CSIE 2136 Algorithm Design and Analysis, Fall 2020

Graph Algorithms - III

Hsu-Chun Hsiao

3.5-week Agenda

- Graph basics
  - Graph terminology [B.4, B.5]
  - Real-world applications
  - Graph representations [Ch. 22.1]
- Graph traversal
  - Breadth-first search (BFS) [Ch. 22.2]
  - Depth-first search (DFS) [Ch. 22.3]
- DFS applications
  - Topological sort [Ch. 22.4]
  - Strongly-connected components [Ch. 22.5]
- Minimum spanning trees [Ch. 23]
  - Kruskal's algorithm
  - Prim's algorithm
- Single-source shortest paths [Ch. 24]
  - Dijkstra algorithm
  - Bellman-Ford algorithm
  - SSSP in DAG
- All-pairs shortest paths [Ch. 25]
  - Floyd-Warshall algorithm
  - Johnson’s algorithm

Today's Agenda

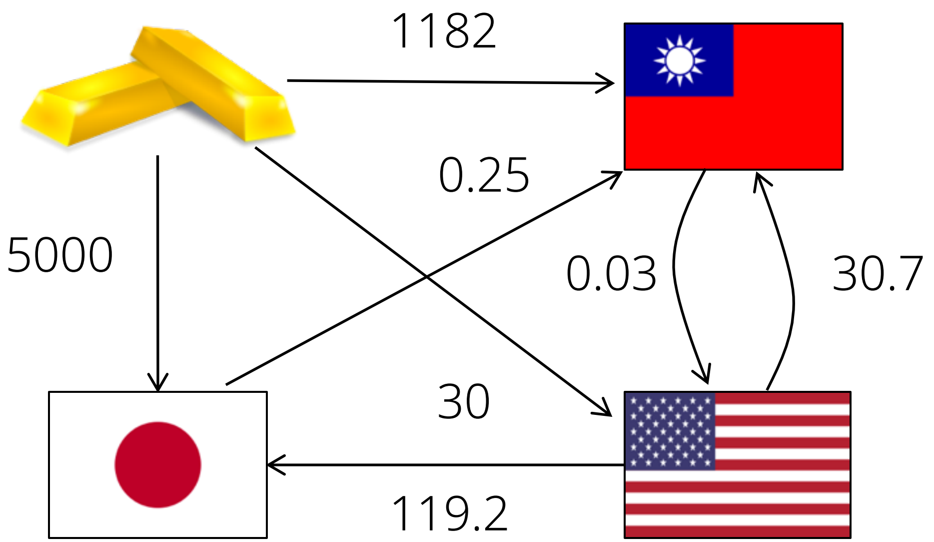
- Shortest paths: terminology and properties
  - Edge relaxation
  - Shortest-paths properties
- Single-source shortest paths [Ch. 24]
  - Bellman-Ford algorithm
  - Dijkstra algorithm
  - SSSP in DAG

Bellman-Ford algorithm

Textbook Chapter 24.1

匯率換算

- 1 公克黃金最多可以換到多少 TWD？（假設零手續費）



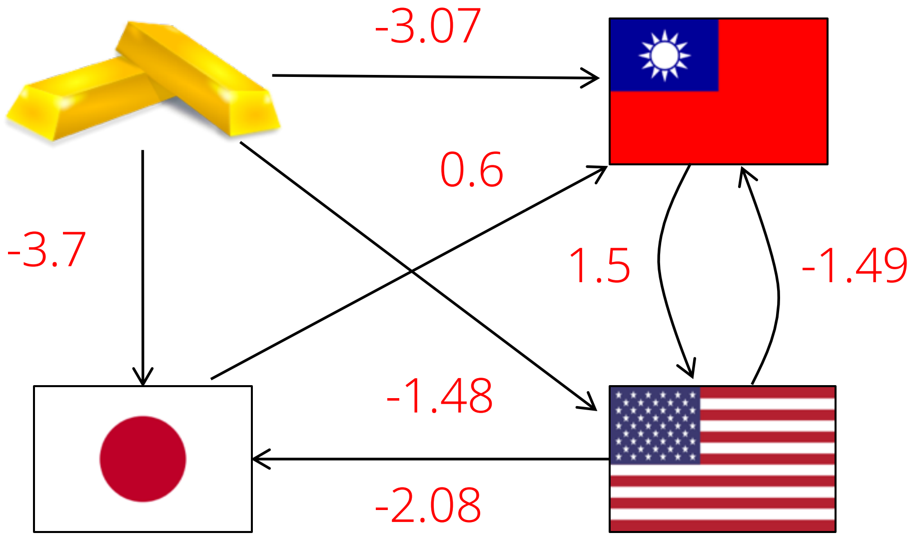
找weight相乘後最大路徑？

如何轉成我們熟悉的最短路徑問題？

匯率換算

- 1 公克黃金最多可以換到多少 TWD？（假設零手續費)

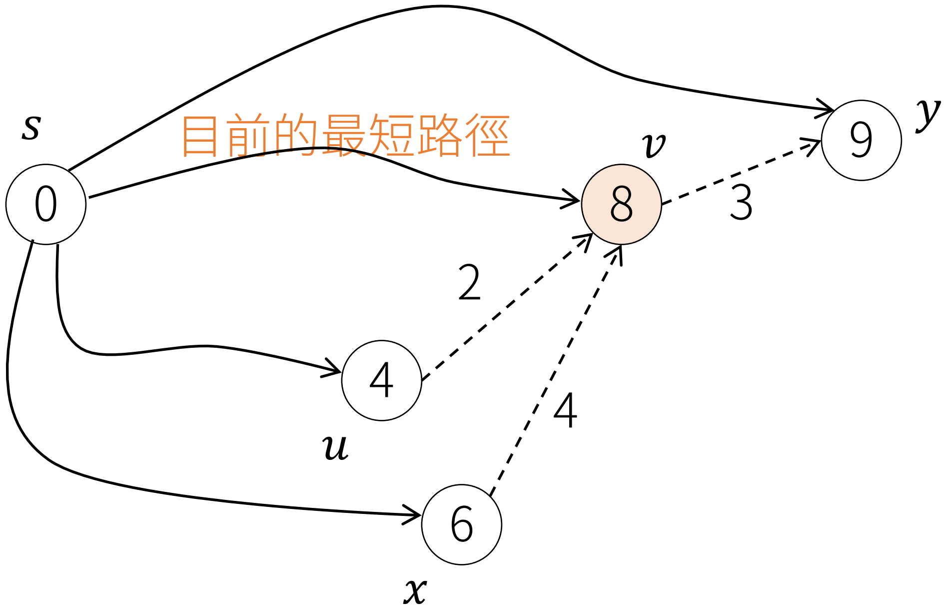
Reweighting:
$w'(e) = -\log w(e)$



We want to detect the existence of negative cycles (利用匯差賺錢)
and find the shortest path (最佳的兌換率)

Bellman-Ford algorithm: intuition

- 共執行 $|V|-1$ 回合
  - 每一回合中，relax 所有的邊
  - 節點 v 一方面接收從各個上游來的最短路徑估計值，試試看改走不同上游會不會比較好，另一方面把自己的估計值傳給下游節點



u 和 x 為 v 的上游，
y 為 v 的下游

以 v 的觀點來看，
每一回合會 relax $(u,v),(x,v),(v,y)$
順序不重要

Bellman-Ford algorithm: intuition

- Bellman-Ford 保證在第 k 回合結束後，節點 v 的最短路徑估計值 $\leq$ 所有邊數至多為 k 的 $s \rightsquigarrow v$ 路徑的最短距離
- => $|V|-1$ 回合結束後，節點v的最短路徑估計值 $\leq$ 所有邊數至多為 $|V|-1$ 的 $s \rightsquigarrow v$ 路徑的最短距離
- => 若最短路徑存在，由於最短路徑的邊數不會大於 $|V|-1$，因此 Bellman-Ford 結束後的確能正確算出最短路徑值

Bellman-Ford algorithm

```
BELLMAN-FORD(G,w,s)
    INITIALIZE-SINGLE-SOURCE(G,s)
    for i = 1 to |G.V|-1
        for (u,v) in G.E
            RELAX(u,v,w)
    for (u,v) in G.E
        if v.d > u.d + w(u,v)
            return FALSE
    return TRUE
```

