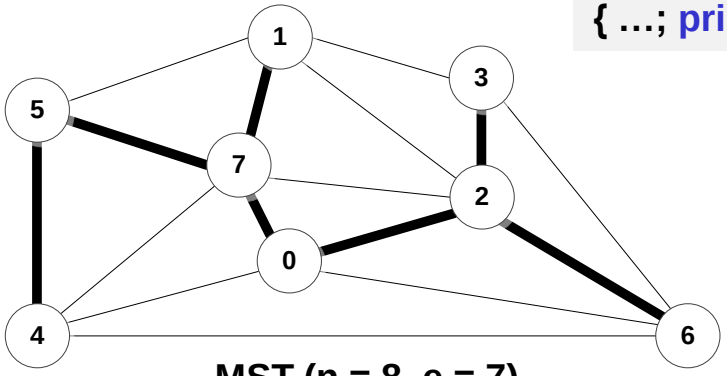
an edge-weighted graph

⇔

	0	1	2	3	4	5	6	7
0	0	0	26	0	38	0	58	16
1	0	0	36	29	0	32	0	19
2	26	36	0	17	0	0	40	34
3	0	29	17	0	0	0	52	0
4	38	0	0	0	0	35	93	37
5	0	32	0	0	35	0	0	28
6	58	0	40	52	93	0	0	0
7	16	19	34	0	37	28	0	0

⇩

```
void algo_MST(Graph *G)
{ …; printMST(e); …; }
```



MST (n = 8, e = 7)

⇔

⇩

v	w	weight
0	7	16
7	1	19
0	2	26
2	3	17
7	5	28
5	4	35
2	6	40

- **Graph applications**

2. Minimum Spanning Tree / 最小生成树 (MST)

最小生成树的生成方法

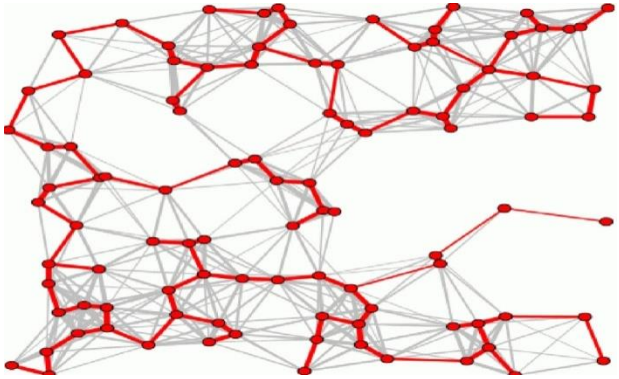
问题的提出: 要在n个城市间建立通信联络网的问题(and other diverse applications)

- 顶点 – 表示城市
- 边 & 权值 – 表示城市间的通信线路 & 建立通信线路所需花费代价(cost)
- MST – 如何找到一棵生成树使得每条边上的权值之和(即总代价)最小?

问题分析:

- n个城市间, 最多可设置n(n-1)/2条线路
- n个城市间建立通信网, 只需n-1条线路

⇨ 如何在可能的线路中选择n-1条, 能把所有城市(顶点)均连起来, 且总耗费(各边权值之和)最小?



问题的解决方法/MST生成方法:

☒ 暴力算法: 找到所有生成树

☑ 贪心算法: MST问题是一种具有最优子结构(optimal substructure)的问题

- 每一步都找出当前局部的最优解
- 且 这个局部的最优解是最终全局最优解的一部分

⇨ Prim算法 & Kruskal算法: 其思路是每一步生长MST的一条边, 且这条边是当前满足一定条件的最小权重边(min-weight edge).

这两个算法的条件各不相同 ⇗　　⇖ greedy criterion

- **Graph applications**

2. Minimum Spanning Tree / 最小生成树 (MST)

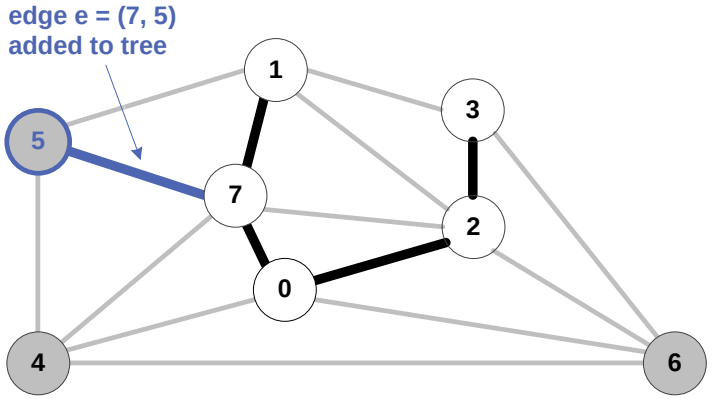
最小生成树的生成方法 – Prim算法

基本思路:

设G=(V, E)是连通图, 生成树(G的子图)T=(U, TE), U和TE分别是T的顶点集和边集.

- 从任意一个顶点开始(作为T的根结点, 假设顶点0), 即初始令U={0}, 0∈V, TE=ϕ;
- 将一个顶点在T中而另一个顶点在T之外的最小权重边(即最小跨边)加入T中, 即
  在所有 v∈U, w∈V–U 的边(v, w)∈E中, 找一条权重最小的边(v, w),
  将这条最小权重边(v, w)并入集合TE, 同时顶点w并入U;
- 重复n-1步上一操作直至全部顶点加入到U即U=V为止, 这时树T即为图G的MST.

例如: U = {0, 7, 1, 2, 3}, V–U = {4, 5, 6}
当前最小权重边 e = (7, 5)
其中, v = 7∈U, w = 5∈V-U.



⇨ 若把边(v, w)上权重视作为w到T的“距离”,
其基本思路也就是通过将T之外的且距离T
最近的顶点并入T中, 来生长树T.

2个关键问题:

- 如何找到当前满足条件的最小权重边?
- 最小权重边及其顶点并入T中后, 如何维护T之外的顶点到T的距离以便下一步处理?

- **Graph applications**

2. Minimum Spanning Tree / 最小生成树 (MST)

最小生成树的生成方法 – Prim算法

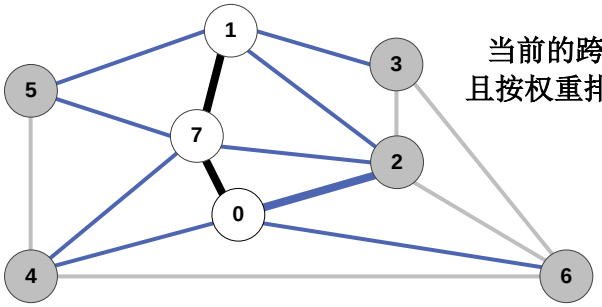
关键问题1: 如何找到当前满足条件的最小权重边?

