Lecture 10 – Graph Algorithms III

AIAA 5037 Advanced Algorithms and Data Structures

Ying Sun, AI Thrust

Outline

- Strongly Connected Component
- Single-Source Shortest Path
  - Dynamic Programming for DAG
  - Bellman-Ford Algorithm
  - Dijkstra's Algorithm
  - Application: Difference Constraints Problem
- All-Pairs Shortest Path
  - Matrix-View Solution
  - Floyd-Warshall Algorithm

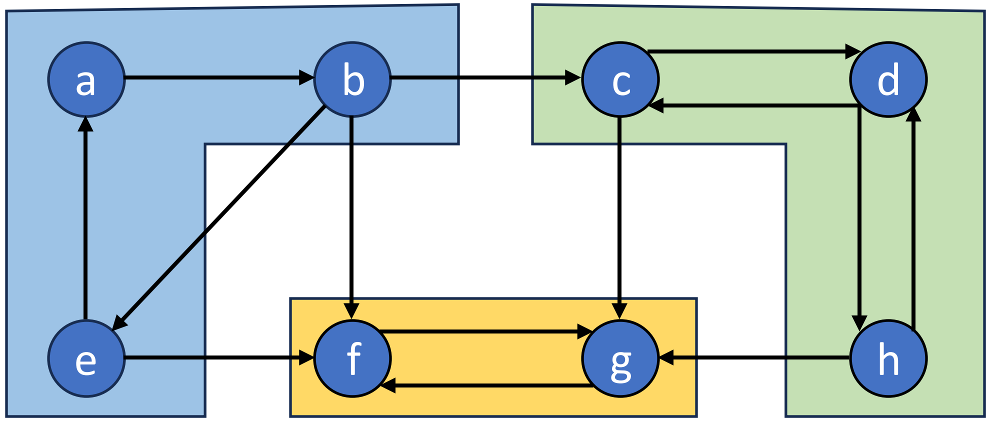
Strongly Connected Components

Strongly Connected Components

Problem: Given a directed graph $G = (V, E)$, find the *strongly connected components.*

Definitions:

- *Strongly connected subgraph:* every vertex is reachable from every other vertex
- *Strongly connected component*: a maximized (极大) strongly connected subgraph
  - Cannot add more nodes to the subgraph and make it still strongly connected
  - Forms a partition of G into strongly connected subgraphs



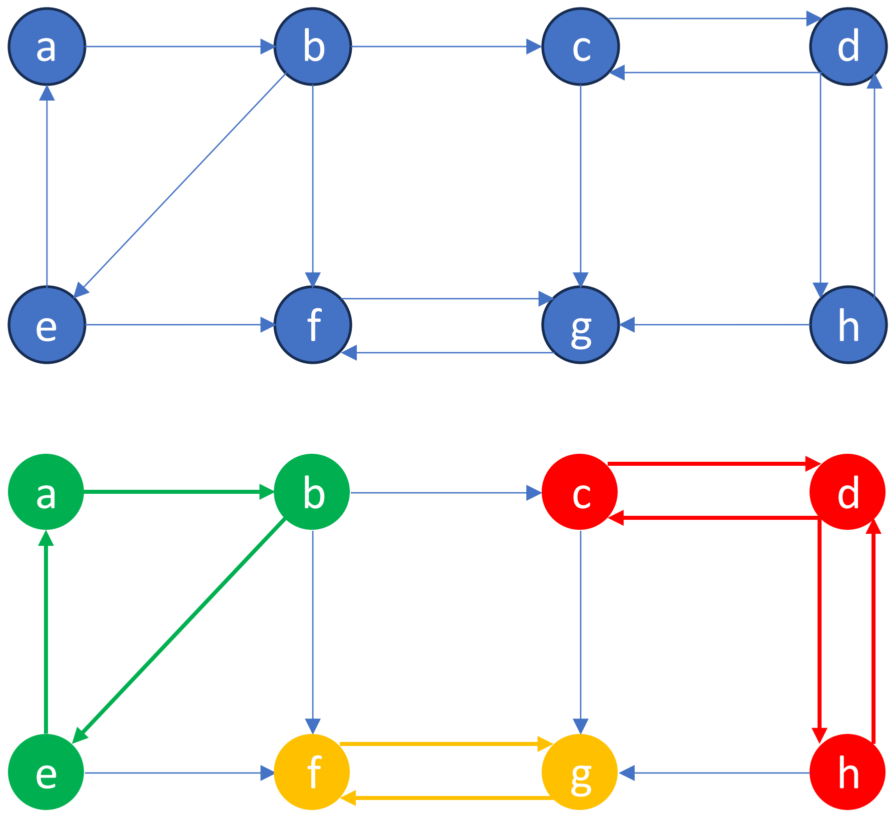
Strongly Connected Components

Problem: given a directed graph $G = (V, E)$, find the *strongly connected components.*

Directly adopt DFS?

- *DFS(g)*
- ~~*DFS(f)*~~
- *DFS(h)*
- *DFS(a)*
- ~~*DFS(d)*~~
- ~~*DFS(c)*~~
- ~~*DFS(b)*~~
- ~~*DFS(e)*~~

Seems correct, always correct?



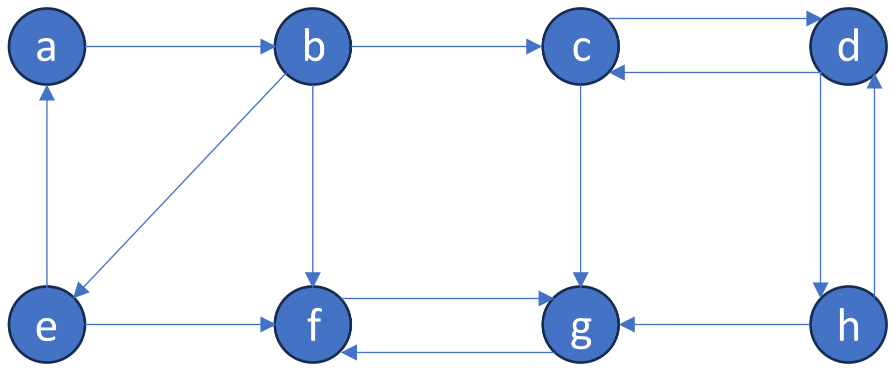
Strongly Connected Components

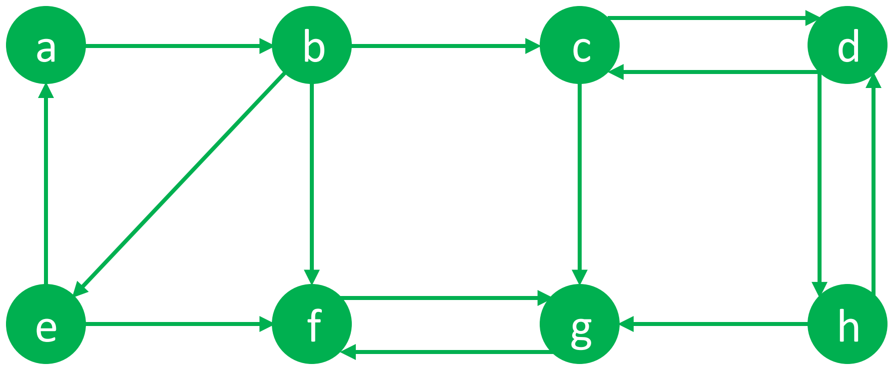
Problem: given a directed graph $G = (V, E)$, find the *strongly connected components.*

Another order of DFS

- *DFS(a)*
- ~~*DFS(f)*~~
- ~~*DFS(h)*~~
- ~~*DFS(a)*~~
- ~~*DFS(d)*~~
- ~~*DFS(c)*~~
- ~~*DFS(b)*~~
- ~~*DFS(e)*~~

The order matters!





Strongly Connected Components

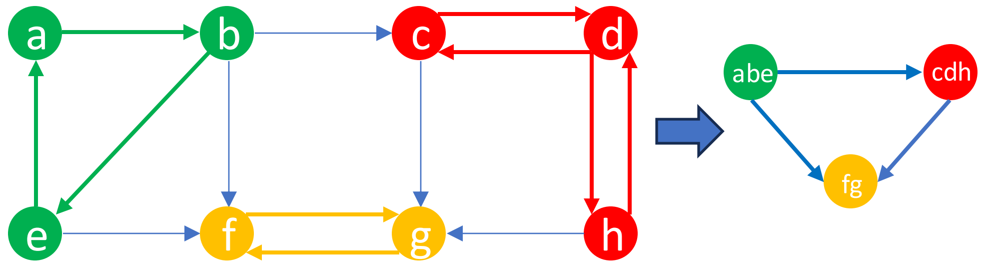
Observe each components as a whole and decide the order

Definition (component graph): Given a G with strongly connected components $C_1, C_2, \dots, C_k$, its component graph $G^{SCC} = (V^{SCC}, E^{SCC})$ represents the connection between components. Specifically, $V^{SCC} = \{v_1, v_2, \dots, v_k\}$ and v_i represents C_i. There is an edge $(v_i, v_j) \in E^{SCC}$ if there is $< x \epsilon C_i, y \epsilon C_j > \in E$

Key property: a component graph is a Directed Acyclic Graph

- Proof: obviously, a cycle will make all nodes in the two components reach each other. These two components are no longer strongly connected maximized subgraphs

Intuition: Decide the order of DFS with topological sort on the component graph



Strongly Connected Components

Notation (for a set of vertices $U \subseteq V$): $d(U) = \min_{u \in U} u.d$ and $f(U) = \max_{u \in U} u.f$.

Lemma: Let C_i and C_j be distinct strongly connected components in directed graph $G = (V, E)$. If there is an edge $< u \in C_i, v \in C_j > \in E$. Then $f(C_i) > f(C_j)$.

Proof:

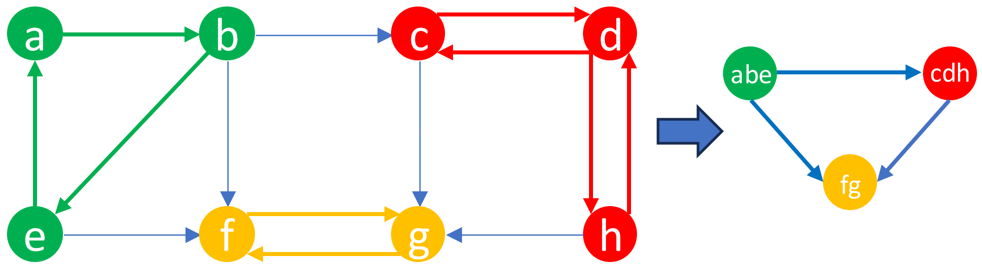
- If $d(C_i) < d(C_j)$:C_i is first discovered in dfs, meaning nodes in C_j are descendants of a node in C_i (denoted as x, possibly u). Since descendants finish earlier than ancestors, $f\left(C_j\right) = \max_{y \in C_i} y.f < x.f \leq f(C_i)$.
- If $d(C_i) > d(C_j)$: since C_i and C_j are distinct strongly connected components, with edge $< u, v >$ pointing from C_i to C_j, there cannot be a path pointing from C_j to C_i. Therefore, nodes in C_i are not reachable during DFS in C_j. i.e., C_j finishes before C_i is discovered. $f\left(C_j\right) < d(C_i) < f(C_i)$

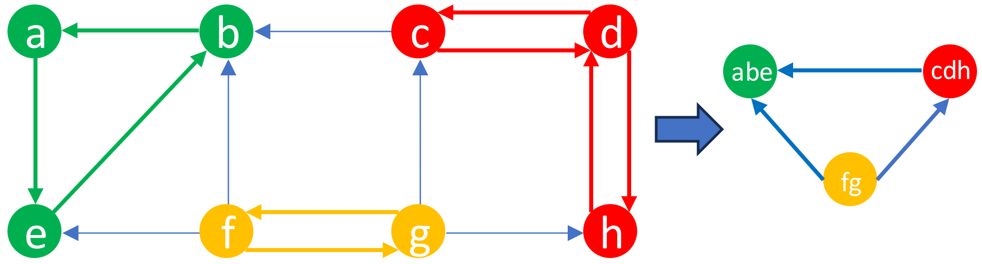
Sort components in the finish time decreasing order produces a topological sort

Strongly Connected Components

Sort components in the finish time decreasing order to produce a topological sort

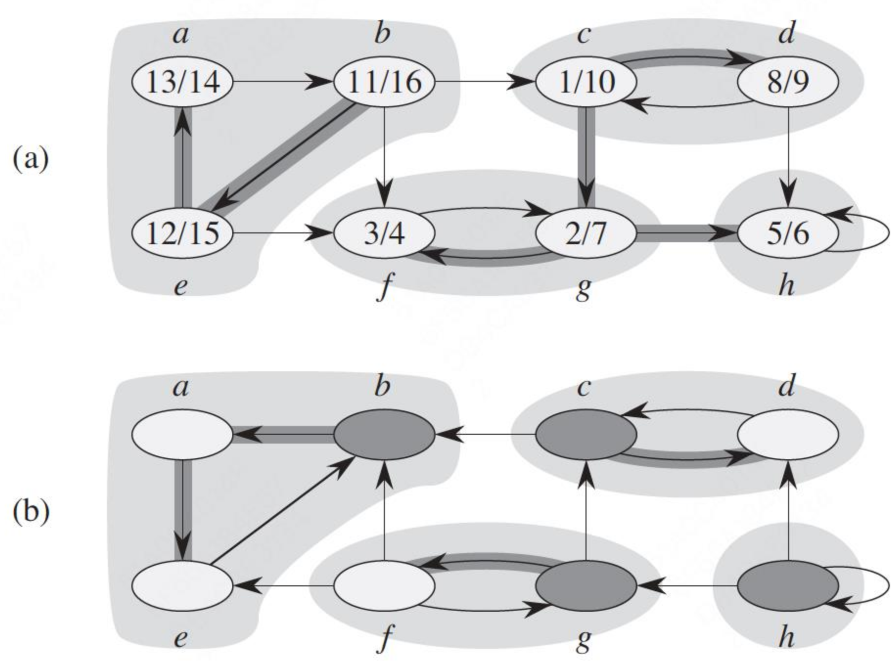
- Problem: composition nodes of each component are unknown to us
- Observation: the last finished node belongs to the last finished component
    - Find and delete the nodes in this component.
    - The next last finished nodes help finding the second last finished component
- Still a problem: the last ended component can reach other components
- Solution: reverse all the edges!