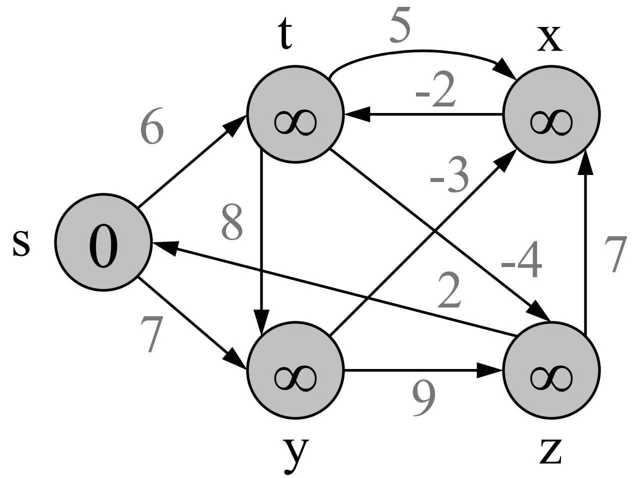
```
INITIALIZE-SINGLE-SOURCE(G,s)
    for v in G.V
        v.d = ∞
        v.π = NIL
    s.d = 0
```

```
RELAX(u, v, w)
    if v.d > u.d + w(u, v)
        //DECREASE-KEY
        v.d = u.d + w(u, v)
        v.π = u
```

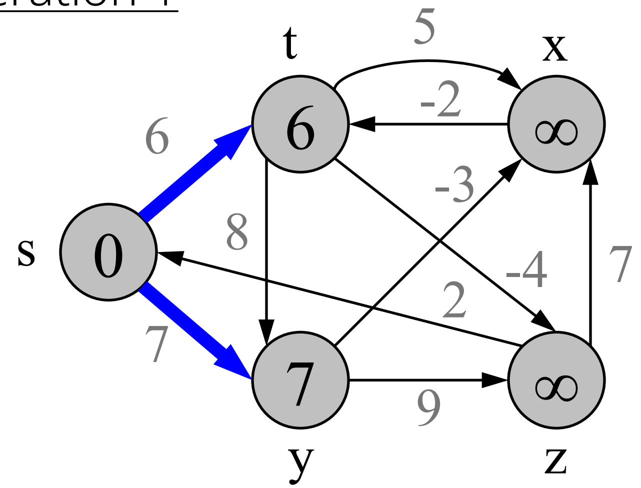
- Relax each edge e; repeat $V - 1$ times
- Detect a negative cycle if exists
- Find shortest simple path if no negative cycle exists

Relaxation sequence in each iteration: $(t,x),(t,y),(t,z),(x,t),(y,x),(y,z),(z,x),(z,s),(s,t),(s,y)$

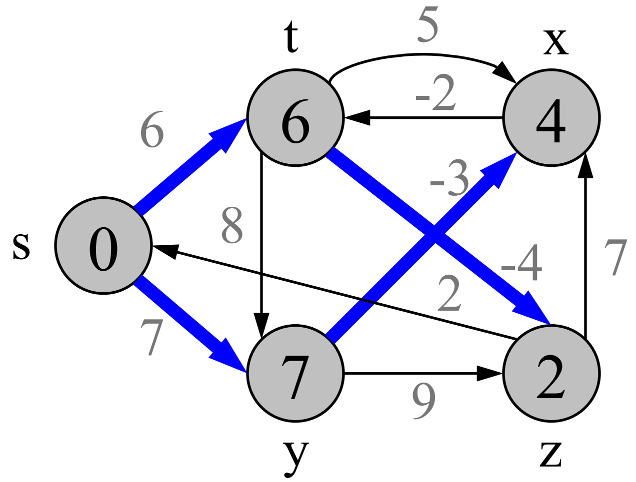
Iteration 0



Iteration 1



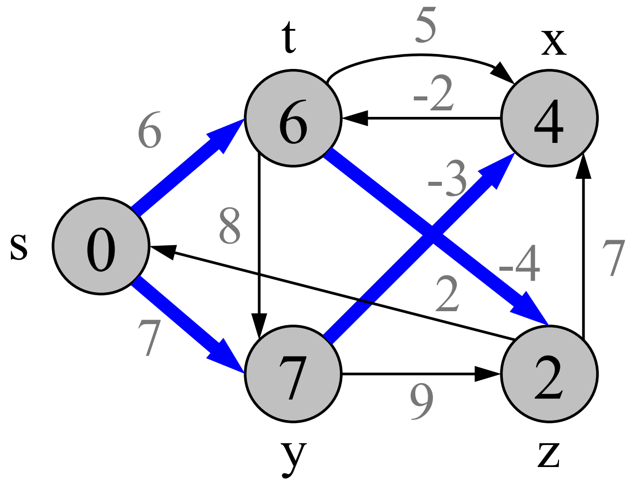
Iteration 2



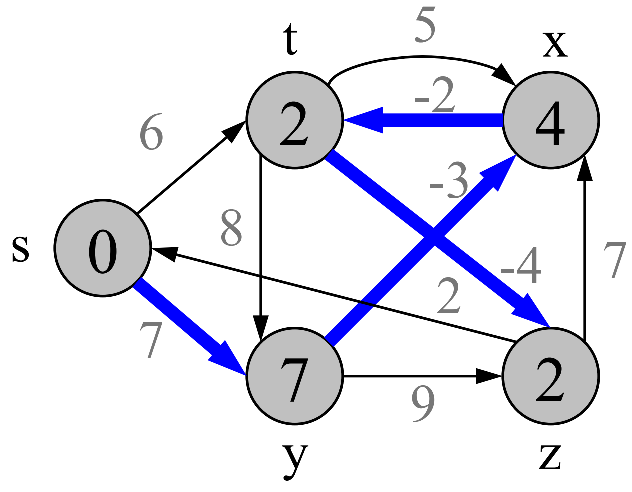

```
BELLMAN-FORD(G,w,s)
    INITIALIZE-SINGLE-SOURCE(G,s)
    for i = 1 to |G.V|-1
        for (u,v) in G.E
            RELAX(u,v,w)
    for (u,v) in G.E
        if v.d > u.d + w(u,v)
            return FALSE
    return TRUE
```

Relaxation sequence in each iteration: $(t, x), (t, y), (t, z), (x, t), (y, x), (y, z), (z, x), (z, s), (s, t), (s, y)$

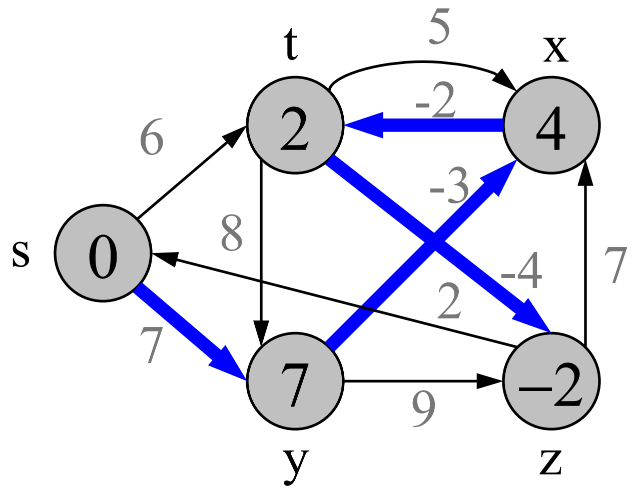
Iteration 2



Iteration 3



Iteration 4




```
BELLMAN-FORD(G,w,s)
    INITIALIZE-SINGLE-SOURCE(G,s)
    for i = 1 to |G.V|-1
        for (u,v) in G.E
            RELAX(u,v,w)
    for (u,v) in G.E
        if v.d > u.d + w(u,v)
            return FALSE
    return TRUE
```

Running time analysis

Using adjacency lists,

```
BELLMAN-FORD(G,w,s)
    INITIALIZE-SINGLE-SOURCE(G,s)
    for i = 1 to |G.V|-1
        for (u,v) in G.E
            RELAX(u,v,w)
    for (u,v) in G.E
        if v.d > u.d + w(u,v)
            return FALSE
    return TRUE
```

$\Theta(V)$

$\Theta((V-1)E)$

$\Theta(E)$

- Adjacency-list representation = $\Theta(VE)$

Q: Running time of adjacency-matrix representation = ?

