Algorithm Design and Analysis

David N. JANSEN, Bohua ZHAN
名 姓

算法设计与分析

詹博华，杨大卫

This week's content

- Today Wednesday:
  - Chapter 23: Minimum Spanning Tree
  - Chapter 24: Single-Source Shortest Paths
  - Exercises
- Tomorrow Thursday:
  - Exercise solutions
  - Chapter 25: All-Pairs Shortest Paths

这周的内容

- 今天周三：
  - 第23章：最小生成树
  - 第24章：单源最短路径
  - 练习
- 明天周四：
  - 练习题解答
  - 第25章：所有结点对的最短路径

Algorithm Design and Analysis

Minimum Spanning Tree

David N. JANSEN

名 姓

算法设计与分析

最小生成树

杨大卫

Weighted Graph

- Graph $G = (V, E)$ consists of vertices V and edges $E \subseteq V \times V$.
- Weigthed graph has a weight function w: $E \to \mathbb{R}$.
- Most often,
  $w((u,v))$ = length of edge (u,v).
- Sometimes,
  $w((u,v))$ = width/capacity of edge (u,v).

权重的图

- 图 $G = (V, E)$ 存在于结点 V 和边 $E \subseteq V \times V$。
- 权重的图还有权重函数 w: $E \to \mathbb{R}$。
- 常常， $w((u,v))$ = 边 (u,v) 的长度。
- 有时候 $w((u,v))$ = 边 (u,v) 的宽度/容量。

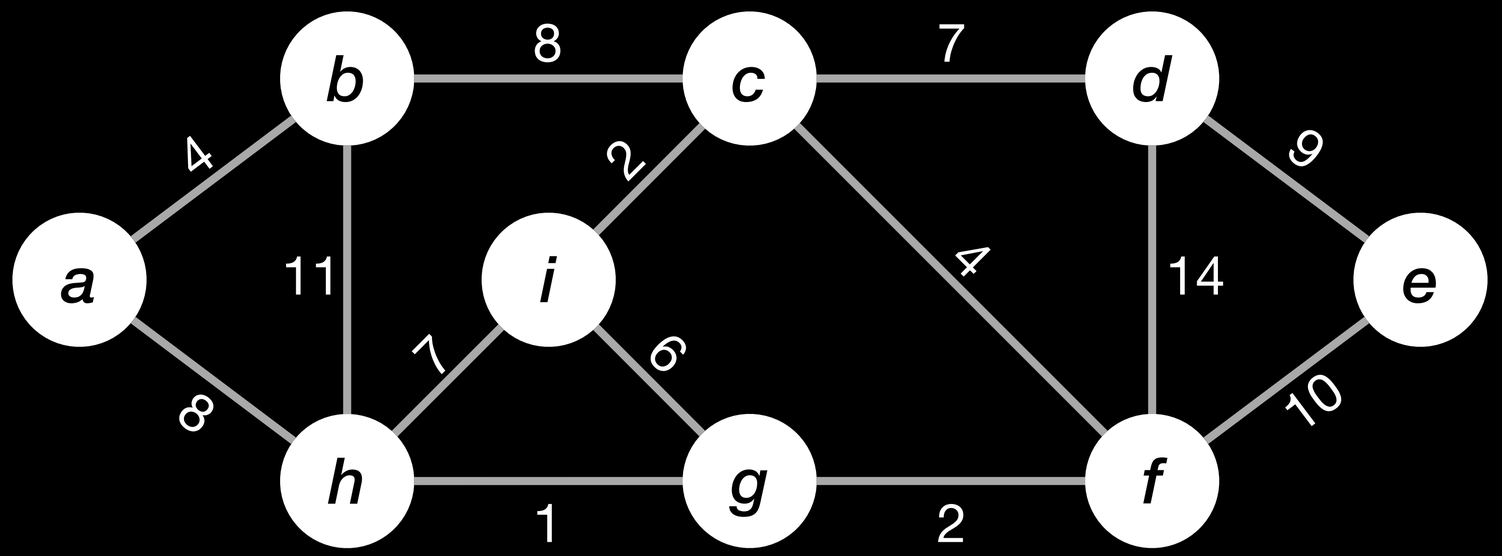
Cheapest Network

最便宜的网络

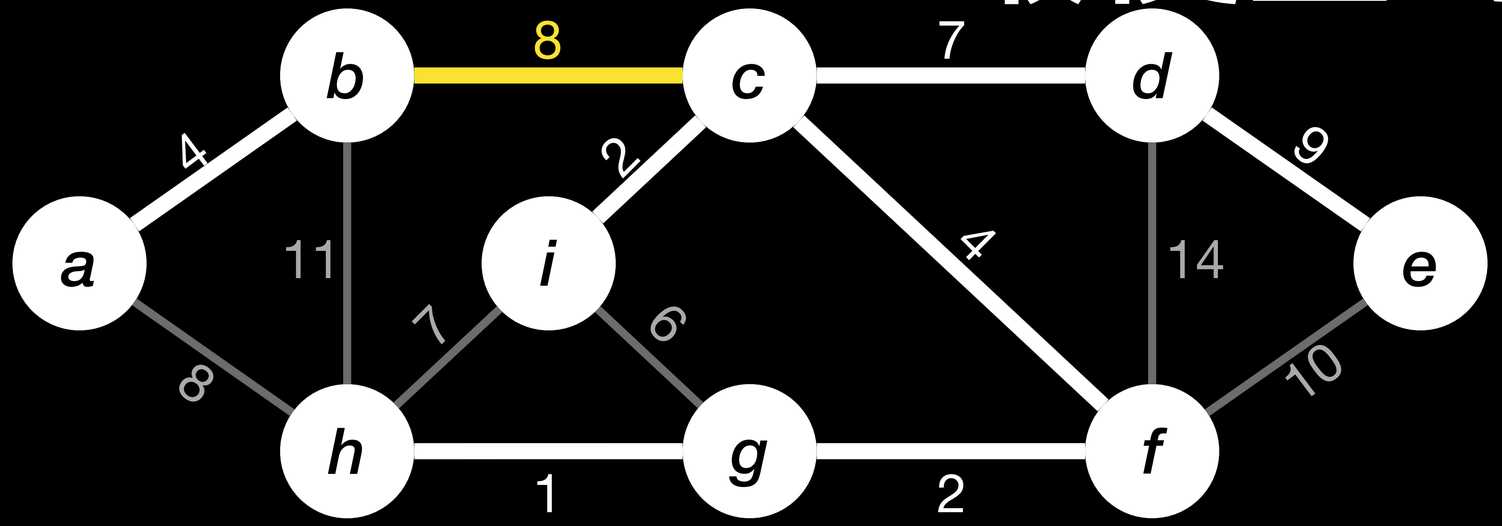
- Example: A number of computers are connected in a network. What is the cheapest network?
- Solution: minimum spanning tree
  - shortest edges
  - all computers / vertices connected
  - no cycles

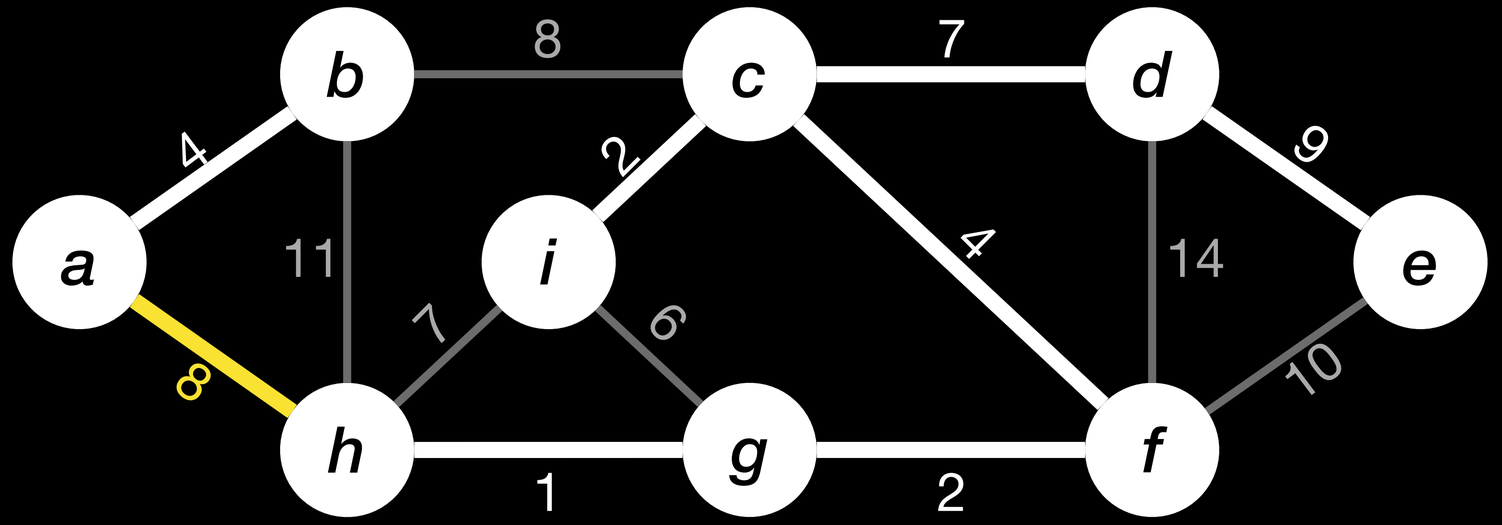
- 例子：许多计算机在网络中连接。最便宜的网络是什么?
- 解决方案：最小生成树
  - 最短边
  - 连接的所有计算机 / 顶点
  - 没有环路

Cheapest Network 最便宜的网络



Cheapest Network 最便宜的网络





Generic Method

- Idea for a greedy algorithm:
  - Start with a subset of the MST
  - In every step, add one “safe” edge, i.e. one edge that is ok to be added
- Q: What is a “safe” edge?
  A: Theorem 23.1

通用方法

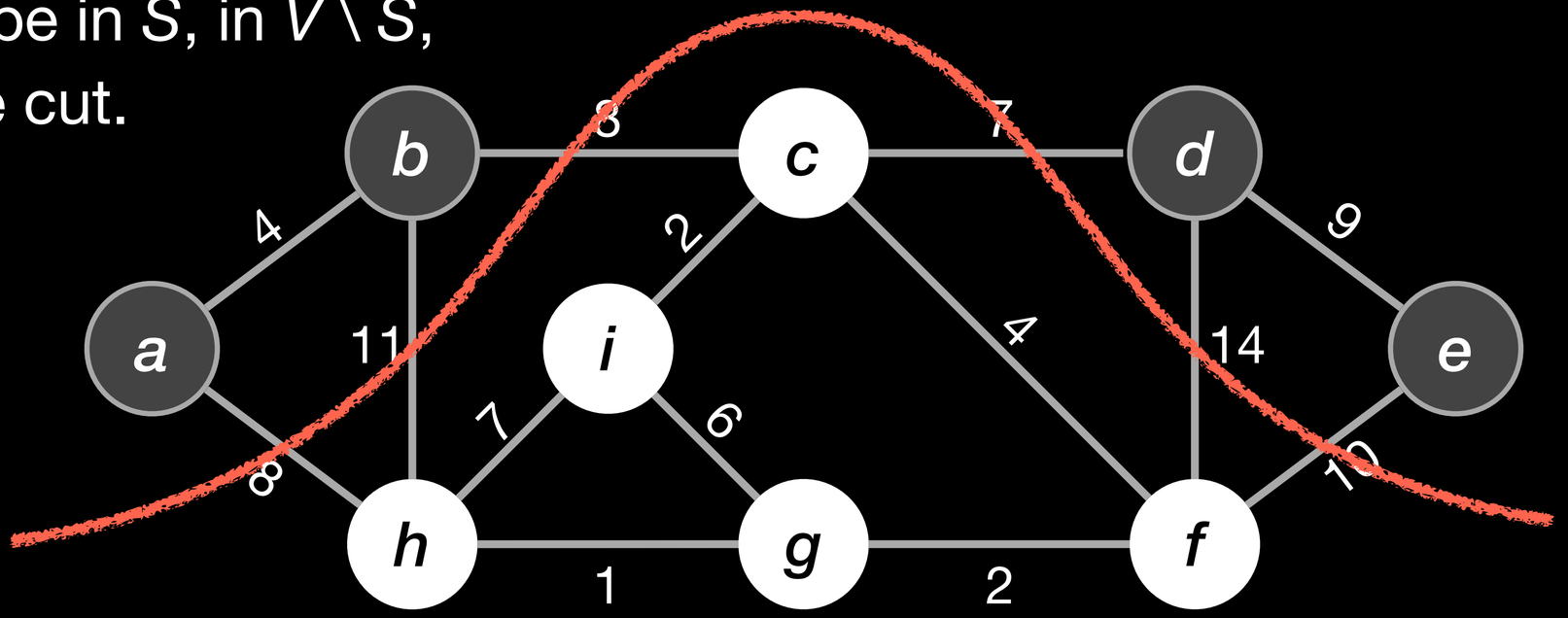
- 贪心算法的方法：
  - 从最小生成树的子集开始
  - 在每一步中，插入一条“安全”的边，i.e.可以添加的一条边
- 问：什么是“安全”的边?
  答：定理23.1

Cut of a Graph

图的切割

- A cut of a graph $G = (V, E)$ separates the vertices into two subsets $S \subseteq V$ and $V \setminus S$.
- Edges can be in S, in $V \setminus S$, or cross the cut.

- 图$G = (V, E)$的切割分开结点成两个子集 $S \subseteq V$ 和 $V \setminus S$。
- 边可以是在 S、$V \setminus S$ 中，或者横跨切割。



Cut of a Graph

- A cut of a graph $G = (V, E)$ separates the vertices into two subsets S and $V \setminus S$.
- Edges can be in S, in $V \setminus S$, or cross the cut.
- A cut **respects** a set of edges $A \subseteq E$ if no edge in A crosses the cut.

图的切割

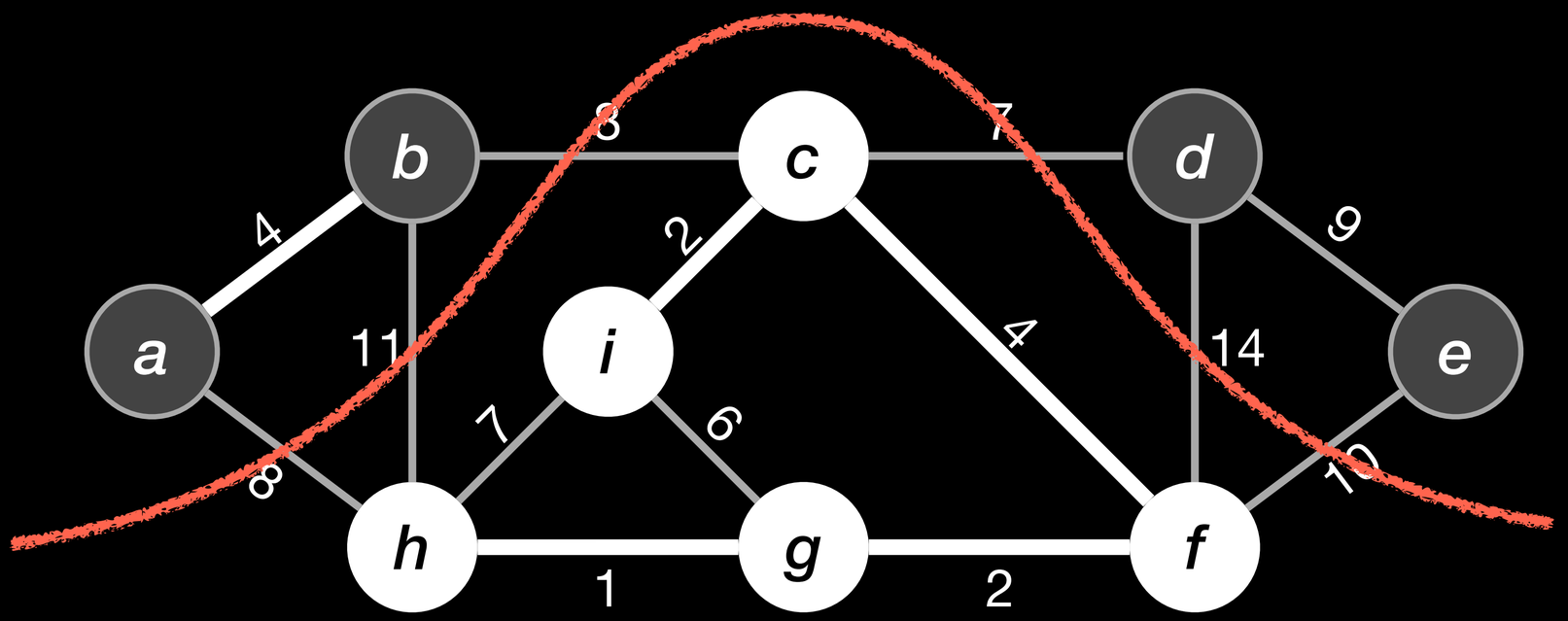
- 图$G = (V, E)$的切割分开结点成两个子集S和$V\setminus S$。
- 边可以是在 S、$V\setminus S$ 中，或者横跨切割。
- 如果集合 $A \subseteq E$ 中不存在横跨切割的边，则称该切割**尊重**集合 A。

Example: Respect a cut

例子：尊重切割

- Cut $S = \{a, b, d, e\}$
  $A = \{(a,b), (c,f), (c,i), (f,g), (g,h)\}$
- The cut S respects the set of edges A.

- 切割 $S = \{a, b, d, e\}$
  $A = \{(a,b), (c,f), (c,i), (f,g), (g,h)\}$
- 切割 S 尊重边集 A。



Safe Edge for MST

- **Theorem 23.1**
  Let *A* be a subset of the edges of a graph that is included in some MST.
  Let (*S*, *V**S*) be any cut of the graph that respects *A*.
  Then every light edge crossing the cut is safe for *A*.

- (light edge = edge with minimal weight)

安全的边

- **定理23.1**
  设集合 A 为一个图的边的子集，并 A 包括在图的某最小生成树中。
  设 (*S*, *V**S*) 是图中尊重集合 A 的任意一个切割。
  所有的横跨切割的轻量级边对于集合 A 是安全的。

- （轻量级边 = 权重最小的边）

Kruskal's Algorithm

- Idea:
  Always add the lightest/shortest edge
  that is allowed.

- Allowed are edges
  that do not form a cycle.

- Q: How do we test
  whether an edge is allowed?
  A: Check whether
  endpoints are in the same tree.

Kruskal的算法

- 想法：
  始终添加允许的最轻/最短边。

- 允许的边
  不会形成一个环路。

- 问：我们如何测试
  边是否允许的?
  答：检查一下
  端点在同一棵 树 中。

Disjoint Sets

- a data structure to describe
  a partition of a set
- operations:
  - UNION: join two subsets into one
  - FIND-SET: find the representative
    of the subset
- sometimes called
  “Union–Find data structure”

不相交集合

- 要描述的数据结构
  集合的划分
- 操作：
  - UNION：将两个子集合并为一个
  - FIND-SET：查找子集的代表
- 有时叫
  “Union–Find数据结构”

Representative

- Every subset is identified
  by a representative element.
  When the set does not change,
  the representative stays the same.

- Operations in detail:
  - MAKE-SET(x): create structure for $\{x\}$
  - UNION(x,y): join the subsets containing
    x and y into one
  - FIND-SET(x): find the representative
    of the subset containing x

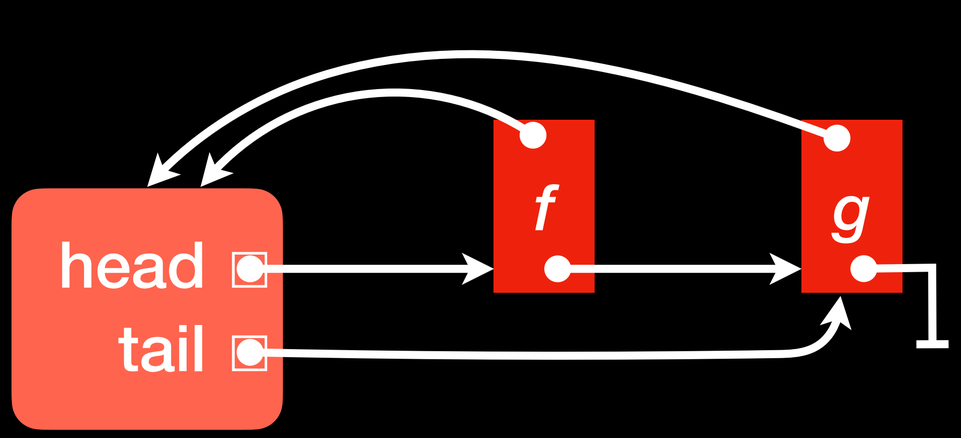
代表

- 每个子集被识别
  由一个代表的成员。
  当集合不变时，
  代表保持不变。

- 具体操作：
  - MAKE-SET(x)：为$\{x\}$创建结构
  - UNION(x,y)：将包含x和y的子集
    并合为一
  - FIND-SET(x)：查找包含x的子集的代表

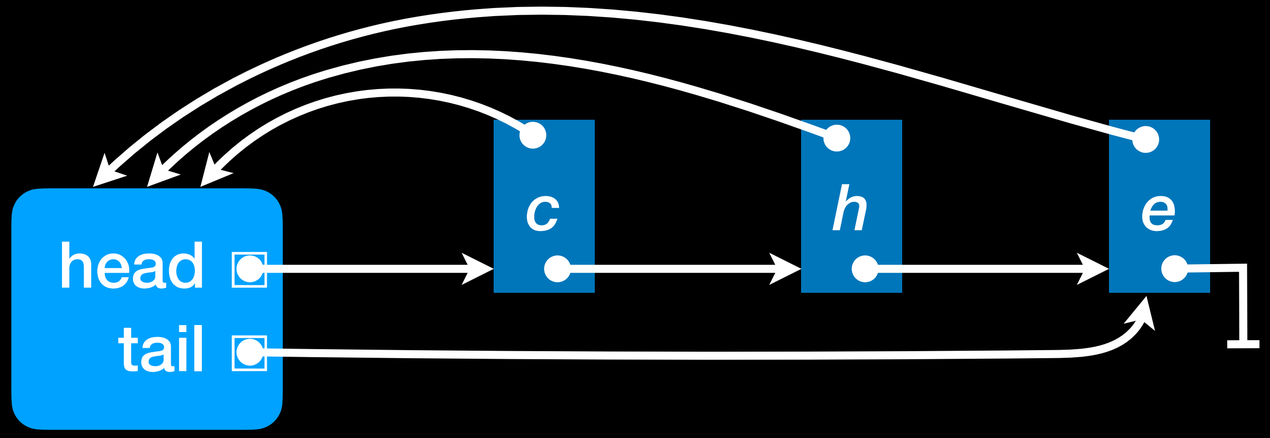
Simple Implementation: Linked Lists

- Store subset as linked list of elements
  representative = first element
- Example: sets {*f*,*g*} and {*c*,*h*,*e*}
- UNION(*g*,*c*)



简单实现：链表

- 为元素的链接列表
  代表是第一要素
- 示例：集合 {*f*,*g*}和{*c*,*h*,*e*}
- UNION(*g*,*c*)

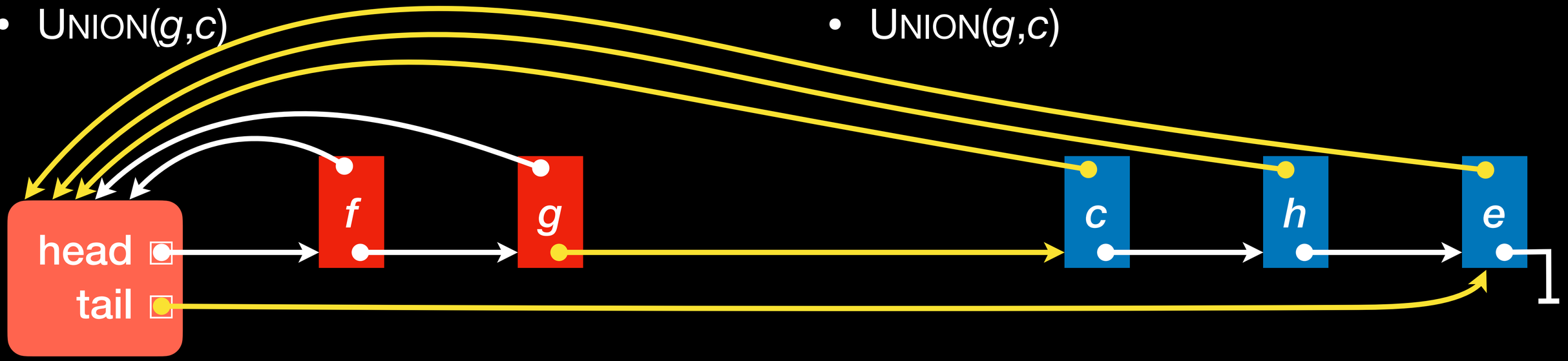


Simple Implementation: Linked Lists

- Store subset as linked list of elements
  representative = first element
- Example: sets {*f*,*g*} and {*c*,*h*,*e*}
- UNION(*g*,*c*)

简单实现：链表

- 为元素的链接列表
  代表是第一要素
- 示例：集合 {*f*,*g*}和{*c*,*h*,*e*}
- UNION(*g*,*c*)



Time Complexity

- Changing head/tail pointers is in $O(1)$, but changing pointers to the set object needs time Ω(size of the second subset).
- The aggregate time needed for a sequence of n operations may be up to $\Omega(n^2)$.