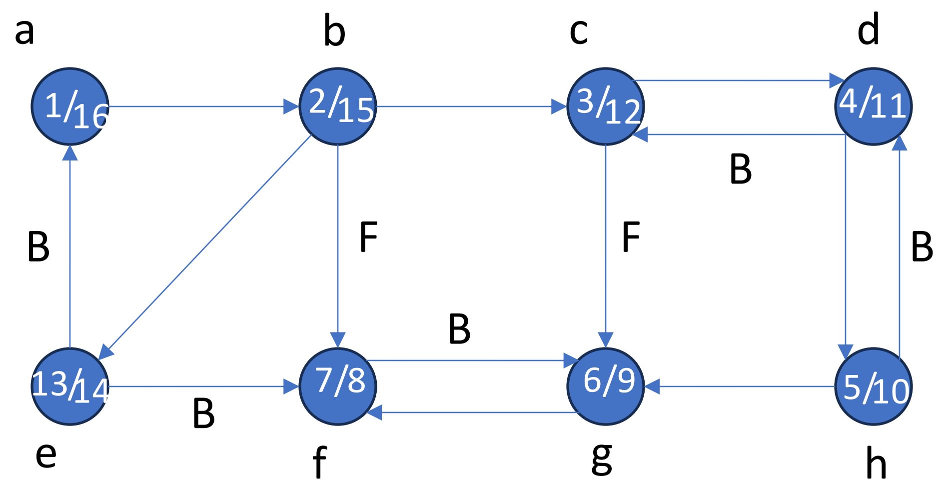


Kosaraju Algorithm

1. DFS in any order to find the finishing time of all the nodes
2. Transpose the graph
3. DFS in finishing time decreasing order



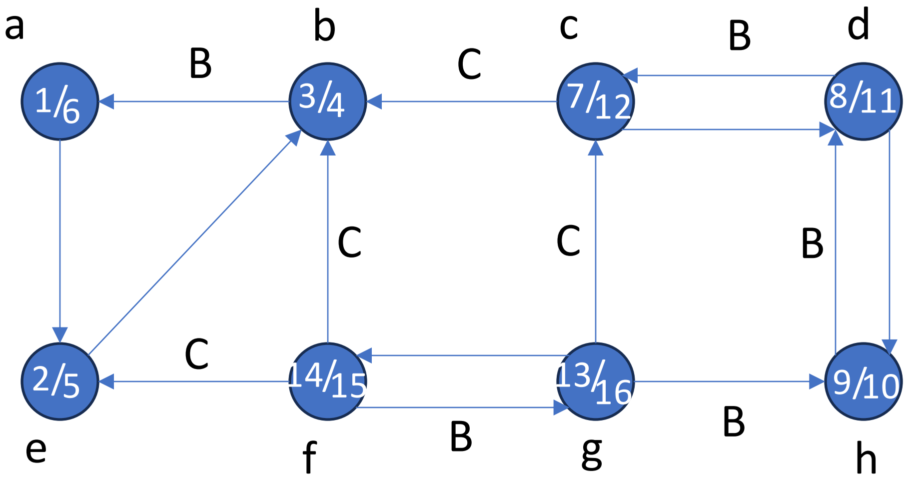
Kosaraju Algorithm: Example



Finishing time decreasing order: a b e c d h g f

Kosaraju Algorithm: Example

Transpose the graph



Finishing time decreasing order: a b e c d h g f

Component 1: a b e

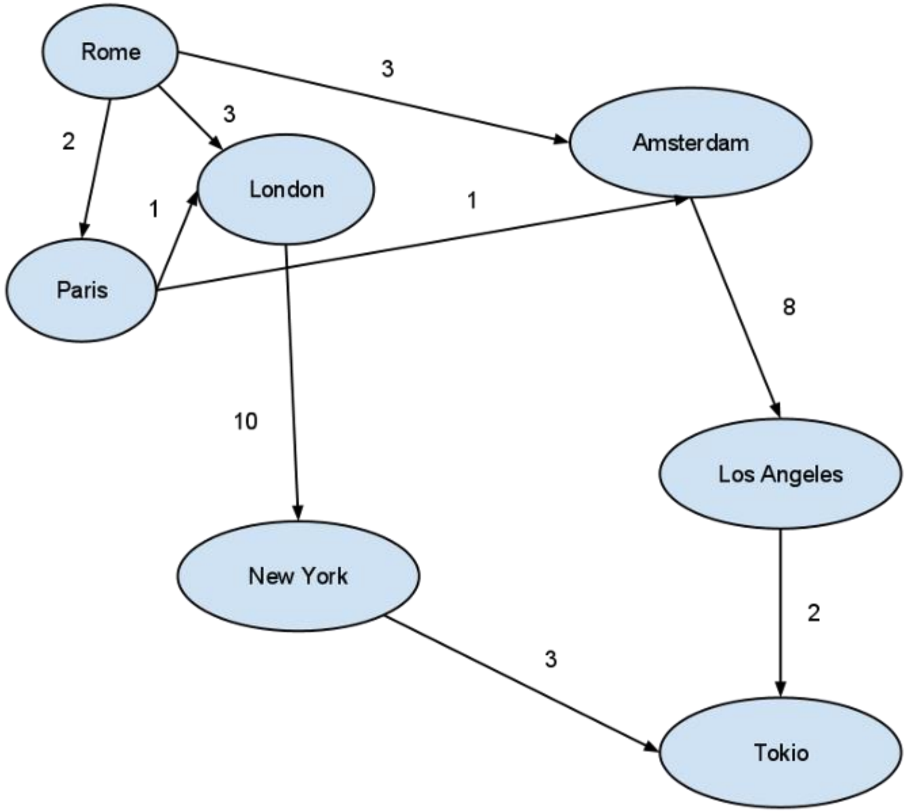
Component 2: c d h

Component 3: g h

Single-Sourced Shortest-Path Problem

Single-Sourced Shortest-Path Problem

Problem: given a map representing N cities and M roads connecting these cities. Each road has a specific travel time, represented as $< o_i, d_i, w_i >$. Determine the shortest travel time from the capital city s to each of the other cities.



Single-Sourced Shortest-Path Problem

Formal definition: Given a weighted, directed graph G=<V,E>, each edge associated with a weight, find a shortest path from a given source vertex $s \in V$ to each vertex $v \in V$

- Find shortest path on undirected graph: regard each undirected edge as two directed edges

Notations:

- A path: $p =< e_{p_1}, e_{p_2}, \ldots, e_{p_k} >$ with $e_{p_{k-1}}.end = e_{p_k}.start$
  - Distance $w(p) = \sum_{i=1}^{k} e_{p_i}.w$
- Shortest path from u to v: any path p linking u and v with the smallest distance
- Shortest distance from u to v

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

Representing Shortest Paths

- *Predecessor*: for each vertex $v \in V$, picking arbitrary shortest path from s to v, record the adjacent node before v as its predecessor $v.\pi$ (can be NIL)
- *Predecessor subgraph* $G =< V_{\pi}, E_{\pi} >$
  - $V_{\pi} = \{\mathrm{v} \in V: v.\pi \neq \mathrm{NIL}\} \cup \{s\}$
  - $E_{\pi} = \{(\mathrm{v}.\pi, v) \in V: v.\pi \neq \mathrm{NIL}\} \cup \{s\}$
- *Shortest-path tree:*
  - V' ∈ V is the set of vertices reachable from s in G
  - For all $v \in V'$, the simple path from s to v in G' is a shortest path from s to v in G

Lemma: predecessor subgraph forms shortest-path tree

- $V_{\pi} = V'$ & |V'|-1 edges: tree structure with the same nodes
- You can prove mathematical induction the path from s to any vertex v is a shortest path

Shortest-Path Problem

Optimal Substructure: given a shortest path $p =< e_{p_1}, e_{p_2}, \ldots, e_{p_k} >$ linking s and v, $p_{1:k-1} = < e_{p_1}, e_{p_2}, \ldots, e_{p_{k-1}} >$ is a shortest path between s and $e_{p_{k-1}}.end$

- (subpaths of shortest paths are shortest paths)
- Proof: contradiction

Recurrence: $\delta(s, v) = \min_{e \in E, e.end = v} \{\delta(s, e.start) + e.w\}$

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

Shortest-Path Problem

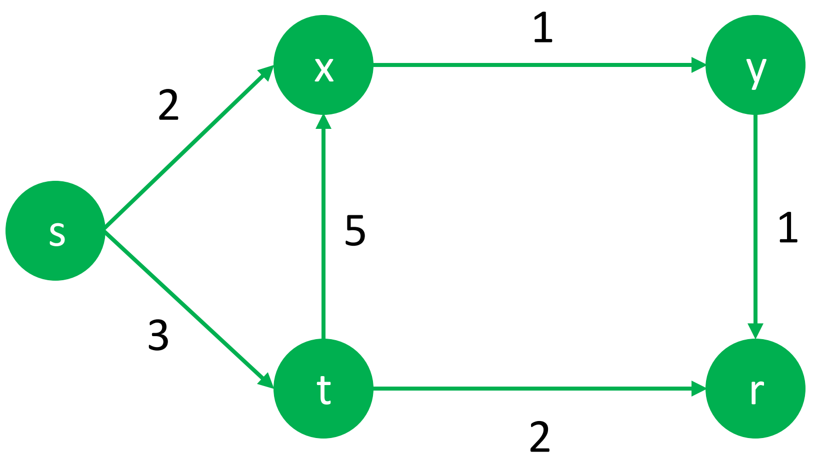
- $\delta(s, v) = \min_{e \in E, e.end = v} \{\delta(s, e.start) + e.w\}$

Top-Down with memorization

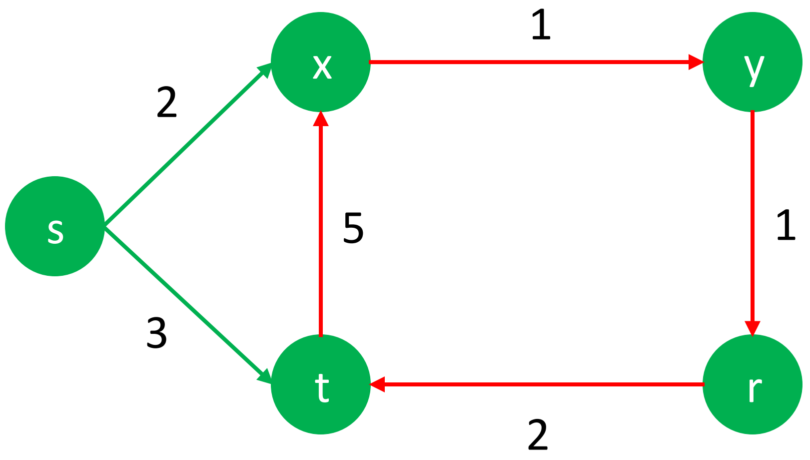
```
vis[N] = False
f(G, u, s)
if vis[u] is True:
    return dp[u]
if u == s:
    dp[u] = 0
for each vertex e ∈ G.in_edges[u]
    dp[u] = min(dp[u], f(G, e.start, s) + e.w)
vis[u] = True
return dp[u]
```

Problem?