设算法已进行至第k步处理, 那么U中已有k个顶点k-1条边, V- U中有n-k个顶点. 这时可能的跨边数量是k×(n- k), 从这么多边中选择一条权重最小的边显然低效. 事实上, 对于每个从w∈V- U顶点邻接到各个v∈U顶点的若干条边中, 只有最短的那一条才有可能是权重最小的边. 因此只需保留这n-k条边作为候选边集, 从中选择一条权重最小的边.

at most one entry per vertex

PQ is the best way

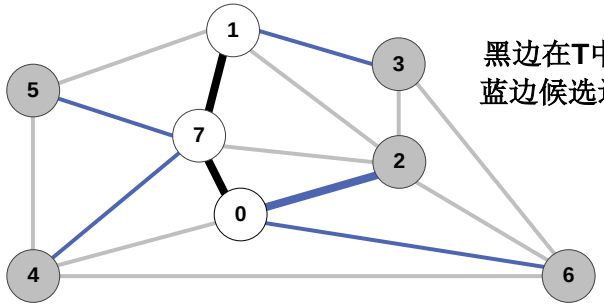
例如: T之外的一个顶点w = 2, 顶点2有3条边邻接到T: (0, 2) (7, 2) (1, 2).



lazy implementation ✗

当前的跨边, ⇨ 且按权重排序

0-2	26
7-5	28
1-3	29
1-5	32
7-2	34
1-2	36
7-4	37
0-4	38
0-6	58



eager implementation ✓

黑边在T中, ⇨
蓝边候选边

0-2	26
7-5	28
1-3	29
7-4	37
0-6	58

- **Graph applications**

2. Minimum Spanning Tree / 最小生成树 (MST)

最小生成树的生成方法 – Prim算法

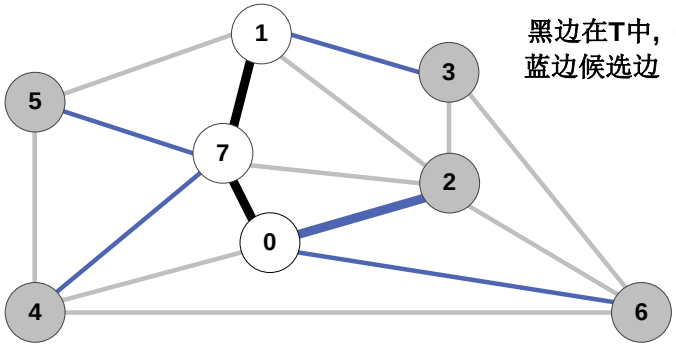
关键问题2: 当前最小权重边并入T中后, 如何维护候选边集?

解决这个问题的意义在于贪心算法的考虑
⇨ 下一步处理时, 确保能够找当前最小权重边, 即: 找到离树T最近的顶点.

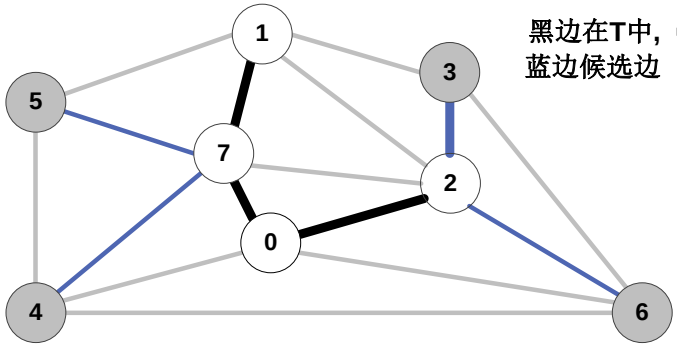
当边 e = (v, w) (包括顶点w)并入T中后, 只需关注与w相邻接的且在T之外的每个顶点x, 即 e = (w, x): 若weight(w, x) < weight(t, x), 则更新x到T距离值: 用weight(w, x)替换weight(t, x).

顶点w并入T之前, 顶点x到T的距离.
其中, t是T中的某个顶点.

例如:
e = (0, 2), w = 2, 边e和顶点2并入T之后:
e = (2, x), x分别等于3和6, t分别等于1和0.



黑边在T中, ⇨ 蓝边候选边		
	0-7	16
	7-1	19
	0-2	26
	7-5	28
	1-3	29
	7-4	37
	0-6	58



黑边在T中, ⇨ 蓝边候选边		
	0-7	16
	7-1	19
	0-2	26
	7-5	28
	2-3	17
	7-4	37
	2-6	58

- **Graph applications**

2. Minimum Spanning Tree / 最小生成树 (MST)

最小生成树的生成方法 – Prim算法

所需的辅助数据结构:

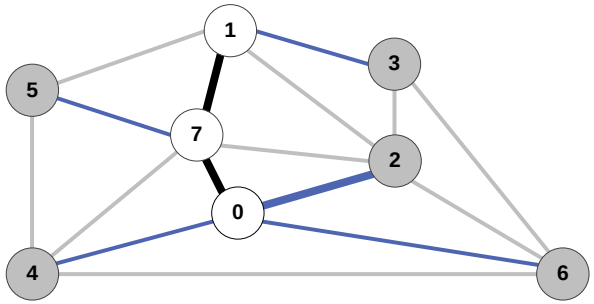
①用大小为n的一维数组mstSet表示一个顶点是否已在T中: **Boolean mstSet[ ];**

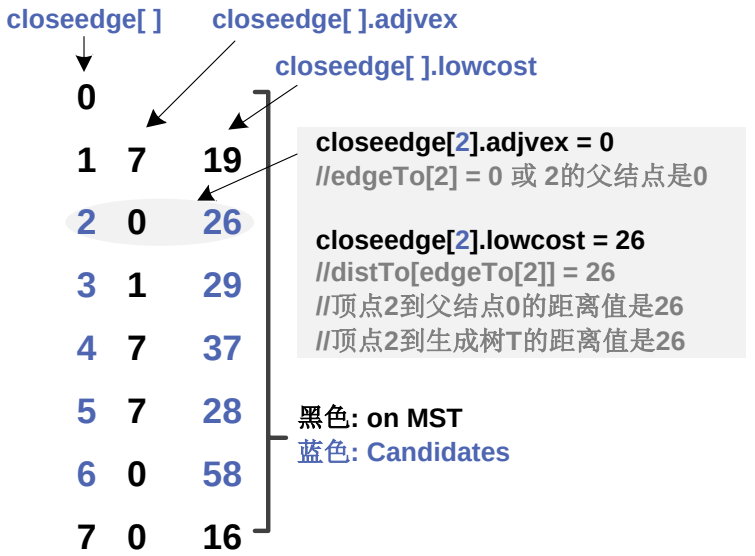
②用大小为n的一维数组closeedge来表示候选边集以及MST的边集:
struct { int adjvex; int lowcost; } closeedge[ ];

➜ 当顶点w在T之外(w∈V-U)时: //mstSet[w] == FALSE

- **closeedge[w].adjvex**是一条候选边**(adjvex, w)**, 其中**adjvex**是与顶点**w**相邻接且已在T中的顶点; //w的父结点是adjvex: edgeTo[w] = adjvex
- **closeedge[w].lowcost**表示候选边**(adjvex, w)**上的权重. //w到T的距离值

➜ 当顶点w已在T中(w∈U)时, **closeedge[w].adjvex**表示T中的一条边**(adjvex, w)**.





