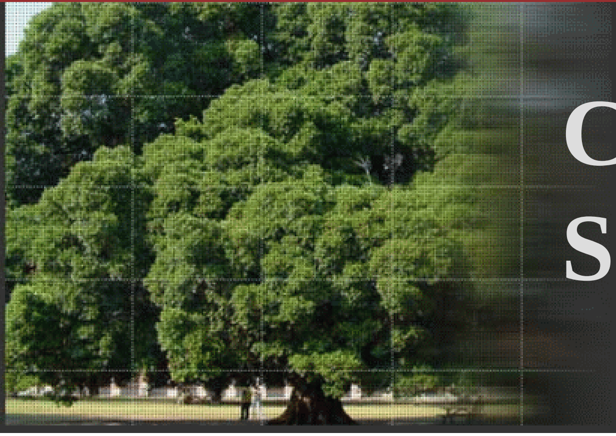
National Cheng Kung University



Chapter 24
Single-Source Shortest Paths

Sun-Yuan Hsieh
謝孫源 教授
成功大學資訊工程學系

Shortest paths

- How to find the shortest route between two points on a map.
  - Input:
    - Directed graph $G = (\boldsymbol{V}, \boldsymbol{E})$
    - Weight function $w : \boldsymbol{E} \rightarrow \mathbf{R}$
  - ***Weight of path p*** $= \langle v_0, v_1, \ldots, v_k \rangle$

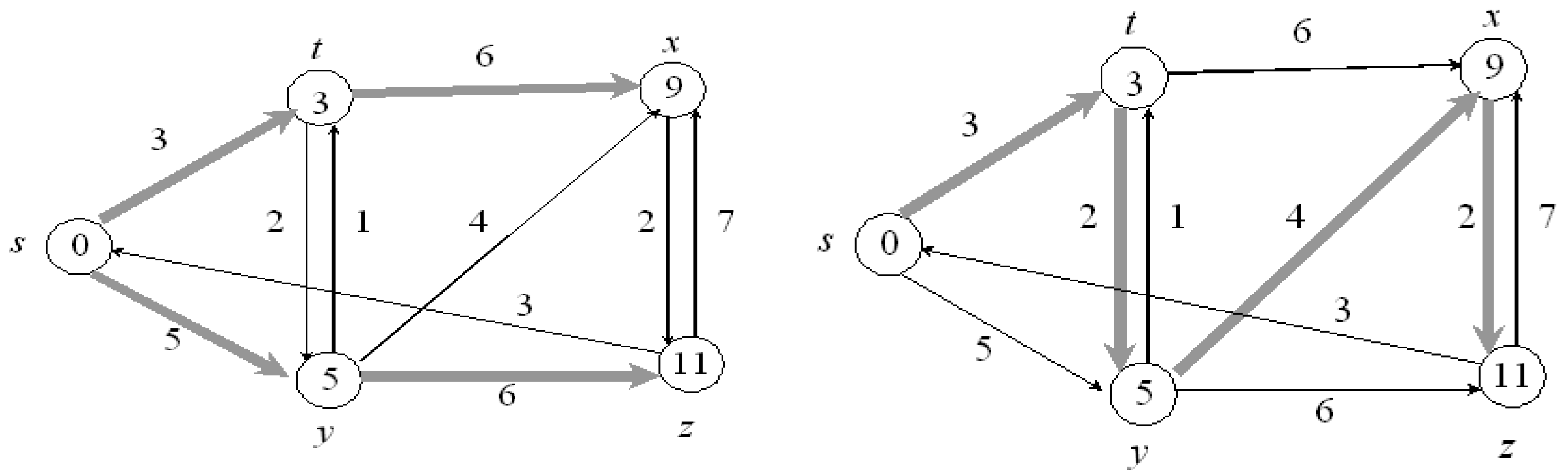
$$= \sum_{i=1}^{k} w(v_{i-1}, v_i)$$

$=$ sum of edge weights on path p.

- ***Shortest-path weight*** u to v :
  - $\delta(u,v) = \begin{cases} \min\{w(p) : u \overset{p}{\leadsto} v\}, & \text{if there exists a path } u \leadsto v \\ \infty & \text{, otherwise.} \end{cases}$
  - Shortest path u to v is any path p such that $w(p) = \delta(u, v)$.

- ***Example***: shortest paths from *s*

  [*d values appear inside vertices. Shaded edges show shortest paths.*]



  - This example shows that the shortest path might not be unique.
  - It also shows that when we look at shortest paths from one vertex to all other vertices, the shortest paths are organized as tree.

- Can think of weights as representing any measure that
  - accumulates linearly along a path.
  - we want to minimize.
- Example: time, cost, penalties, loss.
  - Generalization of breadth-first search to weighted graphs.

Variants

- ***Single-source***
  - Find shortest paths from a given source vertex $s \in V$ to every vertex $v \in V$.
- ***Single-destination***
  - Find shortest paths to a given destination vertex.
- ***Single-pair***
  - Find shortest path from u to v. No way known that's better in worst case than solving single-source.
- ***All-pairs***
  - Find shortest path from u to v for all $u, v \in V$.
  - We'll see algorithms for all-pairs in the next chapter.

Negative-weight edges

- OK, as long as no negative-weight cycles are reachable from the source.
  - If we have a negative-weight cycle, we can just keep going around it, and get $w(s, v) = -\infty$ for all v on the cycle.
  - But OK if the negative-weight cycle is not reachable from the source.
  - Some algorithms work only if there are no negative-weight edges in the graph.

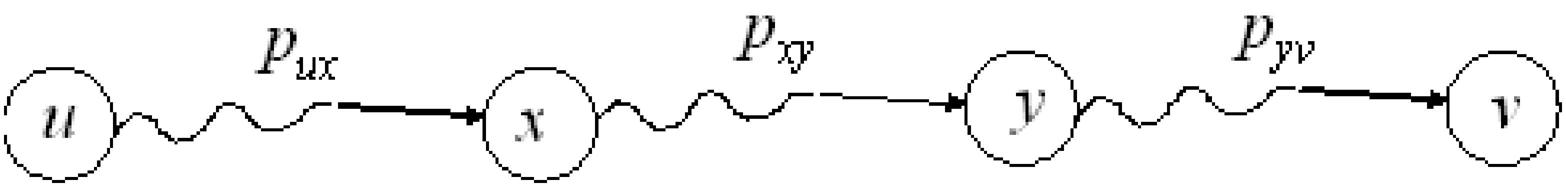
    We'll be clear when they're allowed and not allowed.

Optimal substructure

- ***Lemma***

Any subpath of a shortest path is a shortest path.

***Proof*:** Cut - and - paste.



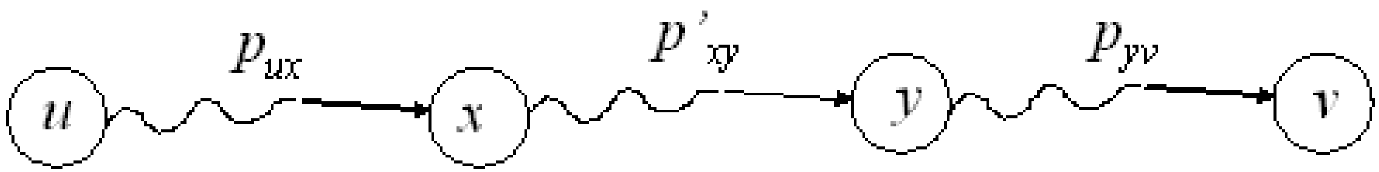
Suppose this path p is a shortest path from u to v.

