Part I

Breadth First Search

Overview

(A) **BFS** is obtained from **BasicSearch** by processing edges using a data structure called a **queue**.

(B) It processes the vertices in the graph in the order of their shortest distance from the vertex **s** (the start vertex).

As such...

- **DFS** good for exploring graph structure
- **BFS** good for exploring *distances*

Queues

A **queue** is a list of elements which supports the following operations

- **enqueue**: Adds an element to the end of the list
- **dequeue**: Removes an element from the front of the list

Elements are extracted in **first-in first-out (FIFO)** order, i.e., elements are picked in the order in which they were inserted.

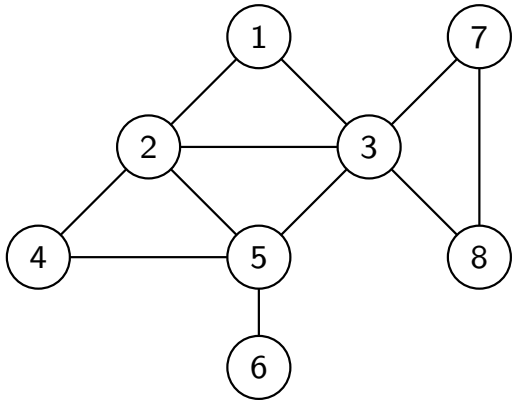
Given (undirected or directed) graph $G = (V, E)$ and node $s \in V$

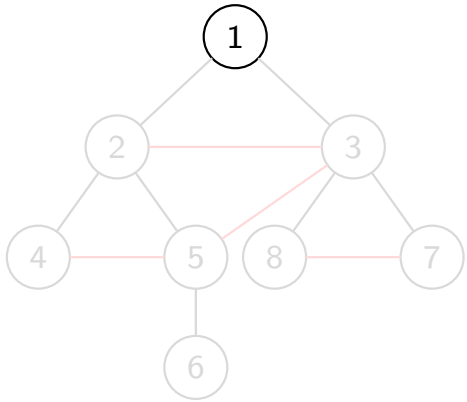
BFS(s)

```
    Mark all vertices as unvisited
    Initialize search tree T to be empty
    Mark vertex s as visited
    set Q to be the empty queue
    enq(s)
    while Q is nonempty do
        u = deq(Q)
        for each vertex v ∈ Adj(u)
            if v is not visited then
                add edge (u, v) to T
                Mark v as visited and enq(v)
```

Proposition

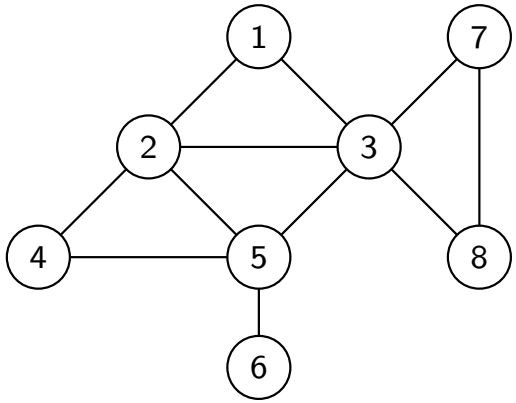
BFS(s) *runs in* $O(n + m)$ *time.*

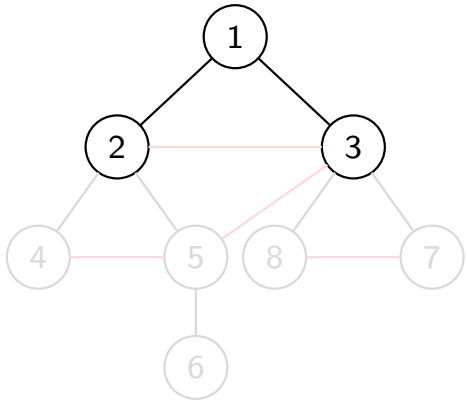




1. [1]
2. [2,3]
3. [3,4,5]
4. [4,5,7,8]
5. [5,7,8]
6. [7,8,6]
7. [8,6]
8. [6]
9. []

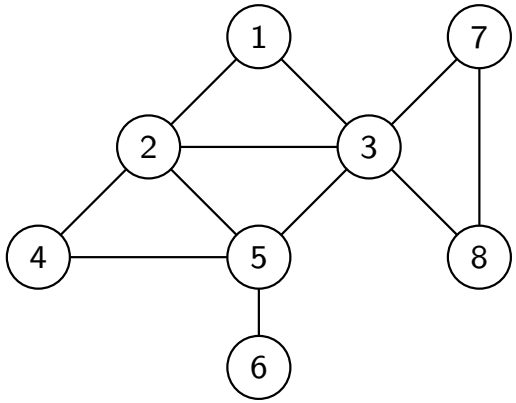
BFS tree is the set of black edges.

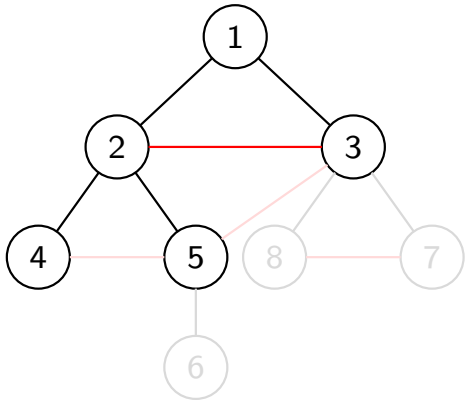




1. [1]
2. [2,3]
3. [3,4,5]
4. [4,5,7,8]
5. [5,7,8]
6. [7,8,6]
7. [8,6]
8. [6]
9. []

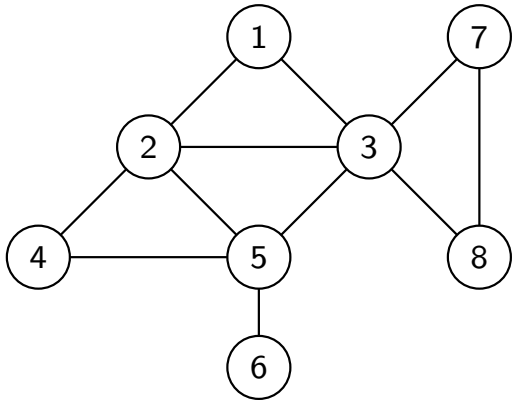
BFS tree is the set of black edges.