It takes $\Theta(V^2)$ to loop through all edges, thus $\Theta(V^3)$ in total

Correctness of Bellman-Ford (Theorem 24.4)

We want to prove the following two statements:

1. Correctly compute $\delta(s, v)$ when no negative-weight cycle
   - After the $|V| - 1$ iterations of relaxation of all edges, it must hold that $v.d = \delta(s, v)$ for all vertices $v \in V$ that are reachable from s
   - For each vertex$v \in V$, there is a path from s to v if and only if the algorithm terminates with $v.d < \infty$.
2. Correctly detect the existence of negative cycles
   - Return FALSE If G does contain a negative-weight cycle reachable from s

Correctness of Bellman-Ford (Theorem 24.4)

1. Correctly compute $\delta(s,v)$ when no negative-weight cycle
   - After the $|V|-1$ iterations of relaxation of all edges, it must hold that $v.d=\delta(s,v)$ for all vertices $v \in V$ that are reachable from s

Proof

Although the shortest path p from s to v is unknown, we know it has at most $V-1$ edges if the path exists

- The relaxation sequence must contain all edges in p in order:

$$\underbrace{\underbrace{e_1,e_2,\dots,e_m}_{\text{Must contain 1st edge in } p};\ \underbrace{e_1,e_2,\dots,e_m}_{\text{Must contain 2nd edge in } p};\cdots\cdots;e_1,e_2,\dots,e_m}_{\text{Repeated } V-1 \text{ times, must contain all edges in } p \text{ in order}} \qquad (m=|E|)$$

- According to the path-relaxation property, $v.d=\delta(s,v)$ for all vertices $v \in V$ that are reachable from s

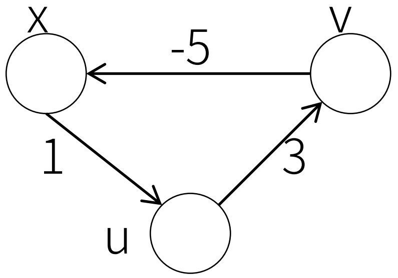
Correctness of Bellman-Ford (Theorem 24.4)

2. Correctly detect the existence of negative cycles
   - Return FALSE If G does contain a negative-weight cycle reachable from s

Proof by contradiction

- Suppose Bellman-Ford returns TRUE while G does contain a negative-weight cycle C reachable from s
- $\Rightarrow v.d \leq u.d + w(u,v), \forall (u,v) \in C$
- $\Rightarrow \sum_{v \in C} v.d \leq \sum_{v \in C} u.d + \sum_{(u,v) \in C} w(u,v)$
- $\Rightarrow 0 \leq \sum_{(u,v) \in C} w(u,v)$
- => contradiction

```
//negative cycle detection
    for (u,v) in G.E
        if v.d > u.d + w(u,v)
            return FALSE
```



Bellman-Ford algorithm: the DP view

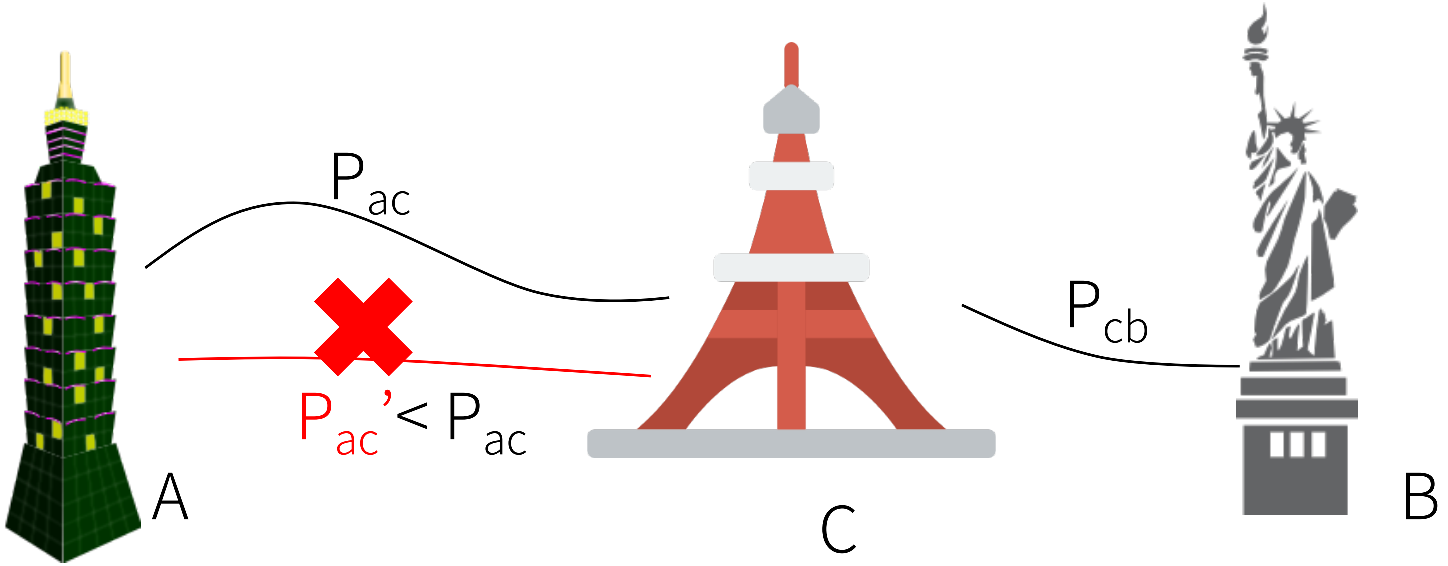
- Bellman-Ford is a dynamic programming algorithm
  - What are the subproblems?
  - Does it have optimal substructure?

Recap: 4 steps to dynamic programming

1. Characterize the structure of an optimal solution
   - Overlapping subproblems: revisits same subproblem repeatedly
   - Optimal substructure: an optimal solution to the problem contains within it optimal solutions to subproblems
2. Recursively define the value of an optimal solution
   - Express the original problem's solution using optimal solutions for smaller problems
3. Compute the value of an optimal solution (typically bottom-up)
4. Construct an optimal solution from computed information

Recap: Optimal substructure

Shortest path problem (最短路徑問題) has optimal substructure (Lemma 24.1)



Path P_{ac}+P_{cb} is a shortest path between A and B

$\Rightarrow$Then P_{ac} must be a shortest path between A and C

Bellman-Ford algorithm: the DP view

- Let $\ell_{sv}^{(k)}$ be the shortest path value from s to v using at most k edges
  - Subproblems: given s, $\ell_{sv}^{(k)}$ for all v, k
  - Optimal substructure: by Lemma 24.1
- Base cases: $\ell_{ss}^{(0)} = 0$; $\ell_{sv}^{(0)} = \infty$ when $s \neq v$
- The recurrence relation can be formulated as

$$\ell_{sv}^{(k)} = \min\left\{\ell_{sv}^{(k-1)}, \min_{u \in \mathrm{V}}\left\{\ell_{su}^{(k-1)} + w_{uv}\right\}\right\}$$

$$= \min_{u \in \mathrm{V}}\left\{\ell_{su}^{(k-1)} + w_{uv}\right\}$$

$$w_{ij} = \begin{cases} 0, & i = j \\ w(i,j), & i \neq j \text{ and } (i,j) \in E \\ \infty, & i \neq j \text{ and } (i,j) \notin E \end{cases}$$

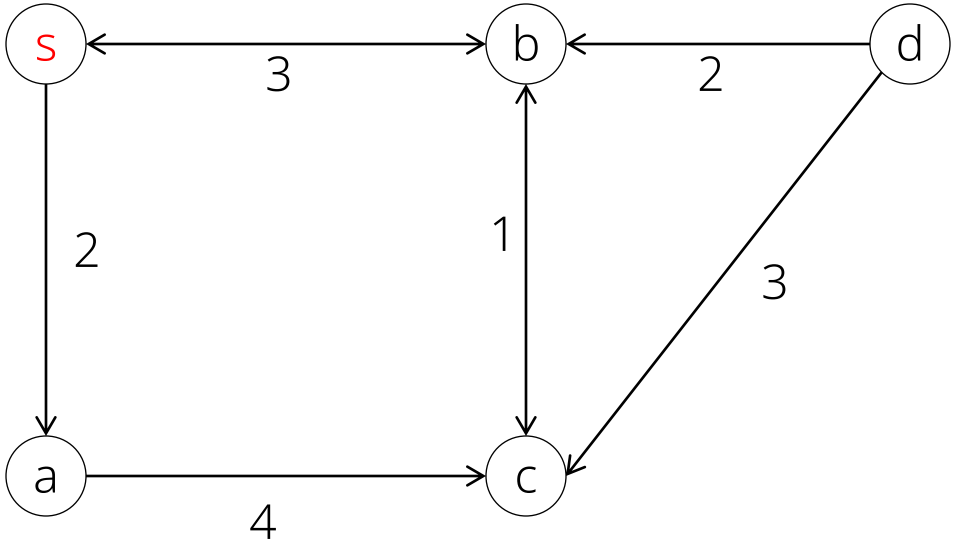
- Optimal values: $\ell_{sv}^{(|V|-1)}$ for all $v \in V$

Dijkstra’s algorithm

Textbook Chapter 24.3

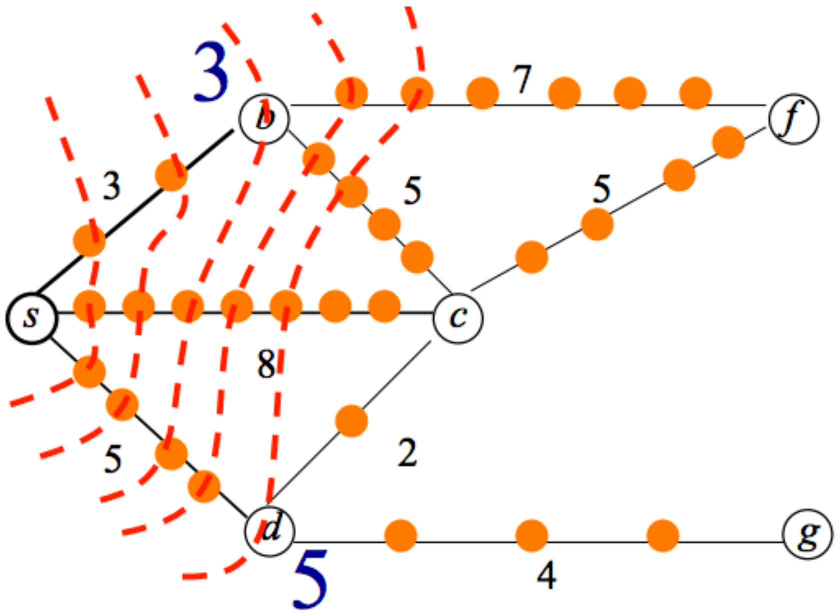
Dijkstra's algorithm: intuition

- Recall that BFS finds shortest paths on an unweighted graph by expanding the search frontier like ripples.
- Can we do the same on weighted graph?