时间复杂性

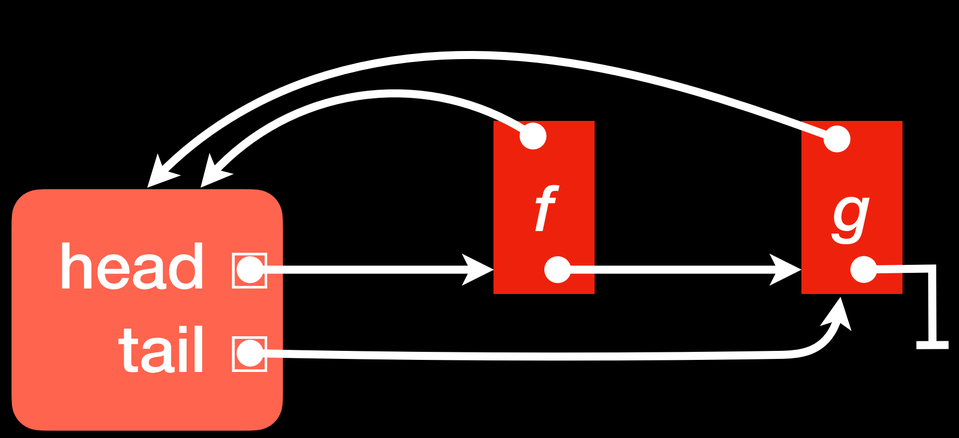
- 改变头/尾指针在$O(1)$中，但是更改指向集合对象的指针需要时间Ω(第二个子集的大小)。
- 所需的总时间对于n个操作序列可能高达$\Omega(n^2)$。

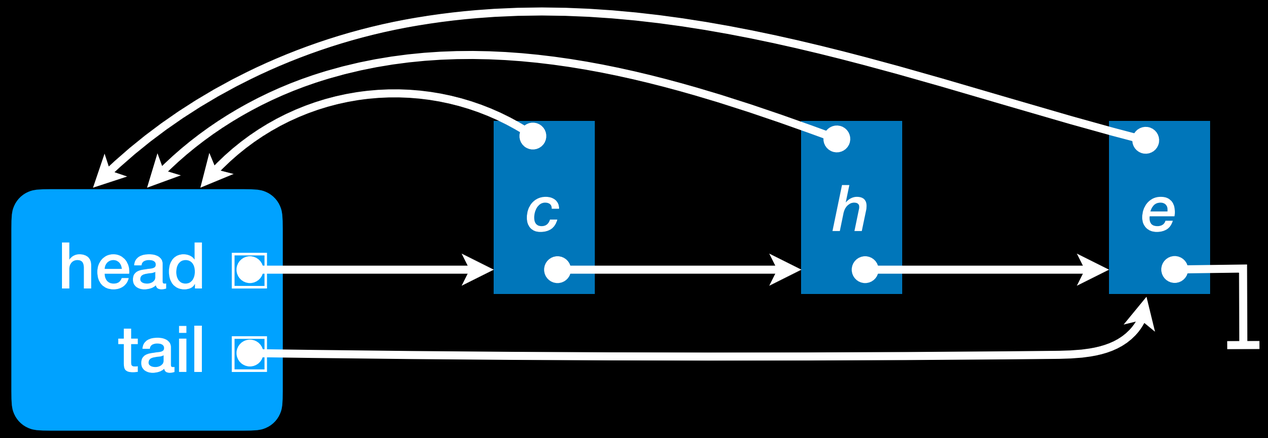
Improved Version

改进版

- Always add the smaller subset to the larger subset.

- 始终添加较小的子集到更大的子集。

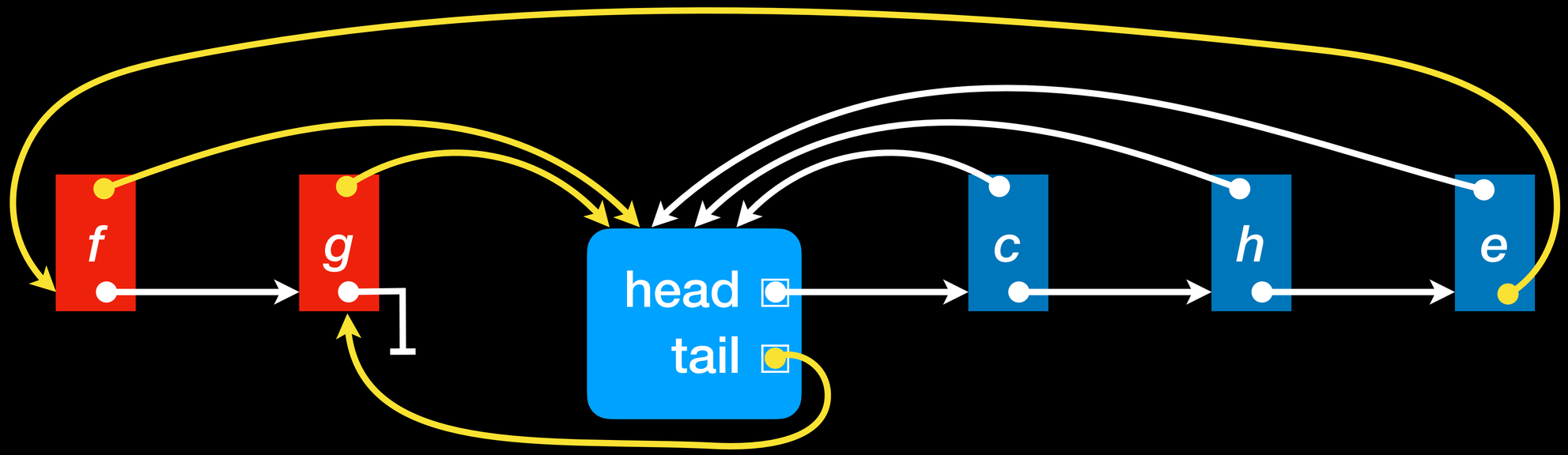




Improved Version

改进版

- Always add the smaller subset to the larger subset.
- 始终添加较小的子集到更大的子集。



Timing Anlysis of Improved version

- How often does the pointer of one element need to be changed?
- If x was in subset S_x and its pointer is changed to become part of subset S'_x, then
  $1 \le |S_x| \le \frac{1}{2}|S'_x| \le$ total number of elements
- x's pointer can be changed at most $\lfloor \lg(\text{total number of elements}) \rfloor$ times.
- n operations take time $O(n \log n)$.

改进版的时间复杂度

- 一个元素的指针需要更改几次?
- 如果x在子集S_x中，且其指针更改为子集S'_x的一部分，则
  $1 \le |S_x| \le \frac{1}{2}|S'_x| \le$ 元素总数
- x的指针最多可以更改 $\lfloor \lg(\text{元素总数}) \rfloor$ 次。
- n个操作需要时间$O(n \log n)$。

More Improvements

- The book presents a more sophisticated data structure to reach almost linear time $O(n\ \alpha(n))$.
- Not a topic of our course.
- We will use the disjoint-set data structure only as part of an algorithm that runs in time $\Theta(n \log n)$.

更多改进

- 书上出了一个更复杂的数据结构，以达到几乎线性时间$O(n\ \alpha(n))$。
- 这不是我们课程的主题。
- 我们只使用不相交集数据结构作为在时间$\Theta(n \log n)$中运行的算法的一部分。

Kruskal's Algorithm

- Idea:
  Always add the lightest/shortest edge that is allowed.
- Allowed are edges that do not form a cycle.
- Q: How do we test whether an edge is allowed?
  A: Check whether endpoints are in the same **subset.**

Kruskal的算法

- 想法：
  始终添加允许的最轻/最短边。
- 允许的边
  不会形成一个环路。
- 问：我们如何测试边是否允许的?
  答：检查一下端点在同一棵 **子集** 中。

Kruskal's Algorithm Kruskal的算法

```
MST-KRUSKAL(G,w)
A = ∅
for each vertex v ∈ G.V
        MAKE-SET(v)
sort the edges in G.E by weight w
for each edge (u,v) ∈ G.E in order of weight
        if FIND-SET(u) ≠ FIND-SET(v)
                A = A ∪ {(u,v)}
                UNION(u,v)
return A
```