Then $\delta(u, v) = w(p) = w(p_{ux}) + w(p_{xy}) + w(p_{yv})$.

Now suppose there exists a shorter path $x \overset{P'_{xy}}{\rightsquigarrow} y$.

Then $w(p'_{xy}) < w(p_{xy})$

Construct p'



Then $$\begin{aligned} w(p') &= w(p_{ux}) + w(p'_{xy}) + w(p_{yv}) \\ &< w(p_{ux}) + w(p_{xy}) + w(p_{yv}) \\ &= w(p) \end{aligned}$$

So p wasn't shortest path after all !

■ (lemma)

Cycles

- Shortest paths can't contain cycles :
  - Already ruled out negative-weight cycles.
  - Positive-weight ⇒ we can get a shorter path by omitting the cycle.
  - Zero-weight: no reason to use them ⇒ assume that our solutions won't use them.

Output of single-source shortest-path algorithm

- For each vertex $v \in \boldsymbol{V}$:
  - $d\,[v] = \delta(s, v)$.
    - Initially, $d\,[v] = \infty$.
    - Reduces as algorithms progress. But always maintain $d\,[v] \geq \delta(s, v)$.
    - Call $d\,[v]$ a ***shortest-path estimate***.
  - $\pi\,[v]$ = predecessor of v on a shortest path from s.
    - If no predecessor, $\pi[v]$ = NIL.
    - π induces a tree ------ ***shortest-path tree***.
    - We won't prove properties of π in lecture ------ see text.

Initialization

- All the shortest-paths algorithms start with INIT-SINGLE-SOURCE.

  - INIT-SINGLE-SOURCE (***V***, *s*)

    For each $v \in \boldsymbol{V}$

    do $d\,[v] \leftarrow \infty$

    $\pi[v] \leftarrow$ NIL

    $d\,[s] \leftarrow 0$

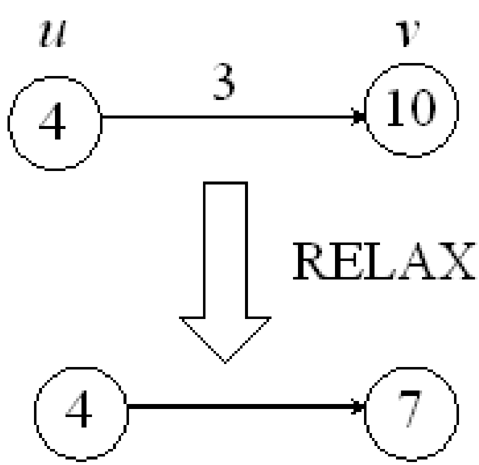
Relaxing an edge (u, v)

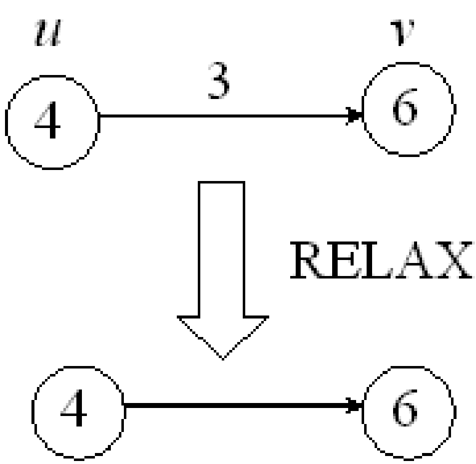
- Can we improve the shortest-path estimate for v by going through u and taking (u, v) ?
  - RELAX (u , v , w)

    If $d[v] > d[u] + w(u, v)$

    then $d[v] \leftarrow d[u] + w(u, v)$

    $\pi[v] \leftarrow u$





- For all the single-source shortest-paths algorithms we'll look at,
  - start by calling INIT-SINGLE-SOURCE,
  - then relax edges.

- The algorithms differ in the order and how many times they relax each edge.

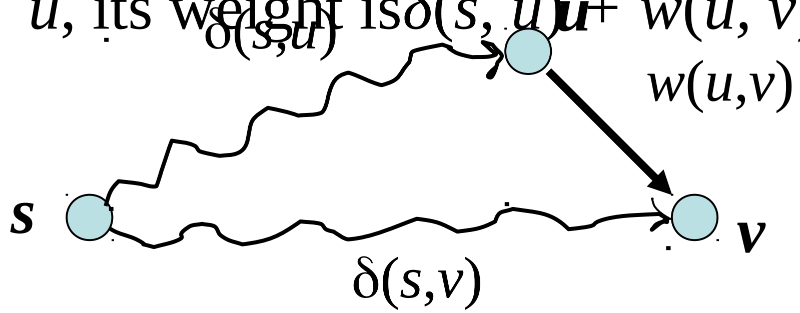
Shortest-paths properties

- Based on calling INIT-SINGLE-SOURCE once and then calling RELAX zero or more times.
- **Triangle inequality**
  - For all $(u, v) \in \boldsymbol{E}$, we have $\delta(s, v) \le \delta(s, u) + w(u, v)$.

    Proof:

    Weight of shortest path $s \rightsquigarrow v$ is $\le$ weight of any path $s \rightsquigarrow v$.

    Path $s \rightsquigarrow u \rightarrow v$ is a path $s \rightsquigarrow v$, and if we use a shortest path $s \rightsquigarrow u$, its weight is $\delta(s, u) + w(u, v)$. ■



Upper-bound property

- Always have $d[v] \geq \delta(s, v)$ for all v.
  Once $d[v] = \delta(s, v)$, it never changes.

Proof: Initially true.

Suppose there exists a vertex such that $d[v] < \delta(s, v)$.

Without loss of generality, v is first vertex for which this happens.

Let u be the vertex that causes $d[v]$ to change.

Then $d[v] = d[u] + w(u, v)$.

So, $d[v] < \delta(s, v)$

$\leq \delta(s, u) + w(u, v)$ (triangle inequality)

$\leq d[u] + w(u, v)$ (v is first violation)

$\Rightarrow d[v] < d[u] + w(u, v)$. (Contradicts $d[v] = d[u] + w(u, v)$)

Once $d[v]$ reaches $\delta(s, v)$, it never goes lower. It never goes up, since relaxations only lower shortest-path estimates. ■

No-path property

- If $\delta(s, v) = \infty$, then $d[v] = \infty$ always.

***Proof*:**

$d[v] \geq \delta(s, v) = \infty \Rightarrow d[v] = \infty$. ■

Convergence property

- If $s \leadsto u \rightarrow v$ is a shortest path, $d[u] = \delta(s, u)$, and we call RELAX(u, v, w), then $d[v] = \delta(s, v)$ afterward.