Dijkstra's algorithm: intuition

- Recall that BFS finds shortest paths on an unweighted graph by expanding the search frontier like ripples.
- Can we do the same on weighted graph?



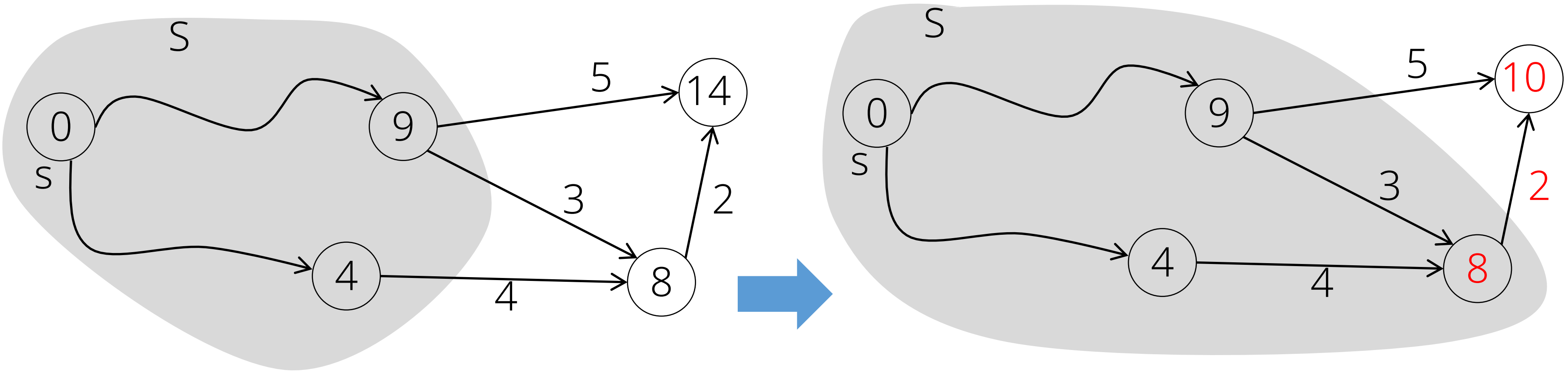
Dijkstra's algorithm speeds up the process by “skipping” layers that do not intersect with any vertex!

Source: http://courses.csail.mit.edu/6.006/spring11/lectures/lec16.pdf

Dijkstra's algorithm

Dijkstra greedily adds vertices by increasing distance

- Maintains a **set of explored vertices S** whose final shortest-path weights have already been determined
  1. Initially, $S = \{s\}, s.d = 0$
  2. At each step, select unexplored vertex u in $V - S$ with **minimum $u.d$**
  3. Add u to S, and **relaxes all edges leaving u**. Go back to Step 2.



Implementation of Dijkstra’s algorithm

```
DIJKSTRA(G,w,s)
    INITIALIZE-SINGLE-SOURCE(G,s)
    S= empty
    Q= G.v //BUILD-PRIORITY-QUEUE
    while Q ≠ empty
        u = EXTRACT-MIN(Q)
        S = S∪{u}
        for v in G.adj[u]
            RELAX(u,v,w)
```

```
INITIALIZE-SINGLE-SOURCE(G,s)
    for v in G.V
        v.d = ∞
        v.π = NIL
    s.d = 0
```

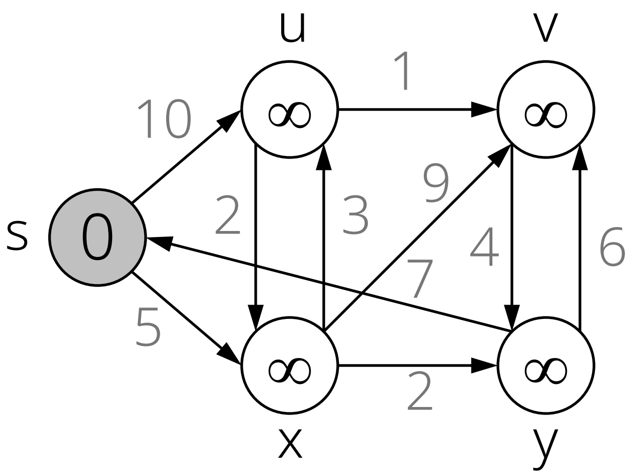
```
RELAX(u, v, w)
    if v.d > u.d + w(u, v)
        //DECREASE-KEY
        v.d = u.d + w(u, v)
        v.π = u
```

- Q is a min-priority queue of vertices, keyed by d values
- Observations (will prove these later)
  - For u in Q (that is, $V - S$), $u.d$ is the shortest-path estimate (i.e., minimum length over all observed $s \rightsquigarrow u$ paths so far).
  - For u in S, $u.d = \delta(s, v)$

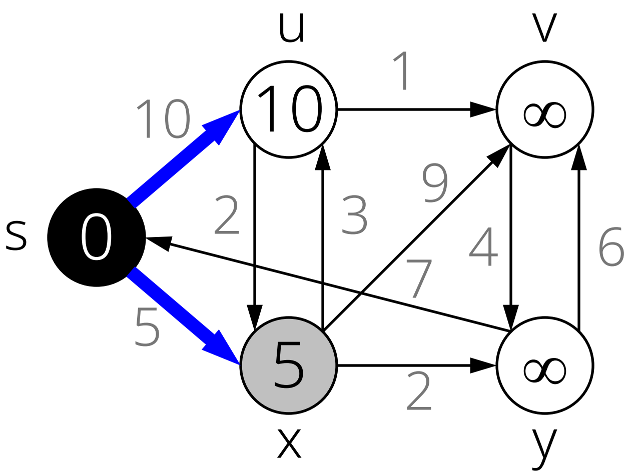
```
DIJKSTRA(G,w,s)
    INITIALIZE-SINGLE-SOURCE(G,s)
    S= empty
    Q= G.v //BUILD-PRIORITY-QUEUE
    while Q ≠ empty
        u = EXTRACT-MIN(Q)
        S = S∪{u}
        for v in G.adj[u]
            RELAX(u,v,w)
```

Black: in S
White: in Q
Grey: selected
Blue lines: predecessors

Step 0

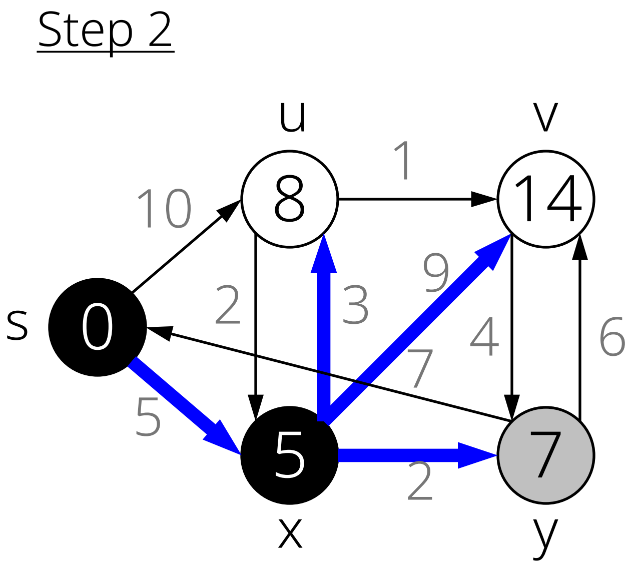


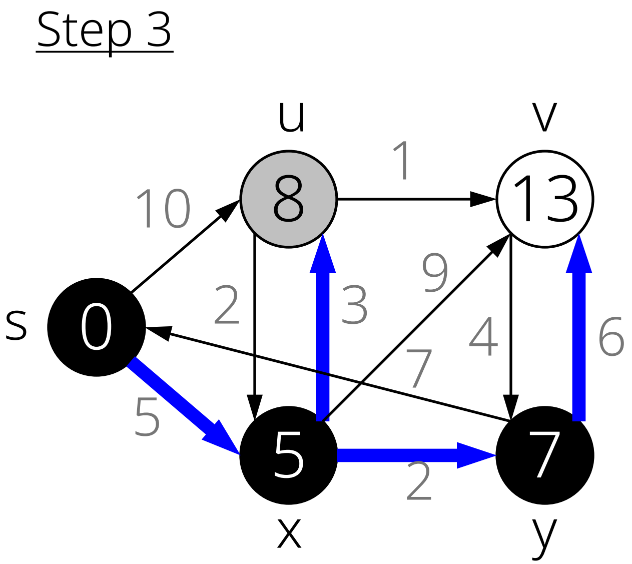
Step 1



```
DIJKSTRA(G,w,s)
    INITIALIZE-SINGLE-SOURCE(G,s)
    S= empty
    Q= G.v //BUILD-PRIORITY-QUEUE
    while Q ≠ empty
        u = EXTRACT-MIN(Q)
        S = S∪{u}
        for v in G.adj[u]
            RELAX(u,v,w)
```

