算法设计与分析

Lecture 8: Graph Algorithm

卢杨

厦门大学信息学院计算机科学系

luyang@xmu.edu.cn

Representations of Graphs

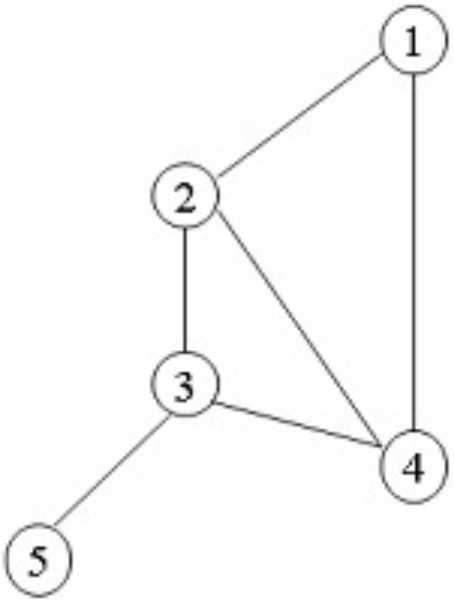
Two standard ways to represent a graph $G = (V, E)$:

- Adjacency matrix (邻接矩阵): a $|V| \times |V|$ matrix $A = (a_{ij})$:

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

- Adjacency lists (邻接表): each vertex u has a linked list $Adj[u]$, constructed by u's neighbors.

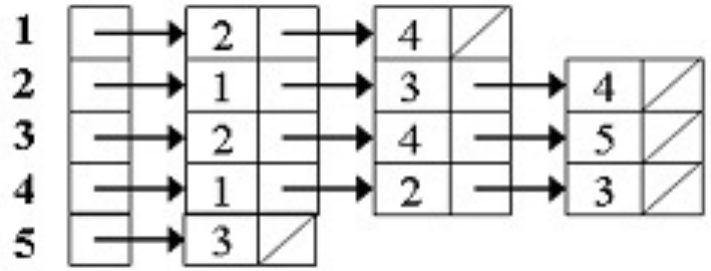
Representations of Graphs



(a)

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

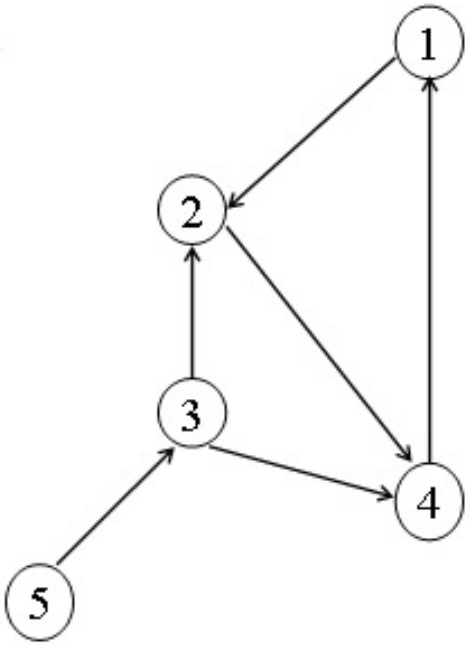
(b)



(c)

Adjacency matrix and adjacency lists for undirected graph.

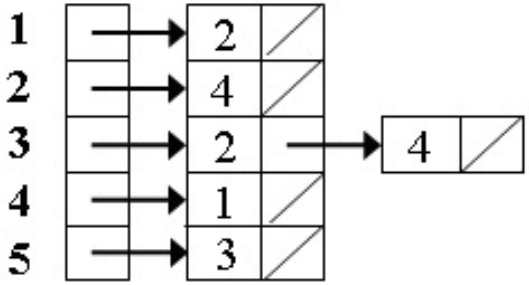
Representations of Graphs



(a)

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

(b)



(c)

Adjacency matrix and adjacency lists for directed graph.

Representations of Graphs

The usage of these two representations is different:

- Adjacency matrix is suitable for representing dense graphs - those for which $|E|$ is close to $|V|^2$.
  - Graphs close to complete graph (完全图).
- Adjacency lists represent sparse graphs - those for which $|E|$ is much less than $|V|^2$.
  - Graphs close to null graph (零图).

GRAPH SEARCH

Graph Search

- Graph search aims to compute the distance (smallest number of edges) from a vertex s to each reachable vertex.
- There are two general searching strategies: Breadth-first search (BFS) (宽度优先搜索) and depth-first search (DFS) (深度优先搜索).

GRAPH SEARCH

BREADTH-FIRST SEARCH

Breadth-First Search

- Given a graph $G = (V, E)$ and a distinguished source vertex s, BFS systematically explores the edges of G to "discover" every vertex that is reachable from s.
  - It computes the distance (smallest number of edges) from s to each reachable vertex.
  - It also produces a "breadth-first tree" with root s that contains all reachable vertices.

```
BFS(G, s)
1  for each vertex u ∈ V − {s} do
2        color[u] ← White
3        d[u] ← ∞
4        π[u] ← NIL
5  color[s] ← Gray
6  d[s] ← 0; π[s] ← NIL; Q ← ∅
7  Enqueue(Q, s)
8  while Q ≠ ∅ do
9        u ←Dequeue(Q)
10       for each v ∈ Adj[u] do
11           if color[v] = White then
12                color[v] ← Gray
13                d[v] ← d[u] + 1
14                π[v] ← u
15                Enqueue(Q, v)
16       color[u] ← DarkGray
```

Initialization for all unvisited vertices.

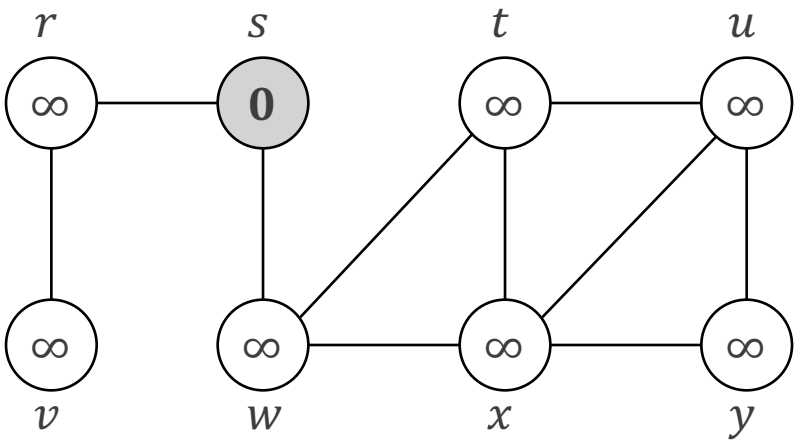
Iterate over all Gray vertices.

Iterate over all neighbors of u.

- $color[u]$: record the color of each vertex u.
  - White: Not visited.
  - Gray: Searched.
  - DarkGray: All its neighbors have been searched.
- $d[u]$: the number of edges on the path from s to u.
- $\pi[u]$: parent vertex of u.
- Q: a queue to store Gray vertices.

```
BFS(G, s)
1  for each vertex u ∈ V − {s} do
2        color[u] ← White
3        d[u] ← ∞
4        π[u] ← NIL
5  color[s] ← Gray
6  d[s] ← 0; π[s] ← NIL; Q ← ∅
7  Enqueue(Q, s)
8  while Q ≠ ∅ do
9        u ←Dequeue(Q)
10       for each v ∈ Adj[u] do
11           if color[v] = White then
12                color[v] ← Gray
13                d[v] ← d[u] + 1
14                π[v] ← u
15                Enqueue(Q, v)
16       color[u] ← DarkGray
```

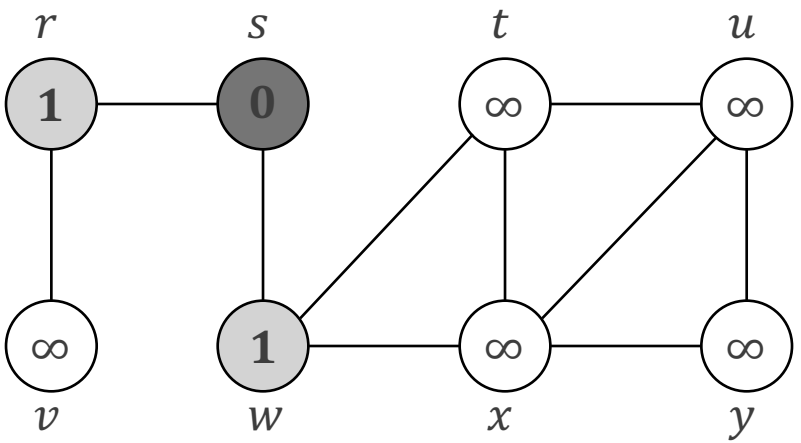
Q | s



```
BFS(G, s)
1  for each vertex u ∈ V − {s} do
2        color[u] ← White
3        d[u] ← ∞
4        π[u] ← NIL
5  color[s] ← Gray
6  d[s] ← 0; π[s] ← NIL; Q ← ∅
7  Enqueue(Q, s)
8  while Q ≠ ∅ do
9        u ←Dequeue(Q)
10       for each v ∈ Adj[u] do
11           if color[v] = White then
12                color[v] ← Gray
13                d[v] ← d[u] + 1
14                π[v] ← u
15                Enqueue(Q, v)
16       color[u] ← DarkGray
```

Dequeue$(Q) = s$

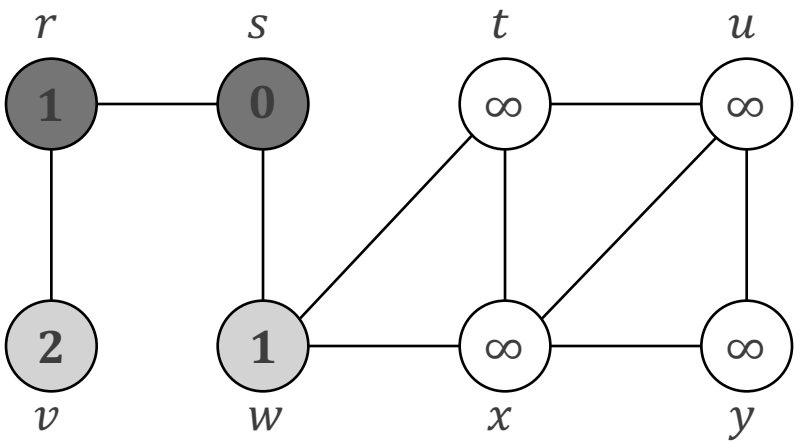
Q	r	w
	1	**1**



```
BFS(G, s)
1  for each vertex u ∈ V − {s} do
2        color[u] ← White
3        d[u] ← ∞
4        π[u] ← NIL
5  color[s] ← Gray
6  d[s] ← 0; π[s] ← NIL; Q ← ∅
7  Enqueue(Q, s)
8  while Q ≠ ∅ do
9        u ←Dequeue(Q)
10       for each v ∈ Adj[u] do
11           if color[v] = White then
12                color[v] ← Gray
13                d[v] ← d[u] + 1
14                π[v] ← u
15                Enqueue(Q, v)
16       color[u] ← DarkGray
```

Dequeue$(Q) = r$

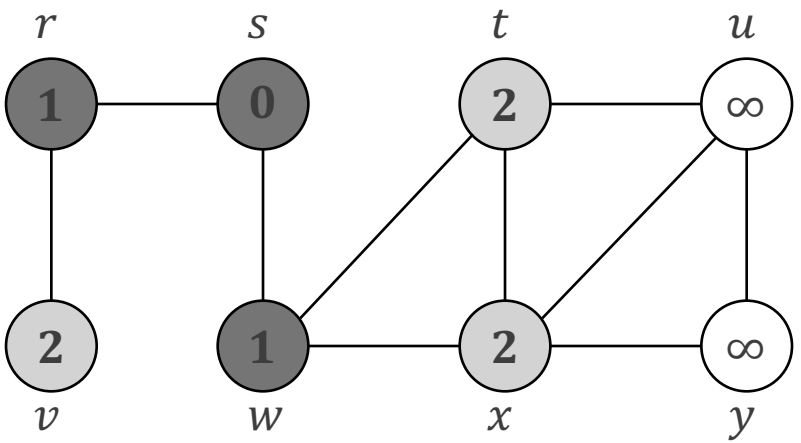
Q	w	v
1	**2**	



```
BFS(G, s)
1  for each vertex u ∈ V − {s} do
2        color[u] ← White
3        d[u] ← ∞
4        π[u] ← NIL
5  color[s] ← Gray
6  d[s] ← 0; π[s] ← NIL; Q ← ∅
7  Enqueue(Q, s)
8  while Q ≠ ∅ do
9        u ←Dequeue(Q)
10       for each v ∈ Adj[u] do
11           if color[v] = White then
12                color[v] ← Gray
13                d[v] ← d[u] + 1
14                π[v] ← u
15                Enqueue(Q, v)
16       color[u] ← DarkGray
```

Dequeue$(Q) = w$

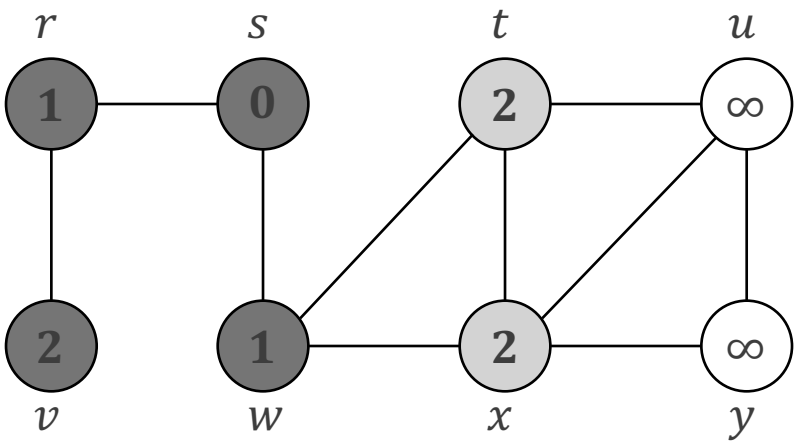
Q	v	t	x
	2	**2**	**2**



```
BFS(G, s)
1  for each vertex u ∈ V − {s} do
2        color[u] ← White
3        d[u] ← ∞
4        π[u] ← NIL
5  color[s] ← Gray
6  d[s] ← 0; π[s] ← NIL; Q ← ∅
7  Enqueue(Q, s)
8  while Q ≠ ∅ do
9        u ←Dequeue(Q)
10       for each v ∈ Adj[u] do
11           if color[v] = White then
12                color[v] ← Gray
13                d[v] ← d[u] + 1
14                π[v] ← u
15                Enqueue(Q, v)
16       color[u] ← DarkGray
```

Dequeue$(Q) = v$

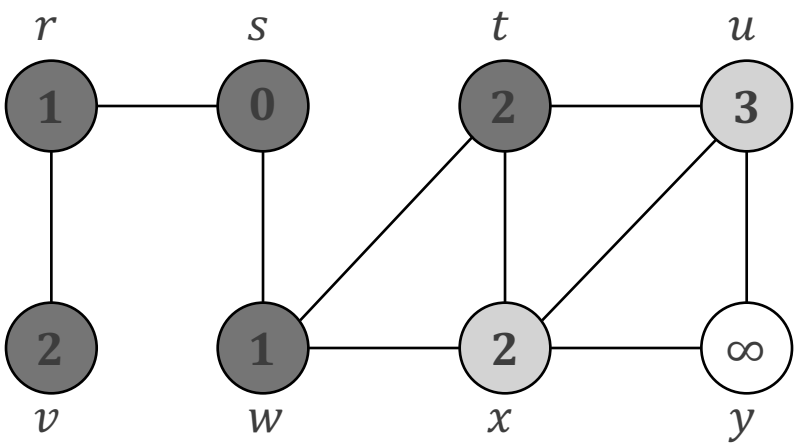
Q	t	x
	2	**2**



```
BFS(G, s)
1  for each vertex u ∈ V − {s} do
2        color[u] ← White
3        d[u] ← ∞
4        π[u] ← NIL
5  color[s] ← Gray
6  d[s] ← 0; π[s] ← NIL; Q ← ∅
7  Enqueue(Q, s)
8  while Q ≠ ∅ do
9        u ←Dequeue(Q)
10       for each v ∈ Adj[u] do
11           if color[v] = White then
12                color[v] ← Gray
13                d[v] ← d[u] + 1
14                π[v] ← u
15                Enqueue(Q, v)
16       color[u] ← DarkGray
```

Dequeue$(Q) = t$

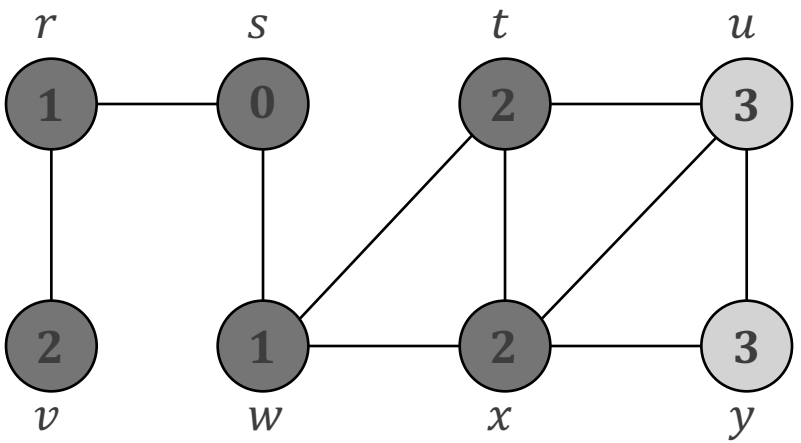
Q	x	u
	2	**3**



```
BFS(G, s)
1  for each vertex u ∈ V − {s} do
2        color[u] ← White
3        d[u] ← ∞
4        π[u] ← NIL
5  color[s] ← Gray
6  d[s] ← 0; π[s] ← NIL; Q ← ∅
7  Enqueue(Q, s)
8  while Q ≠ ∅ do
9        u ←Dequeue(Q)
10       for each v ∈ Adj[u] do
11           if color[v] = White then
12                color[v] ← Gray
13                d[v] ← d[u] + 1
14                π[v] ← u
15                Enqueue(Q, v)
16       color[u] ← DarkGray
```

Dequeue$(Q) = x$

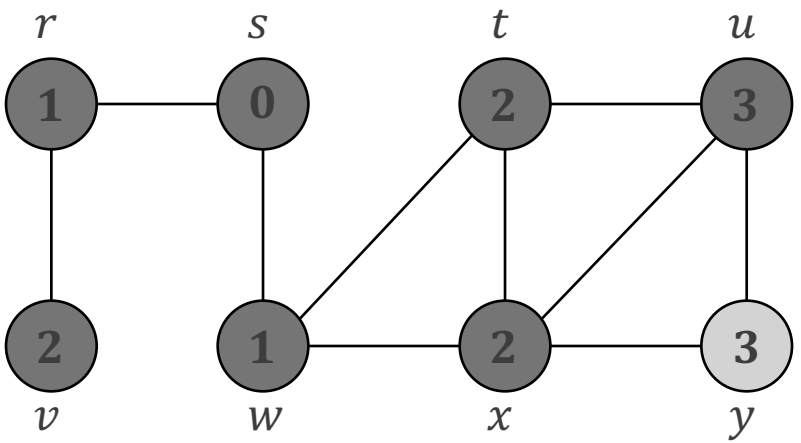
Q	u	y
	3	**3**



```
BFS(G, s)
1  for each vertex u ∈ V − {s} do
2        color[u] ← White
3        d[u] ← ∞
4        π[u] ← NIL
5  color[s] ← Gray
6  d[s] ← 0; π[s] ← NIL; Q ← ∅
7  Enqueue(Q, s)
8  while Q ≠ ∅ do
9        u ←Dequeue(Q)
10       for each v ∈ Adj[u] do
11           if color[v] = White then
12                color[v] ← Gray
13                d[v] ← d[u] + 1
14                π[v] ← u
15                Enqueue(Q, v)
16       color[u] ← DarkGray
```

Dequeue$(Q) = u$

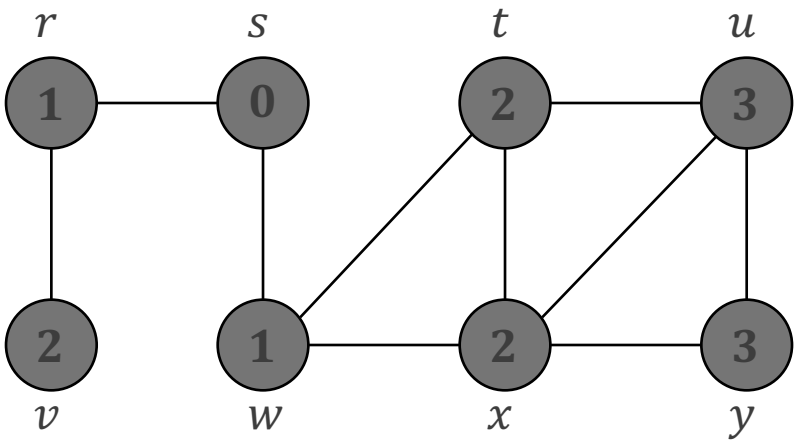
Q	y
	3



```
BFS(G, s)
1  for each vertex u ∈ V − {s} do
2        color[u] ← White
3        d[u] ← ∞
4        π[u] ← NIL
5  color[s] ← Gray
6  d[s] ← 0; π[s] ← NIL; Q ← ∅
7  Enqueue(Q, s)
8  while Q ≠ ∅ do
9        u ←Dequeue(Q)
10       for each v ∈ Adj[u] do
11           if color[v] = White then
12                color[v] ← Gray
13                d[v] ← d[u] + 1
14                π[v] ← u
15                Enqueue(Q, v)
16       color[u] ← DarkGray
```

Dequeue$(Q) = y$

Q



Correctness of BFS

- Now, we are going to prove the correctness of BFS: When BFS terminates, for all $v \in V$, $d[v]$ is the shortest path from s to v.

Definition 8.1

The shortest path distance $\delta(s, v)$ from s to v is the minimum number of edges in any path from vertex s to vertex v; if there is no path from s to v, then $\delta(s, v) = \infty$.

Correctness of BFS

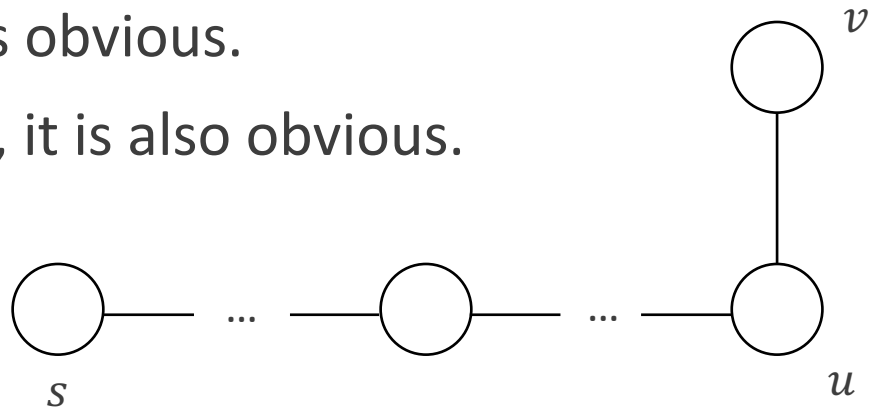
Lemma 8.1

Let $G = (V, E)$ be a directed or undirected graph, and let $s \in V$ be an arbitrary vertex. Then, for any edge $(u, v) \in E$,

$$\delta(s, v) \leq \delta(s, u) + 1.$$

Proof:

- If u is reachable from s, it is obvious.
- If u is not reachable from s, it is also obvious.



Correctness of BFS

Lemma 8.2

Let $G = (V, E)$ be a directed or undirected graph, and suppose that BFS is run on G from a given source vertex $s \in V$. Then before termination, for each vertex $v \in V$, the value $d[v]$ computed by BFS satisfies $d[v] \geq \delta(s, v)$.

- What does this Lemma say? $d[v] \geq \delta(s, v)$: The number of edges on the path from s to v is not less than the shortest path from s to v, before termination of BFS.
- Only two cases:
  - If v is not visited, $d[v] = \infty > \delta(s, v)$.
  - If v is visited, $d[v] = \delta(s, v)$.

Easy to guess, but how to prove?

Correctness of BFS

Proof:

We use induction on the number of Enqueue operations n.

- $n = 1, d[s] = 0 = \delta(s, s)$ and $d[v] = \infty \geq \delta(s, v)$ for all $v \in V - \{s\}$.
- $n = k$, before enqueuing v, we assume $d[u] \geq \delta(s, u)$, where v is a White neighbor of u.
- $n = k + 1$, after enqueuing v, $d[v] = d[u] + 1 \geq \delta(s, u) + 1 \geq \delta(s, v)$.

Lemma 8.1

Correctness of BFS

Lemma 8.3

Suppose that during the execution of BFS on a graph $G = (V, E)$, the queue Q contains the vertices $v_1, v_2, \ldots, v_r$, where v_1 is the head of Q and v_r is the tail. Then, $d[v_r] \leq d[v_1] + 1$ and $d[v_i] \leq d[v_{i+1}]$ for $i = 1,2, \ldots, r-1$.

- What does this Lemma say?
  - $d[v_r] \leq d[v_1] + 1$: the difference of $d[v]$ between head v_1 and tail v_r is not greater than 1 in Q.
  - $d[v_i] \leq d[v_{i+1}]$: $d[v]$ is not greater than its successor in Q.

Correctness of BFS

Goal of proof:
(1) $d[v_r] \leq d[v_1] + 1$
(2) $d[v_i] \leq d[v_{i+1}]$

Q	v_1	v_2	...	v_i	v_{i+1}	...	v_r
	$d[v_1]$	$d[v_2]$		$d[v_i]$	$d[v_{i+1}]$		$d[v_r]$

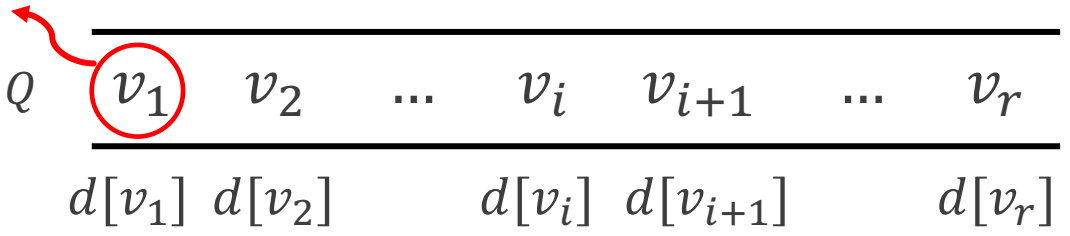
Proof:

We use induction on the number of Enqueue operations n.

- $n = 1$, the only vertex in Q is s. Obviously $d[v_r] \leq d[v_1] + 1$, because $v_1 = v_r = s$.
- $n = k$, we assume $d[v_r] \leq d[v_1] + 1$ and $d[v_i] \leq d[v_{i+1}]$.

Correctness of BFS

Goal of proof:
(1) $d[v_r] \leq d[v_1] + 1$
(2) $d[v_i] \leq d[v_{i+1}]$

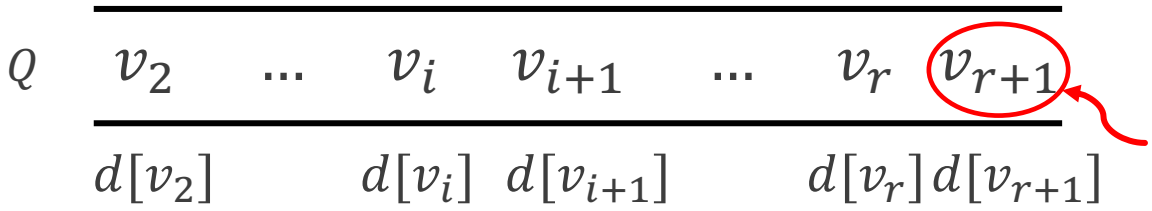


Proof (cont'd):

- $n = k + 1$, consider dequeuing v_1:
  - If the head v_1 of the queue is dequeued, v_2 becomes the new head.
  - By the inductive hypothesis, $d[v_1] \leq d[v_2]$, we have
    $$d[v_r] \leq d[v_1] + 1 \leq d[v_2] + 1.$$
  - Thus, (1) is maintained after dequeue. (2) is also maintained because there is no new vertex added into Q.