- **Graph applications**

2. Minimum Spanning Tree / 最小生成树 (MST)

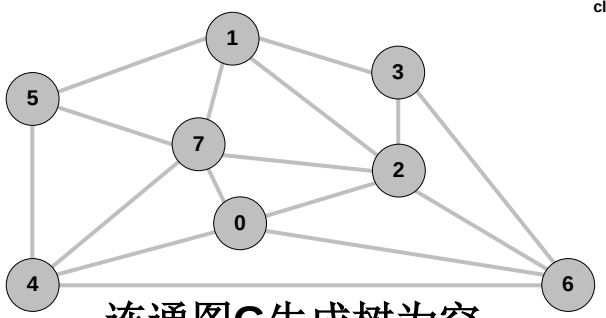
最小生成树的生成方法 – Prim算法实现

```
void PrimMST(Graph *G, int v) {  //图G为邻接矩阵, 从任意顶点v开始, 生成图G的MST
   struct { int adjvex; int lowcost; } closeedge[G->n]; //①表示候选边集②存储MST的边集
   Boolean mstSet[G->n]; //若顶点v已在MST中(v∈U), 则 mstSet[v] == TRUE.
   for ( i = 0; i < G->n; i++ ) { //MST开始为空, 候选边为空, 即全部顶点的距离值定义为∞
      closeedge[i].adjvex = -1; closeedge[i].lowcost = INFINITY; mstSet[i] = FALSE; }
   mstSet[v] = TRUE; //起始顶点v作为第1个顶点也是MST的"根结点"并入MST, 即 U = { v }
   for ( i = 0; i < G->n; i++ )
      if ( (i != v) && (G->edge[v][i]) ) { //对V-U中每个与v相邻接的顶点i, 初始化候选边集
         closeedge[i].adjvex = v; closeedge[i].lowcost = G->edge[v][i]; }
   for ( int count = 0; count < G->n-1; count++ ) { //将剩下的n-1个顶点并入MST
      min = INFINITY;
      for ( i = 0; i < G->n; i++ ) { //从候选边中找出当前最短边, 即 找出距离T最近的T之外的顶点w
         if ( (!mstSet[i]) && (closeedge[i].lowcost < min) ) {
            min = closeedge[i].lowcost; w = i; }
      mstSet[w] = TRUE; //顶点w并入MST
      printf(closeedge[w].adjvex, w); //输出当前最短边(adjvex, w)
      for ( i = 0; i < G->n; i++ )  //顶点w并入U集后, 更新候选边集. 即 更新V-U中相关顶点的距离值.
         if ( (G->edge[w][i]) && (!mstSet[i]) && (G->edge[w][i] < closeedge[i].lowcost) ) {
            closeedge[i].adjvex = w; closeedge[i].lowcost = G->edge[w][i]; } } } //O(n²)
```

- **Graph applications**

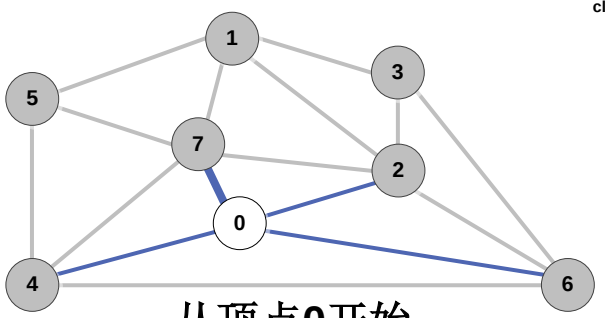
2. Minimum Spanning Tree / 最小生成树 (MST)

最小生成树的生成方法 – Prim算法计算过程



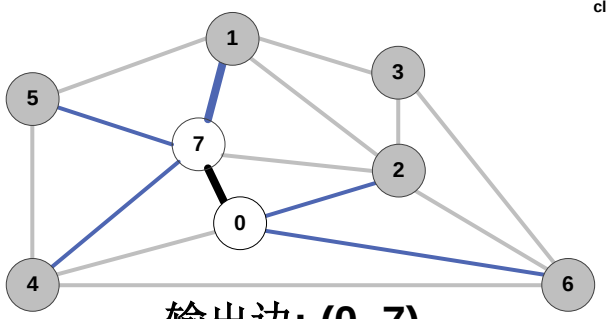
closeedge[ ]		
0	-	-
1	-	∞
2	-	∞
3	-	∞
4	-	∞
5	-	∞
6	-	∞
7	-	∞

连通图G生成树为空



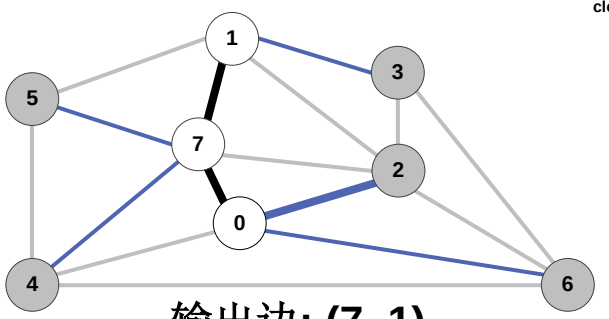
closeedge[ ]		
0	-	-
1	-	∞
2	0	26
3	-	∞
4	0	38
5	-	∞
6	0	58
7	0	16

从顶点0开始



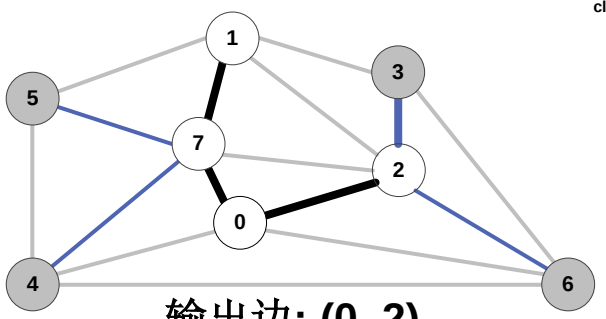
closeedge[ ]		
0	-	-
1	7	19
2	0	26
3	-	∞
4	7	37
5	7	28
6	0	58
7	0	16

输出边: (0, 7)



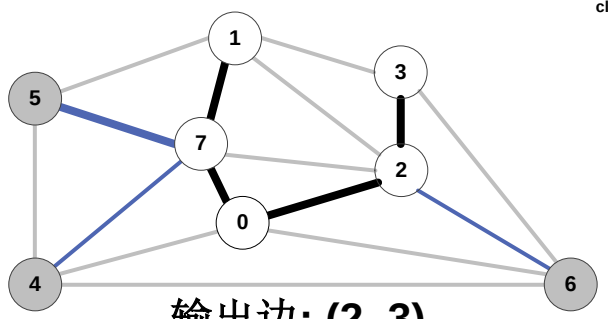
closeedge[ ]		
0	-	-
1	7	19
2	0	26
3	1	29
4	7	37
5	7	28
6	0	58
7	0	16

输出边: (7, 1)



closeedge[ ]		
0	-	-
1	7	19
2	0	26
3	2	17
4	7	37
5	7	28
6	2	40
7	0	16

输出边: (0, 2)



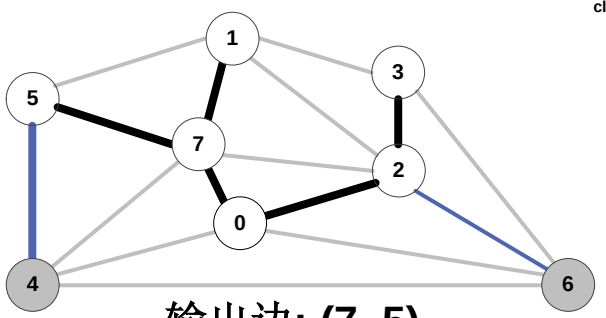
closeedge[ ]		
0	-	-
1	7	19
2	0	26
3	2	17
4	7	37
5	7	28
6	2	40
7	0	16

输出边: (2, 3)

- **Graph applications**

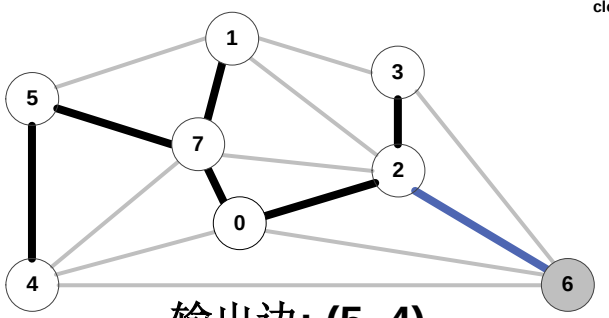
2. Minimum Spanning Tree / 最小生成树 (MST)

最小生成树的生成方法 – Prim算法计算过程(续)



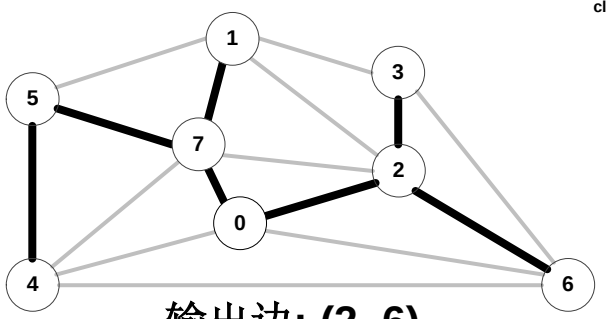
closeedge[ ]		
0	-	-
1	7	19
2	0	26
3	2	17
4	5	35
5	7	28
6	2	40
7	0	16

输出边: (7, 5)



closeedge[ ]		
0	-	-
1	7	19
2	0	26
3	2	17
4	5	35
5	7	28
6	2	40
7	0	16

输出边: (5, 4)



closeedge[ ]		
0	-	-
1	7	19
2	0	26
3	2	17
4	5	35
5	7	28
6	2	40
7	0	16

输出边: (2, 6)

MST: n = 8, e = 7.