Black: in S
White: in Q
Grey: selected
Blue lines: predecessors

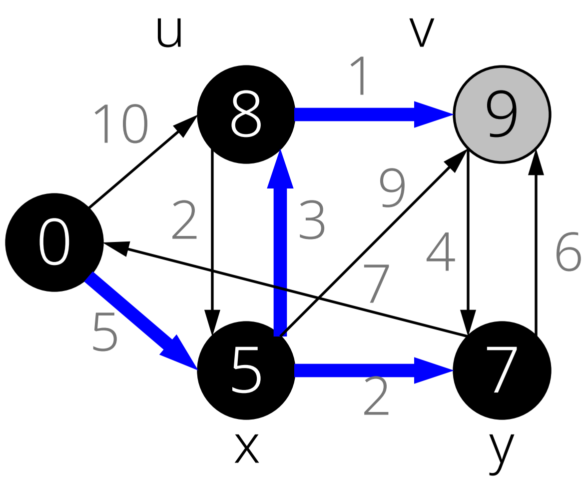




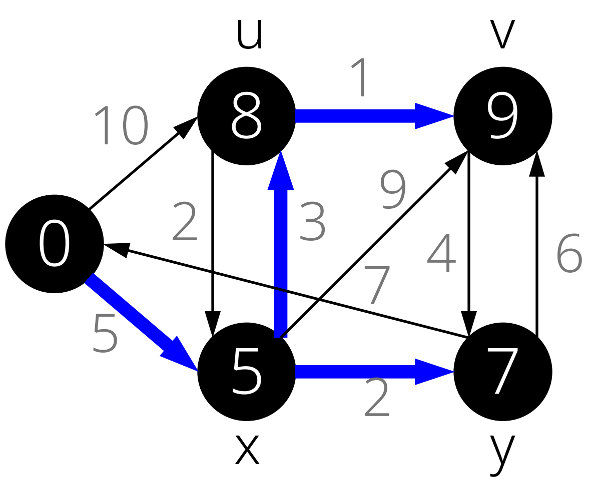
```
DIJKSTRA(G,w,s)
    INITIALIZE-SINGLE-SOURCE(G,s)
    S= empty
    Q= G.v //BUILD-PRIORITY-QUEUE
    while Q ≠ empty
        u = EXTRACT-MIN(Q)
        S = S∪{u}
        for v in G.adj[u]
            RELAX(u,v,w)
```

Black: in S
White: in Q
Grey: selected
Blue lines: predecessors

Step 4



Step 5



Running time analysis

- Q is a min-priority queue of vertices, keyed by d values
  - # of INSERT = $O(V)$
  - # of EXTRACT-MIN = $O(V)$
  - # of DECREASE-KEY = $O(E)$
- The running time depends on queue implementation
- Implementing the min-priority queue using an array indexed by v: $O(V^2 + E) = O(V^2)$
  - INSERT: $O(1)$
  - EXTRACT-MIN: $O(V)$
  - DECREASE-KEY: $O(1)$

Correctness of Dijkstra’s algorithm (Theorem 24.6)

Dijkstra’s algorithm, run on a weighted, directed graph $G = (V, E)$ with non-negative weight function w and source s, terminates with $u.d = \delta(s, u)$ for all vertices $u \in V$.

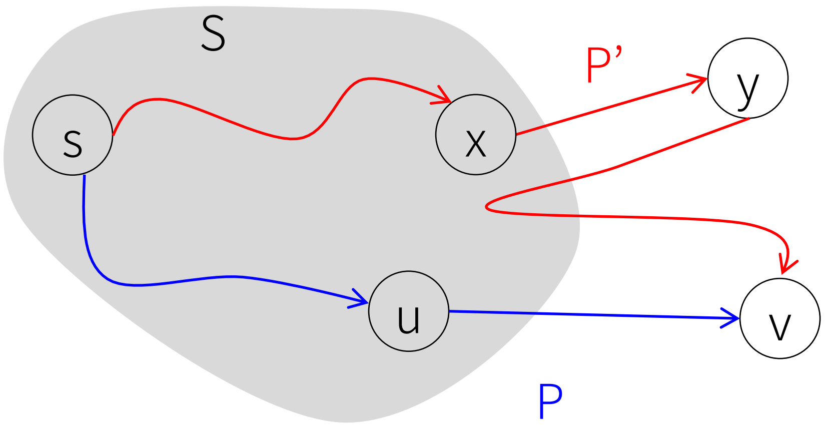
Idea

- S: the set of explored vertices whose final shortest-path weights have already been determined
  - Initially, $S = \{s\}, s.d = 0$
  - **Invariant:** for all u in S, $u.d$ = length of the shortest path from s to u
  - Note that for u in $V - S$, $u.d$ = length of *some* path from s to u
- We want to prove that the loop invariant holds throughout the execution of the algorithm.

Loop invariant: for u in S, $u.d = \delta(s,u)$

Proof by induction on the size of S

- Base case: $|S| = 1$, correct
- Inductive step: Let v be the next vertex to be added to S, $u = v.\pi$, P = shortest path from s to $u + (u, v)$
- $\Rightarrow v.d = w(P) = \delta(s,u) + w(u,v)$
- Consider any other $s \rightsquigarrow v$ path P', and Let y be the first vertex on path P' outside S
- We want to prove that $w(P') \geq w(P)$



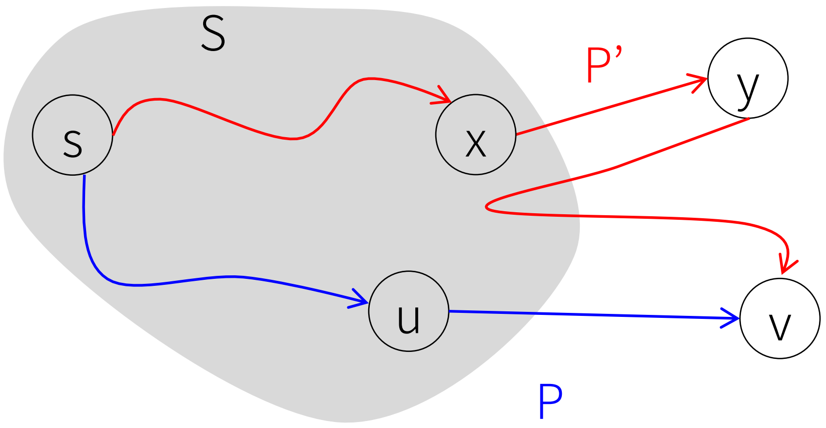
Loop invariant: for u in S, $u.d = \delta(s, u)$

Proof by induction on the size of S (cont'd)

- Prove that $w(P') \geq w(P)$

1. Because of no negative edges, $w(P') \geq \delta(s, x) + w(x, y)$
2. By induction hypothesis, $\delta(s, x) = x.d$
3. By construction, $y.d \geq v.d$
4. By construction, $x.d + w(x, y) \geq y.d$

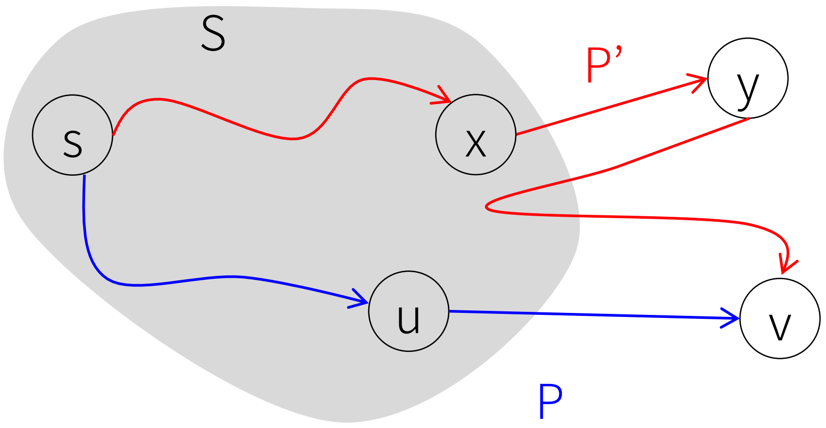
- $\Rightarrow$ $w(P') \geq \delta(s, x) + w(x, y) = x.d + w(x, y) \geq y.d \geq v.d = w(P)$