Correctness of BFS

Goal of proof:
(1) $d[v_r] \leq d[v_1] + 1$
(2) $d[v_i] \leq d[v_{i+1}]$



Proof (cont'd):

- $n = k + 1$, after dequeuing v_1, consider enqueuing v_{r+1}:
  - By BFS, we have $d[v_{r+1}] = d[v_1] + 1$.
  - By the inductive hypothesis, $d[v_1] \leq d[v_2]$, we have
    $$d[v_{r+1}] = d[v_1] + 1 \leq d[v_2] + 1.$$
    Thus, (1) is proved because v_2 and v_{r+1} are the new head and tail of Q.
  - By the inductive hypothesis, $d[v_r] \leq d[v_1] + 1$, we have
    $$d[v_r] \leq d[v_1] + 1 = d[v_{r+1}].$$
    Thus, (2) is maintained when v_{r+1} is enqueued.

Correctness of BFS

Corollary 8.1

Suppose that vertices v_i and v_j are enqueued during the execution of BFS, and that v_i is enqueued before v_j. Then $d[v_i] \leq d[v_j]$ at the time that v_j is enqueued.

- What does this Corollary say? We have $d[v_i] \leq d[v_j]$, if v_i is in front of v_j in Q.
  - It can be easily proved by $d[v_i] \leq d[v_{i+1}]$ for $i = 1,2, \dots, r-1$ in Lemma 8.3.

Correctness of BFS

Theorem 8.1 (Correctness of BFS)

Let $G = (V, E)$ be a directed or undirected graph, and suppose that BFS is run on G from a given source vertex $s \in V$.

(1) When BFS terminates, $d[v] = \delta(s, v)$ for all $v \in V$.

(2) For any vertex $v \neq s$ that is reachable from s, one of the shortest paths from s to v is a shortest path from s to $\pi[v]$ followed by the edge $(\pi[v], v)$.

- What does the Theorem say?

  (1) When BFS terminates, the number of edges from s to v is the shortest path distance.

  (2) One of the shortest path from s to v must go through v's parent $\pi[v]$.

Correctness of BFS

Proof of (1):

- We use proof by contradiction: Assume that there is a vertex v having $d[v] \neq \delta(s, v)$.
- By Lemma 8.2, $d[v] \geq \delta(s, v)$, and thus we have $d[v] > \delta(s, v)$.
- Let u be the vertex immediately preceding v on a shortest path from s to v, so that $\delta(s, v) = \delta(s, u) + 1$.
- By the choice of v, there must exists a u that $d[u] = \delta(s, u)$.
- Thus, we have

$$d[v] > \delta(s, v) = \delta(s, u) + 1 = d[u] + 1.$$

Correctness of BFS

Proof of (1) (cont'd):

- Base on the face that $d[v] > d[u] + 1$, we consider the cases when u is dequeued:
  - If v is White, then Line 15 sets $d[v] = d[u] + 1$.
  - If v is $\mathrm{DarkGray}$, then it was already removed from the queue and, by Corollary 8.1, we have $d[v] \leq d[u]$.
  - If v is Gray, then it was painted gray upon dequeuing some vertex w, which was removed from Q earlier than u and for which $d[v] = d[w] + 1$ and $d[w] \leq d[u]$ by Corollary 8.1. So we have $d[v] \leq d[u] + 1$.
- All three cases show contradiction to $d[v] > d[u] + 1$. Therefore, the assumption fails.

Correctness of BFS

Proof of (2):

- If $\pi[v] = u$, we have $\delta(s, u) = d[u]$ and $\delta(s, v) = d[v]$. Therefore, we get $d[v] = d[u] + 1$.
- We can thus find a path from s to u and from u to v.

Breadth-First Tree

- For a graph $G = (V, E)$ with source s, we define the predecessor subgraph of G as $G_\pi = (V_\pi, E_\pi)$, where

$$V_\pi = \{v \in V : \pi[v] \neq \text{NIL}\} \cup \{s\},$$

$$E_\pi = \{(\pi[v], v) : v \in V_\pi - \{s\}\}.$$

- The predecessor subgraph G_π is a breadth-first tree if V_π consists of the vertices reachable from s and, for all $v \in V_\pi$, there is a unique simple path from s to v in G_π that is also a shortest path from s to v in G.
  - Breadth-first tree is a spanning tree.

Computational Cost of BFS

- The for loop in Line 1 runs $|V|$ times.
- In the worst case, the while loop in Line 8 runs $|V|$ times and the for loop in Line 10 runs $|E|$ times.
- The complexity is $O(|V||E|)$.
- Will this worst case happen? Think of using amortized analysis.

```
BFS(G, s)
1  for each vertex u ∈ V − {s} do
2        color[u] ← White
3        d[u] ← ∞
4        π[u] ← NIL
5  color[s] ← Gray
6  d[s] ← 0; π[s] ← NIL; Q ← ∅
7  Enqueue(Q, s)
8  while Q ≠ ∅ do
9        u ←Dequeue(Q)
10       for each v ∈ Adj[u] do
11             if color[v] = White then
12                   color[v] ← Gray
13                   d[v] ← d[u] + 1
14                   π[v] ← u
15                   Enqueue(Q, v)
16       color[u] ← DarkGray
```

Computational Cost of BFS

- There are at most $|V|$ vertices in Q.
- For each vertex, for loop runs at most $Adj[u]$ times.
- Total cost from Line 8 to Line 16:

$$\sum_{u \in V} |Adj[u]| = O(|E|).$$

- Total cost for BFS: $O(|V| + |E|)$.

```
BFS(G, s)
1  for each vertex u ∈ V − {s} do
2        color[u] ← White
3        d[u] ← ∞
4        π[u] ← NIL
5  color[s] ← Gray
6  d[s] ← 0; π[s] ← NIL; Q ← ∅
7  Enqueue(Q, s)
8  while Q ≠ ∅ do
9        u ←Dequeue(Q)
10       for each v ∈ Adj[u] do
11           if color[v] = White then
12               color[v] ← Gray
13               d[v] ← d[u] + 1
14               π[v] ← u
15               Enqueue(Q, v)
16       color[u] ← DarkGray
```

Print Path of BFS

```
PrintPath(G, s, v)
1  if v = s then
2        print s
3  else if π[v] = NIL then
4        print "no path from" s "to" v "exists"
5        else PrintPath(G, s, π[v])
6             print v
```

GRAPH SEARCH

DEPTH-FIRST SEARCH

Depth-First Search

- Depth-first search (DFS) (深度优先搜索) is a recursive algorithm, which starts from a vertex v and recursively call to all v's neighbors until all vertices are visited.

Image source: https://www.banterly.net/2020/02/09/why-what-you-have-been-thaught-about-dfs-is-wrong-at-least-partially/

Depth-First Search

- DFS timestamps each vertex. Each vertex v has two timestamps:
  - The first timestamp $d[v]$ records when v is first discovered.
  - The second timestamp $f[v]$ records when the search finishes checking v's neighbors.

BFS vs. DFS

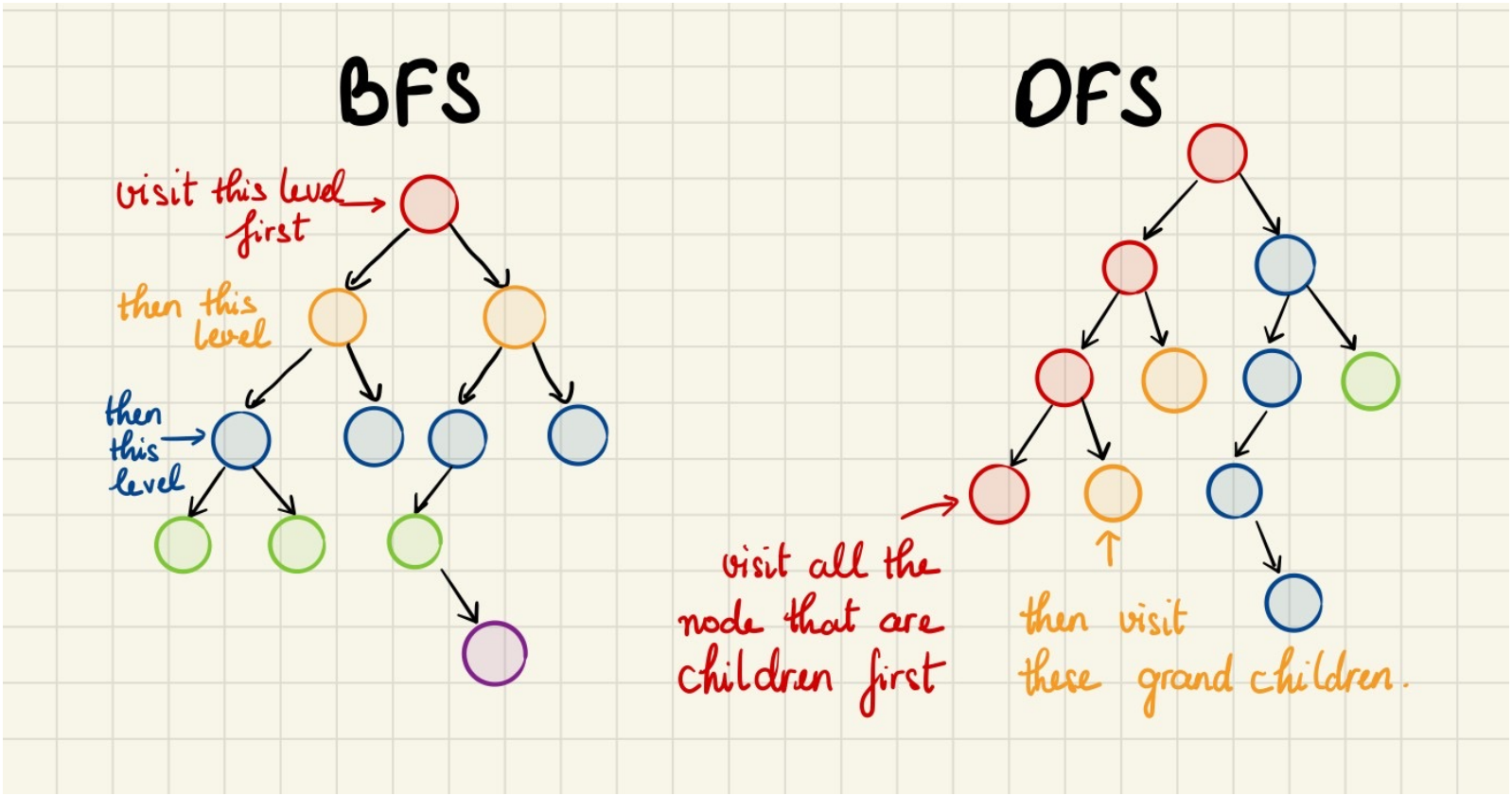


Image source: https://medium.com/nothingaholic/depth-first-search-vs-breadth-first-search-in-python-81521caa8f44

Depth-First Search

```
DFS(G)
1  for each vertex u ∈ V do
2      color[u] ← White
3      π[u] ← NIL
4  time ← 0
5  for each vertex u ∈ V do
6      if color[u] = White then
7          DFSVisit(u)
```

This DFS function search the whole graph, while the previous version of BFS only search from a source vertex s.

```
DFSVisit(u)
1   color[u] ← Gray
2   time ← time + 1
3   d[u] ← time
4   for each v ∈ Adj[u] do
5       if color[v] = White then
6           π[v] ← u
7           DFSVisit(v)
8   color[u] ← DarkGray
9   time ← time + 1
10  f[u] ← time
```

Similar to the analysis of BFS, the computational complexity is $O(|V|+|E|)$.

BFS vs. DFS

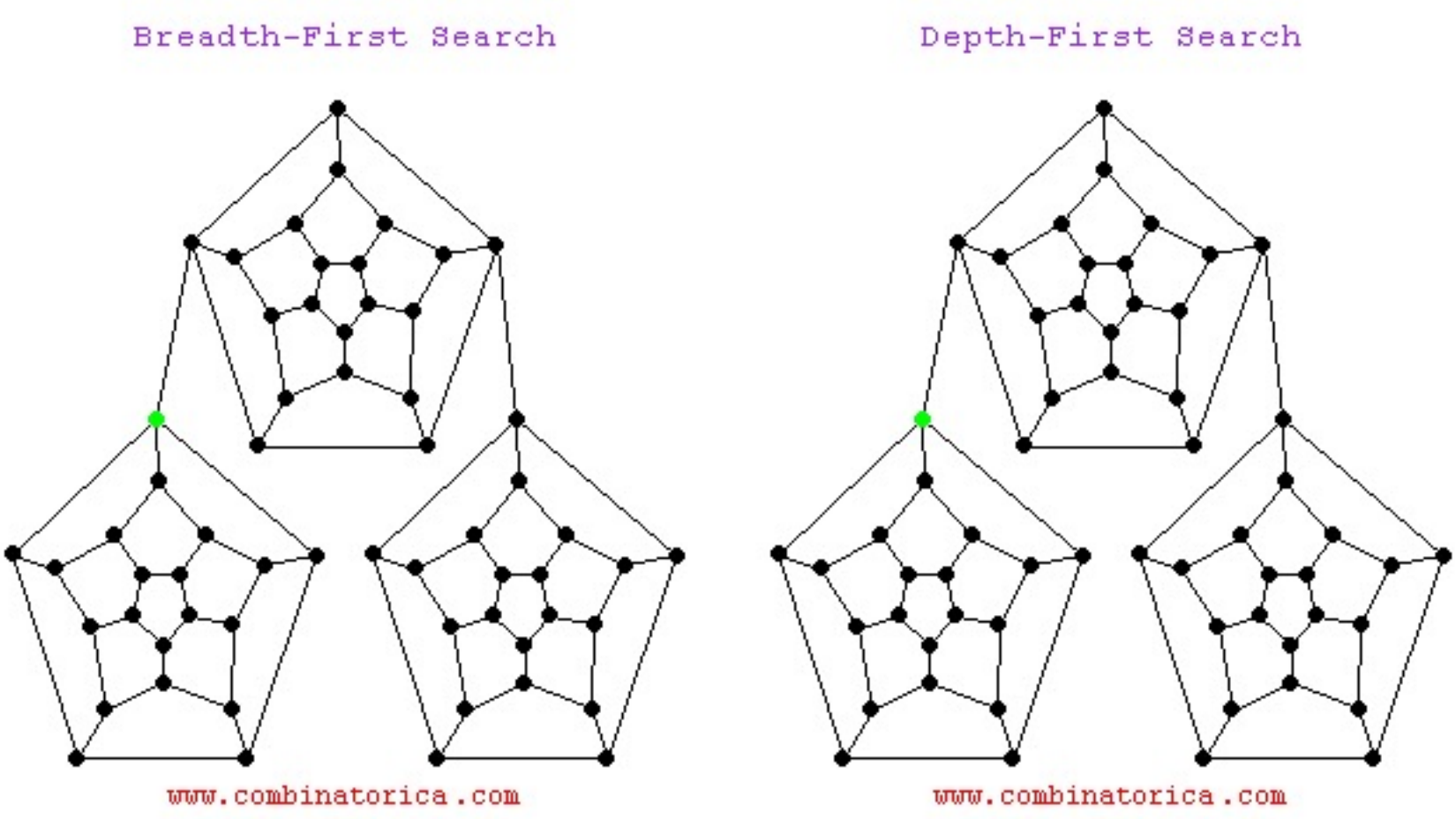
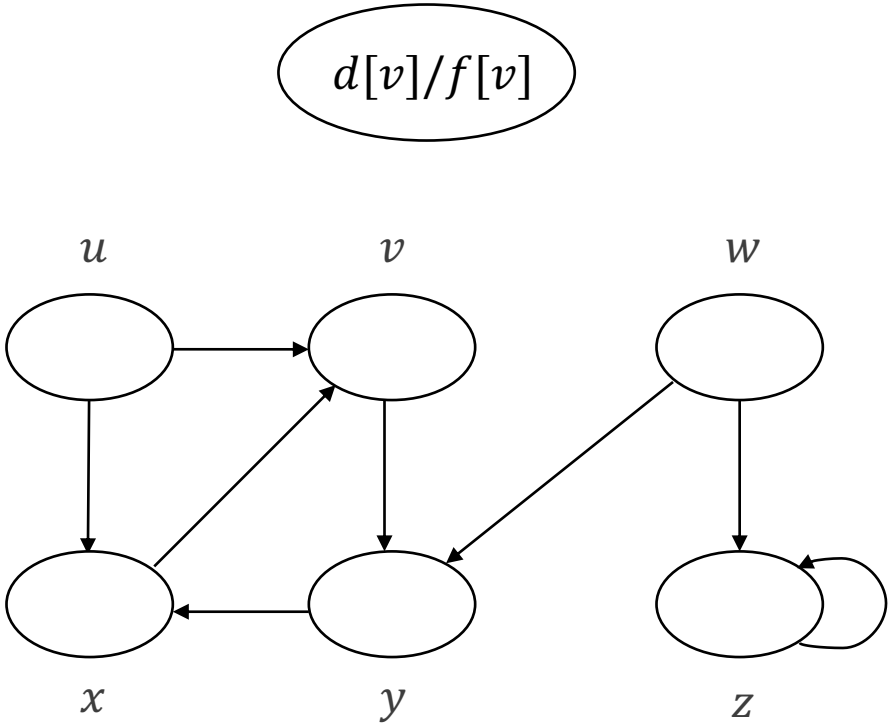
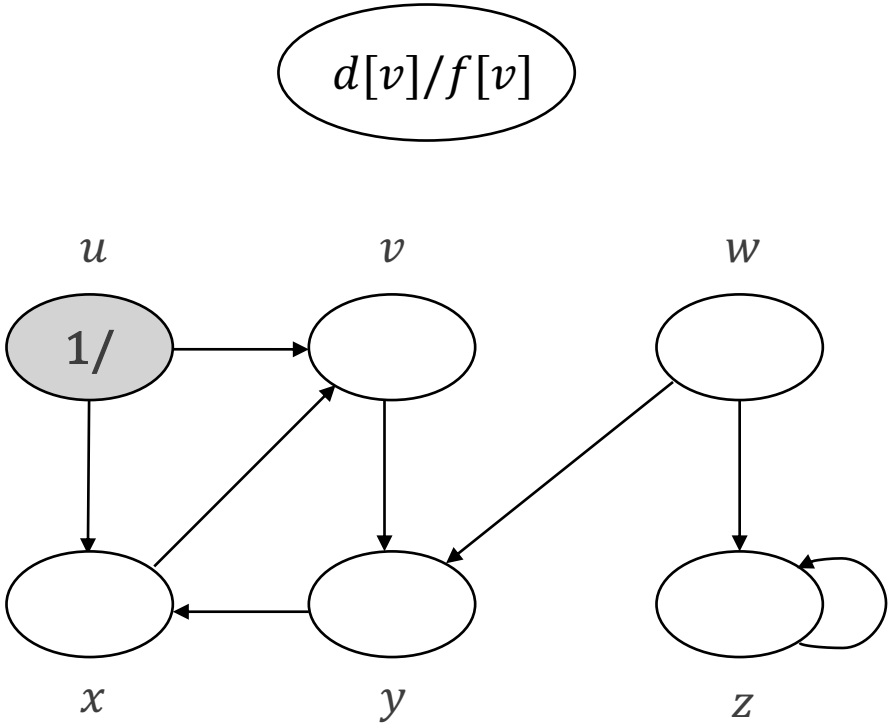


Image source: https://www3.cs.stonybrook.edu/~skiena/combinatorica/animations/search.html