⇔

closeedge[ ] //vertex	adjvex //edgeTo[ ]	lowcost //distTo[ ]
0	-	-
7	0	16
1	7	19
2	0	26
3	2	17
5	7	28
4	5	35
6	2	40

- **Graph applications**

2. Minimum Spanning Tree / 最小生成树 (MST)

最小生成树的生成方法 – Kruskal算法

基本思路: 通过加入当前权重最小边, 来生长树T.

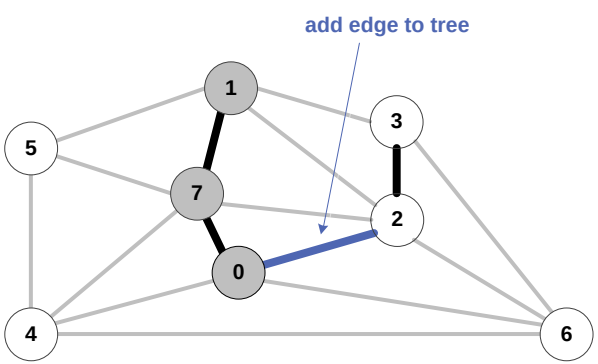
设G=(V, E)是连通图, 生成树(G的子图)T=(U, TE), U和TE分别是T的顶点集和边集

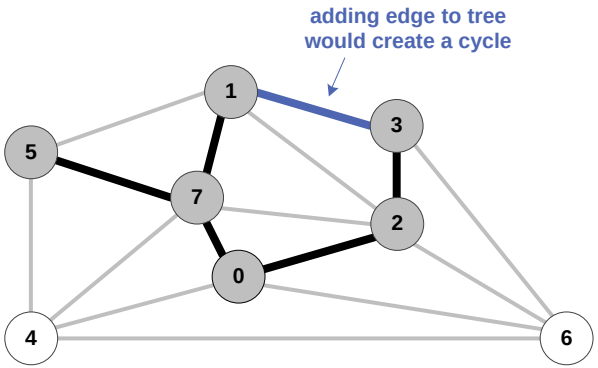
- 令生成树T的初态为含有n个连通分量的非连通图(n个顶点0条边):T=(V, {ϕ}); 并按权重的非递减顺序对图G的所有边进行排序;
- 选取权重最小边(Greedy criterion), 判断这条边是否会与当前生成树T形成一个环/回路(即判断这条边所依附的2个顶点是否会落在T中的同一个连通分量上)? 若未形成环, 则将这条边加入到T中; 否则, 舍去这条边;
- 重复上一操作, 直至T中有n-1条边 (或 T中所有顶点都在同一连通分量上)为止, 这时树T即为图G的MST.

关键问题:

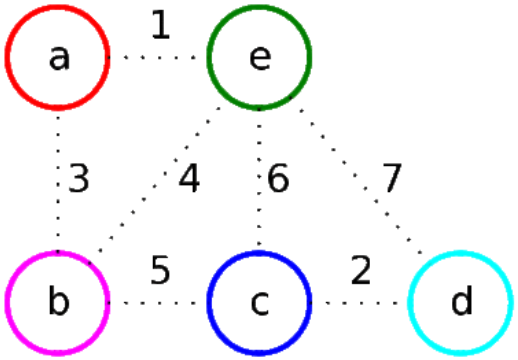
加入当前最小边后, 如何判断树T是否会产生回路?

⇨ 可通过不同的方法来判断(见作业)





Edge	ab	ae	bc	be	cd	ed	ec
Weight	3	1	5	4	2	7	6



MST of Kruskal
– from Wikipedia

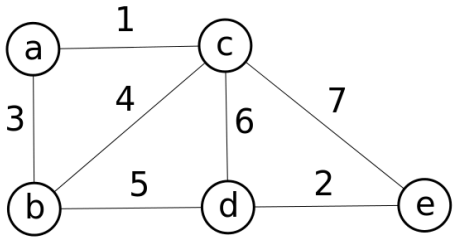
- **Graph applications**

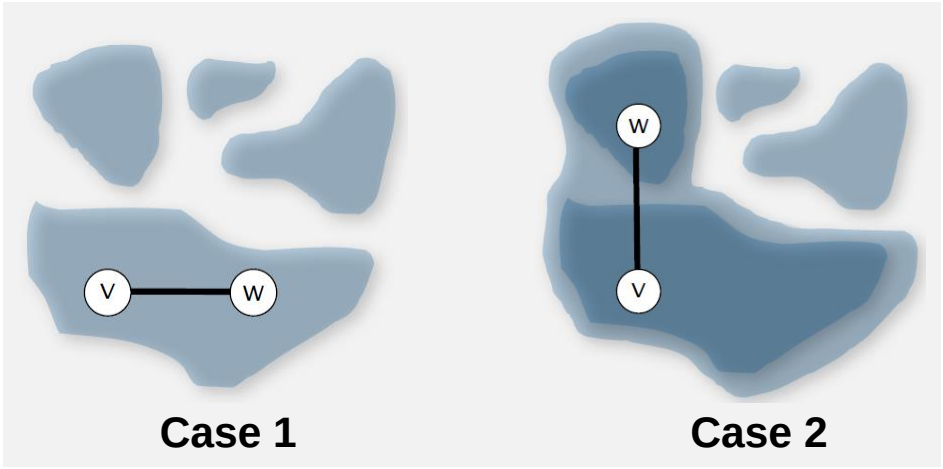
2. Minimum Spanning Tree / 最小生成树 (MST)

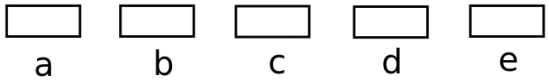
最小生成树的生成方法 – **Kruskal算法**

所需的辅助数据结构 – 并查集**(union-find):**

- 维护**T**中每个连通分量所在的子集;
- 若顶点**v**和**w**在同一个子集中(即查集**find**处理),
  则加入边**(v, w)**将会形成一个回路;
  否则, 将此边加入**T**中, 合并分别含有顶点**v**和**w**的两个子集(即并集**union**处理).







A demo for **Union-Find** when using **Kruskal**'s algorithm to find minimum spanning tree.

Case 1: adding v-w creates a cycle
Case 2: add v-w to T and merge subsets containing v and w

⇨ 并查集算法可用于检测一个图中是否含有回路.