  Proof:

  After relaxation:

  $d[v] \leq d[u] + w(u, v)$ (RELAX code)

  $\quad = \delta(s, u) + w(u, v)$

  $\quad = \delta(s, v)$ (lemma ----- optimal substructure)

  Since $d[v] \geq \delta(s, v)$, must have $d[v] = \delta(s, v)$. ■

Path relaxation property

- Let $p = < v_0, v_1, \ldots, v_k >$ be a shortest path from $s = v_0$ to v_k. If we relax, *in order*, $(v_0, v_1), (v_1, v_2), \ldots, (v_{k-1}, v_k)$, even intermixed with other relaxations, then $d[v_k] = \delta(s, v_k)$.

  Proof:

  Induction to show that $d[v_i] = \delta(s, v_i)$ after (v_{i-1}, v_i) is relaxed.

  Basis:

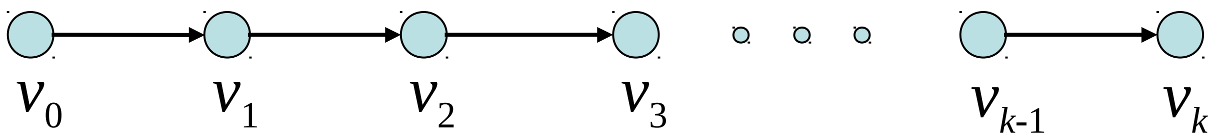
  $i = 0$. Initially, $d[v_0] = 0 = \delta(s, v_0) = \delta(s, s)$.

  Inductive step:

  Assume $d[v_{i-1}] = \delta(s, v_{i-1})$.

  Relax (v_{i-1}, v_i).

  By Convergence Property, $d[v_i] = \delta(s, v_i)$ afterward and $d[v_i]$ never changes.



- $d(v_1) = \delta(s,v_0) + w(v_0,v_1) = \delta(s,v_1)$

  $d(v_2) = \delta(s,v_1) + w(v_1,v_2) = \delta(s,v_2)$

  $d(v_3) = \delta(s,v_2) + w(v_2,v_3) = \delta(s,v_3)$

  $\vdots$

  $d(v_k) = \delta(s,v_{k-1}) + w(v_{k-1},v_k) = \delta(s,v_k)$

The Bellman-Ford algorithm

- Allows negative-weight edges.
- Computes $d\,[v]$ and $\pi[v]$ for all $v \in \boldsymbol{V}$.
- Returns TRUE if no negative-weight cycles reachable from s, FALSE otherwise.

- BELLMAN-FORD ($\boldsymbol{V}$, $\boldsymbol{E}$, w, s)

  INIT-SINGLE-SOURCE ($\boldsymbol{V}$, s)

  for $i \leftarrow 1$ to $|\boldsymbol{V}| - 1$

      do for each edge $(u, v) \in \boldsymbol{E}$

          do RELAX (u, v, w)

  for each edge $(u , v) \in \boldsymbol{E}$

      do if $d[v] > d[u] + w(u, v)$

              then return FALSE

  return TRUE

  Core: The first **for** loop relaxes all edges $|\boldsymbol{V}| - 1$ times.

  Time: $\Theta(\boldsymbol{VE})$.

- $i = 1$

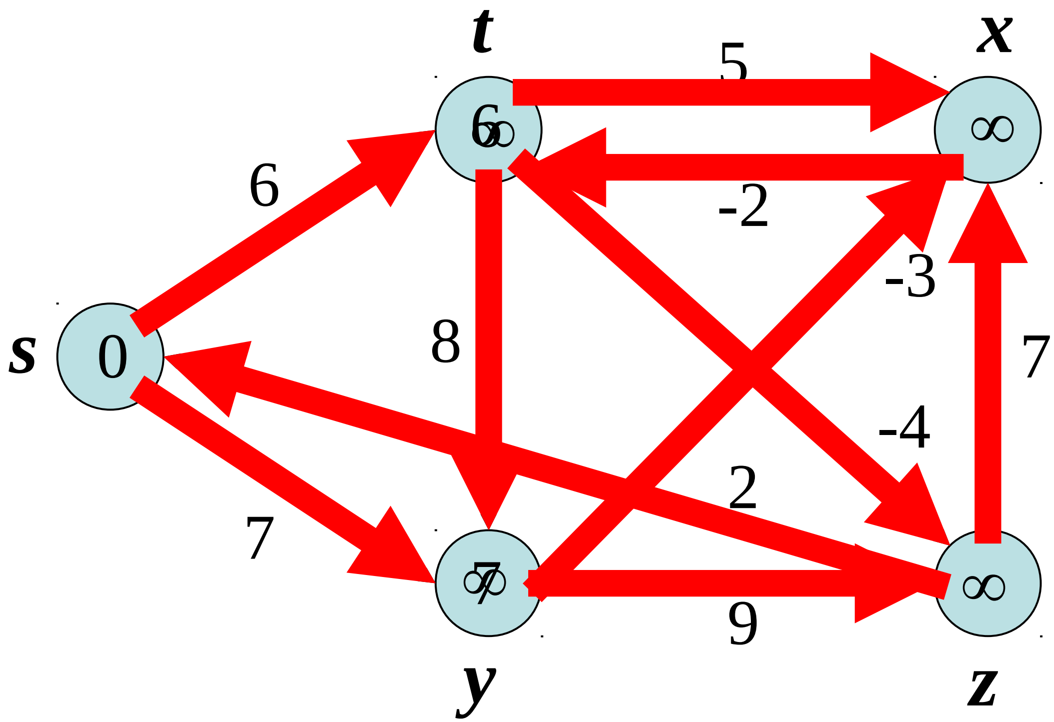


Figure 24.4 The execution of the Bellman-Ford algorithm. The source is vertex s. The d values are shown within the vertices, and shaded edges indicate predecessor values: if edge (u, v) is shaded, then $\pi[v] = u$. In this particular example, each pass relaxes the edges in the order $(t, x), (t, y), (t, z), (x, t), (y, x), (y, z), (z, x), (z, s), (s, t), (s, y)$. **(a)** The situation just before the first pass over the edges. **(b)–(e)** The situation after each successive pass over the edges. The d and π values in part (e) are the final values. The Bellman-Ford algorithm returns TRUE in this example.