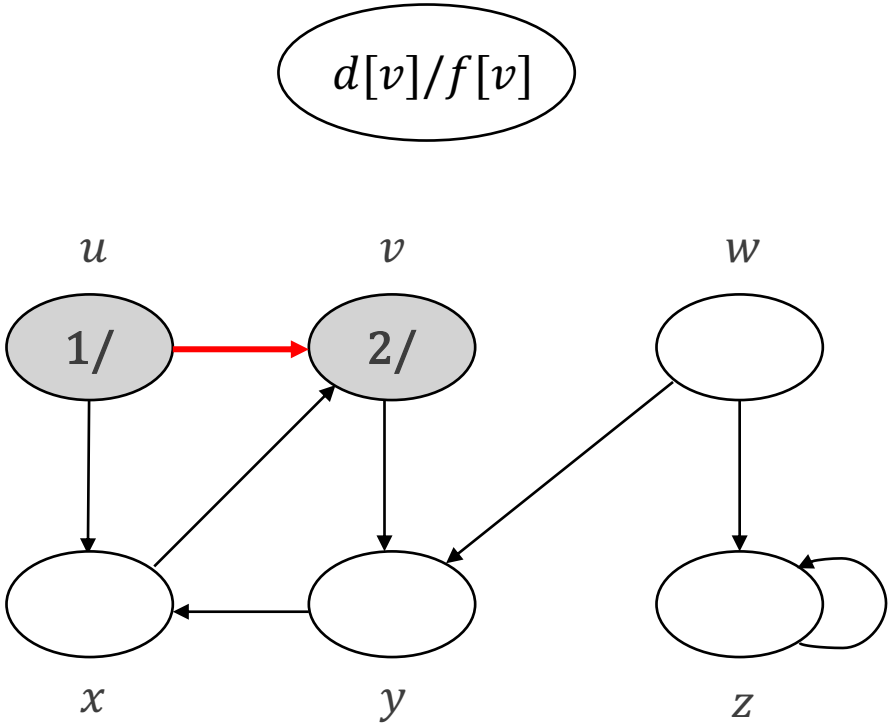
Depth-First Search



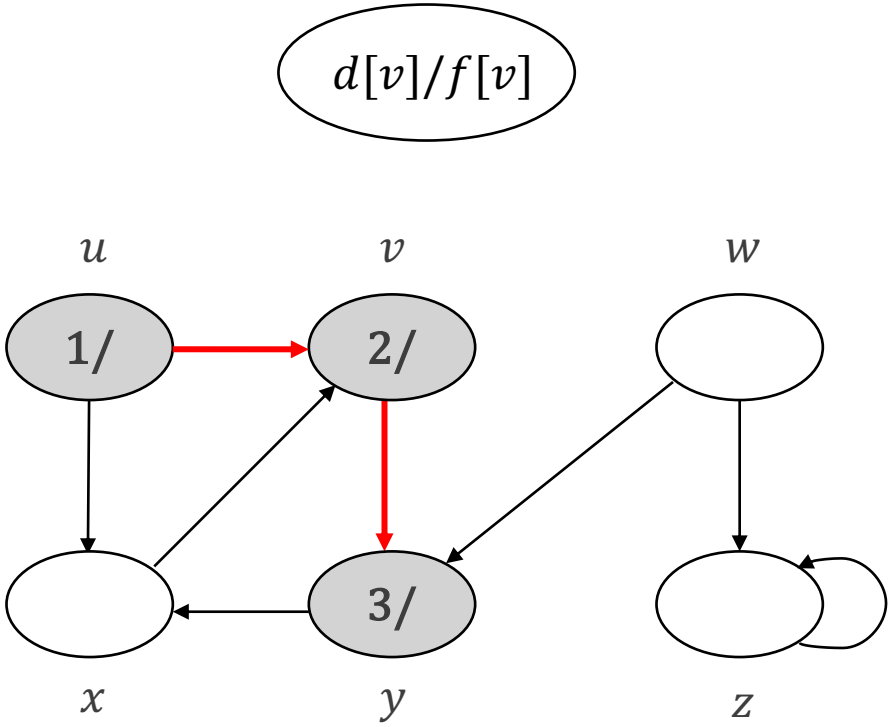
Depth-First Search



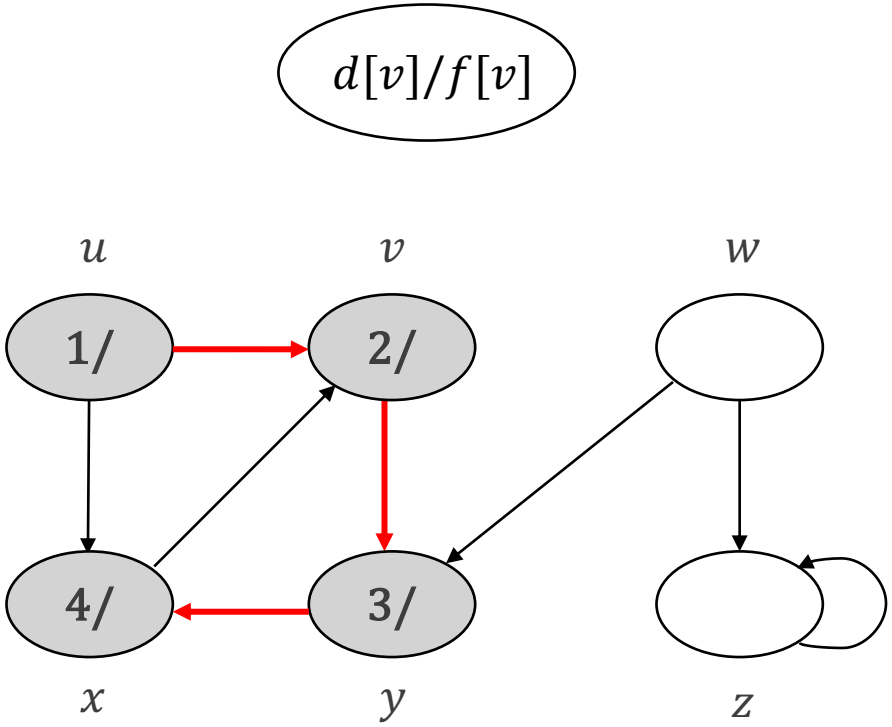
Depth-First Search



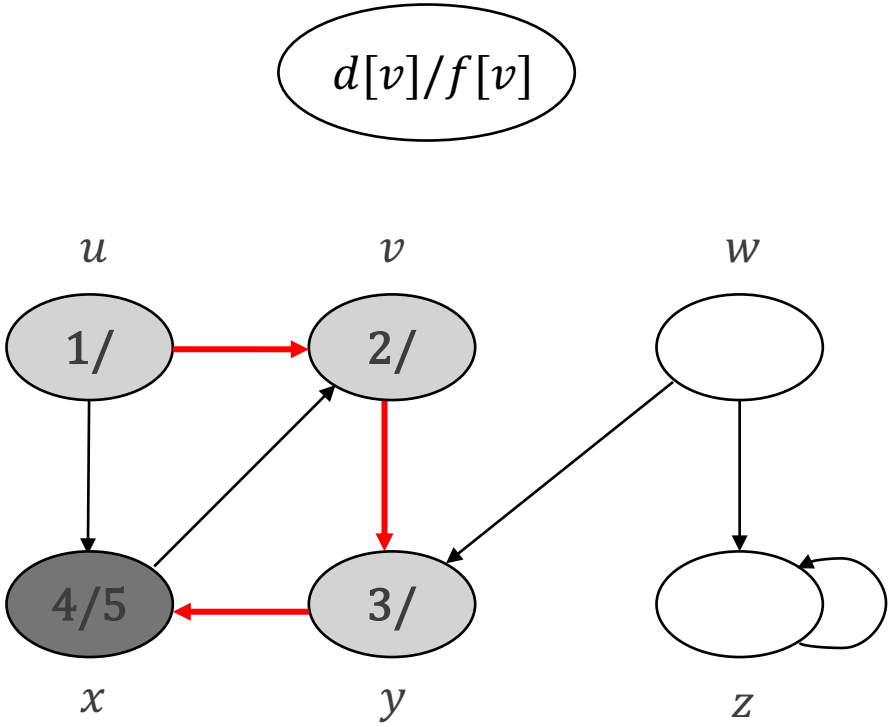
Depth-First Search



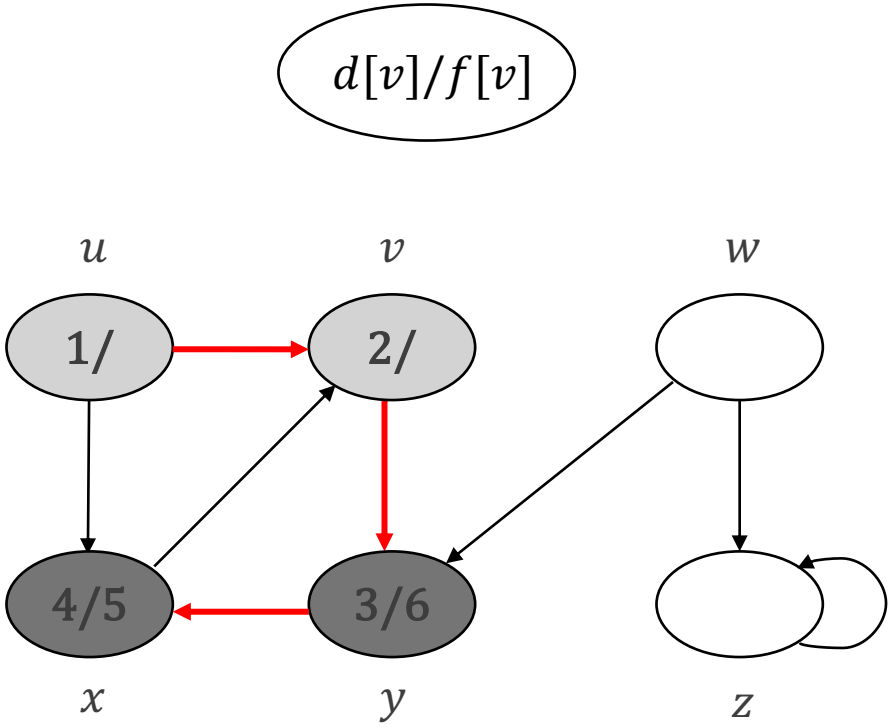
Depth-First Search



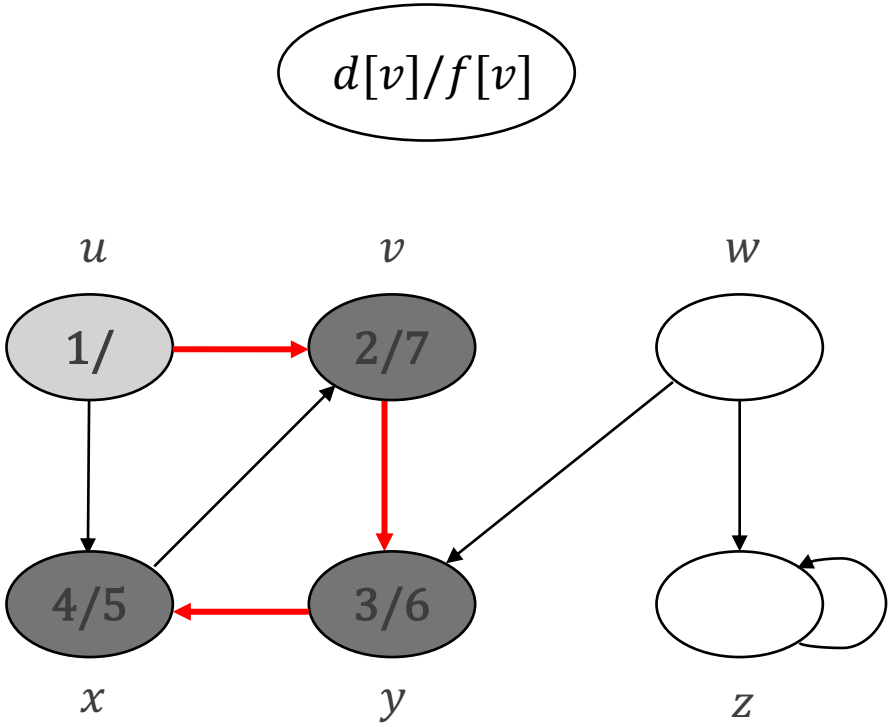
Depth-First Search



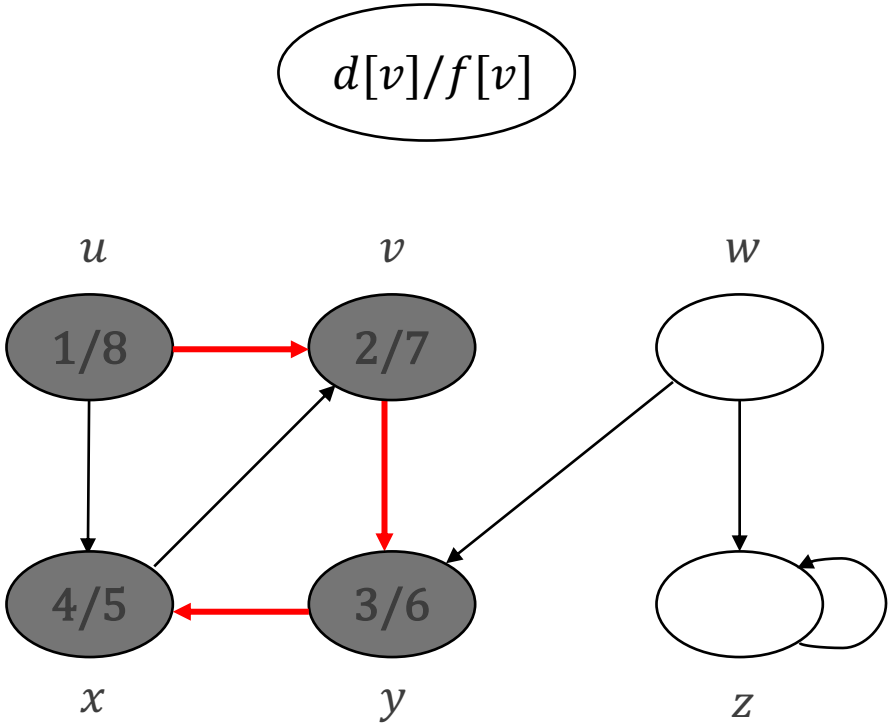
Depth-First Search



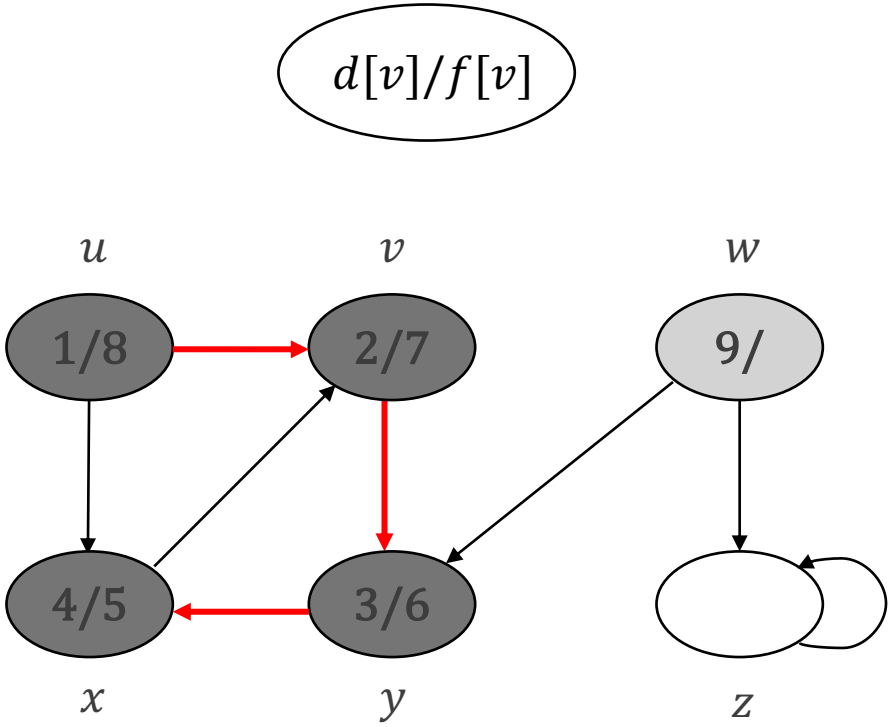
Depth-First Search



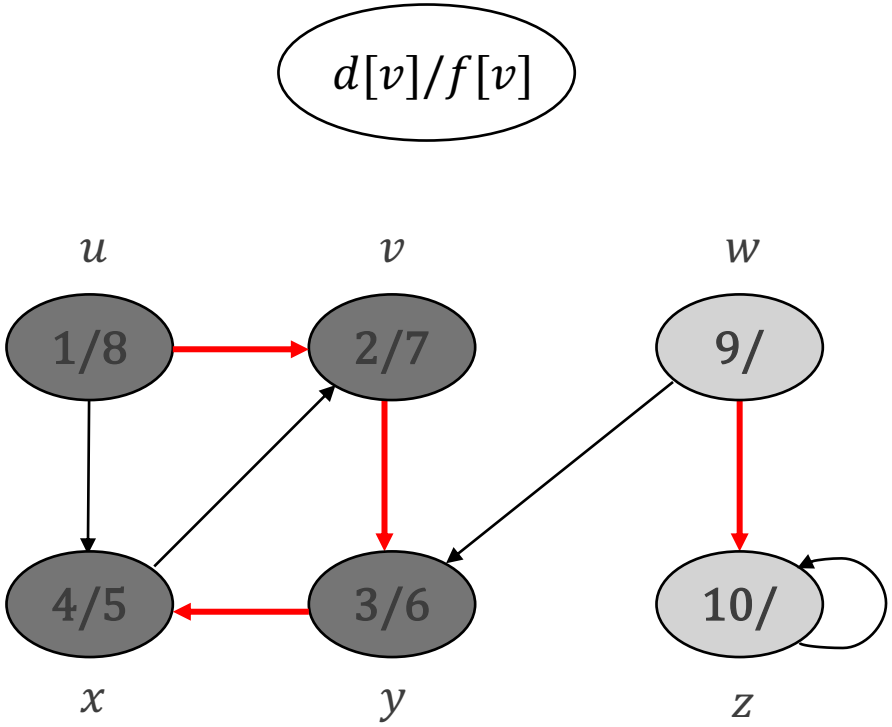
Depth-First Search



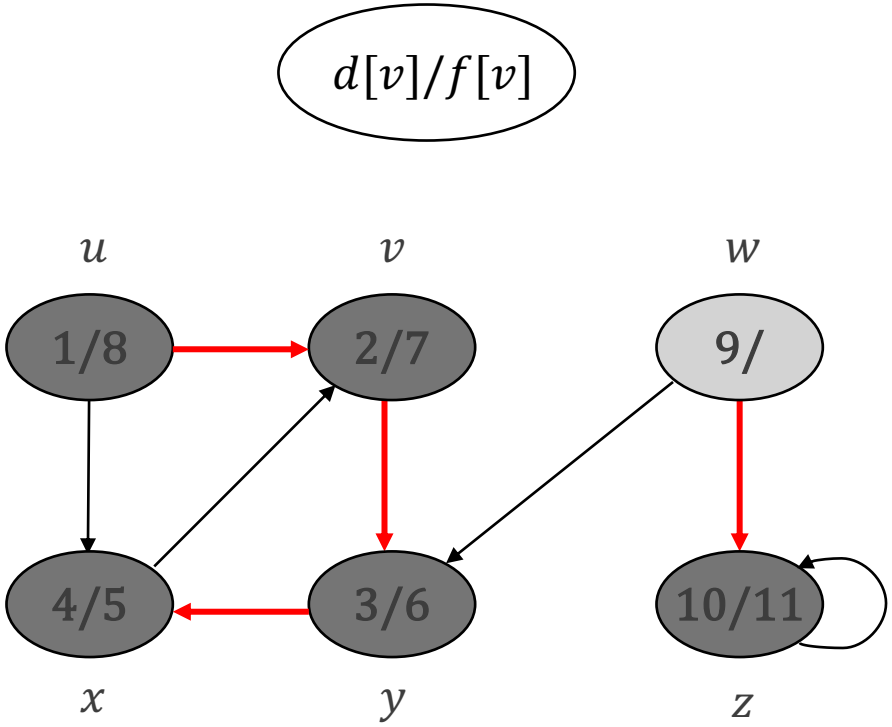
Depth-First Search



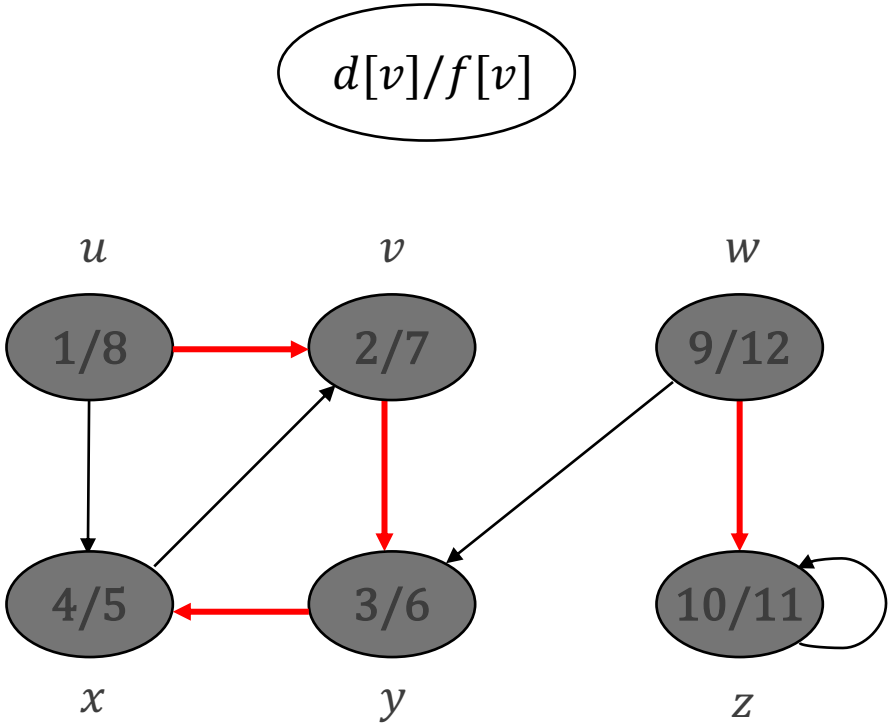
Depth-First Search



Depth-First Search

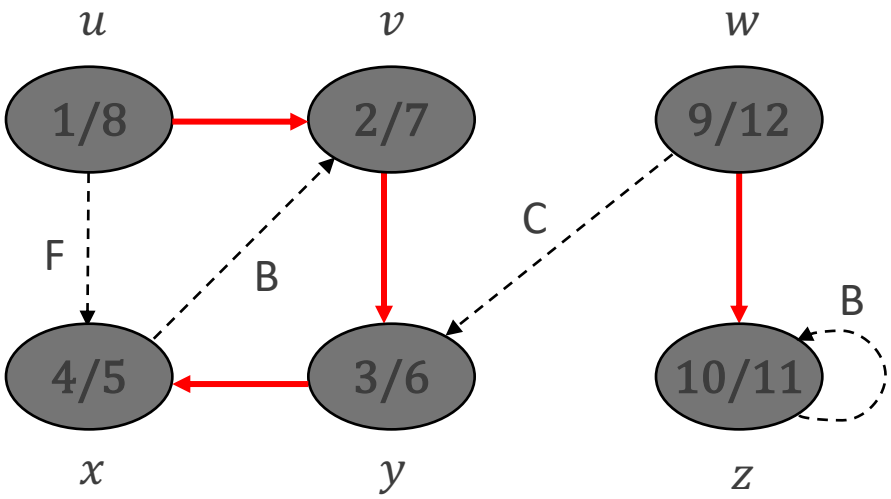


Depth-First Search



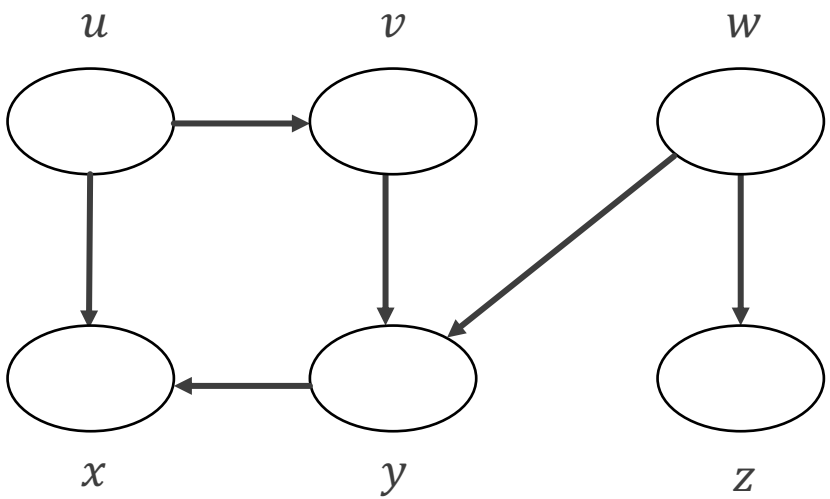
Properties of DFS

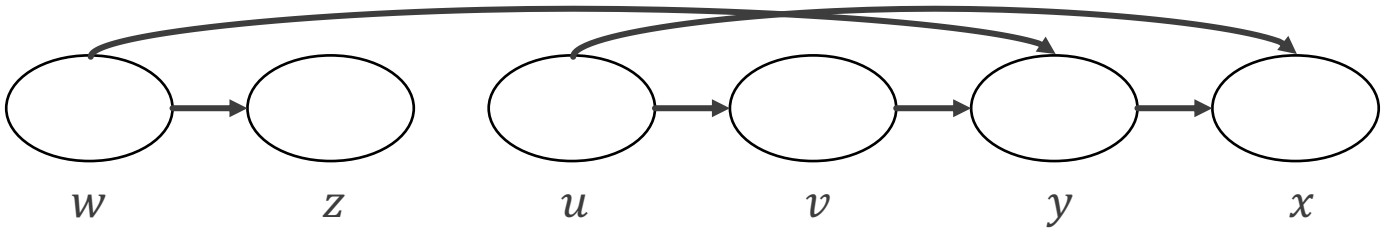
- The search can be used to classify the edges E into four edges:
  - Tree edges: Edges in depth-first trees.
  - Back edges (B): Not in tree edges, but points to its ancestor vertex or itself.
  - Forward edges (F): Not in tree edges, but points to its descendant vertex.
  - Cross edges (C): Others.



Application of DFS: Topological Sort

- A topological sort (拓扑排序) of a directed acyclic graph (DAG) (有向无环图) $G = (V, E)$ is a linear ordering of all its vertices such that if G contains an edge (u, v), then u appears before v in the ordering.

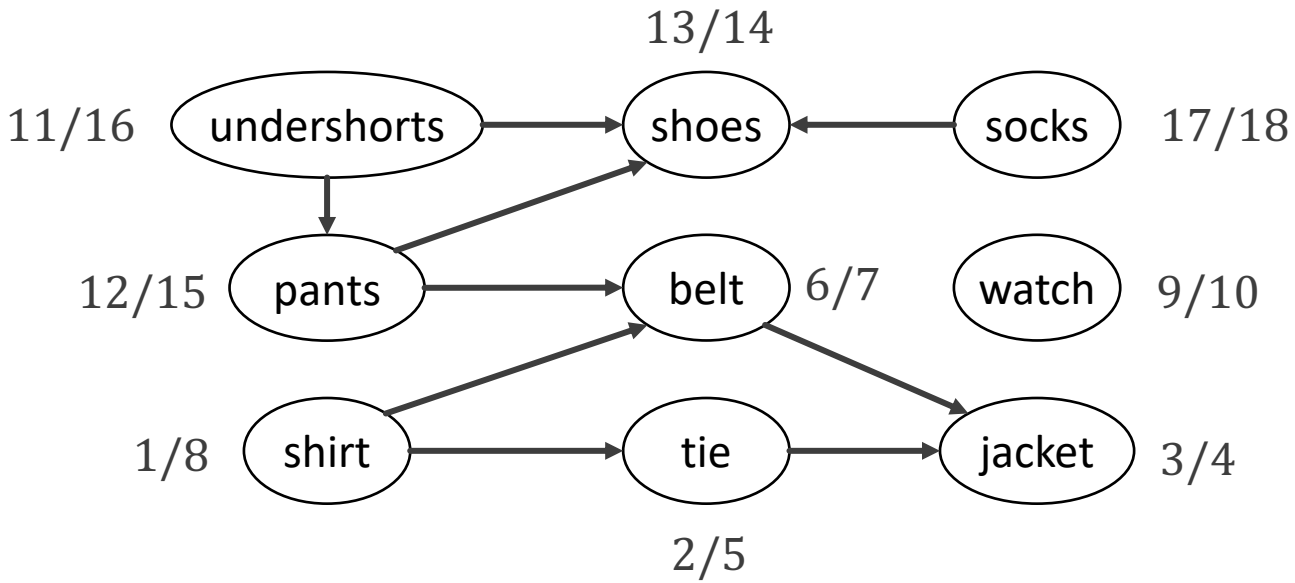


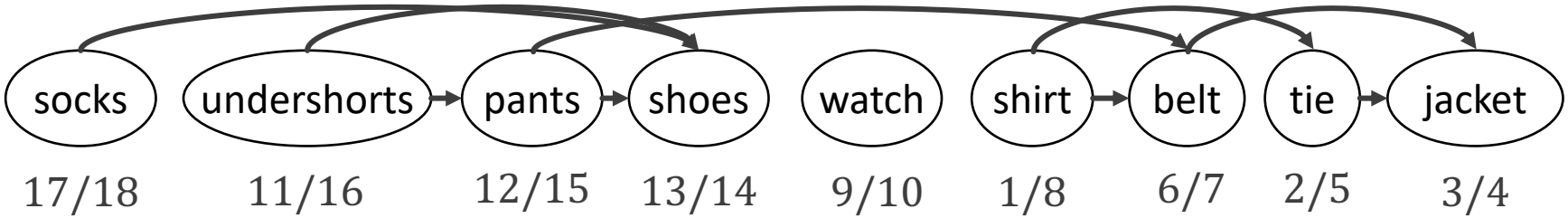


Application of DFS: Topological Sort

TopologicalSort(G)
1 call DFS(G) to compute finishing times $f[v]$ for each vertex v
2 as each vertex is finished, insert it onto the front of a linked list
3 **return** the linked list of vertices

Application of DFS: Topological Sort





Application of DFS: Strongly Connected Components

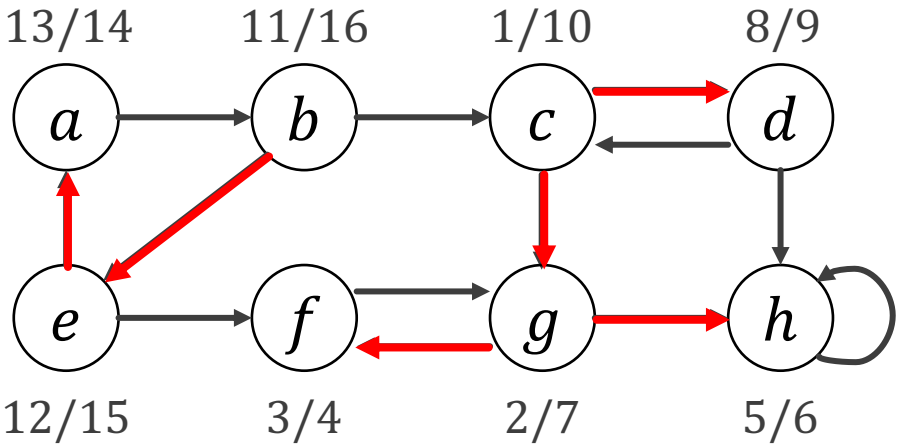
- A connected component (连通分支) of a directed graph $G = (V, E)$ is a set of vertices $C \subseteq V$ such that for every pair of vertices u and v in C, u and v are reachable from each other.
- A strongly connected component (强连通分支) is the maximal one among all connected components.
- The transpose graph (转置图) of G is the graph $G^T = (V, E^T)$, where $E^T = \{(u, v): (v, u) \in E\}$. That is, E^T consists of the edges of G with their directions reversed.

Application of DFS: Strongly Connected Components

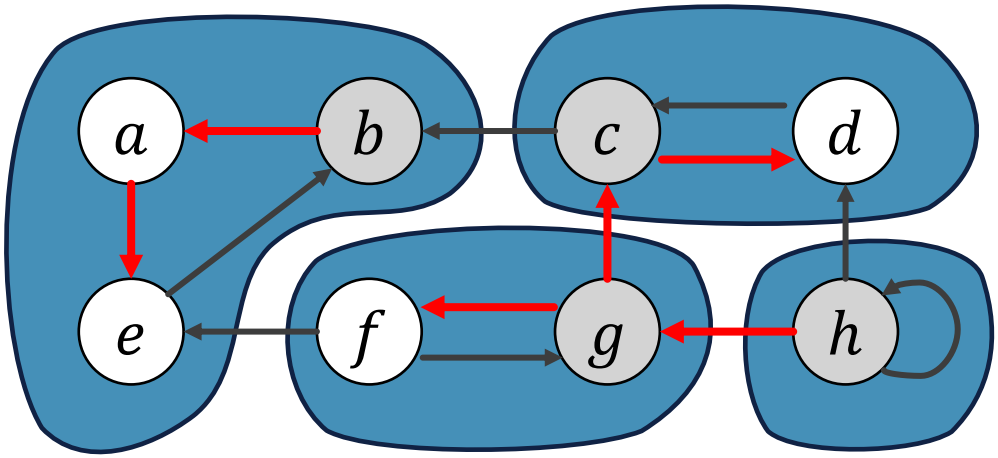
StrongConnectedComponents(G)
1 call DFS(G) to compute finishing times $f[u]$ for each vertex u
2 compute G^T
3 call DFS(G^T), but in the main loop of DFS, consider the vertices in order of decreasing $f[u]$ (as computed in line 1)
4 output the vertices of each tree in the depth-first forest formed in line 3 as a separate strongly connected component

DFS(G)
1 **for** each vertex $u \in V$ **do**
2 $color[u] \leftarrow$ White
3 $\pi[u] \leftarrow$ NIL
4 $time \leftarrow 0$
5 **for** each vertex $u \in V$ **do**
6 **if** $color[u] =$ White **then**
7 DFSVisit(u)

Application of DFS: Strongly Connected Components



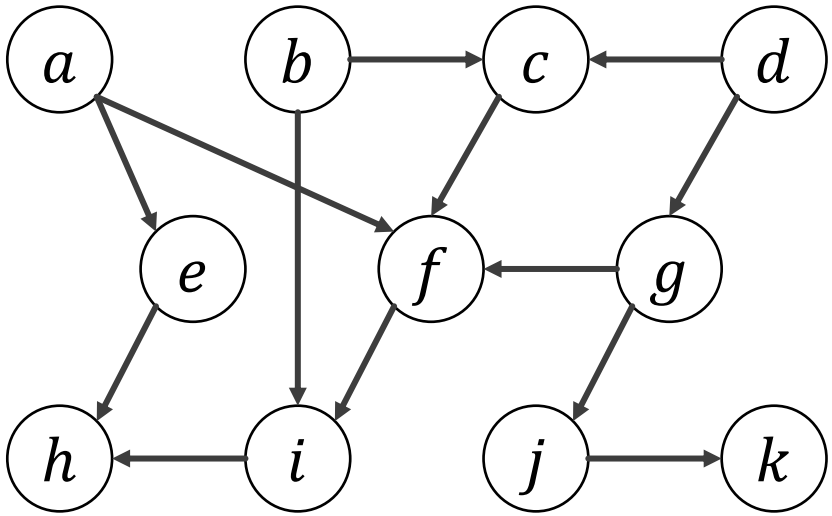
First DFS run.



Second DFS run. Each tree is a connected component with gray vertex as the root. The first tree is the strongly connected components.

Classroom Exercise

Show the ordering of vertices produced by TopologicalSort when it is run on the following DAG.



MINIMUM SPANNING TREE

Minimum Spanning Tree

- Given a connected, undirected graph $G = (V, E)$, for each edge $(u, v) \in E$, we have a weight $w(u, v)$ specifying the cost to connect u and v.
- An acyclic subset $T \subseteq E$ is called minimum spanning tree (MST) (最小生成树). It connects all of the vertices and whose total weight is minimized:

$$w(T) = \sum_{(u,v)\in T} w(u, v)$$

- Next, we introduce two greedy algorithms: Kruskal's algorithm and Prim's algorithm.

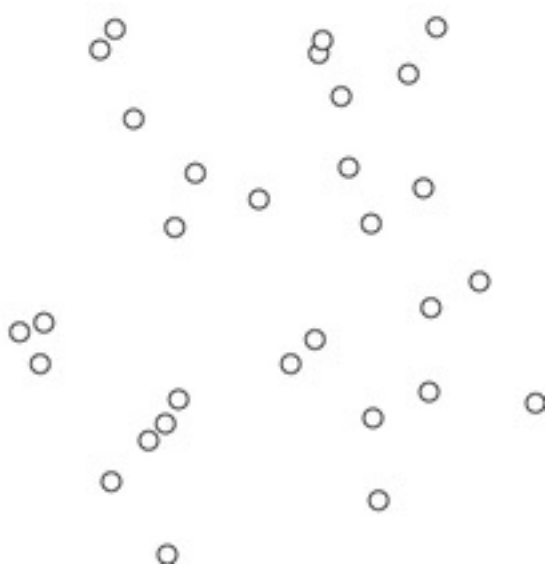
MINIMUM SPANNING TREE

KRUSKAL'S ALGORITHM

Kruskal's Algorithm

High level pseudocode:

- $A = \emptyset$.
- Sort the edges in E in nondecreasing order.
- Iterate over each e in E.
  - If $A \cup e$ do not form a cycle, $A = A \cup e$.