- **Graph applications**

2. Minimum Spanning Tree / 最小生成树 (MST)

最小生成树的生成方法 – Kruskal算法实现

```
typedef struct node { int v, w, weight; } Edge; //边的类型定义
void KruskalMST(Graph *G) { //图G存储结构采用邻接矩阵
    int vset[G->n]; //并查集数据结构: 跟踪一个顶点所在子集. 若数组分量的值相同, 则顶点属于同一子集
    Edge e[G->e]; //用于表示G的边集
    Edge mstE[G->n-1]; //用于存储MST的边
    k = 0; //用于控制数组e的下标, 下标值从0开始计
    for ( i = 0; i < G->n; i++ )
        for ( j = i + 1; j < G->n; j++ )  if ( G->edge[i][j] ) { //图G中存在边(i, j)
            e[k].v = i; e[k].w = j; e[k].weight = G->edge[i][j]; k++; }
    HeapSort(Edge e); //使用效率高的排序方法对边集按权重非降序序排序, O(elog2e)
    for ( i = 0; i < G->n; i++) vset[i] = i; //初始化, 即G的n个顶点分别属于n个不同的子集
    j = 0; //用于控制T中的边数
    k = 0; //用于控制数组e的下标
    while ( j < G->n-1 ) {  //找出满足条件的n-1条最小权重边
        if ( vset[e[k].v] != vset[e[k].w] ) { //查集: 若当前最短边所依附的两个顶点分别属于不同的子集,
                                              //则把这条当前最短边加入到T中; 否则会形成回路, 舍弃此边
            mstE[j++] = e[k];
            for ( i = 0; i < G->n; i++ )  if ( vset[i] == vset[e[k].w] ) vset[i] = vset[e[k].v]; } //并集
        k++; } //准备处理下一条最短边
    printMST(mstE); //输出MST中的n-1条边
} //算法的语句频度为: n^2 + elog2e + n = O(elog2e)
```

- **Graph applications**

2. Minimum Spanning Tree / 最小生成树 (MST)

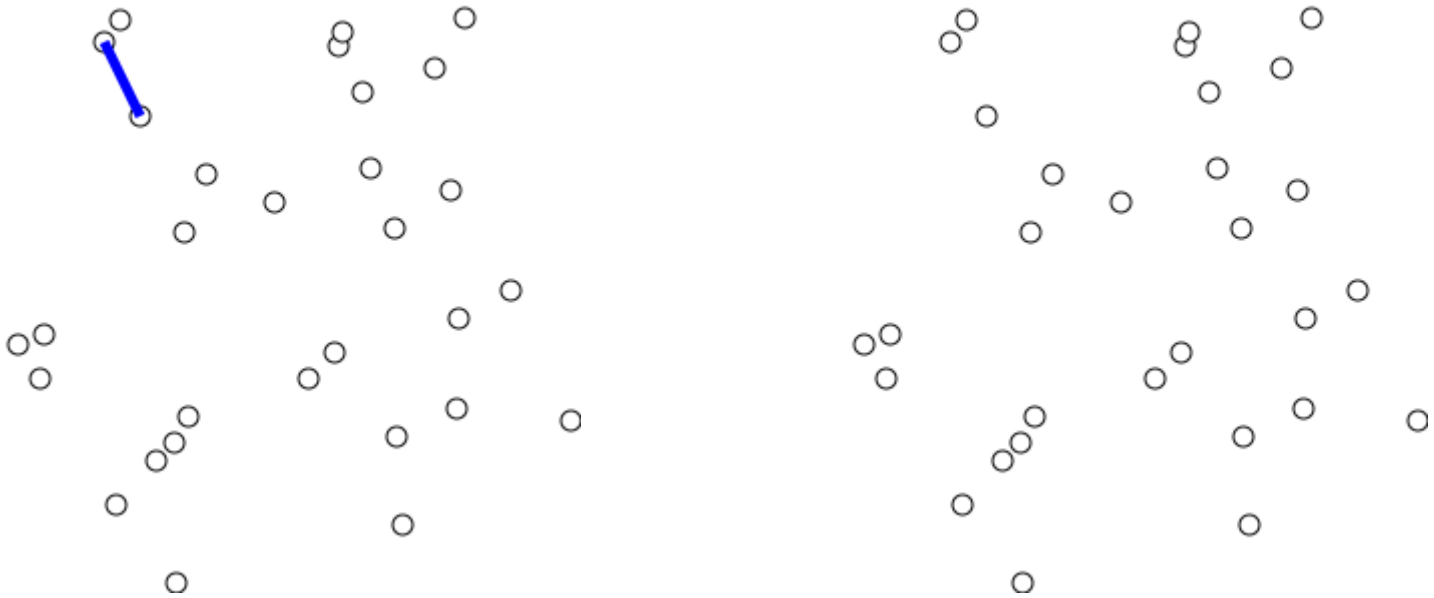
Summary of Prim's algorithm and Kruskal's algorithm

①Both greedily grow the T:

- **Prim算法是一种"加点"的方法, 每次选择一个在T之外的且距离T最近的顶点加入T中**
- **Kruskal算法是一种"加边"的方法, 每次选择一条权重最小且不形成回路的边加入T中**

②算法效率与算法实现方法以及存储结构有关:

- **Prim算法实现只是与顶点相关而与边无关, 所以适合于稠密图, 宜采用邻接矩阵表示法.$O(n^2)$**
- **Kruskal算法实现只与边相关而与顶点无关, 所以适于稀疏图, 宜采用邻接表表示法.$O(elog_2e)$**
- **两者所需辅助存储空间: 用于存储MST的n-1条边. → O(n)**



- **Graph applications**

3. Shortest Path in an edge-weighted digraph / 最短路径

问题的提出: 用一个带有权重的有向图来表示某个交通运行网

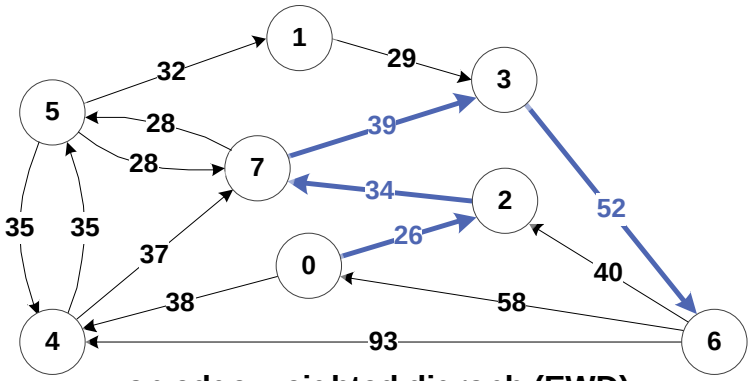
- 顶点 – 表示城市
- 有向边 – 表示城市间的交通联系
- 权重 – 表示此线路(一条有向边)的长度或沿此线路运行所需的时间或费用等花费
- 从一个顶点**s**(源点)出发, 沿着**EWD**的边到达其它一个顶点**t** (目标点)经过的所有可能路径中, 每条边上权重之和最小的一条路径 – 最短路径问题.