- $i = 2$

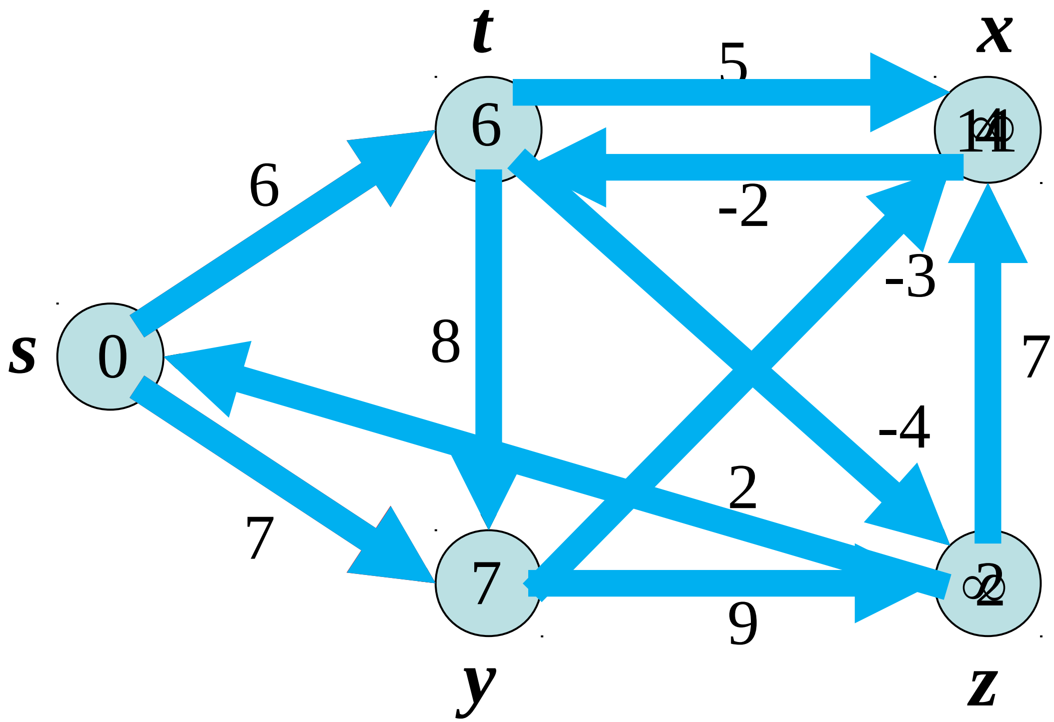


Figure 24.4 The execution of the Bellman-Ford algorithm. The source is vertex s. The d values are shown within the vertices, and shaded edges indicate predecessor values: if edge (u, v) is shaded, then $\pi[v] = u$. In this particular example, each pass relaxes the edges in the order $(t, x), (t, y), (t, z), (x, t), (y, x), (y, z), (z, x), (z, s), (s, t), (s, y)$. **(a)** The situation just before the first pass over the edges. **(b)–(e)** The situation after each successive pass over the edges. The d and π values in part (e) are the final values. The Bellman-Ford algorithm returns TRUE in this example.

- $i = 3$

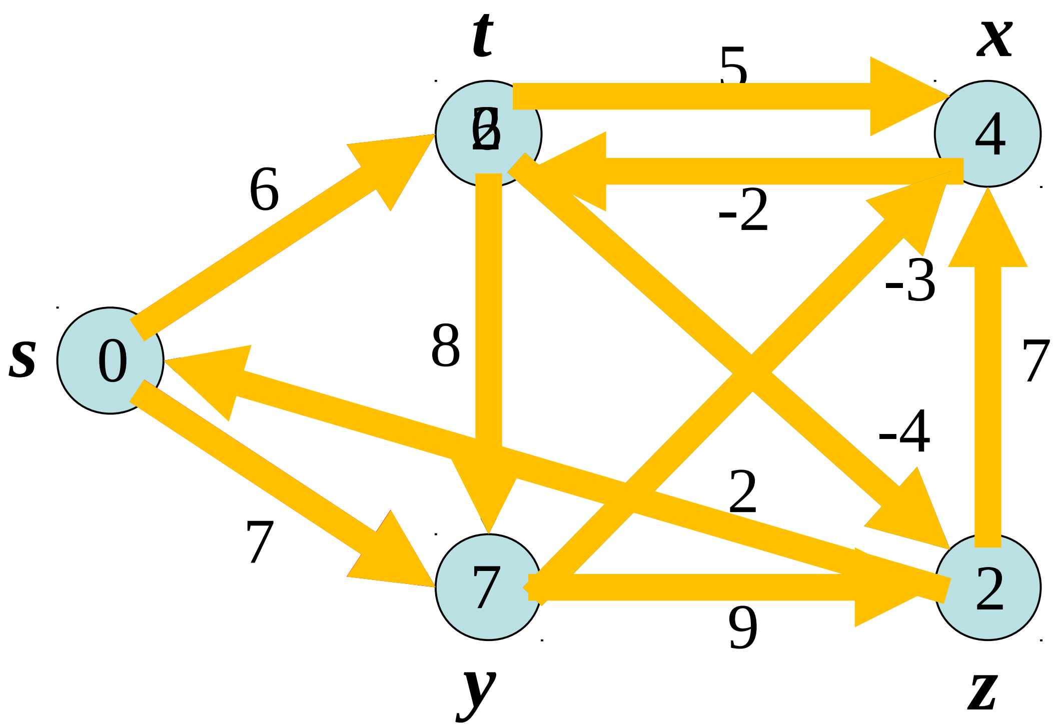


Figure 24.4 The execution of the Bellman-Ford algorithm. The source is vertex s. The d values are shown within the vertices, and shaded edges indicate predecessor values: if edge (u, v) is shaded, then $\pi[v] = u$. In this particular example, each pass relaxes the edges in the order $(t, x), (t, y), (t, z), (x, t), (y, x), (y, z), (z, x), (z, s), (s, t), (s, y)$. **(a)** The situation just before the first pass over the edges. **(b)–(e)** The situation after each successive pass over the edges. The d and π values in part (e) are the final values. The Bellman-Ford algorithm returns TRUE in this example.

- $i = 4$

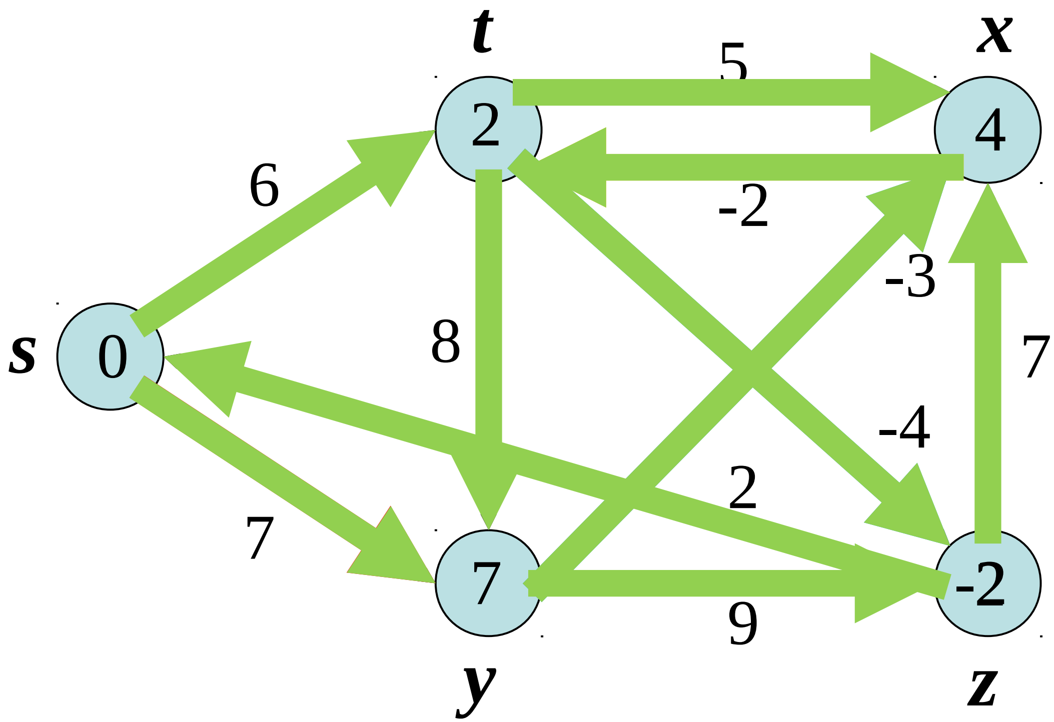
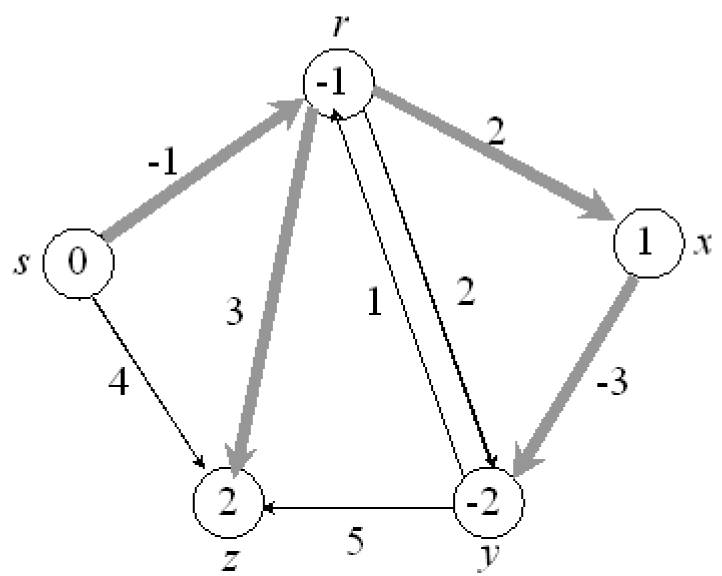