every vertex forms a trivial tree

每个结点形成一棵平凡的树

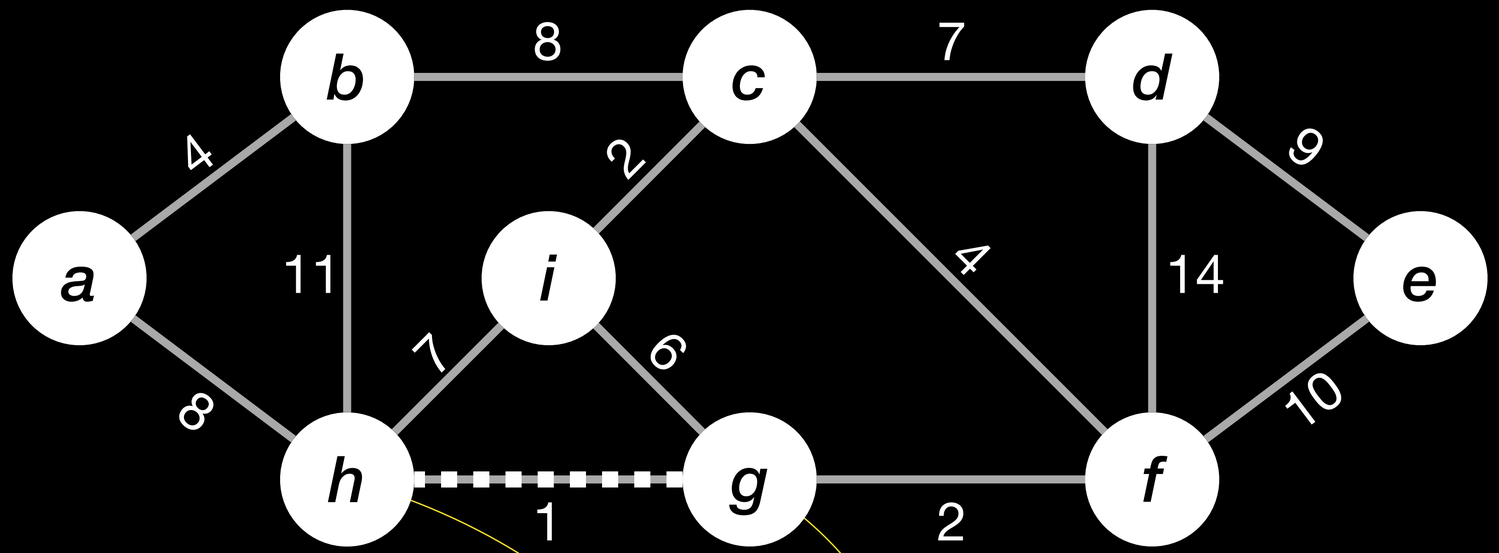
join trees until connected

连接树到连通起来

trees of *u* and *v* are not connected

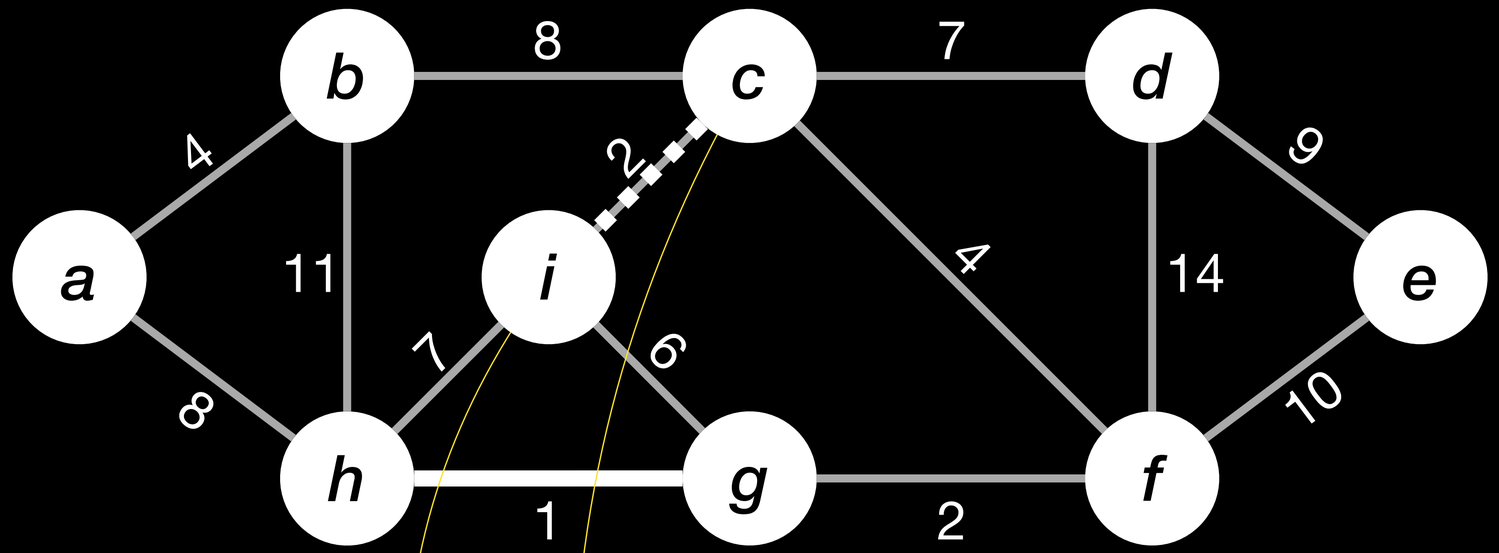
*u*和*v*的树不连通

Example 例子



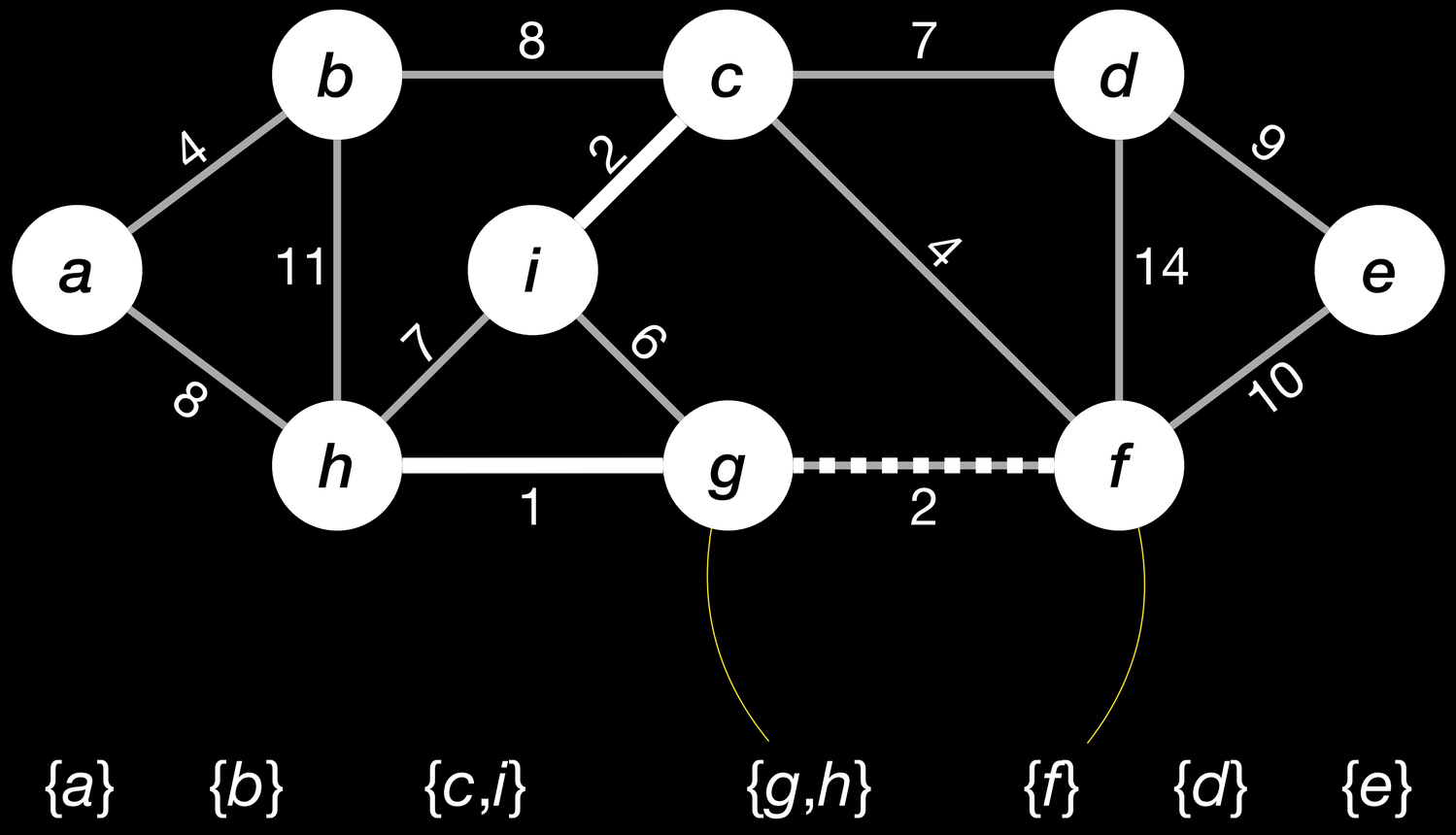
{a} {b} {i} {c} {h} {g} {f} {d} {e}

Example 例子

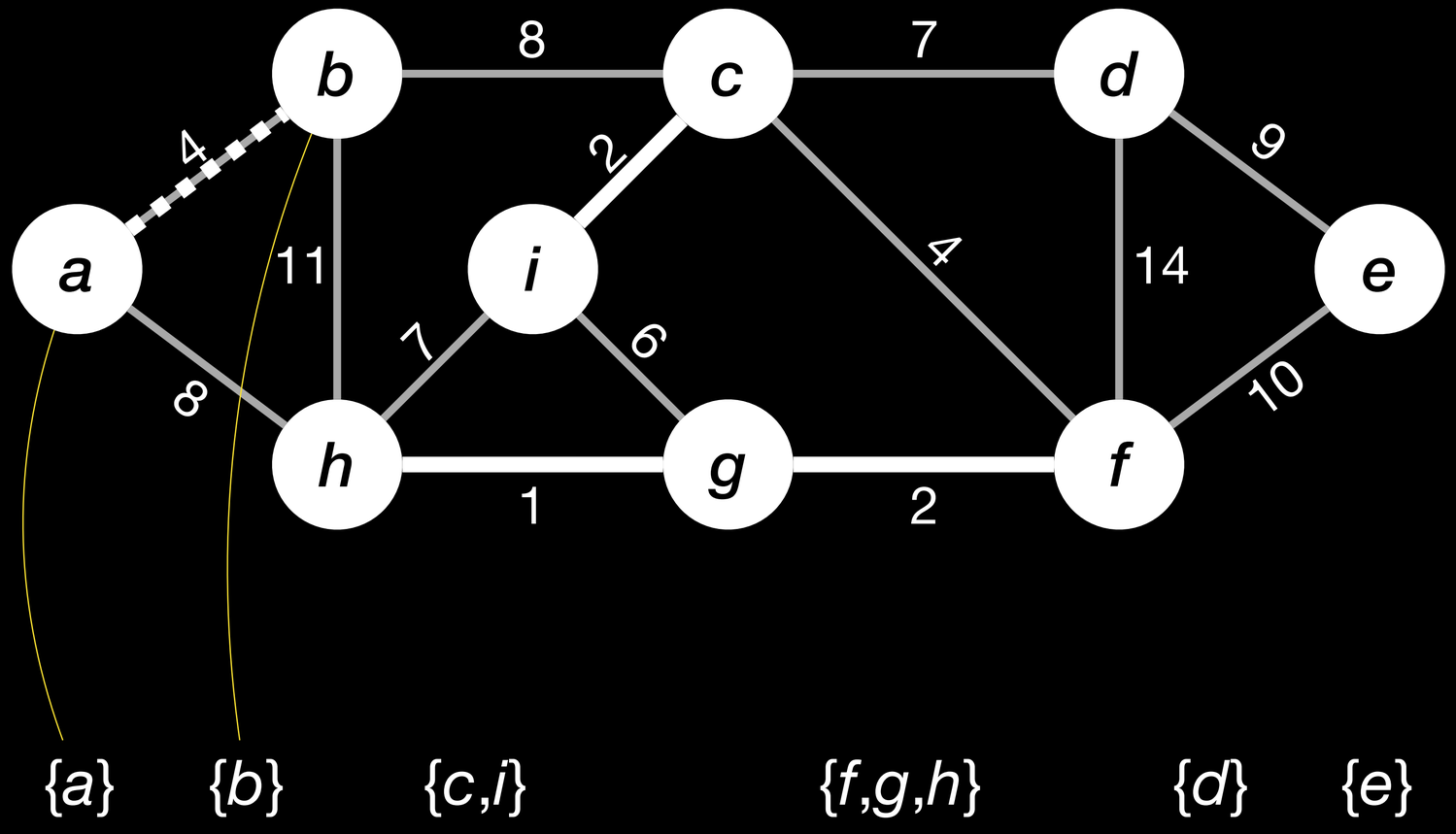


{*a*} {*b*} {*i*} {*c*} {*g*,*h*} {*f*} {*d*} {*e*}

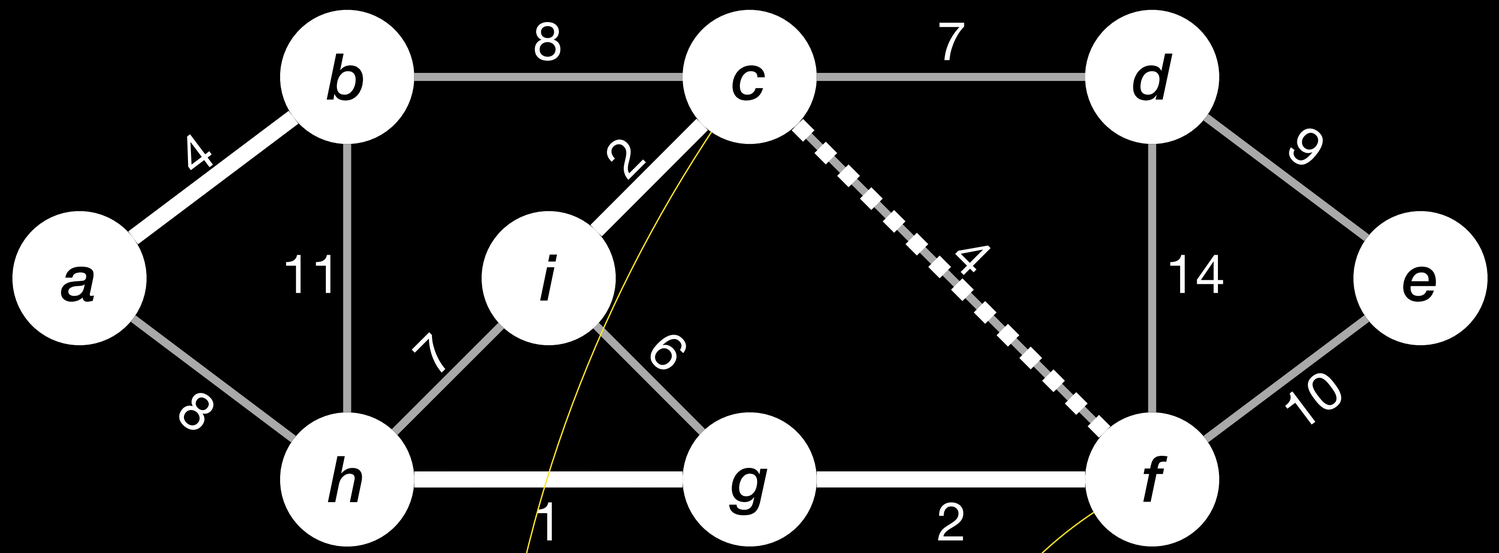
Example 例子



Example 例子

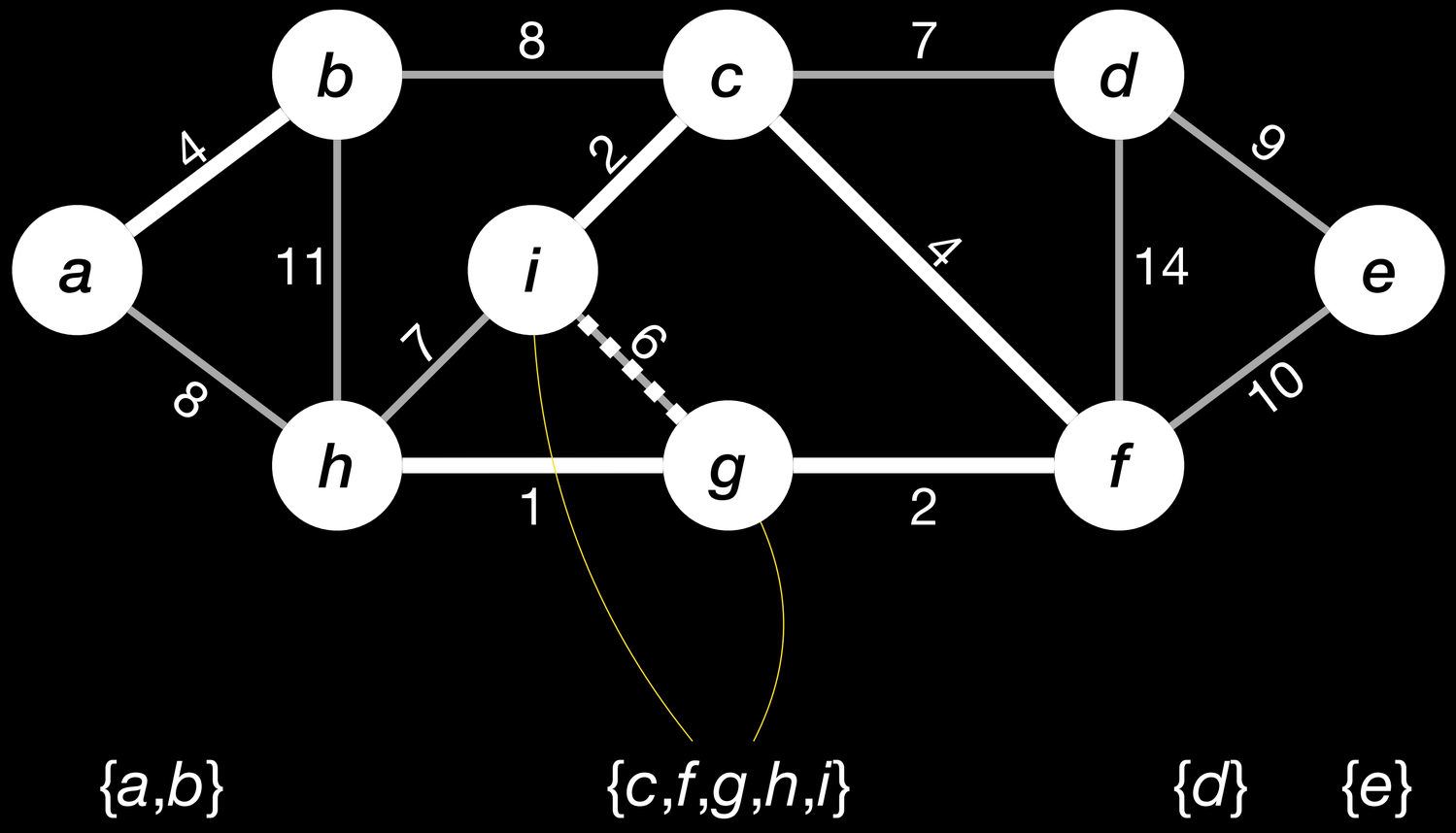


Example 例子

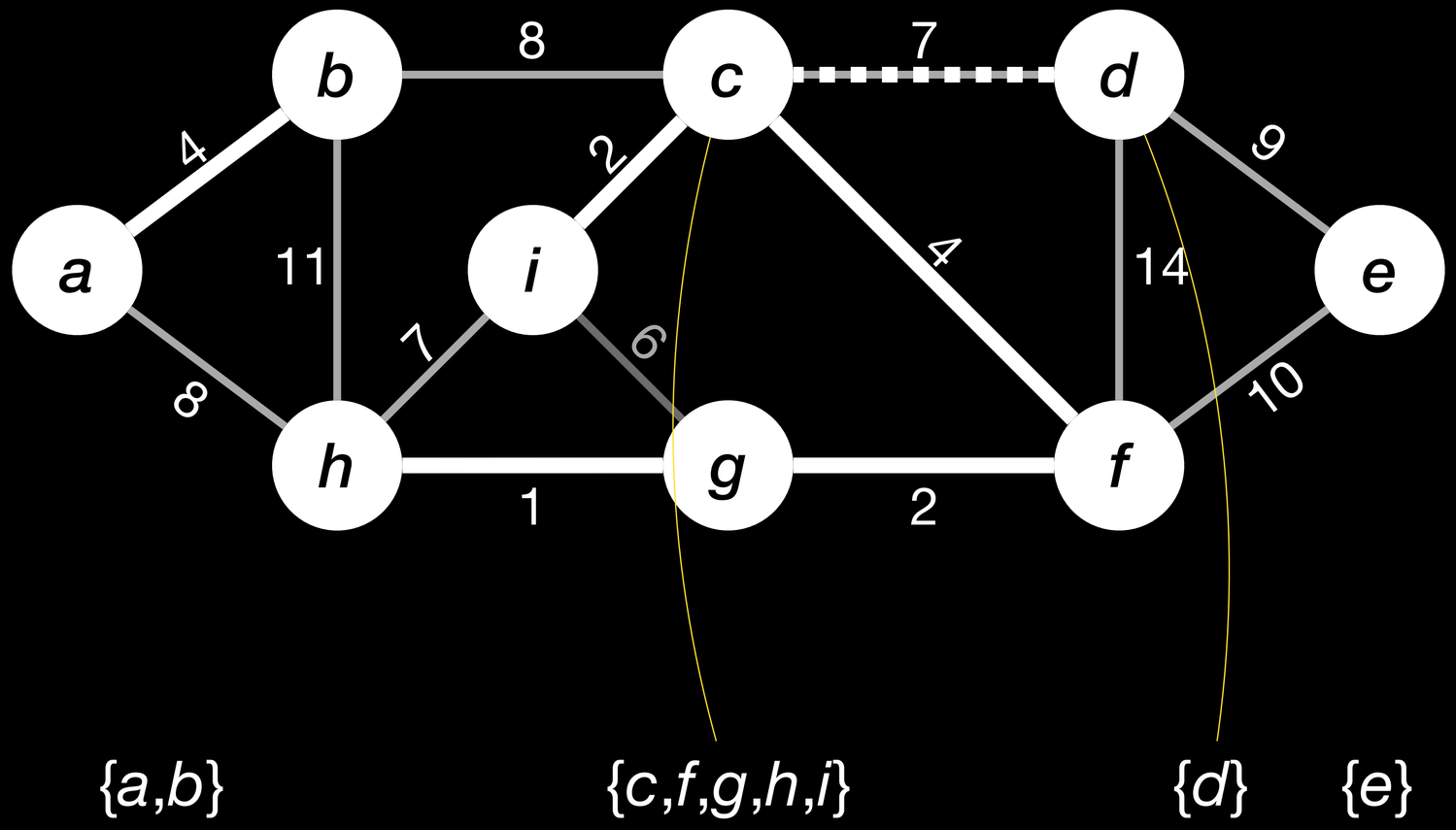


{*a*,*b*} {*c*,*i*} {*f*,*g*,*h*} {*d*} {*e*}

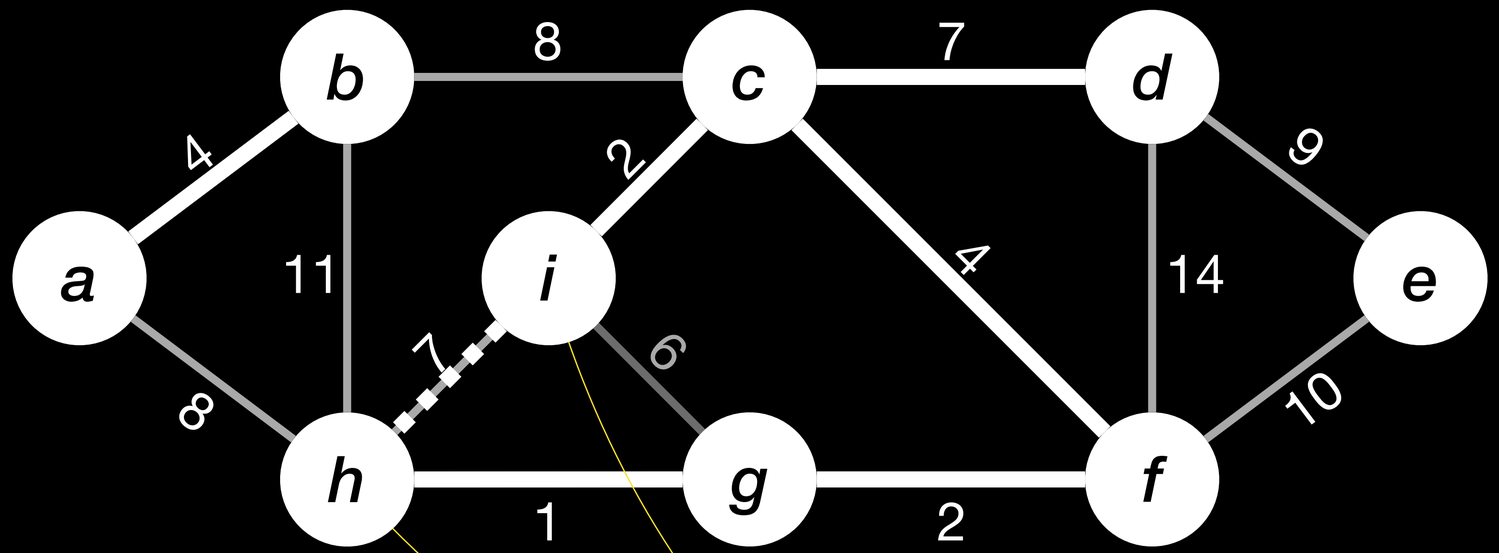
Example 例子



Example 例子

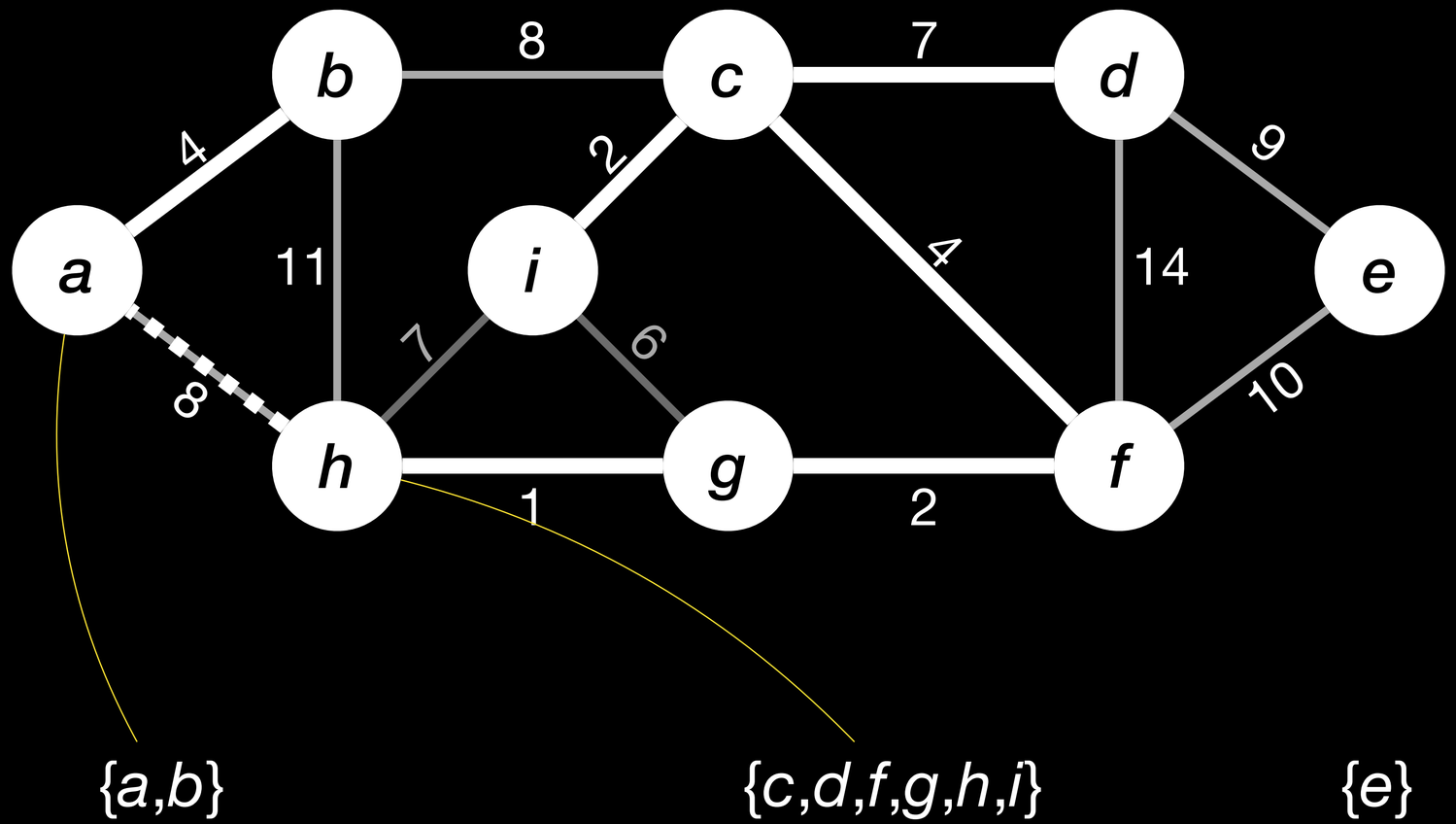


Example 例子

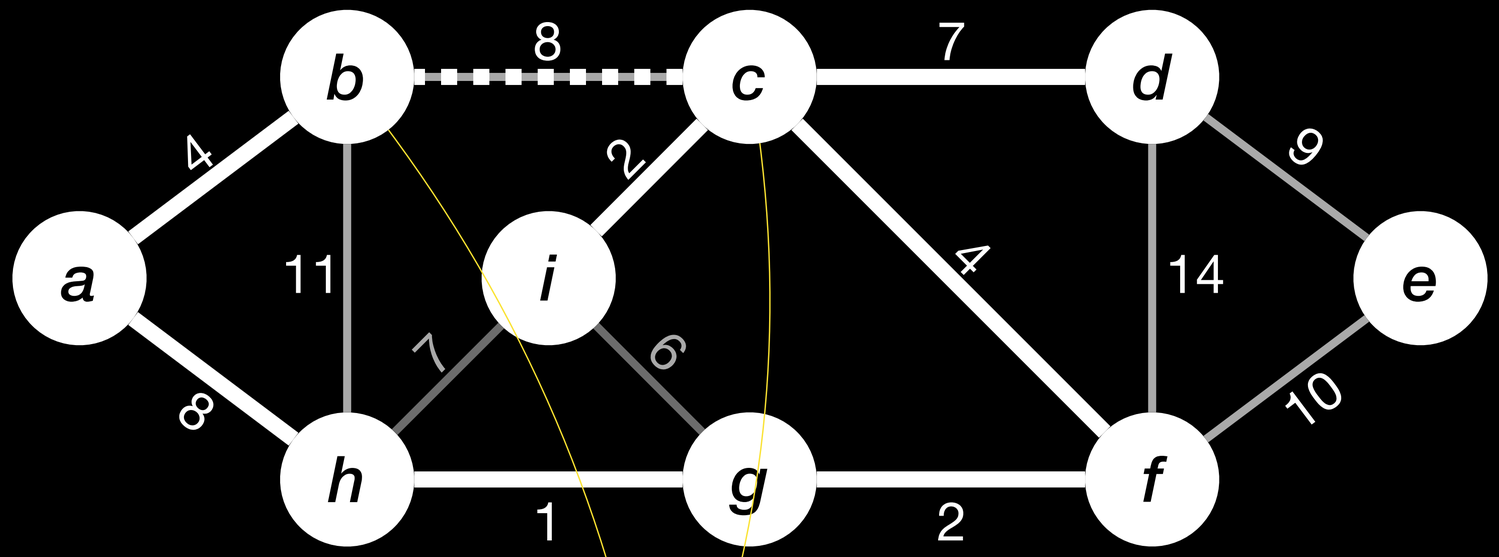


{*a*,*b*} {*c*,*d*,*f*,*g*,*h*,*i*} {*e*}

Example 例子

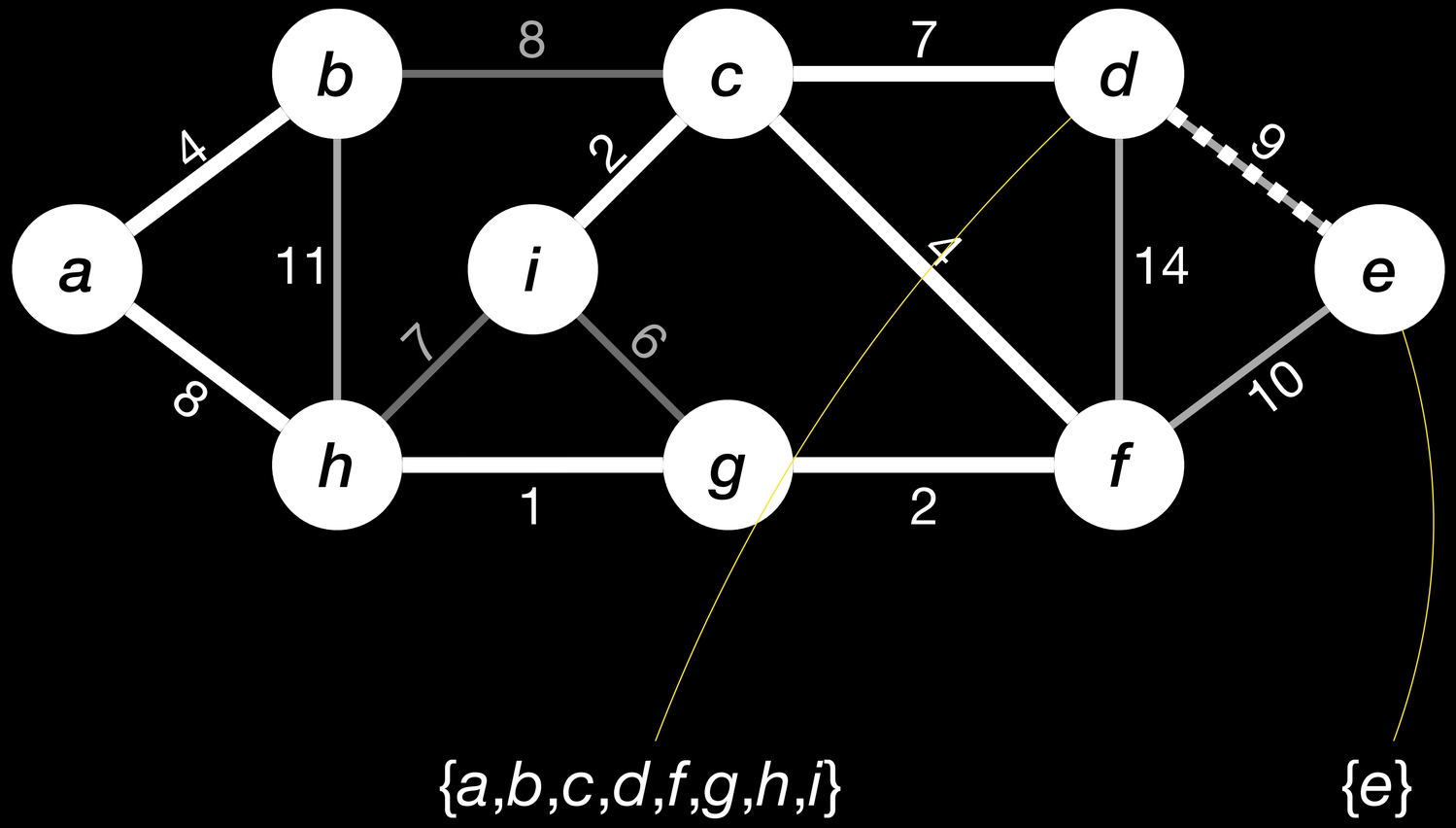


Example 例子

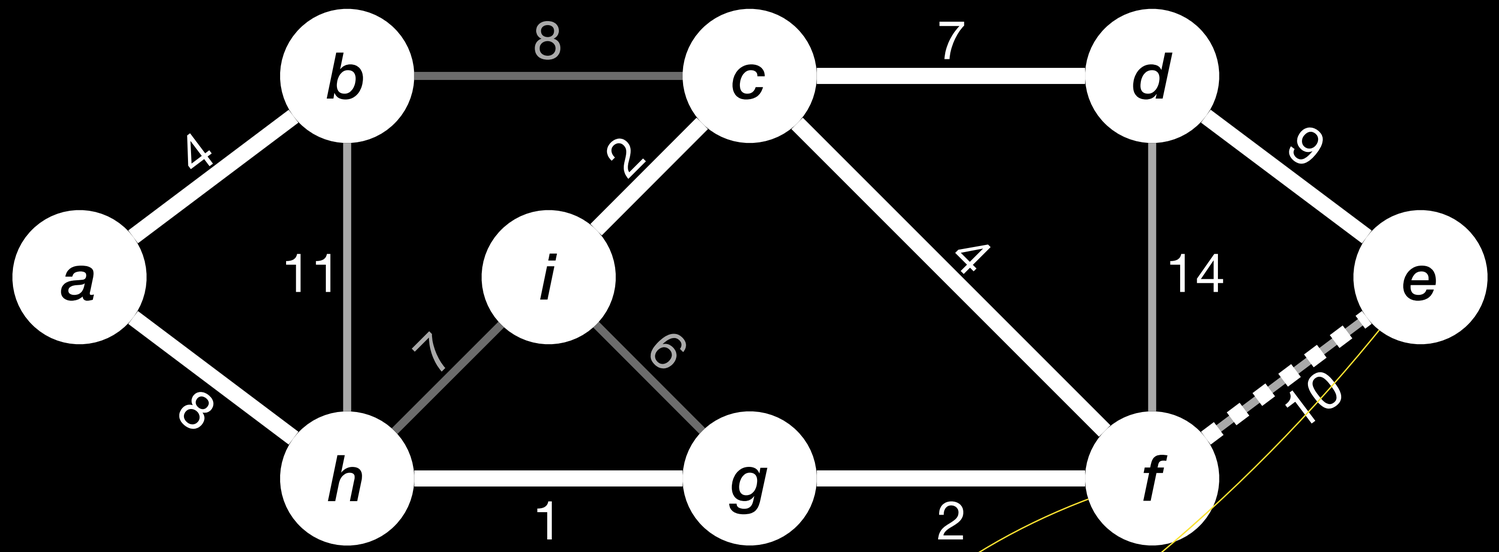


{*a*,*b*,*c*,*d*,*f*,*g*,*h*,*i*} {*e*}

Example 例子

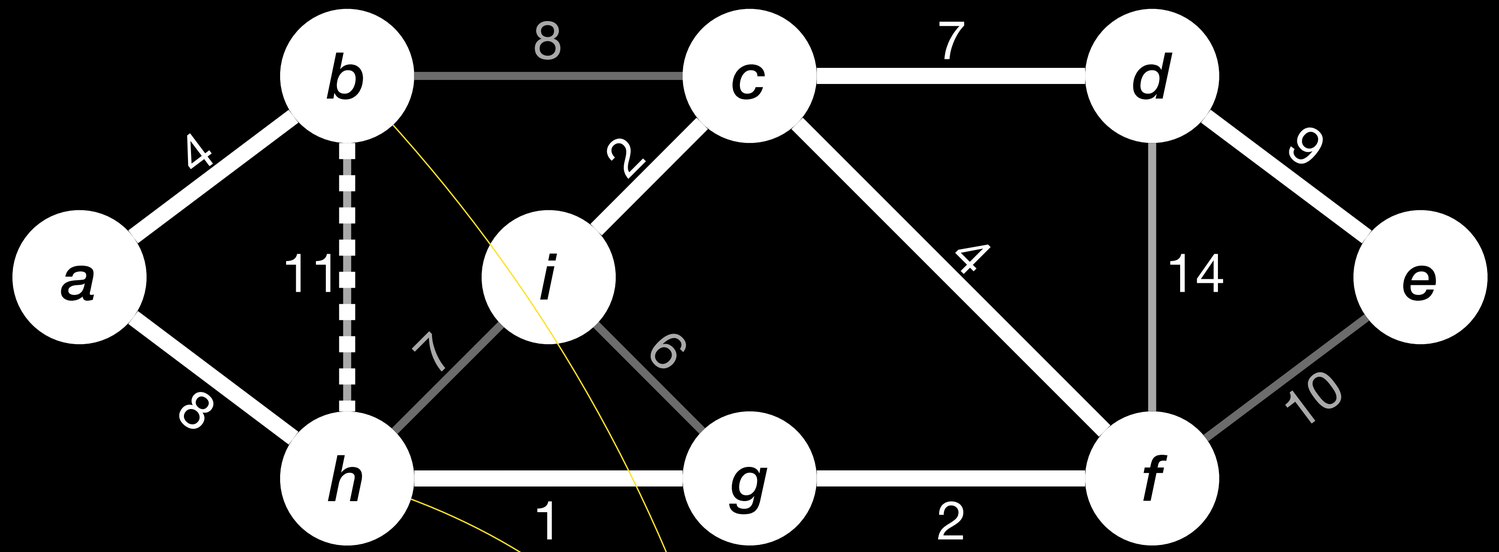


Example 例子



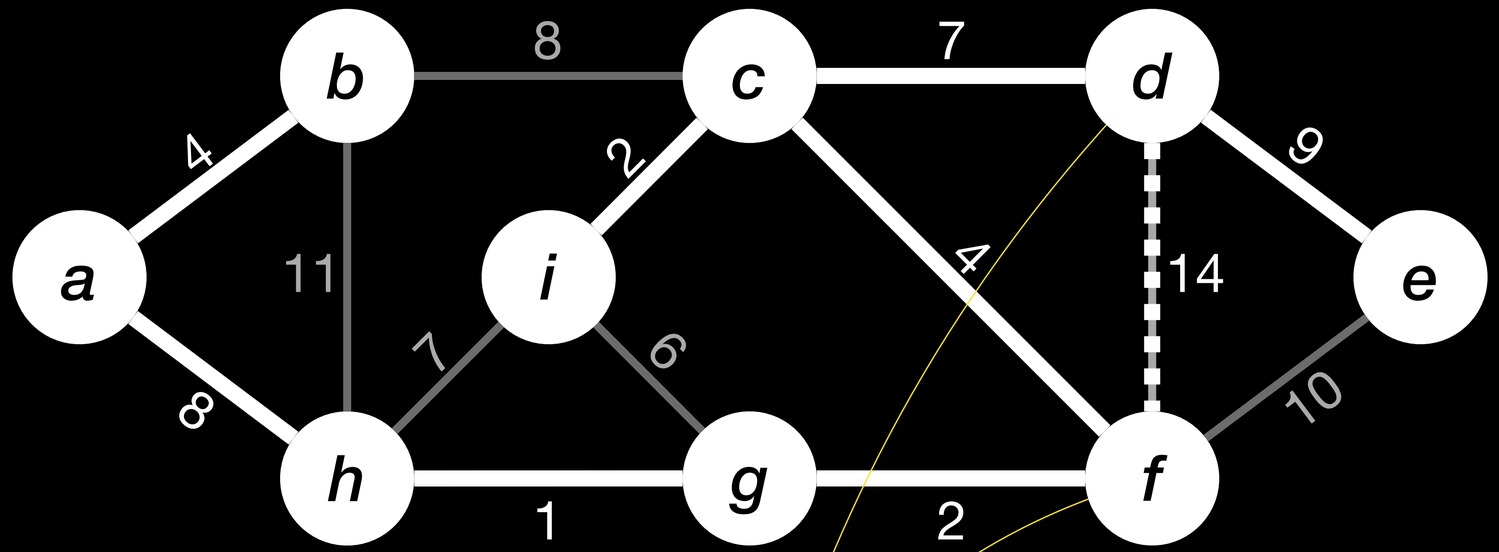
{*a*,*b*,*c*,*d*,*e*,*f*,*g*,*h*,*i*}

Example 例子



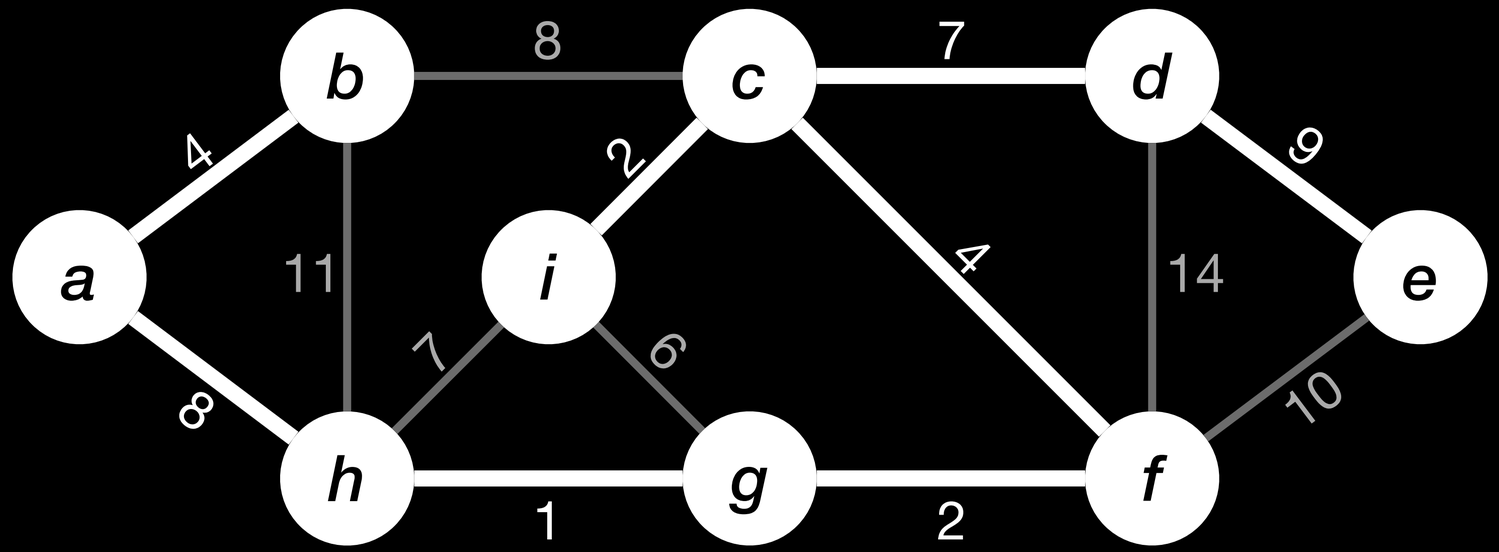
{*a*,*b*,*c*,*d*,*e*,*f*,*g*,*h*,*i*}

Example 例子



{*a*,*b*,*c*,*d*,*e*,*f*,*g*,*h*,*i*}

Example 例子



{*a*,*b*,*c*,*d*,*e*,*f*,*g*,*h*,*i*}

Partial Correctness

- immediate consequence of the partial correctness of the generic method.