A demo for Kruskal's algorithm on a complete graph with weights based on Euclidean distance.

Image source: https://en.wikipedia.org/wiki/Kruskal%27s_algorithm

Pseudocode

```
KruskalMST(G, w)
1  A ← ∅
2  for each vertex v ∈ V do
3       MakeSet(v)
4  sort the edges of E by weight w
5  for each edge (u, v) ∈ E do
6       if FindSet(u) ≠ FindSet(v) then
7            A ← A ∪ {(u, v)}
8            Union(u, v)
9  return A
```

$O(|V|)$

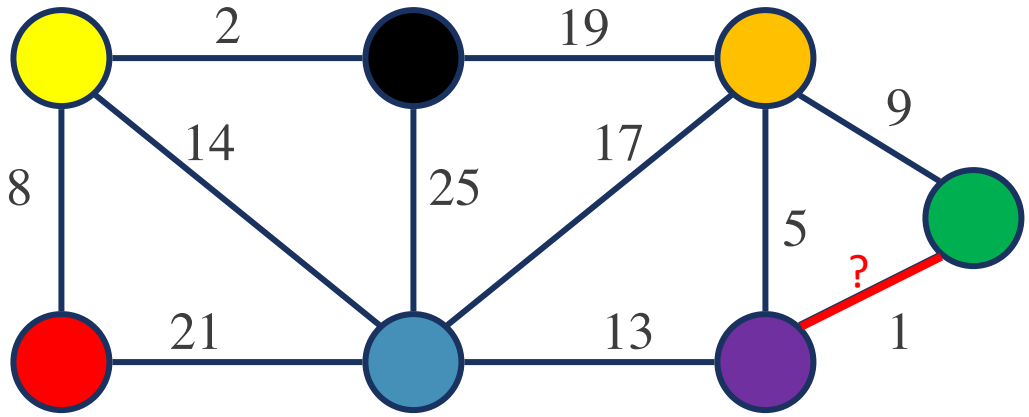
$O(|E| \lg |E|)$

$O(|E| \lg |E|)$

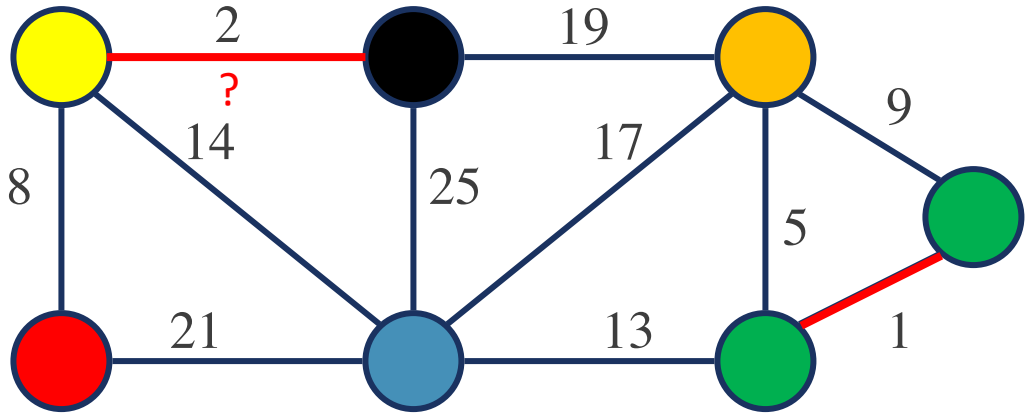
Total: $O(|E| \lg |E|)$

FindSet and Union are both $O(\lg |E|)$

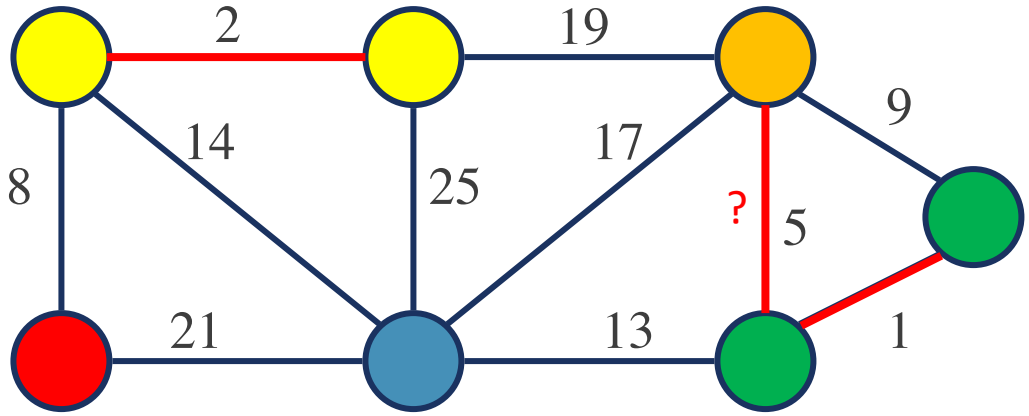
Example



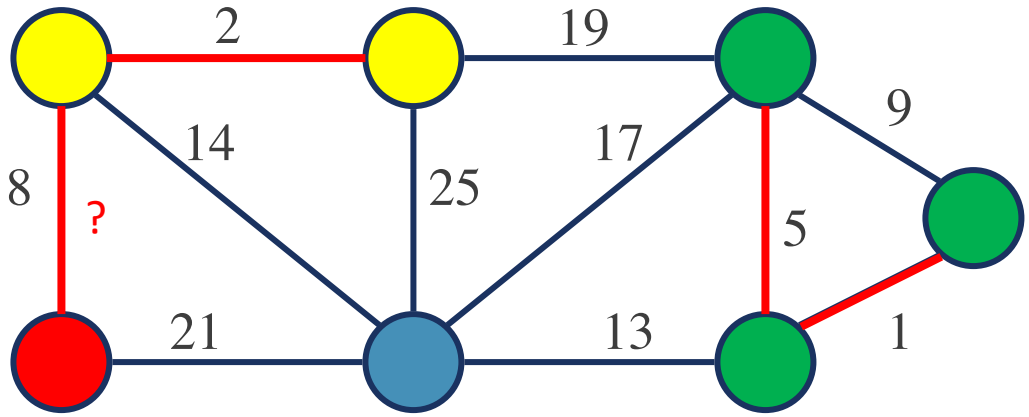
Example



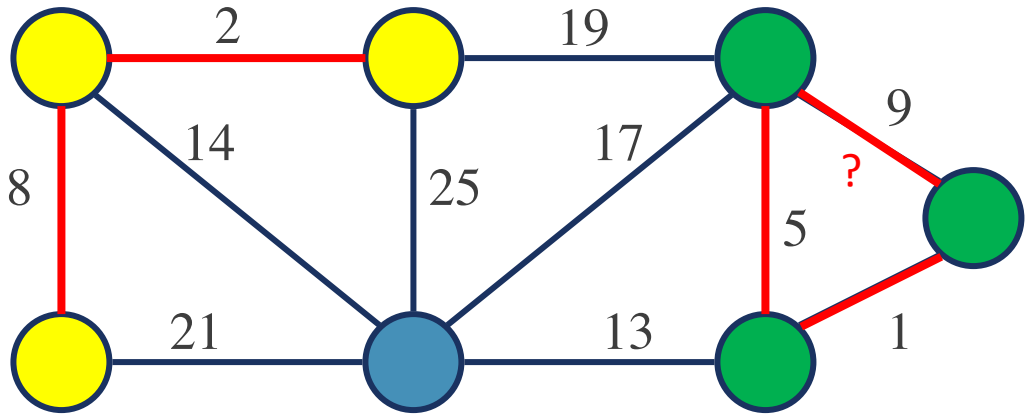
Example



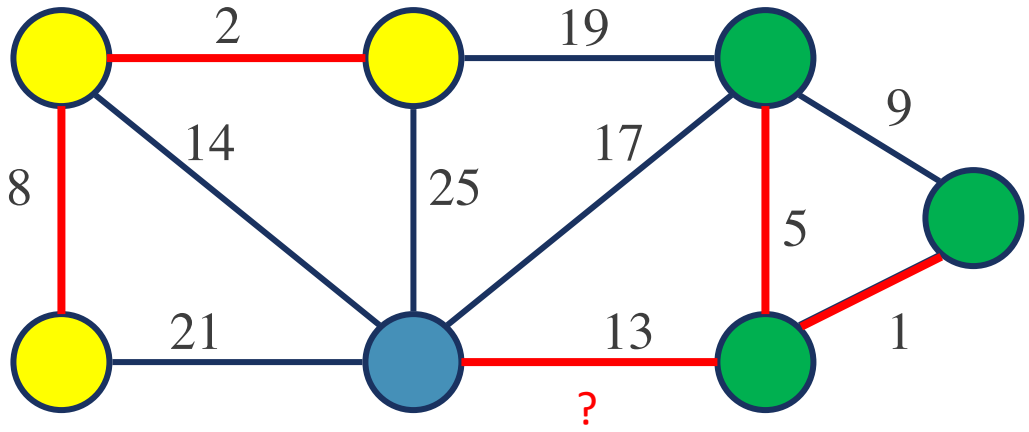
Example



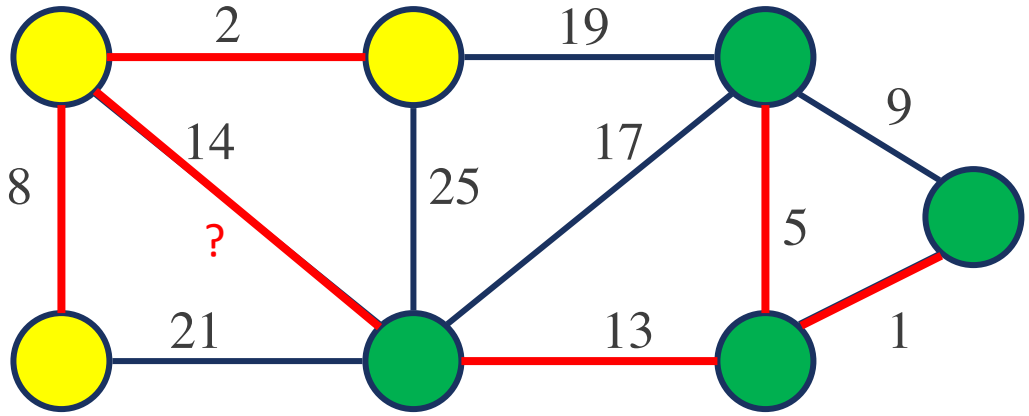
Example



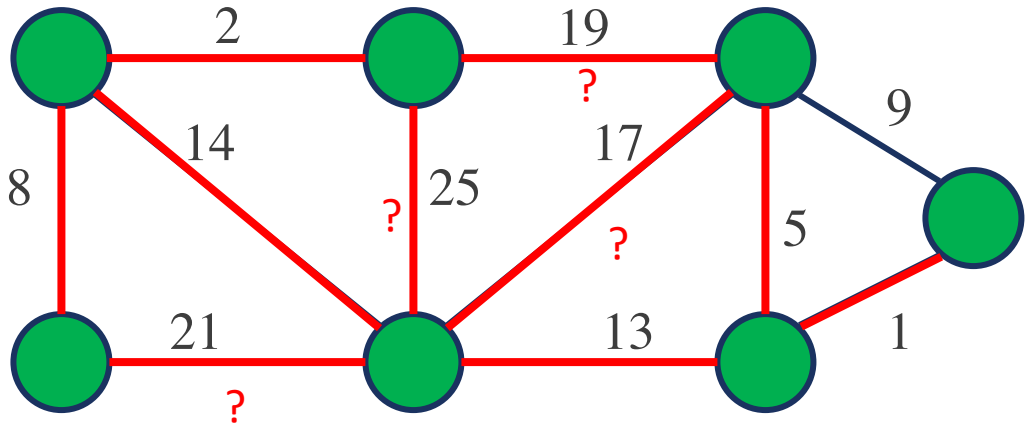
Example



Example



Example



Correctness Proof of Kruskal's Algorithm

Theorem 8.4

Algorithm KruskalMST correctly finds a minimum spanning tree in a weighted connected undirected graph.

Proof:

- We prove that the edge selected by the greedy choice at each step must be in a MST.
- Let T a minimum spanning tree and A^* is the set of edges in T.
- We add an edge to set A at each step. We need to prove the loop invariant $A \subseteq A^*$ at each step.

Correctness Proof of Kruskal's Algorithm

Proof (cont'd):

Initialization:

- After line 1, the set $A = \emptyset$ trivially satisfies the loop invariant $A \subseteq A^*$.

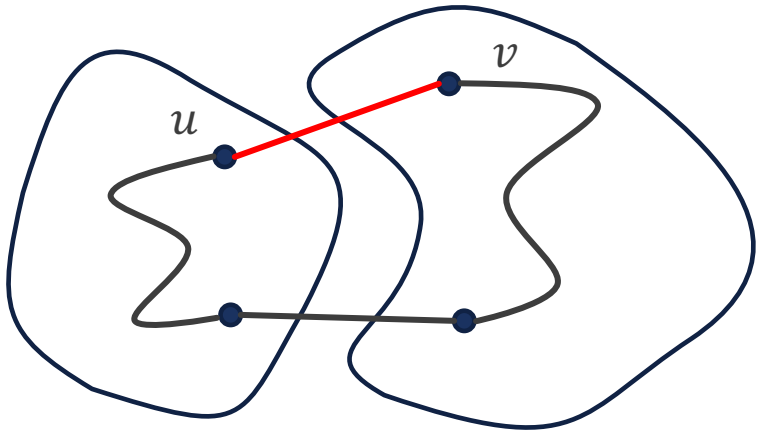
Maintenance:

- Induction hypothesis : Before adding the edge (u, v) to A, $A \subseteq A^*$.
- We will show that after adding the edge (u, v) to A, e.g. $A' = A \cup (u, v)$, $A' \subseteq A^*$.
- By the induction hypothesis, $A \subseteq A^*$. If A^* contains (u, v), then there is nothing to prove because obviously $A \cup (u, v) \subseteq A^*$.

Correctness Proof of Kruskal's Algorithm

Proof (cont'd):

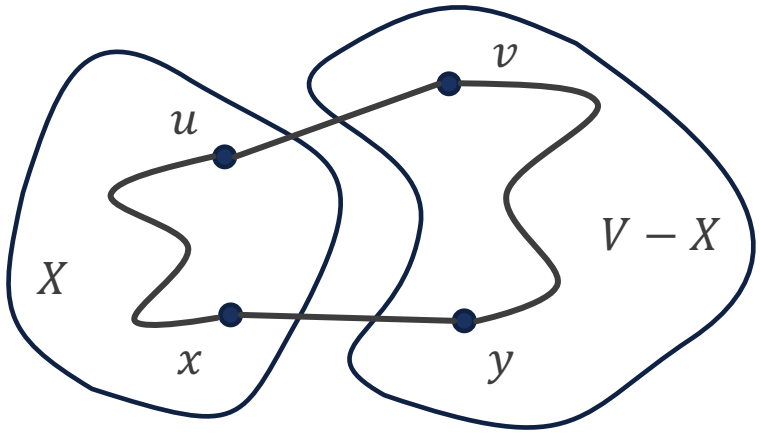
- If A^* does not contain (u, v), $A^* \cup (u, v)$ must have a cycle.
  - Because T is connected and thus there exists a path from u to v. Adding (u, v) provides another path from u to v that forms a cycle.



Correctness Proof of Kruskal's Algorithm

Proof (cont'd):

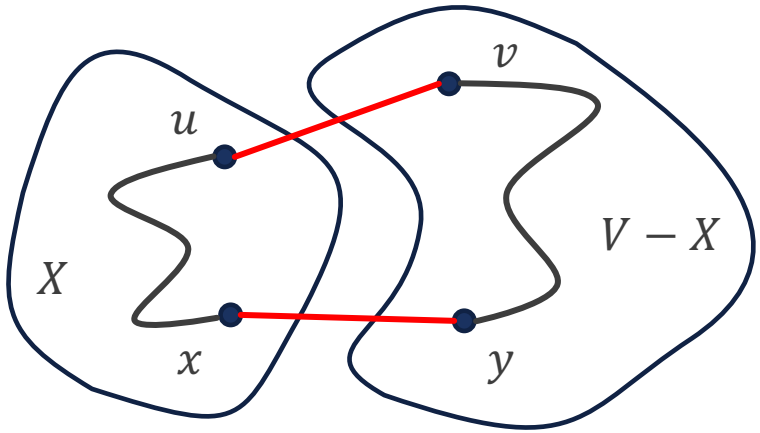
- Let X be the vertex set of the current subtree containing u.
- If A^* does not contain (u, v), A^* must have an edge (x, y) connects X and $V - X$, where $x \in X$ and $y \in V - X$.



Correctness Proof of Kruskal's Algorithm

Proof (cont'd):

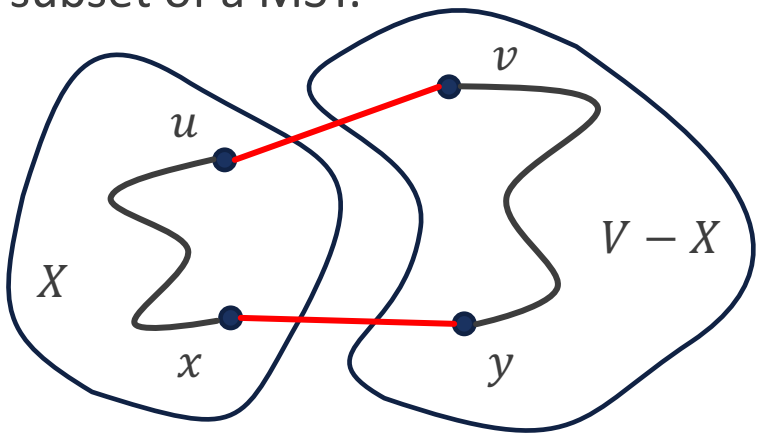
- (u, v) is selected by KruskalMST at this step, which means X and $V - X$ is not connected. Therefore, we must have $w(u, v) \leq w(x, y)$.
  - If $w(u, v) > w(x, y)$, KruskalMST will select (x, y) first and won't select (u, v) this time to avoid cycle.



Correctness Proof of Kruskal's Algorithm

Proof (cont'd):

- Now, if we construct $A^{**} = \{A^* - (x, y)\} \cup (u, v)$, then A^{**} may have smaller total weights because $w(u, v) \leq w(x, y)$.
- However, A^* is optimal which means it is impossible that $w(u, v) < w(x, y)$. The only possibility is $w(u, v) = w(x, y)$, which makes both A^* and A^{**} optimal.
- Because $(u, v) \in A^{**}$, now we have $A' \subseteq A^*$. That means adding (u, v) by $\mathrm{KruskalMST}$ is still a subset of a MST.



Correctness Proof of Kruskal's Algorithm

Proof (cont'd):

Termination:

All edges added to A are in a minimum spanning tree, and so the set A is returned in Line 9 must be a minimum spanning tree.

```
KruskalMST(G, w)
1  A ← ∅
2  for each vertex v ∈ V do
3       MakeSet(v)
4  sort the edges of E by weight w
5  for each edge (u, v) ∈ E do
6       if FindSet(u) ≠ FindSet(v) then
7            A ← A ∪ {(u, v)}
8            Union(u, v)
9  return A
```

MINIMUM SPANNING TREE

PRIM'S ALGORITHM

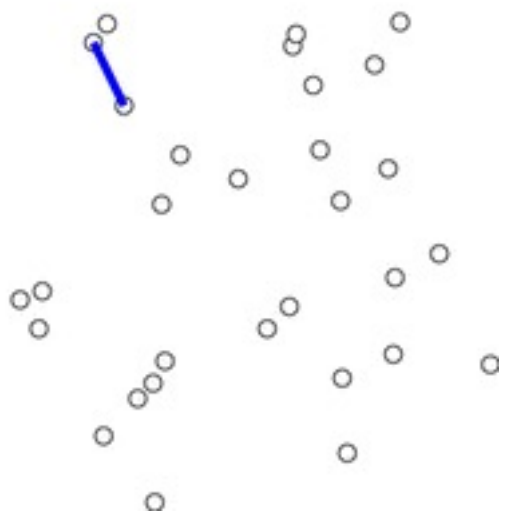
Prim's Algorithm

- Prim's algorithm has the property that the edges in the set A always form a single tree.
  - In Kruskal's algorithm, edges in the set A may not be a tree.
- The tree starts from an arbitrary root vertex r and grows until the tree spans all the vertices in V.

Prim's Algorithm

High level pseudocode:

- Initialize $A = \emptyset$ and $X = \{v_1\}$.
- Iterate when the instance is not solved:
  - Select a vertex in $V - X$ that is nearest to X.
  - Add the vertex to X.
  - Add the edge to A.



A demo for Prim's algorithm on a complete graph with weights based on Euclidean distance.

Image source: https://en.wikipedia.org/wiki/Prim%27s_algorithm

Pseudocode

```
PrimMST(G, w, r)
1  for each u ∈ V do
2      key[u] ← ∞
3      π[u] ← NIL
4  key[r] ← 0
5  Q ← V
6  while Q ≠ ∅ do
7      u ← ExtractMin(Q)
8      for each v ∈ Adj[u] do
9          if v ∈ Q and w(u, v) < key[v] then
10             π[v] ← u
11             key[v] ← w(u, v)
12             DecreaseKey(Q, v, key)
```

$O(|V|)$

$O(\lg |V|)$

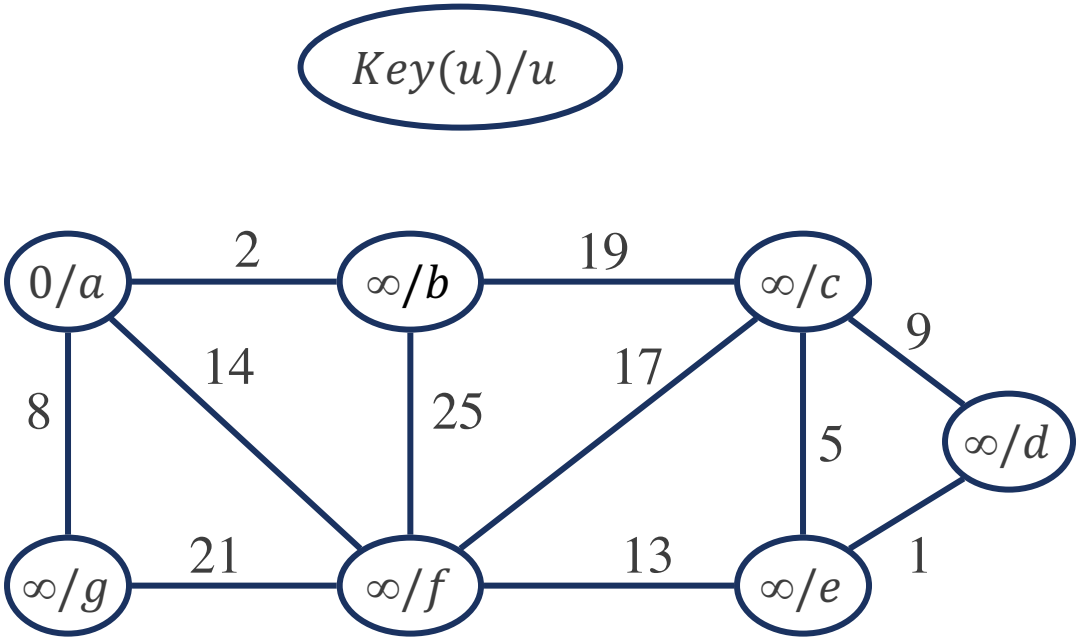
$O(\lg |V|)$

Total: $O(|E| \lg |V|)$

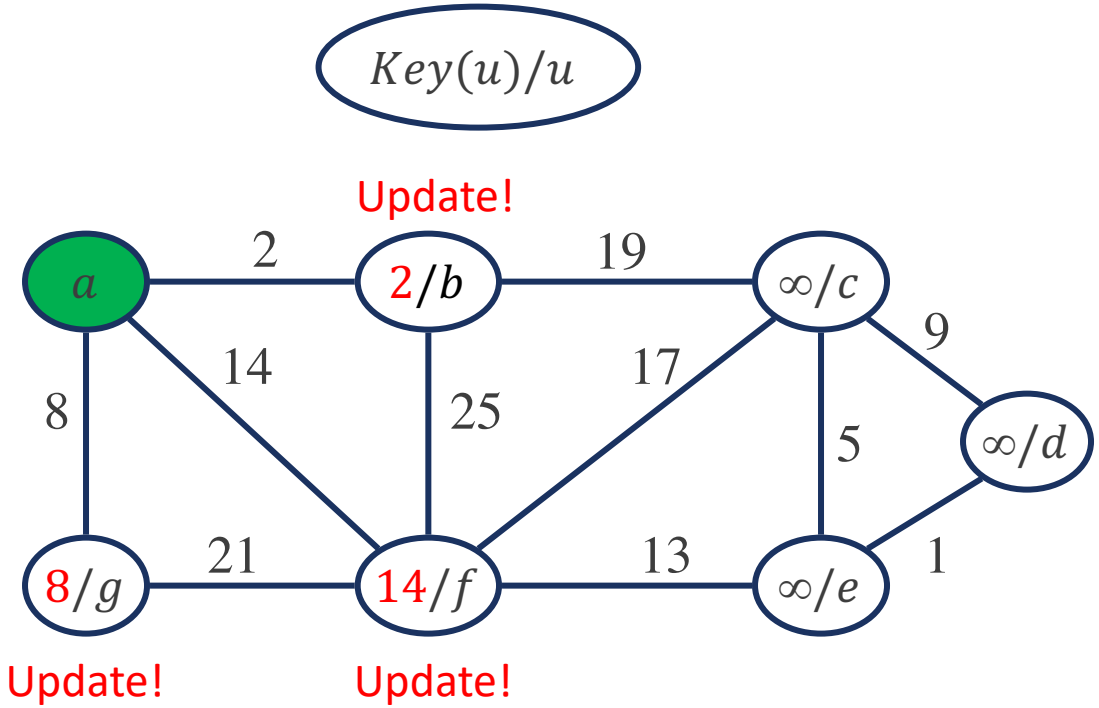
$O(|E| \lg |V|)$

By amortized analysis as BFS

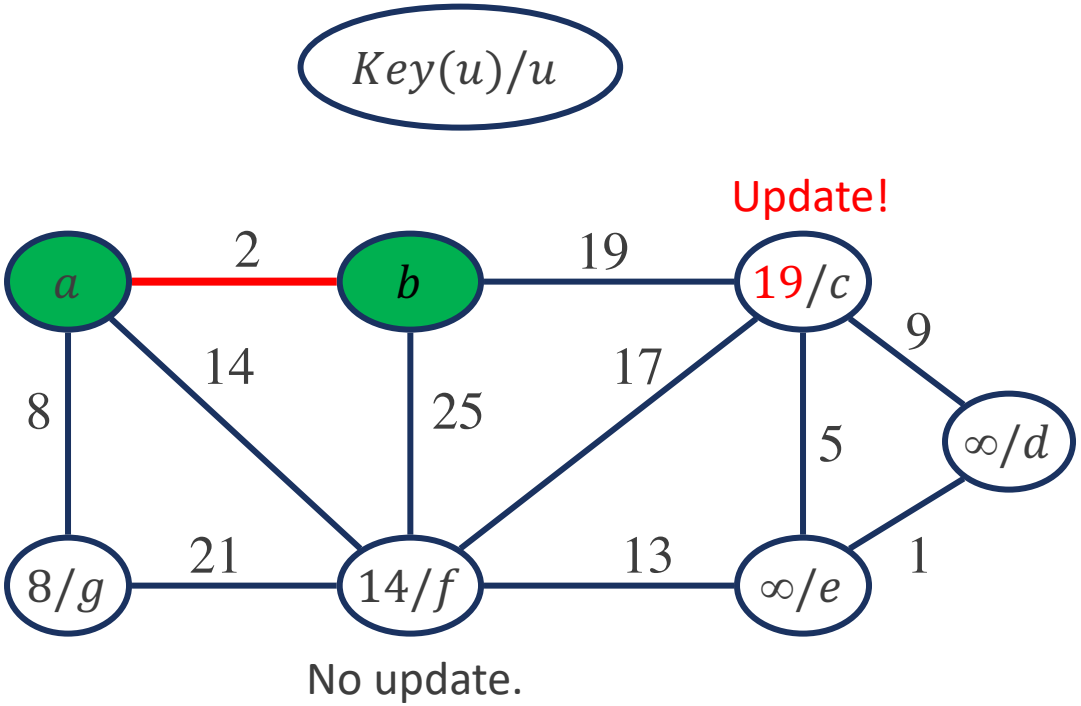
Example



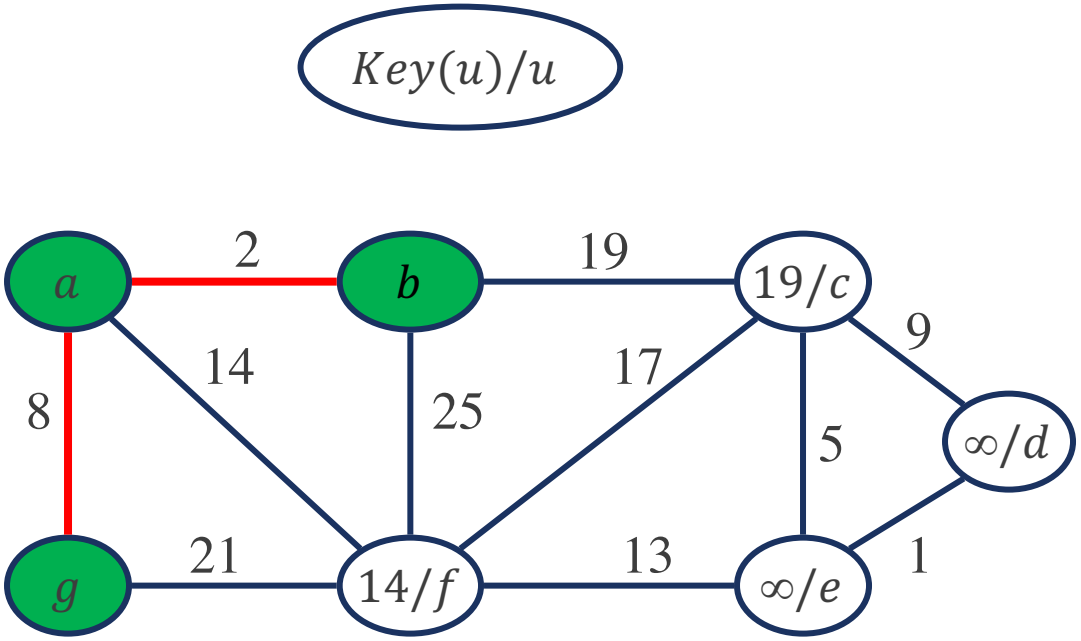
Example


Key(u)/u
Update!
a
2
2/b
19
∞/c
9
14
17
8
25
5
∞/d
21
13
1
8/g
14/f
∞/e
Update!
Update!