例如:

a EWD.txt

V
8
E
15

4 → 5	35
5 → 4	35
4 → 7	37
5 → 7	28
7 → 5	28
5 → 1	32
0 → 4	38
0 → 2	26
7 → 3	39
1 → 3	29
2 → 7	34
6 → 2	40
3 → 6	52
3 → 6	52
6 → 4	93



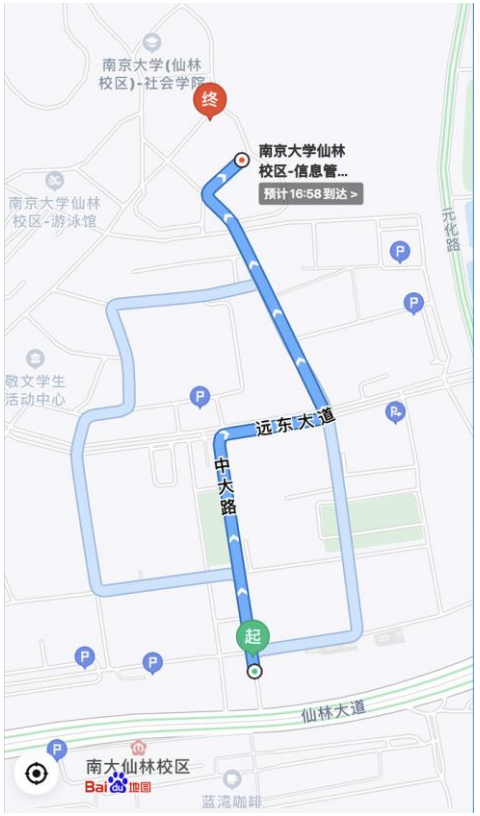
an edge-weighted digraph (EWD)

⇩

Shortest Path from 0 to 6: 151

0 → 2 26
2 → 7 34
7 → 3 39
3 → 6 52

⇨ 0 → 2 → 7 → 3 → 6



- **Graph applications**

3. Shortest Path / 最短路径

最短路径的几点说明: //shortest path variants

①最短路径涉及到图中的哪些顶点(which vertices)?
- 单源点: 从一个顶点 *s* 到每个其它的顶点的最短路径.
- 单目标点: 从每个顶点到一个顶点 *t* 的最短路径.
- 源点-目标点(单个顶点对): 从一个顶点 *s* 到另外一个顶点 *t* 的最短路径.
- 所有顶点对: 任一顶点对之间的最短路径.

②边上权重的有何限制? 均可.
- **Nonnegative weights.**
- **Euclidean distance.**
- **Arbitrary weights.** ← 可能存在负权值边, 或者 权重均为0 / 不带权重的图.

③无向图 或 有向图? 环? (强)连通?
- 均可, 只是有向图的路径是有向的.
- 允许环, 但负权环(negative cycle)(回路上权重之和为负数)例外. (因为无法收敛)
- 并非所有顶点都是可以到达的. (connected / reachable)
- 最短路径不一定唯一. (同MST)

⇨ **Simplifying assumption:** 单源最短路径且所有顶点均可达, 且EWD无负权边.
Goal: Find the shortest path from *s* to *every other vertex* in the given EWD.

➜单源路径的最短路径树(A shortest-paths tree / SPT):
①树的根结点为*s*; ②包含源点*s*在内的所有可达顶点的最短路径.
其中, 源点*s*到其自身的最短路径长度 = 0.

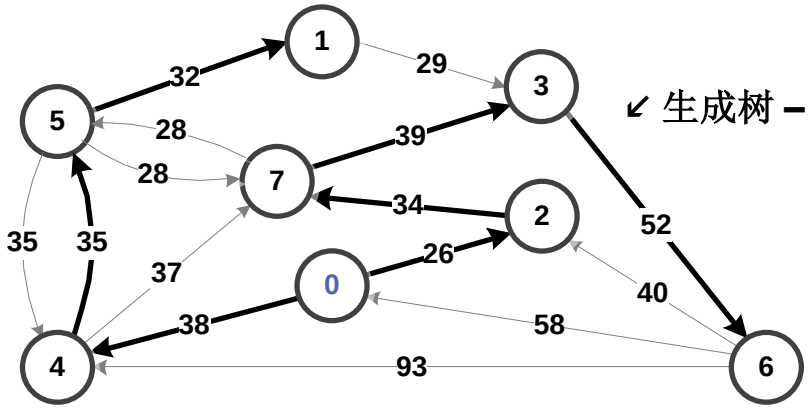
- **Graph applications**

3. Shortest Path / 最短路径

最短路径的表示: **(data structure for single-source shortest paths)**

使用两个数组来表示SPT(图的顶点作为数组的索引):

- **distTo[v]:** 从源点 ***s*** 到顶点 ***v*** 的最短路径长度 (也称为最短距离 / 最小距离).
  //int (or double) distTo[G->n]. ~ 每个顶点的距离值. 其中distTo[s] = 0.
- **edgeTo[v]:** 跟踪从 ***s*** 到 ***v*** 的最短路径的轨迹. //int edgeTo[G->n]; optional.
  //若路径上最后一条边e = u→v, 则v的父结点是u. 如此溯源, 可直至源点s.
  //源点s的父结点为空: edgeTo[s] = Null.



↙ 生成树 – SPT

SPT from vertex 0
//顶点 0 是根结点

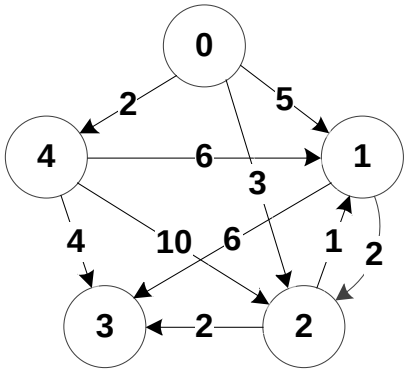
v	distTo[ ]	edgeTo[ ]
0	0	Null
1	105	5
2	26	0
3	99	7
4	38	0
5	73	4
6	151	3
7	60	2

SPT的表示: **distTo[ ] & edgeTo[ ]**

- **Graph applications**

3. Shortest Path / 最短路径

计算最短路径的方法:



A mini-EWD

Shortest Path from v_0 to v_3

The **Brute force** method finding all possible paths between Source and Destination and then finding the minimum.

- ✘ $v_0 \rightarrow v_1 \rightarrow v_3$: 11
- ✘ $v_0 \rightarrow v_1 \rightarrow v_2 \rightarrow v_3$: 9
- ✓ $v_0 \rightarrow v_2 \rightarrow v_3$: 5
- ✘ $v_0 \rightarrow v_2 \rightarrow v_1 \rightarrow v_3$: 10
- ✘ $v_0 \rightarrow v_4 \rightarrow v_3$: 6
- ✘ $v_0 \rightarrow v_4 \rightarrow v_2 \rightarrow v_3$: 14

⇧ the **WORST** strategy

Simple Greedy Method: At each node, choose the shortest outgoing path.

- ✘ $v_0 \rightarrow v_4 \rightarrow v_3$: 6

⇧ also **FAILs(WHY?)**

- **Graph applications**

3. Shortest Path / 最短路径

计算最短路径的方法 – 最短路径的性质

最短路径的子路径是最短路径 **(Subpaths of shortest paths are shortest paths):**
对任一带权有向图**G**, 若设 $p = \langle v_0, v_1, v_2, \ldots, v_k \rangle$ 是从顶点v_0到顶点v_k的最短路径, 对于范围 $0 \le i \le j \le k$ 中的任意**i, j**, 设$p_{ij} = \langle v_i, \ldots, v_j \rangle$ 为从顶点v_i到v_j的子路径, 那么, p_{ij}是从顶点v_i到v_j的最短路.

证明: 假若把最短路径**p**分解为: $v_0 \overset{p_{0i}}{\dashrightarrow} v_i \overset{p_{ij}}{\dashrightarrow} v_j \overset{p_{jk}}{\dashrightarrow} v_k$, 这时最短路径**p**的长度可记为 $w(p) = w(p_{0i}) + w(p_{ij}) + w(p_{jk})$. 现假设另外有一条从顶点$v_i$到$v_j$的路径$p'_{ij}$, 其路径长度记为$w(p'_{ij})$, 若 $w(p'_{ij}) < w(p_{ij})$, 那么 $v_0 \overset{p_{0i}}{\dashrightarrow} v_i \overset{p'_{ij}}{\dashrightarrow} v_j \overset{p_{jk}}{\dashrightarrow} v_k$是另外一条从顶点$v_0$到$v_k$的路径**p'**, 其路径长度 $w(p') = w(p_{0i}) + w(p'_{ij}) + w(p_{jk}) < w(p)$, 与**p**是最短路径相矛盾. 用反证法得证.

