Part VI

Graph Algorithms (I)

songyou@buaa.edu.cn

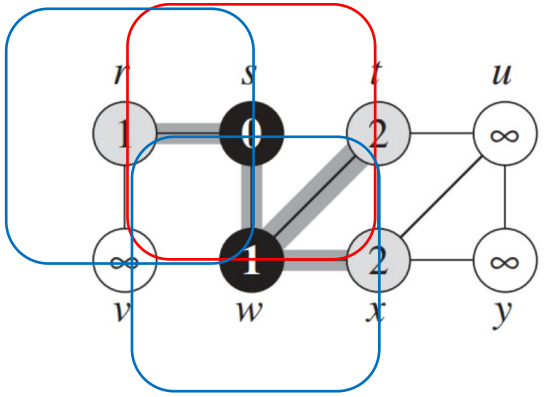
Graph Algorithms

- **Elementary Graph Algorithms**（图算法基础）
  - **Representations of Graphs**
  - **BFS，DFS**
  - **Sort Topologically**
- **Single-Source Shortest Paths**（最短路径问题）
  - **Finding shortest paths from a given source vertex to all other vertices.**
  - **Relaxation** （松弛）
- **All-Pairs Shortest Paths**（任意两点的最短路径问题）
  - **Computing shortest paths between every pair of vertices.**
- **Maximum Flow** （最大流）

- 数据结构
  - Representations: Adjacency-list, Adjacency-matrix
  - Queue, Priority Queue, Stack
- 策略（技术、方法）
  - Search: BFS，DFS
  - Backtrack
  - Recursion
  - Relaxation
  - Greedy
  - DP
- 算法
  - The Bellman-Ford
  - Topological sort
  - Dijkstra
- 分析
  - Aggregate analysis

22 Elementary Graph Algorithms（图算法基础）

- Representing （图的表示）
- Searching （图的遍历、搜索）
  - BFS：从一个点开始广度优先搜索，就是一个“圈”（周边信息），是局部子图结构（结构信息）。
  - DFS：沿一条路径一直遍历，遍历到头，然后回溯，连接信息。



Q

s					..
	w	r			..
		r	t	x	..

选定搜索的源s并入列

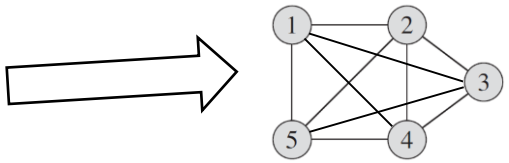
s出列，遍历s的邻接点并将这些邻接点入列

出列，遍历出列点的邻接点并入列

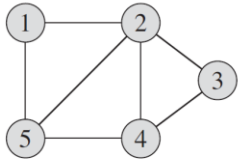
BFS 的示例

22.1 Representations of graphs

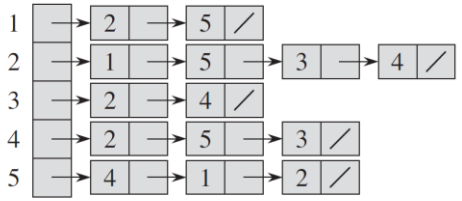
- A graph $G = (V, E)$
- Undirected and Directed （无向图、有向图）
- Dense graphs: $|E|$ is close to $|V|^2$ （稠密图）
- Sparse graphs: $|E|$ is much less than $|V|^2$ （稀疏图）



Undirected



(a)

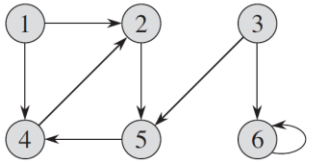


(b)

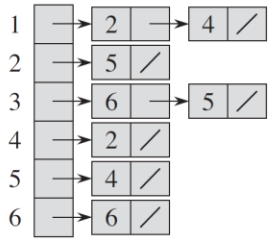
	1	2	3	4	5
1	0	1	0	0	1
2	1	0	1	1	1
3	0	1	0	1	0
4	0	1	1	0	1
5	1	1	0	1	0

(c)

Directed



(a)



(b)

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

(c)

a collection of adjacency lists

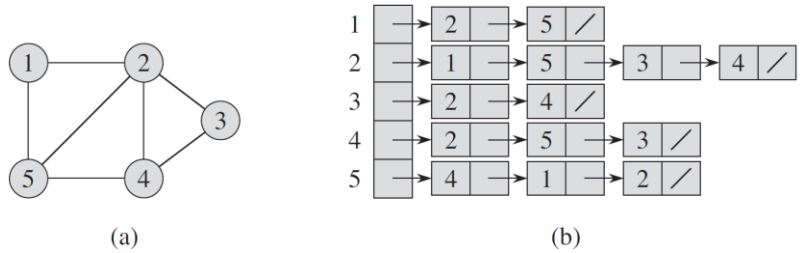
邻接表

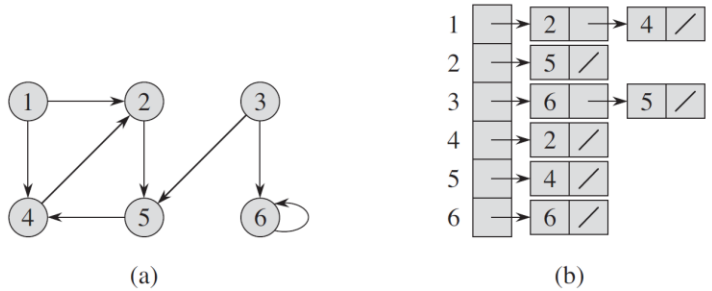
an adjacency matrix

邻接矩阵

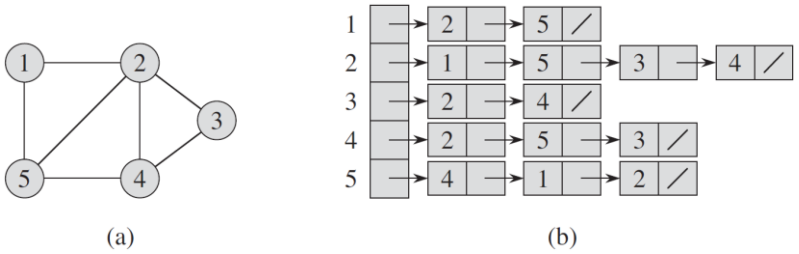
22.1 Representations of graphs

- *Adjacency-list representation*: an array *Adj* of $|V|$ lists, one for each vertex in V.
  邻接表：n 个元素的数组，每个元素表示一个顶点，代表链表的表头，跟该顶点相连的所有顶点构成一个链表
- The sum of the lengths of all the adjacency lists is:
  Directed graph, $|E|$; Undirected graph: $2*|E|$
  所有邻接表的总长度：
  有向图的长度为 $|E|$；无向图的长度为 $2*|E|$
- For both graph, the amount of memory is $\Theta(V+E)$.
  对有向图和无向图，空间复杂度都是 $\Theta(V+E)$



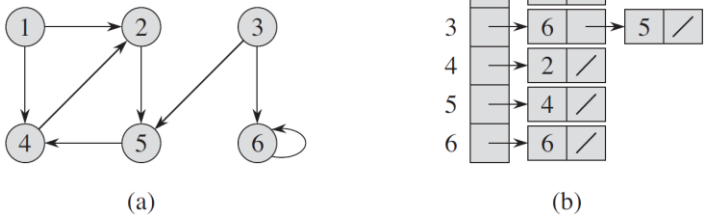


22.1 Representations of graphs



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

(c)



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

(c)

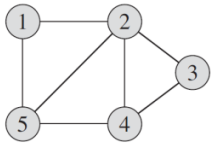
- *Adjacency-matrix representation*, 邻接矩阵:

  $|V| \times |V|$ matrix $A = (a_{ij})$, such that

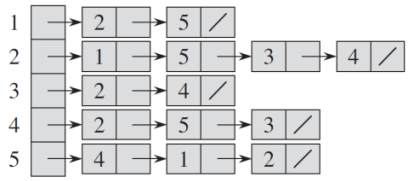
$$a_{ij} = \begin{cases} 1 & \text{if } (i, j) \in E , \\ 0 & \text{otherwise} . \end{cases}$$

- It requires $\Theta(V^2)$ memory.

22.1 Representations of graphs



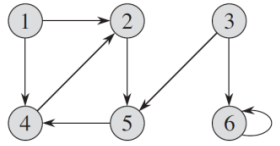
(a)



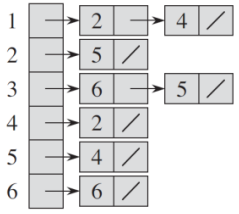
(b)

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

(c)



(a)



(b)

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

(c)

- Memory
  - *adjacency-list (AL)* : $\Theta(V+E)$
  - *adjacency-matrix (AM)* : $\Theta(V^2)$
- *AL*：对不太稠密的图，省空间；搜索高效；方便存储辅助信息（顶点的颜色、访问标记、父节点、子节点、..）
- *AM*：操作简单
- 算法具体实现，依赖于采用的数据结构、依赖于采用的程序设计语言。
- 如果不用STL（Standard Template Library），自己用C语言写，对链表和矩阵运算等操作需要掌握得比较熟练（程序设计和数据结构的基础要求高）。

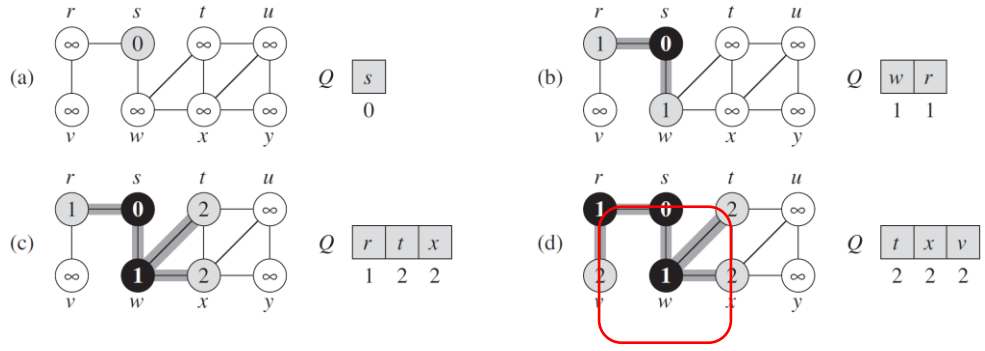
22.2 Breadth-first search（广度优先搜索）

```
BFS(G, s)   //从顶点s开始进行搜索
 1  for each vertex u ∈ G.V − {s}
 2      u.color = WHITE
 3      u.d = ∞
 4      u.π = NIL
 5  s.color = GRAY
 6  s.d = 0
 7  s.π = NIL
 8  Q = ∅
 9  ENQUEUE(Q, s)
10  while Q ≠ ∅
11      u = DEQUEUE(Q)
12      for each v ∈ G.Adj[u]
13          if v.color == WHITE
14              v.color = GRAY
15              v.d = u.d + 1
16              v.π = u
17              ENQUEUE(Q, v)
18      u.color = BLACK
```

初始化

Breadth-first: discovers all vertices at distance k from s first, then at distance k+1.

先遍历（搜索）离原点距离 k 的点，然后是 k+1 的点



关键步骤：

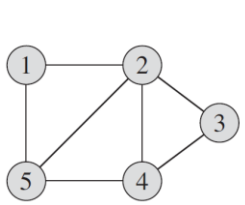
出列、入列

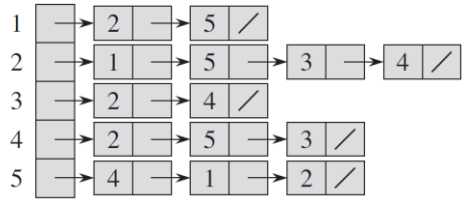
- 重要的数据结构：队列
- 复杂度分析：平摊分析

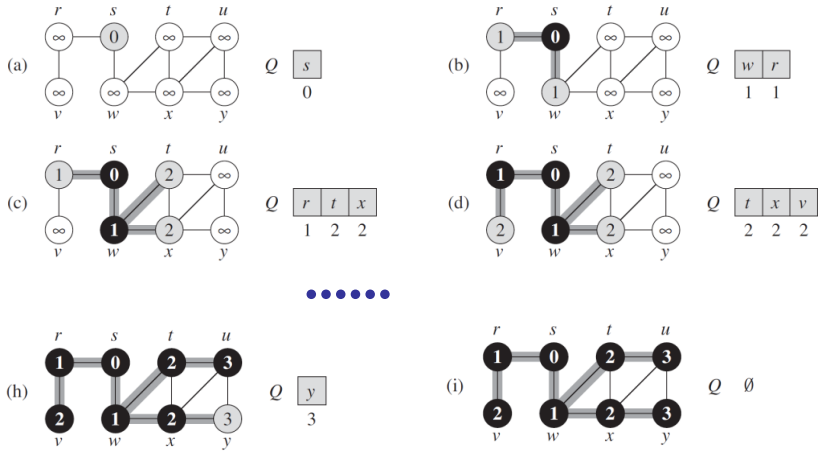
22.2 Breadth-first search - Running time?

Aggregate analysis: $O(V+E)$

聚集分析：每个顶点入队和出队各一次、
每条边检测一次





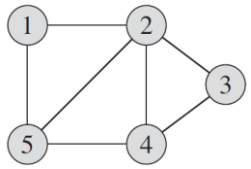



```
BFS(G, s)
 1  for each vertex u ∈ G.V − {s}
 2      u.color = WHITE
 3      u.d = ∞
 4      u.π = NIL
 5  s.color = GRAY
 6  s.d = 0
 7  s.π = NIL
 8  Q = ∅
 9  ENQUEUE(Q, s)
10  while Q ≠ ∅
11      u = DEQUEUE(Q)
12      for each v ∈ G.Adj[u]
13          if v.color == WHITE
14              v.color = GRAY
15              v.d = u.d + 1
16              v.π = u
17              ENQUEUE(Q, v)
18      u.color = BLACK
```

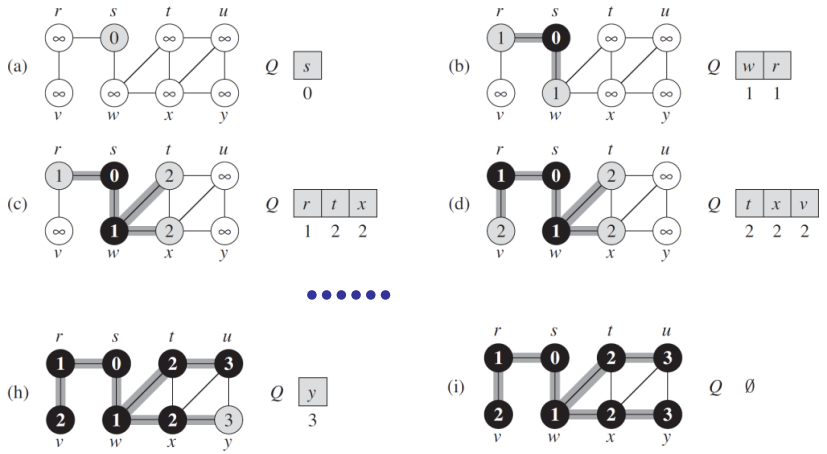
22.2 Breadth-first search - Running time?

Exercise： For adjacency-matrix representation, the running time of BFS?

以邻接矩阵表示图，BFS的计算时间?



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

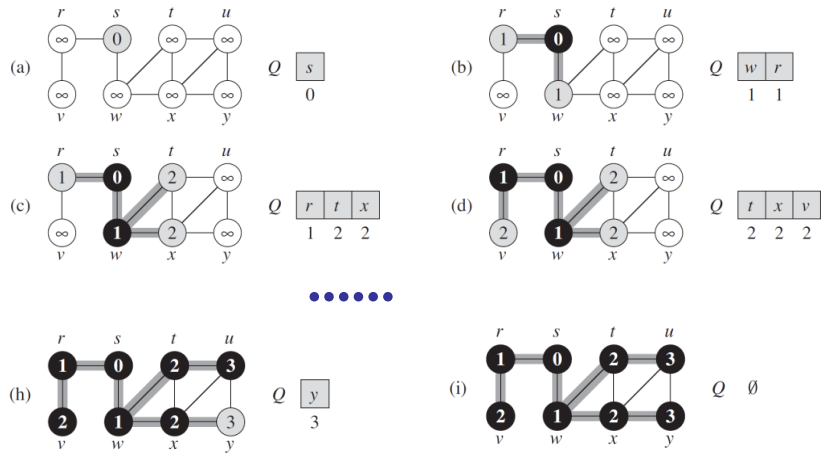



```
BFS(G, s)
 1  for each vertex u ∈ G.V − {s}
 2      u.color = WHITE
 3      u.d = ∞
 4      u.π = NIL
 5  s.color = GRAY
 6  s.d = 0
 7  s.π = NIL
 8  Q = ∅
 9  ENQUEUE(Q, s)
10  while Q ≠ ∅
11      u = DEQUEUE(Q)
12      for each v ∈ G.Adj[u]
13          if v.color == WHITE
14              v.color = GRAY
15              v.d = u.d + 1
16              v.π = u
17              ENQUEUE(Q, v)
18      u.color = BLACK
```

22.2 Breadth-first search - Analysis

During its execution, BFS discovers every vertex $v \in V$ that is reachable from the source s, and upon termination, $v.d = \delta(s, v)$ for all $v \in V$. ($\delta(s, v)$ is the length of a shortest path from s to v.)