Example

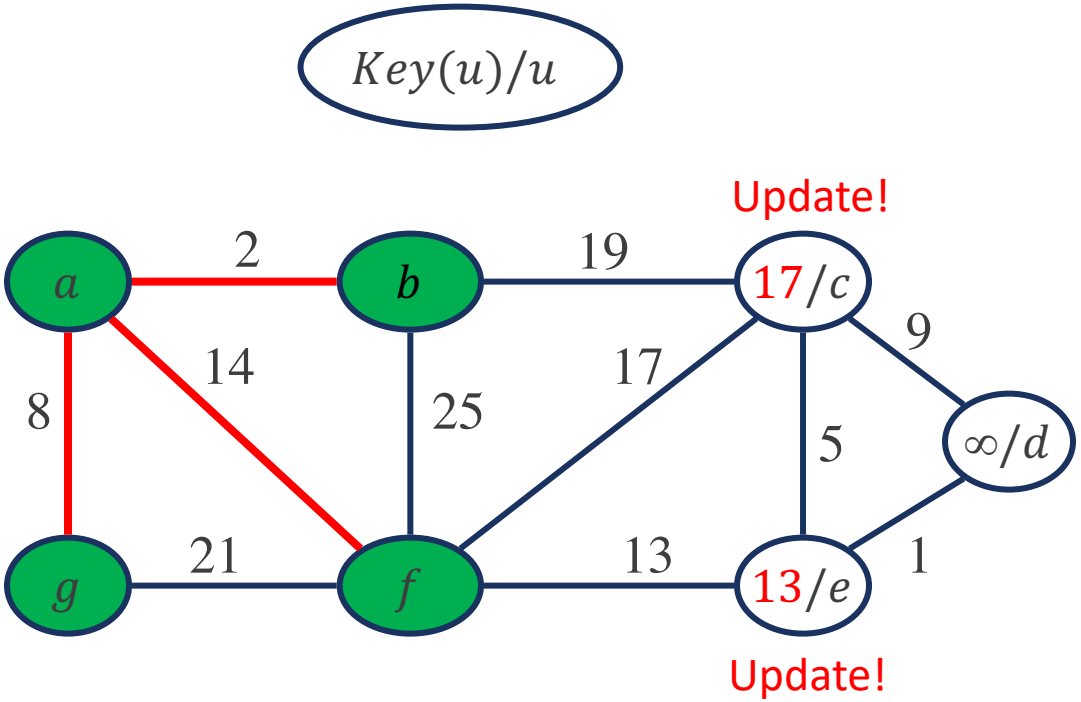


Example

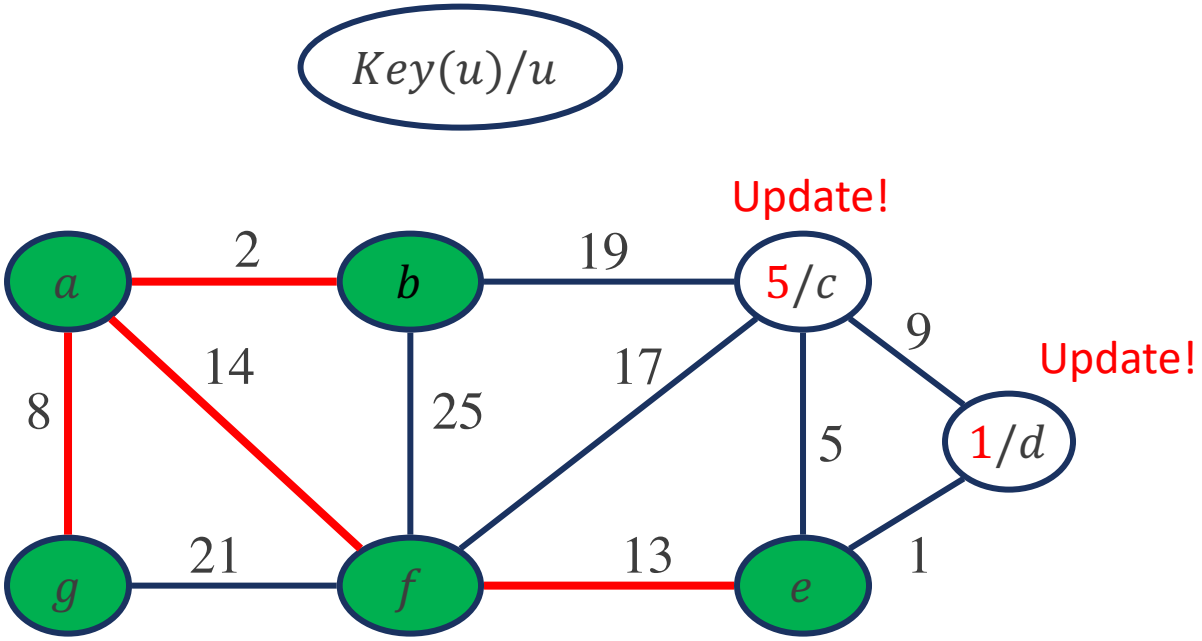


No update.

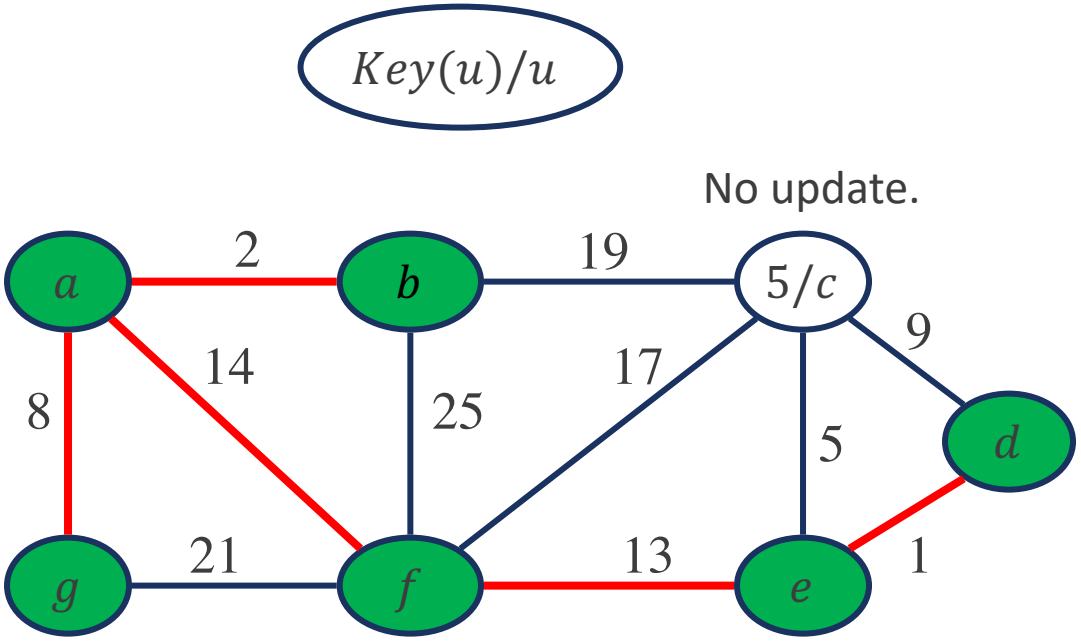
Example



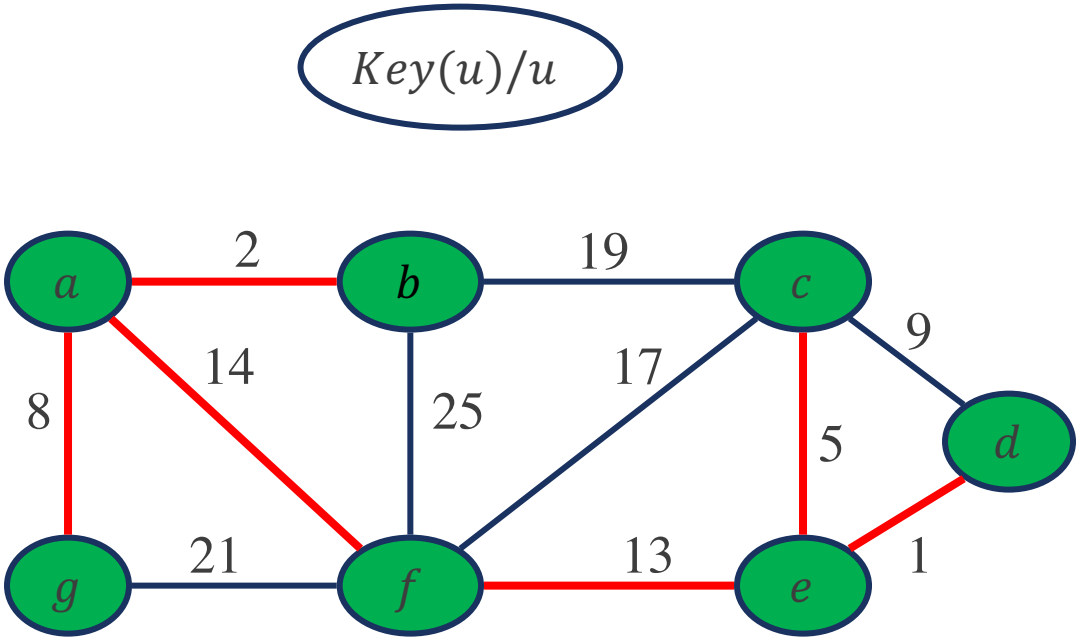
Example



Example



Example

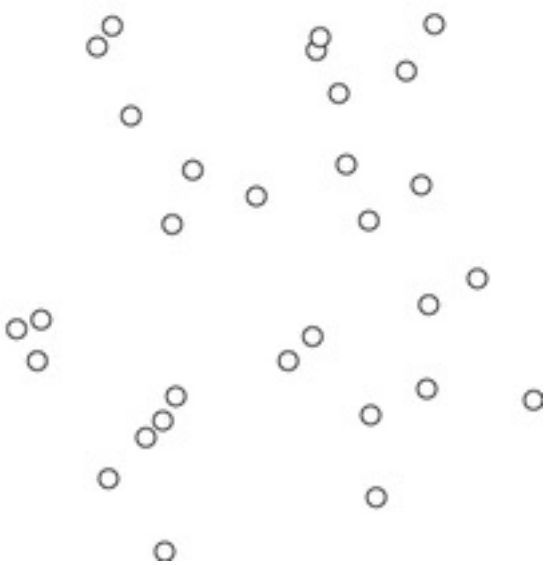


Kruskal's vs. Prim's

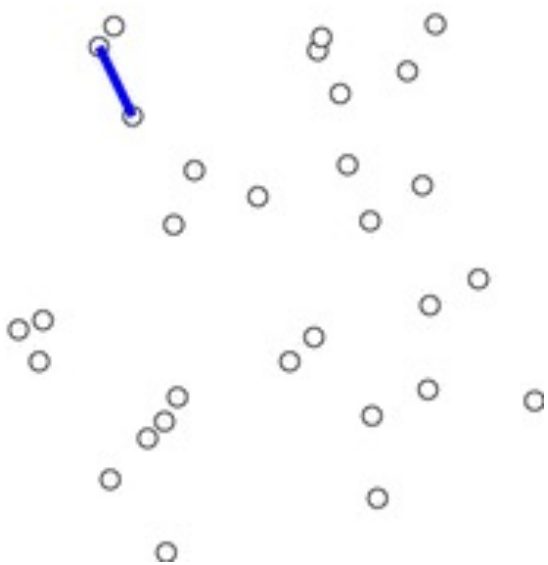
- Kruskal's algorithm: $O(|E| \lg |E|)$.
- Prim's algorithm: $O(|E| \lg |V|)$.
- Just compare $|E|$ and $|V|$.
  - For a sparse graph (close to null graph, $O(|E|) = O(1)$), Kruskal's algorithm is faster.
  - For a dense graph (close to complete graph, $O(|E|) = O(|V|^2)$), Prim's algorithm is faster.

Kruskal's vs. Prim's

Complete graph



Kruskal's algorithm



Prim's algorithm

Image source: https://en.wikipedia.org/wiki/Kruskal%27s_algorithm
https://en.wikipedia.org/wiki/Prim%27s_algorithm

Classroom Exercise

- Kruskal's algorithm and Prim's algorithm both belong to 避圈法.
- Another method to get a MST is called reverse-delete algorithm (破圈法). We sort the edge weights in decreasing order first. Then, remove the edge if the graph is not disconnected.
- Try to prove the correctness of this algorithm.

Classroom Exercise

Proof:

- Prove by contradiction: Assume that the tree T obtained from reverse-delete algorithm is not a MST.
- Then, there exists a MST T' that is different from T.
- Pick the edge e that in T but not in T'. Adding e into T' will form a cycle.
- e being maintained in T means that there is another e' whose weight is larger than e in this cycle.
- Replacing e' with e will obtain a better tree that contradicts with that T' is a MST.

THE SHORTHEST PATHS PROBLEM

Shortest-Paths Problem

The shortest-paths problem:

- Given a weighted, directed graph $G = (V, E)$, with weights $w(u, v)$ for each edge $(u, v) \in E$, the weight of path $p = \langle v_0, v_1, \ldots, v_k \rangle$ is the sum of the weights of its edges:

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

- The shortest-path weight from u to v can be defined as follows:

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

- A shortest path from vertex u to vertex v is then defined as any path p with weight $w(p) = \delta(u, v)$.

Variants of Shortest-Paths Problem

- The single-source shortest-paths problem: Given a graph $G = (V, E)$, we want to find a shortest path from a given source vertex $s \in V$ to each vertex $v \in V$.
- Single-pair shortest-path problem: Given vertices u and v, find a shortest path from u to v.
- All-pairs shortest-paths problem: Find a shortest path from u to v for every pair of vertices u and v.

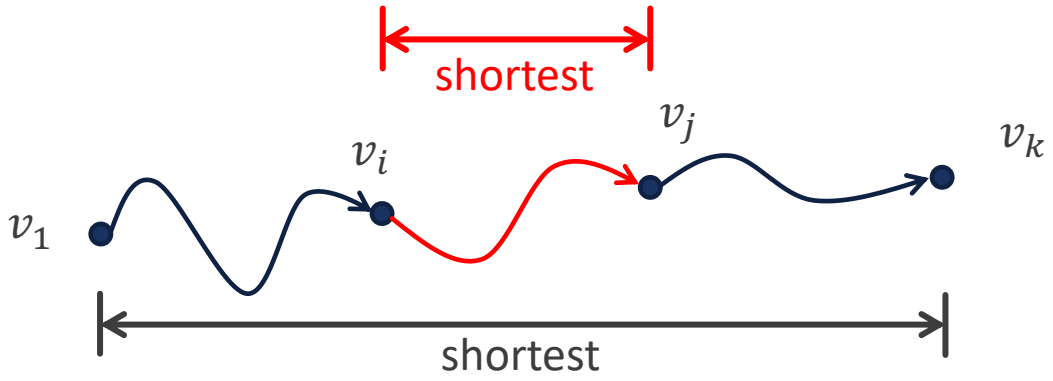
Optimal Substructure of a Shortest Path

Theorem 8.7

Given a weighted, directed graph $G = (V, E)$ with weights $w(u, v)$ for each edge $(u, v) \in E$, let $p = v_1, v_2, \ldots, v_k$ be a shortest path from vertex v_1 to vertex v_k.

For any i and j such that $1 \leq i \leq j \leq k$, let $p_{ij} = v_i, v_{i+1}, \ldots, v_j$ be the subpath of p from vertex v_i to vertex v_j.

Then, p_{ij} is a shortest path from v_i to v_j.



Relaxation

- The algorithms in this chapter use the technique of relaxation (松弛).
- For each vertex $v \in V$, we maintain an attribute $d[v]$, which is an upper bound on the weight of a shortest path from source s to v.
- We call $d[v]$ a shortest-path estimate. It is updated to get smaller as the algorithm goes on.
  - Just similar to the $d[v]$ in BFS, but can be updated more than once to get close to $\delta(s, v)$.

Relaxation

- We initialize the shortest-path estimates $d[v]$ and predecessors $\pi[v]$ by the following $O(V)$-time procedure.
  - Exactly same as how BFS does.

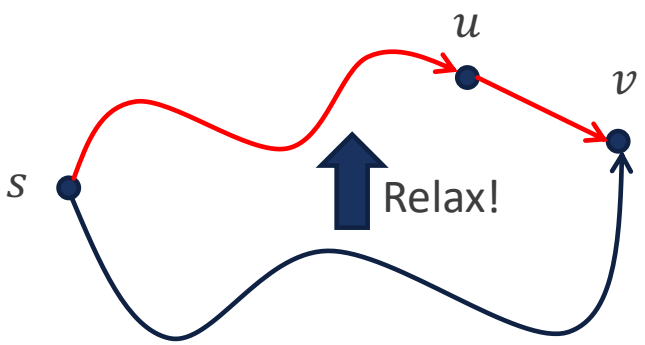
```
InitializeSingleSource(G, s)
1 for each vertex v ∈ V do
2       d[v] ← ∞
3       π[v] ← NIL
4 d[s] ← 0
```

Relaxation

- The process of relaxing an edge (u, v) consists of testing whether we can improve the shortest path to v found so far by going through u and, if so, updating $d[v]$ and $\pi[v]$.

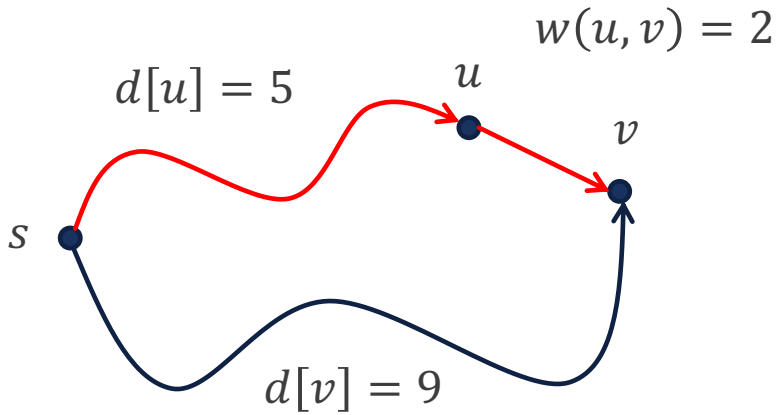
```
Relax(u, v, w)
1 if d[v] > d[u] + w(u, v) then
2       d[v] ← d[u] + w(u, v)
3       π[v] ← u
```

Relaxation in one sentence:
check if going though u is faster

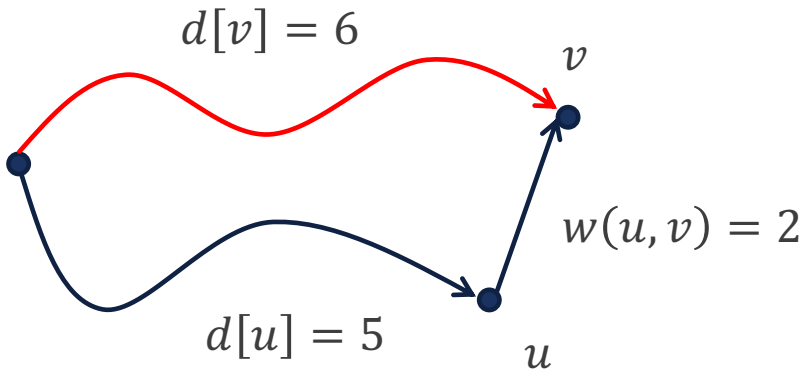


When $d[v] > d[u] + w(u, v)$

Example



Relax!



Don't relax

Properties of Shortest Paths and Relaxation

- Triangle inequality (三角不等式): For any edge $(u, v) \in E$,
$$\delta(s, v) \leq \delta(s, u) + w(u, v).$$
- Upper-bound property (上界性质): We always have $d[v] \geq \delta(s, v)$ for all vertices $v \in V$, and once $d[v]$ achieves the value $\delta(s, v)$, it never changes.
- No-path property (无路径性质): If there is no path from s to v, then $d[v] = \delta(s, v) = \infty$.
- Convergence property (收敛性质): If $s \sim u \rightarrow v$ is a shortest path in G for some $u, v \in V$, and $d[u] = \delta(s, u)$, then after $\text{Relax}(u, v, w)$, we have $d[v] = \delta(s, v)$.

THE SHORTHEST PATHS PROBLEM

SINGLE-SOURCE SHORTEST-PATHS PROBLEM: BELLMAN-FORD ALGORITHM

Single-Source Shortest-Paths Problem

Bellman-Ford algorithm:

- Given a weighted, directed graph $G = (V, E)$ with source s and weights $w(u, v)$ for each edge $(u, v) \in E$, where $w(u, v)$ can be negative.
- The Bellman-Ford algorithm does two things:
  - Calculate $\delta(s, v)$ for all $v \in V$.
  - Return a boolean value indicating whether or not there is a negative-weight cycle that is reachable from the source.
- If there is such a cycle, the algorithm indicates that no solution exists. If there is no such cycle, the algorithm produces the shortest paths and their weights.

Pseudocode

```
BellmanFord(G, w, s)
1  InitializeSingleSource(G, s)
2  for i ← 1 to |V| − 1 do
3     for each edge (u, v) ∈ E do
4           Relax(u, v, w)
5  for each edge (u, v) ∈ E do
6           if d[v] > d[u] + w(u, v) then return False
7  return True
```

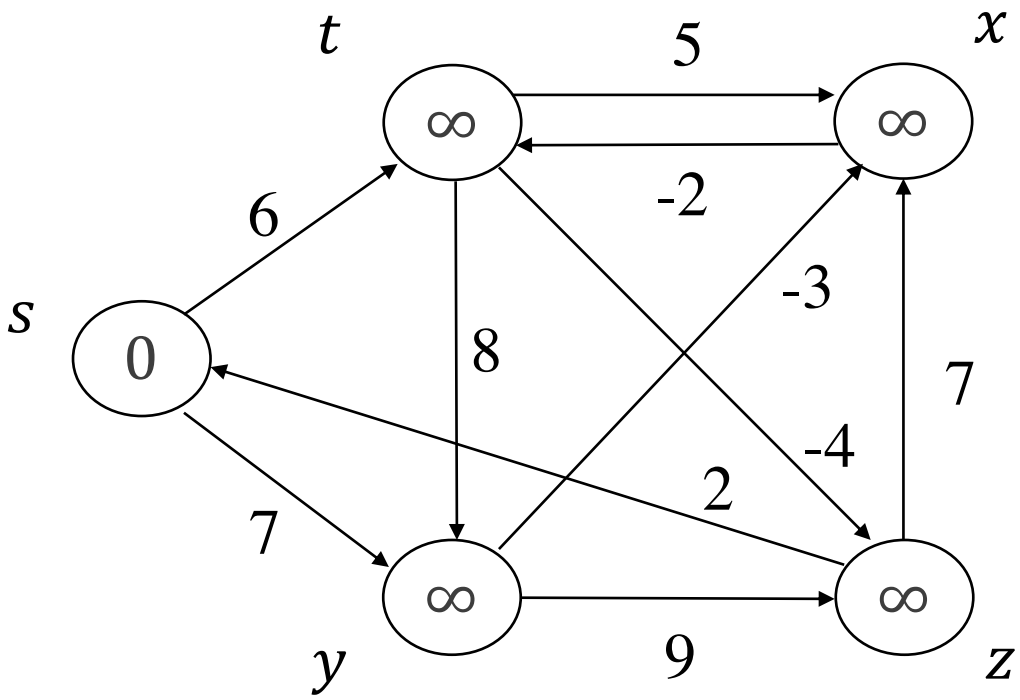
Do $|V| - 1$ times

Each time relax all edges

After $|V| - 1$ times, there still exists an edge can be relaxed, it means negative-weight cycle exists.