G itself is the subproblem graph



Cycled dependencies

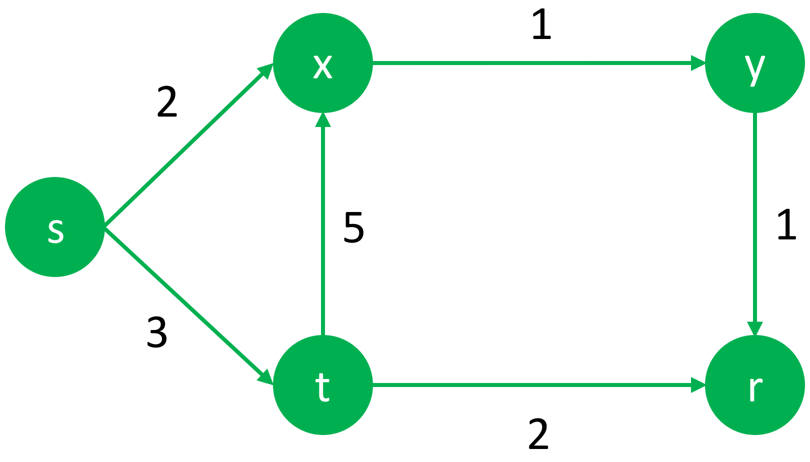


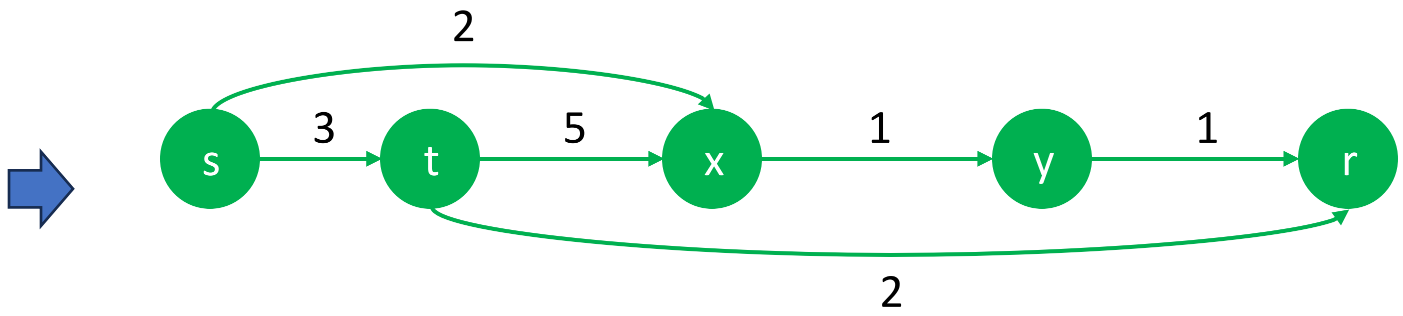
Single-Source Shortest Paths in DAG

Single-Source Shortest Paths in Directed Acyclic Graphs

Start from simple scenario: Directed Acyclic Graphs

- Dependences of subproblems are single directed and no cycled dependencies






```
vis[N] = False
f(G, u, s)
if vis[u] is True:
    return dp[u]
if u == s:
    dp[u] = 0
for each vertex e ∈ G.in_edges[u]
    dp[u] = min(dp[u], f(e.start) + e.w)
vis[u] = True
return dp[u]
```

Time Complexity:

- Nodes: $|V|$, each solved once
- Edges: $|E|$ edges, each retrieved once
- Total: $O(|V| + |E|)$

Single-Source Shortest Paths in Directed Acyclic Graphs

$$\delta(s, v) = \min_{e \in E, e.end = v} \{\delta(s, e.start) + e.w\}$$

Bottom-Up?

- Define the "size" of subproblem in terms of topological order

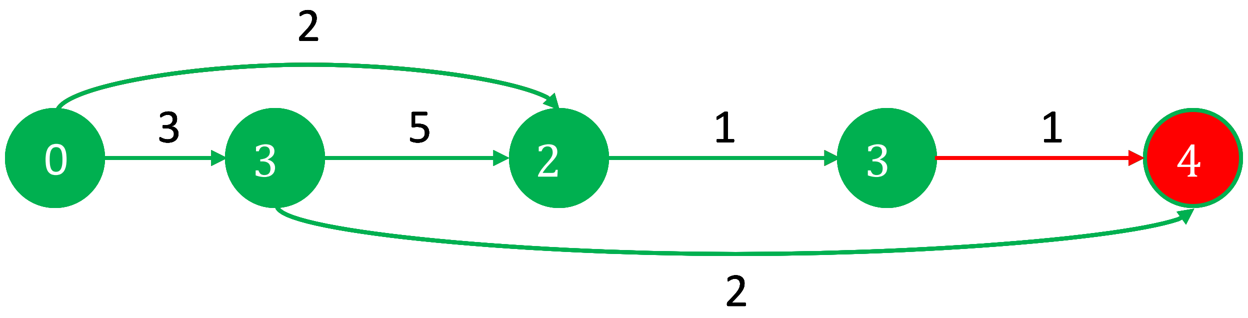
```
DAG-SHORTEST-PATHS (G, s)
O = topological_sort(G)
s.d = 0
for each vertex u in O
    for each vertex e ∈ G.in_edges[u]
        dp[u] = min(dp[u], dp[v.start] + e.w)
```

Time Complexity

- Topological Sort: O(|V|+|E|)
- DP: O(|V|+|E|)

Single-Source Shortest Paths in Directed Acyclic Graphs

Calculate shortest-path distance for vertices in topologically sorted order



How to deal with cycles?

Relaxation

An important operation for many shortest-path algorithms

Iterately update a shortest-path estimate $v.d$: an upper bound for $\delta(s, v)$

- The shortest path from s to v that has been found so far
  - *Upper-bound property*: for all vertices $v \in V, v.d \geq \delta(s, v)$.

Procedure of Estimation:

- Initialize shortest-path estimates as $s.d = 0$ and $v.d$=∞ for $v \in V - \{s\}$: $\Theta(V)$.
- Relaxation an edge $e =< u, v >$: improve v's shortest-path estimate through u's shortest-path estimate $v.d = \min(v.d,\ u.d + e.w)$
  - *No-path property*: If there is no path from s to v, then $v.d = \infty$
    - The non-infinity is passed from s to other nodes through connected edges

Convergence Property

Convergence Property: If there is a shortest path from s to v that contains an edge $e = < u, v >$, and $u.d = \delta(s,u)$ before relaxing edge e, $v.d = \delta(s,v)$ after relaxing e

Proof:

- *(Optimal Substructure)* a shortest path $p = < e_{p_1}, e_{p_2}, \dots, e >$ linking s and v contains $p_{1:k-1} = < e_{p_1}, e_{p_2}, \dots, e_{p_{k-1}} >$, which is a shortest path between s and u
- $\delta(s,v) = w(p) = w(p_{1:k-1}) + e.w = \delta(s,u) + e.w = u.d + e.w \geq v.d$
- $\delta(s,v) = v.d$

Path-relaxation Property

Path-relaxation property: If $p =< e_{p_1}, e_{p_2}, \dots, e_{p_k} >$ is a shortest path, where $e_{p_1}.start = s$. Denoting $e_{p_i}.end = v_i$, relaxing edges in p in order makes $v_i.d = \delta(s, v_i)$

- Proof: based on convergence property, mathematical induction

Lemma: given a sequence of relaxing edges: $p' =< e_{p'_1}, e_{p'_2}, \dots, e_{p'_m} >$, if a shortest path p is a subsequence of p', $v_i.d = \delta(s, v_i)$ after the relaxation if v_i is in p.

- Relaxation will not increase shortest-path estimate. So estimates for nodes with $u.d = \delta(s, u)$ not changed after any additional relaxation

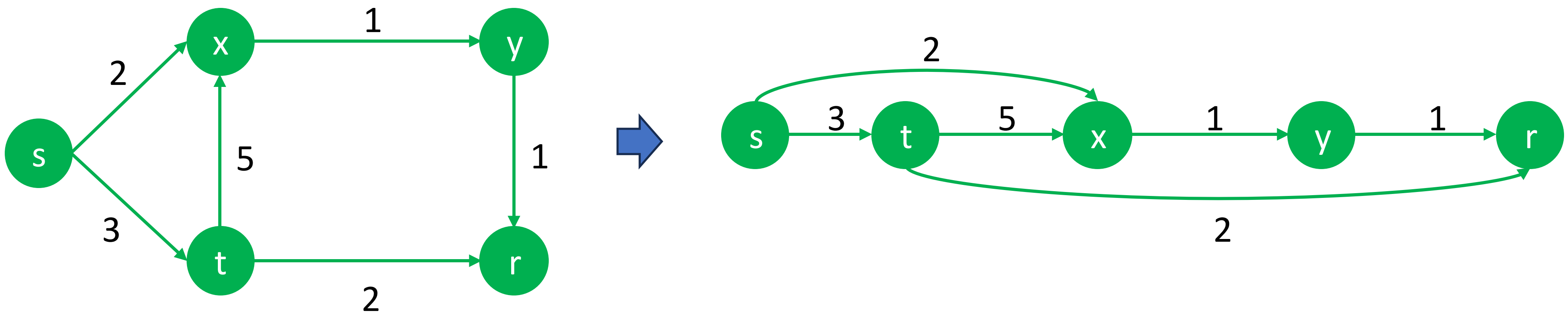
Path-relaxation property is regardless of additional path relaxations!

General solution: construct a relaxation sequence that contains a shortest path for each node as subsequences!

Single-Source Shortest Paths in DAG

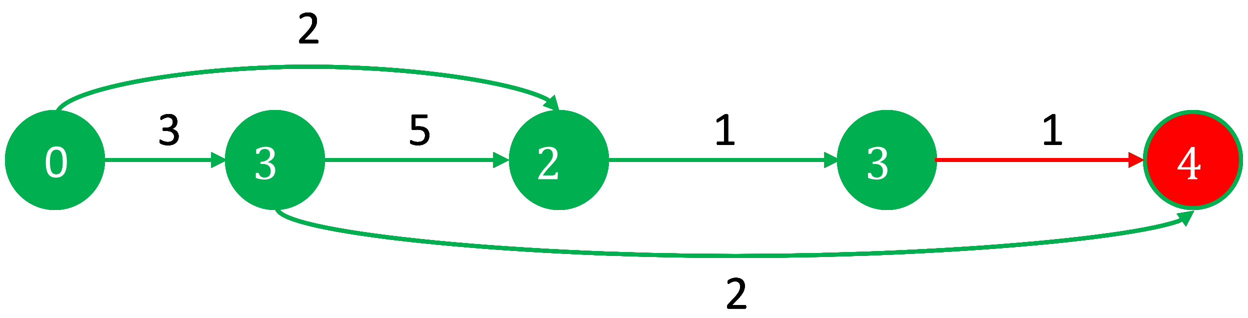
Induce the algorithm from the view of relaxation:

- Ordering edges by the starting node's topological sort order as a sequence E', all the paths from s to any node u are subsequences of E'
- According to *Path-relaxation property* (if $p =< e_{p_1}, e_{p_2}, \dots, e_{p_k} >$ is a shortest path from $s = v_0$ to v_k, relaxing edges in p in order makes $e_{p_i}.end.d = \delta(s, v_k)$), every node's $u.d = \delta(s, u)$



Single-Source Shortest Paths in Directed Acyclic Graphs

Take each vertex in topologically sorted order and relax each out edges



Single-Source Shortest Paths in Directed Acyclic Graphs

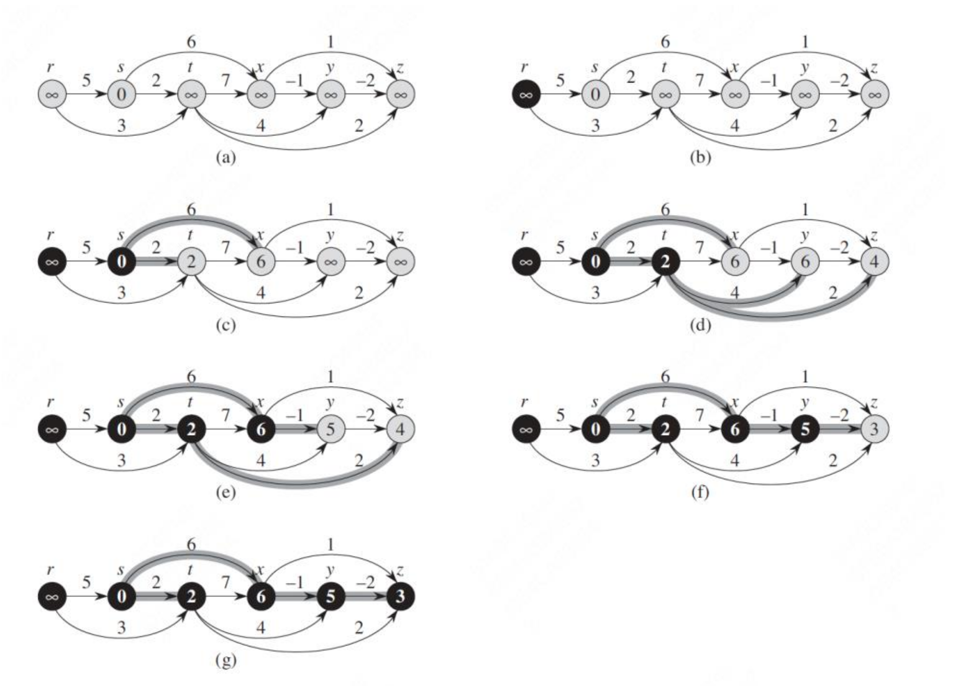
Take each vertex in topologically sorted order relax each edge

Pseudocode

```
DAG-SHORTEST-PATHS (G, w, s)
O = topological_sort(G)
s.d = 0
for each vertex u in O
    for each vertex v ∈ G.Adj[u]
        relax(u, v)
```

Time Complexity

- Topological Sort: O(|V|+|E|)
- Relaxation: O(|V|+|E|)



The Bellman-Ford algorithm

Shortest Paths in a Graph with Cycle

Can shortest-path estimates and relaxation handle cycles?