BFS在搜索过程中会发现所有源点 s 可达的顶点 v，搜索停止时，搜索得到距离 $v.d$ 是 s 到 v 的最短路径 $\delta(s, v)$



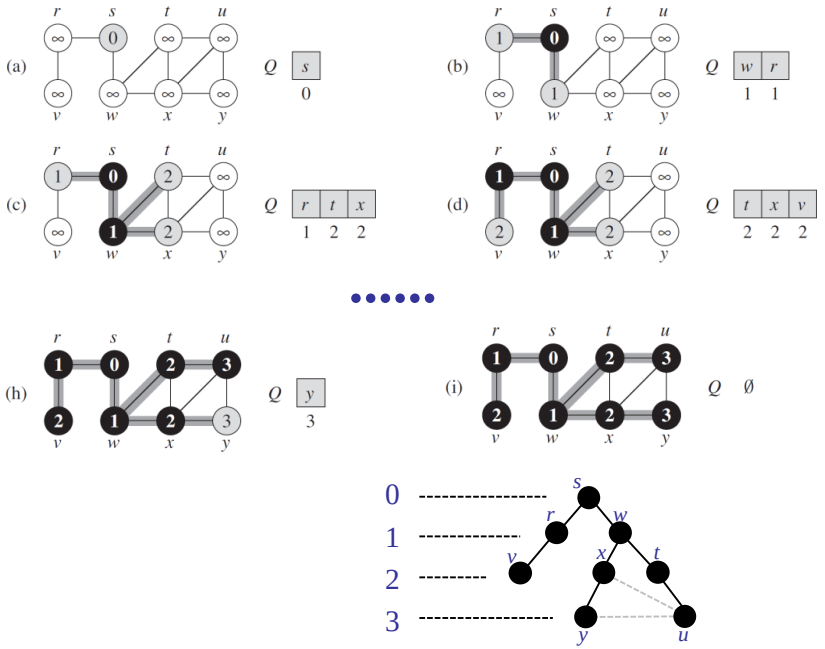

```
BFS(G, s)
 1  for each vertex u ∈ G.V − {s}
 2      u.color = WHITE
 3      u.d = ∞
 4      u.π = NIL
 5  s.color = GRAY
 6  s.d = 0
 7  s.π = NIL
 8  Q = ∅
 9  ENQUEUE(Q, s)
10  while Q ≠ ∅
11      u = DEQUEUE(Q)
12      for each v ∈ G.Adj[u]
13          if v.color == WHITE
14              v.color = GRAY
15              v.d = u.d + 1
16              v.π = u
17              ENQUEUE(Q, v)
18      u.color = BLACK
```

22.2 Breadth-first search - Breadth-first trees

- Define the predecessor subgraph of G as :
  $G_\pi = (V_\pi , E_\pi)$, where
  $V_\pi = \{ v \in V : v.\pi \neq \text{NIL} \} \cup \{s\}$ and
  $E_\pi = \{(v.\pi , v) : v \in V_\pi - \{s\} \}$.

G 的前驱子图 $G_\pi = (V_\pi , E_\pi)$:
V_π，从源点 s 可达的顶点集合
E_π，BFS过程中的边
对任意顶点 v，G_π 包括从 s 到 v 的一条唯一简单路径（也是最短路径）。G_π 没有环路，此图是一棵树，称为广度优先树。

- BFS constructs π so that the predecessor subgraph G_π is a breadth-first tree.
- PRINT-PATH prints out the vertices on a shortest path $s \rightsquigarrow v$.




```
PRINT-PATH(G, s, v)
1  if v == s
2      print s
3  elseif v.π == NIL
4      print "no path from" s "to" v "exists"
5  else PRINT-PATH(G, s, v.π)
6      print v
```

求 s 到 v 的最短路径，依序把路径上的点输出

22.2 Breadth-first search - Application

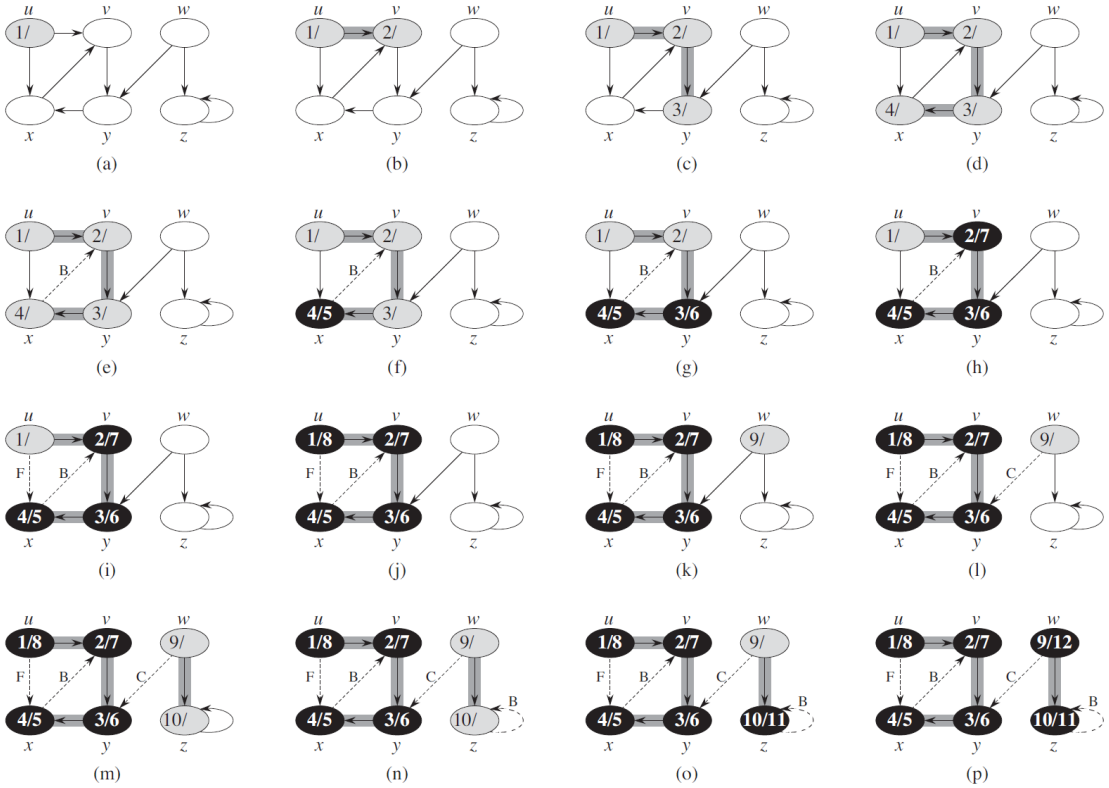
计算地铁站间的最短路径

Depth-first search has been widely used since the late 1950s, especially in artificial intelligence programs.

22.3 Depth-first search （深度优先搜索）

DFS: explores edges out of the most recently discovered vertex that still has unexplored edges leaving it. Once all of v's edges have been explored, "backtracks".

从最新发现的顶点 v 继续出发递归搜索，当从 v 出发的所有边都搜索完时，回溯。

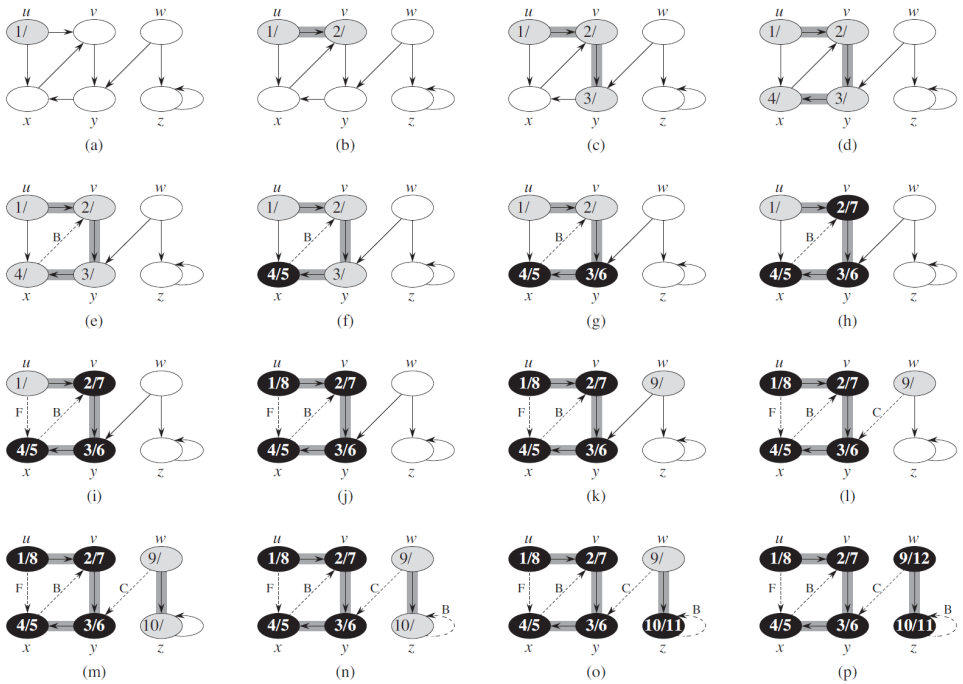



```
DFS(G)
1  for each vertex u ∈ G.V
2      u.color = WHITE
3      u.π = NIL
4  time = 0
5  for each vertex u ∈ G.V
6      if u.color == WHITE
7          DFS-VISIT(G, u)

DFS-VISIT(G, u)
1  time = time + 1
2  u.d = time //发现顶点u的时间
3  u.color = GRAY
4  for each v ∈ G.Adj[u]
5      if v.color == WHITE
6          v.π = u
7          DFS-VISIT(G, v)
8  u.color = BLACK
9  time = time + 1
10 u.f = time
```

递归是核心

22.3 Depth-first search （深度优先搜索）

- 顶点初始化为白色，被发现（搜索到）是置为灰色，当一个顶点的邻接点都被搜索完后，该顶点置为黑色。
- 第一次搜索到某个顶点 v 时，顶点 v 置灰，并打上发现时间戳 $v.d$；当检查到 v 的所有邻接点都被访问时，v 置黑，并打上结束时间戳 $v.f$。
- 顶点 u 在发现时间 $u.d$ 前是白色，$u.d$ 和 $u.f$ 之间是灰色，$u.f$ 后为黑色。




```
DFS(G)
1   for each vertex u ∈ G.V
2       u.color = WHITE
3       u.π = NIL
4   time = 0
5   for each vertex u ∈ G.V
6       if u.color == WHITE
7           DFS-VISIT(G, u)

DFS-VISIT(G, u)
1   time = time + 1
2   u.d = time //发现顶点u的时间
3   u.color = GRAY
4   for each v ∈ G.Adj[u]
5       if v.color == WHITE
6           v.π = u
7           DFS-VISIT(G, v)
8   u.color = BLACK
9   time = time + 1
10  u.f = time
```

递归是核心

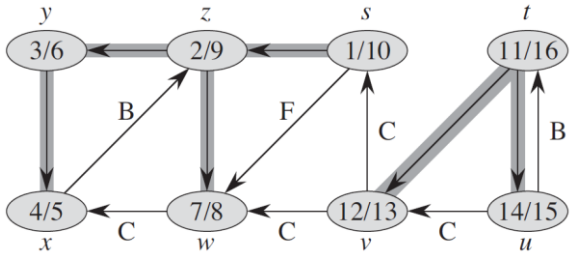
22.3 Depth-first search (*AM* based example)

基于邻接矩阵的示例：以邻接矩阵为例说明遍历过程（邻接表原理相同），当一个顶点的邻接点都被访问完后（不能再往前搜索），该顶点置为黑色。算法的要点是递归实现。

每个顶点 v 的父节点 $v.\pi$ 也可以记录在这里，因显示不全，此处略。

	s 0/0	z 0/0	y 0/0	x 0/0	w 0/0	…
s		1			1	
z			1		1	
y				1		
x		1				
w				1		
…						

初始化




```
DFS(G)
1  for each vertex u ∈ G.V
2      u.color = WHITE
3      u.π = NIL
4  time = 0
5  for each vertex u ∈ G.V
6      if u.color == WHITE
7          DFS-VISIT(G, u)

DFS-VISIT(G, u)
1  time = time + 1
2  u.d = time
3  u.color = GRAY
4  for each v ∈ G.Adj[u]
5      if v.color == WHITE
6          v.π = u
7          DFS-VISIT(G, v)
8  u.color = BLACK
9  time = time + 1
10 u.f = time
```

22.3 Depth-first search (*AM* based example)

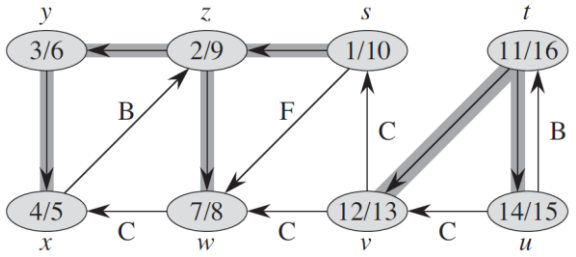
基于邻接矩阵的示例：以邻接矩阵为例说明遍历过程（邻接表原理相同），当一个顶点的邻接点都被访问完后（不能再往前搜索），该顶点置为黑色。算法的要点是递归实现。

	s 0/0	z 0/0	y 0/0	x 0/0	w 0/0	…
s		1			1	
z			1		1	
y				1		
x		1				
w				1		
…						

初始化

	s 1/0	z 0/0	y 0/0	x 0/0	w 0/0	…
s		1 ✓			1	
z			1		1	
y				1		
x		1				
w				1		
…						

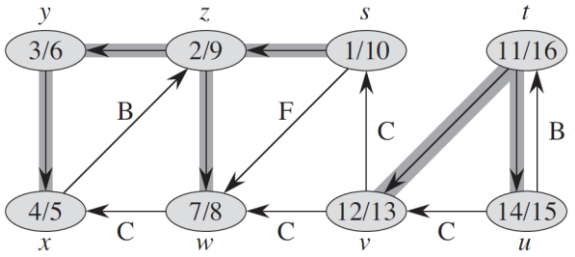
第1次DFS




```
DFS(G)
1  for each vertex u ∈ G.V
2      u.color = WHITE
3      u.π = NIL
4  time = 0
5  for each vertex u ∈ G.V
6      if u.color == WHITE
7          DFS-VISIT(G, u)

DFS-VISIT(G, u)
1  time = time + 1
2  u.d = time
3  u.color = GRAY
4  for each v ∈ G.Adj[u]
5      if v.color == WHITE
6          v.π = u
7          DFS-VISIT(G, v)
8  u.color = BLACK
9  time = time + 1
10 u.f = time
```

22.3 Depth-first search (*AM* based example)

基于邻接矩阵的示例：以邻接矩阵为例说明遍历过程（邻接表原理相同），当一个顶点的邻接点都被访问完后（不能再往前搜索），该顶点置为黑色。算法的要点是递归实现。



	s 0/0	z 0/0	y 0/0	x 0/0	w 0/0	…
s		1			1	
z			1		1	
y				1		
x		1				
w				1		
…						

初始化

	s 1/0	z 0/0	y 0/0	x 0/0	w 0/0	…
s		1 ✓			1	
z			1		1	
y				1		
x		1				
w				1		
…						

第1次DFS

	s 1/0	z 2/0	y 0/0	x 0/0	w 0/0	…
s		1 ✓			1	
z			1 ✓		1	
y				1		
x		1				
w				1		
…						

第2次DFS

	s 1/0	z 2/0	y 3/0	x 0/0	w 0/0	…
s		1 ✓			1	
z			1 ✓		1	
y				1 ✓		
x		1				
w				1		
…						

第3次DFS

```
DFS(G)
1  for each vertex u ∈ G.V
2      u.color = WHITE
3      u.π = NIL
4  time = 0
5  for each vertex u ∈ G.V
6      if u.color == WHITE
7          DFS-VISIT(G, u)

DFS-VISIT(G, u)
1  time = time + 1
2  u.d = time
3  u.color = GRAY
4  for each v ∈ G.Adj[u]
5      if v.color == WHITE
6          v.π = u
7          DFS-VISIT(G, v)
8  u.color = BLACK
9  time = time + 1
10 u.f = time
```

	s 0/0	z 0/0	y 0/0	x 0/0	w 0/0	…
s		1			1	
z			1		1	
y				1		
x		1				
w				1		
…						

初始化

	s 1/0	z 0/0	y 0/0	x 0/0	w 0/0	…
s		1 ✓			1	
z			1		1	
y				1		
x	1					
w				1		
…						

第1次DFS

	s 1/0	z 2/0	y 0/0	x 0/0	w 0/0	…
s		1 ✓			1	
z			1 ✓		1	
y				1		
x	1					
w				1		
…						

第2次DFS

	s 1/0	z 2/0	y 3/0	x 0/0	w 0/0	…
s		1 ✓			1	
z			1 ✓		1	
y				1 ✓		
x		1				
w				1		
…						