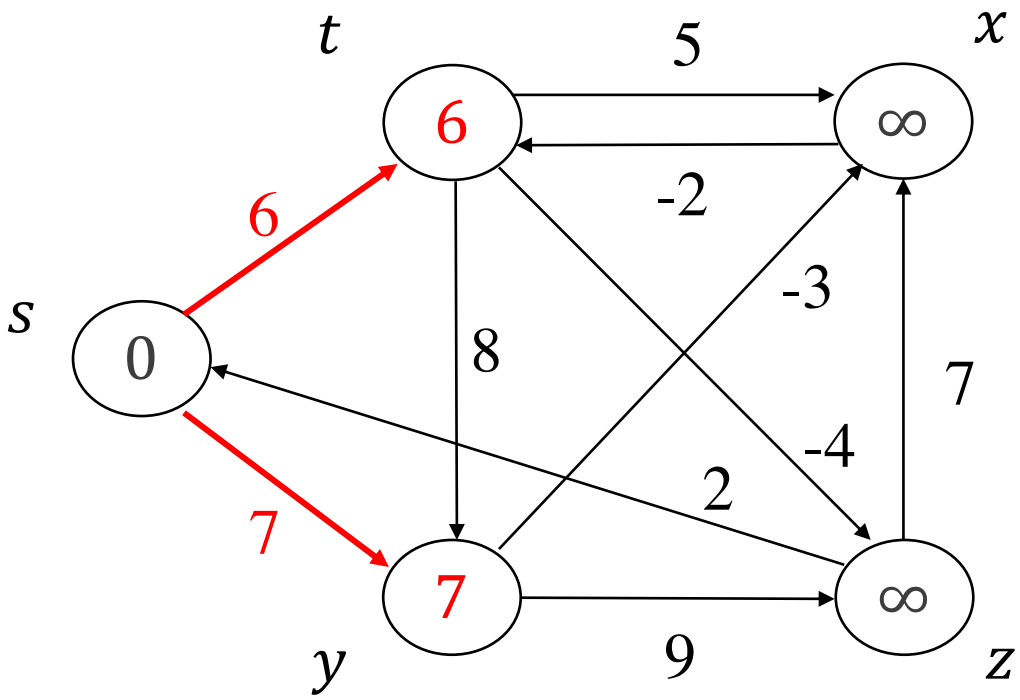
Total running time: $O(|V||E|)$

Example



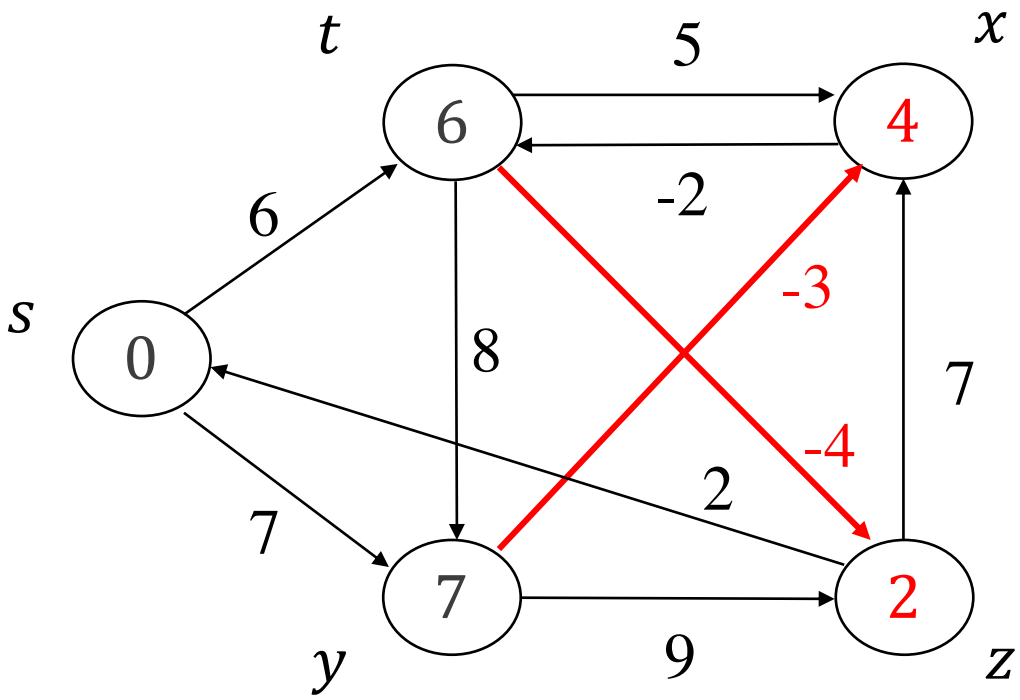
Initialization

Example



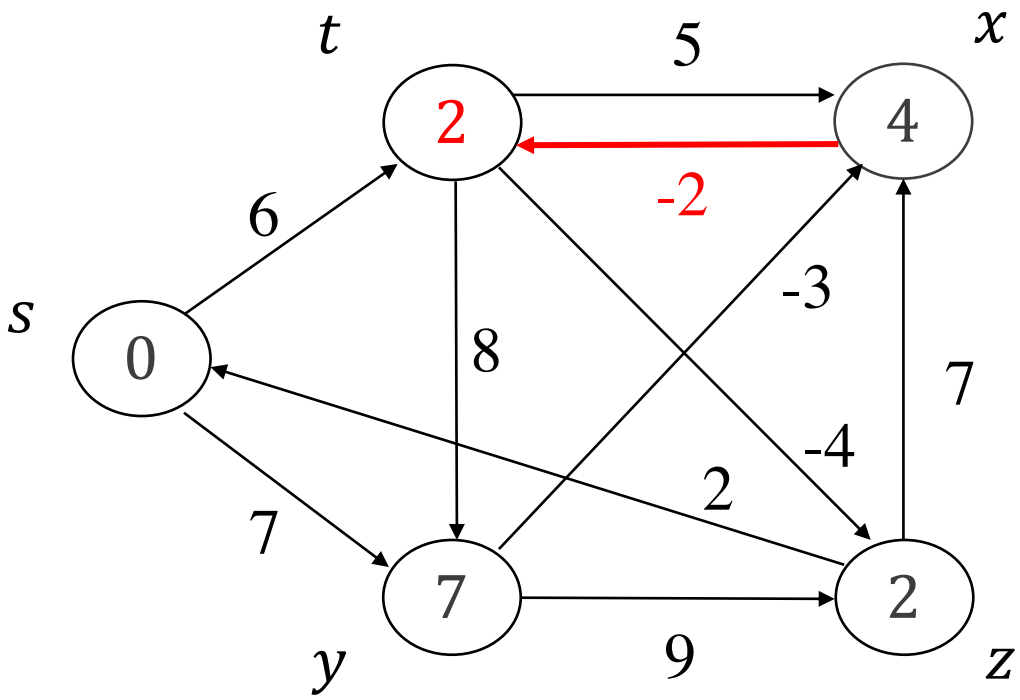
Round 1

Example



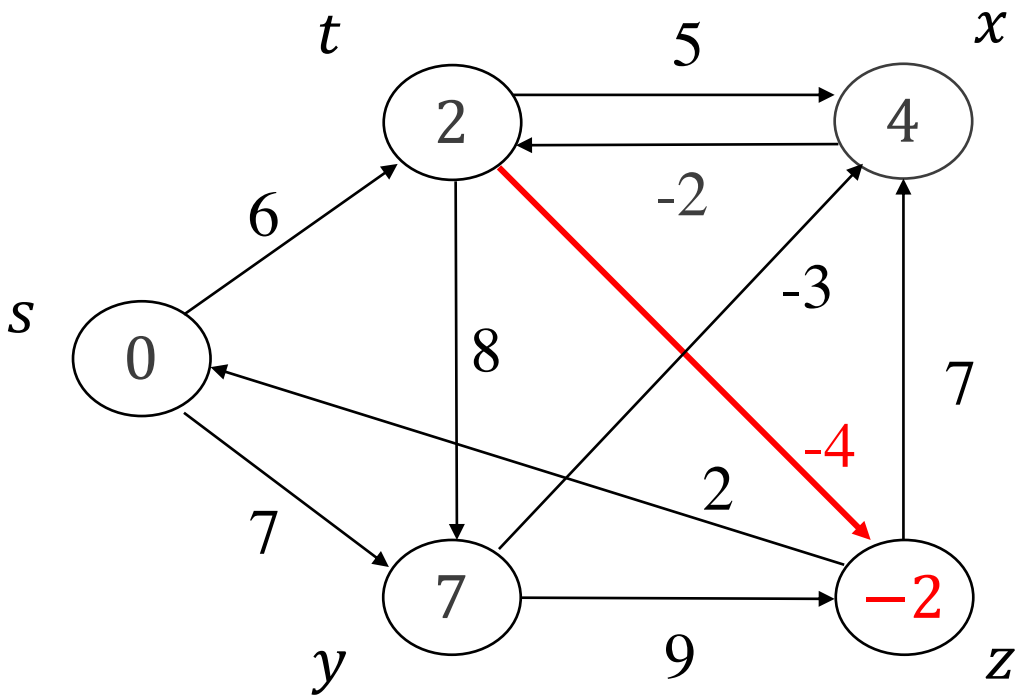
Round 2

Example



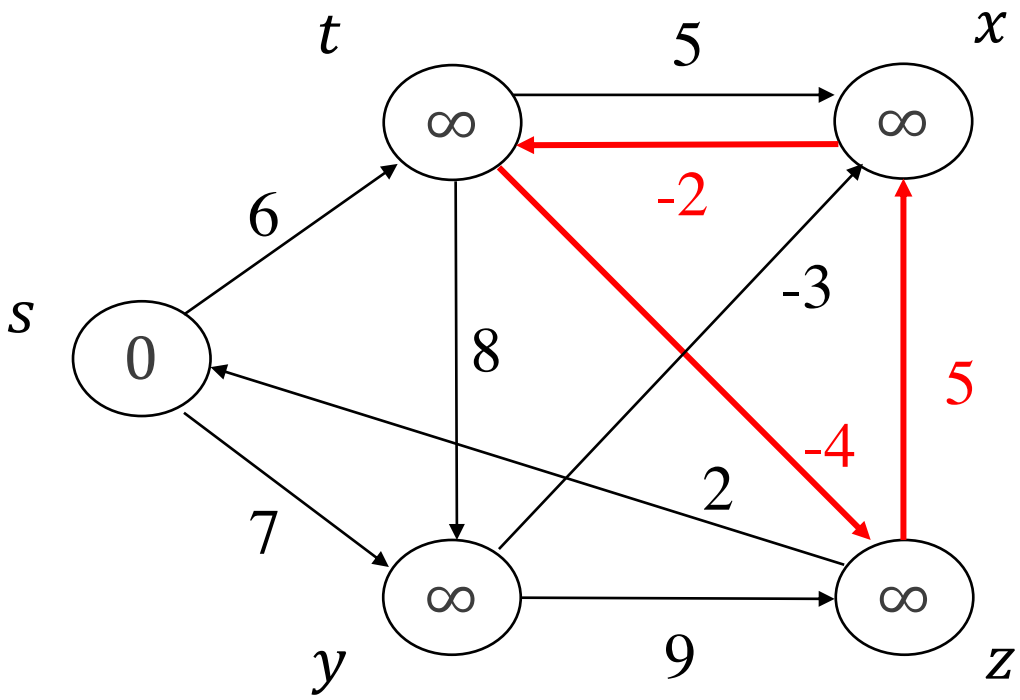
Round 3

Example



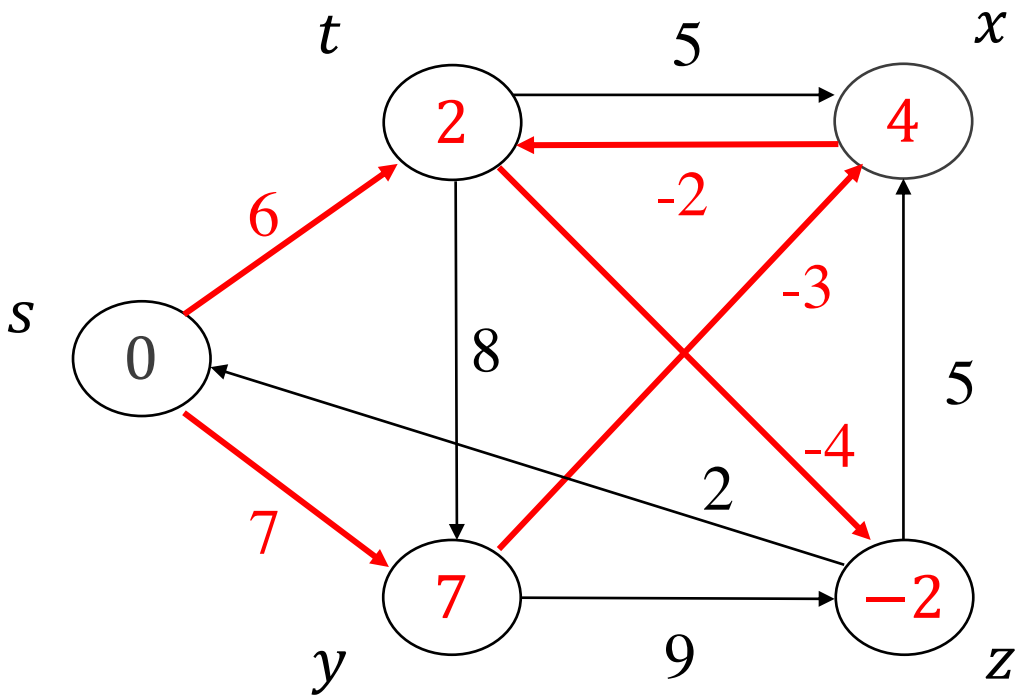
Round 4: No negative-weight cycle, return TRUE

Example



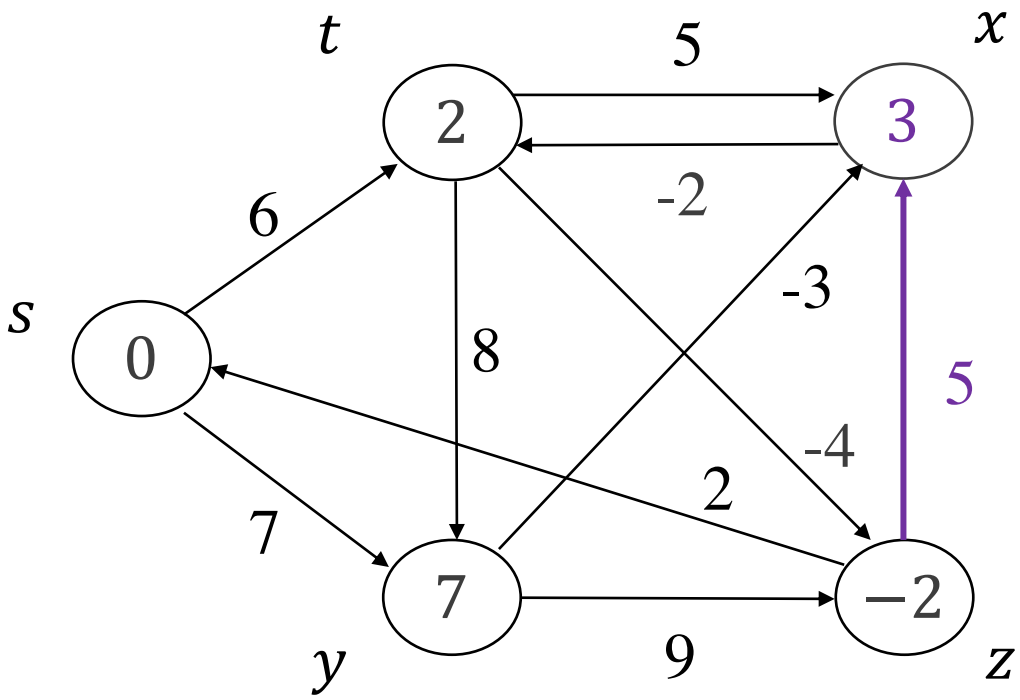
Now, if we indeed have a negative-weight cycle…

Example



Round 1-4 are same

Example



We can still relax on an edge, return FALSE

THE SHORTHEST PATHS PROBLEM

SINGLE-SOURCE SHORTEST-PATHS PROBLEM: DIJKSTRA'S ALGORITHM

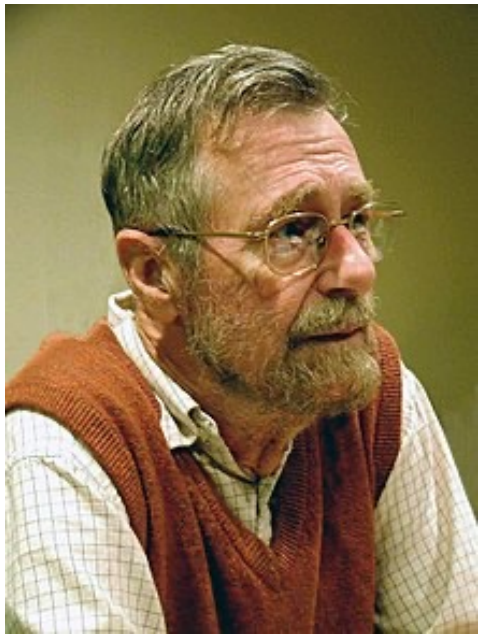
Single-Source Shortest-Paths Problem

- Although Bellman-Ford algorithm is useful for negative-weight cycle detection, it is too slow ($O(|E||V|)$).
- If we know that there's no negative weight on the graph, we may have a better algorithm.

Single-Source Shortest-Paths Problem

Dijkstra's (/ˈdaɪkstrə/) (戴克斯特拉, 迪杰斯特拉) algorithm

- Maintains a set S of vertices whose final shortest-path weights from the source s have already been determined.
- Repeatedly selects the vertex $u \in V - S$ with the minimum shortest-path estimate, adds u to S, and relaxes all edges leaving u.



Edsger Dijkstra
(1930-2002)

Image source: https://en.wikipedia.org/wiki/Edsger_W._Dijkstra

Dijkstra's Algorithm

```
Dijkstra(G, w, s)
1  InitializeSingleSource(G, s)
2  S ← ∅
3  Q ← V
4  while Q ≠ ∅ do
5      u ← ExtractMin(Q)
6      S ← S ∪ {u}
7      for each vertex v ∈ Adj[u] do
8          Relax(u, v, w)
```

S maintains the vertices with determined shortest-path weight

Select $v \in Q$ with minimal $d[v]$

Relax all v's neighbors

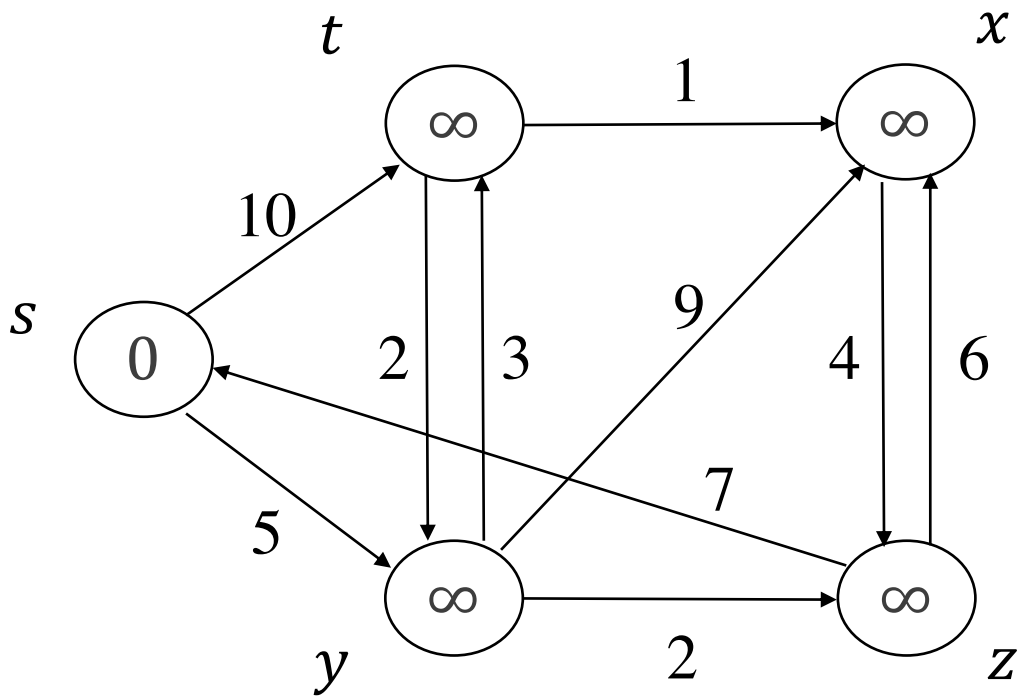
Total running time: $O(|V|^2)$

Example

$S = \{\}$

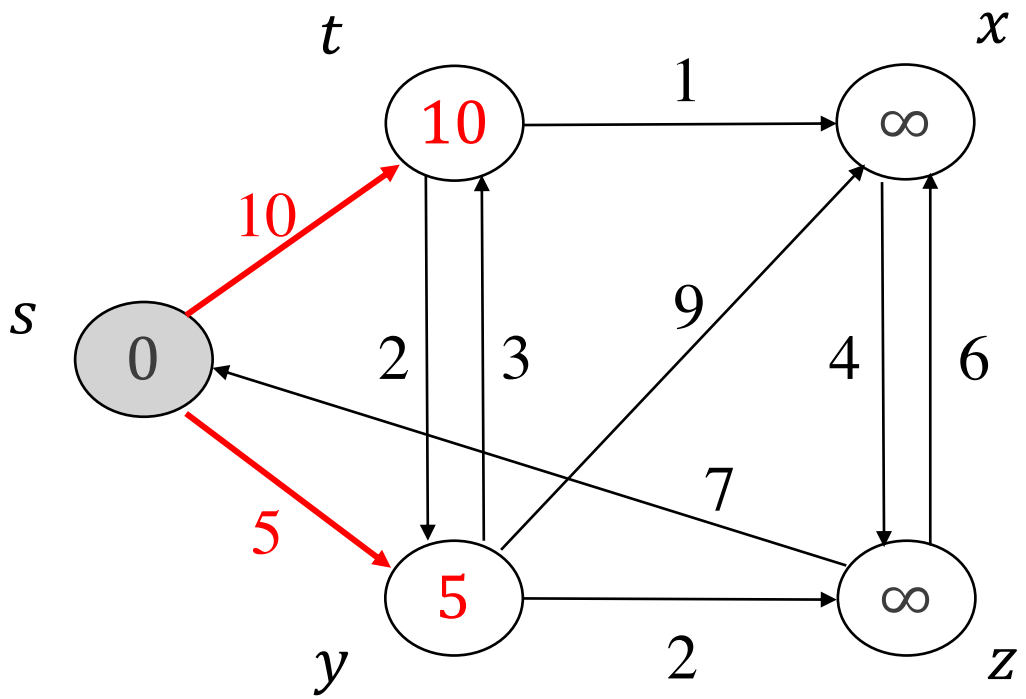
Q

s	t	x	y	z
0	∞	∞	∞	∞



Initialization

Example



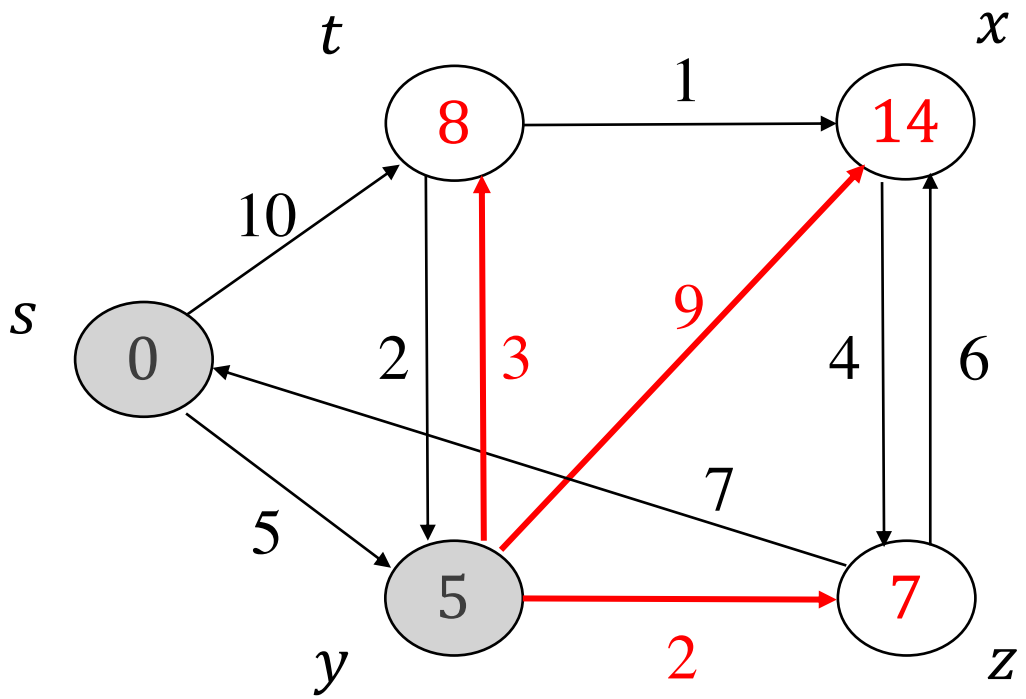
$S = \{s\}$

Q

t	x	y	z
10	∞	5	∞

$\text{ExtractMin}(Q) = s$

Example



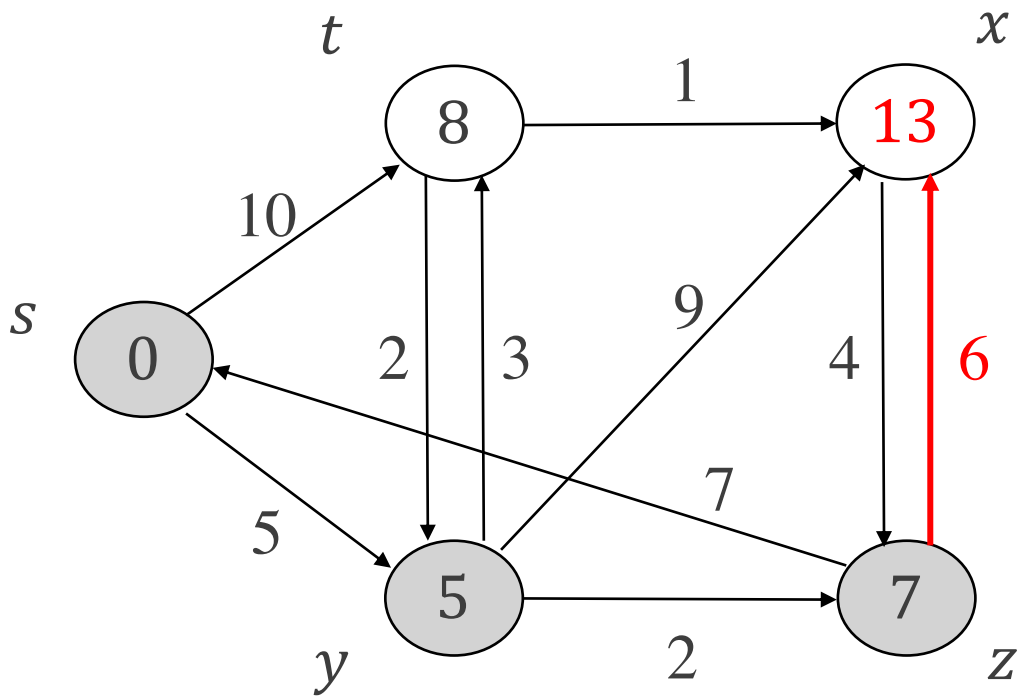
$S = \{s, y\}$

Q

t	x	z
8	14	7

$\text{ExtractMin}(Q) = y$

Example



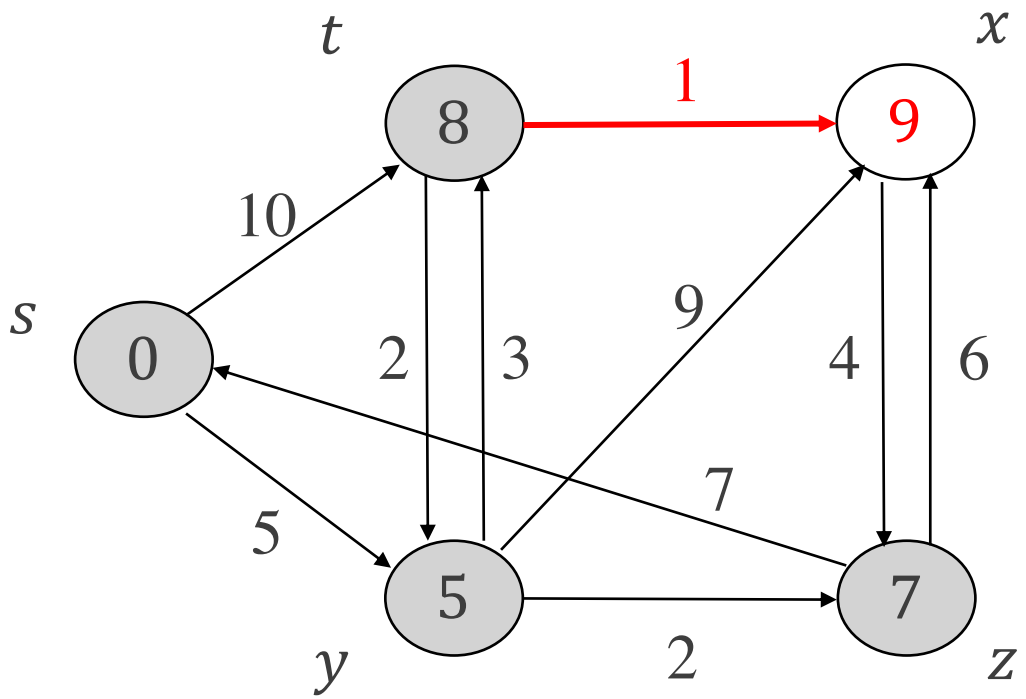
$S = \{s, y, z\}$

Q

t	x
8	13

$\text{ExtractMin}(Q) = z$

Example



$S = \{s, y, z, t\}$

Q $\boxed{x}$

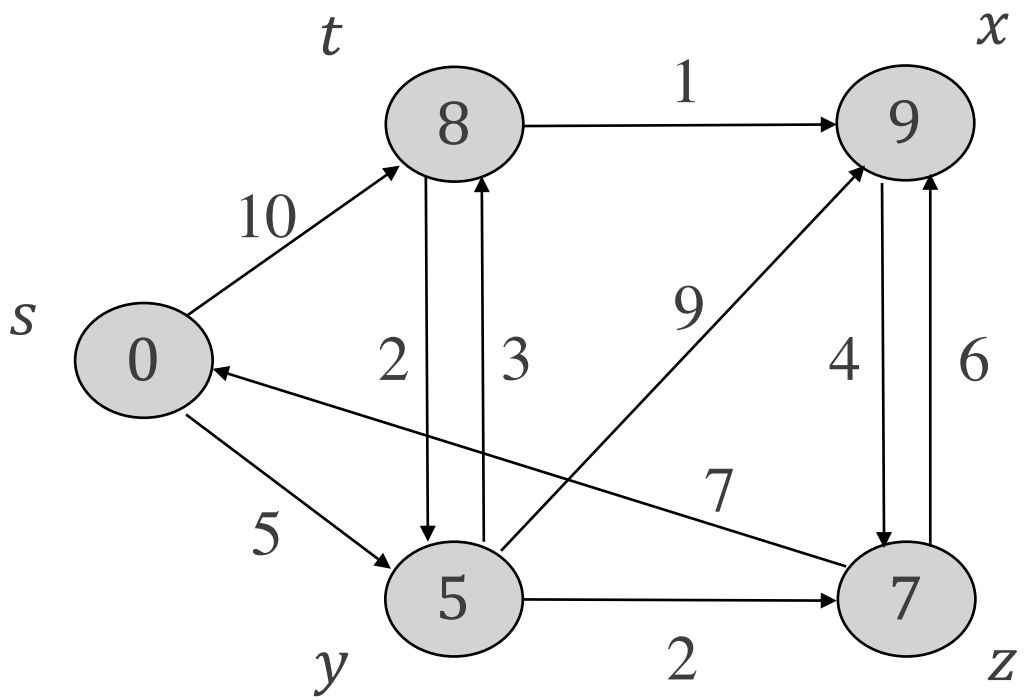
9

$\text{ExtractMin}(Q) = t$

Example

$S = \{s, y, z, t, x\}$

Q



$\text{ExtractMin}(Q) = x$

Example

A demo of Dijkstra's algorithm based on Euclidean distance.