- Find a cut that respects A:
  If edge endpoints are in different trees, let S = one of the trees.
  Then the edge is a light edge crossing the cut, so it is safe (Theorem 23.1).

  If edge endpoints are in the same tree, there is no cut that respects A.

部分正确性

- 通用方法部分正确性的直接后果。

- 找到尊重A的切割：
  如果边端点在不同的树中，
  则设 S = 其中一棵树。
  那么该边是横跨切割的轻量级边，
  所以是安全的（定理23.1）。

  如果边端点在同一棵树中，
  没有尊重A的切割。

Running Time

- Sorting edges takes time $O(|E| \log |E|)$.
  Note that $|E| \leq |V|^2$, so $\log |E| \leq 2 \log |V|$.
  We can also write:
  Sorting edges takes time $O(|E| \log |V|)$.

- We execute $O(|V| + |E|)$ disjoint set operations over the set V.
  Note that $|V| \leq |E|+1$ (if G is connected).
  Together, they take time in $O(|E| \log |V|)$.

- The total running time is in $O(|E| \log |V|)$.

运行时间

- 对边进行排序需要时间$O(|E| \log |E|)$。
  注意$|E| \leq |V|^2$，所以$\log |E| \leq 2 \log |V|$。
  我们还可以写：
  对边进行排序需要时间$O(|E| \log |V|)$。

- 我们在集合V上执行$O(|V| + |E|)$不相交集合操作。
  注意$|V| \leq |E|+1$（如果G连接）。
  它们一起在$O(|E| \log |V|)$中运行时间。

- 总运行时间为$O(|E| \log |V|)$。

Prim's Algorithm

- Idea:
  Only keep one (rooted) tree.
  Always extend the tree
  by the lightest edge that is allowed.

- Allowed are edges
  that do not form a cycle.

- To find the lightest edge,
  use a priority queue of nodes.
  key/priority = weight of shortest edge
  from tree to node.

Prim的算法

- 想法：
  只保留一棵（有根的）树。
  始终延伸树
  在允许的最轻量级边。

- 允许的边
  是不会形成一个环路。

- 寻找最轻量级边，
  使用结点的优先队列。
  关键字/优先度 = 从树到结点的
  最短边的权重

Priority Queue

- We discussed (max-)priority queues on 20 September.
- A min-priority queue has the operations:
  - INSERT(Q,x) inserts the element x into the queue Q
  - EXTRACT-MIN(Q) returns an element of Q with minimal key
  - DECREASE-KEY(Q,x,k) decreases the value of x's key to the new value k

优先队列

- （最大）优先队列是9月20号的课的内容。
- 最小优先队列有以下的操作：
  - INSERT(Q,x) 把元素 x 插入队列 Q 中
  - EXTRACT-MIN(Q) 去掉并返回 Q中的具有最小关键字的元素
  - DECREASE-KEY(Q,x,k) 将元素x的关键字值减少到k

Prim's Algorithm　　Prim的算法

```
MST-PRIM(G,w,r)
Q = ∅
for each vertex u ∈ G.V \ {r}
        u.key = ∞, INSERT(Q,u)
        u.π = NIL
r.key = 0, INSERT(Q,r)
r.π = NIL
while Q ≠ ∅
    {   u = EXTRACT-MIN(Q)                    }
        for each v ∈ G.V adjacent to u
            if v ∈ Q and w(u,v) < v.key
                v.π = u
                DECREASE-KEY(Q,v,w(u,v))
```