- Find a relaxation sequence containing a shortest path for each node as subsequences

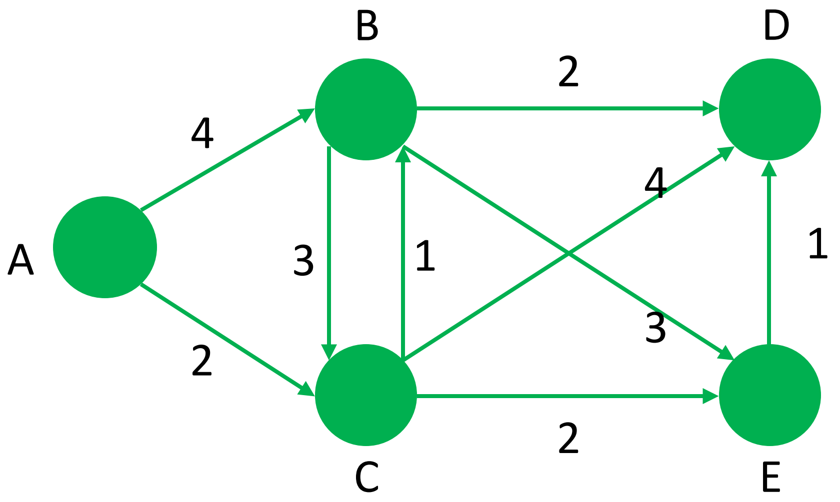
Intuition: A shortest path between two vertices can have at most |V|-1 edges

- A cycle can be removed without increasing the distance
- By repeating $< e_1, e_2, \ldots, e_n >$ for $|V| - 1$ times, all shortest paths can be found as subsequences

Idea of Bellman-Ford algorithm: relax all edges for |V|-1 times

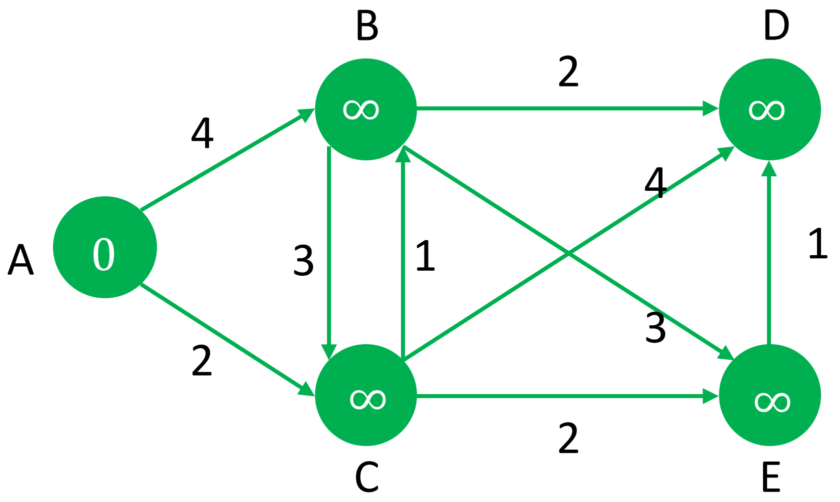
Bellman-Ford Algorithm

Given a graph



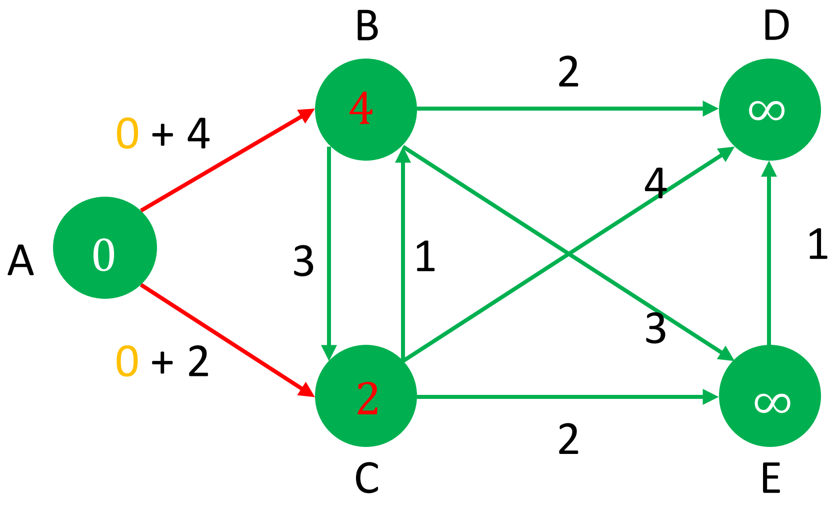
Bellman-Ford Algorithm

Initialize the estimates



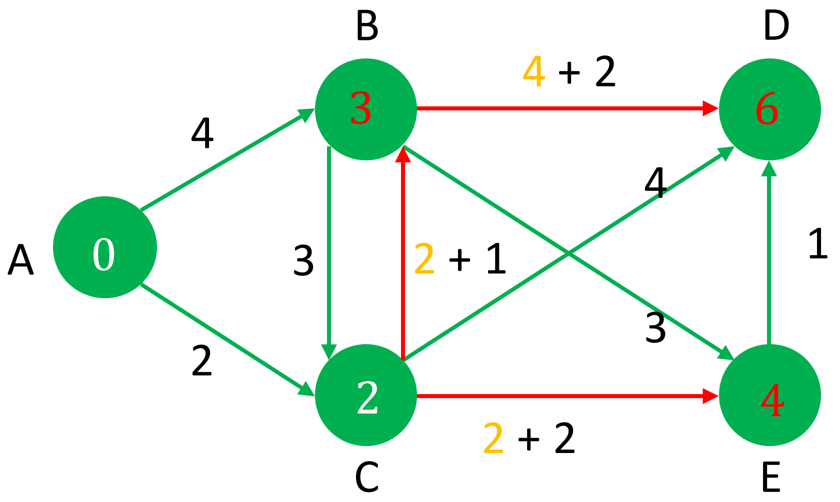
Bellman-Ford Algorithm

Relax all the edges: round 1



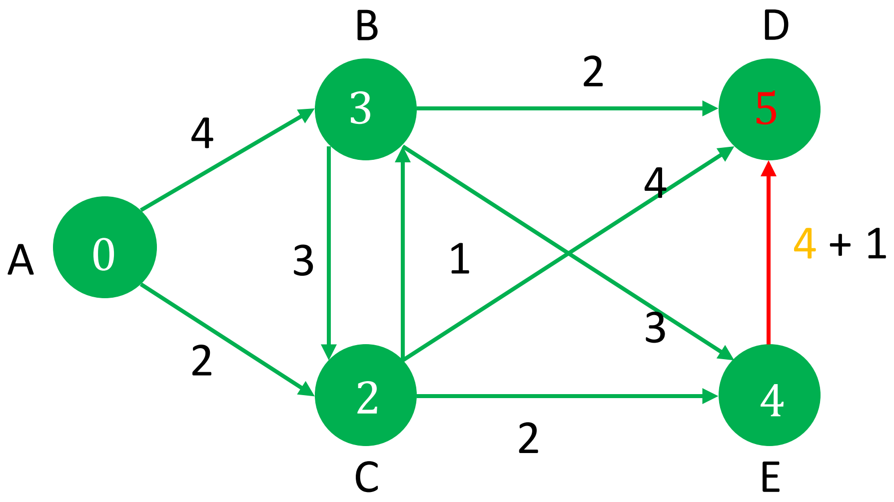
Bellman-Ford Algorithm

Relax all the edges: round 2



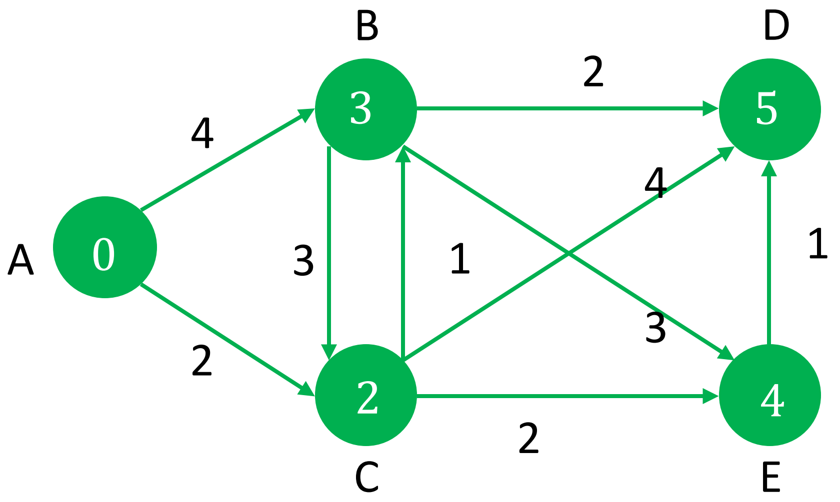
Bellman-Ford Algorithm

Relax all the edges: round 3



Bellman-Ford Algorithm

Relax all the edges: round 4



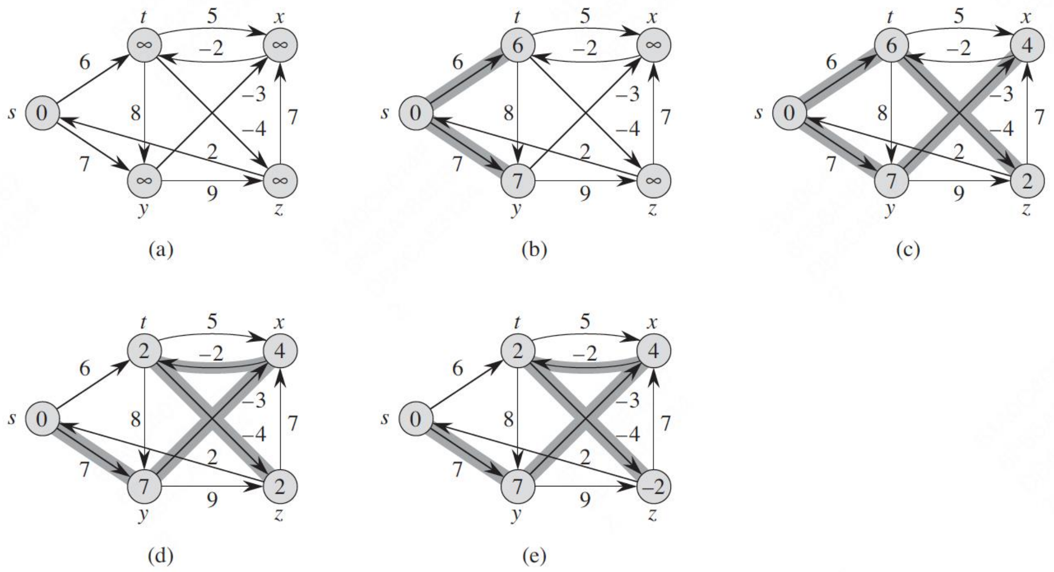
Bellman-Ford Algorithm

Steps:

1. Initialize shortest-path estimates to all the vertices
2. Visit each edge and relax the path distances
3. Repeat V - 1 times

Time Complexity: $O(VE)$

```
BELLMAN-FORD(G, w, s)
1. INITIALIZE-SINGLE-SOURCE(G, s)
2. for i=1 to |G.V|-1
3.     for each edge (u,v) ∈ G.E
4.         RELAX(u,v,w)
```

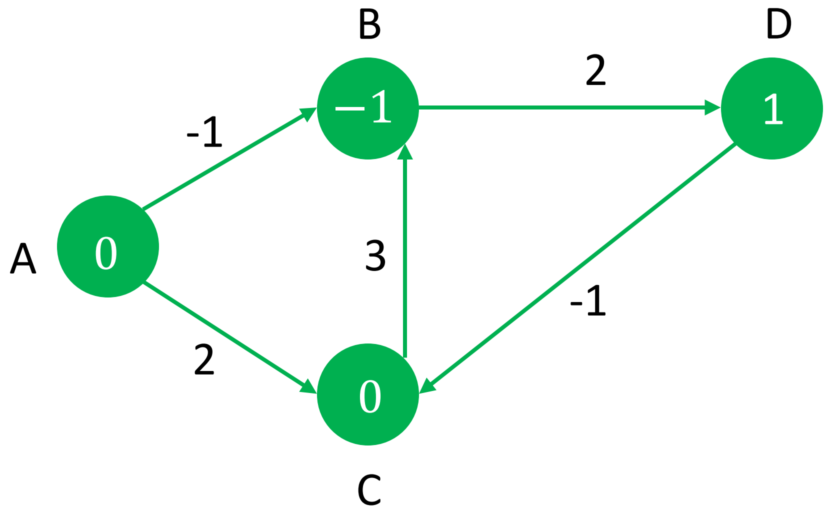


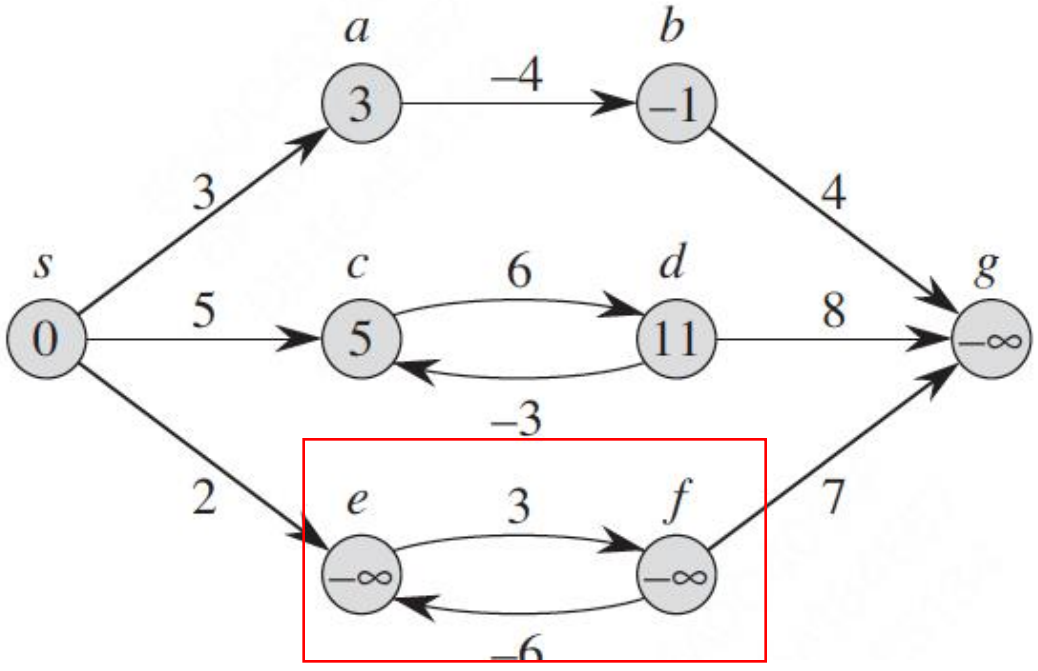
Negative-Weighted Edges Problem

Problem: a graph may contain a negative-weighted edge

Observation: if there is a negative-weight cycle, nodes that can reach s through that cycle have Infinitesimal weight

- If no negative-weight cycles are reachable from s, the problem remains well defined





Detect Negative-weight Cycles with Bellman-Ford Algorithm

Lemma: G contains a negative-weight cycle reachable from s if and only if there is an edge $e \ = < u, v >$ where $v.d > u.d + e.w$ after $|V| \ - \ 1$ rounds of relaxation

Proof:

- If there is no negative-weight cycle reachable from s: according to triangle inequality, $\forall e =< u, v > \in \mathrm{E}, v.d = \delta(s, v) \leq \delta(s, u) + w(u, v) = u.d + w(u, v)$
- If there is a negative-weight cycle $\mathrm{c} =< v_0, v_1, \ldots, v_k >$ reachable from s, where $v_0 = v_k$. e_i is the edge connecting v_{i-1} and v_i:
  - Assume for contradiction that all the edges have $v.d \leq u.d + e.w$. Then $\sum_{i=1}^{k} v_i.d \leq \sum_{i=1}^{k} v_{i-1}.d + \sum_{i=1}^{k} w(v_{i-1}, v_i)$. Since negative weight cycle, $\sum_{i=1}^{k} w(v_{i-1}, v_i) < 0$. $\sum_{i=1}^{k} v_i.d \leq \sum_{i=1}^{k} v_{i-1}.d + \sum_{i=1}^{k} w(v_{i-1}, v_i) < \sum_{i=1}^{k} v_{i-1}.d = \sum_{i=0}^{k-1} v_i.d$
  - However, with $v_0 = v_k$, $\sum_{i=1}^{k} v_i.d = \sum_{i=0}^{k-1} v_i.d$.

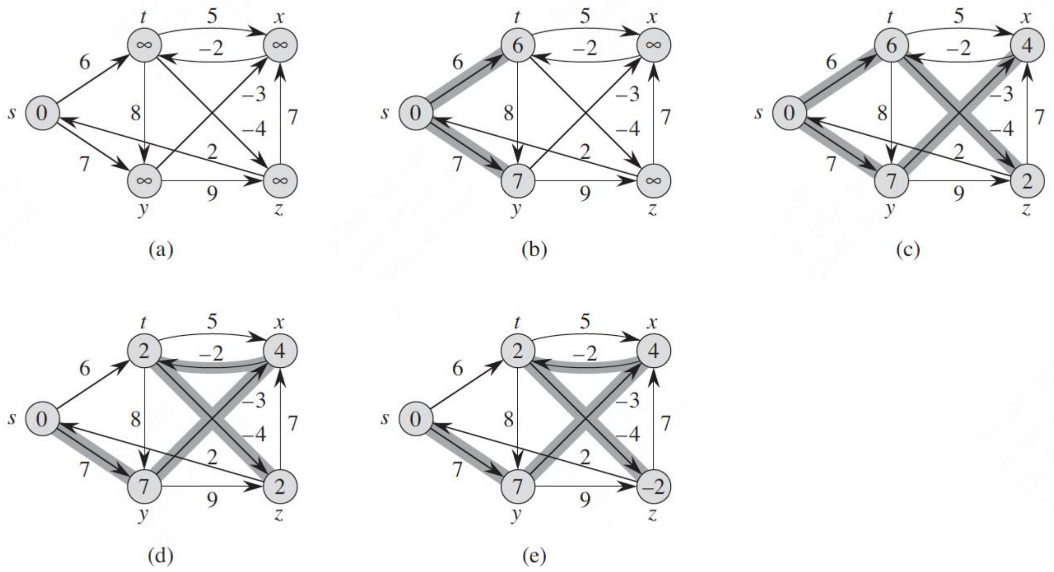
Bellman-Ford Algorithm

Steps:

1. Initialize shortest-path estimates to all the vertices
2. Visit each edge and relax the path distances
3. Repeat V - 1 times

Time Complexity: $O(VE)$

```
BELLMAN-FORD(G, w, s)
INITIALIZE-SINGLE-SOURCE(G, s)
for i=1 to |G.V|-1
    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 FALSE
return TRUE
```



Dijkstra’s algorithm

Accelerate the Algorithm

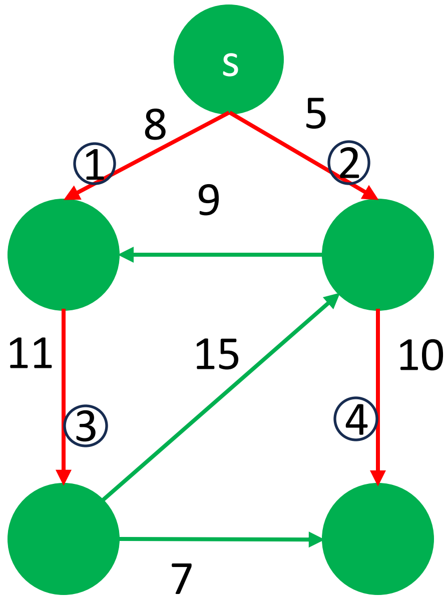
Path-relaxation property: If $p = < e_{p_1}, e_{p_2}, \ldots, e_{p_k} >$ is a shortest path from $s = v_0$ to v_k, relaxing edges in p in order makes $e_{p_i}.end.d = \delta(s, v_k)$, regardless of additional relaxations.

General solution: construct a relaxation sequence that contains a shortest path for each node as subsequences! —— complexity is the length of the sequence

How to make the sequence shorter?

The shortest possible sequence?

- Relaxing edges layer-by-layer in a shortest-path tree
- Complexity: O(V)
- Problem: shortest-path tree is unknown



Intuition: construct the shortest-path tree layer-by-layer to relax each edge only once

Try to Construct the Shortest-Path Tree

When all the edge weights are non-negative:

1. The out-edge of s with the smallest weight must be in a shortest-path tree
   - Proof: Assume the edge is e and $e.end = u$, for any other path $p = s \to x \leadsto u, w(p') \geq w(s,x) \geq e.w$. Therefore, e is a shortest path from s to u.
2. Consider out-edges of u and s: $e^* = \operatorname{argmin}_{e'.start \in \{s,u\}, e' \neq e} w(p(e'))$ is in a shortest-path tree, where $p(e') =< e' > if\ e'.start = s,\ p(e') =< e, e' > if\ e'.start = u$
   - Proof: Assume $e^*.end = v$ and the generated path is p'. For any other paths $p'' = s \to x \leadsto v$, we have $w(p'') \geq w(s,x) \geq \min_{e'.start=s, e' \neq e} e'.w = \min_{e'.start=s, e' \neq e} w(p(e')) \geq w(p')$
3. Consider out-edges of u, v and s, the one producing shortest $p = s \leadsto \{u, v\} \to$ *un-visited node* must be in a shortest-path tree