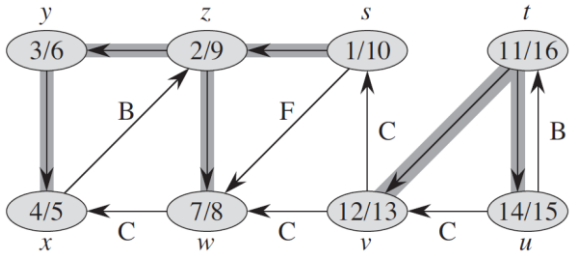
第3次DFS

	s 1/0	z 2/0	y 3/0	x 4/0	w 0/0	…
s		1 ✓			1	
z			1 ✓		1	
y				1 ✓		
x		1 ✓				
w				1		
…						

第4次DFS，不能再DFS

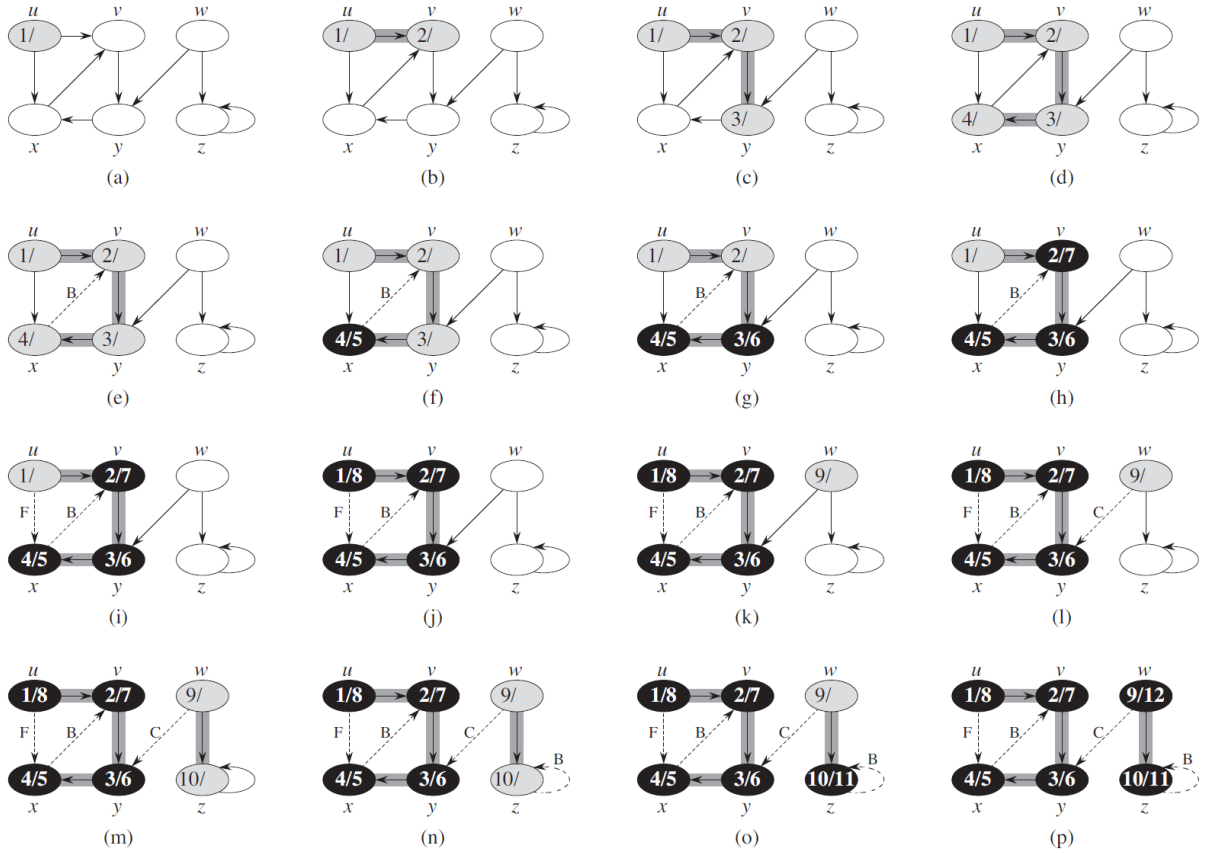
	s 1/0	z 2/0	y 3/0	x 4/5	w 0/0	…
s		1 ✓			1	
z			1 ✓		1	
y				1 ✓		
x		1 ✓				
w				1		
…						

x的邻接元素遍历完，回溯...




```
DFS(G)
1  for each vertex u ∈ G.V
2      u.color = WHITE
3      u.π = NIL
4  time = 0
5  for each vertex u ∈ G.V
6      if u.color == WHITE
7          DFS-VISIT(G, u)

DFS-VISIT(G, u)
1  time = time + 1
2  u.d = time
3  u.color = GRAY
4  for each v ∈ G.Adj[u]
5      if v.color == WHITE
6          v.π = u
7          DFS-VISIT(G, v)
8  u.color = BLACK
9  time = time + 1
10 u.f = time
```

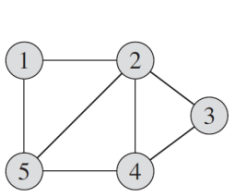
Running time?

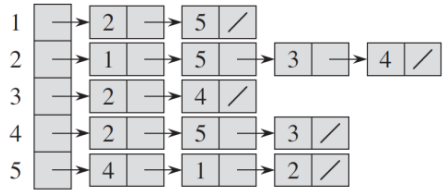



```
DFS(G)
1  for each vertex u ∈ G.V
2      u.color = WHITE
3      u.π = NIL
4  time = 0
5  for each vertex u ∈ G.V
6      if u.color == WHITE
7          DFS-VISIT(G, u)

DFS-VISIT(G, u)
1  time = time + 1
2  u.d = time
3  u.color = GRAY
4  for each v ∈ G.Adj[u]
5      if v.color == WHITE
6          v.π = u
7          DFS-VISIT(G, v)
8  u.color = BLACK
9  time = time + 1
10 u.f = time
```

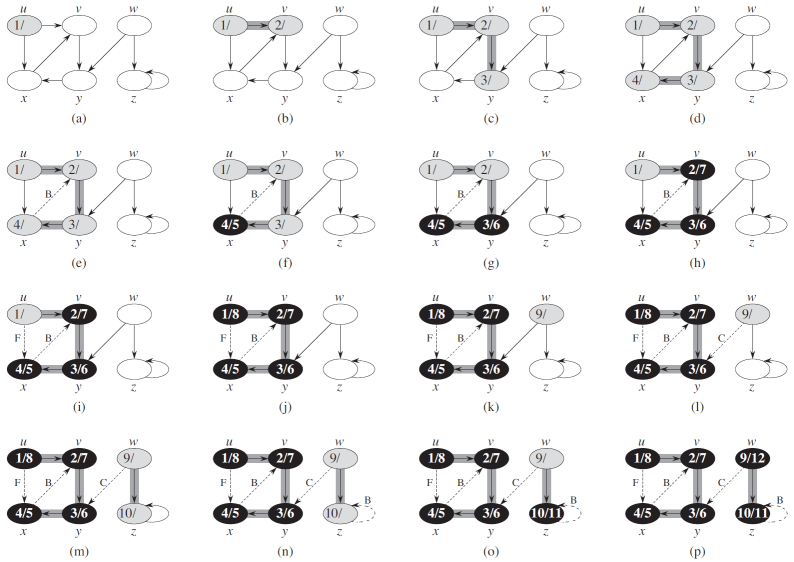
22.3 Depth-first search - Running time?

Aggregate analysis, 聚集分析：顶点为白色时，才会作为递归子树的“源点”进行递归调用（调用后变为非白色，不可逆）；对每个顶点，其邻接表的每条边遍历一次。顶点数 V，所有边数 E。





$O(V+E)$




```
DFS(G)
1  for each vertex u ∈ G.V
2      u.color = WHITE
3      u.π = NIL
4  time = 0
5  for each vertex u ∈ G.V
6      if u.color == WHITE
7          DFS-VISIT(G, u)

DFS-VISIT(G, u)
1  time = time + 1
2  u.d = time
3  u.color = GRAY
4  for each v ∈ G.Adj[u]
5      if v.color == WHITE
6          v.π = u
7          DFS-VISIT(G, v)
8  u.color = BLACK
9  time = time + 1
10 u.f = time
```

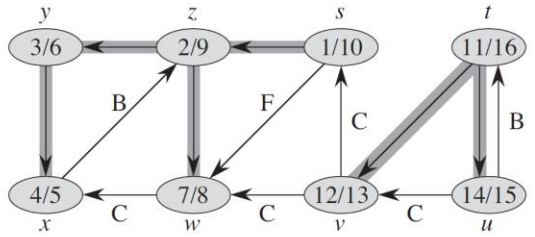
22.3 Depth-first search - Properties of DFS

顶点 u 的发现时间 $u.d$ 和结束时间 $u.f$ 具有的嵌套的加括号结构。

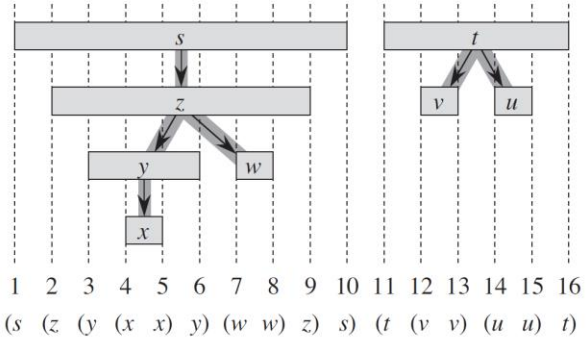
- “(u” : 左括号表示顶点 u 的发现时间
- “u)” : 右 → 结束时间
- 如果顶点 u 发现 v，则 v 的发现和结束时间区间嵌套在 u 的时间区间内部

$$\underrightarrow{(u\ ..\ (v\ ..\ v)\ ..\ u)}_{time}$$

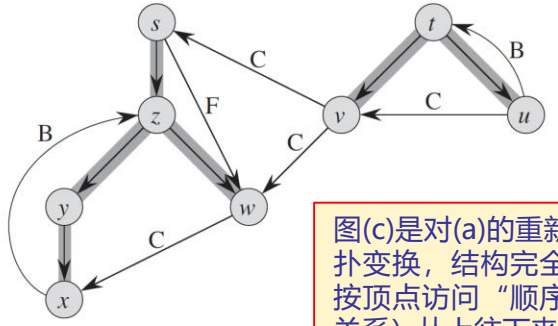
(a)



(b)



(c)



图(c)是对(a)的重新画法（拓扑变换，结构完全一样），按顶点访问“顺序”（辈分关系）从上往下来画。

22.3 Depth-first search - Properties of DFS

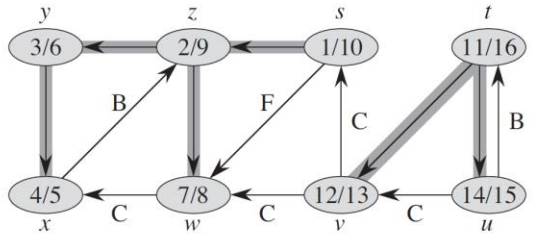
Theorem 22.7 (Parenthesis theorem) : 对一个图 G 的任意 DFS，任意两个顶点 u 和 v，有如下条件之一满足：

- 时间区间 $[u.d, u.f]$ 和 $[v.d, v.f]$ 不相交，则 u 和 v 在深度优先搜索森林中无亲缘（辈分）关系。
- 时间区间 $[v.d, v.f]$ 包含在 $[u.d, u.f]$ 内部，则 v 在深度优先搜索森林中是顶点 u 的后代。

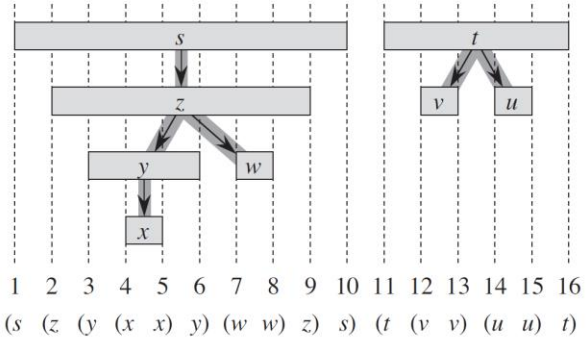
$(u\ ..\ u)\quad (v\ ..\ v)$ → u 和 v 无亲缘关系

$(u\ ..\ (v\ ..\ v)\ ..\ u)$ → v 是 u 的后代

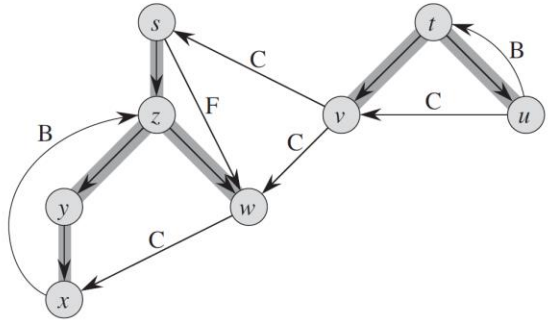
(a)



(b)



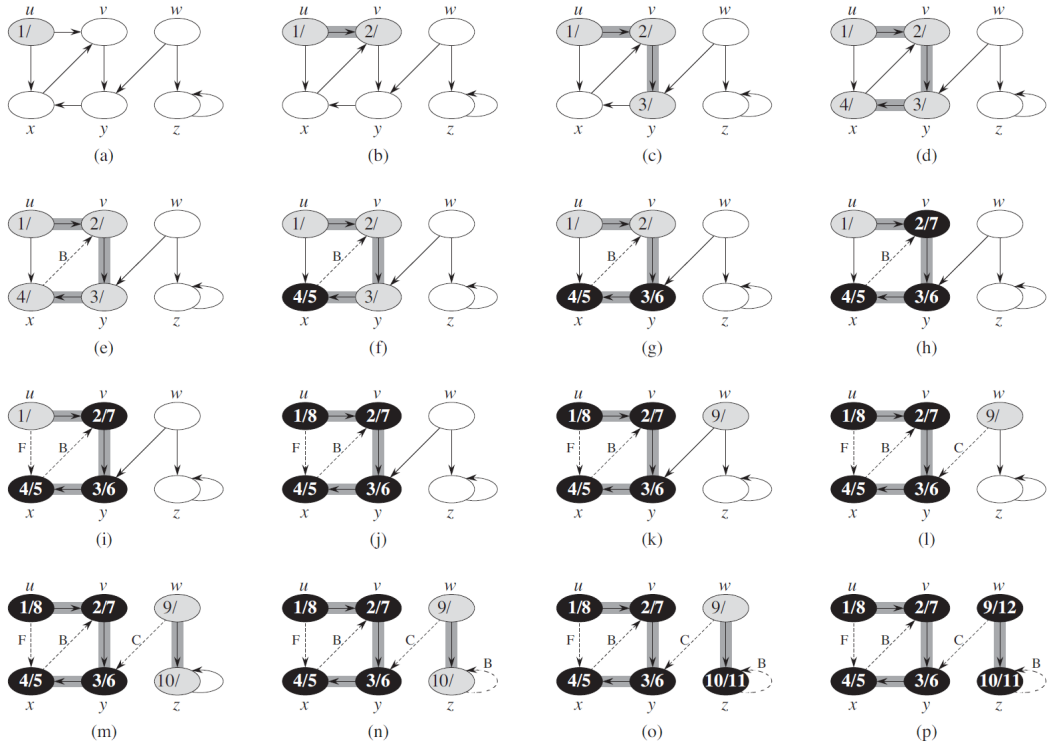
(c)



22.3 Depth-first search - Properties of DFS

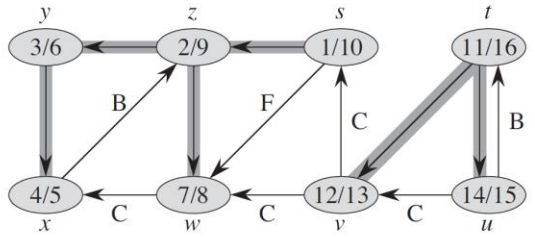
Exercise (22.3-7): Rewrite the procedure DFS, using a stack to eliminate recursion.

练习：采用栈操作，不用递归，重写 DFS.