initialize priority queue

初始化优先队列

ensure that the tree starts with *r*

确保树以*r*开头

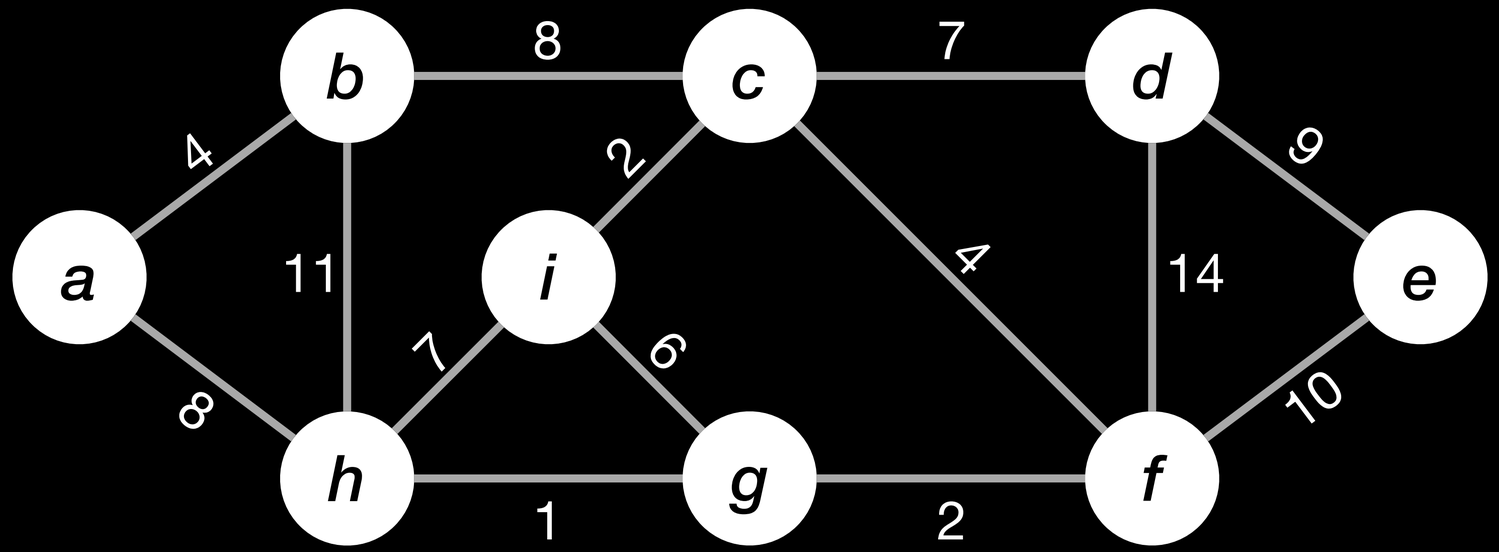
add *u* to the tree

把*u*添加到树中

correct keys of vertices adjacent to *u*

更正与*u*相邻顶点的关键

Example 例子



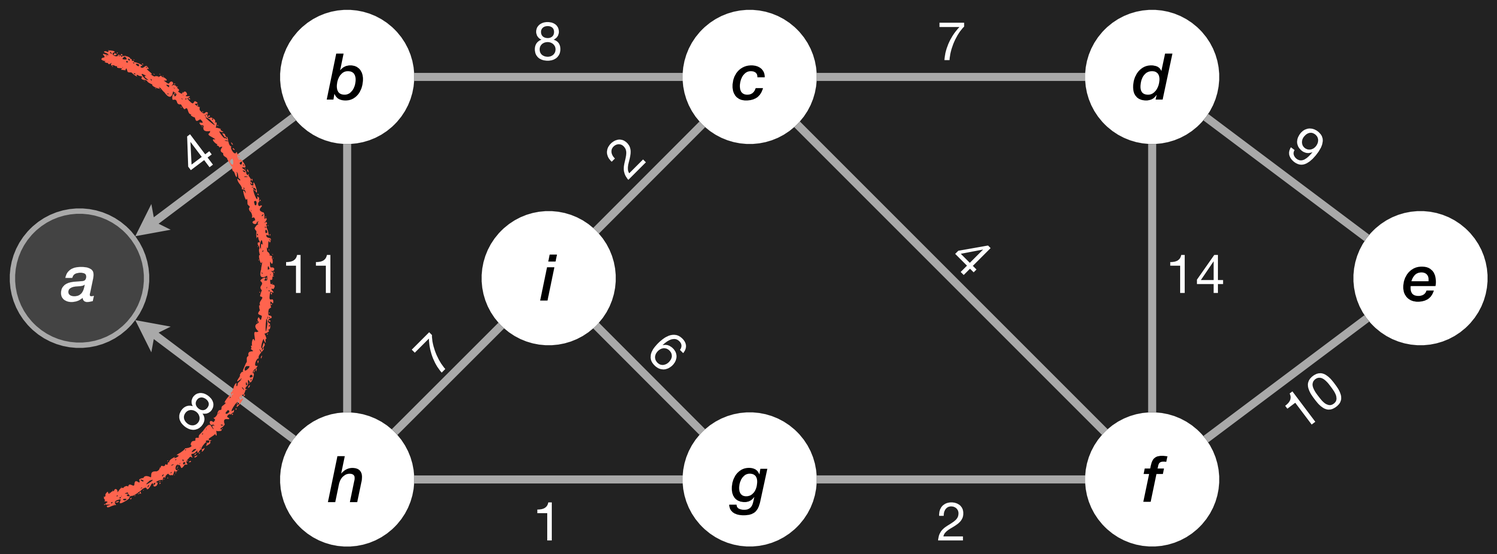
0

a

∞

b c d e f g h i

Example 例子

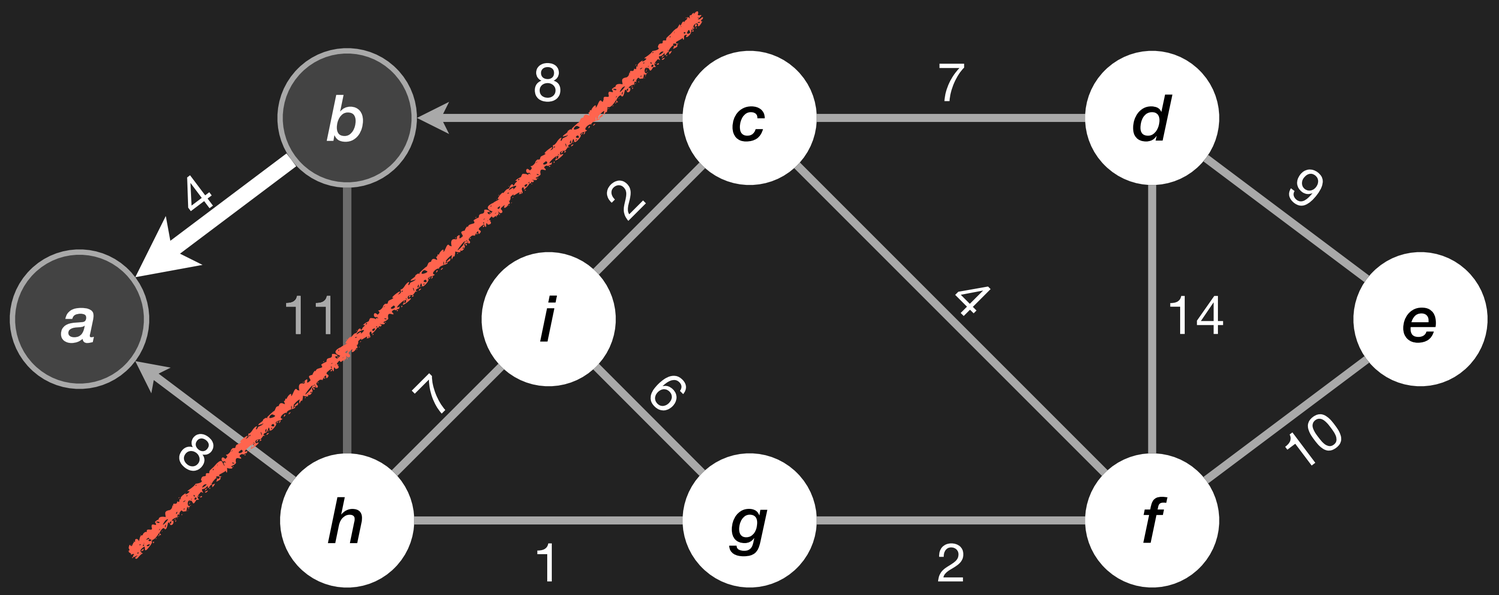


4
b

8
h

∞
c d e f g i

Example 例子



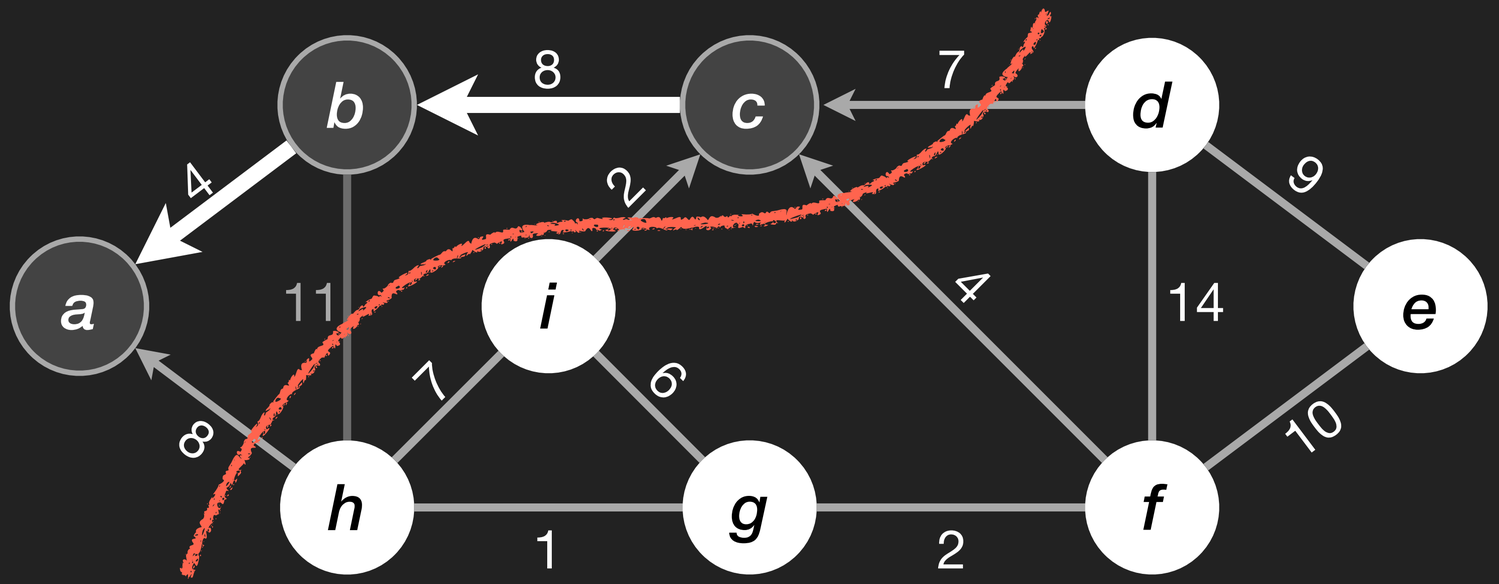
8

c h

∞

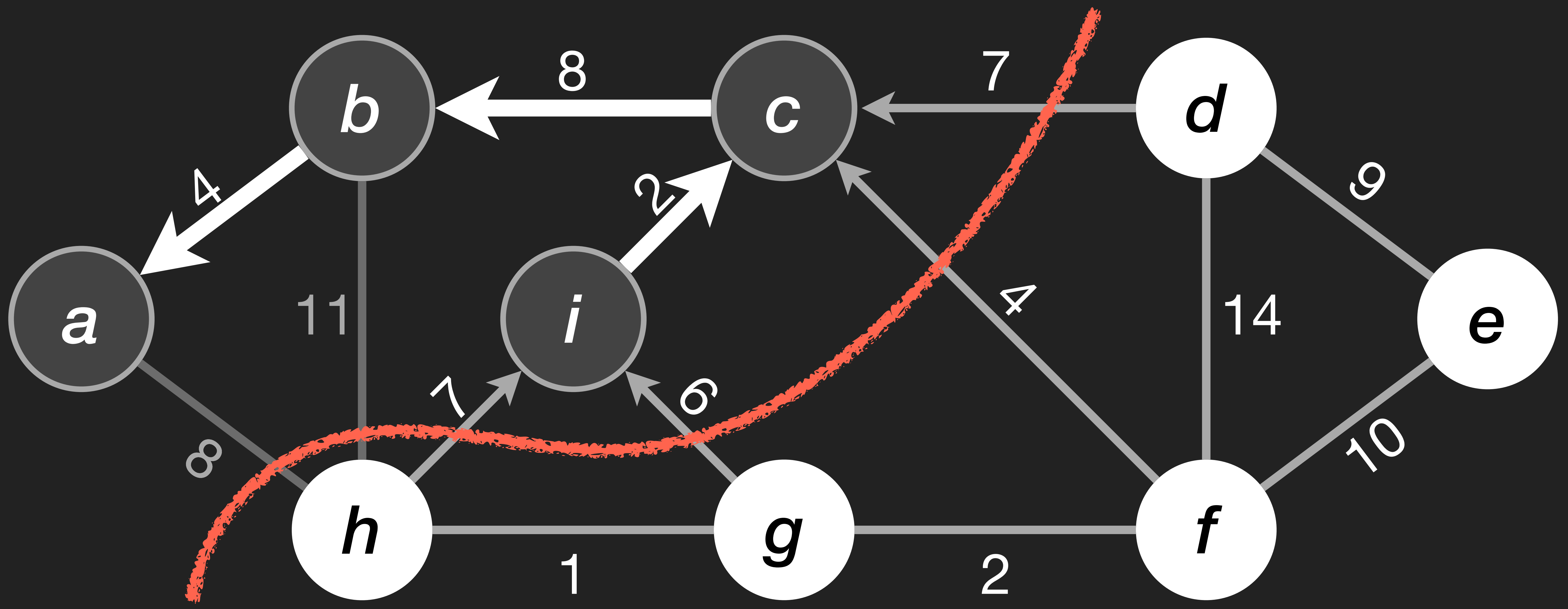
d e f g i

Example 例子



2		**4**		**7**	**8**				**∞**
i		*f*		*d*	*h*				*e* *g*

Example 例子



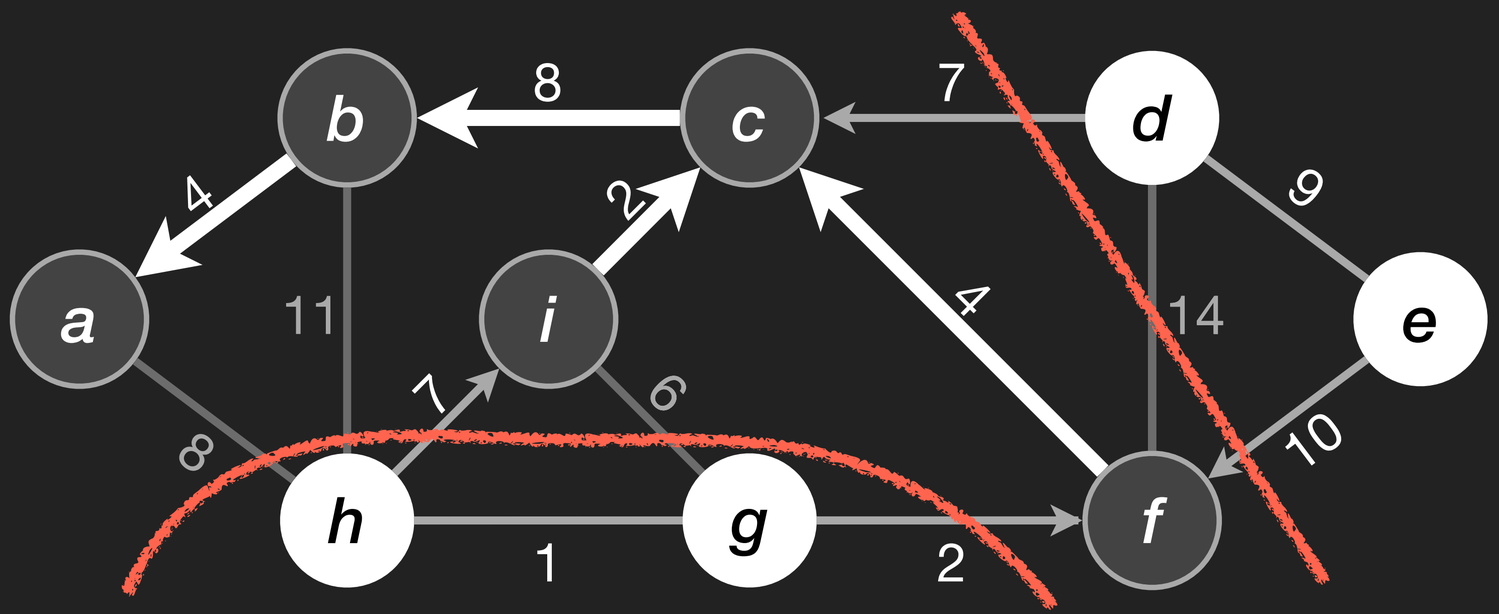
4 *f*

6 *g*

7 *d* *h*

∞ *e*

Example 例子

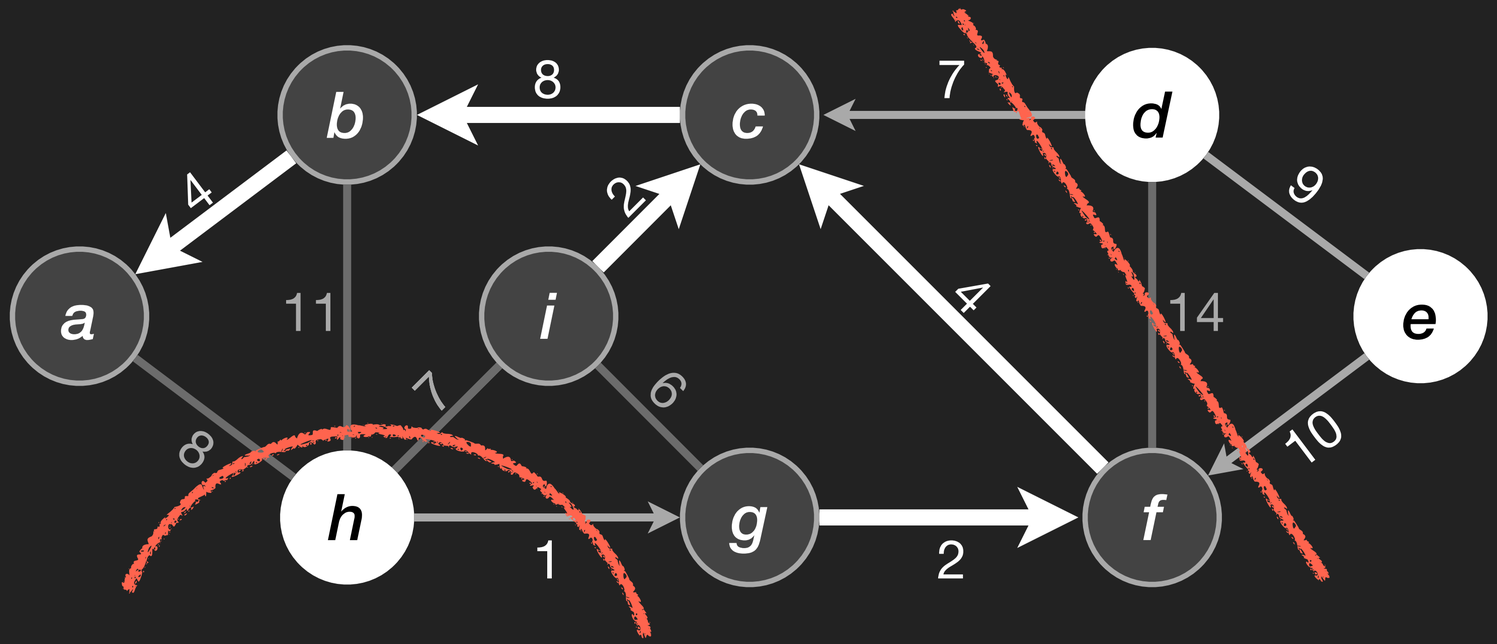


2
g

7
d h

10
e

Example 例子

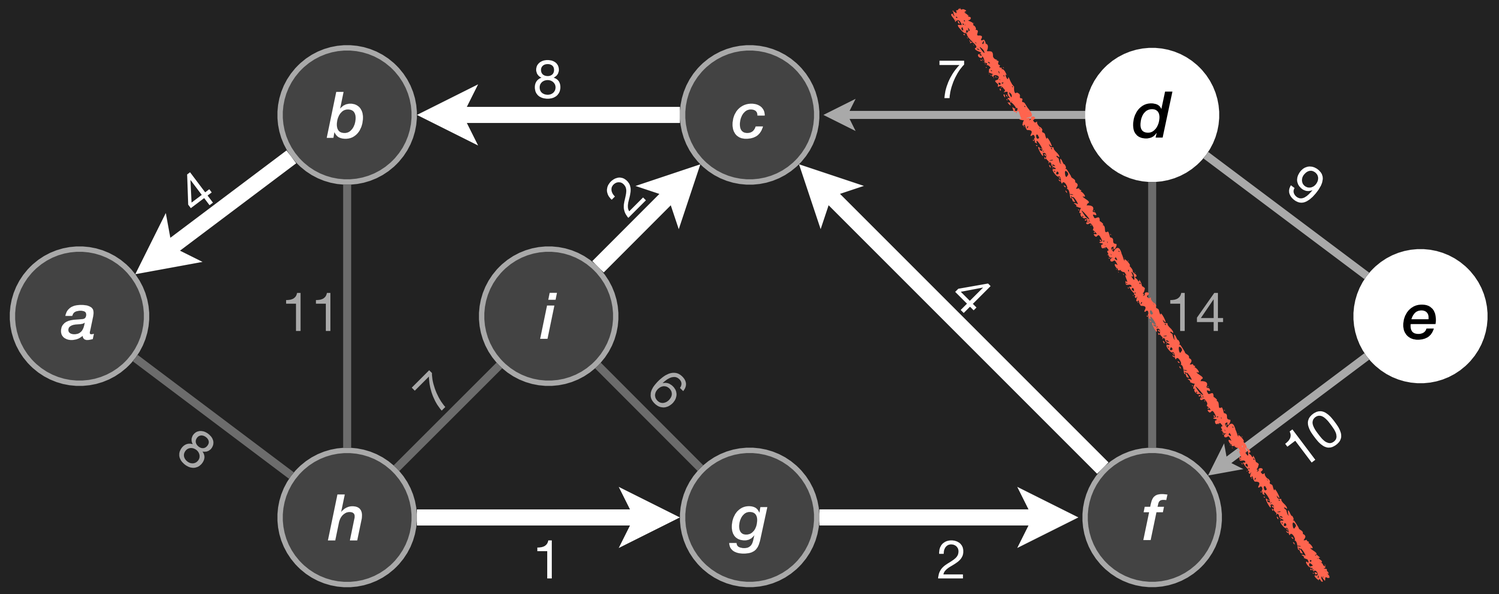


1
h

7
d

10
e

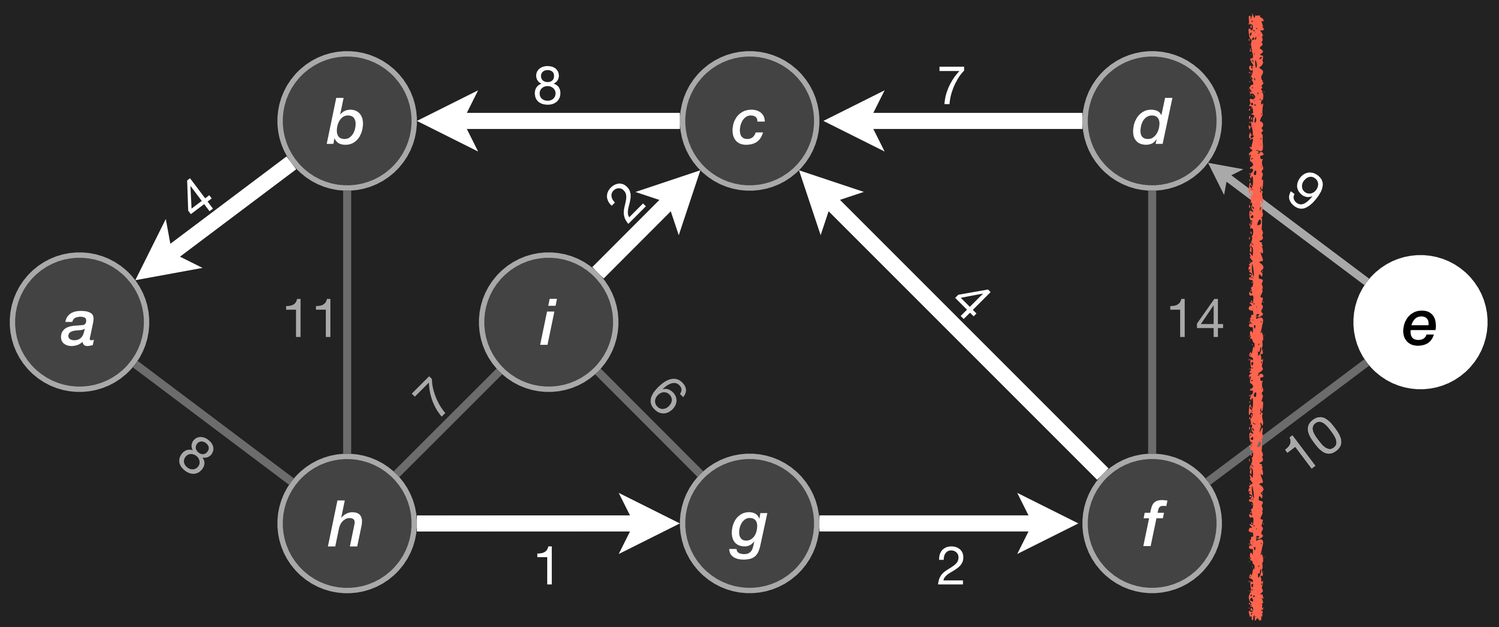
Example 例子



7
d

10
e

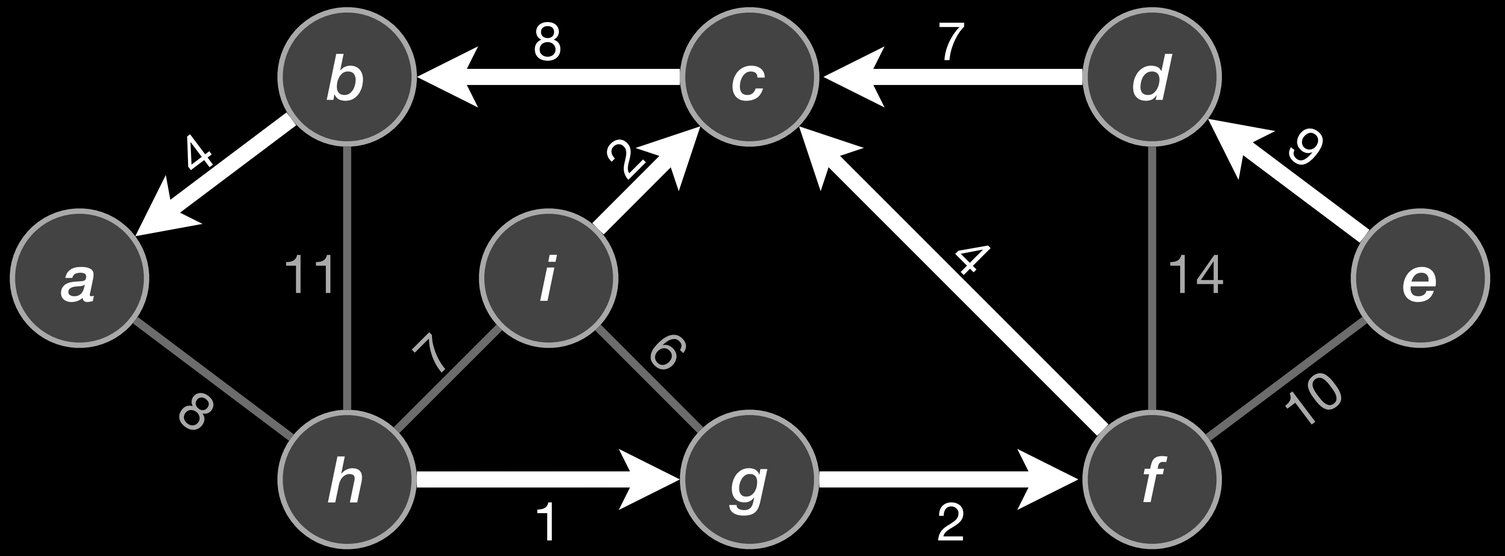
Example 例子



9

e

Example 例子



$Q = \varnothing$

Partial Correctness

- immediate consequence of the partial correctness of the generic method.
- The (partial) tree consists of
  the vertices $V \setminus Q$ and
  the edges $(v, v.\pi)$ for $v \in V \setminus Q$, $v \neq r$.
- Find a cut that respects A:
  S = the vertices that are in the tree
  $= V \setminus Q$.

部分正确性

- 通用方法部分正确性的直接后果。
- （部分）树包括
  顶点$V \setminus Q$和
  边 $(v, v.\pi)$ for $v \in V \setminus Q$，$v \neq r$
- 找到尊重 A 的切割：
  S = 树中的顶点
  $= V \setminus Q$。

Running Time

- Assume that the priority queue uses a min-heap.
- The initialization of the (very simple) priority queue takes time in $O(|V|)$.
- We execute $|V|$ times EXTRACT-MIN, which take total time $O(|V| \log |V|)$.
- We execute $O(|E|)$ times DECREASE-KEY, which take total time $O(|E| \log |V|)$.
- The total running time is in $O(|E| \log |V|)$.

运行时间

- 假设优先级队列使用最小堆。
- （非常简单）优先级队列的初始化需要$O(|V|)$的时间。
- 我们执行 $|V|$ 次EXTRACT-MIN，它占用总时间$O(|V| \log |V|)$。
- 我们执行 $O(|E|)$ 次DECREASE-KEY，总时间为$O(|E| \log |V|)$。
- 总运行时间以$O(|E| \log |V|)$为单位。

Algorithm Design and Analysis

Single-Source Shortest Paths

David N. JANSEN

名 姓

算法设计与分析

单源最短路径

杨大卫

What are “Single-Source Shortest Paths”?

- Example:
  The tourist office wants to know
  the distance from Hangzhou Station
  to every tourist attraction in the city,
  to display this information at the station.

单源最短路径是什么？

- 例如：
  旅游局想知道从杭州城站
  到每个市内的旅游胜地的距离，
  在站里显示这个信息。

What are “Single-Source Shortest Paths”?

Input: a weighted graph $G = (V,E)$, $w:E \to \mathbb{R}$ and a source vertex $s \in G.V$

Output: for every vertex $d \in G.V$:

- the shortest distance $\delta(s,d)$
- a shortest path $s \rightsquigarrow d$
  (stored efficiently as: the predecessor vertex of d on the shortest path, denoted $d.\pi$.)

单源最短路径是什么？

输入： 权重的有向图$G = (V,E)$, $w:E \to \mathbb{R}$
源结点$s \in G.V$

输出： 为所有的结点$d \in G.V$：

- 最短距离$\delta(s,d)$
- 一条最短路径$s \rightsquigarrow d$
  （高效地存储为：
  最短路径d的前驱结点$d.\pi$,
  最短路径为$s \rightsquigarrow d.\pi \to d$）

Vocabulary and General Properties

Some researchers only work on shortest paths!
There is a body of specialized knowledge,
and there are some special terms.

In this part:

- optimal substructure
- negative-weight edges
- cycles
- relaxation
- shortest-path algorithm invariants (general properties)

词汇与一般性质

有科学家全职地研究最短路径算法！
有一套专门的知识体系，
还有一些特殊的术语。

在这个部分：

- 最优子结构
- 负权重的边
- 环路
- 松弛操作
- 最短路径算法的性质（一般性质）

Optimal Substructure

- Subpaths of shortest paths are shortest paths.
- **Lemma 24.1:** Given a weighted, directed graph $(G = (V,E), w)$.
  Let $p = \langle v_0, v_1, \ldots, v_k \rangle$ be
  a shortest path from v_0 to v_k.
  Then, for any $0 \le i \le j \le k$,
  the subpath $p_{ij} = \langle v_i, v_{i+1}, \ldots, v_{j-1}, v_j \rangle$ is
  a shortest path from v_i to v_j.
- Proof: Let p' be a path from v_i to v_j that is shorter than p_{ij}. Then, $\langle v_0, \ldots, v_{i-1}, p', v_{j+1}, \ldots, v_k \rangle$ is a path from v_0 to v_k that is shorter than p.
  Contradiction!

最优子结构

- 最短路径的子路径也是最短路径。
- **引理24.1：** 给定带权重的有向图 $(G = (V,E), w)$。
  设 $p = \langle v_0, v_1, \ldots, v_k \rangle$ 为
  从结点 v_0 到结点 v_k 的一条最短路径。
  那么对于任何的 $0 \le i \le j \le k$,
  子路径 $p_{ij} = \langle v_i, v_{i+1}, \ldots, v_{j-1}, v_j \rangle$ 是
  从结点 v_i 到结点 v_j 的一条最短路径。
- 证明：假设 p' 是从结点 v_i 到结点 v_j 的路径，并且比 p_{ij} 更短。那么 $\langle v_0, \ldots, v_{i-1}, p', v_{j+1}, \ldots, v_k \rangle$ 是比 p 更短的从结点 v_0 到结点 v_k 的路径。
  矛盾！

Optimal Substructure

- Subpaths of shortest paths are shortest paths.
- **Lemma 24.1:** Given a weighted, directed graph $(G = (V,E), w)$.
  Let $p = \langle v_0, v_1, \ldots, v_k \rangle$ be
  a shortest path from v_0 to v_k.
  Then, for any $0 \le i \le j \le k$,
  the subpath $p_{ij} = \langle v_i, v_{i+1}, \ldots, v_{j-1}, v_j \rangle$ is
  a shortest path from v_i to v_j.
- This makes the greedy method (Dijkstra) and dynamic programming (Floyd–Warshall, tomorrow) work.

最优子结构

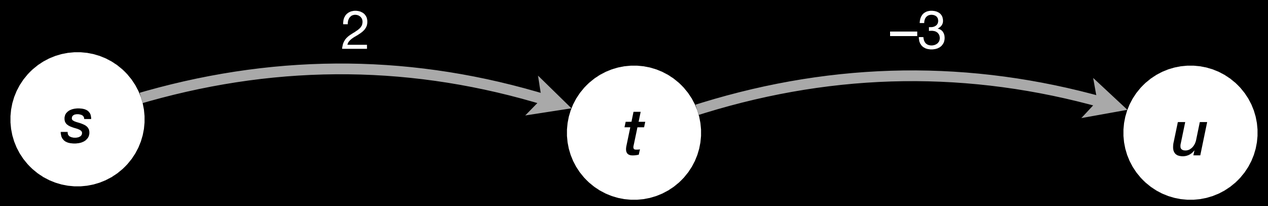
- 最短路径的子路径也是最短路径。
- **引理24.1：** 给定带权重的有向图 $(G = (V,E), w)$。
  设 $p = \langle v_0, v_1, \ldots, v_k \rangle$ 为
  从结点 v_0 到结点 v_k 的一条最短路径。
  那么对于任何的 $0 \le i \le j \le k$,
  子路径 $p_{ij} = \langle v_i, v_{i+1}, \ldots, v_{j-1}, v_j \rangle$ 是
  从结点 v_i 到结点 v_j 的一条最短路径。
- 因此，贪心算法（Dijkstra）和动态规划（Floyd–Warshall，明天）发挥作用。

Edges with Negative Weight

负值权重的边

- Some graphs allow edges with negative weight.
  [Please give an example.]

- 有些图的边可能有负值的权重。
  [请给出个负权重图的例子。]



The shortest path from *s* to *u* has weight –1.

从*s*到*u*的最短的路径权重为–1。

Edges with Negative Weight

- Some graphs allow edges with negative weight.
- Dijkstra's algorithm requires that there are no edges with negative weight.

负值权重的边

- 有情况一个图的边可能有负值的权重。
- Dijkstra的算法要求没有负值权重的边。

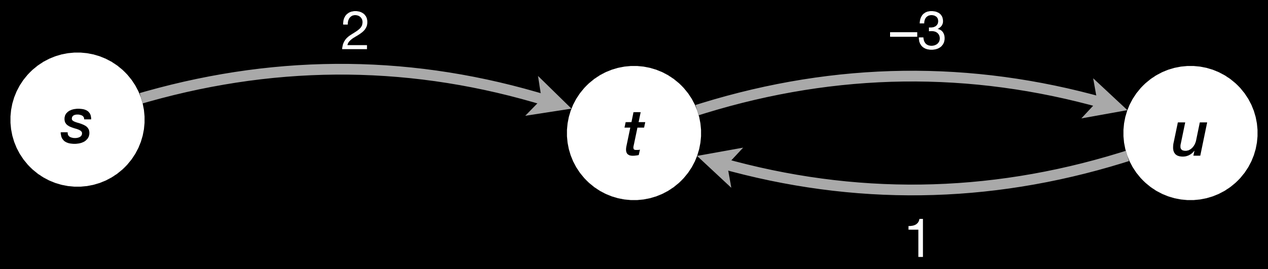
Edges with Negative Weight

负值权重的边

- Negative-value cycles never make sense.
  [Check with the examples what a negative-value cycle would ensue.]

- 负值权重的环路没有意思。

  [在你的例子中，负权重的环路是什么？]



- There is no shortest path from *s* to *t*:
  For every path *p*, the path ⟨*p*,*u*,*t*⟩ is even shorter.

- 不存在最短从*s*到*t*的路径：
  对于任何路径*p*，那么路径⟨*p*,*u*,*t*⟩更短。

Cycles (with non-negative weight)

Can a shortest path contain a cycle?

- Negative-weight cycle: Shortest path does not exist.
- Positive-weight cycle: Removing cycle makes path shorter; not a shortest path.
- Zero-weight cycle: Removing cycle doesn't change weight; both paths are shortest.

Conclusion: If a shortest path exists, then there exists an acyclic/simple shortest path.

➥ One can store shortest paths efficiently with predecessors $v.\pi$.

（无负值的）环路

最短路径可以包括环路吗?

- 负值的环路：不存在最短路径
- 正值的环路：去除环路变到更短的路径
  有正值的环路不可能最短
- 零值的环路：去除环路不改变权重
  两个路径都最短的

结论： 如果存在最短路径，
那么存在无环的最短路径。

➥ 可以用前驱$v.\pi$高效地存储最短路径。

Relaxation

- For every vertex v, store $v.d$ = length of shortest path found until now = estimate for $\delta(s,v)$
- "Relaxing" edge (u,v) = Testing whether this edge can be used to improve the shortest-path estimate $v.d$.
- Relaxation is the basic step of shortest-path algorithms.

松弛操作

- 为每个边存储$v.d$ = 已找到的最短路径的权重 = $\delta(s,v)$的估计
- "松弛"边(u,v) = 判断这条边能不能改进$v.d$的估计
- 松弛是最短路径的算法的基本操作。

If the shortest known $s \rightsquigarrow v$ is longer than $s \rightsquigarrow u \rightarrow v$,

then $s \rightsquigarrow u \rightarrow v$ becomes the new shortest known path to v.

RELAX(u,v,w)
if $v.d > u.d + w(u,v)$
$$\begin{cases} v.d = u.d + w(u,v) \\ v.\pi = u \end{cases}$$

如果已找到的$s \rightsquigarrow v$比$s \rightsquigarrow u \rightarrow v$长,

那么$s \rightsquigarrow u \rightarrow v$是新的已找到的最短到v的路径.

Initialization 初始化

INITIALIZE-SINGLE-SOURCE(G,s)

for each vertex $v \in G.V$

$v.d = \infty$

$v.\pi = \text{NIL}$

$s.d = 0$

In the beginning, no paths are known.

初始的时，没有已找到的最短路径。

The trivial path $s \rightsquigarrow s$ is known and has length 0.

空的路径$s \rightsquigarrow s$已找到了。

Algorithm Invariants

Only INITIALIZE-SINGLE-SOURCE and RELAX change $v.d$.

- Triangle inequality: For any edge $(u,v) \in E$, we have $\delta(s,v) \leq \delta(s,u) + w(u,v)$.
- Upper-bound property: We always have $v.d \geq \delta(s,v)$, and when $v.d$ becomes equal to $\delta(s,v)$, it never changes afterward.
- No-path property: If there is no path from s to v, we always have $v.d = \delta(s,v) = \infty$.

最短路径算法的性质

- 除了INITIALIZE-SINGLE-SOURCE和RELAX以外没有改变$v.d$的操作。
- 三角等式性质：对于任何边$(u,v) \in E$，我们有$\delta(s,v) \leq \delta(s,u) + w(u,v)$。
- 上界性质：对于所有的结点v，我们总是有$v.d \geq \delta(s,v)$。一旦$v.d = \delta(s,v)$，其值将不再发生变化。
- 非路径性质：如果从s到v之间不存在路径，则总是有$v.d = \delta(s,v) = \infty$。

Algorithm Invariants

- Convergence property: If $s \rightsquigarrow u \rightarrow v$ is a shortest path for some $u,v \in V$, and if $u.d = \delta(s,u)$ at any time before relaxing edge (u,v), then $v.d = \delta(s,v)$ at all times afterward.
- Path-relaxation property: If $p = \langle v_0, v_1, \ldots, v_k \rangle$ is a shortest path from $s = v_0$ to v_k, and we have relaxed the edges of p in order, then $v_k.d = \delta(s,v_k)$.
- Predecessor-subgraph property: Once $v.d = \delta(s,v)$ for all vertices $v \in V$, the predecessor subgraph is a shortest-path tree rooted at s.

最短路径算法的性质

- 收敛性质：如果$s \rightsquigarrow u \rightarrow v$是一条最短路径，并且在对边$(u,v)$进行松弛前的任意时间有$u.d = \delta(s,u)$，则松弛以后$v.d = \delta(s,v)$。
- 路径松弛性质：如果$p = \langle v_0, v_1, \ldots, v_k \rangle$是从$s = v_0$到$v_k$的最短路径，并且对$p$中的边按顺序进行松弛，则以后$v_k.d = \delta(s,v_k)$。
- 前驱子图性质：一旦对于所有的$v \in V$有$v.d = \delta(s,v)$，则前驱子图是一颗根为s的最短路径树。

Two Main Algorithms

- **Bellman–Ford:**
  relax every edge $|V|-1$ times
  - algorithm for dags:
    After topological sorting,
    1 relaxation / edge is enough.
- **Dijkstra:**
  Relax in correct sequence
  ➥ 1 relaxation / edge is enough.
  Combines **breadth-first search** (first handle vertex closest to source) and **Prim's algorithm** (priority queue to find that vertex)

allow negative edge weights

faster for sparse graphs

两个主要算法

- **Bellman–Ford：**
  对所有的边进行松弛$|V|-1$次。
  - 有向无环图算法：拓扑排序以后，
    对所有的边进行松弛1次就够了。
- **Dijkstra：**
  按次顺序进行松弛
  ➥ 对所有的边进行松弛1次就够了。
  结合**广度优先搜索**（先检查离源结点最近的结点）与**Prim算法**（使用优先队列找到这个节点）

可以处理负值权重的边

更快处理稀疏图

Bellman–Ford Algorithm

Bellman–Ford算法

Idea: A shortest path has no cycles.
So it has at most $|V|-1$ edges.
➥ relax all edges $|V|-1$ times and
use the **path-relaxation property.**

注意： 最短的路径没有环路。
所以最多有$|V|-1$条边。
➥ 松弛所有的边$|V|-1$次,
证明使用路径松弛性质。

initialize

relax edges $|V|-1$ times

decide if there are negative cycles

```
BELLMAN-FORD(G,w,s)
{INITIALIZE-SINGLE-SOURCE(G,s) }
repeat |V|–1 times
        for each edge (u,v) ∈ G.E
                RELAX(u,v,w)
for each edge (u,v) ∈ G.E
        if v.d > u.d + w(u,v)
                return "There is a negative-weight cycle."
return "The calculated shortest paths are correct."
```

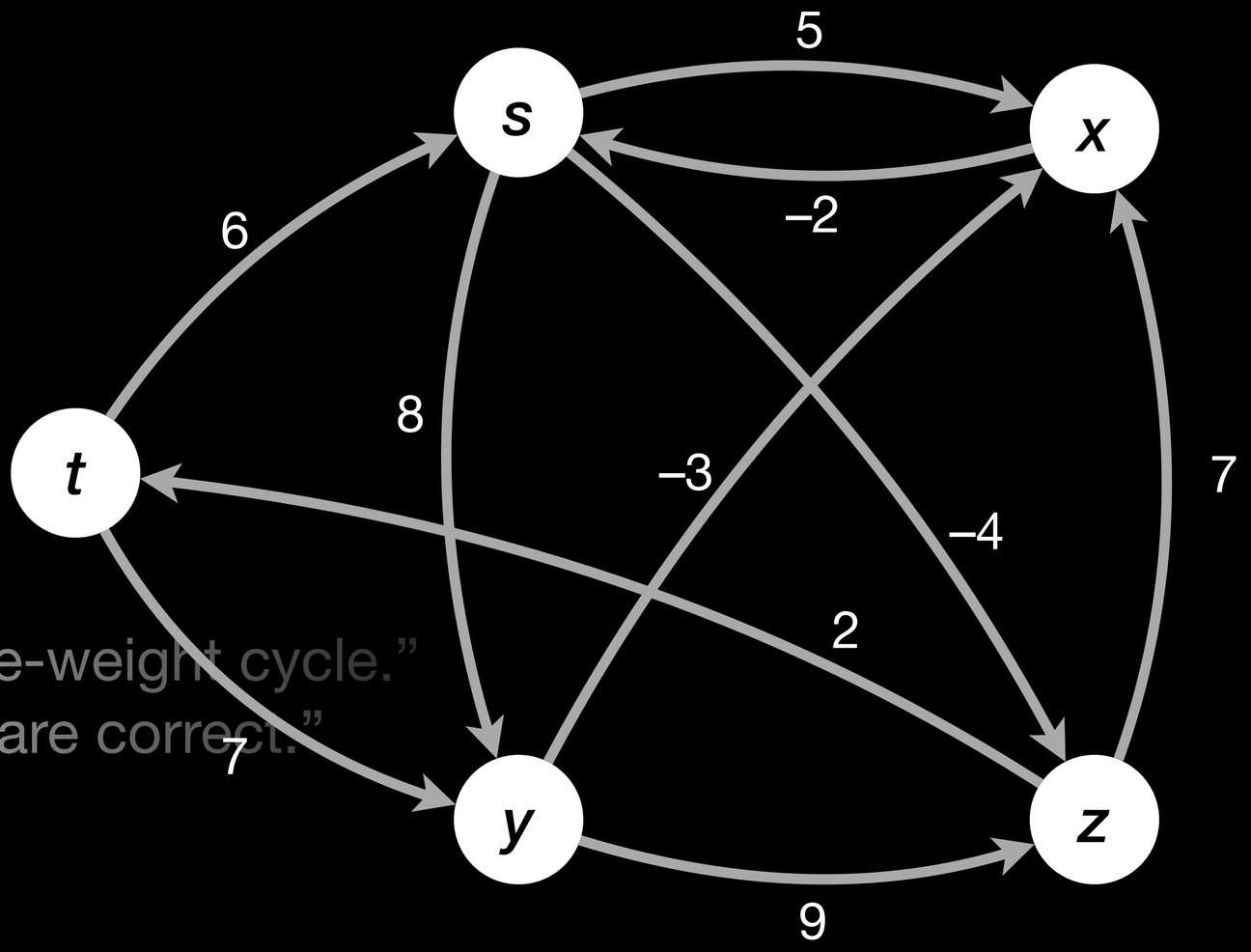
初始化

松弛每条边$|V|-1$次

确定是否有负值的环路

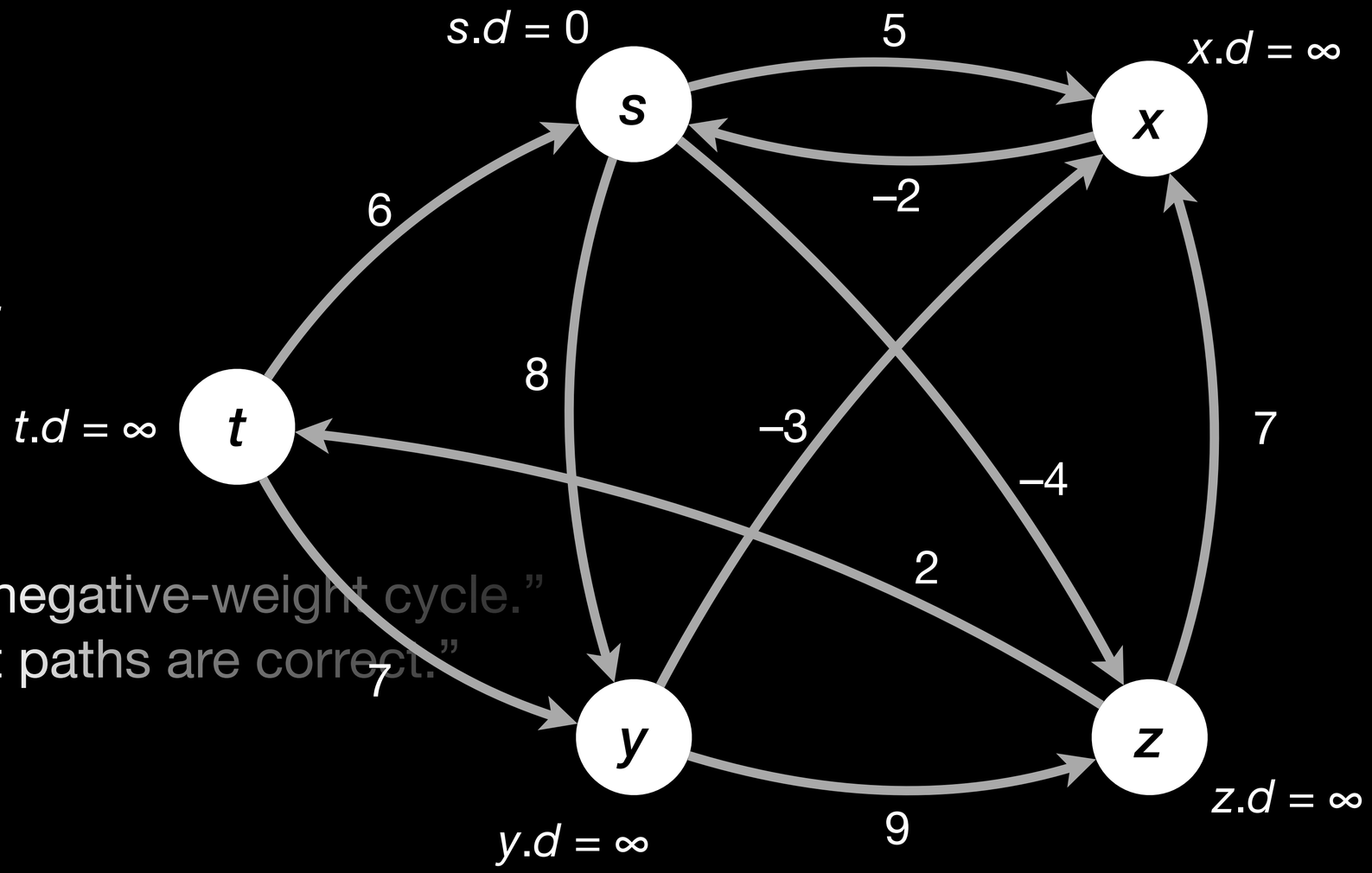
Bellman–Ford: Example

```
BELLMAN-FORD(G,w,s)
INITIALIZE-SINGLE-SOURCE(G,s)
repeat |V|–1 times
        for each edge (u,v) ∈ G.E
                RELAX(u,v,w)
for each edge (u,v) ∈ G.E
        if v.d > u.d + w(u,v)
                return "There is a negative-weight cycle."
return "The calculated shortest paths are correct."
```