Figure 24.4 The execution of the Bellman-Ford algorithm. The source is vertex s. The d values are shown within the vertices, and shaded edges indicate predecessor values: if edge (u, v) is shaded, then $\pi[v] = u$. In this particular example, each pass relaxes the edges in the order $(t, x), (t, y), (t, z), (x, t), (y, x), (y, z), (z, x), (z, s), (s, t), (s, y)$. **(a)** The situation just before the first pass over the edges. **(b)–(e)** The situation after each successive pass over the edges. The d and π values in part (e) are the final values. The Bellman-Ford algorithm returns TRUE in this example.

- ***Example***:



  - Values you get on each pass and how quickly it converges depends on order of relaxation.
  - But guaranteed to converge after $|V| - 1$ passes, assuming no negative-weight cycles.

- ***Proof***: Use path-relaxation property.

Let v be reachable from s, and let $p = \langle v_0, v_1, \ldots, v_k \rangle$ be a shortest path from s to v, where $v_0 = s$ and $v_k = v$.

Since p is acyclic, it has $\le |\boldsymbol{V}| - 1$ edges, so $k \le |\boldsymbol{V}| - 1$.

Each iteration of the **for** loop relaxes all edges:

- First iteration relaxes (v_0, v_1).
- Second iteration relaxes (v_1, v_2).
- kth iteration relaxes (v_{k-1}, v_k).

By the path-relaxation property, $d[v] = d[v_k] = \delta(s, v_k) = \delta(s, v)$.

■

- How about the TRUE/FALSE return value?
  - Suppose there is no negative-weight cycle reachable from s.

    At termination, for all $(u, v) \in \boldsymbol{E}$,

    $d\,[v] = \delta(s, v)$

    $\qquad \le \delta(s, u) + w(u, v)$ (triangle inequality)

    $\qquad = d\,[u] + w(u, v)$

    So BELLMAN-FORD returns TRUE.

- Now suppose there exists negative-weight cycle $c = \langle v_0, v_1, \ldots, v_k \rangle$ where $v_0 = v_k$, reachable from s.
  Then $\sum_{i=1}^{k} w(v_{i-1}, v_i) < 0$
  Suppose (for contradiction) that BELLMAN-FORD returns TRUE.
  Then $d[v_i] \le d[v_{i-1}] + w(v_{i-1}, v_i)$ for $i = 1,2,\ldots,k$.
  sum around c:

$$\sum_{i=1}^{k} d[v_i] \le \sum_{i=1}^{k} (d[v_{i-1}] + w(v_{i-1}, v_i))$$

$$= \sum_{i=1}^{k} d[v_{i-1}] + \sum_{i=1}^{k} w(v_{i-1}, v_i)$$

Each vertex appears once in each summation $\sum_{i=1}^{k} d[v_i]$ and $\sum_{i=1}^{k} d[v_{i-1}]$.

$$\Rightarrow \; 0 \le \sum_{i=1}^{k} w(v_{i-1}, v_i) .$$

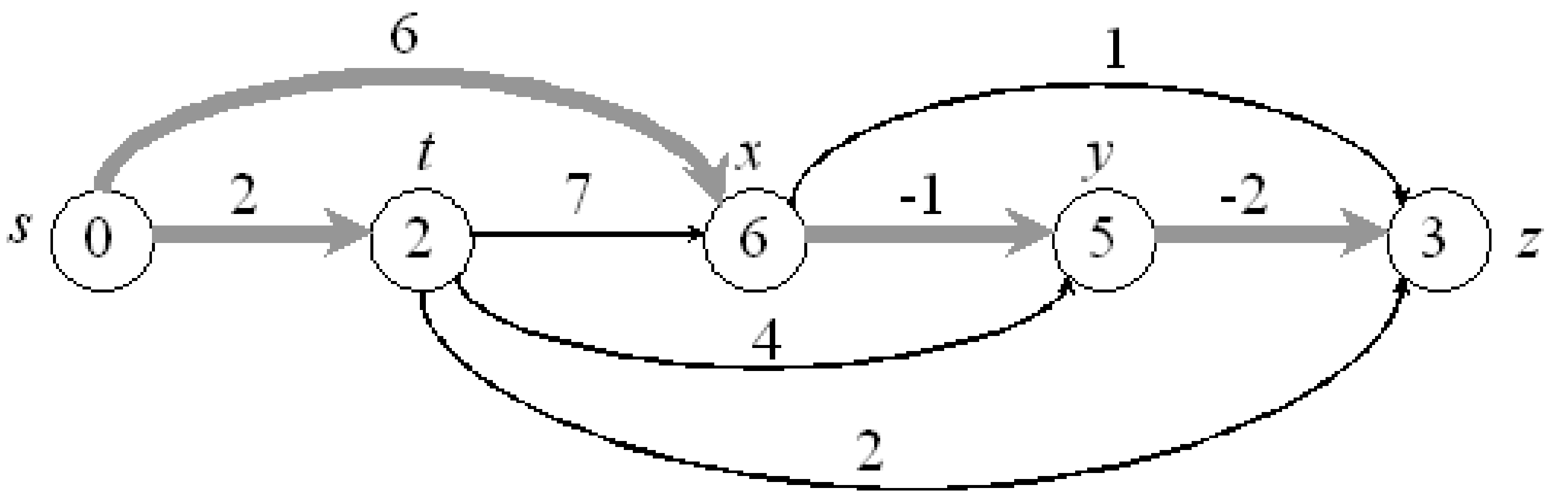
This contradicts C being a negative-weight cycle! ■

Single-source shortest paths in a directed acyclic graph

- Since a dag, we're guaranteed no negative-weight cycles.