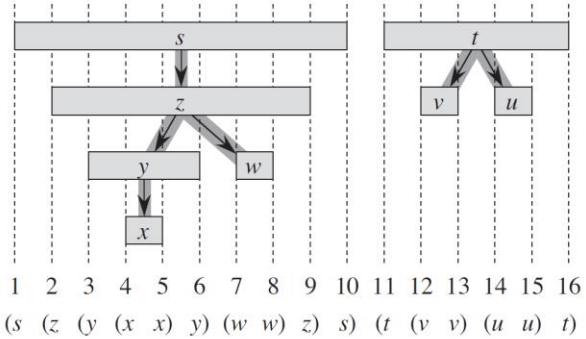

```
DFS(G)
1  for each vertex u ∈ G.V
2      u.color = WHITE
3      u.π = NIL
4  time = 0
5  for each vertex u ∈ G.V
6      if u.color == WHITE
7          DFS-VISIT(G, u)

DFS-VISIT(G, u)
1  time = time + 1
2  u.d = time
3  u.color = GRAY
4  for each v ∈ G.Adj[u]
5      if v.color == WHITE
6          v.π = u
7          DFS-VISIT(G, v)
8  u.color = BLACK
9  time = time + 1
10 u.f = time
```

(a)

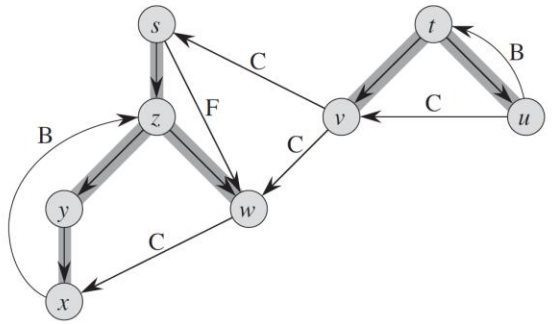


栈性质：先发现后结束

(b)



(c)



22.3 Depth-first search - Classification of edges

1. **Tree edges**: edge (u, v) is a tree edge if v was first discovered by exploring edge (u, v).
在探索边 (u, v) 的过程中 v 被首次发现，则称 (u, v) 是树边（粗线）。

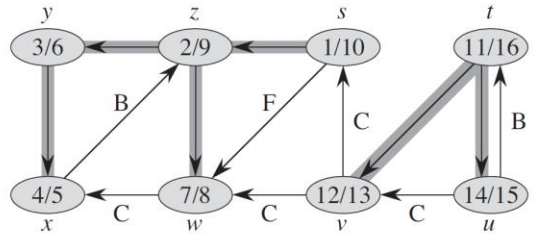
2. **Back edges**: 深度优先树中（粗线边），连接顶点 u 到其前辈顶点 v 的边 (u, v)，被称为后向边（晚辈指向前辈的边）。

3. **Forward edges**: 深度优先树中（粗线边），前辈指向到晚辈的边（树边除外），被称为前向边。

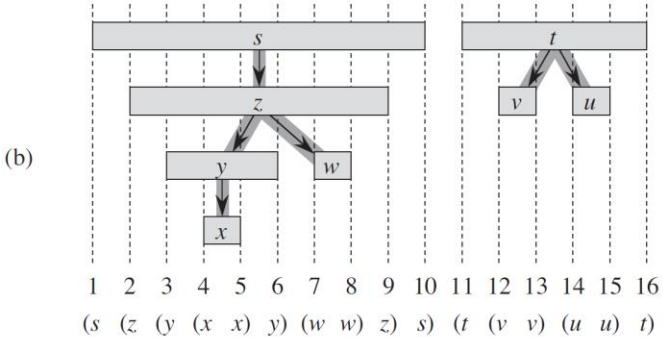
4. **Cross edges**: 除上述三类边之外的其他边，被称为交叉边。

练习题：对无向图进行DFS，不会出现前向边和交叉边？试举例说明。

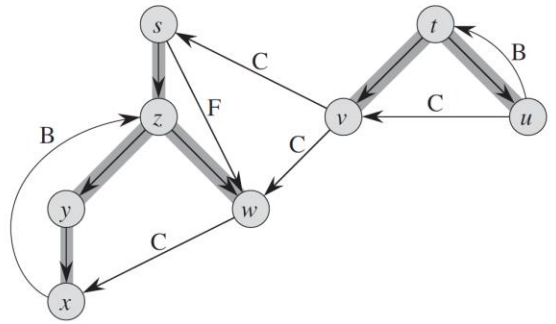
(a)



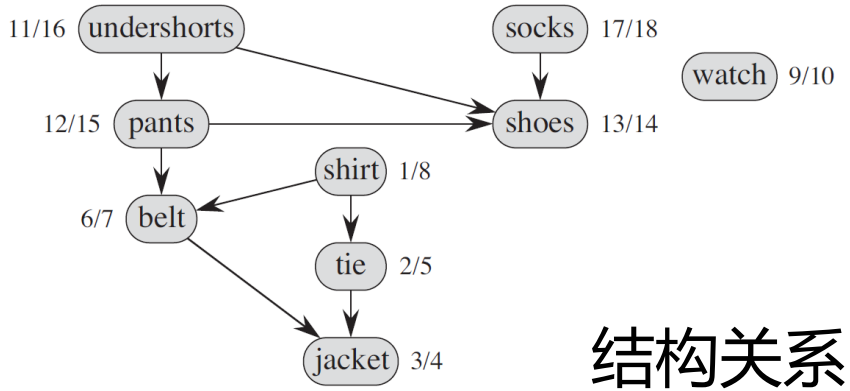
(b)



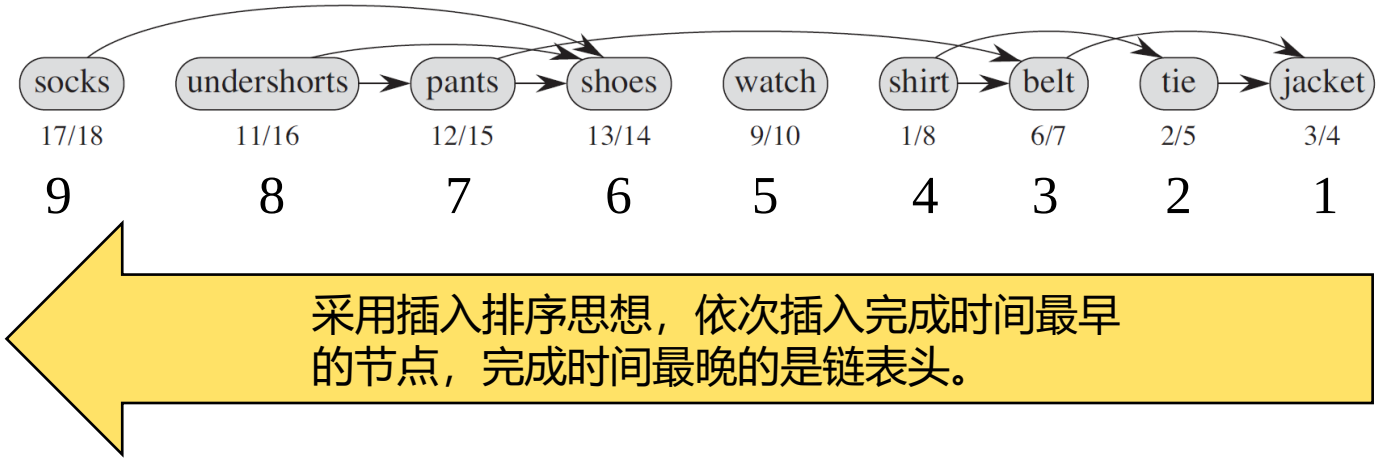
(c)



22.4 Topological sort （拓扑排序）



结构关系、连接关系不变，观察视角变化了，从左到右表示了各个活动依赖（先后）关系。



DAG: 有向无环图（有向无环路图）

拓扑排序算法：按照结束时间，从大到小排序。算法中采用插入排序思想，依次插入完成时间最早的节点（作为链表头），算法结束后，完成时间最晚的是链表头。

22.4 Topological sort

```
TOPOLOGICAL-SORT(G)
1  call DFS(G) to compute finishing times v.f for each vertex v
2  as each vertex is finished, insert it onto the front of a linked list
3  return the linked list of vertices
```

- DFS 可以对一个有向无环图(dag)进行拓扑排序（给出图的搜索顺序）。
- 拓扑排序的常见应用：任务安排、特别是并行计算。
- 拓扑排序：在一个有向无环图(dag)中，如果包括边 (u, v)，则 u 先执行，v 后执行。
- 拓扑排序算法：DFS后，按照顶点遍历的“结束”时间，从大到小排序?

拓扑序：按结束时间从大到小排序? 直觉上看，先进后出：先进（后出），结束时间晚（发现【开始】时间早），先执行，再执行其他顶点。没有直接关联的顶点，可并行计算（或先后无关）。

1	2	3	4	5	6	7	8	9	10	..							

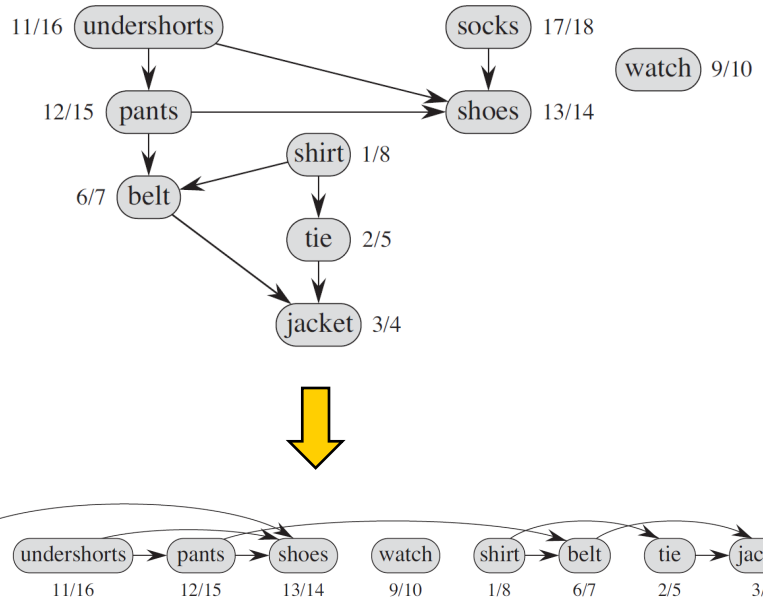
tie 打领带　belt 系腰带

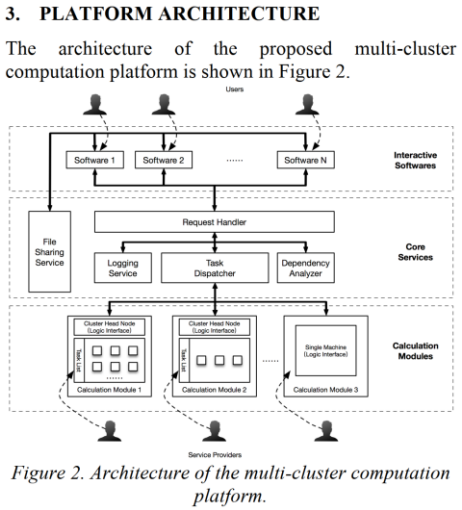
shirt 穿衬衫

穿衬衫→系腰带→打领带。当然，系腰带前还需要先传裤子（图略）。

22.4 Topological sort

- 许多应用程序使用dag来指示事件之间的优先级。
- 一些应用：最短路径（先拓扑排序，然后需要对每一条边进行松弛操作，后文介绍），并行计算, …





3. PLATFORM ARCHITECTURE

The architecture of the proposed multi-cluster computation platform is shown in Figure 2.

Figure 2. Architecture of the multi-cluster computation platform.

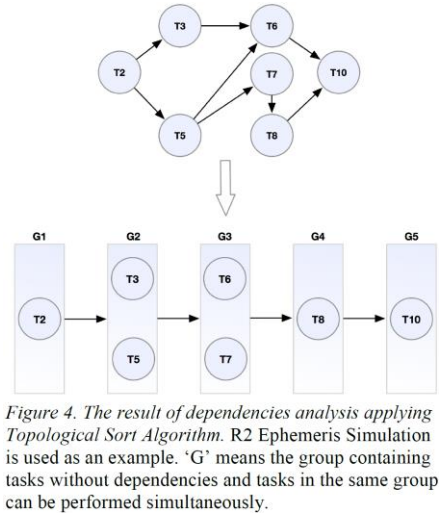
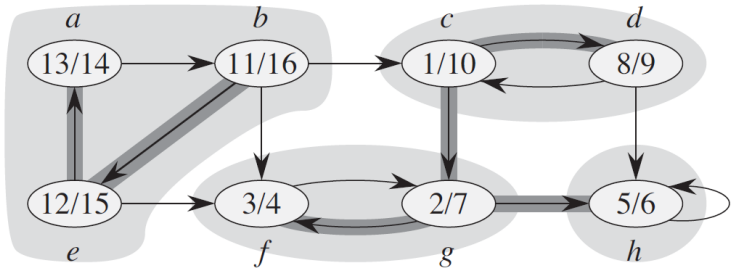


Figure 4. The result of dependencies analysis applying Topological Sort Algorithm. R2 Ephemeris Simulation is used as an example. 'G' means the group containing tasks without dependencies and tasks in the same group can be performed simultaneously.

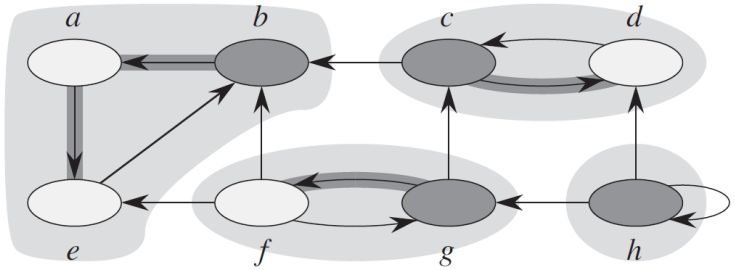
You Song, etc., A Microservice-based Multi-cluster Computation Platform For Space Mission Design, 8th International Systems & Concurrent Engineering for Space Applications Conference, Glasgow, UK, 2018

*22.5 Strongly connected components

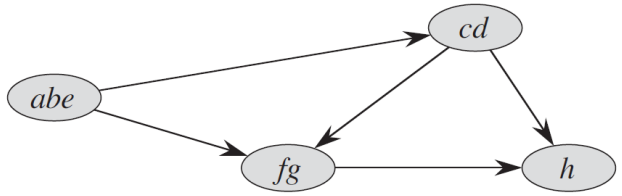
(a)



(b)



(c)

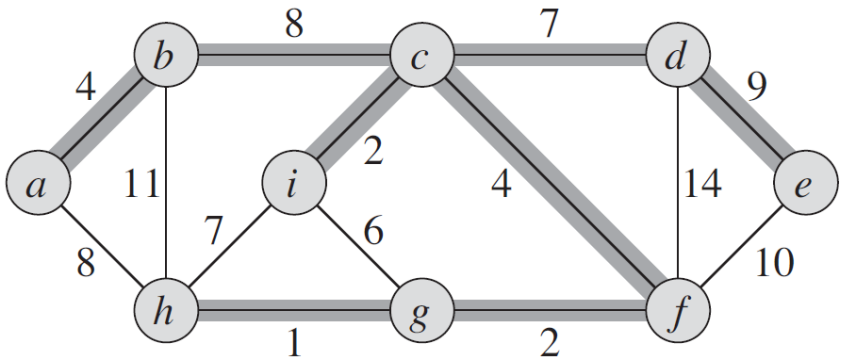


两次使用DFS

应用，复杂图的降维

复杂应用中，BFS和DFS经常结合一起用。

*23 Minimum Spanning Trees



Application: Electronic circuit designs （电子电路设计）

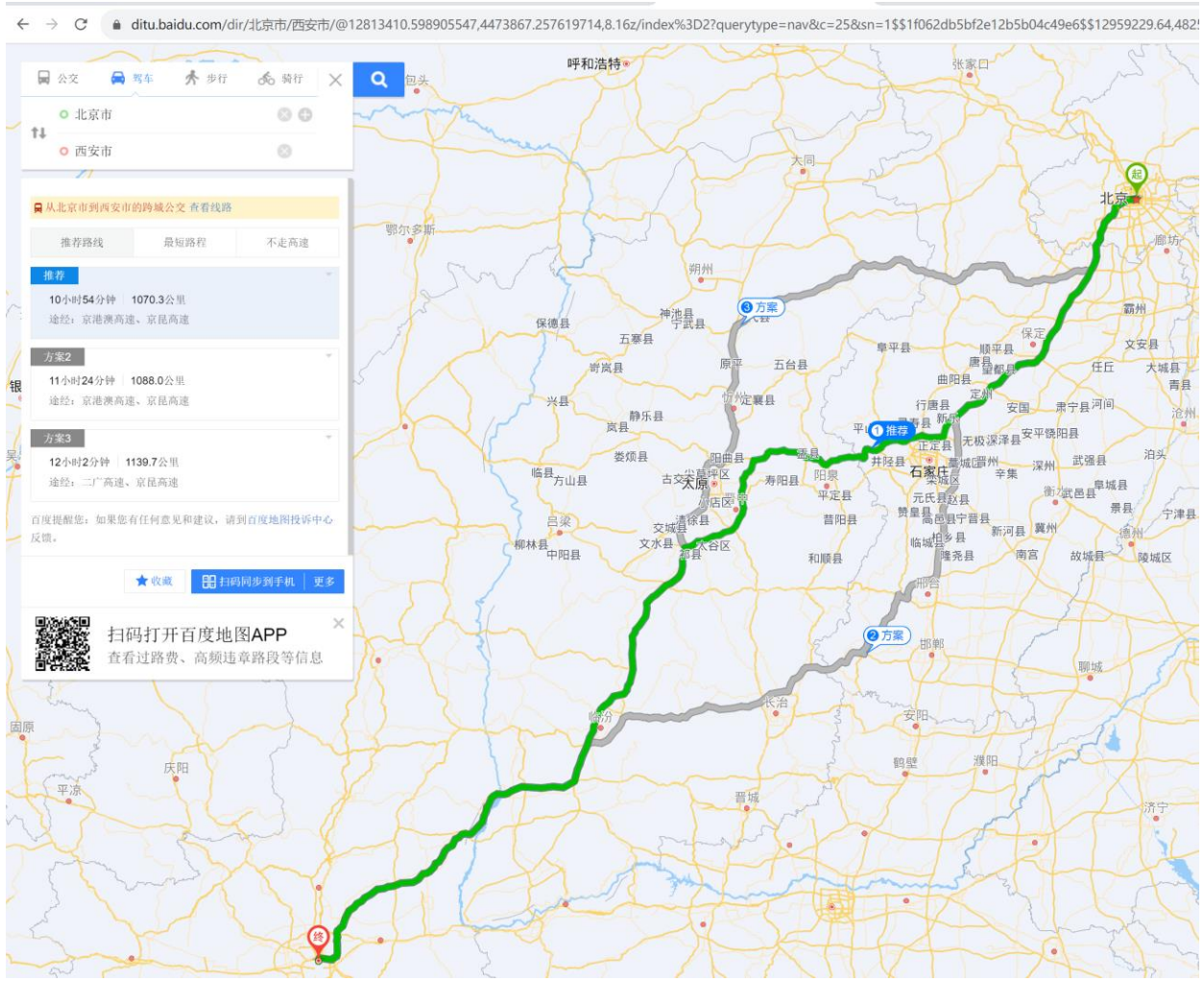
Algorithms: Kruskal , Prim （贪心算法的经典应用）

- Kruskal（**鲁斯卡尔**）算法主要是针对边展开，边数少时效率会非常高，对稀疏图有很大的优势
- Prim （**普里姆**）算法对于稠密图，即边数非常多的情况会好一些

24 Single-Source Shortest Paths

Find the shortest (fastest, cheapest) possible route from City X to Y, e.g., Beijing to Xian?