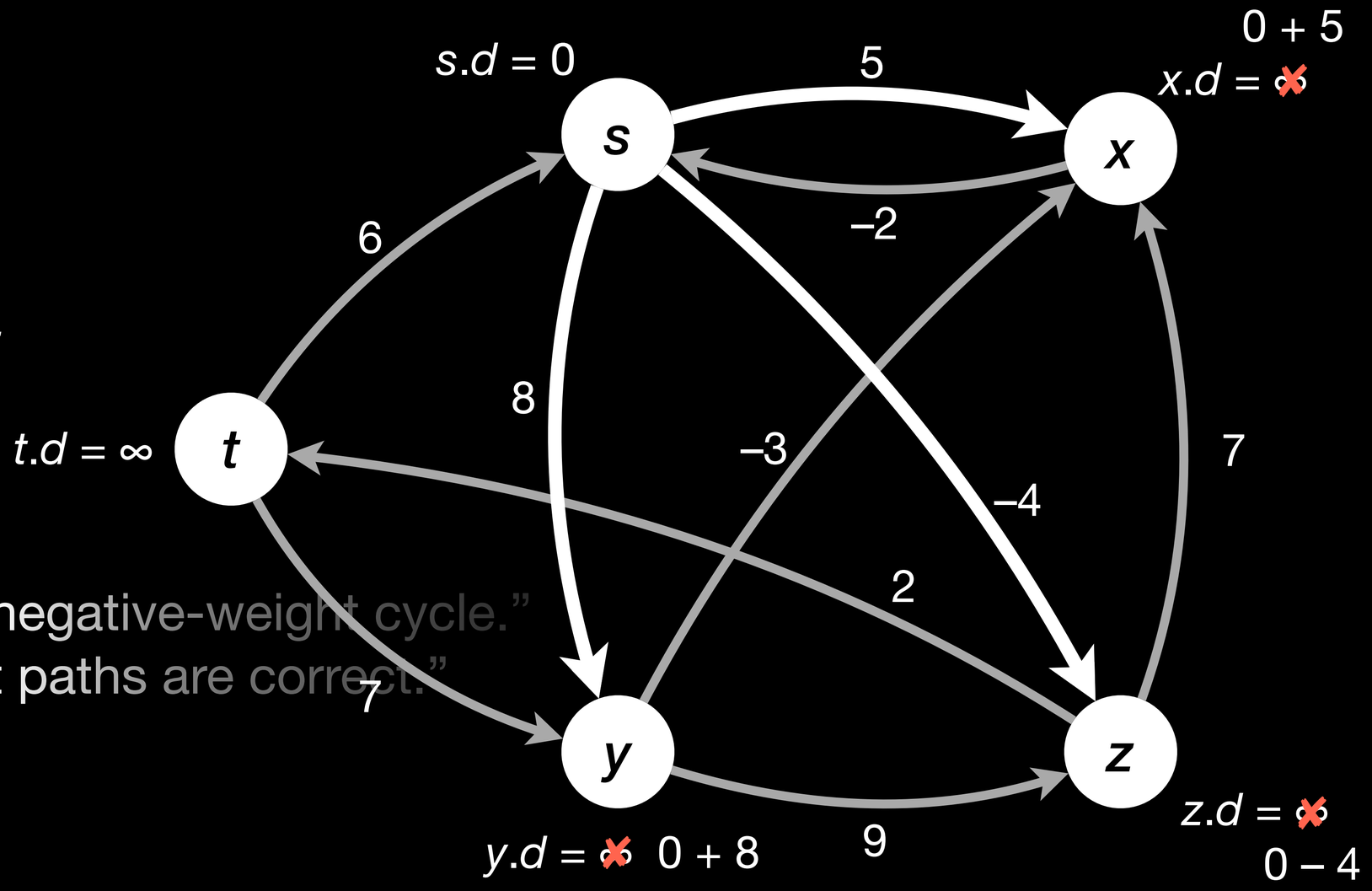
Bellman–Ford: Example

```
BELLMAN-FORD(G,w,s)
INITIALIZE-SINGLE-SOURCE(G,s)
repeat |V|–1 times
        for each edge (u,v) ∈ G.E
                RELAX(u,v,w)
for each edge (u,v) ∈ G.E
        if v.d > u.d + w(u,v)
                return "There is a negative-weight cycle."
return "The calculated shortest paths are correct."
```



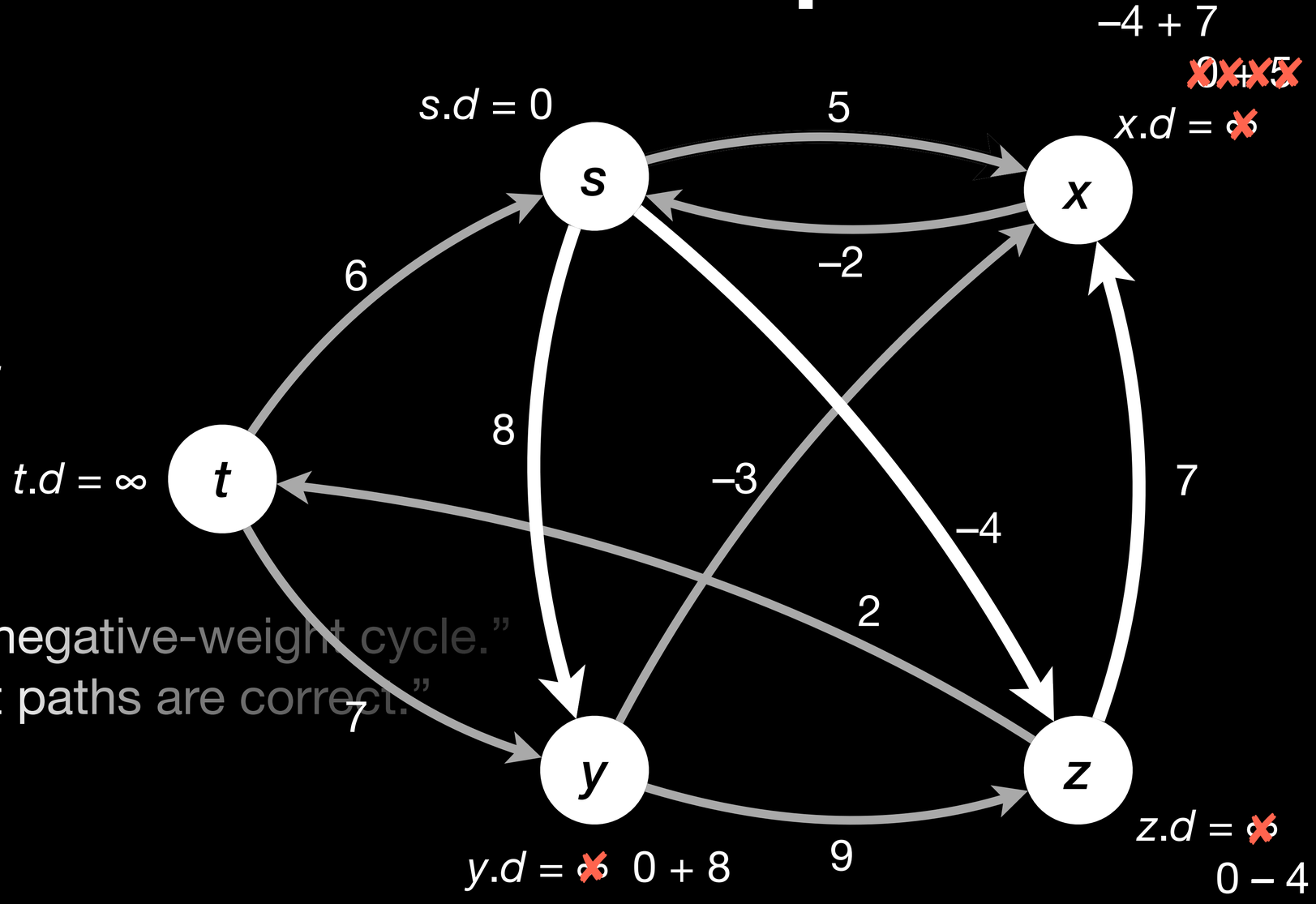
Bellman–Ford: Example

```
BELLMAN-FORD(G,w,s)
INITIALIZE-SINGLE-SOURCE(G,s)
repeat |V|–1 times
        for each edge (u,v) ∈ G.E
                RELAX(u,v,w)
for each edge (u,v) ∈ G.E
        if v.d > u.d + w(u,v)
                return "There is a negative-weight cycle."
return "The calculated shortest paths are correct."
```



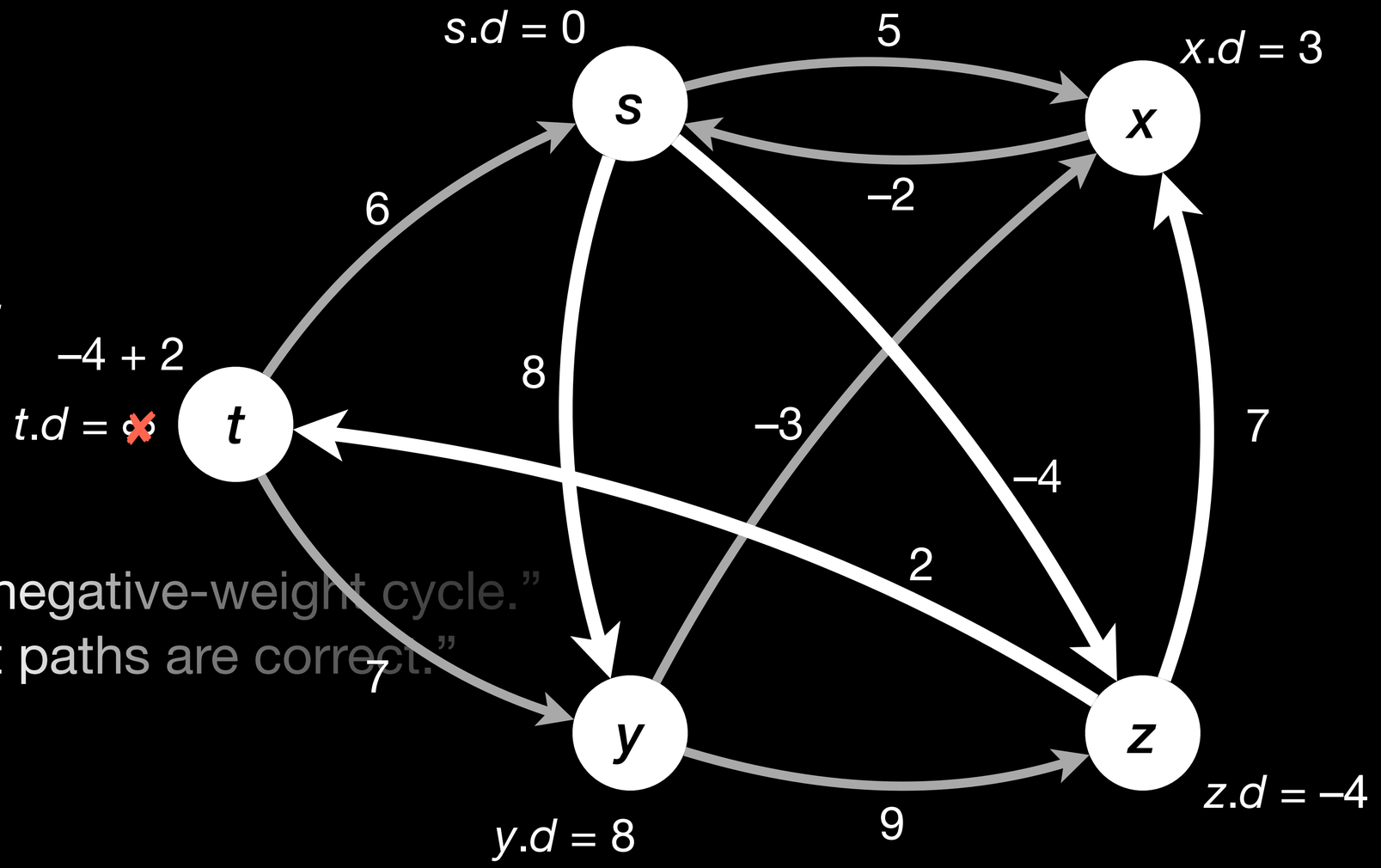
Bellman–Ford: Example

```
BELLMAN-FORD(G,w,s)
INITIALIZE-SINGLE-SOURCE(G,s)
repeat |V|–1 times
        for each edge (u,v) ∈ G.E
                RELAX(u,v,w)
for each edge (u,v) ∈ G.E
        if v.d > u.d + w(u,v)
                return "There is a negative-weight cycle."
return "The calculated shortest paths are correct."
```



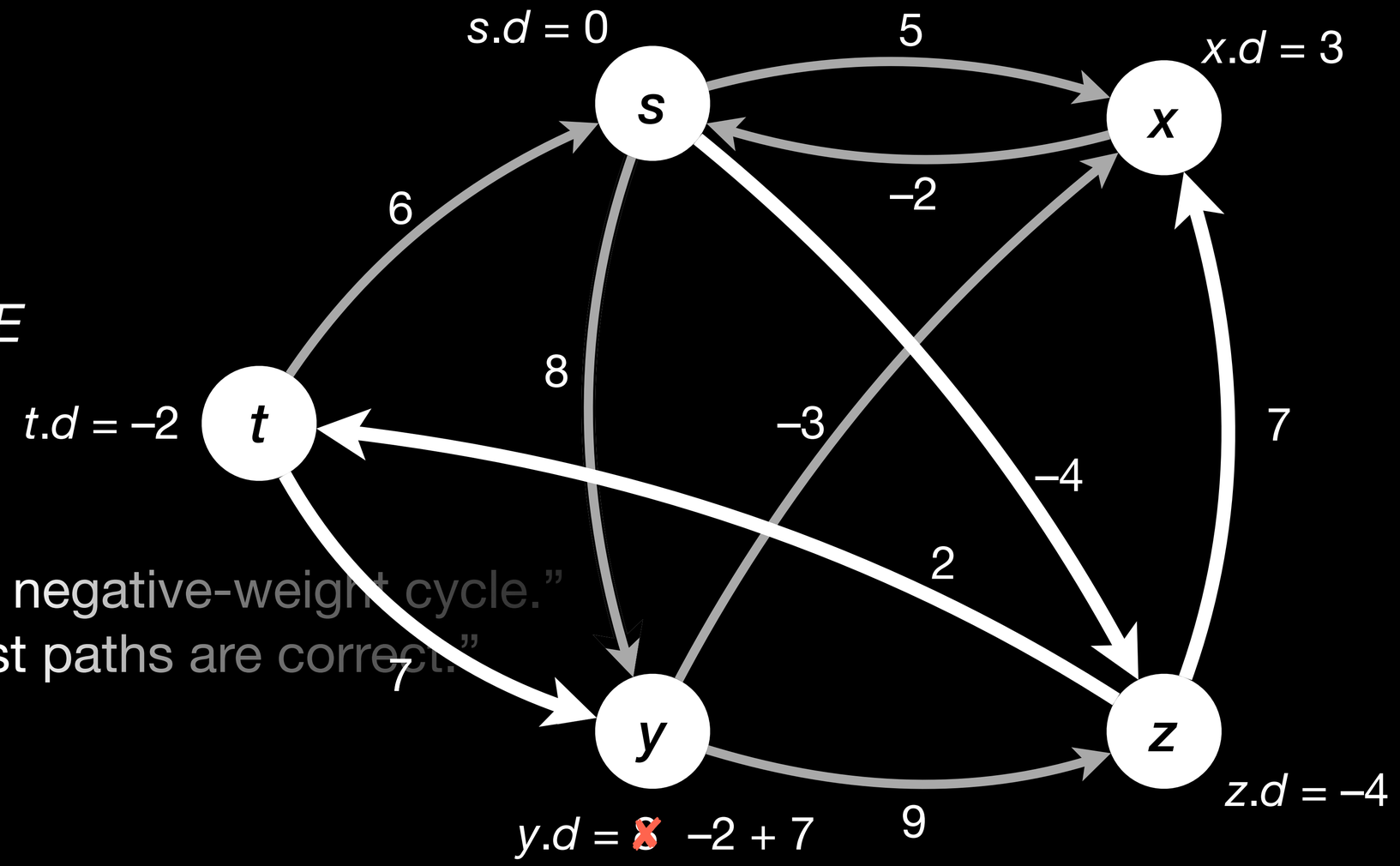
Bellman–Ford: Example

BELLMAN-FORD(G,w,s)
INITIALIZE-SINGLE-SOURCE(G,s)
repeat $|V|-1$ times
    for each edge $(u,v) \in G.E$
        RELAX(u,v,w)
for each edge $(u,v) \in G.E$
    if $v.d > u.d + w(u,v)$
        return "There is a negative-weight cycle."
return "The calculated shortest paths are correct."



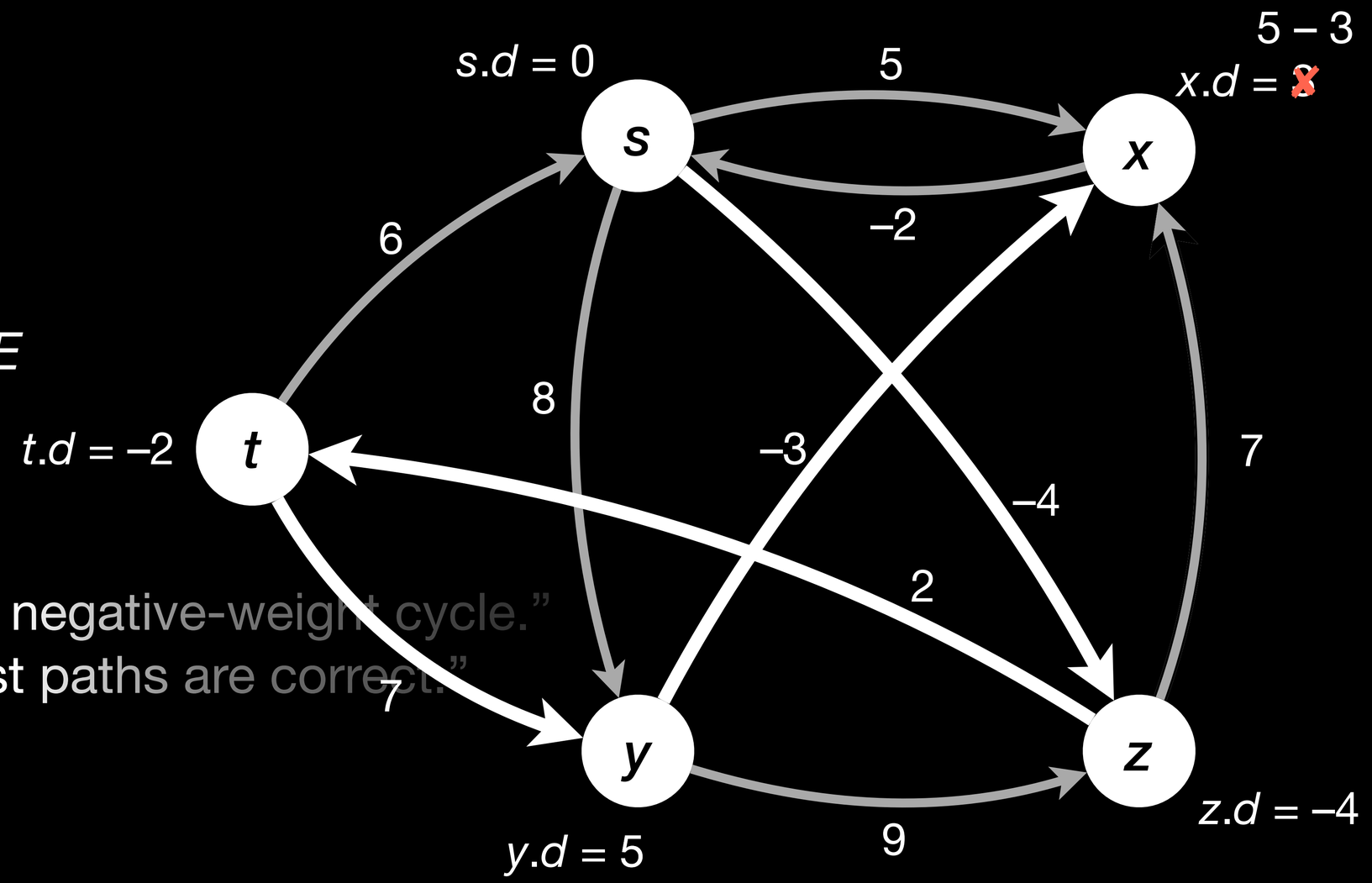
Bellman–Ford: Example

```
BELLMAN-FORD(G,w,s)
INITIALIZE-SINGLE-SOURCE(G,s)
repeat |V|–1 times
        for each edge (u,v) ∈ G.E
                RELAX(u,v,w)
for each edge (u,v) ∈ G.E
        if v.d > u.d + w(u,v)
                return "There is a negative-weight cycle."
return "The calculated shortest paths are correct."
```



Bellman–Ford: Example

```
BELLMAN-FORD(G,w,s)
INITIALIZE-SINGLE-SOURCE(G,s)
repeat |V|–1 times
        for each edge (u,v) ∈ G.E
                RELAX(u,v,w)
for each edge (u,v) ∈ G.E
        if v.d > u.d + w(u,v)
                return "There is a negative-weight cycle."
return "The calculated shortest paths are correct."
```



Bellman–Ford: Correctness

- Loop Invariant: After the *i*th iteration, the algorithm has found all shortest path with ≤*i* edges (and possibly some shortest paths with more edges).
- Lemma 24.2: If *G* does not contain negative-weight cycles, then BELLMAN-FORD ends with $v.d = \delta(s,v)$ for all vertices *v* reachable from *s*.

  Proof: If there is a shortest path $s \rightsquigarrow v$, then there is a shortest path with ≤ $|V|-1$ edges. Therefore, after $|V|-1$ iterations, all shortest path distances have been found by the path-relaxation property.

Bellman–Ford：正确性

- 循环不变式：第i次迭代以后，算法找到了所有的有≤i边的最短路径（可能也找到了几个有更多边的最短路径）。
- 引理24.2：如果G没有负值权重的环路，那么BELLMAN-FORD以后对于所有从*s*可以到达的结点，我们有$v.d = \delta(s,v)$。
  - 证明：如果存在最短路径$s \rightsquigarrow v$，那么存在有一条≤ $|V|-1$边的最短路径。所以$|V|-1$迭代以后，由于路径松弛性质所有的最短路径权重都找到了。

Bellman–Ford: Running Time

- Initialization takes time $O(|V|)$.
- The main loop visits every edge $|V|-1$ times, so it visits $O(|V| \cdot |E|)$ edges. Every visit does constant-time work.
- The check loop visits every edge once, so it takes time $O(|E|)$.
- The total running time is therefore in $O(|V| \cdot |E|)$.