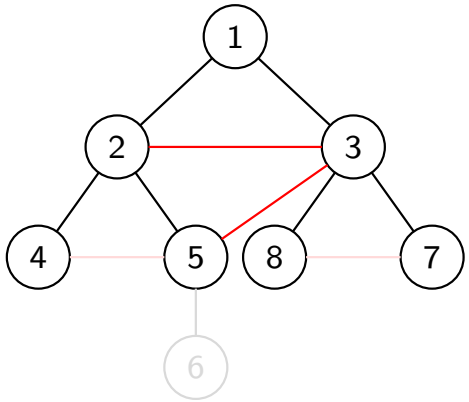




1. [1]
2. [2,3]
3. [3,4,5]
4. [4,5,7,8]
5. [5,7,8]
6. [7,8,6]
7. [8,6]
8. [6]
9. []

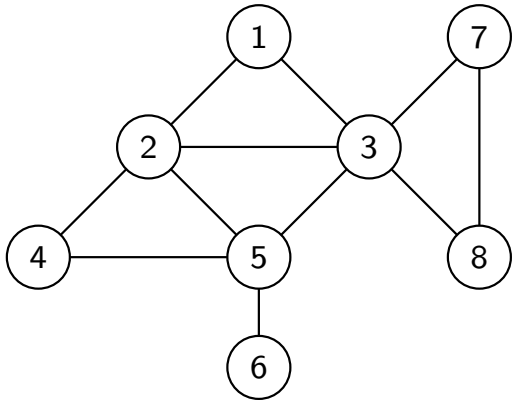
BFS tree is the set of black edges.

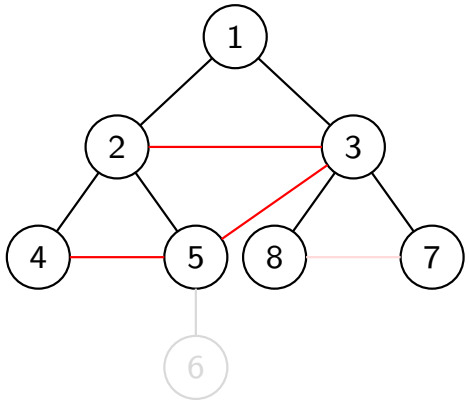




1. [1]
2. [2,3]
3. [3,4,5]
4. [4,5,7,8]
5. [5,7,8]
6. [7,8,6]
7. [8,6]
8. [6]
9. []

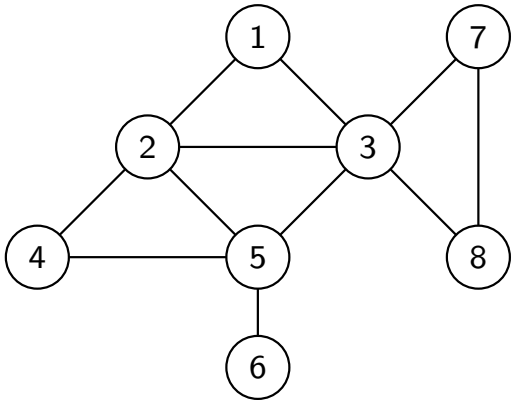
BFS tree is the set of black edges.

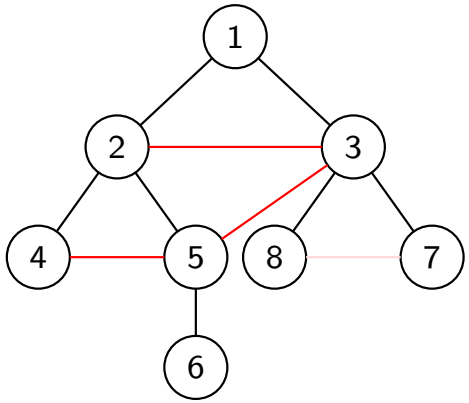




1. [1]
2. [2,3]
3. [3,4,5]
4. [4,5,7,8]
5. [5,7,8]
6. [7,8,6]
7. [8,6]
8. [6]
9. []

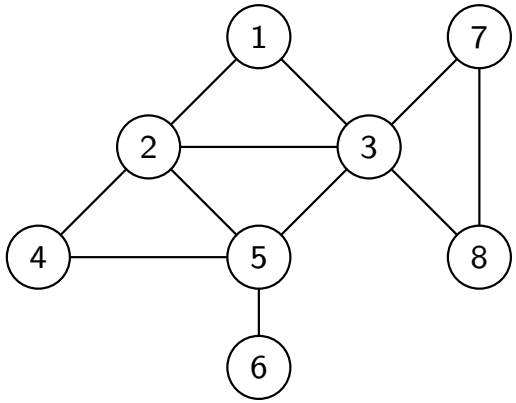
BFS tree is the set of black edges.