求城市 X 到 Y 的距离最短（或用时最少、或收费最少）的路线?



24 Single-Source Shortest Paths

Shortest-paths problem : given a weighted, directed graph $G = (V, E)$, with weight function $w : E \rightarrow \mathrm{R}$ mapping edges to real-valued weights. The weight $w(p)$ of path $p = < v_0 , \ldots , v_k >$ is the sum of the weights of its constituent edges:

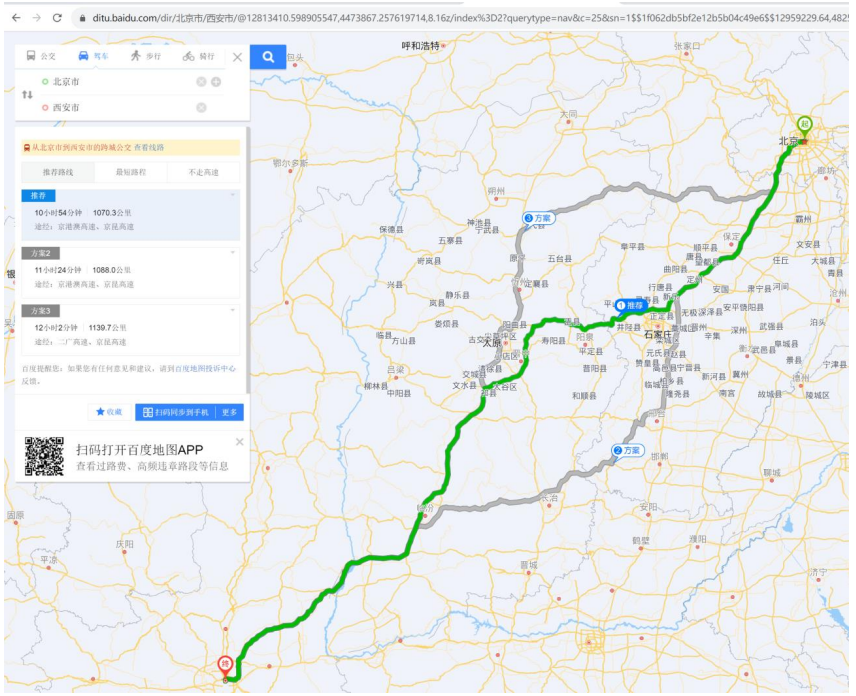
$$w(p) = \sum_{i=1}^{k} w(v_{i-1}, v_i) .$$

Define the **shortest-path weight** $\delta(u, v)$ from u to v by

$$\delta(u, v) = \begin{cases} \min\{w(p) : u \overset{p}{\rightsquigarrow} v\} & \text{if there is a path from } u \text{ to } v , \\ \infty & \text{otherwise} . \end{cases}$$

已知一个加权有向图，每条边有一个实数值的权值，一条路径 $p = < v_0 , \ldots , v_k >$ 的权值是该路径上组成边的权值和 $w(p)$。顶点 u 到 v 的最短路径，

$\delta(u, v)$：u 到 v 所有路径中权值最小的那条路径。



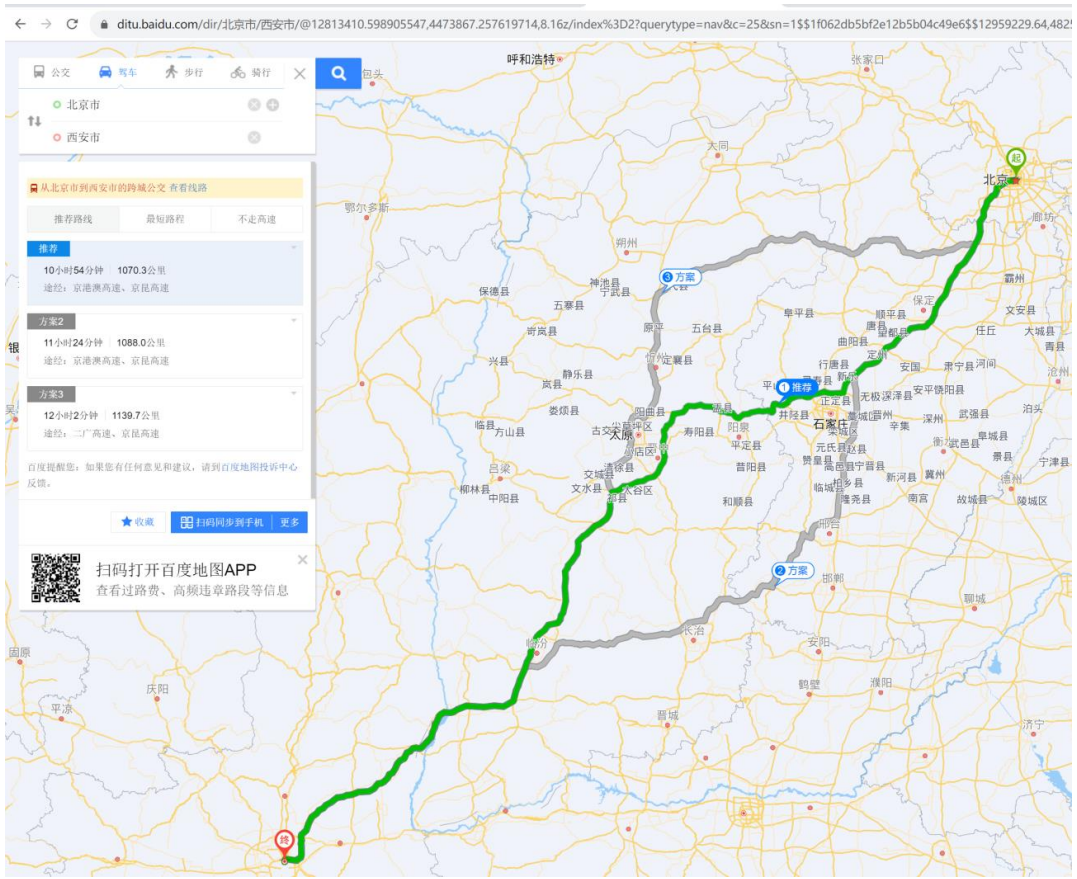
24 Single-Source Shortest Paths

- BFS is a shortest-paths algorithm that works on unweighted graphs.
  BFS可用于求解一个无权图的最短路径

- **How about weighted?**

24 Single-Source Shortest Paths

- **Single-source shortest-paths problem**
- **Single-destination shortest-paths problem**
- **Single-pair shortest-path problem**
- **All-pairs shortest-paths problem**



四类最短路径问题，其特点：

- ✓ 单源与单目标最短路径算法原理相同
- ✓ 某两点最短路径是单源最短路径的特殊情况
- ✓ 任意两点最短路径，也是单源最短路径的特殊情况（运行单源最短路径 n 次即可）

24 Single-Source Shortest Paths

- Optimal substructure: Shortest-paths algorithms typically rely on the property that a shortest path between two vertices contains other shortest paths within it.

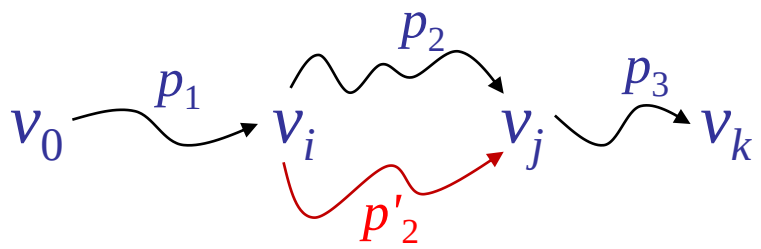
  最短路径问题具有最优子结构性质

- **Lemma 24.1** (Subpaths of shortest paths are shortest paths)

  Given a weighted, directed graph $G = (V, E)$ with weight function $w : E \to \mathrm{R}$, let $p = < v_0 , v_1 , \ldots , v_k >$ be a shortest path from vertex v_0 to vertex v_k and, for any i and j such that $0 \le i \le j \le k$, let $p_{ij} = < v_i , v_{i+1} , \ldots , v_j >$ be the subpath of p from vertex v_i to vertex v_j . Then, p_{ij} is a shortest path from v_i to v_j .

Proof …

$$v_0 \overset{p_{0i}}{\rightsquigarrow} v_i \overset{p_{ij}}{\rightsquigarrow} v_j \overset{p_{jk}}{\rightsquigarrow} v_k$$



$p = p_1 p_2 p_3$ is shortest,
that is, $p = \delta(v_0 , v_k)$

⇨

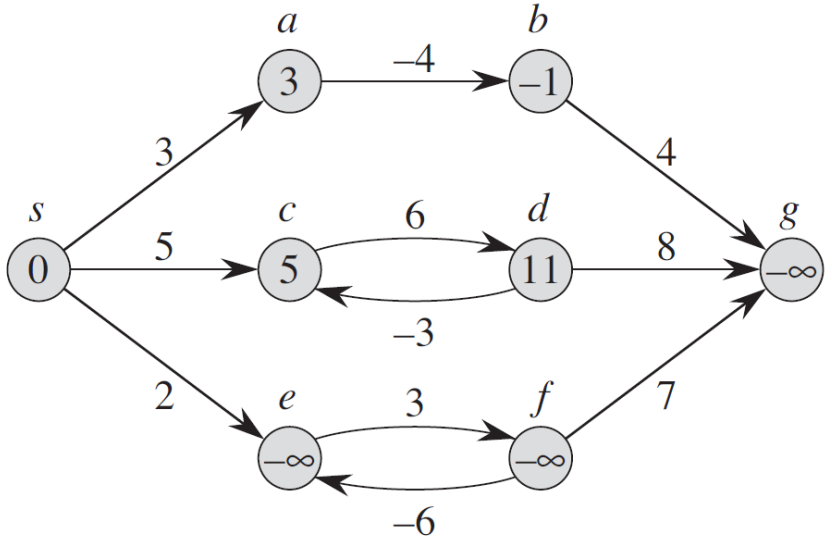
p_2 is shortest,
that is, $p_2 = \delta(v_i , v_j)$

Cycles: Can a shortest path contain a cycle?

No cycle: It cannot contain a negative-weight cycle, nor can it contain a positive-weight cycle.

最短路径不会包括环（但输入的图中可以有环路）：

- 如果有负环，则负环可以一圈一圈地加，最短路径可为无穷小；
- 如果有正环，则去掉一个环后能够得到更短的路径。

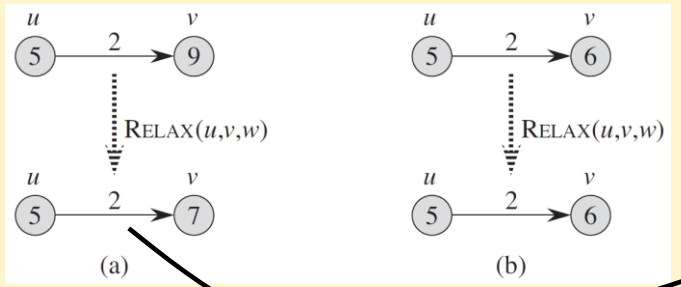


24 Single-Source Shortest Paths

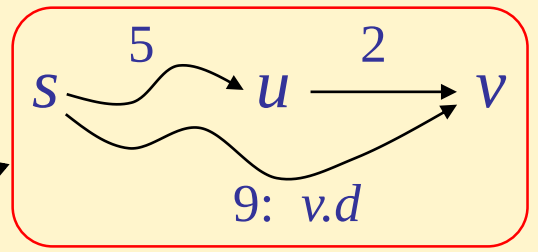
Relaxation (actually Tight)**:** 对每条边 (u, v) 执行松弛操作，能让源点 s 到 v 的距离 $v.d$ 减少（不增）。

最短路径问题最重要的操作！

```
INITIALIZE-SINGLE-SOURCE(G, s)
1  for each vertex v ∈ G.V
2      v.d = ∞
3      v.π = NIL
4  s.d = 0
```



对边 $u \to v$ 进行松弛，$v.d$ 从9变为7，即 dis(s, v) 变小。



Relaxation

```
RELAX(u, v, w)
1  if v.d > u.d + w(u, v)
2      v.d = u.d + w(u, v)
3      v.π = u
```

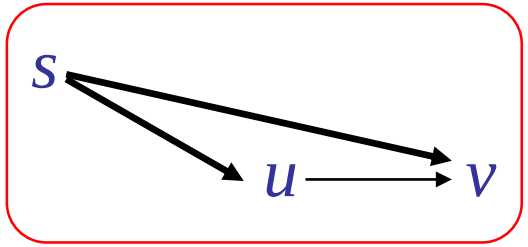
这个步骤实际上为“收紧”过程，称为“松弛”过程，是延续历史的叫法。正如哈夫曼编码里的“前缀码”，实际上应为“前缀无关码”。本章的**每个算法都会调用INITIALIZE-SINGLE-SOURCE，然后反复对每一条边执行relax**操作。

24 Single-Source Shortest Paths

最短路径和松弛的性质：

- Triangle inequality，三角不等式 (Lemma 24.10)

  For any edge $(u, v) \in E$, we have $\delta(s, v) \le \delta(s, u) + w(u, v)$.



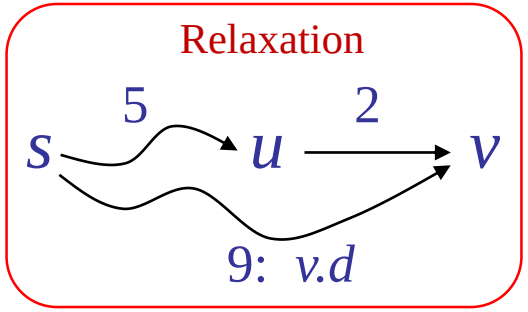
- Upper-bound property，上界属性 (Lemma 24.11)

  We always have $v.d \ge \delta(s, v)$ for all vertices $v \in V$, and once $v.d$ achieves the value $\delta(s, v)$, it never changes.

- Convergence property，收敛属性 (Lemma 24.14)

  If $s \rightsquigarrow u \rightarrow v$ is a shortest path in G for some $(u, v) \in V$, and if $u.d = \delta(s, u)$ at any time prior to relaxing edge (u, v), then $v.d = \delta(s, v)$ at all times afterward.

  假设 $s \rightsquigarrow u \rightarrow v$ 为一条最短路径，如果对边(u, v) 进行松弛前，$u.d$ 达到最短路径，即 $u.d = \delta(s, u)$，则对边(u, v) 进行松弛后，$v.d$ 达到最短路径（即最短路径已经求出），即 $v.d = \delta(s, v)$



24.1 The Bellman-Ford algorithm

BF algorithm solves the single-source shortest-paths problem in the **general case** in which edge weights may be negative, cycle may exists.