Bellman–Ford：运行时间

- 初始化时间为$O(|V|)$。
- 主要循环检查所有的边$|V|-1$次，所以检查$O(|V| \cdot |E|)$条边。单独的检查为$O(1)$。
- 确认是否有负值的环路把所有的边检查1次，运行时间为$O(|E|)$。
- 总体的运行时间为$O(|V| \cdot |E|)$。

Algorithm for Dags

Idea:

Relax edges in the correct sequence,
so that every edge needs to be relaxed
only once.

Q: What is the correct sequence?

A: If $u \to v \to w$, relax $u \to v$ before $v \to w$.
Any sequence compatible with
the order induced by the edges.
Topological Sort.

有向无环图的算法

注意：

安排顺序进行松弛，
以便所有的边仅进行1次。

问：什么顺序正确的?

答：如果 $u \to v \to w$， 先松弛 $u \to v$，
后 $v \to w$。
与边诱导的顺序兼容的任何序列。
拓扑排序。

Algorithm for Dags　有向无环图的算法

Idea

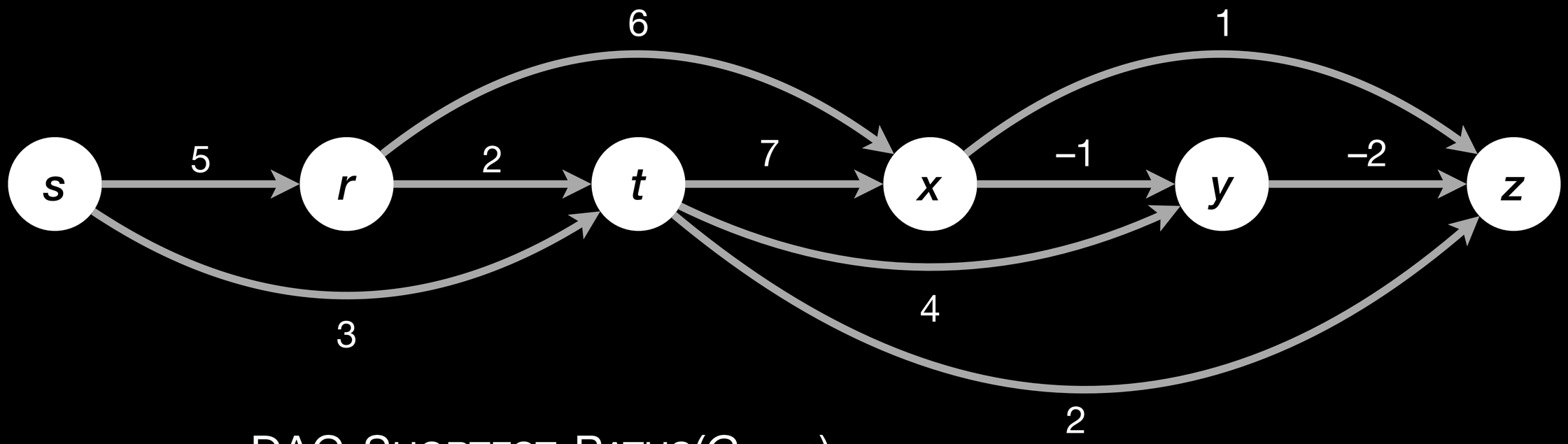
Relax edges in the correct sequence,
so that every edge needs to be relaxed
only once.

注意：

安排顺序进行松弛，
以便所有的边仅进行1次。

DAG-SHORTEST-PATHS(G,w,s)
Topologically sort the vertices of G
INITIALIZE-SINGLE-SOURCE(G,s)
for each vertex $u \in G.V$ (in topologically sorted order)
　　for each vertex v adjacent to u
　　RELAX(u,v,w)

Algorithm for Dags: Example



DAG-SHORTEST-PATHS(*G*,*w*,*s*)
Topologically sort the vertices of G
INITIALIZE-SINGLE-SOURCE(*G*,*s*)
for each vertex $u \in G.V$ (in topologically sorted order)
    for each vertex *v* adjacent to *u*
        RELAX(*u*,*v*,*w*)

Algorithm for Dags: Partial Correctness

- Theorem 24.5: If G does not contain any cycles,
  then DAG-SHORTEST-PATHS ends with $v.d = \delta(s,v)$
  for all vertices v that are reachable from s.

  Proof: If there is a shortest path s $\rightsquigarrow$ v, then it follows the topological order. Therefore, edges in this path are relaxed
  from the first to the last. By the path-relaxation property,
  we get the result.

Algorithm for Dags: Running Time

- Sorting the vertices in topological order takes time $O(|V| + |E|)$.
- Initialization takes time in $O(|V|)$.
- In the main loop, every vertex is visited once,
  and every edge is relaxed exactly once,
  which takes time in $O(|V| + |E|)$.
- The total running time is in $O(|V| + |E|)$.

有向无环图：运行时间

- 拓扑排序的运行时间为 $O(|V| + |E|)$。
- 初始化运行时间为 $O(|V|)$。
- 在珠环路中，所有的结点被检查一次，
  并且所有的边被松弛一次。
  这个使用运行时间为 $O(|V| + |E|)$。
- 因此，总体运行时间为 $O(|V| + |E|)$。

Dijkstra's Algorithm

Idea: Relax edges in the correct sequence,
so that every edge needs to be relaxed only once.

Q: What is the correct sequence?
A: Edges whose start vertex is close to the source should be relaxed first.
(similar to breadth-first search)

Q: What is "close to the source"?
A: Use path lengths to determine closeness.
Implement with priority queue.

Q: How should I pronounce "Dijkstra"?
A: pronounce "ij" as "y". In *cursive or script* writing, "ij" looks almost as "y": $i + j = ij \approx y$

Dijkstra的算法

注意： 安排顺序进行松弛，
以便所有的边仅进行1次。

问：什么顺序正确的?
答：先松弛起点离源结点附近的边
（类似广度优先搜索）。

问：什么是"离源结点附近的"?
答：使用路径的权重计算附近度。
用优先队列实现。

问：怎么样发音"Dijkstra"?
答："ij"的发音为英语的"y"一样。

Dijkstra's Algorithm

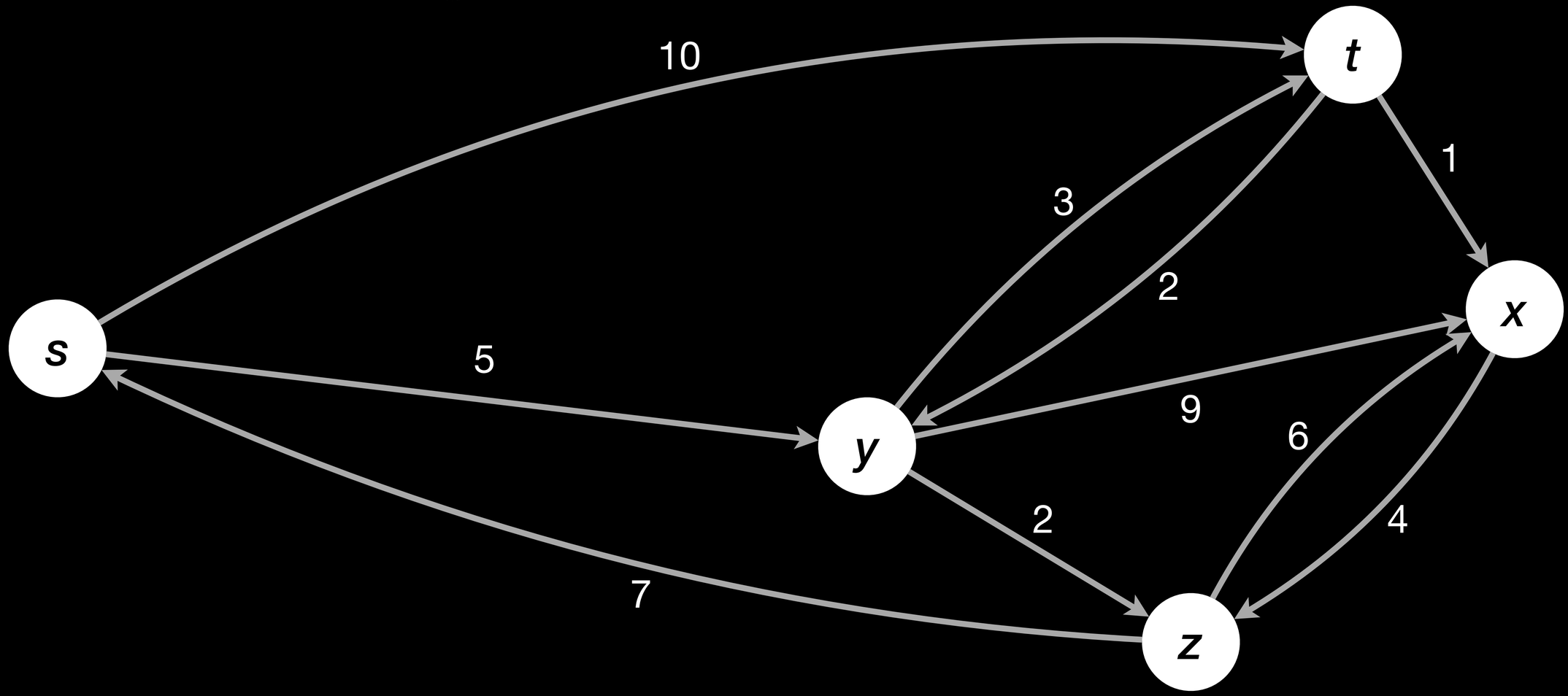
Dijkstra的算法

Idea: Relax edges in the correct sequence,
so that every edge needs to be relaxed only once.

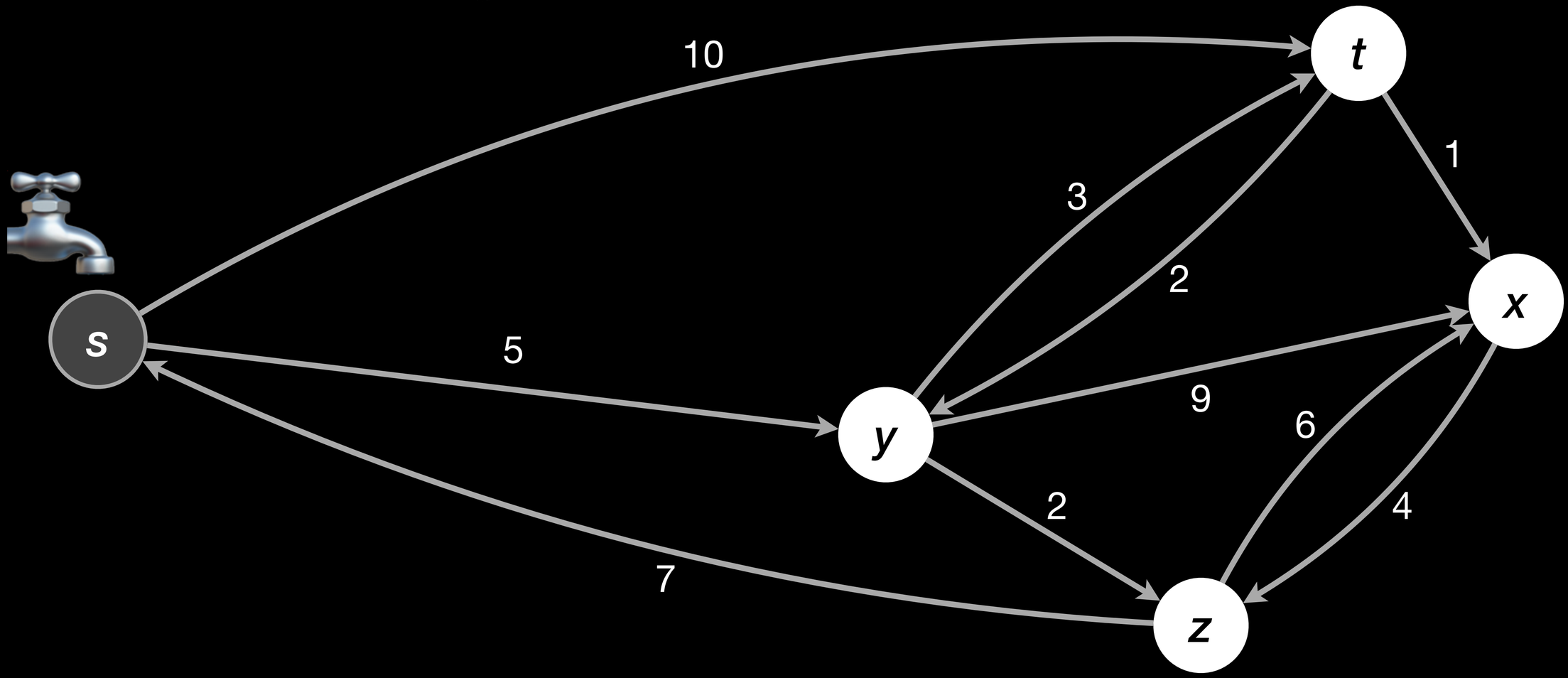
注意： 安排顺序进行松弛，
以便所有的边仅进行1次。

```
DIJKSTRA(G,w,s)
INITIALIZE-SINGLE-SOURCE(G,s)
S = ∅
Q = G.V (priority queue with key u.d)
while Q ≠ ∅
        u = EXTRACT-MIN(Q)
        S = S ∪ {u}
        for each vertex v adjacent to u
                RELAX(u,v,w)
```