Dijkstra's Algorithm

A greedy algorithm that handles the case without negative edge weights

Key Idea

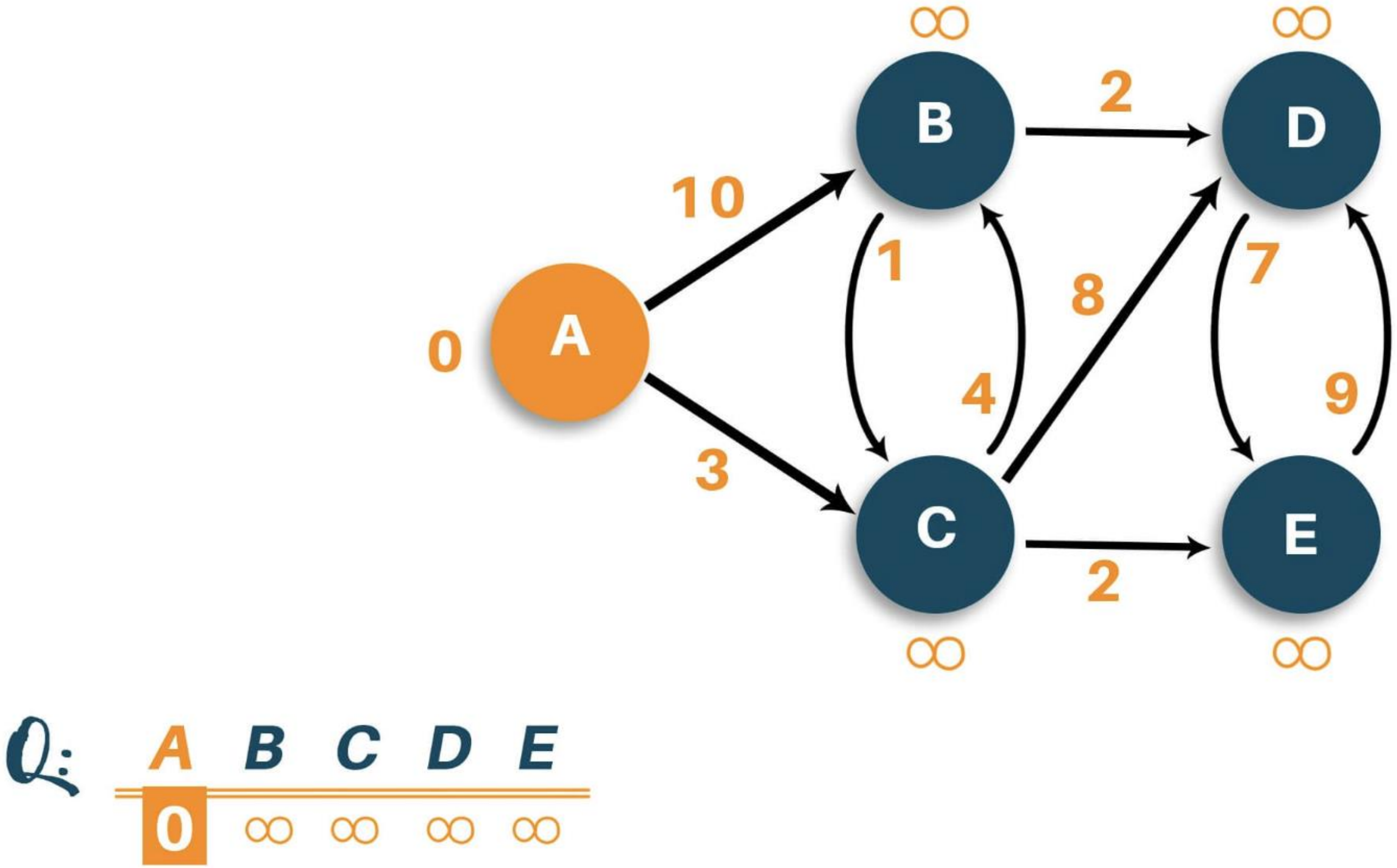
- Start from the source, construct the shortest-path tree by adding nodes
- Maintain each un-visited node v's shortest-path estimate $v.d$ as their shortest distance to s through paths composed of selected nodes S
  - Relax the adjacent edges when a new node is added
- The unvisited node with smallest $v.d$ has $v.d = \delta(s, v)$, can be added to the tree
  - Proof: consider all the possible paths linking s and v
    - $s \rightsquigarrow^{S} v$: shortest distance is $v.d$
    - $s \rightsquigarrow^{S} x \rightsquigarrow v$: shortest distance is at least $x.d \geq v.d$

Dijkstra’s Algorithm

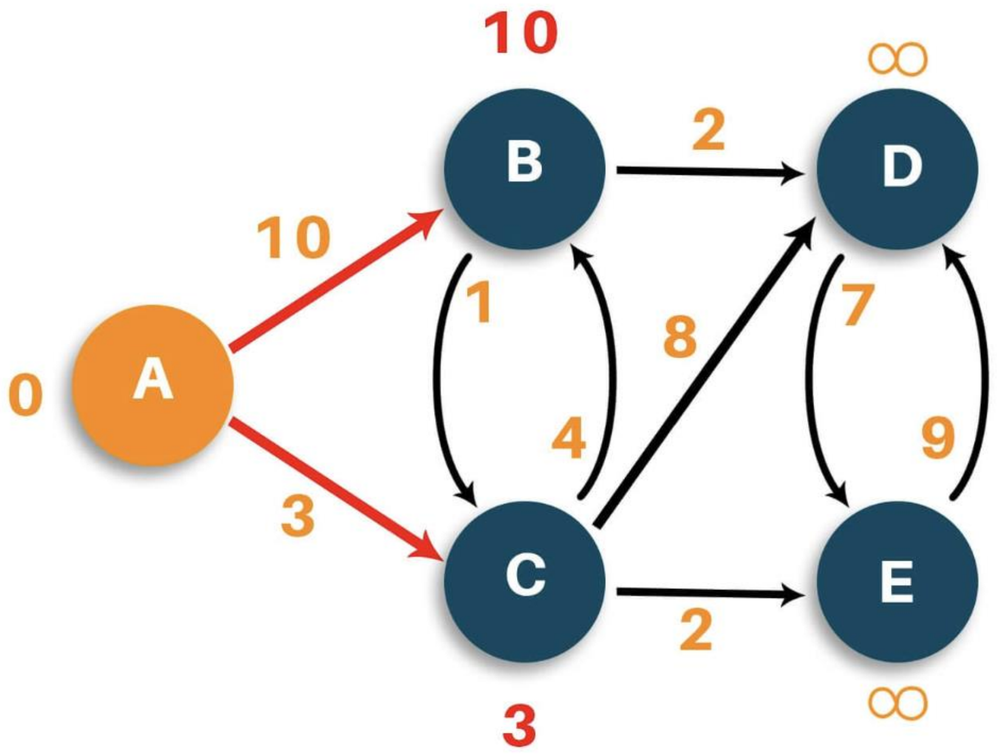
Steps:

1. Initialize $S = \{\}$ and $s.d = 0$, all the other nodes are $u.d = \infty$
2. Select the vertex $u \in V - S$ with the minimum shortest-path estimate and add u to S
3. Relaxes all edges leaving u
4. Repeat 3-4 until all the nodes have been visited