⇨ 最短路径问题是具有<u>最优子结构</u>问题的典型实例. 计算单源最短路径的**Dijkstra**算法是一种贪心算法.

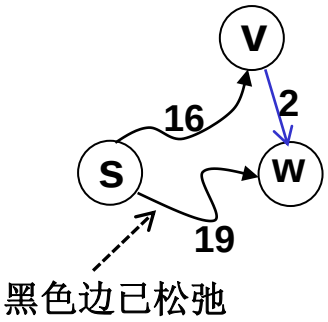
- **Graph applications**

3. Shortest Path / 最短路径

计算最短路径的方法 – 松弛操作 ← 数学上的一种逐步逼近的技术

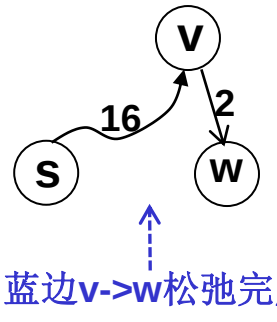
边的松弛 (edge relaxation):

已知 distTo[v], distTo[w]分别是顶点v和w到源点s的当前距离值(最短路径长度), 若存在边e=v→w, 使得经过v到w的路径更短, 则更新顶点w的距离值distTo[w].
⇨ 松弛边v->w, 以使得顶点w到源点s的路径长度变得更短.



黑色边已松弛

```
//Relax edge e = v->w
void Relax(int v, int w, int weight) {
   if ( distTo[w] > distTo[v] + weight ) {
      distTo[w] = distTo[v] + weight;
      edgeTo[w] = v; } } //松弛边⇨缩小distTo[w]的值
```



蓝边v->w松弛完成

⇨ 通过松弛操作, 逐步减少(progressively decresing)从源点s到每个顶点w的最短路径长度的暂定值(a tentative value), 直到它达到实际最短路径长度.

顶点的松弛 (vertex relaxation): 本质上就是边的松弛, 即松弛与顶点v相依附的每条边v→w

```
void Relax(Graph *G, int v) { //Relax vertex v. Prim算法实际上就是使用了这里的顶点松弛.
   for each Edge e = v->w leaving from v
      if ( distTo[w] > distTo[v] + weight ) {
         distTo[w] = distTo[v] + weight;
         edgeTo[w] = v; } } } //松弛顶点⇨缩小顶点v的每个邻接点w的distTo[w]的值
```

- **Graph applications**

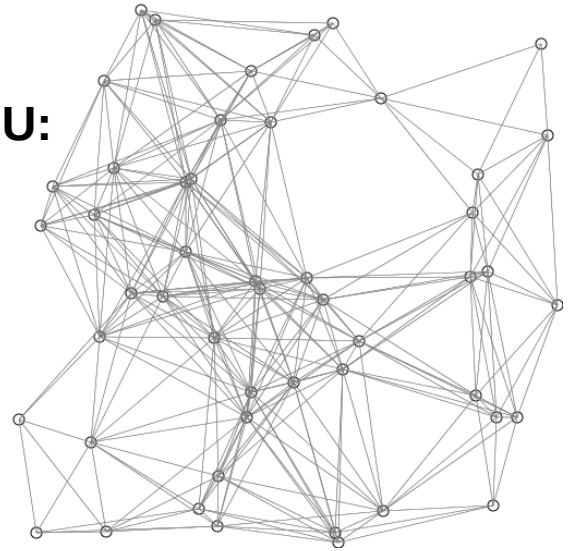
3. Shortest Path / 最短路径

计算最短路径的方法 **– Dijkstra(迪杰斯特拉)算法:**

基本思路:

①初始化. 将一个带权有向图的V集分割成两个子集U和V-U:

- **U –** 已求出最短路径的顶点的子集;
  //初始时为空集 ~ SPT为空
- **V-U –** 尚未确定最短路径的顶点子集;
  //初始时所有顶点都设置为未确定/FALSE
- G中每个顶点距离值distTo[v](从源点s到v的最短路径长度)初始均暂定为正无穷大, 并令源点s的距离值为0, 以便首先加入SPT.
  //distTo[s] = 0, 其它顶点的distTo[v] = ∞



⇧

A demo of Dijkstra's algo:
红线表示已得到的最短路径;
蓝线表示更新距离的位置.
//蓝线是松弛发生的位置.

②将V-U(不在SPT)中各个顶点按距离值distTo[v]非递减的顺序依次加入到U(=SPT)中, 直到所有顶点加入到U为止(U=V):

- 在V-U(不在SPT)中选择一个当前距离值最小(greedy)的顶点v;
- 将顶点v加入到U(=SPT)中; //将该顶点v设置为TRUE
- 更新顶点v相邻接的且在V-U中的所有顶点的距离值: //relax each edge leaving from v

```
for each Edge e = (v, w)  leaving from v //与v相邻接的每条出边
    distTo[w] = min { distTo[w], distTo[v] + weight of edge(v,w) } //使得w到s的距离值更短
```

- **Graph applications**

3. Shortest Path / 最短路径

计算最短路径的方法 – **Dijkstra(迪杰斯特拉)算法的描述:**

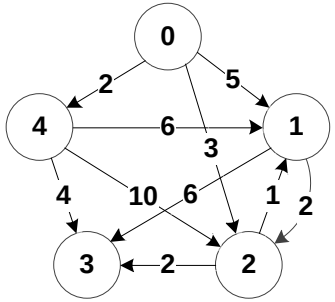
```
//Dijkstra's single source shortest path algorithm: 维护两个数组 distTo & edgeTo
void DijkstraSPT(Graph* G, int src) { //G uses adjacency matrix representation.
   int distTo[G->n]; //distTo[v] will hold the shortest distance from src to v
   int edgeTo[G->n]; //keep track of shortest distance paths. src is the root of the SPT
   Boolean sptSet[G->n]; //sptSet[v] will be true iff v is in the set U.
   for (int v = 0; v < G->n; v++) { //Initialize each vertex as a tentative distance value
      distTo[v] = INFINITY; sptSet[v] = FALSE;   }
   distTo[src] = 0; edgeTo[src] = -1; //Distance of source vertex from itself is always 0.
   for (int count = 0; count < G->n; count++) { //Find shortest path for all vertices
      int v = minDistance(distTo, sptSet); //Find the closest vertex to src
      sptSet[v] = TRUE; //Add the picked vertex to the set U
      for (w = 0; w < G->n; w++)
      //Update distTo[w] only if is not in sptSet, there is an edge from v to w, and total
      //weight of path from src to w through v is less than current value of distTo[w]
         if ( !sptSet[w] && G->edge[v][w] && (distTo[v] + G->edge[v][w] < distTo[w])) {
            distTo[w] = distTo[v] + G->edge[v][w]; edgeTo[w] = v; }
   printSolution(src, distTo, edgeTo); } } //Time: O(n^2), Space: O(n)
```

```
int minDistance(int distTo[ ], Boolean sptSet[ ]) { //←minheap or PQ
   int min = INFINITY, min_index;
   for (int v = 0; v < G->n; v++)
      if ( !sptSet[v] && distTo[v] < min ) { min = distTo[v]; min_index = v; }
   return min_index; }
```

- **Graph applications**

3. Shortest Path / 最短路径

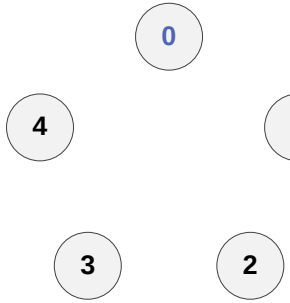
计算最短路径的方法 – **Dijkstra(迪杰斯特拉)算法的计算过程:**



adjacency matrix representation of the EWD

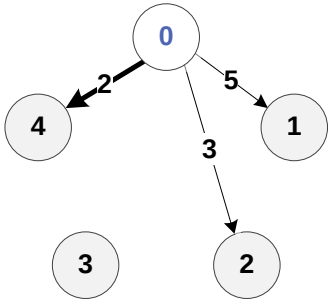
	0	1	2	3	4
0	0	5	3	0	2
1	0	0	2	6	0
2	0	1	0	2	0
3	0	0	0	0	0
4	0	6	10	4	0

初始化. SPT = U = { ϕ },
V - U = { 0, 1, 2, 3, 4 }.