Loop invariant: for u in S, $u.d = \delta(s, u)$

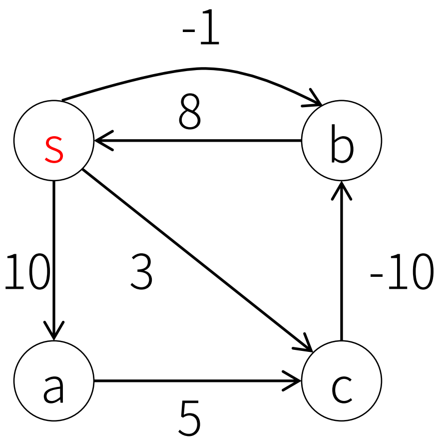
Proof by induction on the size of S (cont'd)

- Hence, the greedy choice v (and the corresponding path P) is at least as good as any other path from s to v
- => The invariant still holds after adding one more vertex v to S
- At termination, every vertex is in S
- Thus, $u.d = \delta(s, v)$ for all u in V

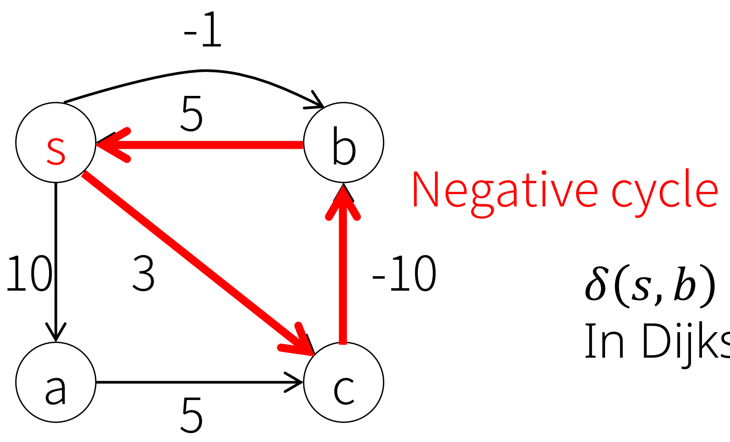


Dijkstra's algorithm may work incorrectly with negative-weight edges

- Let's go back to the proof and see where it breaks!
  - This greedy algorithm assumed adding edges always increases path weight, which is not true in case of negative-weight edges
- C.f. Bellman-Ford: a dynamic programming algorithm either detects negative cycles or returns the shortest-path tree



$\delta(s,b) = -7$
In Dijkstra, $b.d = -1$



$\delta(s,b) = ?$
In Dijkstra, $b.d = ?$

Q: See any similarity between BFS, DFS, Prim and Dijkstra?

- They are all greedy algorithms for graph search
- They are each a special case of priority-first search

Priority-first search

- Maintain a set of explored vertices S
- Grow S by exploring highest-priority edges with exactly one endpoint leaving S

Q: What's the priority in each variant (BFS, DFS, Prim and Dijkstra)?

BFS: edges from vertex discovered least recently

DFS: edges from vertex discovered most recently

Prim: edges of minimum weight

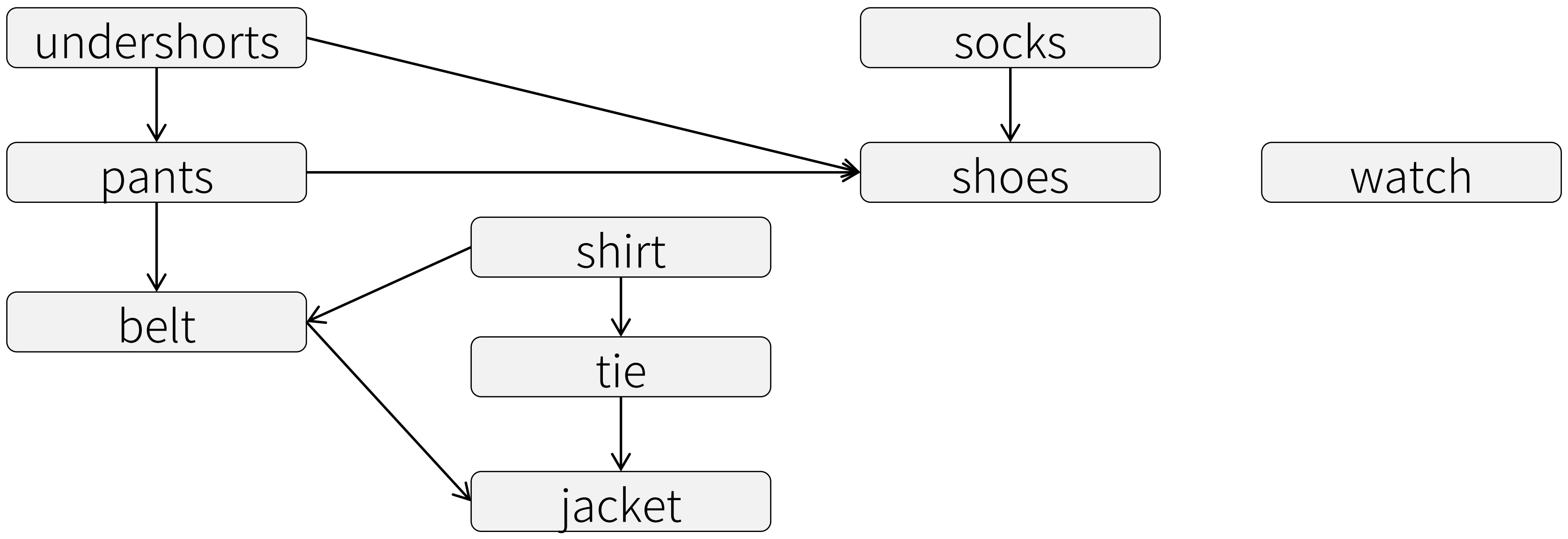
Dijkstra: edges to vertex closest to s

Single-source shortest paths in directed acyclic graphs

Textbook Chapter 24.2

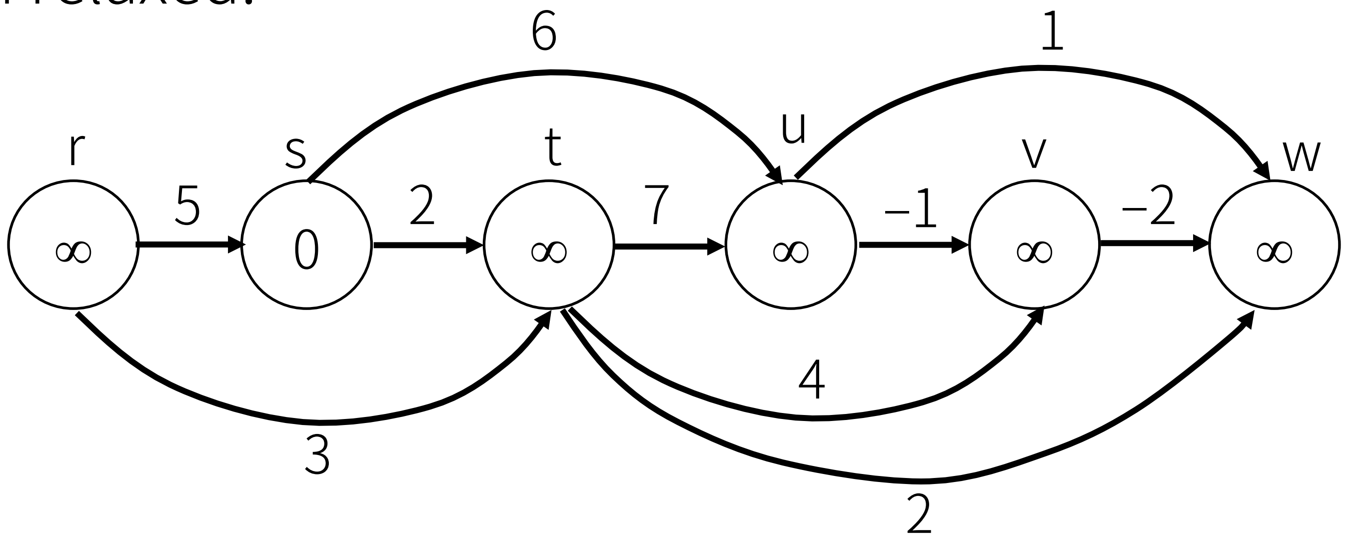
Recap: Directed Acyclic Graphs (DAGs)