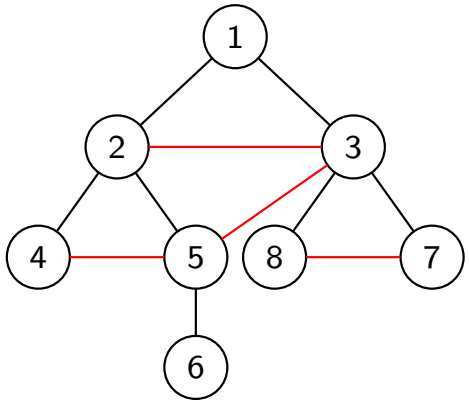




1. [1]
2. [2,3]
3. [3,4,5]
4. [4,5,7,8]
5. [5,7,8]
6. [7,8,6]
7. [8,6]
8. [6]
9. []

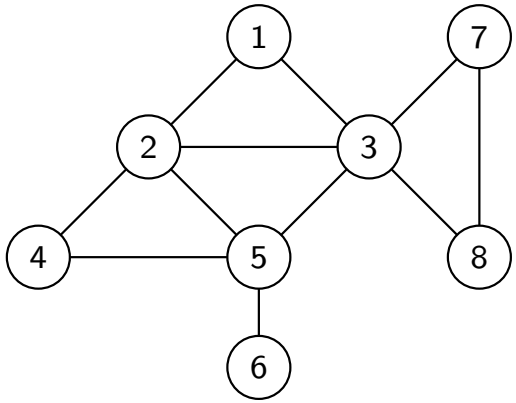
BFS tree is the set of black edges.

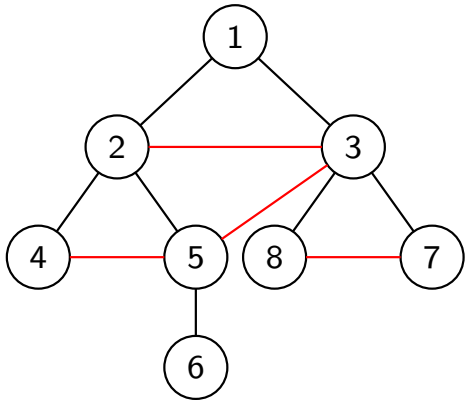




1. [1]
2. [2,3]
3. [3,4,5]
4. [4,5,7,8]
5. [5,7,8]
6. [7,8,6]
7. [8,6]
8. [6]
9. []

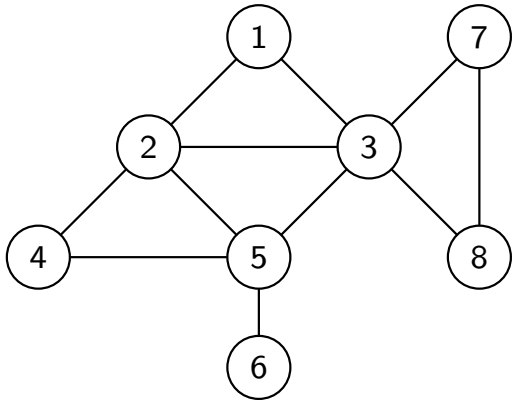
BFS tree is the set of black edges.

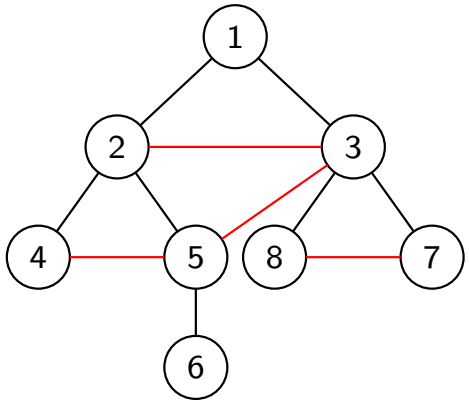




1. [1]
2. [2,3]
3. [3,4,5]
4. [4,5,7,8]
5. [5,7,8]
6. [7,8,6]
7. [8,6]
8. [6]
9. []

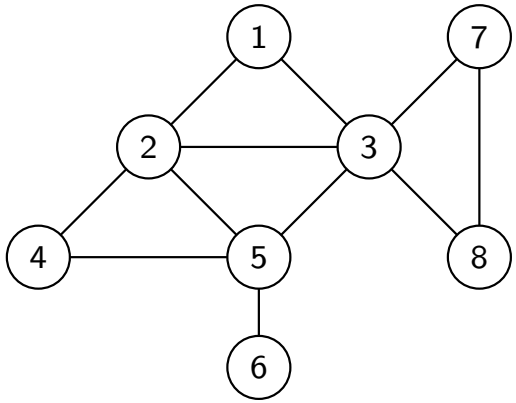
BFS tree is the set of black edges.

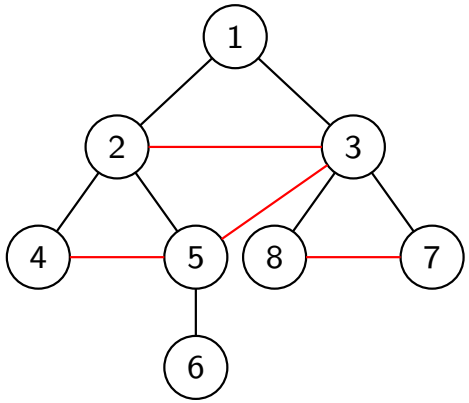




1. [1]
2. [2,3]
3. [3,4,5]
4. [4,5,7,8]
5. [5,7,8]
6. [7,8,6]
7. [8,6]
8. [6]
9. []

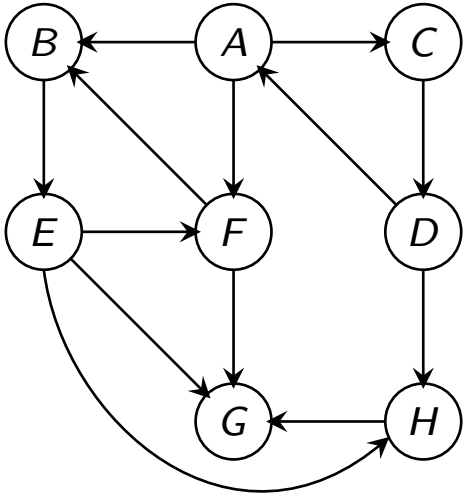
BFS tree is the set of black edges.