DAG-SHORTEST-PATHS (***V***, ***E***, *w*, *s*)
topologically sort the vertices
INIT-SINGLE-SOURCE (***V***, *s*)
for each vertex *u*, taken in topologically sorted order
    do for each vertex $v \in Adj[u]$
        do RELAX (*u*, *v*, *w*)

- ***Example***:



- ***Time***:$\Theta(V + E)$.

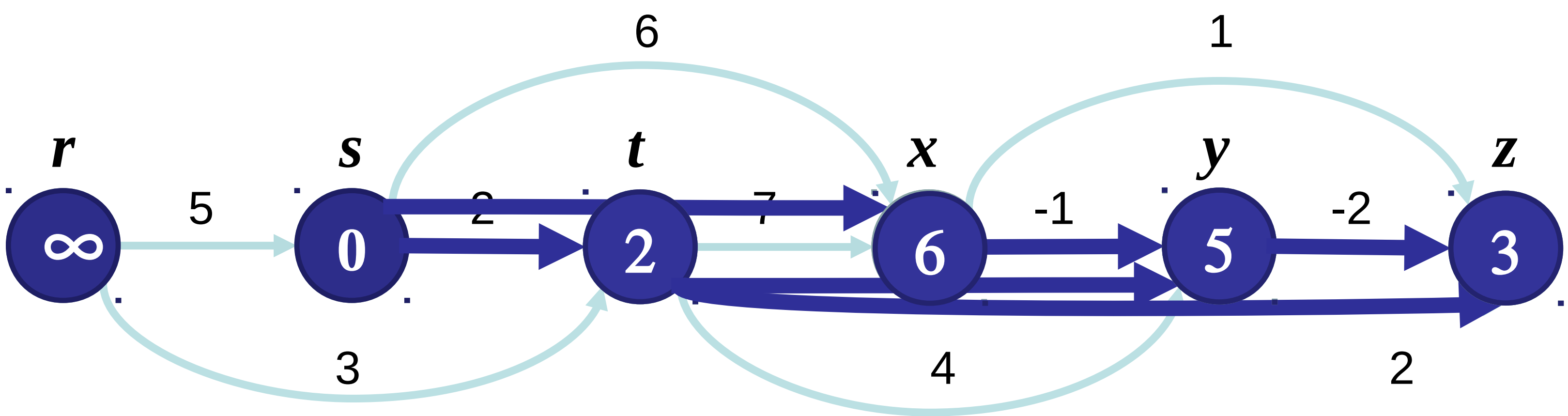


Figure 24.5 The execution of the algorithm for shortest paths in a directed acyclic graph. The vertices are topologically sorted from left to right. The source vertex is *s*. The *d* values are shown within the vertices, and shaded edges indicate the π values.

- ***Correctness***: Because we process vertices in topologically sorted order, edges of any path must be relaxed in order of appearance in the path.

  ⇒ Edges on any shortest path are relaxed in order.

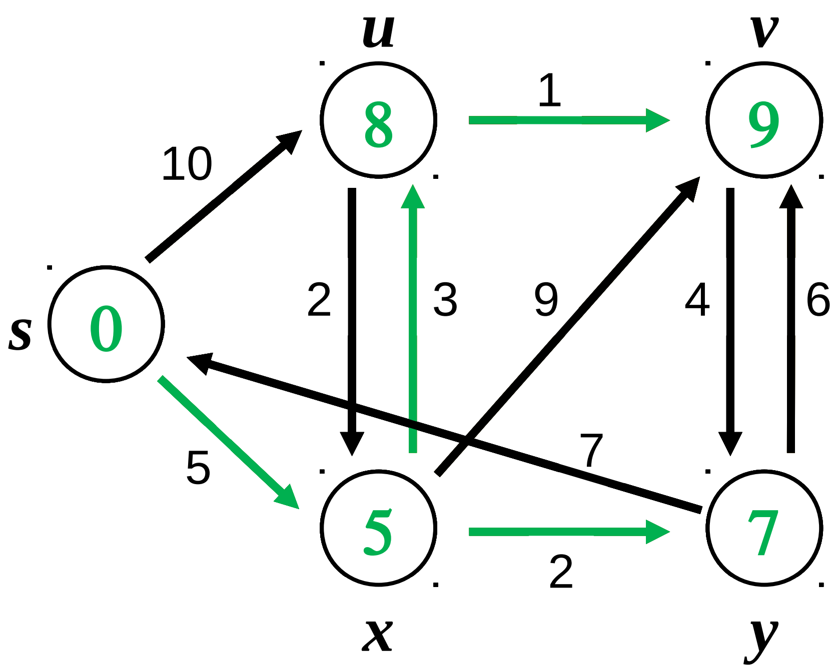
  ⇒ By path-relaxation property, correct. ■

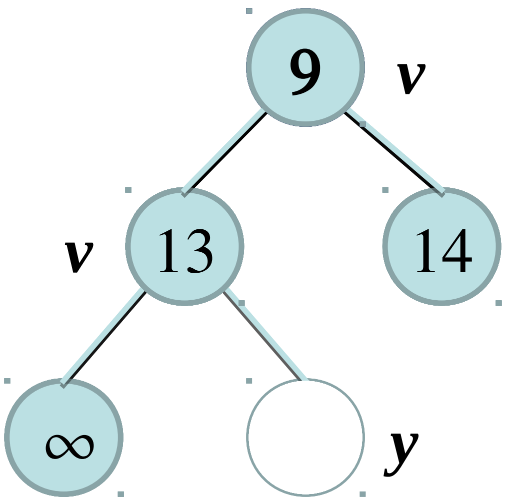
Dijkstra's algorithm

- No negative-weight *edges*.
- Essentially a weighted version of breadth-first search.
  - Instead of a **FIFO** queue, uses a priority queue.
  - Keys are shortest-path weights ($d[v]$).
- Have two sets of vertices:
  - S = vertices whose final shortest-path weights are determined.
  - Q = priority queue = $\boldsymbol{V} - S$.