- Red lines are the shortest path covering, i.e., connecting u and $\pi[u]$.
- Blue lines indicate where relaxing happens, i.e., connecting v with a node u in Q, which gives a shorter path from the source to v.

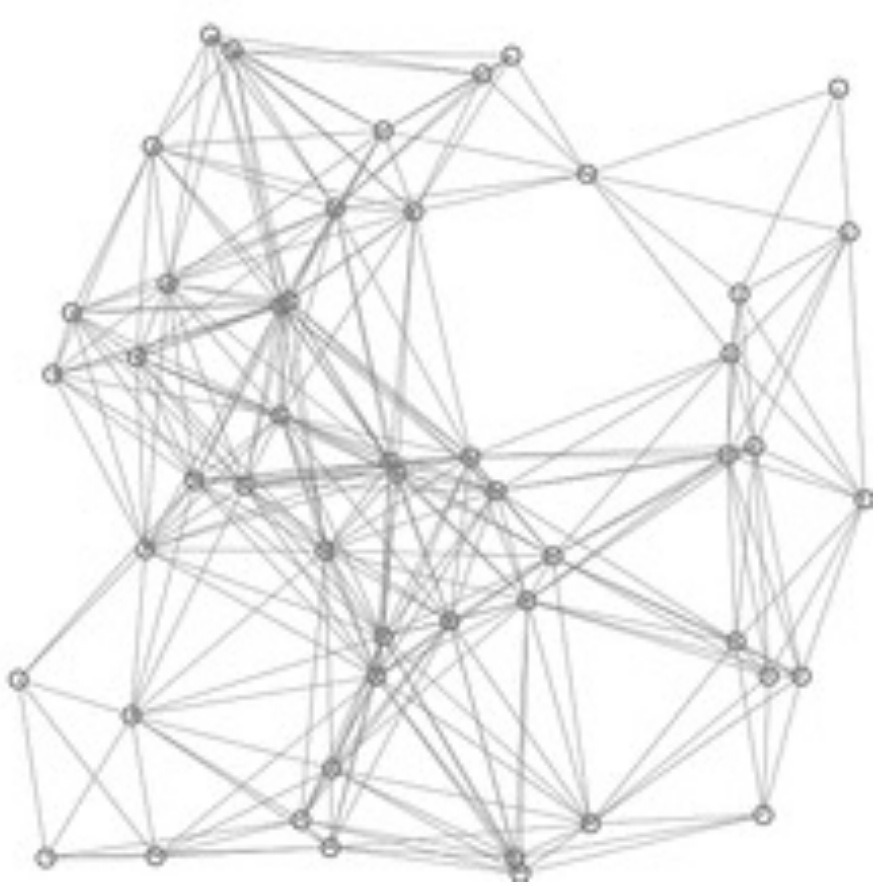


Image source: https://en.wikipedia.org/wiki/Dijkstra%27s_algorithm

Dijkstra's Algorithm vs. Prim's Algorithm

```
Dijkstra(G, w, s)
1  InitializeSingleSource(G, s)
2  S ← ∅
3  Q ← V
4  while Q ≠ ∅ do
5      u ← ExtractMin(Q)
6      S ← S ∪ {u}
7      for each vertex v ∈ Adj[u] do
8          Relax(u, v, w)
```

```
Relax(u, v, w)
1 if d[v] > d[u] + w(u, v) then
2      d[v] ← d[u] + w(u, v)
3      π[v] ← u
```

```
PrimMST(G, w, r)
1  for each u ∈ V do
2      key[u] ← ∞
3      π[u] ← NIL
4  key[r] ← 0
5  Q ← V
6  while Q ≠ ∅ do
7      u ← ExtractMin(Q)
8      for each v ∈ Adj[u] do
9          if v ∈ Q and w(u, v) < key[v] then
10             π[v] ← u
11             key[v] ← w(u, v)
12             DecreaseKey(Q, v, key)
```

Correctness of Dijkstra's Algorithm

Theorem 8.8

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

- We use loop invariant to prove: At the start of each iteration of the while loop (Line 4-8), $d[v] = \delta(s, v)$ for each vertex $v \in S$.

Correctness of Dijkstra's Algorithm

Proof:

Initialization: Initially, $S = \emptyset$, and so the invariant is trivially true.

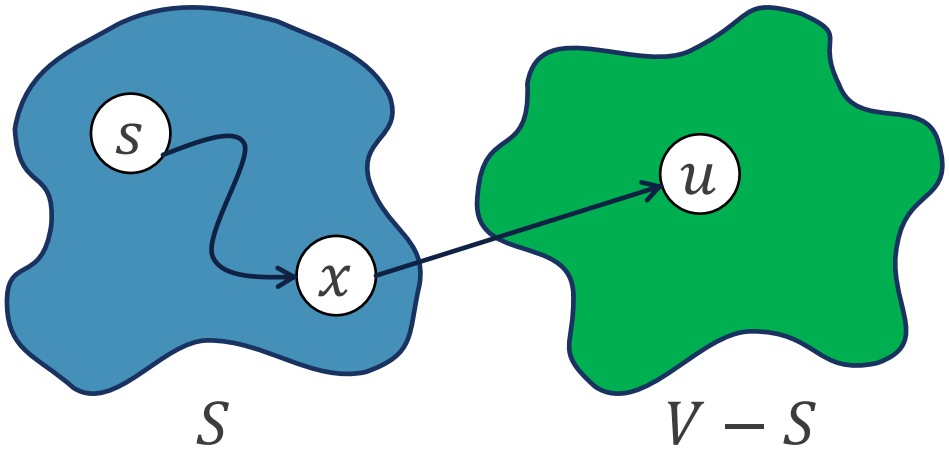
Maintenance: We wish to show that after each iteration, $d[u] = \delta(s, u)$ for the vertex u just added to set S.

- If there is no shortest path from s to u, $d[u] = \delta(s, u) = \infty$.

Correctness of Dijkstra's Algorithm

Proof (cont'd):

- Now we consider when there is a shortest path p from s to u.
- If in the path p, u is connected to a vertex $x \in S$, without any other vertex $y \in V - s$ between x and u, $d[u] = \delta(s, u)$ by the convergence property.
  - Because $x \in S$, we know $d[x] = \delta(s, x)$.

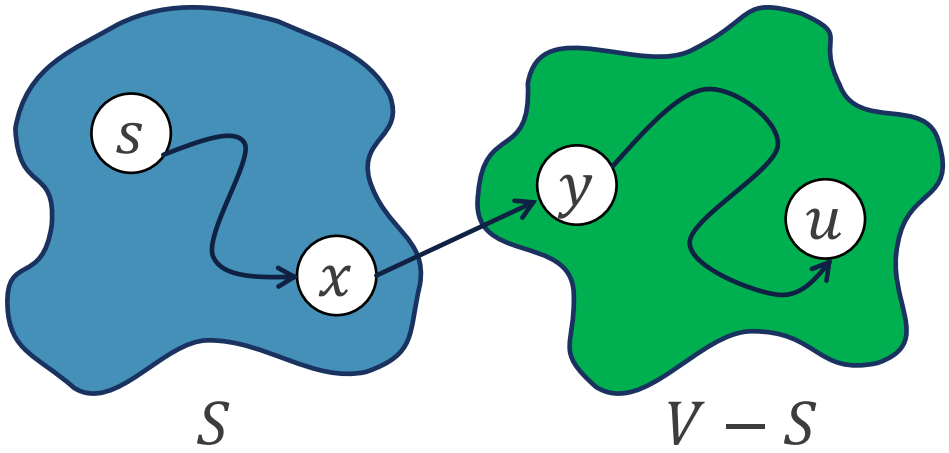


Convergence property: If s ~ $u \rightarrow v$ is a shortest path in G for some $u, v \in V$, and $d[u]=\delta(s,u)$, then after Relax(u, v, w), we have $d[v]=\delta(s,v)$.

Correctness of Dijkstra's Algorithm

Proof (cont'd):

- If u is not directly connected to x, there is a vertex $y \in V - s$ between x and u, we can assume y is the one connected to x.
- Again, by the convergence property we have $d[y] = \delta(s, y)$.

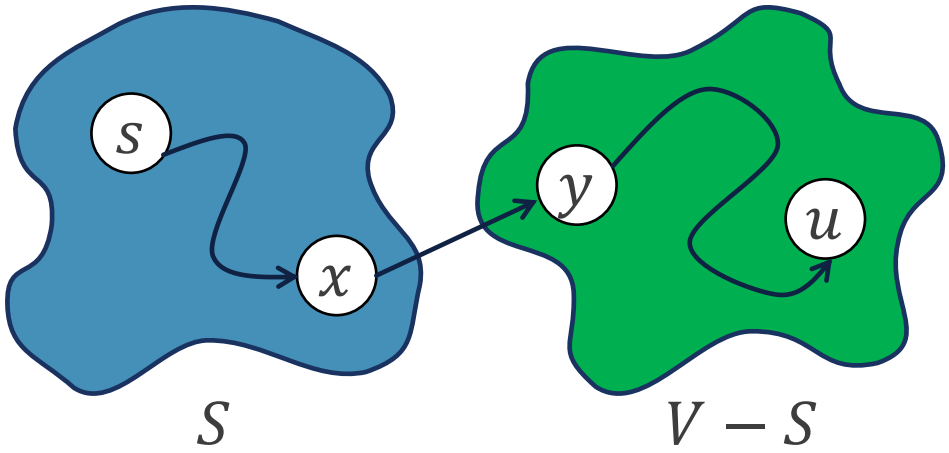


Correctness of Dijkstra's Algorithm

Proof (cont'd):

- Because y appears before u in the shortest path p, we have $\delta(s, y) \leq \delta(s, u)$.
- By the upper bound property, we have $\delta(s, u) \leq d[u]$, then we get:

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



Correctness of Dijkstra's Algorithm

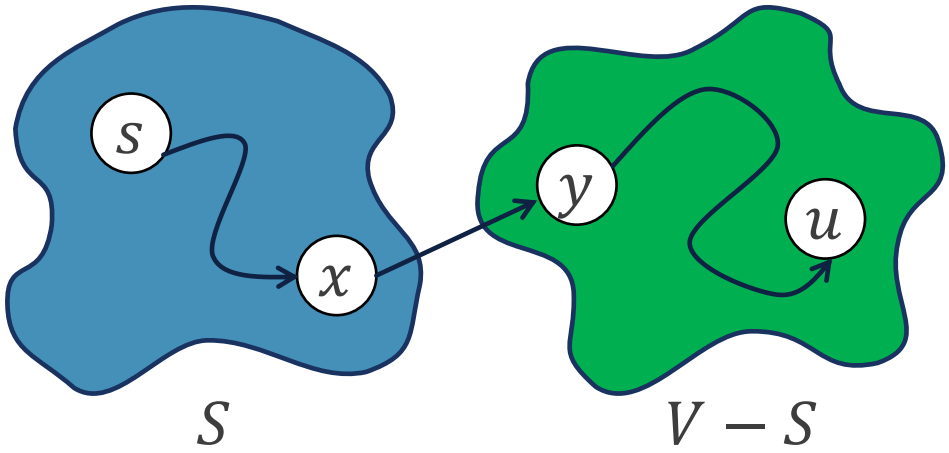
Proof (cont'd):

- However, u is chosen by Dijkstra's algorithm, which means:

$$d[u] \leq d[y].$$

- We have both $d[y] \leq d[u]$ and $d[u] \leq d[y]$, we get:

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



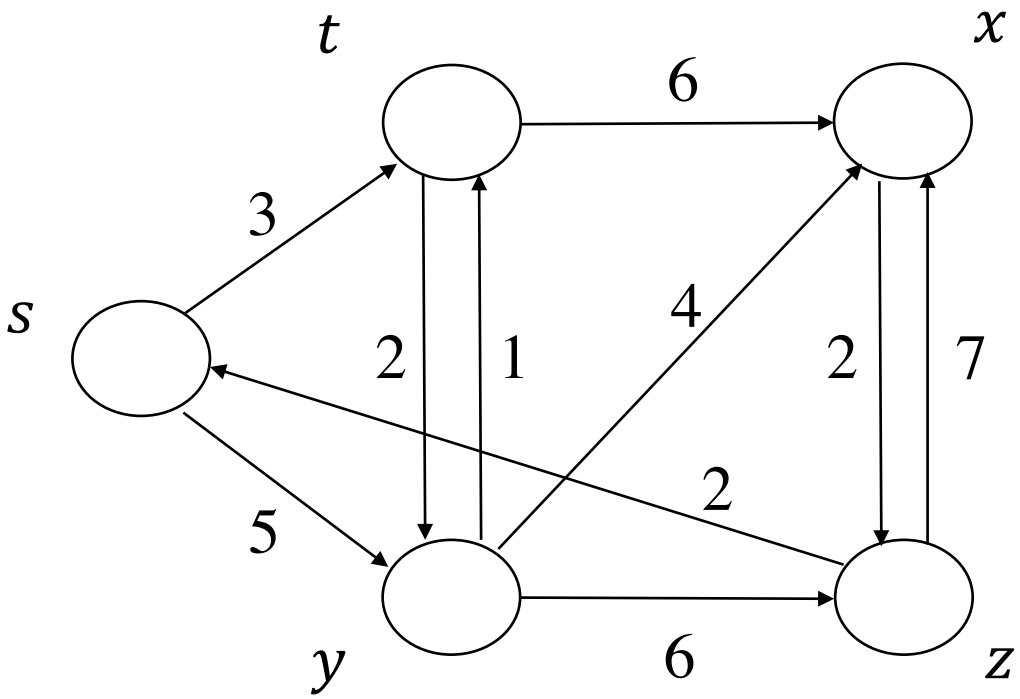
Correctness of Dijkstra's Algorithm

Proof (cont'd):

Termination: At termination, $Q = \emptyset$ and $S = V$. Thus, $d[u] = \delta(s, u)$ for all vertices $u \in V$.

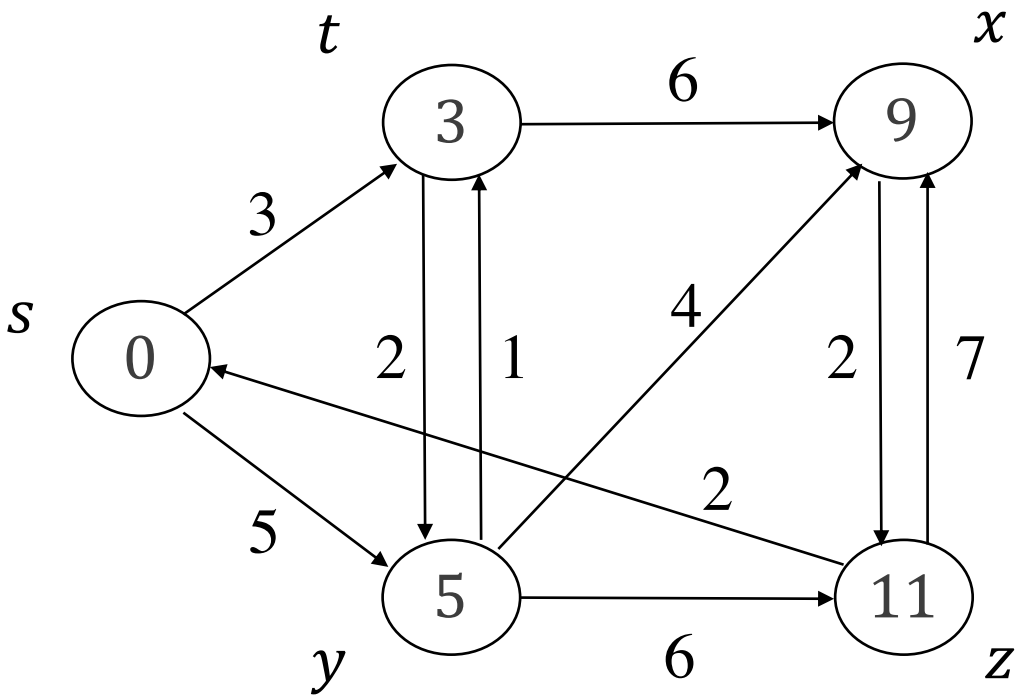
Classroom Exercise

Run Dijkstra's algorithm on the following graph, starting from s.



Classroom Exercise

Solution:



THE SHORTHEST PATHS PROBLEM

ALL-PAIRS SHORTEST-PATHS PROBLEM

All-Pairs Shortest-Paths Problem

- We can solve an all-pairs shortest-paths problem by running a single-source shortest-paths algorithm $|V|$ times, once for each vertex as the source.
- Can we do better? Can we get the shortest paths for all pairs of vertices at the same time?
- Yes, we have proved that the shortest path problem has optimal substructure property, dynamic programming is ready to go!

All-Pairs Shortest-Paths Problem

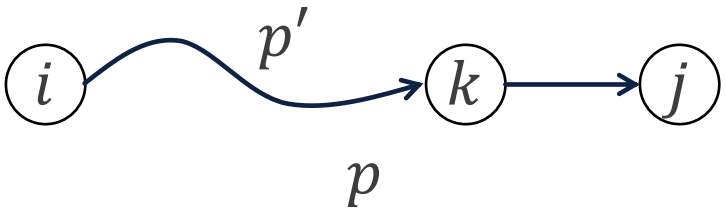
- For convenience, we assume that the vertices are numbered $1, 2, \ldots, |V|$, so that the input is an $|V| \times |V|$ matrix W representing the edge weights of an $|V|$-vertex directed graph $G = (V, E)$. That is, $W = (w_{ij})$, where

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

- Negative-weight edges are allowed, but we assume for the time being that the input graph contains no negative-weight cycles.

Dynamic Programming Solution

- Consider a shortest path p from vertex i to vertex j, and suppose that p contains at most m edges.
- Assuming that there are no negative-weight cycles, m is finite.
  - If $i = j$, then p has weight 0 because of no edges.
  - If vertices i and j are distinct, then we decompose path p into



  where path p' now contains at most $m - 1$ edges.
- By Theorem 8.7, p' is a shortest path from i to k, and so $\delta(i, j) = \delta(i, k) + w_{kj}$.

Dynamic Programming Solution

- Let $l_{ij}^{(m)}$ be the minimum weight of any path from vertex i to vertex j that contains at most m edges.
- When $m = 0$, there is a shortest path from i to j with no edges if and only if $i = j$. Thus,

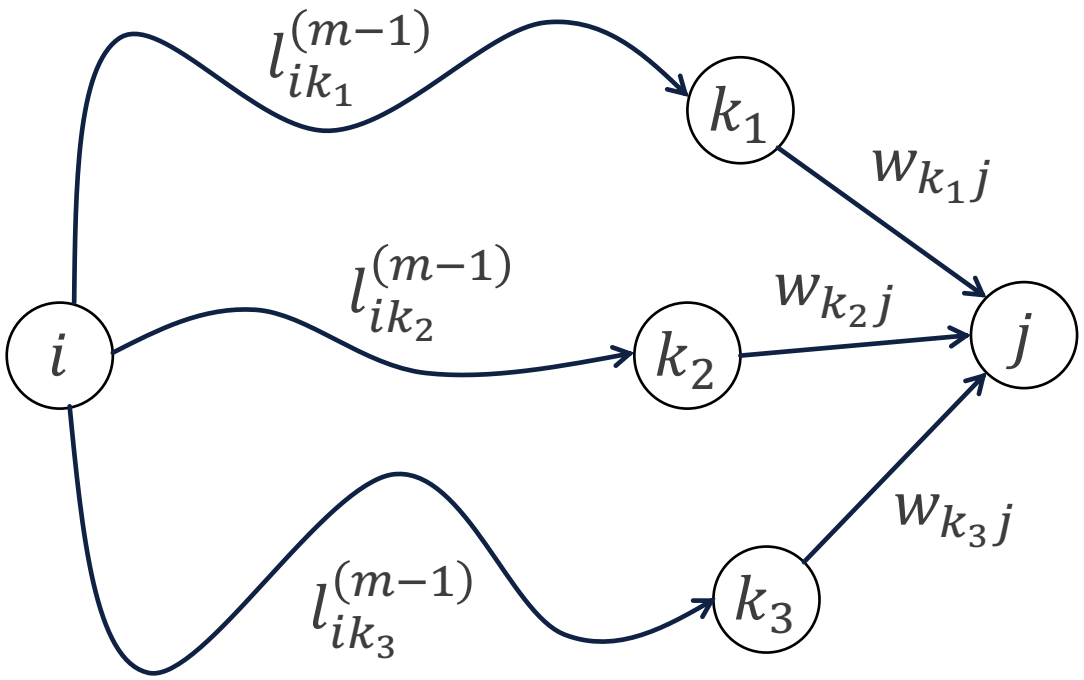
$$l_{ij}^{(0)} = \begin{cases} 0 & \text{if } i = j \\ \infty & \text{if } i \neq j \end{cases}$$

- When $m \geq 1$, we check all j's neighbor k and select the minimal one:

$$l_{ij}^{(m)} = \min_{1 \leq k \leq |V|} \left\{ l_{ik}^{(m-1)} + w_{kj} \right\}$$

Dynamic Programming Solution

$$l_{ij}^{(m)} = \min_{1 \le k \le |V|} \left\{ l_{ik}^{(m-1)} + w_{kj} \right\}$$



Dynamic Programming Solution

- If a shortest path contains more than $|V| - 1$ vertices, it must contain a cycle.
  - If the cycle weight is positive, we can remove it to get a shorter path.
  - If the cycle weight is negative, we can go though this cycle infinite number of times to obtain $\delta(i, j) = -\infty$.
- Therefore, we assume that the graph contains no negative-weight cycles.
- For every pair of vertices i and j for which $\delta(i, j) < \infty$, there is a shortest path from i to j that is simple and thus contains at most $|V| - 1$ edges.
- Finally: $\delta(i, j) = l_{ij}^{(|V|-1)}$.

Dynamic Programming Solution

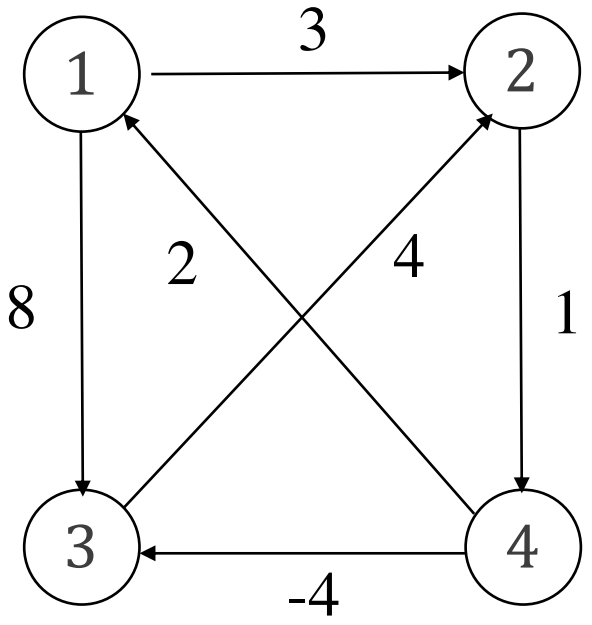
- Let matrix $L^{(m)} = (l_{ij}^{(m)})$, we iteratively calculate

$$L^{(1)}, L^{(2)}, \dots, L^{(|V|-1)}$$

- The matrix $L^{(|V|-1)}$ contains the shortest path distance of all vertex pairs.

Example

$$L^{(0)} = \begin{bmatrix} 0 & \infty & \infty & \infty \\ \infty & 0 & \infty & \infty \\ \infty & \infty & 0 & \infty \\ \infty & \infty & \infty & 0 \end{bmatrix}$$

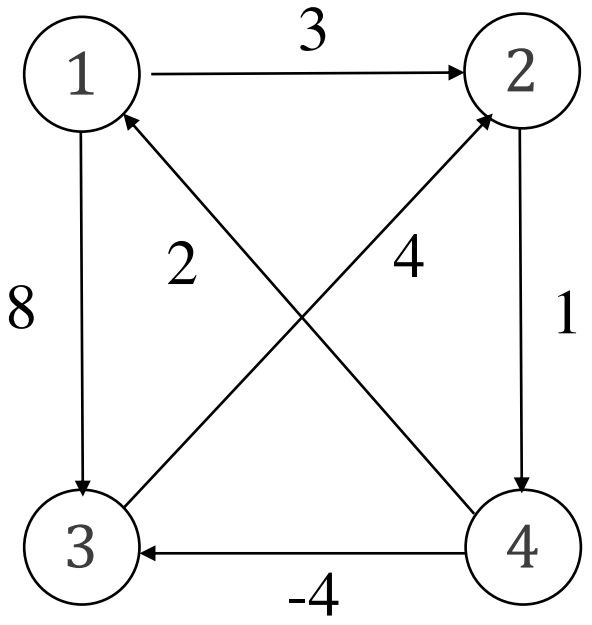


Initialization:

Shortest path distance with 0 edges.

Example

$$L^{(1)} = \begin{bmatrix} 0 & 3 & 8 & \infty \\ \infty & 0 & \infty & 1 \\ \infty & 4 & 0 & \infty \\ 2 & \infty & -4 & 0 \end{bmatrix}$$



Shortest path distance within 1 edges.