1. [1]
2. [2,3]
3. [3,4,5]
4. [4,5,7,8]
5. [5,7,8]
6. [7,8,6]
7. [8,6]
8. [6]
9. []

BFS tree is the set of black edges.


B
A
C
E
F
D
G
H

```
BFS(s)
    Mark all vertices as unvisited and for each v set dist(v) = ∞
    Initialize search tree T to be empty
    Mark vertex s as visited and set dist(s) = 0
    set Q to be the empty queue
    enq(s)
    while Q is nonempty do
        u = deq(Q)
        for each vertex v ∈ Adj(u) do
            if v is not visited do
                add edge (u, v) to T
                Mark v as visited, enq(v)
                and set dist(v) = dist(u) + 1
```

Proposition

The following properties hold upon termination of **BFS(s)**

(A) The search tree contains exactly the set of vertices in the connected component of **s**.

(B) If $\mathbf{dist(u) < dist(v)}$ *then* **u** *is visited before* **v**.

(C) For every vertex **u**, $\mathbf{dist(u)}$ *is indeed the length of shortest path from* **s** *to* **u**.

(D) If **u, v** *are in connected component of* **s** *and* $\mathbf{e = \{u, v\}}$ *is an edge of* **G**, *then either* **e** *is an edge in the search tree, or* $\mathbf{|dist(u) - dist(v)| \leq 1}$.

Proof.

Exercise. □

Proposition

The following properties hold upon termination of **BFS(s)**:

(A) *The search tree contains exactly the set of vertices reachable from* **s**

(B) *If* $\mathbf{dist(u) < dist(v)}$ *then* **u** *is visited before* **v**

(C) *For every vertex* **u**, $\mathbf{dist(u)}$ *is indeed the length of shortest path from* **s** *to* **u**

(D) *If* **u** *is reachable from* **s** *and* $\mathbf{e = (u, v)}$ *is an edge of* **G**, *then either* **e** *is an edge in the search tree, or* $\mathbf{dist(v) - dist(u) \leq 1}$. *Not necessarily the case that* $\mathbf{dist(u) - dist(v) \leq 1}$.

Proof.

Exercise. □

BFSLayers(s):

Mark all vertices as unvisited and initialize T to be empty

Mark s as visited and set $L_0 = \{s\}$

$i = 0$

while L_i is not empty **do**

    initialize L_{i+1} to be an empty list

    for each u in L_i **do**

        for each edge $(u, v) \in Adj(u)$ **do**

        if v is not visited

            mark v as visited

            add (u, v) to tree T

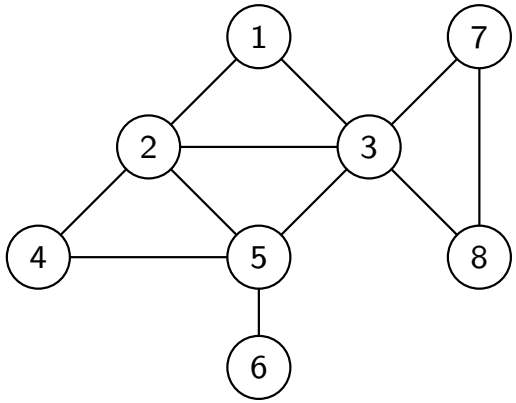
            add v to L_{i+1}

    $i = i + 1$

Running time: $O(n + m)$

```
BFSLayers(s):
Mark all vertices as unvisited and initialize T to be empty
Mark s as visited and set L_0 = {s}
i = 0
while L_i is not empty do
        initialize L_{i+1} to be an empty list
        for each u in L_i do
            for each edge (u, v) ∈ Adj(u) do
            if v is not visited
                    mark v as visited
                    add (u, v) to tree T
                    add v to L_{i+1}
        i = i + 1
```

Running time: $O(n + m)$


1
7
2
3
4
5
8
6

Proposition

The following properties hold on termination of **BFSLayers**(**s**).

- **BFSLayers**(**s**) *outputs a* **BFS** *tree*
- $\mathbf{L_i}$ *is the set of vertices at distance exactly* **i** *from* **s**
- *If* **G** *is undirected, each edge* $\mathbf{e} = \{\mathbf{u}, \mathbf{v}\}$ *is one of three types:*
  - **tree** *edge between two* consecutive *layers*
  - *non-tree* **forward**/**backward** *edge between two consecutive layers*
  - *non-tree* **cross-edge** *with both* **u**, **v** *in same layer*
  - $\Longrightarrow$ *Every edge in the graph is either between two vertices that are either (i) in the same layer, or (ii) in two consecutive layers.*

Proposition

The following properties hold on termination of **BFSLayers(s)**, *if* **G** *is directed.*

For each edge $\mathbf{e = (u, v)}$ *is one of four types:*

- *a* **tree** *edge between consecutive layers,* $\mathbf{u \in L_i, v \in L_{i+1}}$ *for some* $\mathbf{i \geq 0}$
- *a non-tree* **forward** *edge between consecutive layers*
- *a non-tree* **backward** *edge*
- *a* **cross-edge** *with both* **u, v** *in same layer*

Part III

Shortest Paths and Dijkstra's Algorithm

Shortest Path Problems

Input A (undirected or directed) graph $\mathbf{G = (V, E)}$ with edge lengths (or costs). For edge $\mathbf{e = (u, v)}$, $\boldsymbol{\ell(e) = \ell(u, v)}$ is its length.