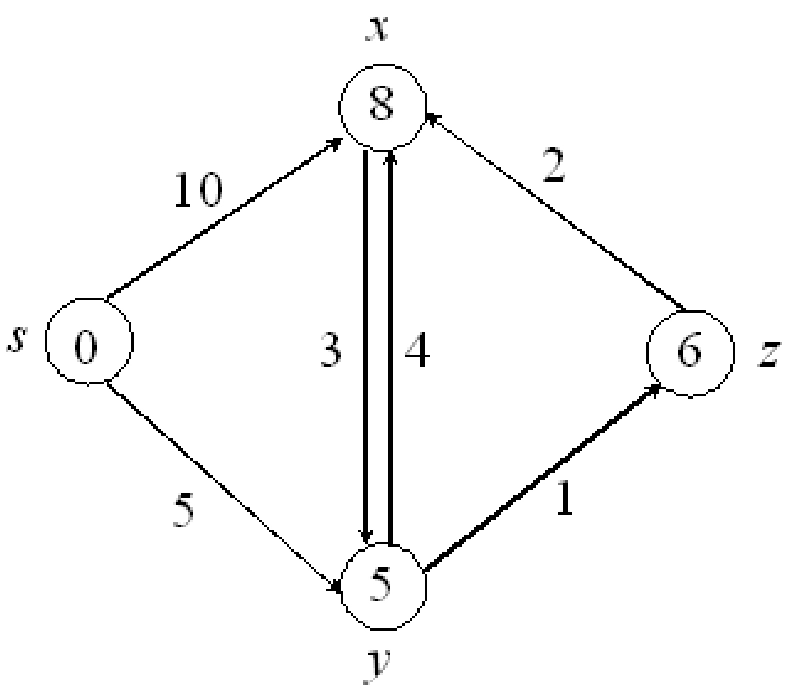
DIJKSTRA ($\boldsymbol{V}$, $\boldsymbol{E}$, w, s)
INIT-SINGLE-SOURCE ($\boldsymbol{V}$, s)
$S \leftarrow \emptyset$
$Q \leftarrow \boldsymbol{V}$ // i.e., insert all vertices into Q
While $Q \neq \emptyset$
  do $u \leftarrow$ EXTRACT-MIN(Q)
    $S \leftarrow S \cup \{u\}$
    for each vertex $v \in$ A*dj* [u]
      do RELAX (u, v, w)

- Like Prim's algorithm, but computing d [v], and using shortest-path weights as keys.
- Dijkstra's algorithm can be viewed as greedy, since it always chooses the "lightest" ("closest"?) vertex in $\boldsymbol{V} - S$ to add to S.


u
v
s
x
y
8
9
0
5
7
10
1
2
3
9
4
6
5
7
2


9
v
v
13
14
∞
y

- ***Example:***



Order of adding to *S*: *s*, *y*, *z*, *x*.

- ***Correctness***:
  - **Loop invariant** : At the start of each iteration of the **while** loop, $d[v] = \delta(s, v)$ for all $v \in S$.
  - **Initialization**: Initially, $S = \emptyset$, so trivially true.
  - **Termination**: At end, $Q = \emptyset \Rightarrow S = V \Rightarrow d[v] = \delta(s, v)$ for all $v \in V$.

- ***Maintenance***:

  Need to show that $d[u] = \delta(s, u)$ when u is added to S in each iteration.

  Suppose there exists u such that $d[u] \neq \delta(s, u)$. Without loss of generality, let u be the first vertex for which $d[u] \neq \delta(s, u)$ when u is added to S.

  Observations:

  - $u \neq s$, since $d[s] = \delta(s, s) = 0$.
  - Therefore, $s \in S$, so $S \neq \emptyset$.
  - There must be some path $s \leadsto u$,

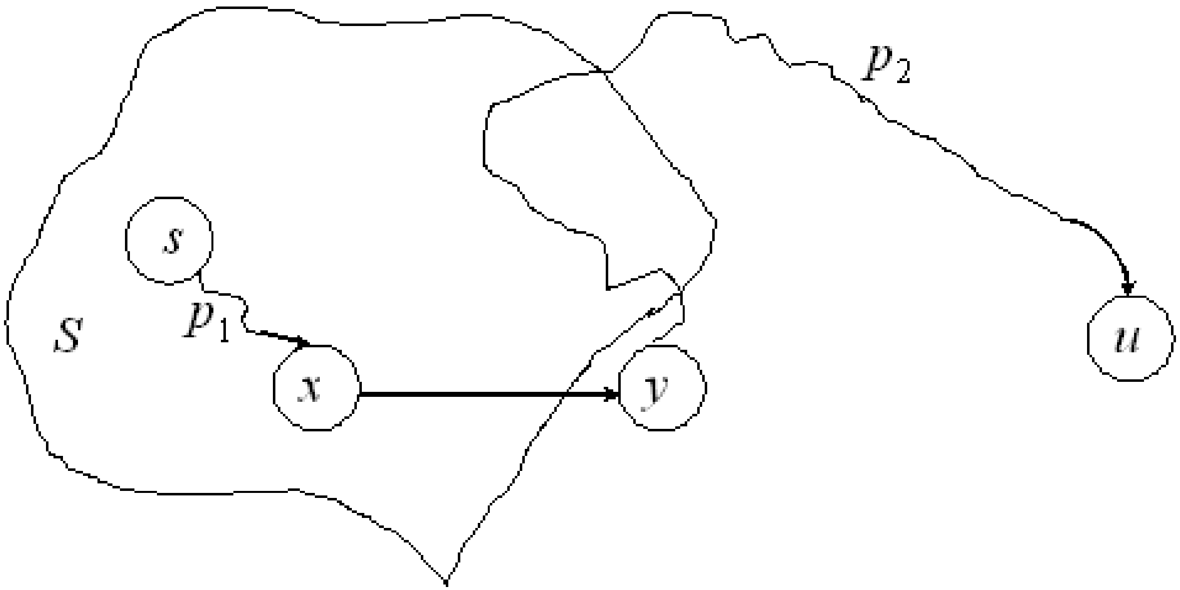
    since otherwise $d[u] = \delta(s, u) = \infty$ by no-path property.

So, there's a path $s \rightsquigarrow u$.

This means there's a shortest path $s \overset{p}{\rightsquigarrow} u$.

Just before u is added to S, path p connects a vertex in S (i.e., s) to a vertex in $\boldsymbol{V} - S$ (i.e., u).

Let y be first vertex along p that's in $\boldsymbol{V} - S$, and let $x \in S$ be y's predecessor.



Decompose p into $s \overset{p_1}{\leadsto} x \to y \overset{p_2}{\leadsto} u$.

(Could have $x = s$ or $y = u$, so that p_1 or p_2 may have no edges.)

- ***Claim***

  $d\,[y] = \delta(s, y)$ when u is added to S.

  Proof

  $x \in S$ and u is the first vertex such that $d\,[u] \neq \delta(s, u)$ when u is added to $S \Rightarrow d\,[x] = \delta(s, x)$ when x is added to S.

  Relaxed (x, y) at that time, so by the convergence property,

  $d\,[y] = \delta(s, y)$. ■ (claim)

Now can get a contradiction to $d[u] \neq \delta(s, u)$:

y is on shortest path $s \rightsquigarrow u$, and all edge weights are nonnegative

$\Rightarrow \delta(s, y) \leq \delta(s, u)$

$\Rightarrow d[y] = \delta(s, y)$

$\leq \delta(s, u)$

$\leq d[u]$ (upper-bound property).

Also, both y and u were in Q when we chose u, so $d[u] \leq d[y]$

$\Rightarrow d[u] = d[y]$.

Therefore, $d[y] = \delta(s, y) = \delta(s, u) = d[u]$.

Contradicts assumption that $d[u] \neq \delta(s, u)$.