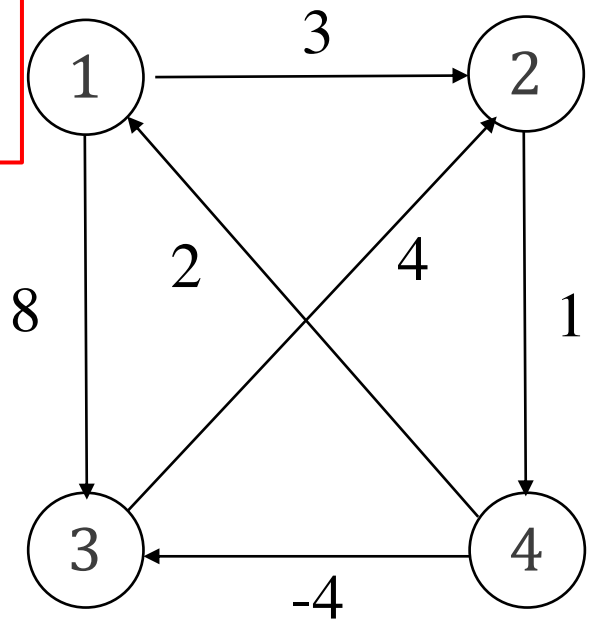
Example

$l_{11}^{(1)} + w_{12} = 0 + 3 = 3$
$l_{12}^{(1)} + w_{22} = 3 + 0 = 3$
$l_{13}^{(1)} + w_{32} = 8 + 4 = 12$
$l_{14}^{(1)} + w_{42} = \infty + \infty = \infty$

$l_{11}^{(1)} + w_{14} = 0 + \infty = \infty$
$l_{12}^{(1)} + w_{24} = 3 + 1 = 4$
$l_{13}^{(1)} + w_{34} = 8 + \infty = \infty$
$l_{14}^{(1)} + w_{44} = \infty + 0 = \infty$

$l_{12}^{(2)}$ $l_{14}^{(2)}$

$$L^{(2)} = \begin{bmatrix} 0 & 3 & 8 & 4 \\ 3 & 0 & -3 & 1 \\ \infty & 4 & 0 & 5 \\ 2 & 0 & -4 & 0 \end{bmatrix}$$

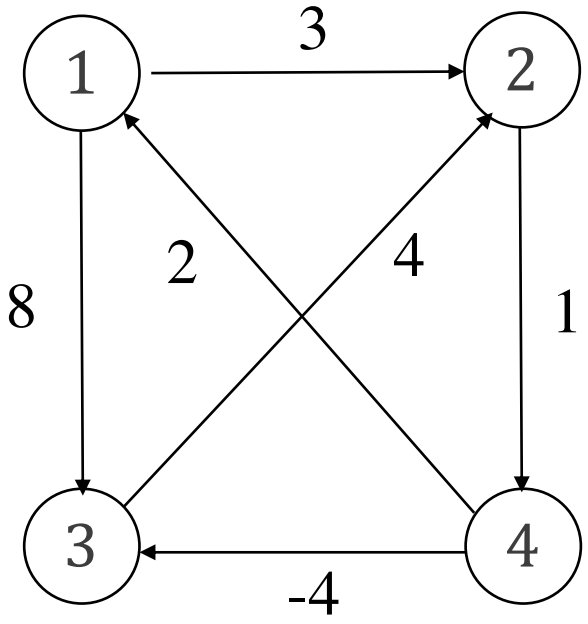


Shortest path distance within 2 edges.

Example

$$L^{(3)} = \begin{bmatrix} 0 & 3 & 0 & 4 \\ 3 & 0 & -3 & 1 \\ 7 & 4 & 0 & 5 \\ 2 & 0 & -4 & 0 \end{bmatrix}$$

Shortest path distance within 3 edges.



Dynamic Programming Solution

- Now look at the recursive equation, what is the computational complexity for this DP algorithm?

$$l_{ij}^{(m)} = \min_{1 \le k \le |V|} \left\{ l_{ik}^{(m-1)} + w_{kj} \right\}$$

- $O(|V|^4)$. Can we improve?

THE SHORTHEST PATHS PROBLEM

FLOYD-WARSHALL ALGORITHM

Floyd-Warshall Algorithm

Robert W. Floyd
(1936-2001)
Turing Award in 1978

Stephen Warshall
(1935-2006)

- Floyd-Warshall algorithm (弗洛伊德算法) is an example of dynamic programming, and was published by Robert Floyd in 1962 and also by Stephen Warshall in 1962.

Floyd-Warshall Algorithm

- Can we use the idea of 0/1 knapsack problem? Determine at each step that we take item i or not.
- First, we number all the vertices: $V = \{1,2, \dots, |V|\}$.
- Let $d_{ij}^{(k)}$ be the weight of a shortest path from vertex i to vertex j for which we can only go though vertices are in the set $\{1, 2, \dots, k\}$.
  - "Go though" does not include vertex i and j.
- When $k = 0$, $d_{ij}^{(0)} = w_{ij}$, because the path is directly from vertex i to vertex j, without going though any vertex.

Floyd-Warshall Algorithm

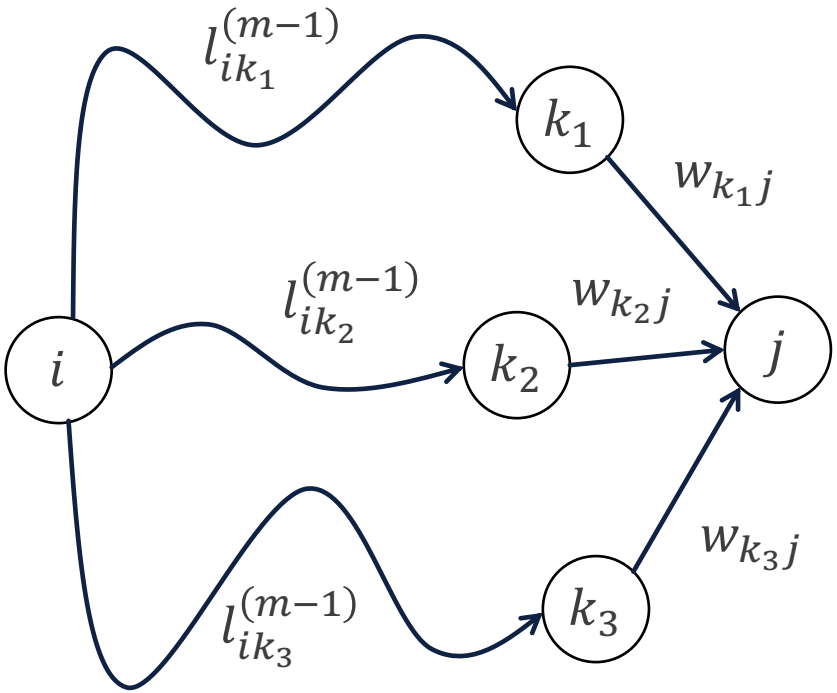
- We have know all shortest paths when we can go though vertices $\{1, ., \ldots, k-1\}$.
- Given a new vertex k to let you use it to go though, we may decide:
  - Don't go though it: $d_{ij}^{(k)} = d_{ij}^{(k-1)}$.
  - Go though it: $d_{ij}^{(k)} = d_{ik}^{(k-1)} + d_{kj}^{(k-1)}$.
- The recursive equation is:

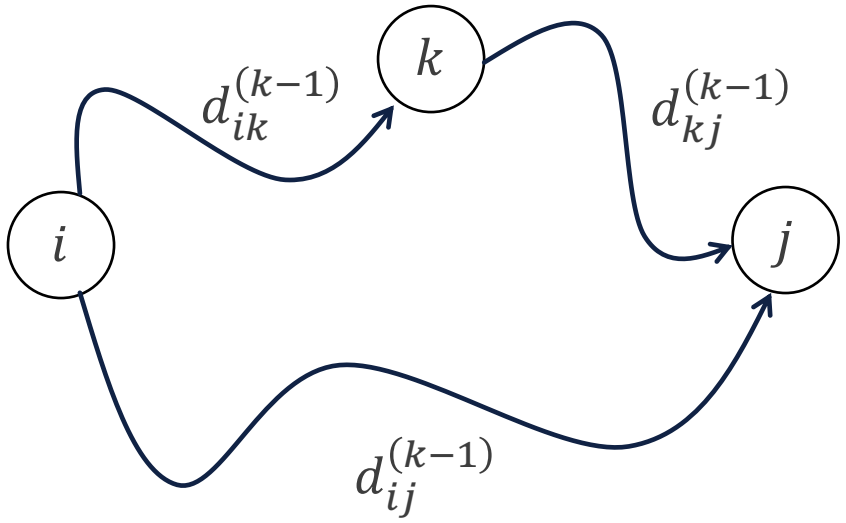
$$d_{ij}^{(k)} = \begin{cases} w_{ij} & \text{if } k = 0 \\ \min\{d_{ij}^{(k-1)}, d_{ik}^{(k-1)} + d_{kj}^{(k-1)}\} & \text{if } k \geq 1 \end{cases}$$

Floyd-Warshall Algorithm

$$l_{ij}^{(m)} = \min_{1 \le k \le |V|} \left\{ l_{ik}^{(m-1)} + w_{kj} \right\}$$

$$d_{ij}^{(k)} = \min\{ d_{ij}^{(k-1)}, d_{ik}^{(k-1)} + d_{kj}^{(k-1)} \}$$





Pseudocode

FloydWarshall(W)

1 $D^{(0)} \leftarrow W$

2 **for** $k \leftarrow 1$ **to** $|V|$ **do**

3 **for** $i \leftarrow 1$ **to** $|V|$ **do**

4 **for** $j \leftarrow 1$ **to** $|V|$ **do**

5 **if** $d_{ij}^{(k-1)} < d_{ik}^{(k-1)} + d_{kj}^{(k-1)}$ **then** $d_{ij}^{(k)} \leftarrow d_{ij}^{(k-1)}$

6 **else** $d_{ij}^{(k)} = d_{ik}^{(k-1)} + d_{kj}^{(k-1)}$

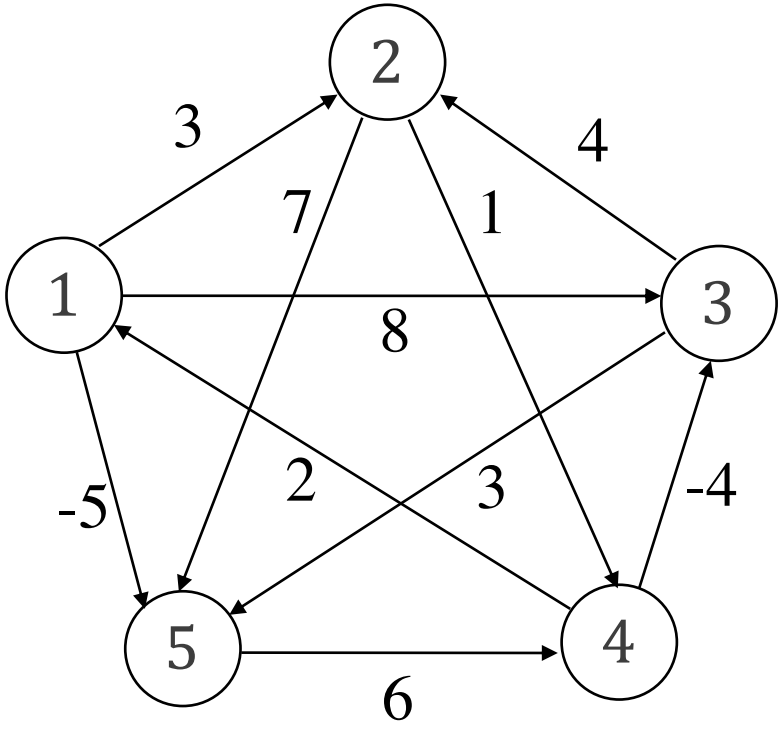
7 **return** $D^{(|V|)}$

Total running time: $O(|V|^3)$

Example

$$D^{(0)} = \begin{bmatrix} 0 & 3 & 8 & \infty & -5 \\ \infty & 0 & \infty & 1 & 7 \\ \infty & 4 & 0 & \infty & 3 \\ 2 & \infty & -4 & 0 & \infty \\ \infty & \infty & \infty & 6 & 0 \end{bmatrix}$$

Initialization

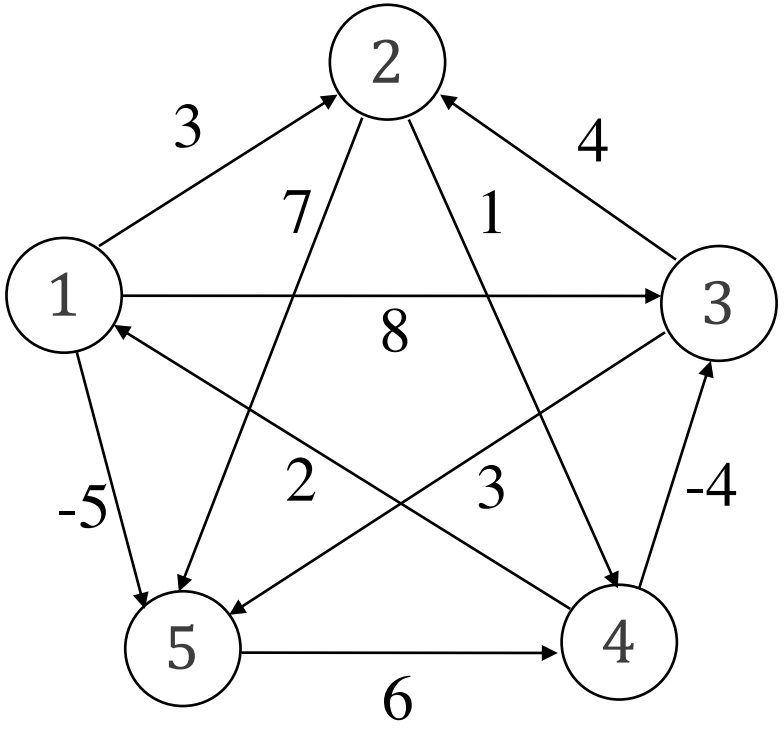


Example

$d_{42}^{(0)} = \infty$
$d_{41}^{(0)} + d_{12}^{(0)} = 5$

$d_{45}^{(0)} = \infty$
$d_{41}^{(0)} + d_{15}^{(0)} = 3$

$$D^{(1)} = \begin{bmatrix} 0 & 3 & 8 & \infty & -5 \\ \infty & 0 & \infty & 1 & 7 \\ \infty & 4 & 0 & \infty & 3 \\ 2 & 5 & -4 & 0 & -3 \\ \infty & \infty & \infty & 6 & 0 \end{bmatrix}$$



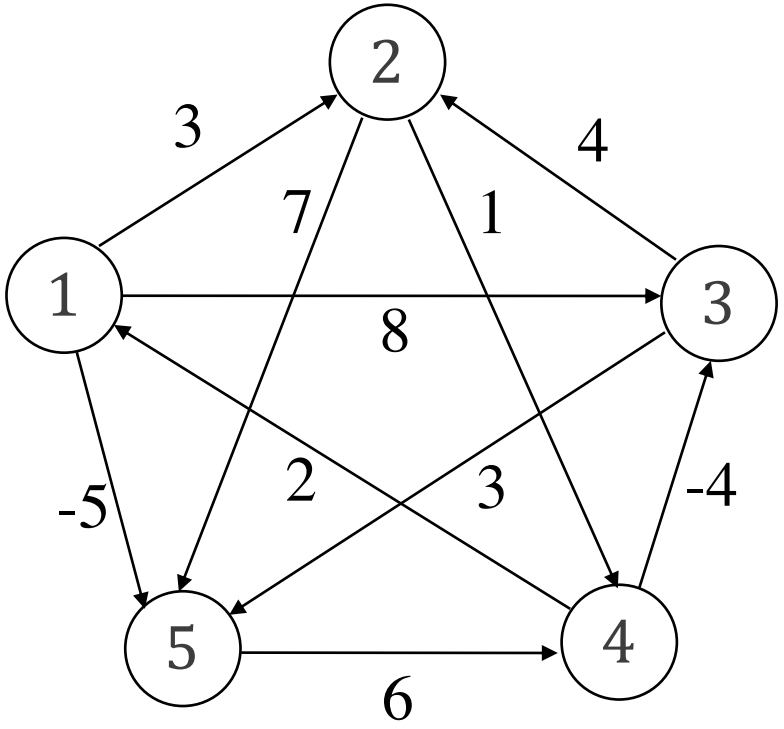
Only go though $\{v_1\}$

Example

$d_{34}^{(1)} = \infty$
$d_{32}^{(1)} + d_{24}^{(1)} = 5$

$d_{14}^{(1)} = \infty$
$d_{12}^{(1)} + d_{24}^{(1)} = 4$

$$D^{(2)} = \begin{bmatrix} 0 & 3 & 8 & 4 & -5 \\ \infty & 0 & \infty & 1 & 7 \\ \infty & 4 & 0 & 5 & 3 \\ 2 & 5 & -4 & 0 & -3 \\ \infty & \infty & \infty & 6 & 0 \end{bmatrix}$$

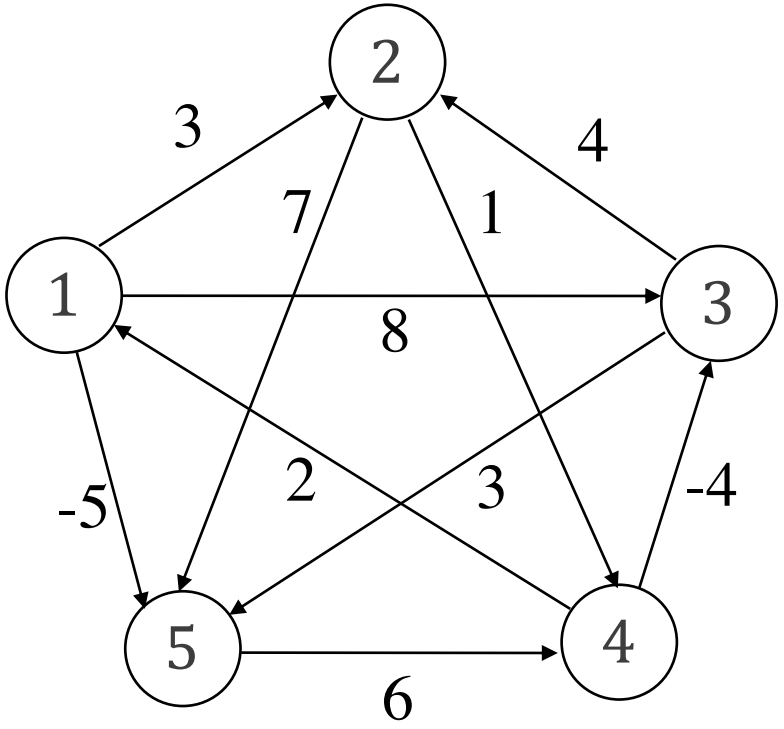


Only go though $\{v_1, v_2\}$

Example

$d_{42}^{(2)} = \infty$
$d_{43}^{(2)} + d_{32}^{(2)} = 0$

$$D^{(3)} = \begin{bmatrix} 0 & 3 & 8 & 4 & -5 \\ \infty & 0 & \infty & 1 & 7 \\ \infty & 4 & 0 & 5 & 3 \\ 2 & 0 & -4 & 0 & -3 \\ \infty & \infty & \infty & 6 & 0 \end{bmatrix}$$

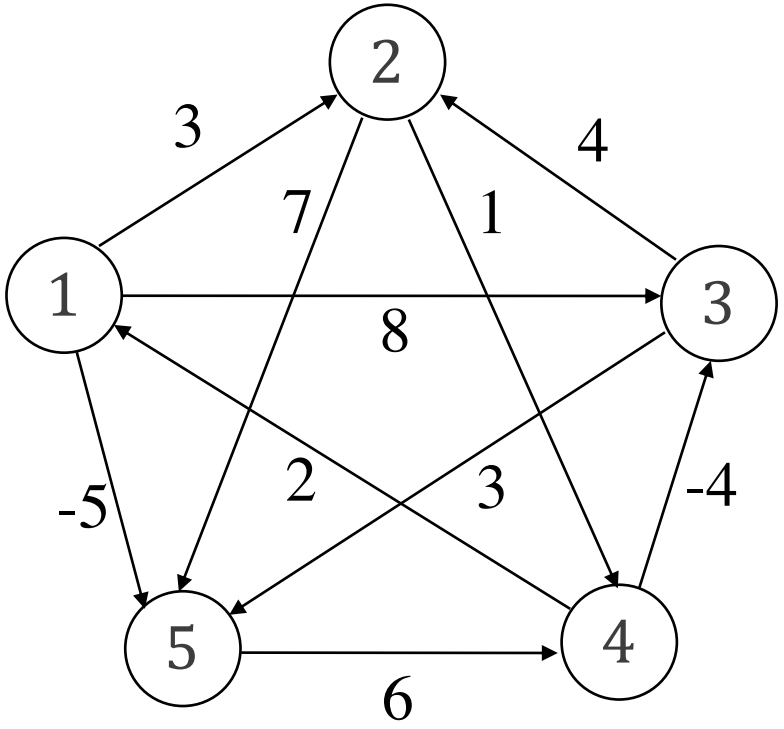


Only go though $\{v_1, v_2, v_3\}$

Example

$$D^{(4)} = \begin{bmatrix} 0 & 3 & 0 & 4 & -5 \\ 3 & 0 & -3 & 1 & -2 \\ 7 & 4 & 0 & 5 & 2 \\ 2 & 0 & -4 & 0 & -3 \\ 8 & 6 & 2 & 6 & 0 \end{bmatrix}$$

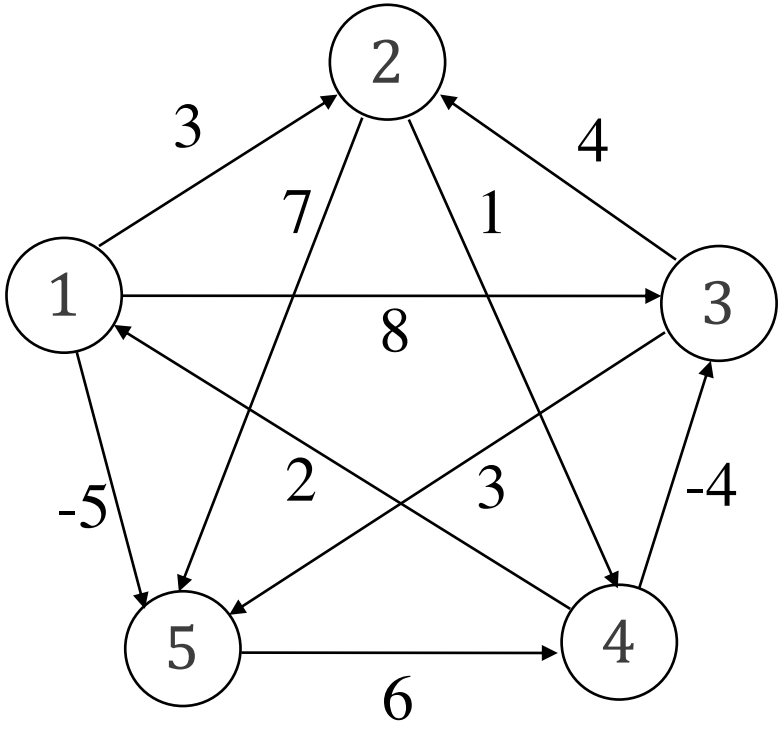
Only go though $\{v_1, v_2, v_3, v_4\}$



Example

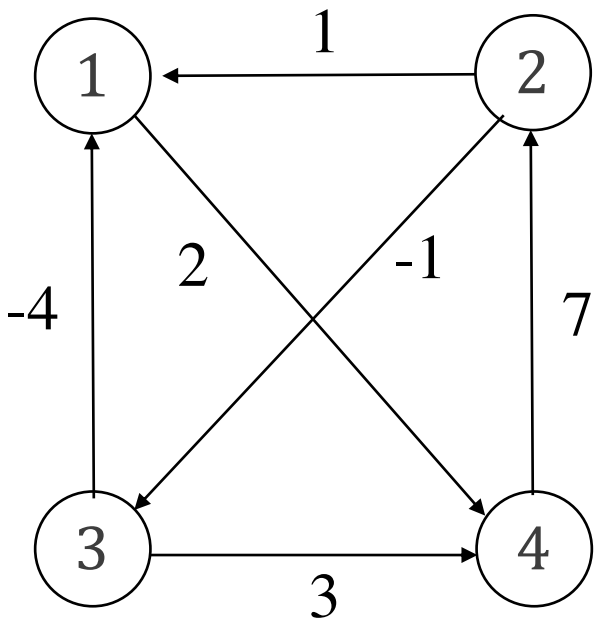
$$D^{(5)} = \begin{bmatrix} 0 & 1 & -3 & 1 & -5 \\ 3 & 0 & -3 & 1 & -2 \\ 7 & 4 & 0 & 5 & 2 \\ 2 & 0 & -4 & 0 & -3 \\ 8 & 6 & 2 & 6 & 0 \end{bmatrix}$$

Only go though $\{v_1, v_2, v_3, v_4, v_5\}$



Classroom Exercise

Run Floyd-Warshall algorithm on the following graph.



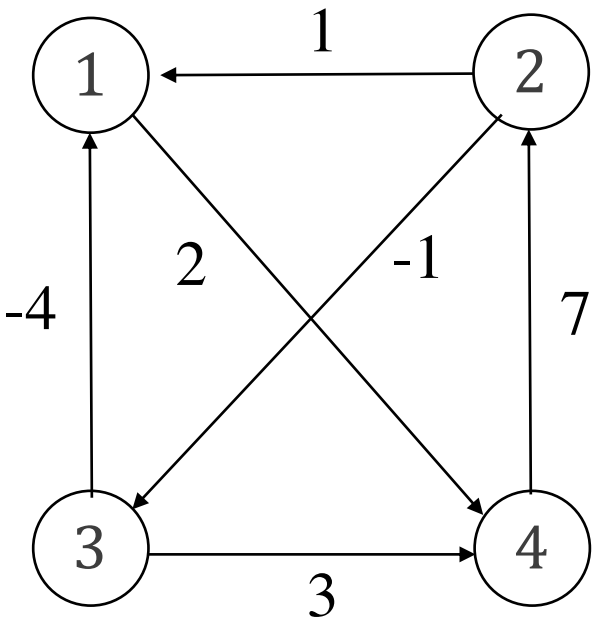
Classroom Exercise

Solution:

$$D^{(0)} = \begin{bmatrix} 0 & \infty & \infty & 2 \\ 1 & 0 & -1 & \infty \\ -4 & \infty & 0 & 3 \\ \infty & 7 & \infty & 0 \end{bmatrix} \quad D^{(1)} = \begin{bmatrix} 0 & \infty & \infty & 2 \\ 1 & 0 & -1 & 3 \\ -4 & \infty & 0 & -2 \\ \infty & 7 & \infty & 0 \end{bmatrix}$$

$$D^{(2)} = \begin{bmatrix} 0 & \infty & \infty & 2 \\ 1 & 0 & -1 & 3 \\ -4 & \infty & 0 & -2 \\ 8 & 7 & 6 & 0 \end{bmatrix} \quad D^{(3)} = \begin{bmatrix} 0 & \infty & \infty & 2 \\ 1 & 0 & -1 & 3 \\ -4 & \infty & 0 & -2 \\ 2 & 7 & 6 & 0 \end{bmatrix}$$

$$D^{(4)} = \begin{bmatrix} 0 & 9 & 8 & 2 \\ 1 & 0 & -1 & 3 \\ -4 & 5 & 0 & -2 \\ 2 & 7 & 6 & 0 \end{bmatrix}$$



Conclusion

After this lecture, you should know:

- What is the difference between BFS and DFS.
- How to prove the correctness of BFS.
- How to obtain a minimal spanning tree.
- How to prove the correctness of Kruskal's and Prim's algorithm.
- What is relaxation.
- How to solve single-source shortest-paths problem.
- How to solve all-pairs shortest-paths problem.

Homework

- Page 135-137

  8.2 8.5 8.6 8.7

  8.8 8.9 8.10 8.19

  8.22 8.27

Experiment

- There are n different types of items.
- There are m seller, each of them sells one set of items at a certain price (price is for the set). Each of these m people sells either nothing or all the items in the set he has.
- Information of these m sellers is represented by matrix M of size $m \times n$.

$$M_{ij} = \begin{cases} 1 & \text{seller } i \text{ sells item } j \\ 0 & \text{otherwise} \end{cases}$$

- The seller i will sell set of all his items at price C_i.
- You want to buy at least one item of each type (it is guaranteed that it is always possible to do so). Use graph algorithms to find the minimal cost needed to pay to achieve it.

Experiment

- Input:
  - The first line contains two integers n, m separated by space.
  - The $(i+1)$th $(1 \leq i \leq m)$ line contains $(n+1)$ integers $M_{i,1}, M_{i,2}, M_{i,3}, \ldots, M_{i,n}, C_i$.
- Output: Minimal cost.
- Sample:

Input	Output
3 3 0 1 1 2 1 0 0 3 1 1 1 9	5

谢谢

有问题欢迎随时跟我讨论

厦门大学信息学院
SCHOOL OF INFORMATICS XIAMEN UNIVERSITY

厦门大学计算机科学系
Computer Science Department of Xiamen University