Dijkstra’s Algorithm



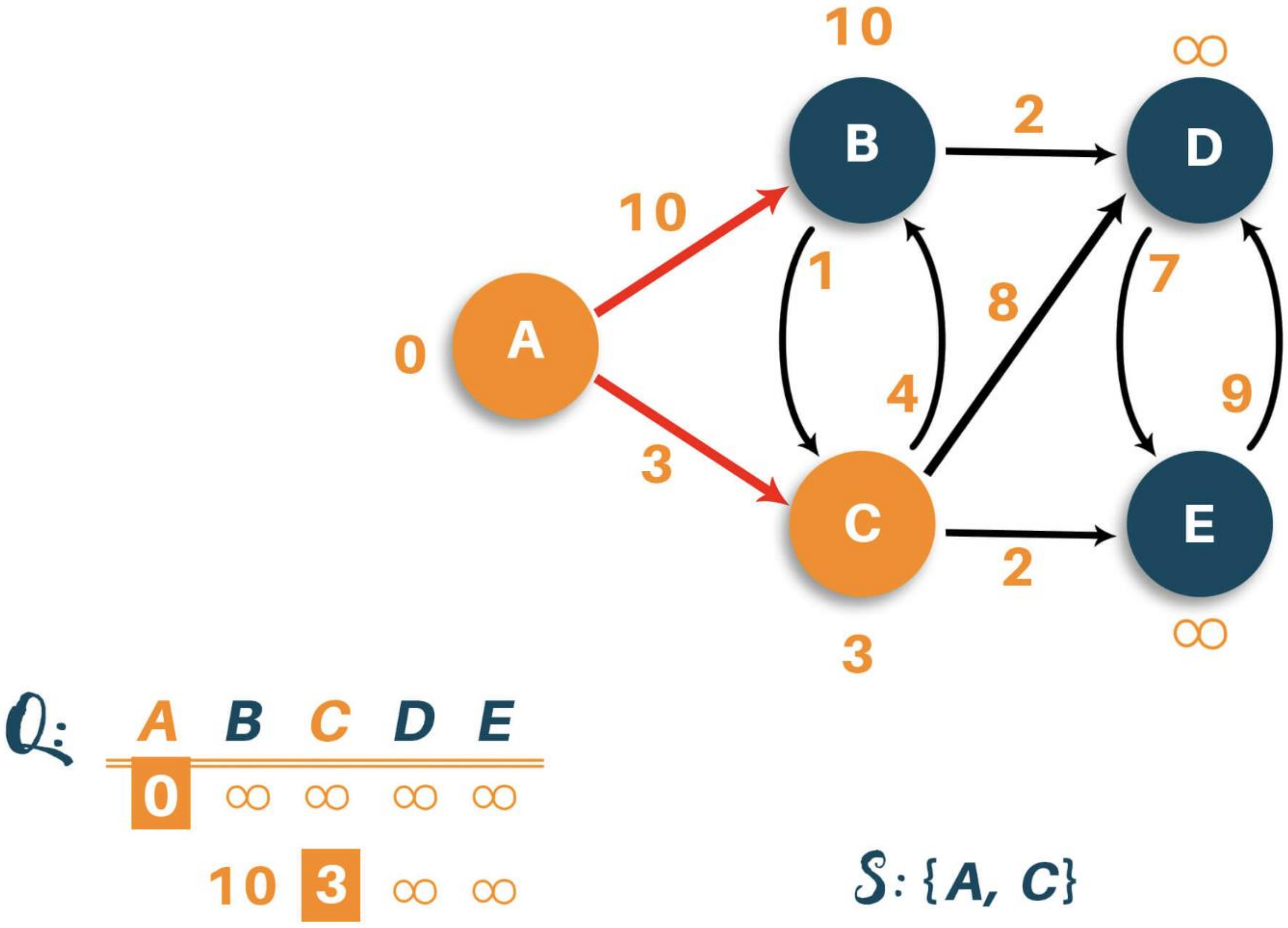
Dijkstra's Algorithm



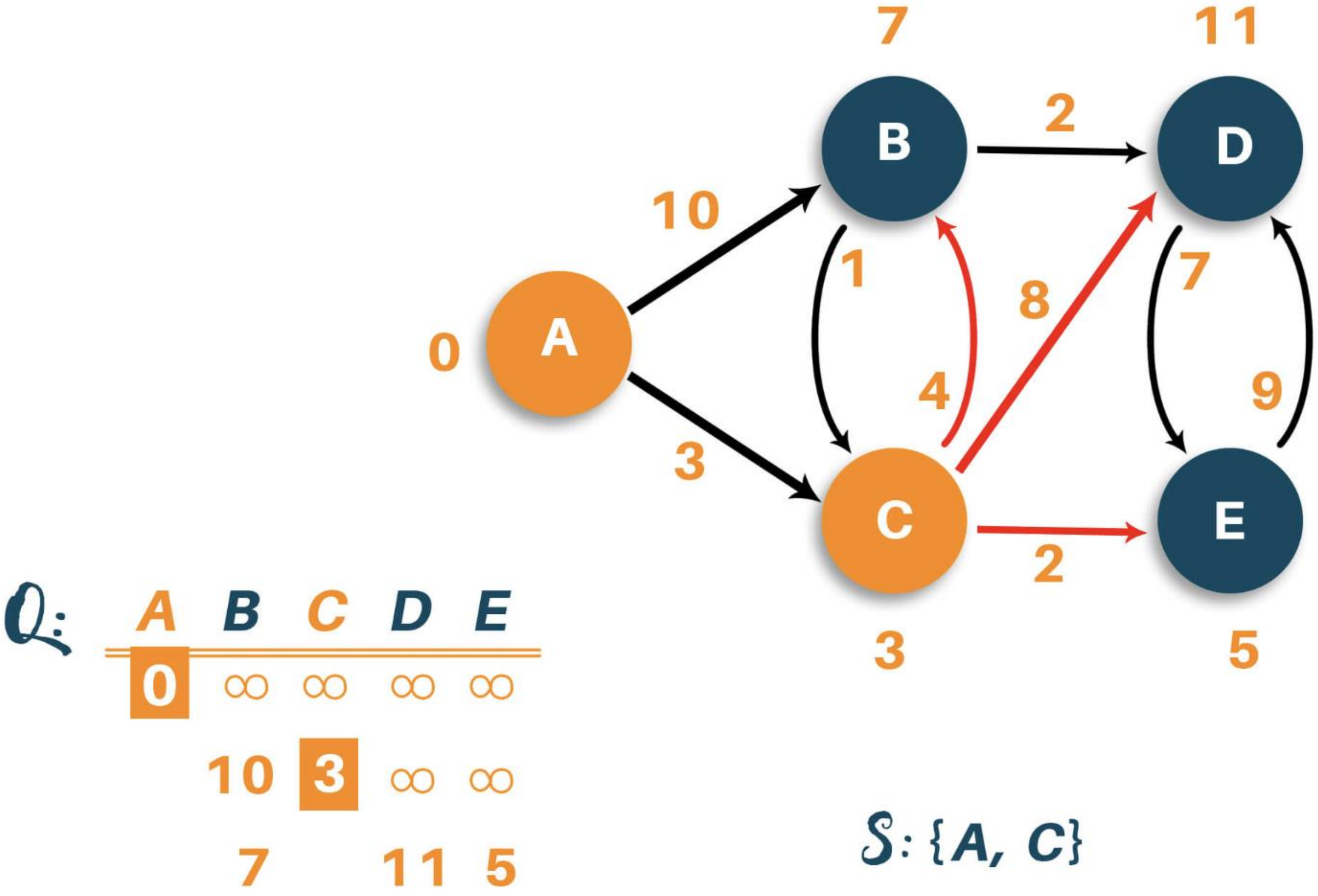
Q:	A	B	C	D	E
	0	∞	∞	∞	∞
		10	3	∞	∞

S: {A}

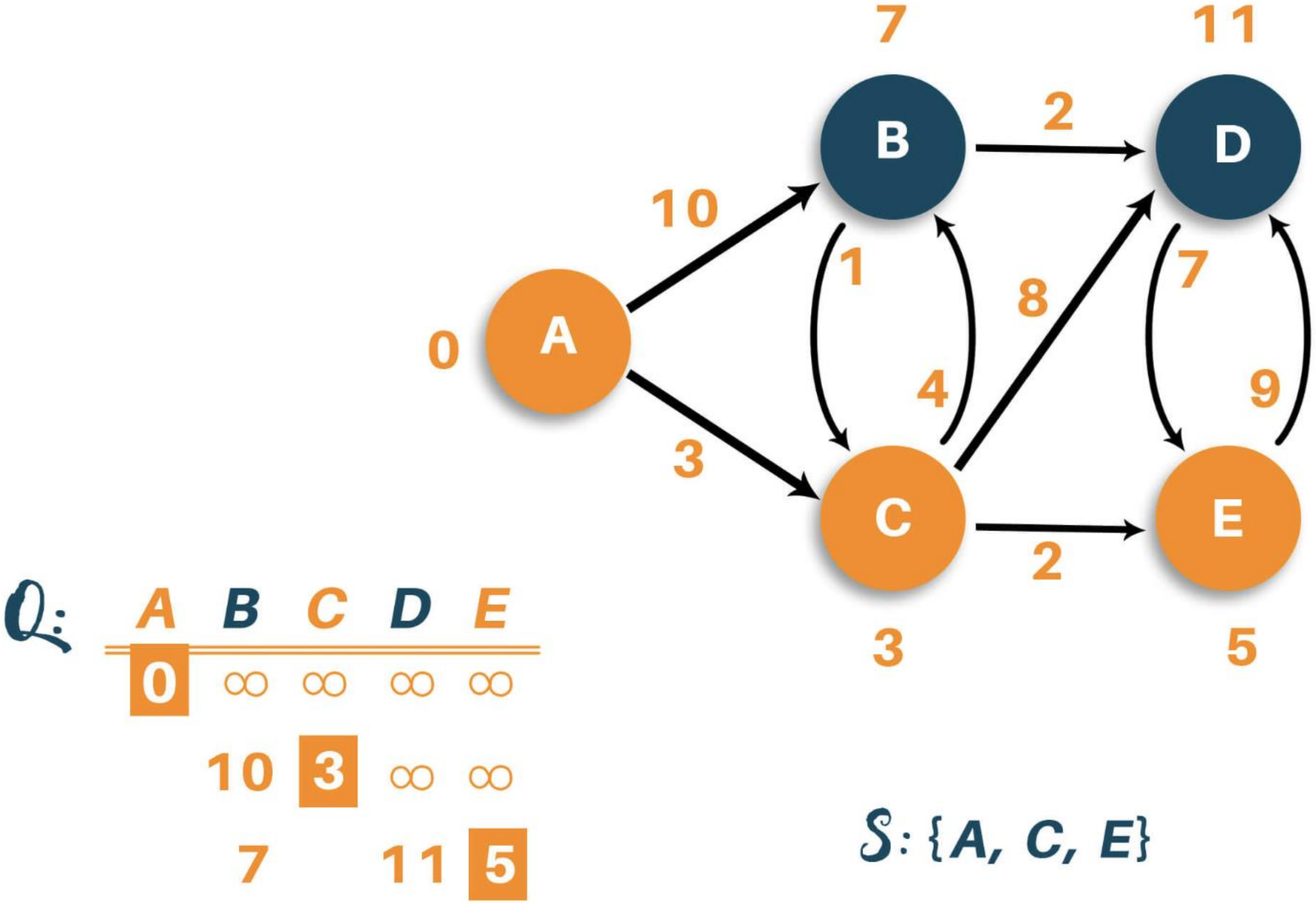
Dijkstra’s Algorithm



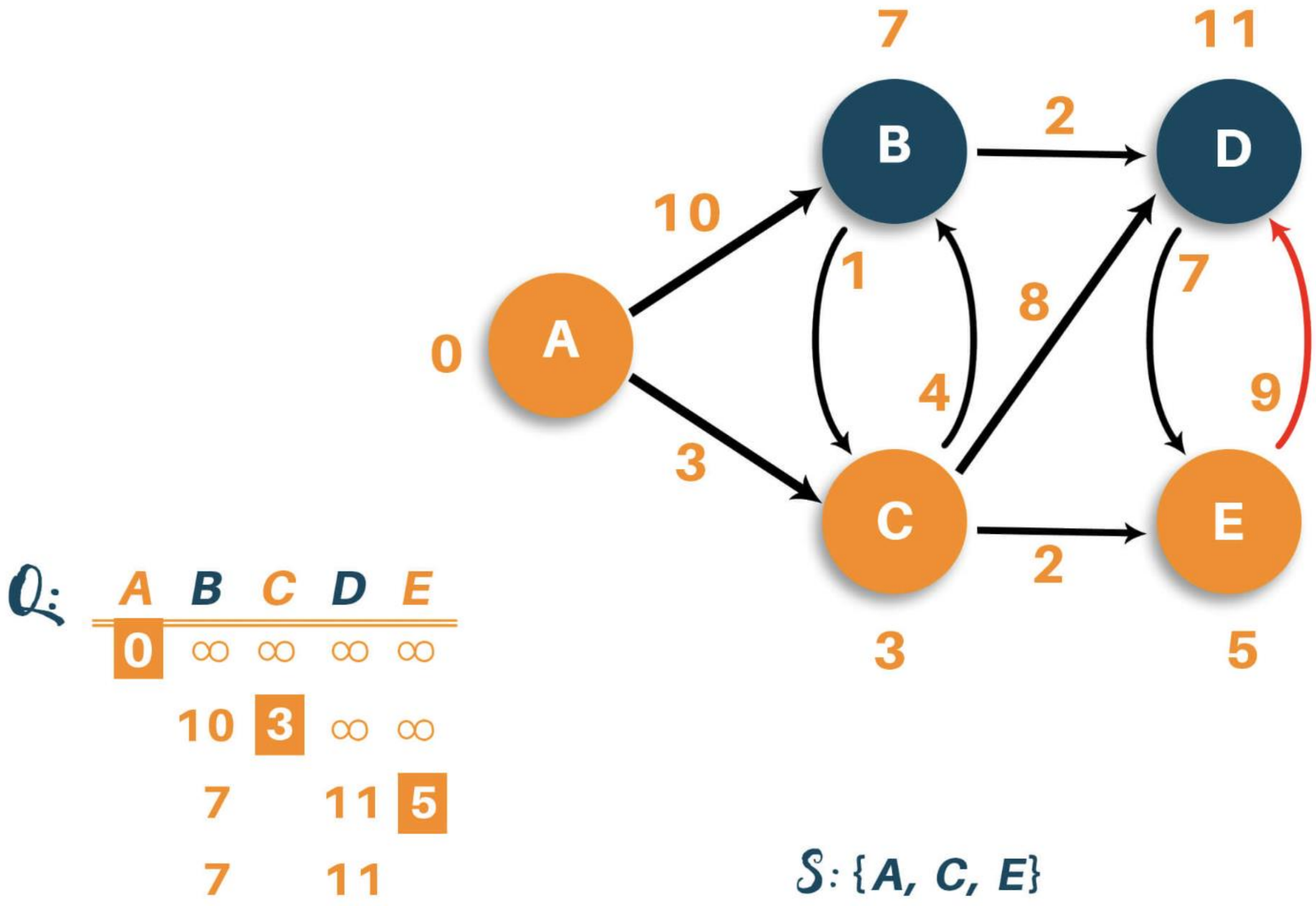
Dijkstra’s Algorithm



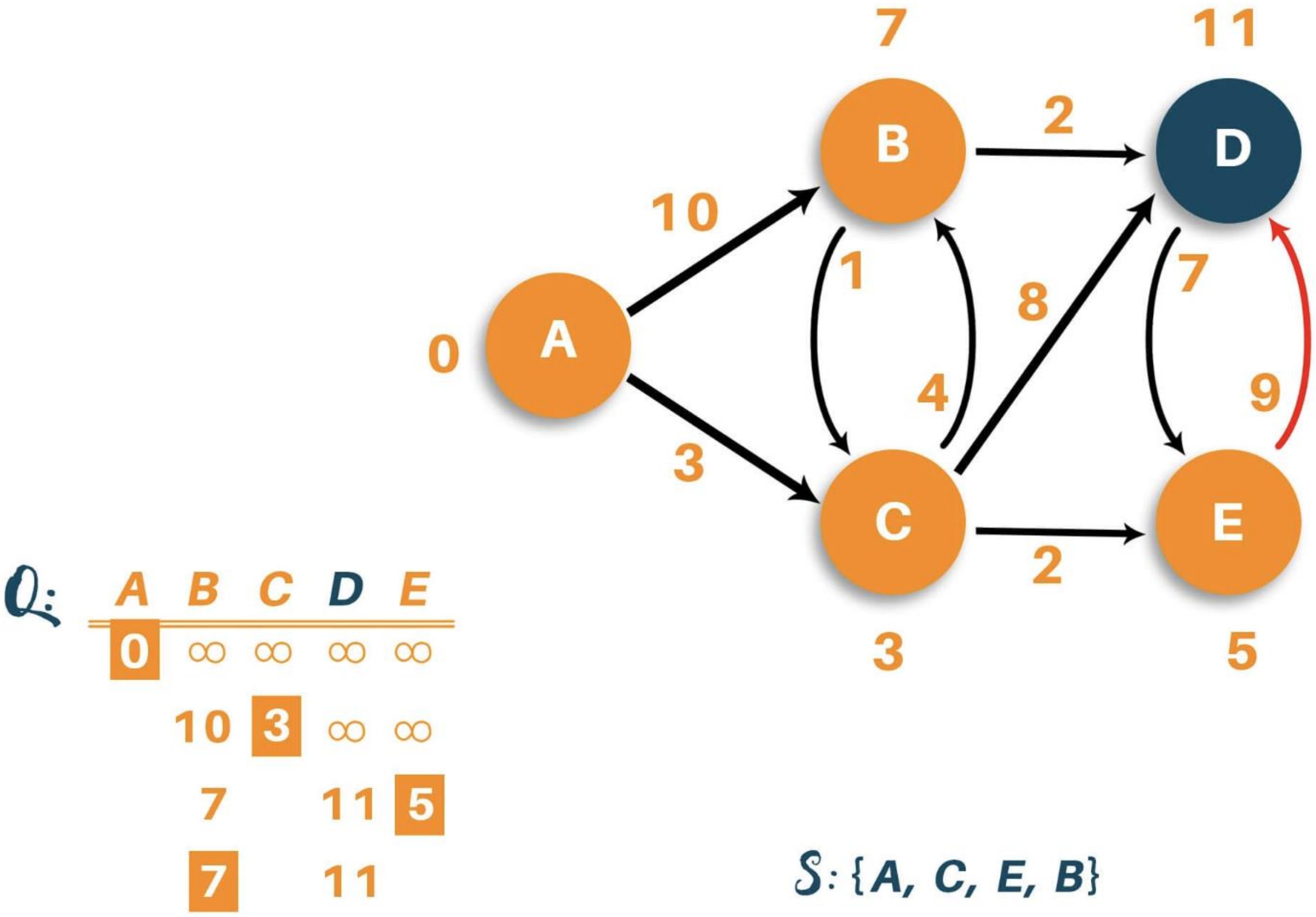
Dijkstra's Algorithm



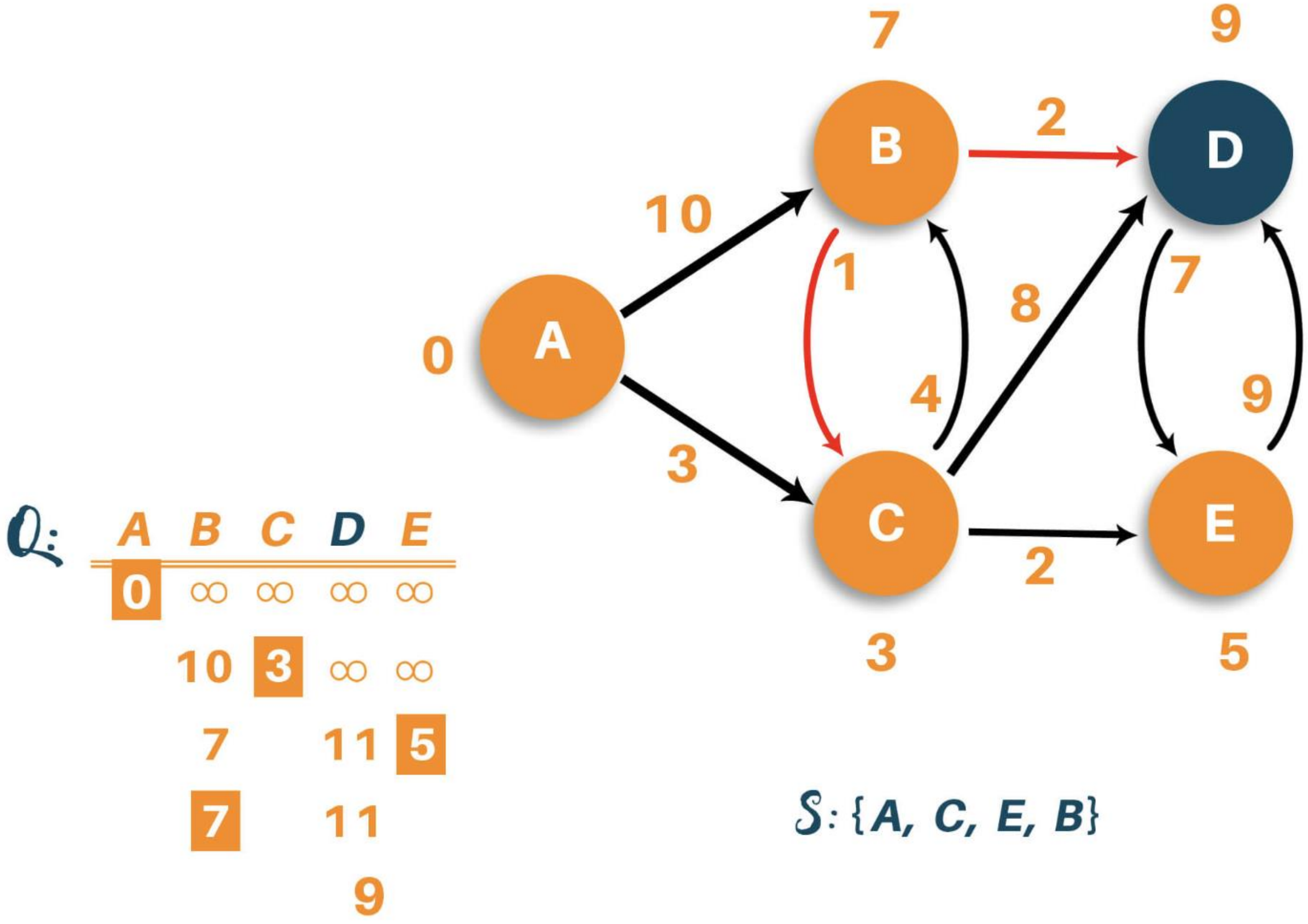
Dijkstra’s Algorithm



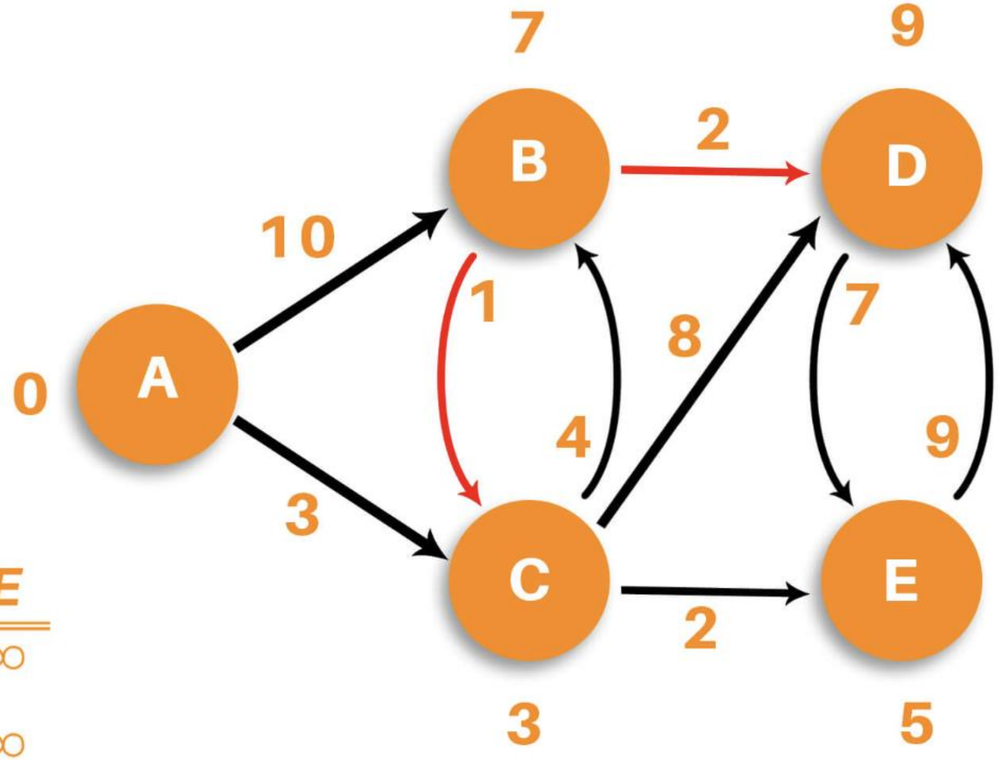
Dijkstra’s Algorithm



Dijkstra’s Algorithm



Dijkstra's Algorithm



Q:

A	B	C	D	E
0	∞	∞	∞	∞
	10	**3**	∞	∞
	7		11	**5**
	7		11	
			9	

S: {A, C, E, B, D}

Dijkstra’s Algorithm

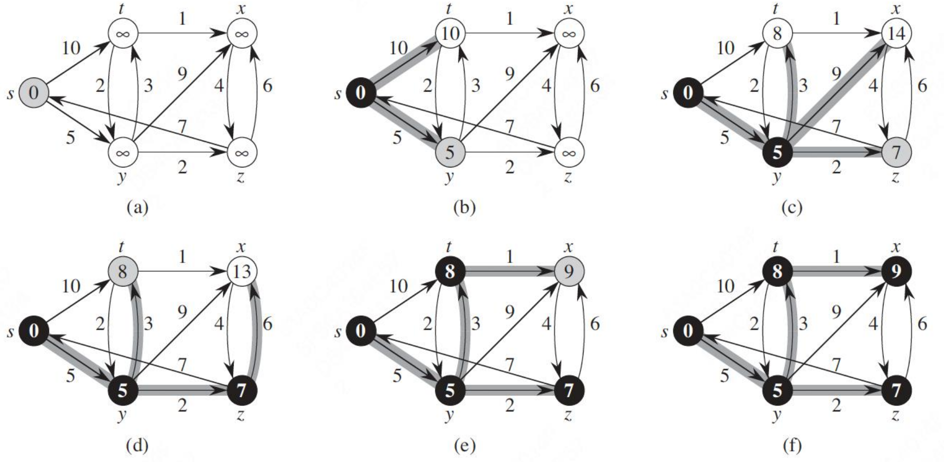
Repeatedly select the vertex $u \in V - S$ with the minimum shortest-path estimate, adds u to S, and relaxes all edges leaving u

Implementation: min-heap/binary search tree

Time Complexity:

- Visit each node once, each time $O(\log V)$: $O(V\log V)$
- Relaxing each edge and update the edge in the heap for only once: $O(E\log V)$
- Total: $O(E\log V)$

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



Application: Difference Constraints Problem

Application: Difference Constraints Problem

Given $b \in R^m$ and $A \in R^{m \times n}$, where each row in A has one 1 and one -1, the rest are all 0.

Find a feasible solution $x \in R^n$ so that $Ax \leq b$

- Understanding: find a feasible value assignment for n variables so that they satisfy m pair-wise constraints on the difference upper bound, each formulated as $x_j - x_i \leq b_k$

$$\begin{pmatrix} 1 & -1 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & -1 \\ 0 & 1 & 0 & 0 & -1 \\ -1 & 0 & 1 & 0 & 0 \\ -1 & 0 & 0 & 1 & 0 \\ 0 & 0 & -1 & 1 & 0 \\ 0 & 0 & -1 & 0 & 1 \\ 0 & 0 & 0 & -1 & 1 \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \\ x_5 \end{pmatrix} \leq \begin{pmatrix} 0 \\ -1 \\ 1 \\ 5 \\ 4 \\ -1 \\ -3 \\ -3 \end{pmatrix}$$