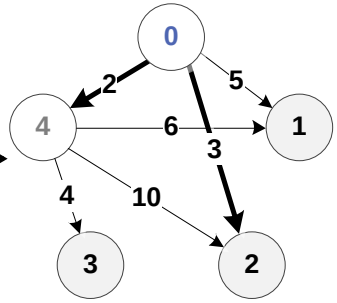


v	distTo[ ]	edgeTo[ ]
0	0	-1
1	∞	-
2	∞	-
3	∞	-
4	∞	-

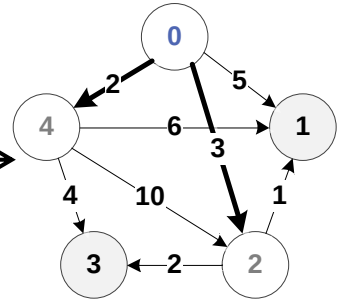
循环处理



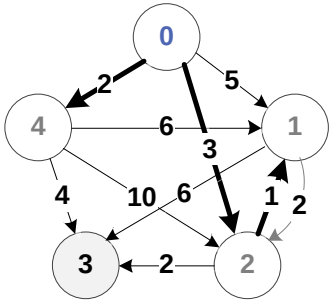
v	distTo[ ]	edgeTo[ ]
0	0	-1
1	5	0
2	3	0
3	∞	-
4	2	0



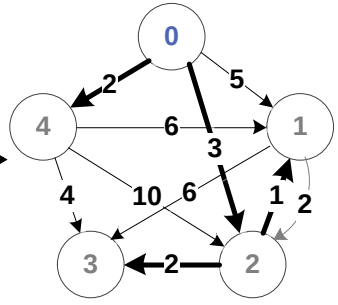
v	distTo[ ]	edgeTo[ ]
0	0	-1
1	5	0
2	3	0
3	6	4
4	2	0



v	distTo[ ]	edgeTo[ ]
0	0	-1
1	4	2
2	3	0
3	5	2
4	2	0



v	distTo[ ]	edgeTo[ ]
0	0	-1
1	4	2
2	3	0
3	5	2
4	2	0



v	distTo[ ]	edgeTo[ ]
0	0	-1
1	4	2
2	3	0
3	5	2
4	2	0

循环结束,
输出结果.

s to v	cost	path
0→1:	4,	0→2→1
0→2:	3,	0→2
0→3:	5,	0→2→3
0→4:	2,	0→4

- **Graph applications**

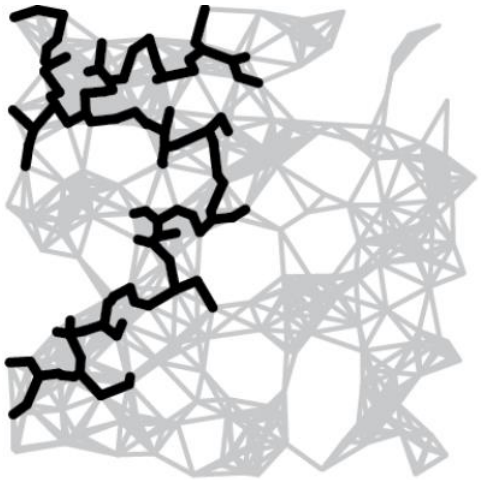
3. Shortest Path / 最短路径 – 小结

①Dijkstra算法看着“面熟”? yes.

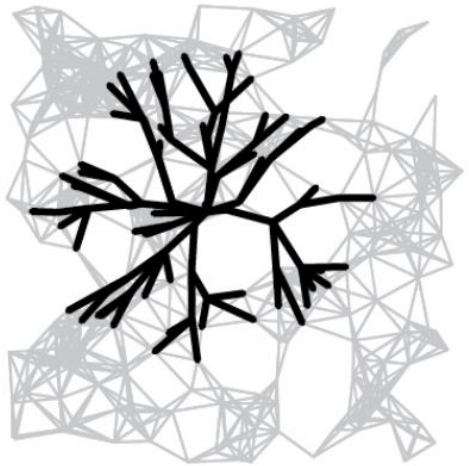
- **Dijkstra算法和Prim算法本质上相同;**
- **这两者同根同源, 都属于生成树系列的范畴.**

②两者主要区别在于: 生长树时, 用于选择下一个顶点的规则不同.

- **Prim算法: (通过无向边)选择离树T最近的顶点.**
- **Dijkstra算法: (通过有向边)选择离源点s最近的顶点.**



a minimum panning tree



a single-source shortest paths tree

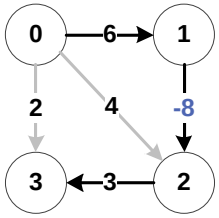
③WHY Dijkstra’s algorithm doesn’t work with negative edge weights? →

- **Graph applications**

3. Shortest Path / 最短路径 – 小结

③WHY Dijkstra's algorithm doesn't work with negative edge weights?

Dijkstra算法是按顶点到源点距离值递增的次序, 逐个将顶点加入到SPT; 而且已经加入到SPT的顶点, 其距离值将不再会改变. 若存在负权边, 则可能出现与之相矛盾的情形:



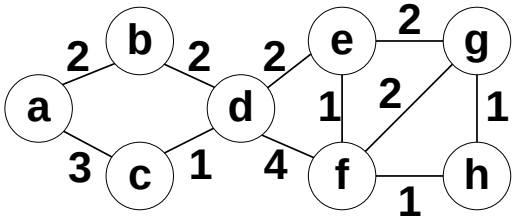
Dijkstra算法在把源点0加入SPT之后, 首先选择的是将顶点3加入SPT, 也就是0到3的最短路径是0→3. 而实际的最短路径应该是0→1→2→3.

⇨ 计算最短路径需要其它不同的算法: **(After class)**

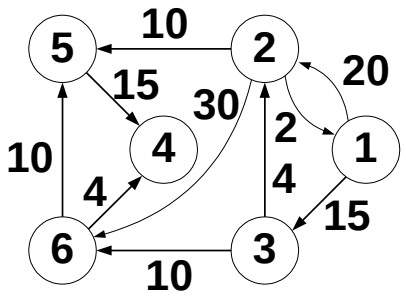
- **Bellman-Ford算法: 单源SP. 适用于含负权边, 但不含负权环. 效率不好, O(V*E).**
- **Floyd-Warshall算法: 计算所有顶点对之间的最短路径. 优于执行V次的Dijkstra算法. $O(V^3)$, 但简洁有效. 更适稠密图.**

- **Exercises (After class):**

9. 分别用**Prim**和**Kruskal**算法画出构造下列连通图的最小生成树的过程.



10. 利用**Dijkstra**算法求出如下有向网的从源点**1**到其它各顶点的最短路径.



11. 给出用**DFS**搜索下图所得到的逆拓扑序列. **[**注**:** 逆拓扑序列的形成是指每一步均是输出图中当前无后继**(**即出度为**0)**的顶点**]**