- Given nodes $\mathbf{s, t}$ find shortest path from $\mathbf{s}$ to $\mathbf{t}$.
- Given node $\mathbf{s}$ find shortest path from $\mathbf{s}$ to all other nodes.
- Find shortest paths for all pairs of nodes.

Many applications!

Shortest Path Problems

Input A (undirected or directed) graph $\mathbf{G = (V, E)}$ with edge lengths (or costs). For edge $\mathbf{e = (u, v)}$, $\boldsymbol{\ell(e) = \ell(u, v)}$ is its length.

- Given nodes $\mathbf{s, t}$ find shortest path from $\mathbf{s}$ to $\mathbf{t}$.
- Given node $\mathbf{s}$ find shortest path from $\mathbf{s}$ to all other nodes.
- Find shortest paths for all pairs of nodes.

Many applications!

Lengths

Single-Source Shortest Path Problems

Input A (undirected or directed) graph $\mathbf{G = (V, E)}$ with non-negative edge lengths. For edge $\mathbf{e = (u, v)}$, $\boldsymbol{\ell(e) = \ell(u, v)}$ is its length.

- Given nodes $\mathbf{s, t}$ find shortest path from $\mathbf{s}$ to $\mathbf{t}$.
- Given node $\mathbf{s}$ find shortest path from $\mathbf{s}$ to all other nodes.

- Restrict attention to directed graphs
- Undirected graph problem can be reduced to directed graph problem - how?
  - Given undirected graph $\mathbf{G}$, create a new directed graph $\mathbf{G'}$ by replacing each edge $\{\mathbf{u, v}\}$ in $\mathbf{G}$ by $\mathbf{(u, v)}$ and $\mathbf{(v, u)}$ in $\mathbf{G'}$.
  - set $\ell(u, v) = \ell(v, u) = \ell(\{u, v\})$

Lengths

Single-Source Shortest Path Problems

Input A (undirected or directed) graph $\mathbf{G = (V, E)}$ with non-negative edge lengths. For edge $\mathbf{e = (u, v)}$, $\boldsymbol{\ell(e) = \ell(u, v)}$ is its length.

- Given nodes $\mathbf{s, t}$ find shortest path from $\mathbf{s}$ to $\mathbf{t}$.
- Given node $\mathbf{s}$ find shortest path from $\mathbf{s}$ to all other nodes.

- Restrict attention to directed graphs
- Undirected graph problem can be reduced to directed graph problem - how?
  - Given undirected graph $\mathbf{G}$, create a new directed graph $\mathbf{G'}$ by replacing each edge $\{\mathbf{u, v}\}$ in $\mathbf{G}$ by $\mathbf{(u, v)}$ and $\mathbf{(v, u)}$ in $\mathbf{G'}$.
  - set $\ell(u, v) = \ell(v, u) = \ell(\{u, v\})$

Lengths

Single-Source Shortest Path Problems

Input A (undirected or directed) graph $\mathbf{G = (V, E)}$ with non-negative edge lengths. For edge $\mathbf{e = (u, v)}$, $\boldsymbol{\ell(e) = \ell(u, v)}$ is its length.

- Given nodes $\mathbf{s, t}$ find shortest path from $\mathbf{s}$ to $\mathbf{t}$.
- Given node $\mathbf{s}$ find shortest path from $\mathbf{s}$ to all other nodes.

- Restrict attention to directed graphs
- Undirected graph problem can be reduced to directed graph problem - how?
  - Given undirected graph $\mathbf{G}$, create a new directed graph $\mathbf{G'}$ by replacing each edge $\mathbf{\{u, v\}}$ in $\mathbf{G}$ by $\mathbf{(u, v)}$ and $\mathbf{(v, u)}$ in $\mathbf{G'}$.
  - set $\boldsymbol{\ell(u, v) = \ell(v, u) = \ell(\{u, v\})}$

Special case: All edge lengths are 1.

- Run **BFS**(s) to get shortest path distances from s to all other nodes.
- $O(m + n)$ time algorithm.

Special case: Suppose $\ell(e)$ is an integer for all e?
Can we use **BFS**? Reduce to unit edge-length problem by placing $\ell(e) - 1$ dummy nodes on e

Let $L = \max_e \ell(e)$. New graph has $O(mL)$ edges and $O(mL + n)$ nodes. **BFS** takes $O(mL + n)$ time. Not efficient if L is large.

Special case: All edge lengths are **1**.

- Run **BFS(s)** to get shortest path distances from s to all other nodes.
- **O(m + n)** time algorithm.

Special case: Suppose $\ell(e)$ is an integer for all **e**?
Can we use **BFS**? Reduce to unit edge-length problem by placing $\ell(e) - 1$ dummy nodes on **e**

Let $L = \max_e \ell(e)$. New graph has **O(mL)** edges and **O(mL + n)** nodes. **BFS** takes **O(mL + n)** time. Not efficient if **L** is large.

Special case: All edge lengths are $\mathbf{1}$.

- Run **BFS**($\mathbf{s}$) to get shortest path distances from s to all other nodes.
- $\mathbf{O(m + n)}$ time algorithm.

Special case: Suppose $\boldsymbol{\ell(e)}$ is an integer for all $\mathbf{e}$?
Can we use **BFS**? Reduce to unit edge-length problem by placing $\ell(e) - 1$ dummy nodes on $\mathbf{e}$

Let $\mathbf{L = \max_e \ell(e)}$. New graph has $\mathbf{O(mL)}$ edges and $\mathbf{O(mL + n)}$ nodes. **BFS** takes $\mathbf{O(mL + n)}$ time. Not efficient if $\mathbf{L}$ is large.

Special case: All edge lengths are $\mathbf{1}$.