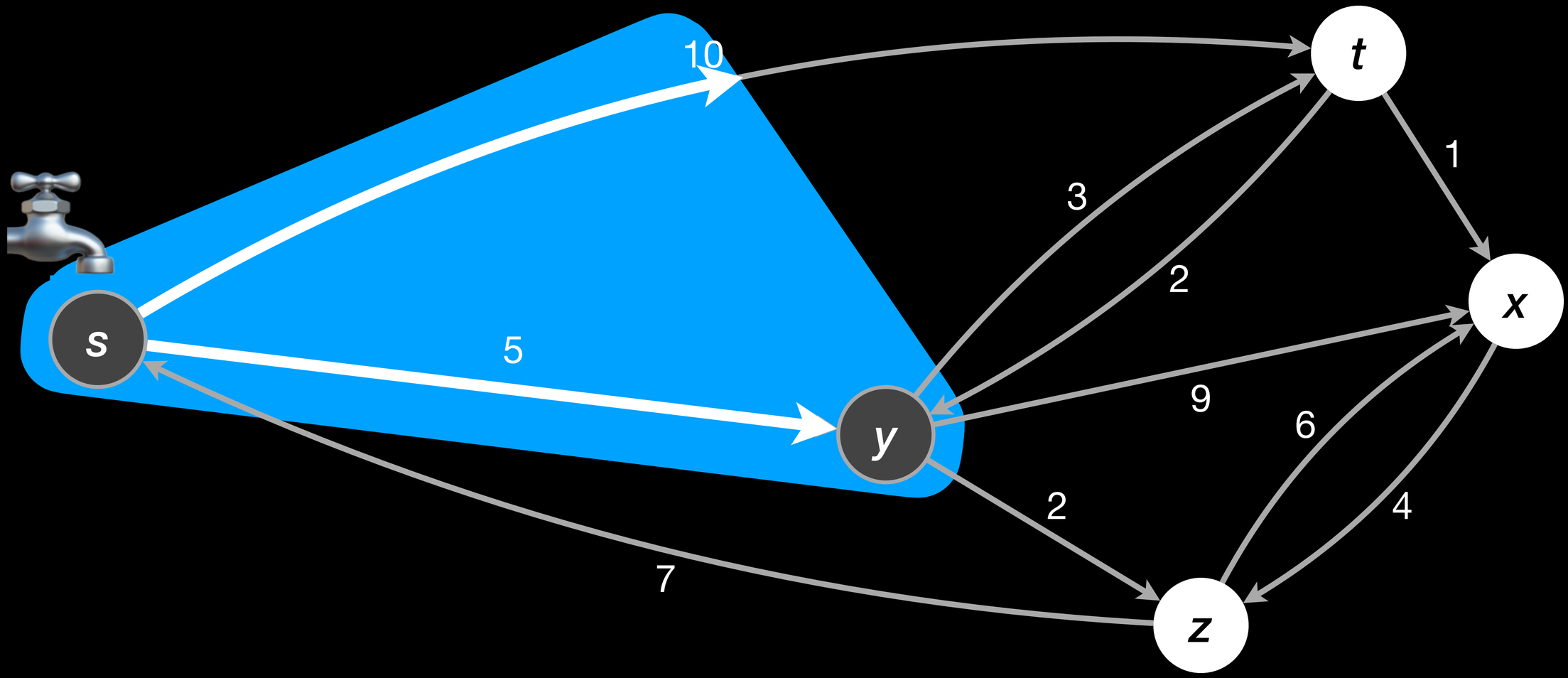
Dijkstra: Example



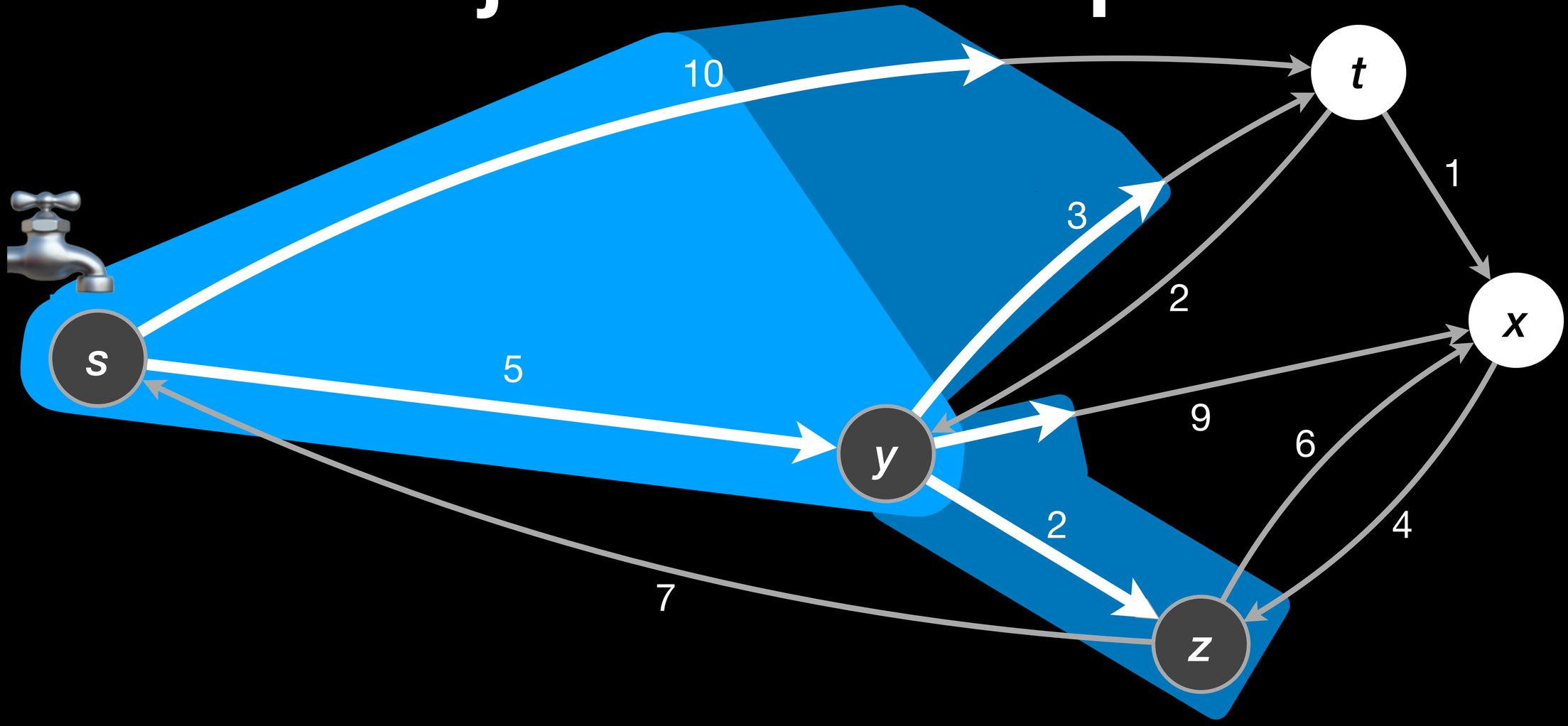
Dijkstra: Example



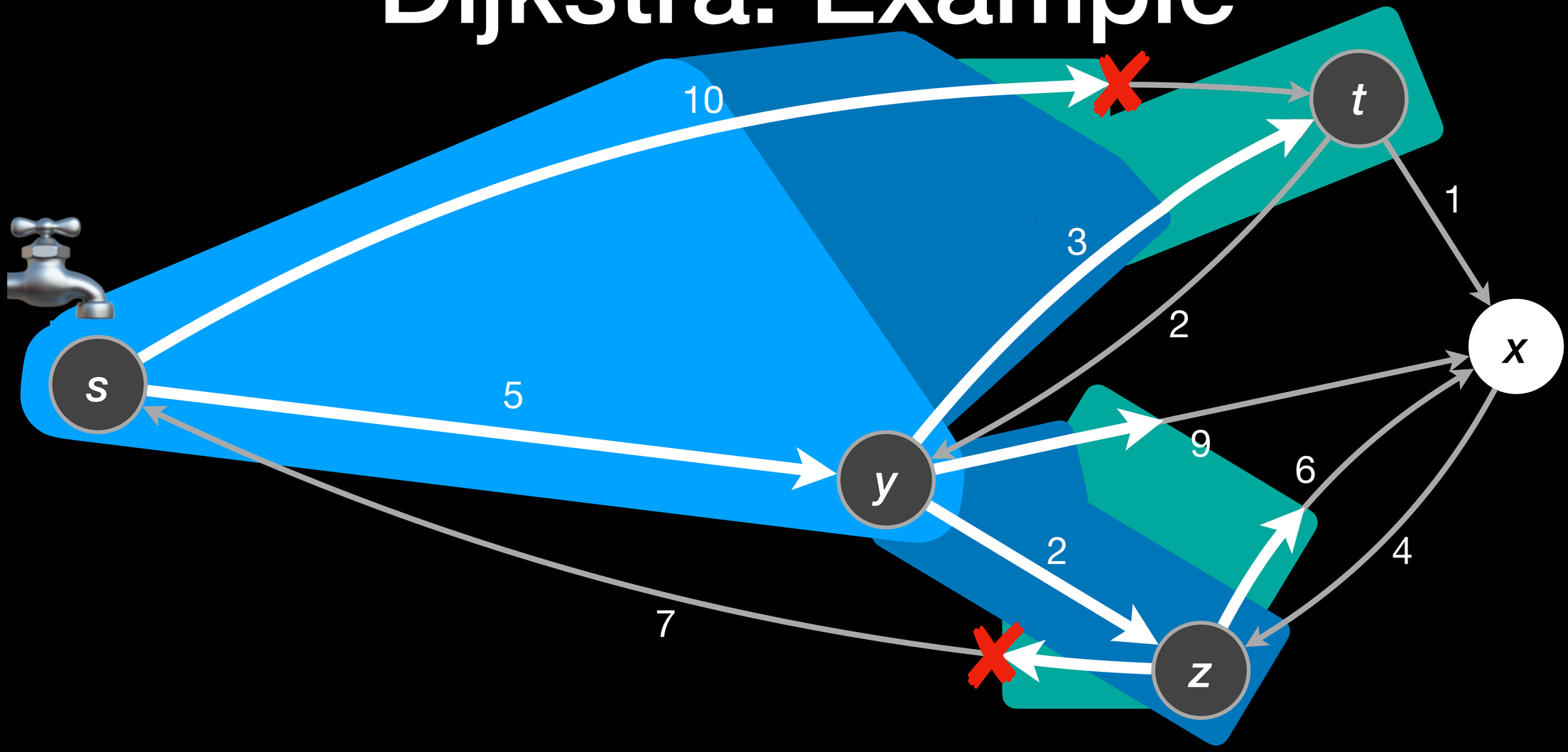
Dijkstra: Example



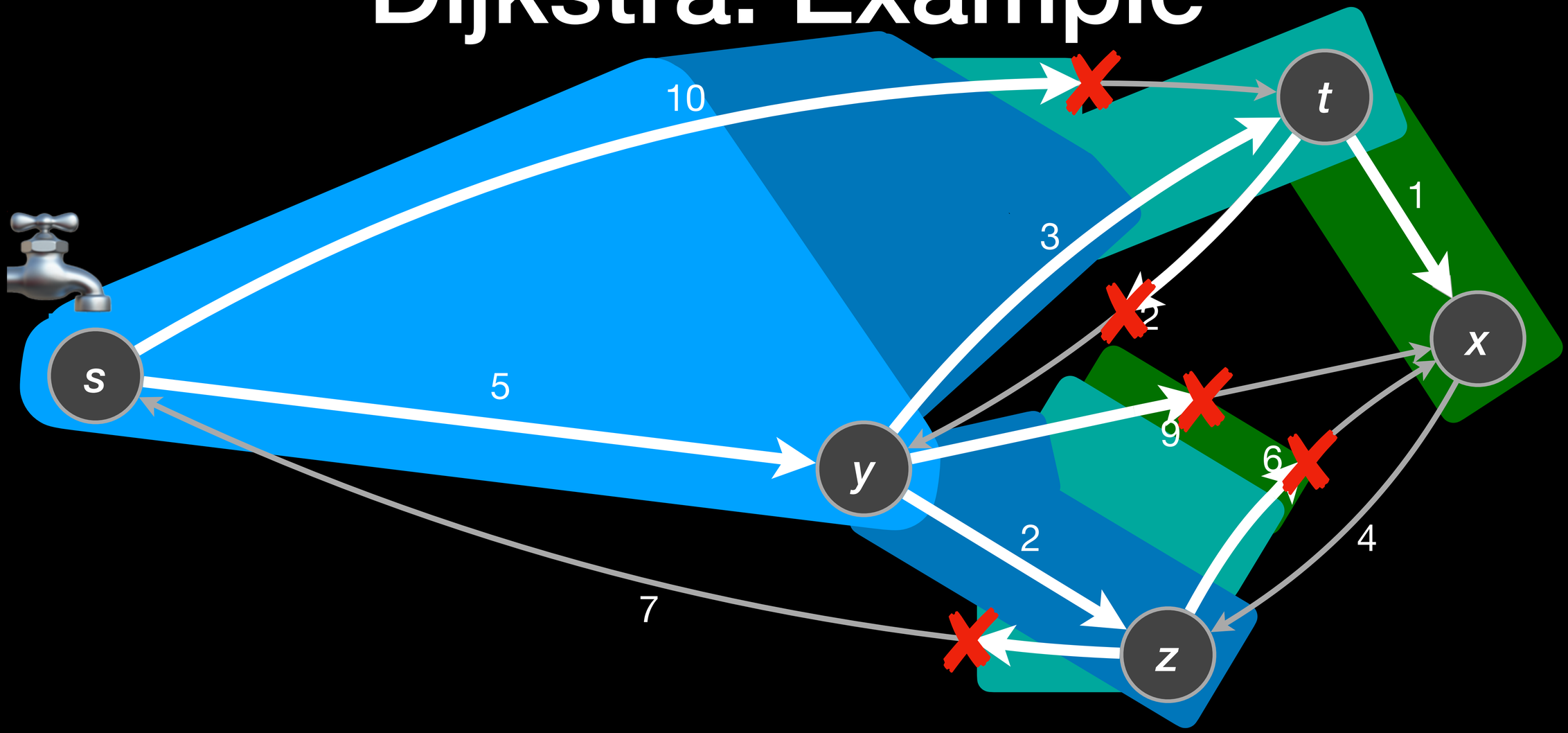
Dijkstra: Example



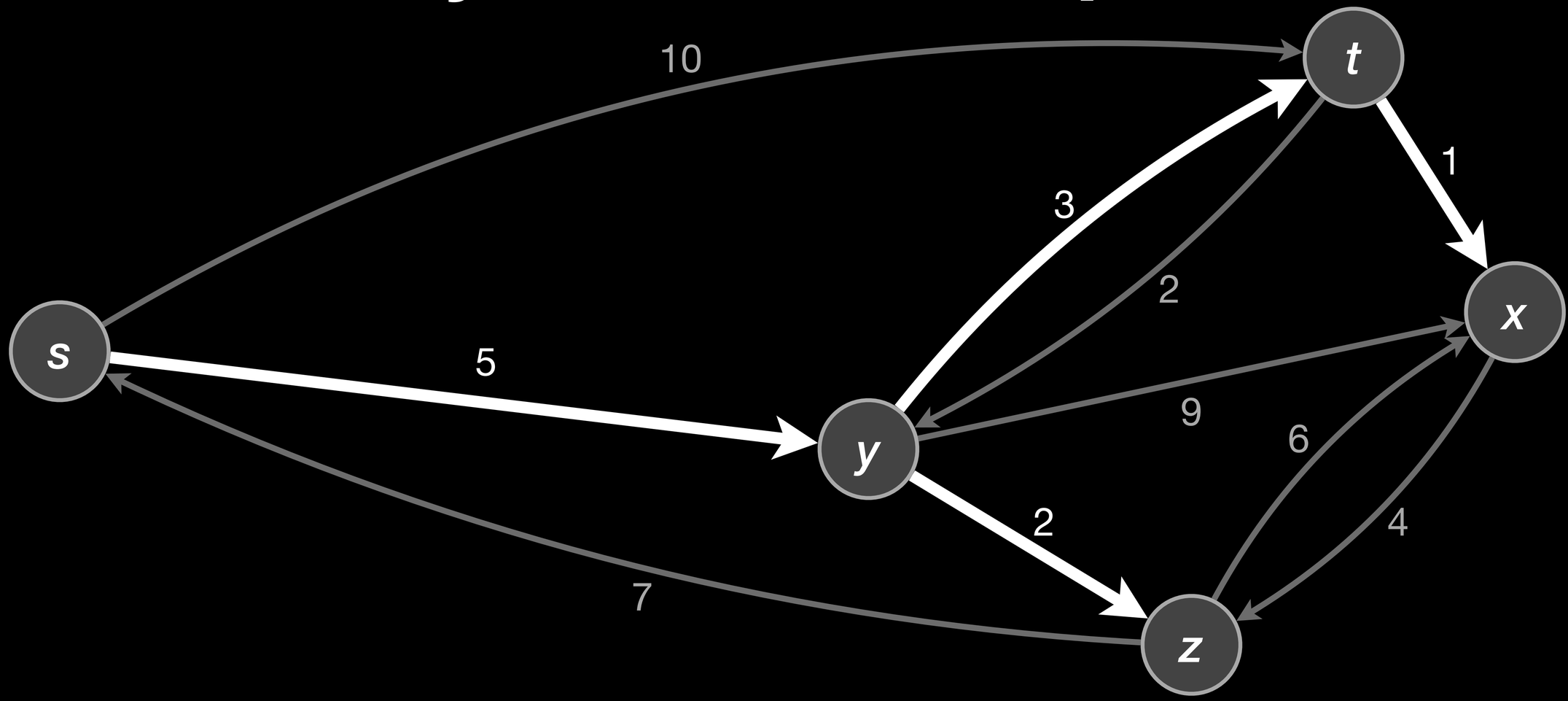
Dijkstra: Example



Dijkstra: Example



Dijkstra: Example



Dijkstra: Correctness

- Loop Invariant: For all vertices $v \in S = V \backslash Q$, their shortest paths have been found (i.e. $v.d = \delta(s,v)$).
  Also, all shortest paths with length $\leq v.d$ have been found.

- Lemma 24.2: If G does not contain negative-weight edges, then DIJKSTRA ends with $v.d = \delta(s,v)$ for all vertices v that are reachable from s.

  Proof idea: use induction over the loop.

Dijkstra：正确性

- 循环不变式：为所有的结点$v \in S = V \backslash Q$，到最短路径已找到了（$v.d = \delta(s,v)$）。
  此外，所有的有权重$\leq v.d$的最短路径已找到了。

- 引理24.2：如果G里没有负值权重的边，那么DIJKSTRA结束以后为所有从源结点s可以到达的边有$v.d = \delta(s,v)$。

Dijkstra: Running Time

- The priority queue operations are the slowest part of the algorithm.
  - Initialization takes time in $O(|V|)$.
    Also, EXTRACT-MIN is called $|V|$ times.
  - Every edge is relaxed exactly once,
    and every relaxation calls DECREASE-KEY at most once.
- Priority queue implementations:
  - super simple: EXTRACT-MIN does a linear search through the vertices.
    Total running time in $O(|V|^2 + |E|) = O(|V|^2)$.
  - with a heap: Calling EXTRACT-MIN/DECREASE-KEY requires time $O(\log |V|)$.
    Total running time in $O((|V| + |E|)\log |V|) = O(|E| \log |V|)$.

Dijkstra：运行时间

- 优先队的操作是算法最慢的部分。
  - 预置位为$O(|V|)$。
    此外EXTRACT-MIN$|V|$次执行。
  - 所有的边进行松弛1次，
    所有的松弛操作调用DECREASE-KEY
    至多1次。
- 优先队的实现：
  - 简单的：EXTRACT-MIN使用线性搜索。
    总体运行时间为$O(|V|^2 + |E|) = O(|V|^2)$。
  - 用最小堆：
    调用EXTRACT-MIN/DECREASE-KEY时间为$O(\log |V|)$。
    总体运行时间为$O((|V| + |E|)\log |V|) = O(|E| \log |V|)$。

Shortest-Path Invariants: Proofs

最短路径的不变式：证明

Triangle inequality: For any edge $(u,v) \in E$, we have $\delta(s,v) \leq \delta(s,u) + w(u,v)$.

三角等式性质：对于任何边$(u,v) \in E$，我们有$\delta(s,v) \leq \delta(s,u) + w(u,v)$。

Proof: Let p be a shortest path from s to u.
If p' is a shortest path from s to v,
then it cannot be longer than $\langle p,v \rangle$.

Shortest-Path Invariants: Proofs

最短路径的不变式：证明

Upper-bound property: We always have $v.d \geq \delta(s,v)$,
and when $v.d$ becomes equal to $\delta(s,v)$, it never changes afterward.

上界性质：对于所有的结点v，
我们总是有$v.d \geq \delta(s,v)$。
一旦$v.d = \delta(s,v)$，其值将不再发生变化。

Proof: by induction over the number of relaxation steps executed by a shortest-path algorithm.
[Induction step: When relaxing edge (u,v),
the algorithm has found a path p from s to u with length $u.d$;
therefore, the shortest path from s to v cannot be longer than $\langle p,v \rangle$.]

Also, $v.d$ never increases, so it cannot change after it has become equal to $\delta(s,v)$.

Shortest-Path Invariants: Proofs

最短路径的不变式：证明

No-path property: If there is no path from s to v, we always have $v.d = \delta(s,v) = \infty$.

- 非路径性质：如果从s到v不存在路径，则总是$v.d = \delta(s,v) = \infty$。

Proof: $v.d$ is initialized to ∞,
and no sequence of relaxations is going to change that.

Shortest-Path Invariants: Proofs

最短路径的不变式：证明

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

收敛性质：如果$s \rightsquigarrow u \rightarrow v$是一条最短路径，并且在对边$(u,v)$进行松弛前的任意时间有$u.d = \delta(s,u)$，则松弛以后$v.d = \delta(s,v)$。

Proof: When $u.d = \delta(s,u)$,
the algorithm has found a shortest path p from s to u.
It has the same length as the subpath $p' = s \rightsquigarrow u$
of $\langle p',v \rangle = s \rightsquigarrow u \rightarrow v$ mentioned above.
So $\langle p,v \rangle$ is also a shortest path from s to v,
and this is the path that the algorithm finds by relaxing (u,v).

Shortest-Path Invariants: Proofs

最短路径的不变式：证明

Path-relaxation property: If $p = \langle v_0, v_1, \ldots, v_k \rangle$ is a shortest path from $s = v_0$ to v_k, and we have relaxed the edges of p in order, then $v_k.d = \delta(s,v_k)$. This holds regardless of any other relaxation steps that occur, even if there are additional relaxations of the edges in p.

路径松弛性质：如果$p = \langle v_0, v_1, \ldots, v_k \rangle$是从$s = v_0$到$v_k$的最短路径，并且对$p$中的边所按次进行松弛，则以后$v_k.d = \delta(s,v_k)$。

Proof: We show by induction that after i relaxations in the order of p, we have $v_i.d = \delta(s,v_i)$.

[Induction step: The algorithm has found a shortest path p to v_i.
Its length must be the same as $\langle v_0, v_1, \ldots, v_i \rangle$.
The length of $\langle p,v_{i+1} \rangle$ is the same as the length of $\langle v_0, v_1, \ldots, v_{i+1} \rangle$,
which is also a shortest path from s to v_{i+1}.
By relaxing edge (v_i,v_{i+1}), the algorithm finds the path $\langle p,v_{i+1} \rangle$.]

Shortest-Path Invariants: Proofs

最短路径的不变式：证明

Predecessor-subgraph property:
Once $v.d = \delta(s,v)$ for all vertices $v \in V$, the predecessor subgraph is a shortest-path tree rooted at s.

前驱子图性质：一旦对于所有的$v \in V$有$v.d = \delta(s,v)$，则前驱子图一颗根为s的最短路径树。

Proof: We can assume that there are no negative-weight cycles.

1. During the algorithm,
   s and the vertices v with $v.\pi \neq$ NIL form a tree rooted at s.

2. At the end of the algorithm,
   all vertices reachable from s are in the tree,
   and the paths in the tree
   are shortest paths in the original graph.

Outlook: Relation with Linear Programming

"Single-source shortest paths" can be used
to solve a simple kind of linear programming problem:

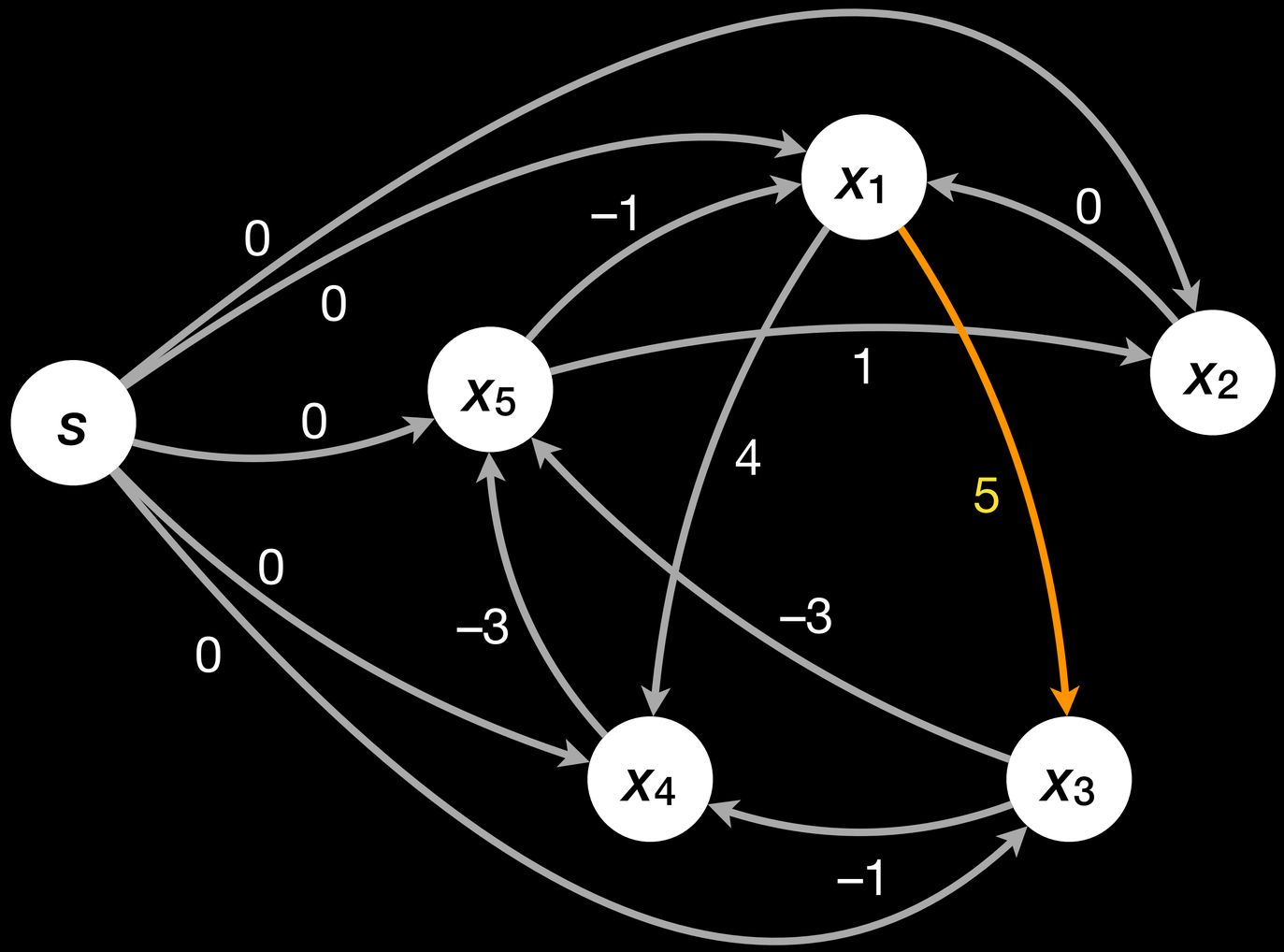
- Linear programming feasibility problem:
  - Given: n variables $x_1, ..., x_n$,
    m constraints $\Sigma_i\, a_{ij} x_i \leq b_j$ (for $j = 1, ..., m$),
    for constants a_{ij} and b_j.
  - To be found: Do values for x_i satisfying the constraints exist?

- If the constraints are difference constraints $x_{i1} - x_{i2} \leq b_j$,
  answer can be found by checking for negative-weight cycles in a graph.

Full Linear Programming later

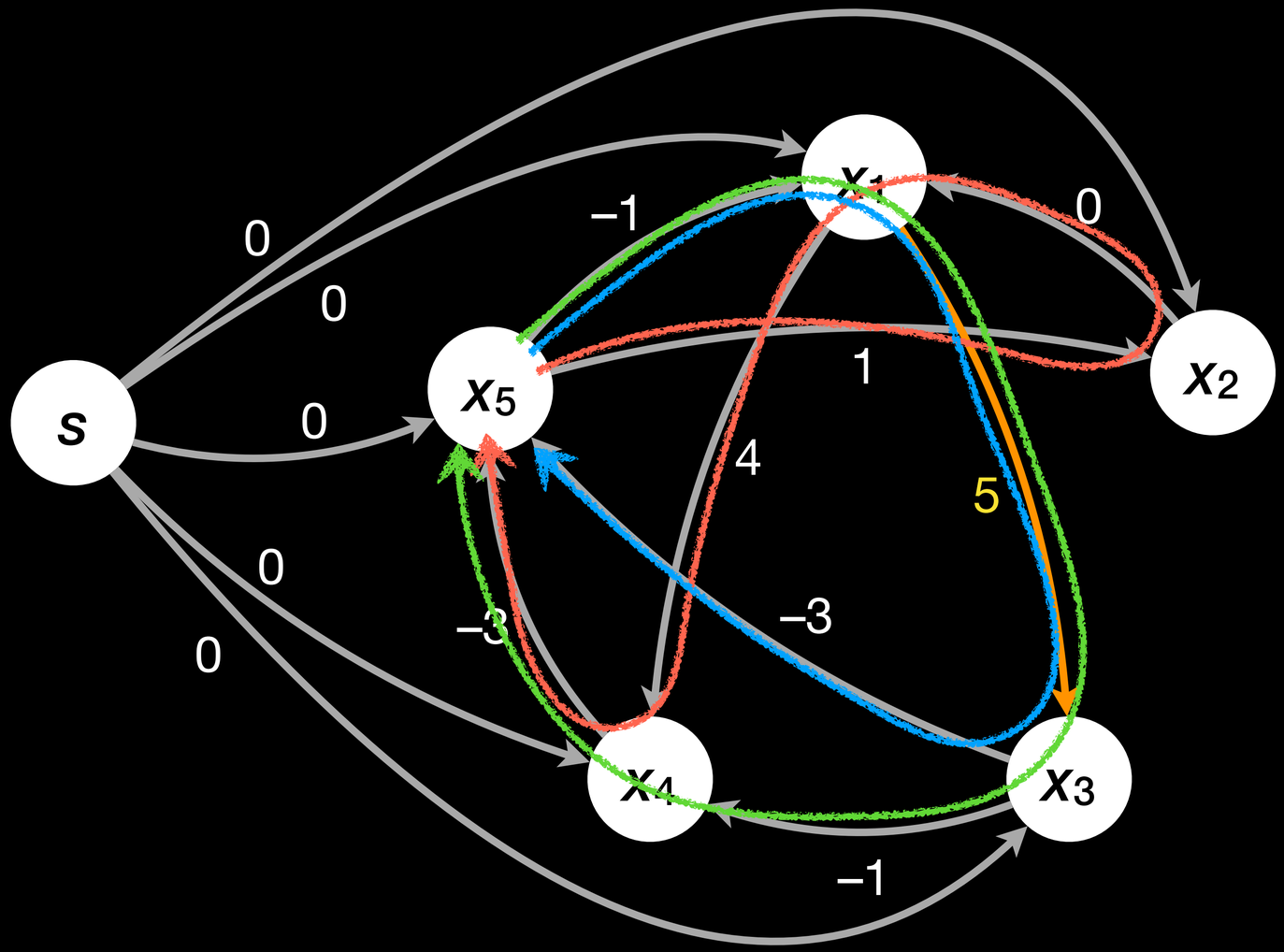
Difference Constraints Example

$x_1 - x_2 \leq 0$
$x_1 - x_5 \leq -1$
$x_2 - x_5 \leq 1$
$x_3 - x_1 \leq 5$
$x_4 - x_1 \leq 4$
$x_4 - x_3 \leq -1$
$x_5 - x_3 \leq -3$
$x_5 - x_4 \leq -3$



Difference Constraints Example

$x_1 - x_2 \leq 0$
$x_1 - x_5 \leq -1$
$x_2 - x_5 \leq 1$
$x_3 - x_1 \leq 5$
$x_4 - x_1 \leq 4$
$x_4 - x_3 \leq -1$
$x_5 - x_3 \leq -3$
$x_5 - x_4 \leq -3$



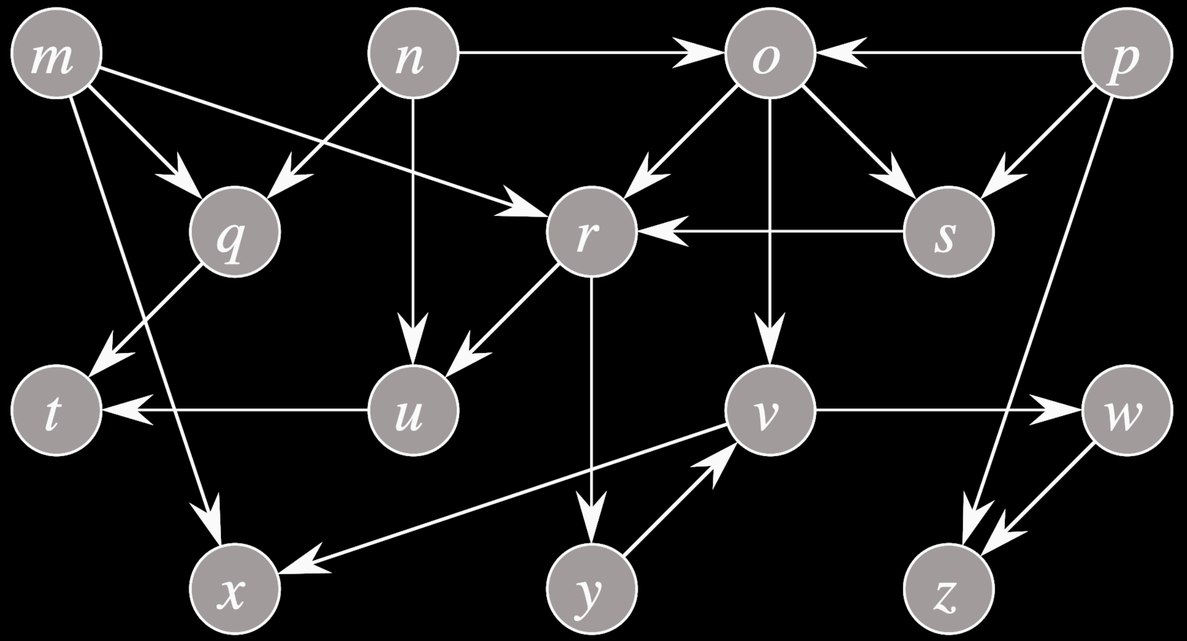
Difference Constraints

- Assume given a set of difference constraints over the variables $x_1,\ldots,x_n$.
  The difference constraint graph has vertices $V = \{s,x_1,\ldots,x_n\}$,
  edges $E = \{(x_{i1},x_{i2}) \mid x_{i1} - x_{i2} \leq b_j$ is a constraint$\} \cup \{(s,x_1),\ldots,(s,x_n)\}$
  with weights $w(x_{i1},x_{i2}) = b_j$ and $w(s,x_i) = 0$.

- Theorem 24.9:
  Assume given a set of difference constraints.
  Let G be the corresponding difference constraint graph.
  If G contains no negative-weight cycles, then
  $x = (\delta(s,x_1),\ldots,\delta(s,x_n))$ is a feasible solution for the system.
  If G contains negative-weight cycles, no feasible solution exists.

Exercise 22.4-1

Show the ordering of vertices produced by TOPOLOGICAL-SORT when it is run on the dag right. Assume that the for loop for lines 5–7 of DFS considers the vertices in alphabetical order, and assume that each adjacency list is ordered alphabetically.



给出算法 TOPOLOGICAL-SORT 运行于图 22-8 上时所生成的结点次序。

假定深度优先搜索算法的第5～7行的 **for** 循环是以字母表顺序依次处理每个结点，并假定每条邻接链表皆以字母表顺序对里面的结点进行了排序。

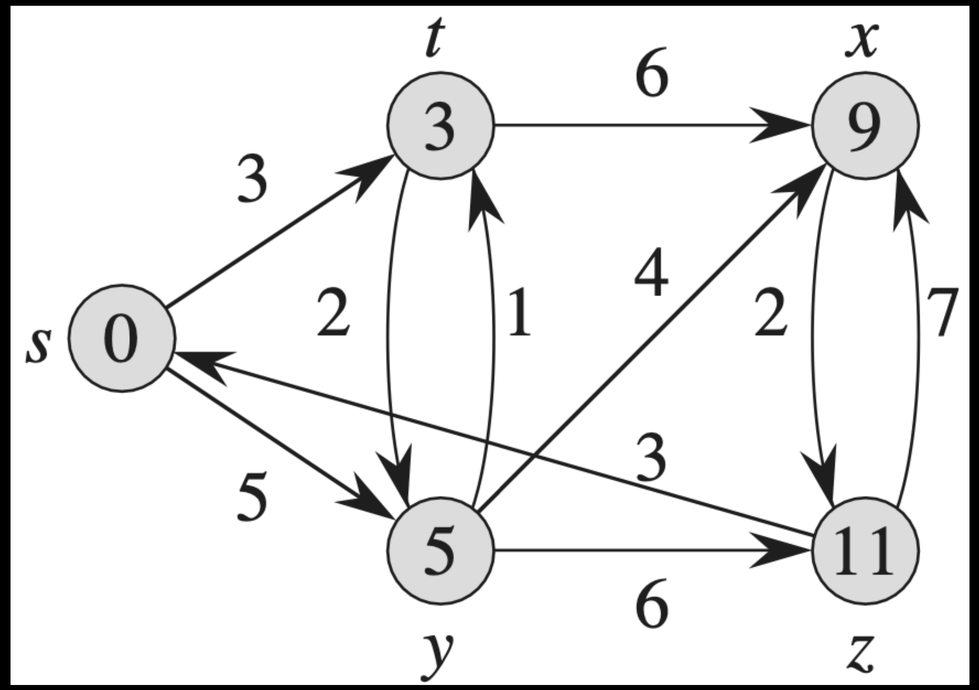
Exercise 23.1-7

Argue that if all edge weights of a graph are positive, then any subset of edges that connects all vertices and has minimum total weight must be a tree. Give an example to show that the same conclusion does not follow if we allow some weights to be nonpositive.

证明：如果一个图的所有边的权重都是正值，则任意一个连接所有结点且总权重最小的一个边集合必然形成一棵树。另外，请举出例子来证明：如果允许某些边的权重为负值，则该论断不成立。

Exercise 24.3-1

Run Dijkstra's algorithm on the directed graph to the right, first using vertex *s* as the source and then using vertex *z*.
In the style of Fig. 24.6, show the *d* and π values and the vertices in set S after each iteration of the **while** loop.



在图 24-2 上运行 Dijkstra 算法，第一次使用结点 s 作为源结点，第二次使用结点 z 作为源结点。以类似于图 24-6 的风格，给出每次 **while** 循环后的 d 值和 π 值，以及集合 S 中的所有结点。