- A DAG is a directed graph with no cycles
- Often used to indicate precedence among events (X must happen before Y)
  - E.g., cooking, taking courses, clothing⋯

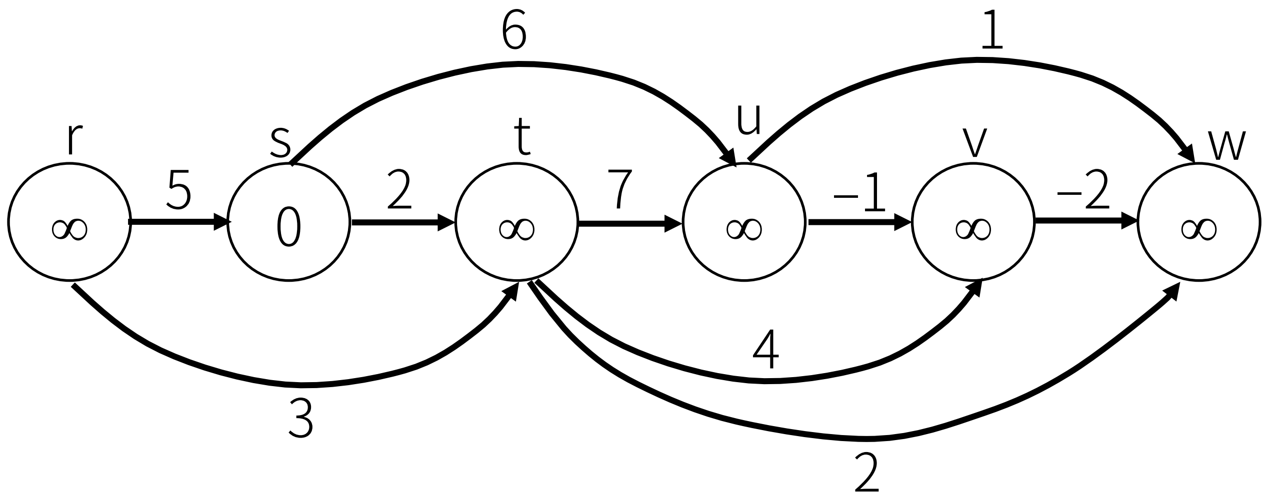


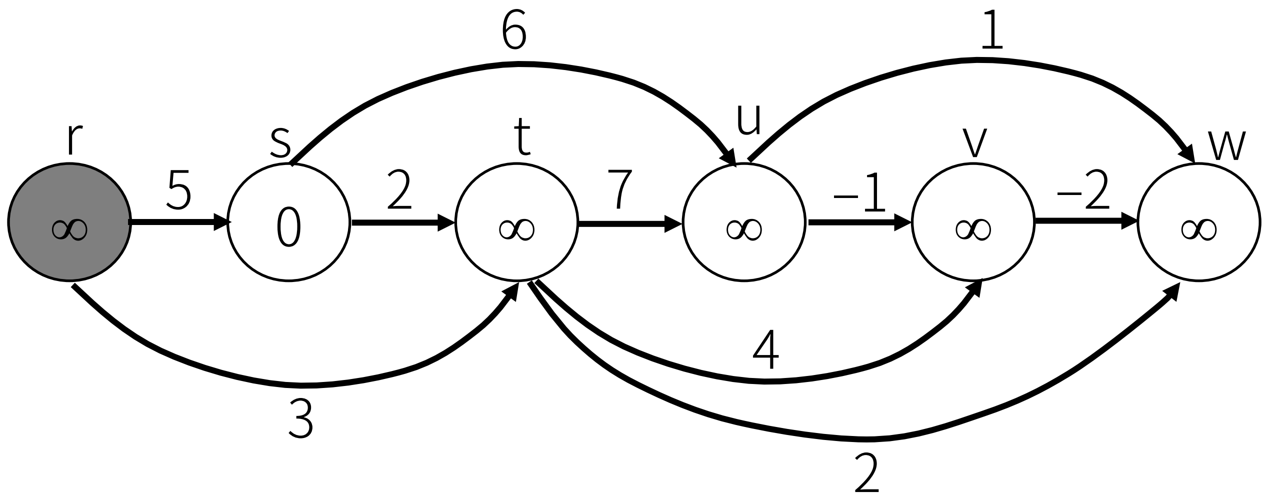
Single-source shortest paths in DAG

- Claim: relaxing the edges in topologically sorted order correctly computes the shortest paths in DAG
- Intuition: putting vertices in a topologically sorted order, edges only go from left to right; so when relaxing an edge (u, v), all edges to u must have been relaxed.

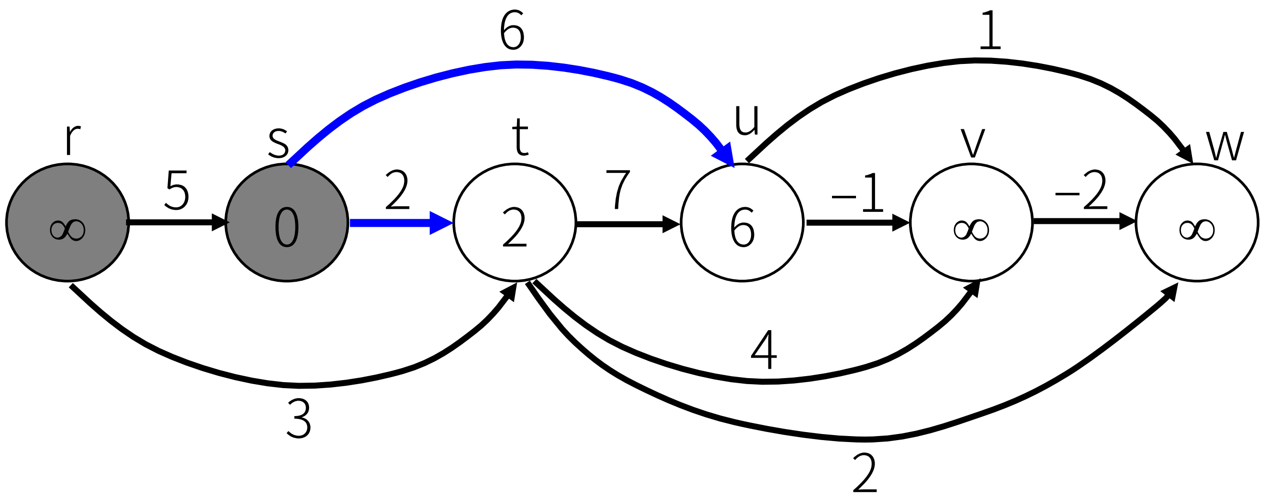


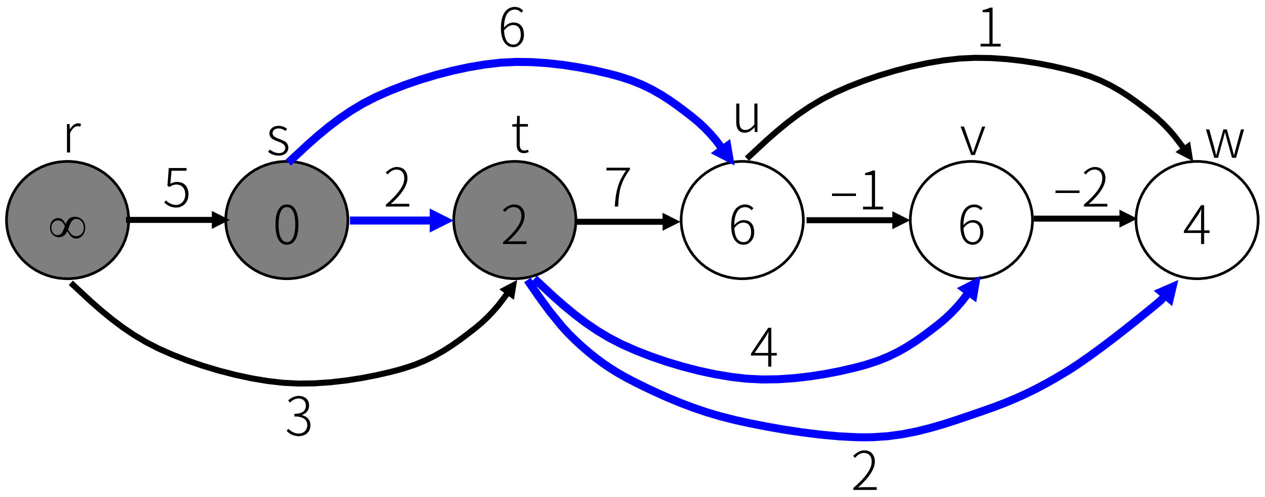
```
DAG-SHORTEST-PATHS(G,w,s)
    topologically sort the vertices of G
    INITIALIZE-SINGLE-SOURCE(G,s)
    for each vertex u, taken in topologically sorted order
        for each vertex v in G.adj[u]
            RELAX(u,v,w)
```