- Run **BFS**($\mathbf{s}$) to get shortest path distances from s to all other nodes.
- $\mathbf{O(m + n)}$ time algorithm.

Special case: Suppose $\boldsymbol{\ell(e)}$ is an integer for all $\mathbf{e}$?
Can we use **BFS**? Reduce to unit edge-length problem by placing $\boldsymbol{\ell(e) - 1}$ dummy nodes on $\mathbf{e}$

Let $\mathbf{L = \max_e \ell(e)}$. New graph has $\mathbf{O(mL)}$ edges and $\mathbf{O(mL + n)}$ nodes. **BFS** takes $\mathbf{O(mL + n)}$ time. Not efficient if $\mathbf{L}$ is large.

Special case: All edge lengths are $\mathbf{1}$.

- Run **BFS**($\mathbf{s}$) to get shortest path distances from s to all other nodes.
- $\mathbf{O(m + n)}$ time algorithm.

Special case: Suppose $\boldsymbol{\ell(e)}$ is an integer for all $\mathbf{e}$?
Can we use **BFS**? Reduce to unit edge-length problem by placing $\boldsymbol{\ell(e) - 1}$ dummy nodes on $\mathbf{e}$

Let $\mathbf{L = \max_e \ell(e)}$. New graph has $\mathbf{O(mL)}$ edges and $\mathbf{O(mL + n)}$ nodes. **BFS** takes $\mathbf{O(mL + n)}$ time. Not efficient if $\mathbf{L}$ is large.

Why does **BFS** work?

BFS(s) explores nodes in increasing distance from **s**

Lemma

Let **G** *be a directed graph with non-negative edge lengths. Let* $\mathbf{dist(s, v)}$ *denote the shortest path length from* **s** *to* **v**. *If* $\mathbf{s = v_0 \to v_1 \to v_2 \to \ldots \to v_k}$ *is a shortest path from* **s** *to* $\mathbf{v_k}$ *then for* $\mathbf{1 \leq i < k}$*:*

- $\mathbf{s = v_0 \to v_1 \to v_2 \to \ldots \to v_i}$ *is a shortest path from* **s** *to* $\mathbf{v_i}$
- $\mathbf{dist(s, v_i) \leq dist(s, v_k)}$.

Proof.

Suppose not. Then for some $\mathbf{i < k}$ there is a path $\mathbf{P'}$ from **s** to $\mathbf{v_i}$ of length strictly less than that of $\mathbf{s = v_0 \to v_1 \to \ldots \to v_i}$. Then $\mathbf{P'}$ concatenated with $\mathbf{v_i \to v_{i+1} \ldots \to v_k}$ contains a strictly shorter

Why does **BFS** work?

BFS(s) explores nodes in increasing distance from **s**

Lemma

Let G *be a directed graph with non-negative edge lengths. Let* $\text{dist}(s, v)$ *denote the shortest path length from* s *to* v*. If* $s = v_0 \to v_1 \to v_2 \to \ldots \to v_k$ *is a shortest path from* s *to* v_k *then for* $1 \leq i < k$*:*

- $s = v_0 \to v_1 \to v_2 \to \ldots \to v_i$ *is a shortest path from* s *to* v_i
- $\text{dist}(s, v_i) \leq \text{dist}(s, v_k)$.

Proof.

Suppose not. Then for some $i < k$ there is a path P' from s to v_i of length strictly less than that of $s = v_0 \to v_1 \to \ldots \to v_i$. Then P' concatenated with $v_i \to v_{i+1} \ldots \to v_k$ contains a strictly shorter

Why does **BFS** work?

BFS(s) explores nodes in increasing distance from **s**

Lemma

Let G *be a directed graph with non-negative edge lengths. Let* $\mathbf{dist(s, v)}$ *denote the shortest path length from* s *to* v*. If* $s = v_0 \to v_1 \to v_2 \to \ldots \to v_k$ *is a shortest path from* s *to* v_k *then for* $1 \leq i < k$*:*

- $s = v_0 \to v_1 \to v_2 \to \ldots \to v_i$ *is a shortest path from* s *to* v_i
- $\mathbf{dist(s, v_i) \leq dist(s, v_k)}$.

Proof.

Suppose not. Then for some $i < k$ there is a path P' from s to v_i of length strictly less than that of $s = v_0 \to v_1 \to \ldots \to v_i$. Then P' concatenated with $v_i \to v_{i+1} \ldots \to v_k$ contains a strictly shorter