➡

$$\begin{aligned} x_1 - x_2 &\leq 0\,, \\ x_1 - x_5 &\leq -1\,, \\ x_2 - x_5 &\leq 1\,, \\ x_3 - x_1 &\leq 5\,, \\ x_4 - x_1 &\leq 4\,, \\ x_4 - x_3 &\leq -1\,, \\ x_5 - x_3 &\leq -3\,, \\ x_5 - x_4 &\leq -3\,. \end{aligned}$$

Lemma: Let x be a solution to $Ax \leq b$. For any constant d, $x + d$ is also a solution

Constraint Graphs

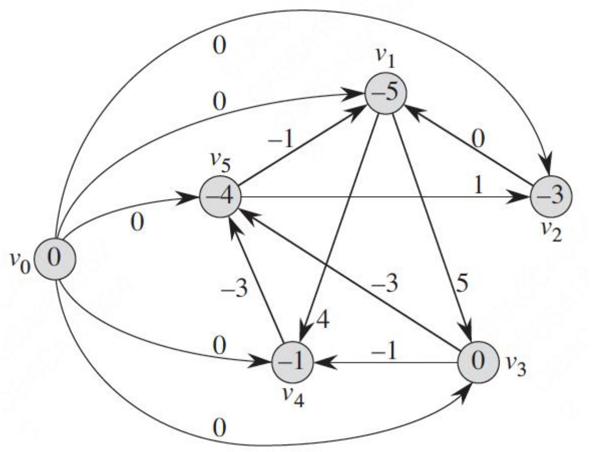
Given $Ax \leq b$, construct a constraint graph $G = (V, E)$

- Nodes: $v_1, v_2, \ldots, v_n$ corresponding to variables in x and an additional source node v_0
- Edges:
  - An edge $< v_i, v_j >$ with the edge weight as b_k for each constraint $x_j - x_i \leq b_k$
  - An edge $< v_0, v_i >$ from the source to each variable with 0 edge weight

Observation: if G contains a negative-weight cycle, there is no feasible solution

- Adding the constraints on the cycle together, we have

$$x_j - x_i + x_i - x_k + x_k \ldots - x_j = 0 < 0$$

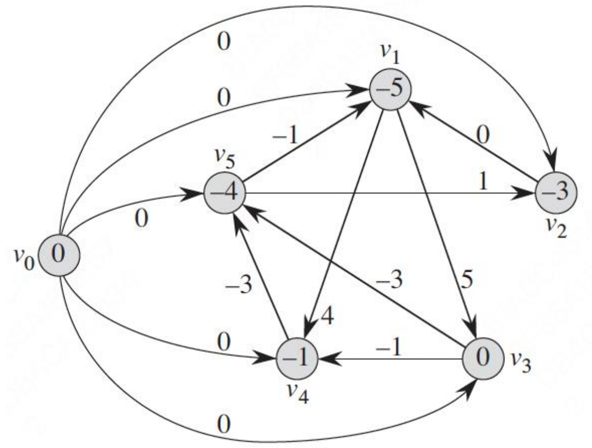


Constraint Graphs

Given $Ax \leq b$, construct a constraint graph $G = (V, E)$

- Nodes: $v_1, v_2, \ldots, v_n$ corresponding to variables in x and an additional source node v_0
- Edges:
  - An edge $< v_i, v_j >$ with the edge weight as b_k for each constraint $x_j - x_i \leq b_k$
  - An edge $< v_0, v_i >$ from the source to each variable with 0 edge weight

Lemma: $x = (\delta(v_0, v_1), \delta(v_0, v_2), \ldots, \delta(v_0, v_n))$ is a solution if there is no negative-weight cycle



- Proof: $v_j - v_i = \delta(v_0, v_j) - \delta(v_0, v_i) \leq e.w$ for any edge $e = (v_i, v_j) \in E$ (triangle inequality)

Solution: Bellman-Ford $O(VE)$

Thank you!

AIAA 5037 Advanced Algorithms and Data Structures