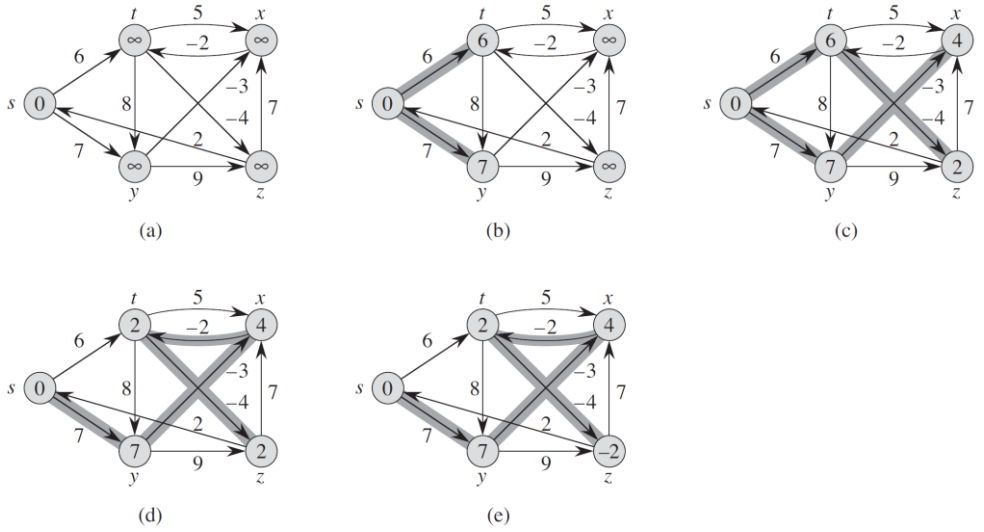
BF算法，通用的算法（允许图有负边和环路）：对顶点进行遍历，针对每个顶点，对所有的边进行松弛操作。

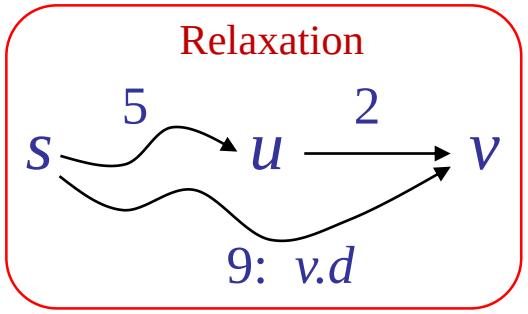
```
BELLMAN-FORD(G, w, s)
1  INITIALIZE-SINGLE-SOURCE(G, s)
2  for i = 1 to |G.V| − 1   //遍历顶点
3      for each edge (u, v) ∈ G.E
4          RELAX(u, v, w)   //松弛边
5  for each edge (u, v) ∈ G.E
6      if v.d > u.d + w(u, v)
7          return FALSE
8  return TRUE
```

Running Time?

```
INITIALIZE-SINGLE-SOURCE(G, s)
1  for each vertex v ∈ G.V
2      v.d = ∞
3      v.π = NIL
4  s.d = 0
```

```
RELAX(u, v, w)
1  if v.d > u.d + w(u, v)
2      v.d = u.d + w(u, v)
3      v.π = u
```





24.1 The Bellman-Ford algorithm

BF算法，通用的算法（允许图有负边和环路）：对顶点进行遍历，针对每个顶点，对所有的边进行松弛操作。

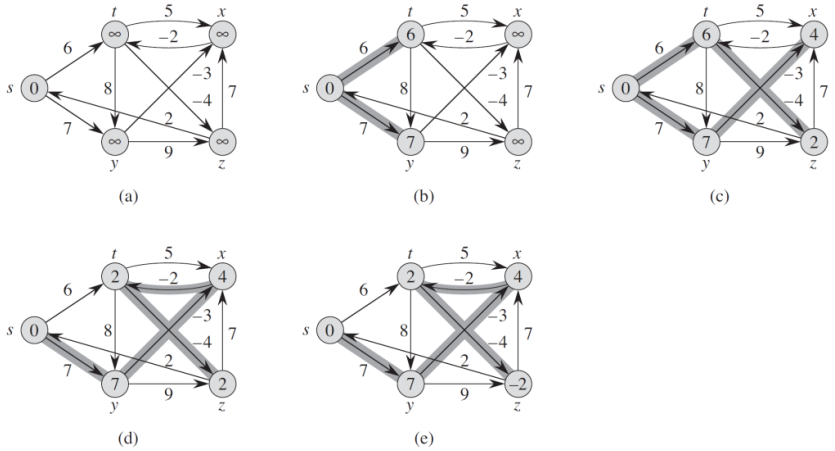
Running Time: 对每个顶点（共 $|V|$-1 个顶点，第 2 行的遍历），所有边都执行一次松弛操作（共 E 条边，第 3 行的遍历），因此，执行时间为 $O(VE)$.

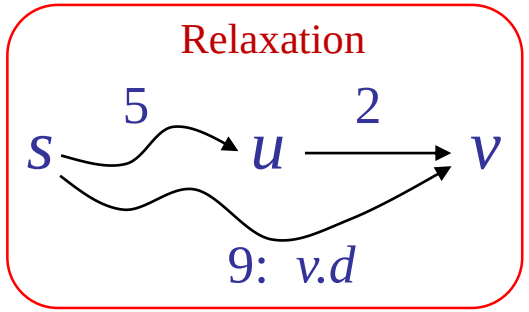
```
BELLMAN-FORD(G, w, s)
1  INITIALIZE-SINGLE-SOURCE(G, s)
2  for i = 1 to |G.V| − 1   //遍历顶点
3      for each edge (u, v) ∈ G.E
4          RELAX(u, v, w)   //松弛边
5  for each edge (u, v) ∈ G.E
6      if v.d > u.d + w(u, v)
7          return FALSE
8  return TRUE
```

检测到有负环路（可以把图中(t, x)的权值改为1，此时有负环路，验证该算法的错误），此时不存在最短路径。

```
INITIALIZE-SINGLE-SOURCE(G, s)
1  for each vertex v ∈ G.V
2      v.d = ∞
3      v.π = NIL
4  s.d = 0
```

```
RELAX(u, v, w)
1  if v.d > u.d + w(u, v)
2      v.d = u.d + w(u, v)
3      v.π = u
```

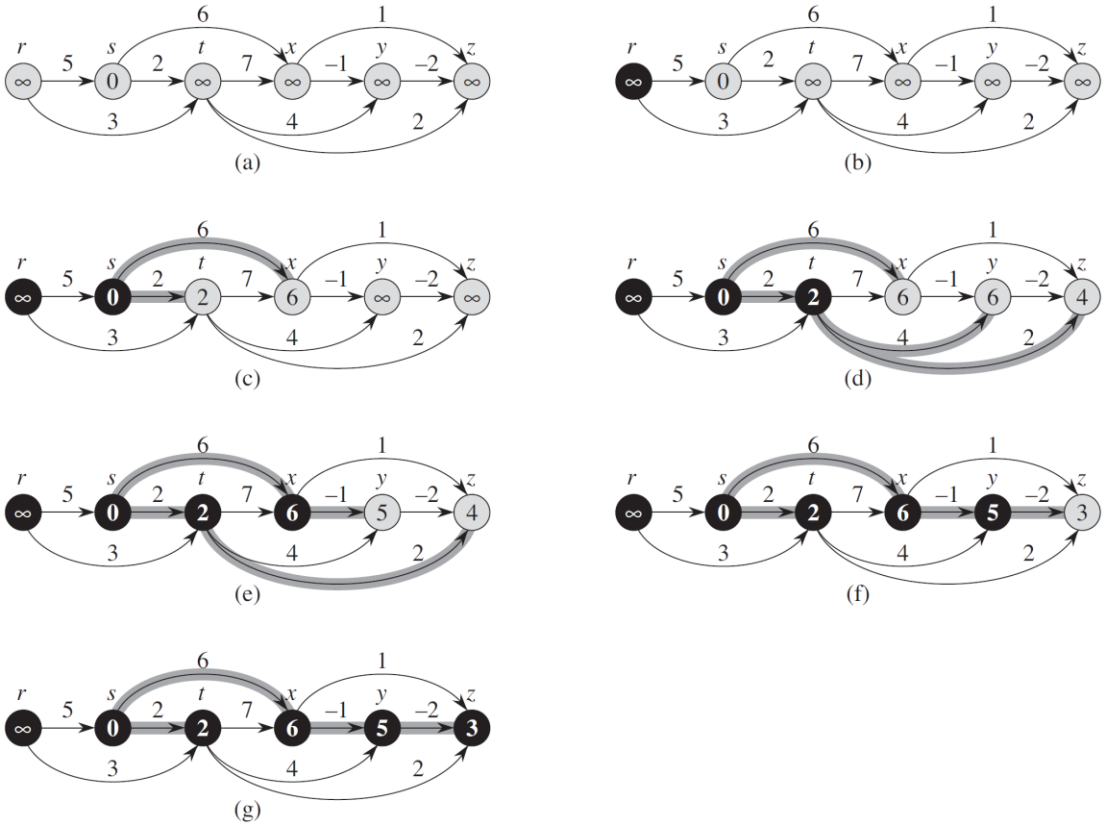




24.2 Topological-sort based algorithm （TS算法）

By relaxing the edges of a weighted dag (directed acyclic graph) $G = (V, E)$, (supports negative edges, but acyclic), according to a topological sort of its vertices.

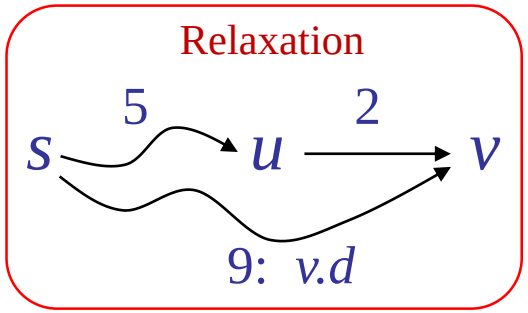
考虑有向无环路图（可以有负边）这类特别情况，先对顶点进行拓扑排序，然后依序对边进行松弛。




```
DAG-SHORTEST-PATHS(G, w, s)
1  topologically sort the vertices of G
2  INITIALIZE-SINGLE-SOURCE(G, s)
3  for each vertex u, taken in topologically sorted order
4       for each vertex v ∈ G.Adj[u]
5            RELAX(u, v, w)
```

```
INITIALIZE-SINGLE-SOURCE(G, s)
1  for each vertex v ∈ G.V
2       v.d = ∞
3       v.π = NIL
4  s.d = 0
```

```
RELAX(u, v, w)
1  if v.d > u.d + w(u, v)
2       v.d = u.d + w(u, v)
3       v.π = u
```

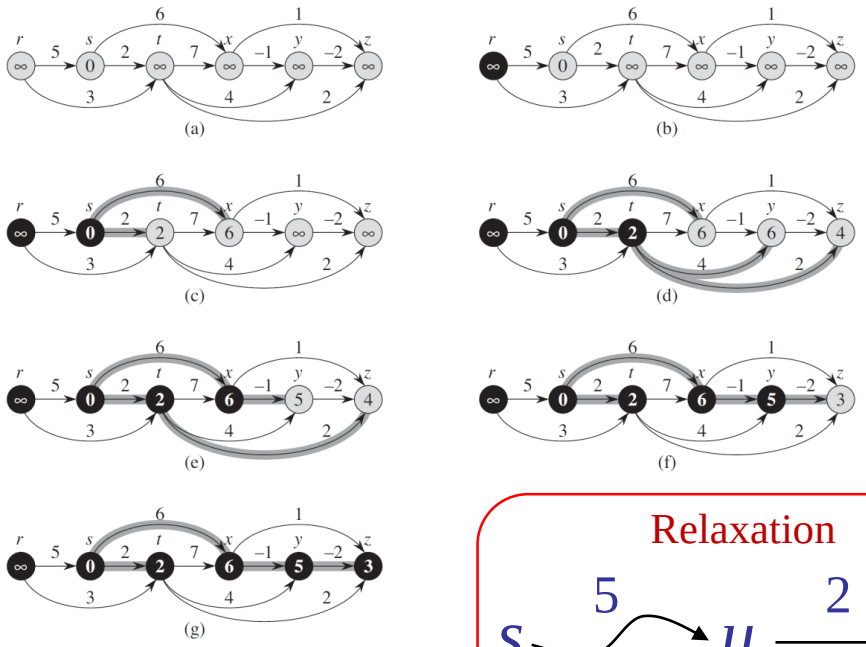


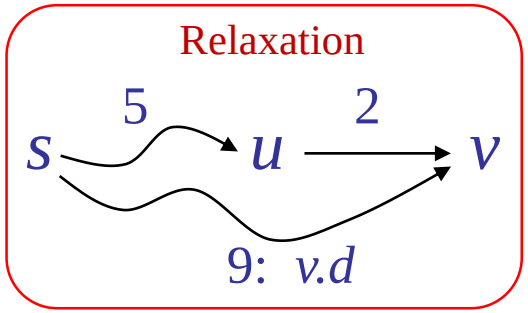
24.2 Topological-sort based algorithm （TS算法）

```
DAG-SHORTEST-PATHS(G, w, s)
1  topologically sort the vertices of G
2  INITIALIZE-SINGLE-SOURCE(G, s)
3  for each vertex u, taken in topologically sorted order
4       for each vertex v ∈ G.Adj[u]
5            RELAX(u, v, w)
```

```
INITIALIZE-SINGLE-SOURCE(G, s)
1  for each vertex v ∈ G.V
2       v.d = ∞
3       v.π = NIL
4  s.d = 0
```

```
RELAX(u, v, w)
1  if v.d > u.d + w(u, v)
2       v.d = u.d + w(u, v)
3       v.π = u
```





第一个开始点 r , 对从 r 出发的边开始，依序松弛每条边：

1. Rx(u, v, w) = Rx(r, s, 5): $v.d = s.d = 0$, $u.d = r.d = \infty$, $w(r, s) = 5$, so, $0 > \infty+5$ 不成立，边 (r, s) 不松弛 relax，见图(a)
2. Rx(u, v, w) = Rx(r, t, 3): $v.d = t.d = \infty$, $u.d = r.d = \infty$, $w(r, s) = 3$, so, $\infty > \infty+3$ 不成立，边 (r, t) 不 relax，见图(b)

第二个开始点 s, 对从 s 出发的边开始，依序松弛每条边：

1. Rx(u, v, w) = Rx(s, t, 2): $v.d = t.d = \infty$, $u.d = s.d = 0$, $w(s, t) = 2$, so, $\infty > 0+2$ 成立，边 (s, t) relax, then $t.d = 0+2 = 2$, $t.\pi = s$，见图(c)
2. Rx(u, v, w) = Rx(s, x, 6): $v.d = x.d = \infty$, $u.d = s.d = 0$, $w(s, x) = 6$, so, $\infty > 0+6$ 成立，边 (s, x) relax, then $x.d = 0+6 = 6$, $x.\pi = s$，图(b)到(c)

第三个开始点 t …

……

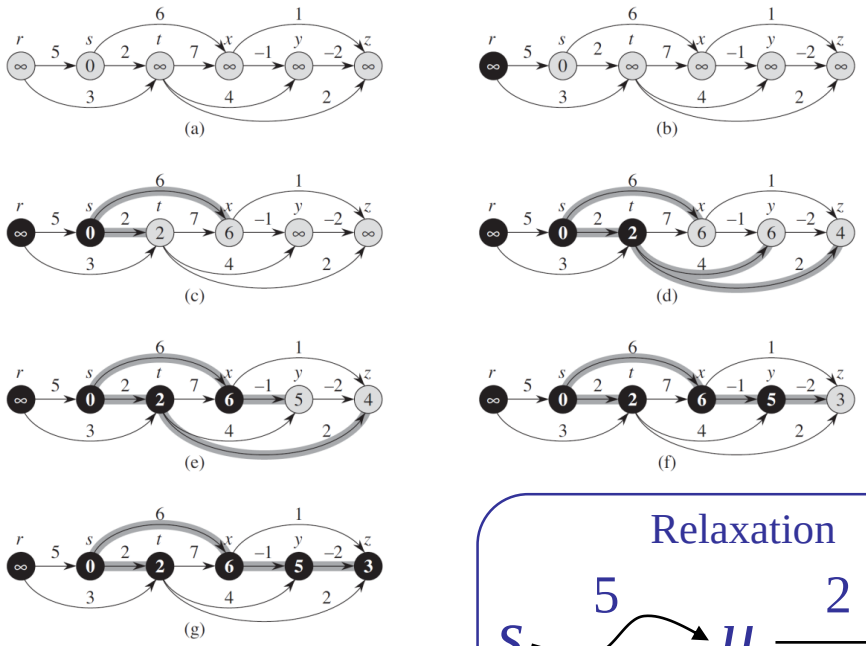
running time?