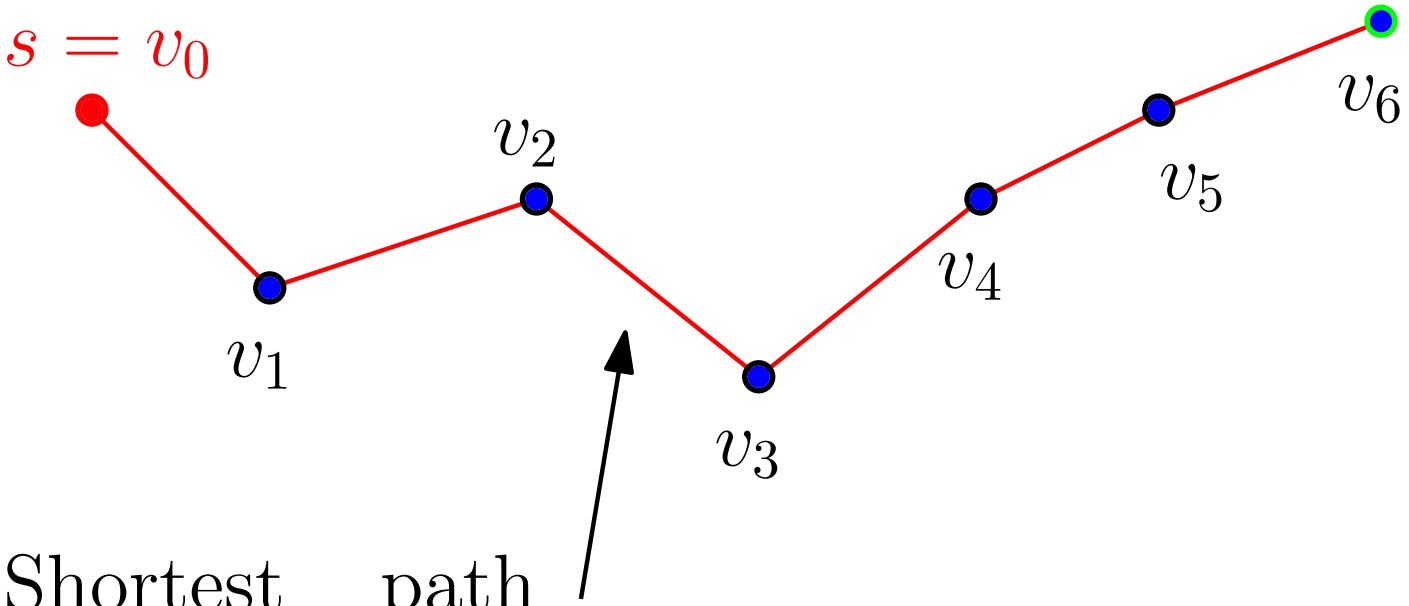
Lemma

Let G *be a directed graph with non-negative edge lengths. Let* $\mathrm{dist}(s, v)$ *denote the shortest path length from* s *to* v*. If* $s = v_0 \to v_1 \to v_2 \to \ldots \to v_k$ *is a shortest path from* s *to* v_k *then for* $1 \leq i < k$*:*

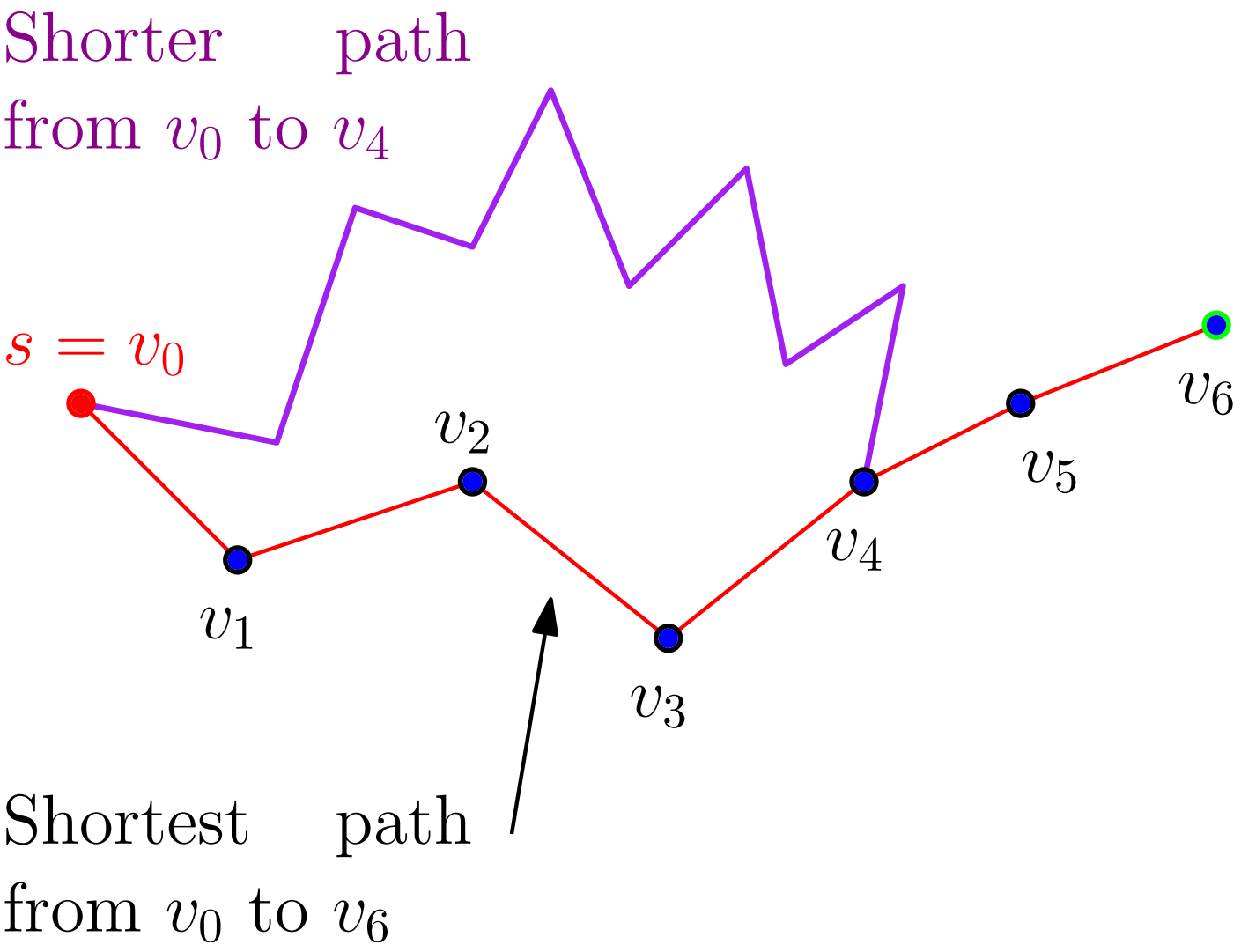
- $s = v_0 \to v_1 \to v_2 \to \ldots \to v_i$ *is a shortest path from* s *to* v_i
- $\mathrm{dist}(s, v_i) \leq \mathrm{dist}(s, v_k)$.