Hence, Dijkstra's algorithm is correct. ■

- ***Analysis***: Like Prim's algorithm, depends on implementation of priority queue.
  - If binary heap, each operation takes O (lg $\boldsymbol{V}$) time
    $\Rightarrow O$ ($\boldsymbol{E}$ lg $\boldsymbol{V}$).
  - If a Fibonacci heap:
    - Each EXTRACT-MIN takes $O(1)$ amortized time.
    - There are O ($\boldsymbol{V}$) other operations, taking O (lg $\boldsymbol{V}$) amortized time each.
    - Therefore, time is O ($\boldsymbol{V}$ lg $\boldsymbol{V}$ + $\boldsymbol{E}$).

Difference constraints

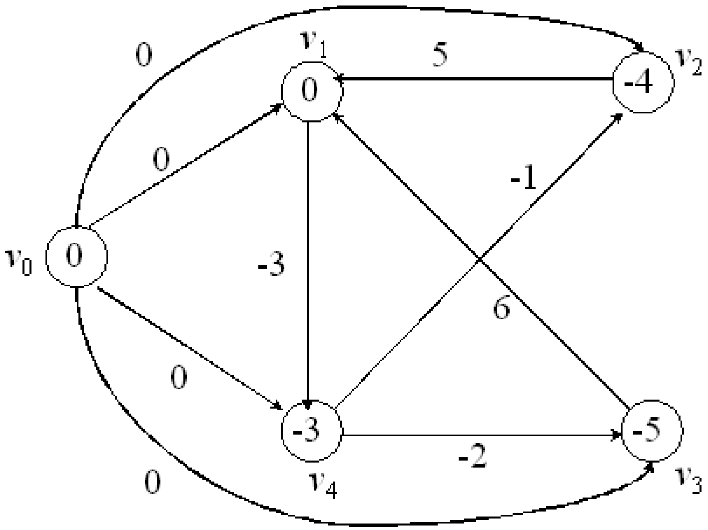
- Given a set of inequalities of the form $x_j - x_i \le b_k$.
  - x's are variables, $1 \le i, j \le n$,
  - b's are constants, $1 \le k \le m$.

Want to find a set of values for the x's that satisfy all m inequalities, or determine that no such values exist.

Call such a set of values a ***feasible solution***.

- ***Example*:**

$$x_1 - x_2 \le 5$$
$$x_1 - x_3 \le 6$$
$$x_2 - x_4 \le -1$$
$$x_3 - x_4 \le -2$$
$$x_4 - x_1 \le -3$$



Solution: x = (0, -4, -5, -3)

Also: x = (5, 1, 0, 2) = [above solution] + 5

Lemma

If x is a feasible solution, then so is $x + d$ for any constant d.

Proof

x is a feasible solution

$\Rightarrow \quad x_j - x_i \le b_k$ for all i, j, k.

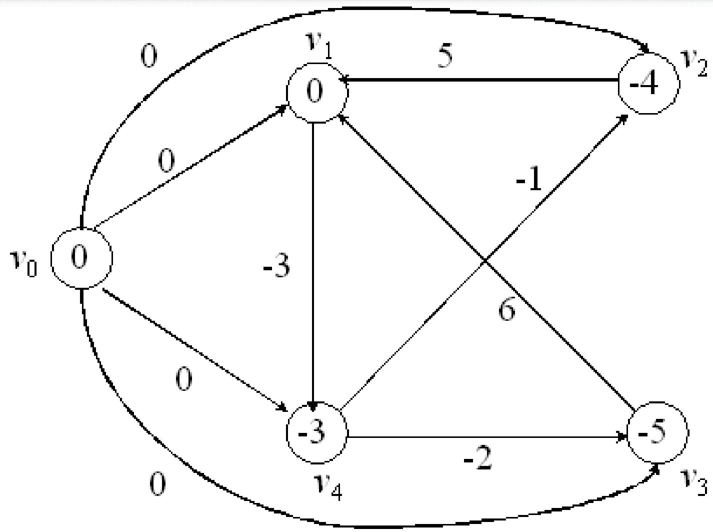
$\Rightarrow \ (x_j + d) - (x_i + d) \le b_k$. ■ (lemma)

- **Constraint graph**

  $\boldsymbol{G} = (\boldsymbol{V}, \boldsymbol{E})$, weighted, directed.

  - $V = < v_0, v_1, v_2, \ldots, v_n >$: one vertex per variable + v_0
  - $E = \{(v_i, v_j) : x_j - x_i \le b_k$ is a constraint$\}$

    $\cup \{(v_0, v_1), (v_0, v_2), \ldots, (v_0, v_n)\}$
  - $w(v_0, v_j) = 0$ for all j
  - $w(v_i, v_j) = b_k$ if $x_j - x_i \le b_k$



- ***Theorem***

  Given a system of difference constraints, let $\boldsymbol{G} = (\boldsymbol{V}, \boldsymbol{E})$ be the corresponding Constraint graph.

  1. If G has no negative-weight cycles, then
     $x = (\delta(v_0, v_1), \delta(v_0, v_2), \ldots, \delta(v_0, v_n))$ is a feasible solution.
  2. If G has a negative-weight cycle, then there is no feasible solution.

- ***Proof***

1. Show no negative-weight cycles ⇒ feasible solution.

Need to show that $x_j - x_i \le b_k$ for all constraints. Use

$$x_j = \delta(v_0, v_j)$$

$$x_i = \delta(v_0, v_i)$$

$$b_k = w(v_i, v_j)$$

By the triangle inequality,

$$\delta(v_0, v_j) \le \delta(v_0, v_i) + w(v_i, v_j)$$

$$x_j \le x_i + b_k$$

$$x_j - x_i \le b_k$$

Therefore, feasible.

2. Show negative-weight cycles ⇒ no feasible solution.

Without loss of generality, let a negative-weight cycle be $C = \langle v_1, v_2, \ldots, v_k \rangle$ where $v_1 = v_k$ (v_0 can't be on C, since v_0 has no entering edges.) C corresponds to the constraints

$$x_2 - x_1 \le w(v_1, v_2)$$
$$x_3 - x_2 \le w(v_2, v_3)$$
$$\vdots$$
$$x_{k-1} - x_{k-2} \le w(v_{k-2}, v_{k-1})$$
$$x_k - x_{k-1} \le w(v_{k-1}, v_k)$$

(The last two inequalities above are incorrect in the first three printings of the book. They were corrected in the fourth printing.)

If x is a solution satisfying these inequalities, it must satisfy their sum.

So add them up.

Each x_i is added once and subtracted once. ($v_1 = v_k \Rightarrow x_1 = x_k$)

We get $0 \le w(C)$.

But $w(C) < 0$, since C is a negative-weight cycle.

Contradiction $\Rightarrow$ no such feasible solution x exists.

■ (theorem)

- How to find a feasible solution

1. Form constraint graph.
   - $n + 1$ vertices.
   - $m + n$ edges.
   - $\Theta(m + n)$ time.
2. Run BELLMAN-FORD from v_0 .
   - $O((n + 1)(m + n)) = O(n^2 + nm)$ time.
3. If BELLMAN-FORD returns FALSE ⇒ no feasible solution.

   If BELLMAN-FORD returns TURE

   ⇒ set $x_i = \delta(v_0, v_i)$ for all i.