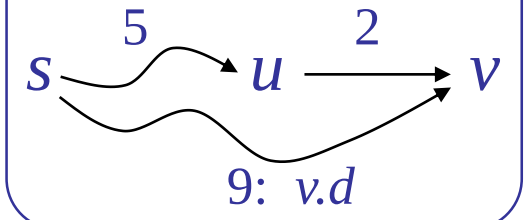
24.2 Topological-sort based algorithm （TS算法）

```
DAG-SHORTEST-PATHS(G, w, s)
1  topologically sort the vertices of G
2  INITIALIZE-SINGLE-SOURCE(G, s)
3  for each vertex u, taken in topologically sorted order
4       for each vertex v ∈ G.Adj[u]
5            RELAX(u, v, w)
```

```
INITIALIZE-SINGLE-SOURCE(G, s)
1  for each vertex v ∈ G.V
2       v.d = ∞
3       v.π = NIL
4  s.d = 0
```

```
RELAX(u, v, w)
1  if v.d > u.d + w(u, v)
2       v.d = u.d + w(u, v)
3       v.π = u
```





第一个开始点 r , 对从 r 出发的边开始，依序松弛每条边：

1. Rx(u, v, w) = Rx(r, s, 5): 0 > ∞+5 不成立，边 (r, s) 不 relax，见图(a)
2. Rx(u, v, w) = Rx(r, t, 3): ∞ > ∞+3 不成立，边 (r, t) 不 relax，见图(b)

第二个开始点 s, 对从 s 出发的边开始，依序松弛每条边：

1. Rx(u, v, w) = Rx(s, t, 2): ∞ > 0+2 成立，边 (s, t) relax, 见图(c)
2. Rx(u, v, w) = Rx(s, x, 6): ∞ > 0+6 成立，边 (s, x) relax, 图(b)到(c)

第三个开始点 t …

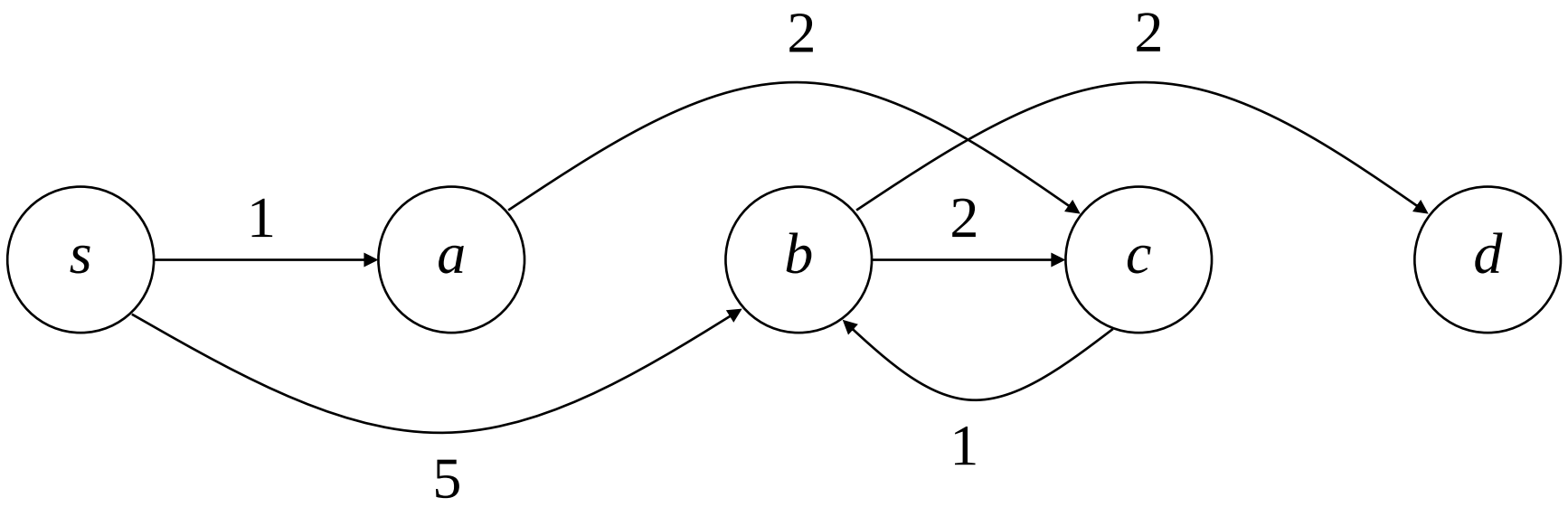
……

running time?	$\Theta(V+E)$, aggregate analysis, 每个顶点和每条边都走一遍

Exercise: 如果不是 dag（有向无环路图），TS算法能否可行？若可行，请证明，若不可行，给一个反例。

24.2 Topological-sort based algorithm （TS算法）

Exercise: 如果不是 dag，TS算法能否可行？若可行，请证明，若不可行，给一个反例。

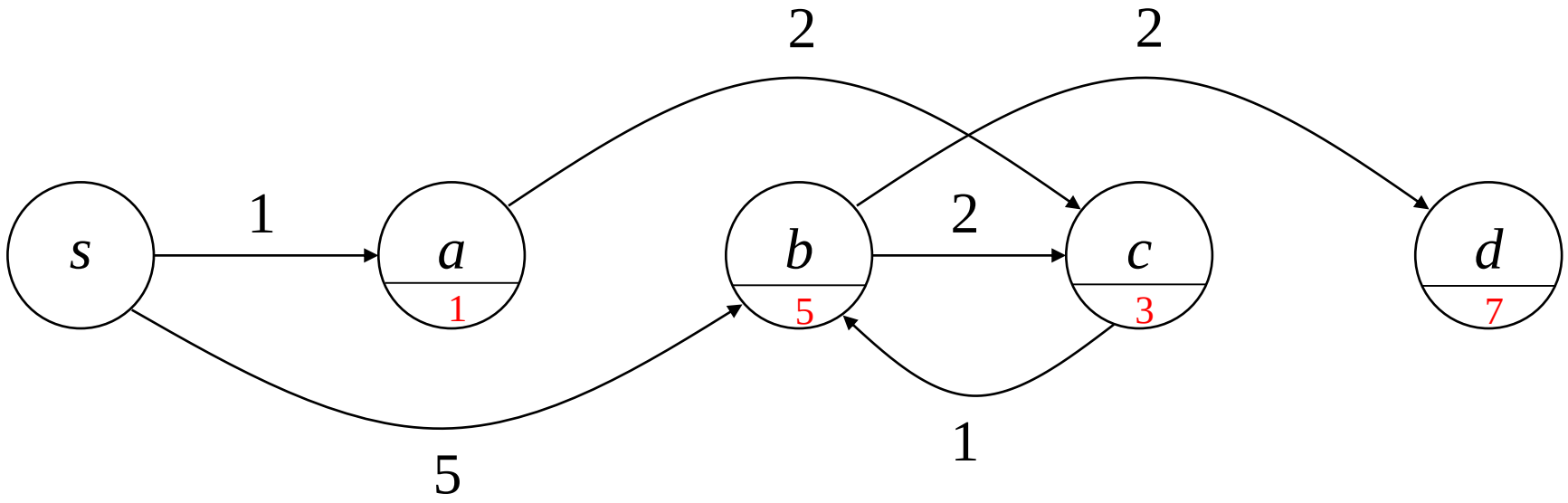


求 $\delta(s, x)$ 为多少，其中 x 为其他任意顶点。

24.2 Topological-sort based algorithm （TS算法）

如果不是 dag，TS算法能否可行？若可行，请证明，若不可行，给一个反例。

```
RELAX(u, v, w)
1  if v.d > u.d + w(u, v)
2      v.d = u.d + w(u, v)
3      v.π = u
```



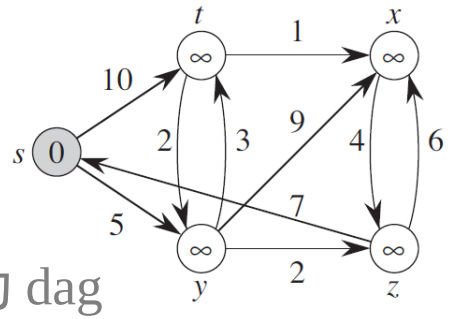
采用拓扑排序的算法，求出 $\delta(s, d) = 7$，但实际的最短路径的长度为 6 (路径为 $s \xrightarrow{1} a \xrightarrow{2} c \xrightarrow{1} b \xrightarrow{2} d$).

因此，基于 TS 算法的最短路径问题，不适用于有环路的情况.

如上图，环路为 $b \to c \to b$.

24.3 Dijkstra's algorithm (greedy strategy)

- **Dijkstra 算法：能求解包括环路的有向图（但要求边是非负的）的最短路径问题。**
- 24.2 Topo可解决含“负边”的最短路问题，但要求图为 dag
- 24.1 BF方法可解决包括“负边”和“环路”的最短路问题

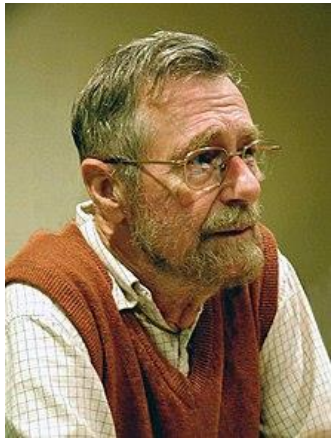


A note on two problems in connexion with graphs
EW Dijkstra
Edsger Wybe Dijkstra: his life, work, and legacy, 2022 • dl.acm.org

> We start the construction by choosing an arbitrary node as the only member of set A, and by placing all branches that end in this node in set II. To start with, set I is empty. From then onwards we perform the following two steps repeatedly. Step 1. The shortest branch of set II is removed from this set and added to set I. As a result one node is transferred from set B to set A.

ACM Digital Library

☆ 保存 引用 被引用次数: 36124 相关文章 所有 39 个版本



Edsger Wybe Dijkstra
1930/5/11-2002/8/6
1972年图灵奖获得者

① 提出“goto有害论”
② 提出信号量和PV原语
③ 解决了“哲学家聚餐”问题
④ 最短路径算法和银行家算法
⑤ Algol 60的设计者和实现者
⑥ THE操作系统的设计者和开发者

与Knuth并称为我们这个时代最伟大的计算机科学家的人

24.3 Dijkstra's algorithm (greedy strategy)

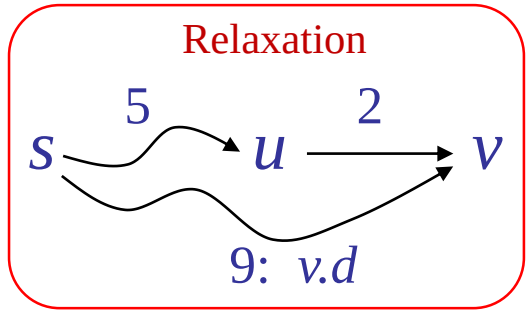
- **Dijkstra** 维护一个顶点集 S，集合里每个点的距离（从 s 到该顶点的距离）是最短路径。
- 数据结构(DS)：优先队列；　策略：贪心（优先队列）；　方法：BFS, Relax

```
DIJKSTRA(G, w, s)
1  INITIALIZE-SINGLE-SOURCE(G, s)
2  S = ∅
3  Q = G.V
4  while Q ≠ ∅
5      u = EXTRACT-MIN(Q)
6      S = S ∪ {u}
7      for each vertex v ∈ G.Adj[u]
8          RELAX(u, v, w)
```

Q：最小优先队列

广度优先搜索

```
RELAX(u, v, w)
1  if v.d > u.d + w(u, v)
2      v.d = u.d + w(u, v)
3      v.π = u
```

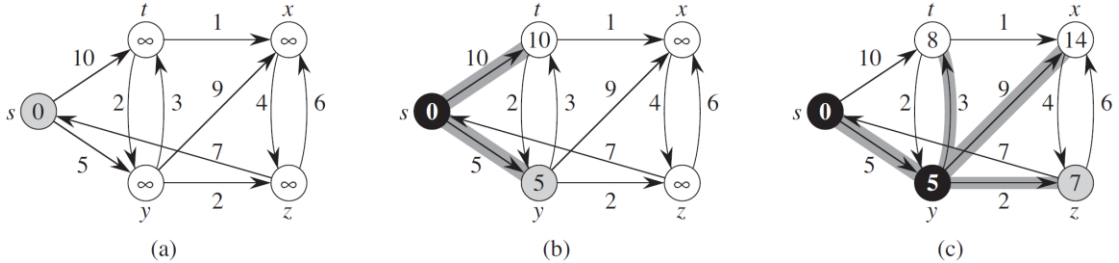



```
INITIALIZE-SINGLE-SOURCE(G, s)
1  for each vertex v ∈ G.V
2      v.d = ∞
3      v.π = NIL
4  s.d = 0
```

Q: s:0, ?:∞…
S: Φ

Q: y:5, t:10 …
S: s

Q: z:7, t:8, x:14
S: s, y:5



断言证明：Q 是最小优先队列（最小堆），5 最小，显然 s 到其他顶点的路径都不小于 5（否则，其他顶点为最小堆的根，跟 y 是根矛盾）。从 s 经过其他顶点再到 y，路径权值一定大于 5。

断言：s 到 y 的最短路径的长度一定是 5？

24.3 Dijkstra's algorithm (greedy strategy)

- **Dijkstra** 维护一个顶点集 S，集合里每个点的距离（从 s 到该顶点的距离）是最短路径。
- 数据结构(DS)：优先队列；　策略：贪心（优先队列）；　方法：BFS, Relax

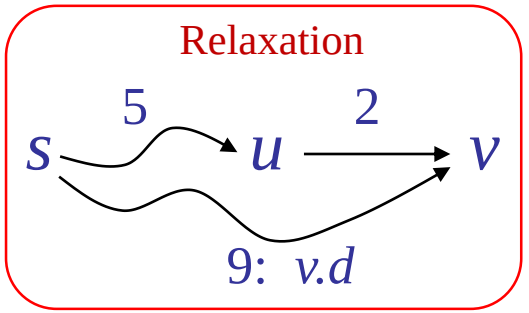
```
DIJKSTRA(G, w, s)
1  INITIALIZE-SINGLE-SOURCE(G, s)
2  S = ∅
3  Q = G.V
4  while Q ≠ ∅
5      u = EXTRACT-MIN(Q)
6      S = S ∪ {u}
7      for each vertex v ∈ G.Adj[u]
8          RELAX(u, v, w)
```

Q：最小优先队列

广度优先搜索

```
INITIALIZE-SINGLE-SOURCE(G, s)
1  for each vertex v ∈ G.V
2      v.d = ∞
3      v.π = NIL
4  s.d = 0
```

```
RELAX(u, v, w)
1  if v.d > u.d + w(u, v)
2      v.d = u.d + w(u, v)
3      v.π = u
```

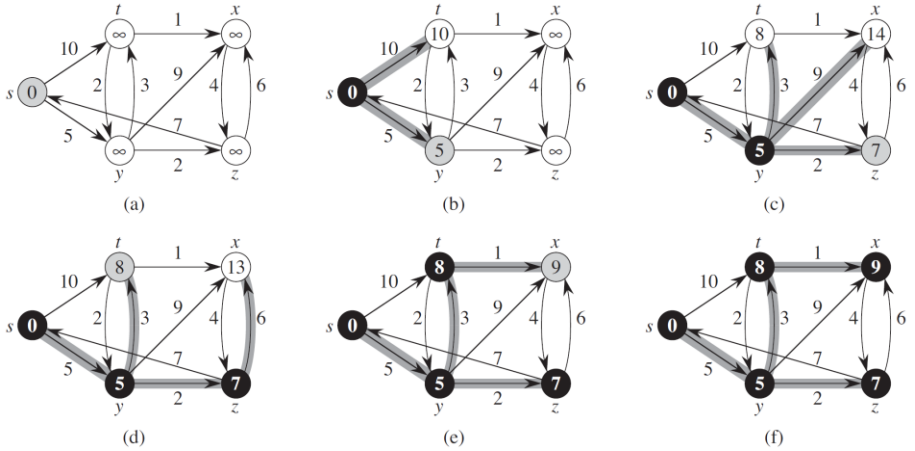


Because Dijkstra's algorithm always chooses the "lightest" or "closest" vertex in $V - S$ to add to set S, we say that it uses a greedy strategy.

Theorem 24.6 (Correctness of DJ)
Proof loop invariant.

Q: s:0, ?:∞…　　Q: y:5, t:10 …　　Q: z:7, t:8, x:14

S: Φ　　S: s　　S: s, y:5



24.3 Dijkstra's algorithm (greedy strategy)

这是一个非常高效的算法

```
DIJKSTRA(G, w, s)
1  INITIALIZE-SINGLE-SOURCE(G, s)
2  S = ∅
3  Q = G.V
4  while Q ≠ ∅
5      u = EXTRACT-MIN(Q)
6      S = S ∪ {u}
7      for each vertex v ∈ G.Adj[u]
8          RELAX(u, v, w)
```

```
INITIALIZE-SINGLE-SOURCE(G, s)
1  for each vertex v ∈ G.V
2      v.d = ∞
3      v.π = NIL
4  s.d = 0
```

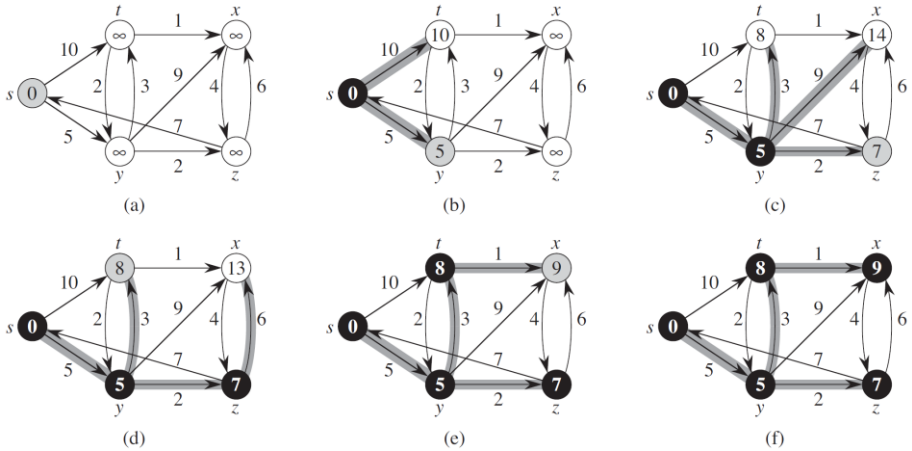
```
RELAX(u, v, w)
1  if v.d > u.d + w(u, v)
2      v.d = u.d + w(u, v)
3      v.π = u
```

Running time ?