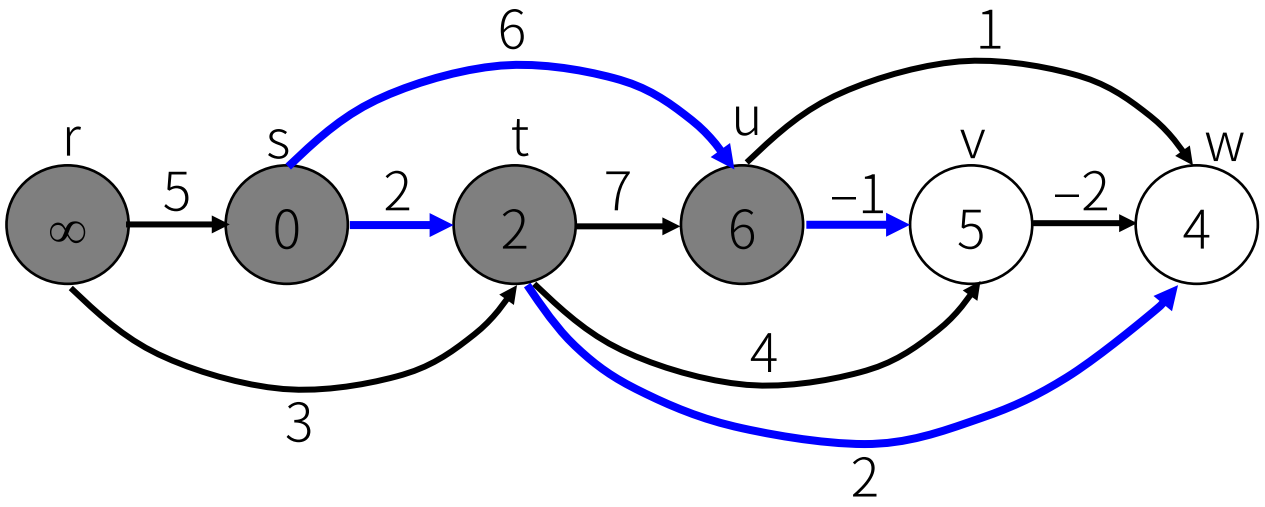


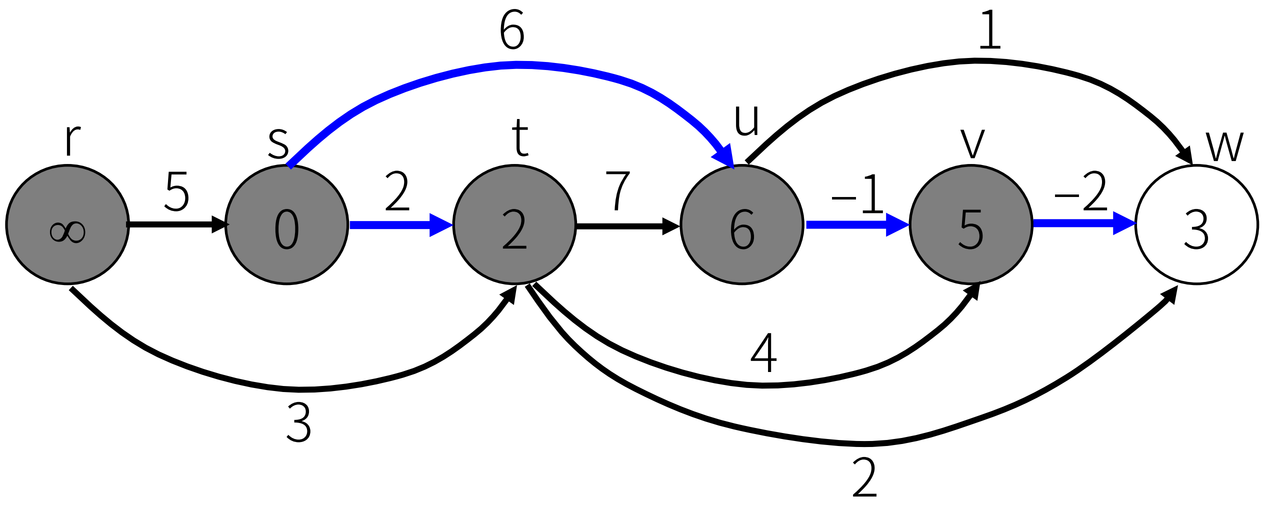
```
DAG-SHORTEST-PATHS(G,w,s)
    topologically sort the vertices of G
    INITIALIZE-SINGLE-SOURCE(G,s)
    for each vertex u, taken in topologically sorted order
        for each vertex v in G.adj[u]
            RELAX(u,v,w)
```





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

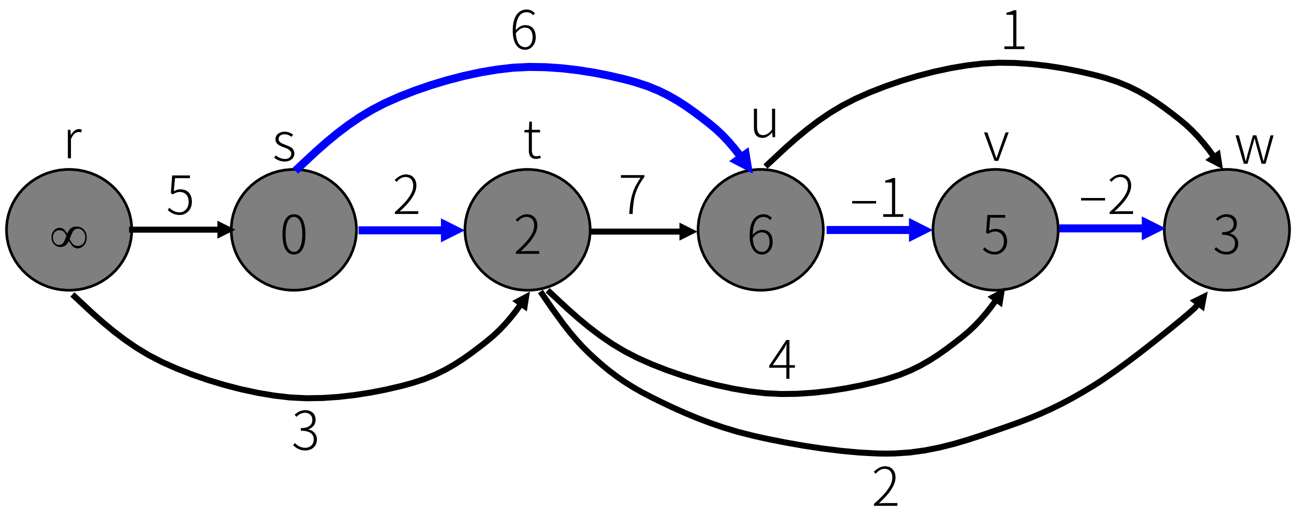




Running time analysis

```
DAG-SHORTEST-PATHS(G,w,s)
    topologically sort the vertices of G  // Θ(V+E)
    INITIALIZE-SINGLE-SOURCE(G,s)  // Θ(V)
    for each vertex u, taken in topologically sorted order  ┐
        for each vertex v in G.adj[u]                        ├ Θ(V+E)
            RELAX(u,v,w)                                     ┘
```

=> total running time is $\Theta(V + E)$, same as topological sort



Theorem 24.5

If $G = (V, E)$ is a DAG, then at the termination of DAG-SHORTEST-PATHS, $v.d = \delta(s, v)$, for all $v \in V$

Proof by induction on the position in topological sort order

- Inductive hypothesis: if all the vertices before v in a topological sort order have been updated, then $v.d = \delta(s, v)$
- Base case:
  - For all v before s, $v.d = \infty = \delta(s, v)$
  - For s, $s.d = 0 = \delta(s, s)$

Theorem 24.5

If $G = (V, E)$ is a DAG, then at the termination of DAG-SHORTEST-PATHS, $v.d = \delta(s, v)$, for all $v \in V$

Proof by induction on the position in topological sort order (Cont.)

- Inductive hypothesis: if all the vertices before v in a topological sort order have been updated, then $v.d = \delta(s, v)$
- Inductive step:
  - Consider a vertex v after s
  - By construction, $v.d = \min_{(u,v) \in E} (u.d + w(u, v))$
  - By inductive hypothesis, $u.d + w(u, v) = \delta(s, u) + w(u, v)$
  - Since some (u, v) must be on the shortest path, by optimal substructure, $v.d = \delta(s, v)$

Summary of single-source shortest-path algorithms

SSSP algorithm	Applicable graph types	Running time
Dijkstra	Nonnegative weights	$\Theta(V^2)$ (array-based)
Topological sort based	DAG	$\Theta(V + E)$
Bellman-Ford	generic	$\Theta(EV)$