Q: s:0, ?:∞… S: Φ

Q: y:5, t:10 … S: s

Q: z:7, t:8, x:14 S: s, y:5



24.3 Dijkstra's algorithm (greedy strategy)

这是一个非常高效的算法

```
DIJKSTRA(G, w, s)
1  INITIALIZE-SINGLE-SOURCE(G, s)
2  S = ∅
3  Q = G.V
4  while Q ≠ ∅
5      u = EXTRACT-MIN(Q)
6      S = S ∪ {u}
7      for each vertex v ∈ G.Adj[u]
8          RELAX(u, v, w)
```

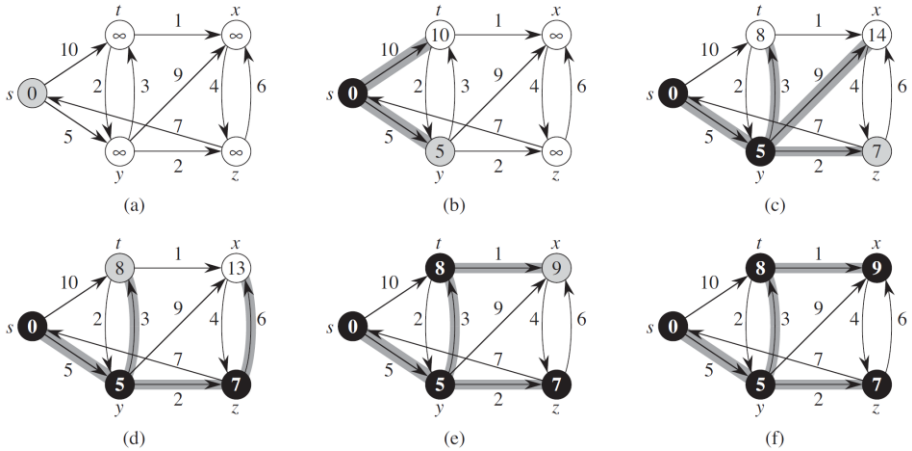
```
INITIALIZE-SINGLE-SOURCE(G, s)
1  for each vertex v ∈ G.V
2      v.d = ∞
3      v.π = NIL
4  s.d = 0
```

```
RELAX(u, v, w)
1  if v.d > u.d + w(u, v)
2      v.d = u.d + w(u, v)
3      v.π = u
```

Q: $\overline{s{:}0,\ ?{:}\infty\ldots}$ S: Φ

Q: $\overline{y{:}5,\ t{:}10\ \ldots}$ S: s

Q: $\overline{z{:}7,\ t{:}8,\ x{:}14}$ S: $s, y{:}5$



Running time ?

The running time of DJ depends on how we implement the min-priority queue. It may be

$$O(V^2)\ ,\ O(V + E\lg V)\ ,\ \text{or}\ O(V\lg V + E)$$

$O(V + E\lg V)$的解释：第8行，每一次松弛后点值 $d(s,v)$ 可能更新，此时需要维护一次最小堆，每次维护是 $O(\lg V)$；整个松弛是 E 次，因此有 $E\lg V$。

24.3 Dijkstra's algorithm (greedy strategy)

这是一个非常高效的算法

```
DIJKSTRA(G, w, s)
1  INITIALIZE-SINGLE-SOURCE(G, s)
2  S = ∅
3  Q = G.V
4  while Q ≠ ∅
5      u = EXTRACT-MIN(Q)
6      S = S ∪ {u}
7      for each vertex v ∈ G.Adj[u]
8          RELAX(u, v, w)
```

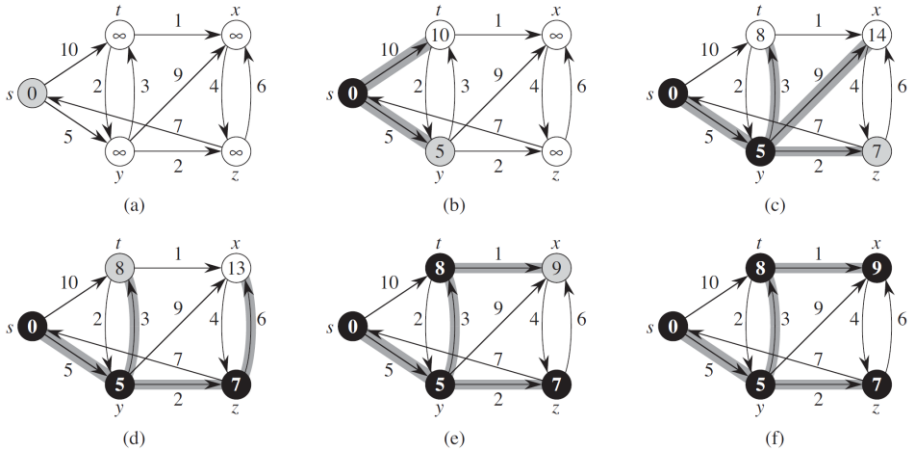
```
INITIALIZE-SINGLE-SOURCE(G, s)
1  for each vertex v ∈ G.V
2      v.d = ∞
3      v.π = NIL
4  s.d = 0
```

```
RELAX(u, v, w)
1  if v.d > u.d + w(u, v)
2      v.d = u.d + w(u, v)
3      v.π = u
```

Q: $\overline{s{:}0,\ ?{:}\infty\ldots}$ S: Φ

Q: $\overline{y{:}5,\ t{:}10\ \ldots}$ S: s

Q: $\overline{z{:}7,\ t{:}8,\ x{:}14}$ S: $s, y{:}5$



Exercise:

如果有负边，DJ 算法能否可行？如果可行，请证明，如果不可行，给一个反例。

24.3 Dijkstra's algorithm (code from 通义)

```c
#include <stdio.h>
#include <limits.h>
#define V 9 // 定义图中顶点的数量
// 找出距离数组dist中的最小值顶点（除了已包含在sptSet中的）
int minDistance(int dist[], int sptSet[])
{
    int min = INT_MAX, min_index;
    for (int v = 0; v < V; v++)
    if (sptSet[v] == 0 && dist[v] <= min)
        min = dist[v], min_index = v;
    return min_index;
}

// 打印从源顶点到所有顶点的最短路径
void printSolution(int dist[], int n)
{
    printf("Vertex \t Distance from Source\n");
    for (int i = 0; i < V; i++)
        printf("%d \t\t %d\n", i, dist[i]);
}
```

```c
// 主函数
int main()
{
    /* 假设以下为图的邻接矩阵表示 */
    int graph[V][V] = { {0, 4, 0, 0, 0, 0, 0, 8, 0},
        {4, 0, 8, 0, 0, 0, 0, 11, 0},
        {0, 8, 0, 7, 0, 4, 0, 0, 2},
        {0, 0, 7, 0, 9, 14, 0, 0, 0},
        {0, 0, 0, 9, 0, 10, 0, 0, 0},
        {0, 0, 4, 14, 10, 0, 2, 0, 0},
        {0, 0, 0, 0, 0, 2, 0, 1, 6},
        {8, 11, 0, 0, 0, 0, 1, 0, 7},
        {0, 0, 2, 0, 0, 0, 6, 7, 0}
    };
    dijkstra(graph, 0); // 以顶点0作为源
    return 0;
}
```

```c
// Dijkstra算法函数
void dijkstra(int graph[V][V], int src)
{
    int dist[V]; // 存储每个顶点到源顶点的最短距离
    int sptSet[V]; // sptSet[i]为真当顶点i包含在最短路径树中
    // 初始化距离数组和sptSet
    for (int i = 0; i < V; i++)
    {
        dist[i] = INT_MAX, sptSet[i] = 0;
    }
    dist[src] = 0; // 源顶点的距离设置为0
    // 处理所有的顶点
    for (int count = 0; count < V - 1; count++)
    {
        // 选取当前最短距离的顶点
        int u = minDistance(dist, sptSet);

        // 将选中的顶点加入sptSet
        sptSet[u] = 1;

        // 更新dist值
        for (int v = 0; v < V; v++)

       // 更新dist[v]仅当不在sptSet中，且有边从u到v，且总权重小
       于当前dist[v]
       if (!sptSet[v] && graph[u][v] && dist[u] != INT_MAX
                    && dist[u] + graph[u][v] < dist[v])
           dist[v] = dist[u] + graph[u][v];
    }
    // 打印构建好的最短路径树
    printSolution(dist, V);
}
```

这段代码定义了一个简单的图，并使用Dijkstra算法找到从顶点0到所有其他顶点的最短路径。你可以根据实际需要修改`V`的值以及`graph`数组来适应不同的图结构。

Exercise:

根据通义的代码，写出算法伪代码，类似如下，并分析复杂度。

DIJKSTRA(G, w, s)

```
1  INITIALIZE-SINGLE-SOURCE(G, s)
2  S = ∅
3  Q = G.V
4  while Q ≠ ∅
5      u = EXTRACT-MIN(Q)
6      S = S ∪ {u}
7      for each vertex v ∈ G.Adj[u]
8          RELAX(u, v, w)
```

INITIALIZE-SINGLE-SOURCE(G, s)

```
1  for each vertex v ∈ G.V
2      v.d = ∞
3      v.π = NIL
4  s.d = 0
```

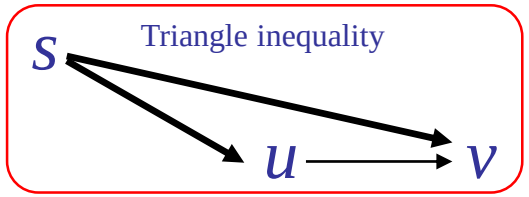
RELAX(u, v, w)

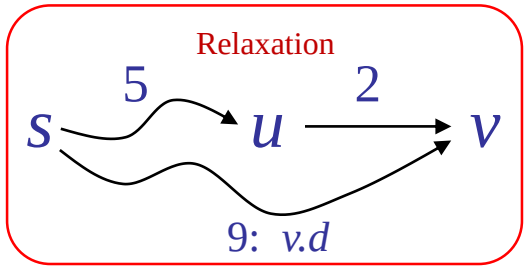
```
1  if v.d > u.d + w(u, v)
2      v.d = u.d + w(u, v)
3      v.π = u
```

24 Single-Source Shortest Paths

Summary

- 单源最短路径问题的关键技术
  - 三角不等式
  - 松弛原理
  - 最小优先队列
  - DFS and BFS
  - 拓扑排序





方法	负边	环路
BF	√	√
topo sort	√	
Dijkstra		√

	环路	非环路
负边	BF	topo sort BF
非负边	Dijkstra BF	DJ topo sort BF

- 三个算法
  - Bellman-Ford(BF) 算法（通用，可包括负边和环路）
  - 基于拓扑排序的BF算法（可包括负边，但须是有向无环路图dag）
  - Dijkstra(DJ) 算法（贪心策略，最小优先队列）（可环路，但不能有负边）
- 复杂度分析：聚集分析

Summary

数据结构

- Representations:
  Adjacency-list, Adjacency-matrix
- Queue, Priority Queue, Stack

策略（技术、方法）

- Search: BFS，DFS
- Backtrack
- Recursion
- Relaxation
- Greedy
- DP

算法

- The Bellman-Ford
- Topological sort
- Dijkstra

分析

- Aggregate analysis

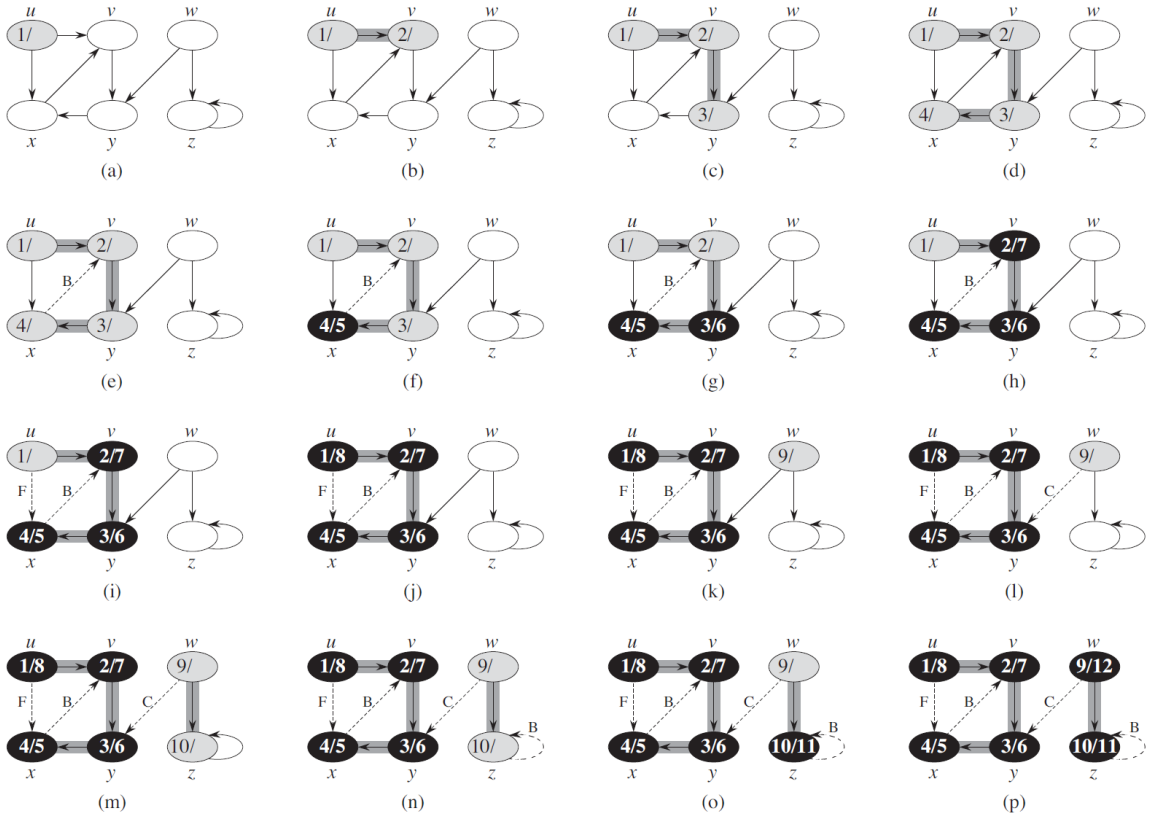
Exercise

邻接表AL(adjacency-list)和邻接矩阵AM(adjacency-matrix)，分别适合于哪种类型的图表示？AL和AM各有什么优缺点?

Exercise

Exercise: For adjacency-matrix representation, the running time of DFS?

用邻接矩阵表示，DFS的计算时间?




```
DFS(G)
1  for each vertex u ∈ G.V
2      u.color = WHITE
3      u.π = NIL
4  time = 0
5  for each vertex u ∈ G.V
6      if u.color == WHITE
7          DFS-VISIT(G, u)

DFS-VISIT(G, u)
 1  time = time + 1
 2  u.d = time
 3  u.color = GRAY
 4  for each v ∈ G.Adj[u]
 5      if v.color == WHITE
 6          v.π = u
 7          DFS-VISIT(G, v)
 8  u.color = BLACK
 9  time = time + 1
10  u.f = time
```

Exercise

Bellman-Ford、拓扑搜索、Dijkstra算法，其运行时间分别是什么？

这三种算法适用什么性质的图（负边、环路）？

```
BELLMAN-FORD(G, w, s)
1  INITIALIZE-SINGLE-SOURCE(G, s)
2  for i = 1 to |G.V| − 1
3       for each edge (u, v) ∈ G.E
4            RELAX(u, v, w)
5  for each edge (u, v) ∈ G.E
6       if v.d > u.d + w(u, v)
7            return FALSE
8  return TRUE
```

```
DAG-SHORTEST-PATHS(G, w, s)
1  topologically sort the vertices of G
2  INITIALIZE-SINGLE-SOURCE(G, s)
3  for each vertex u, taken in topologically sorted order
4       for each vertex v ∈ G.Adj[u]
5            RELAX(u, v, w)
```

```
DIJKSTRA(G, w, s)
1  INITIALIZE-SINGLE-SOURCE(G, s)
2  S = ∅
3  Q = G.V
4  while Q ≠ ∅
5       u = EXTRACT-MIN(Q)
6       S = S ∪ {u}
7       for each vertex v ∈ G.Adj[u]
8            RELAX(u, v, w)
```

```
INITIALIZE-SINGLE-SOURCE(G, s)
1  for each vertex v ∈ G.V
2       v.d = ∞
3       v.π = NIL
4  s.d = 0
```

```
RELAX(u, v, w)
1  if v.d > u.d + w(u, v)
2       v.d = u.d + w(u, v)
3       v.π = u
```