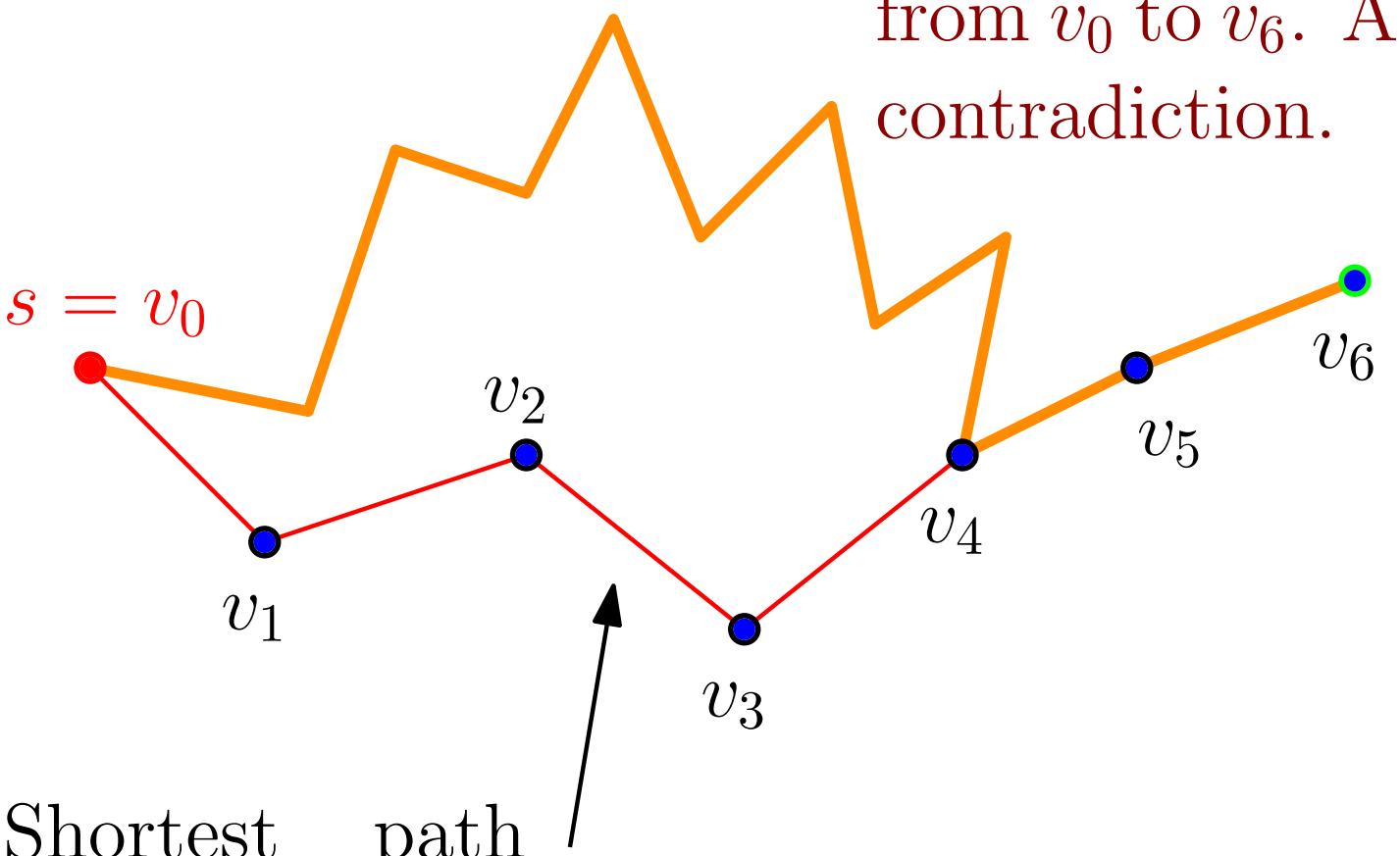
Proof.

Suppose not. Then for some $i < k$ there is a path P' from s to v_i of length strictly less than that of $s = v_0 \to v_1 \to \ldots \to v_i$. Then P' concatenated with $v_i \to v_{i+1} \ldots \to v_k$ contains a strictly shorter path to v_k than $s = v_0 \to v_1 \ldots \to v_k$. □



Shortest path from v_0 to v_6


Shorter path
from v_0 to v_4
$s = v_0$
v_1
v_2
v_3
v_4
v_5
v_6
Shortest path
from v_0 to v_6


A shorter path from v_0 to v_6. A contradiction.
$s = v_0$
v_1
v_2
v_3
v_4
v_5
v_6
Shortest path from v_0 to v_6

Explore vertices in increasing order of distance from s:
(For simplicity assume that nodes are at different distances from s and that no edge has zero length)

Initialize for each node v, $\mathbf{dist(s, v) = \infty}$
Initialize $S = \emptyset$,
for $i = 1$ to $|V|$ **do**
    (* Invariant: S contains the $i - 1$ closest nodes to s *)
    Among nodes in $V \setminus S$, find the node v that is the
        ith closest to s
    Update $\mathbf{dist(s, v)}$
    $S = S \cup \{v\}$

How can we implement the step in the for loop?

Explore vertices in increasing order of distance from s:
(For simplicity assume that nodes are at different distances from s and that no edge has zero length)

Initialize for each node v, $\mathbf{dist(s, v) = \infty}$
Initialize $S = \emptyset$,
for $i = 1$ to $|V|$ **do**
    (* Invariant: S contains the $i - 1$ closest nodes to s *)
    Among nodes in $V \setminus S$, find the node v that is the
        ith closest to s
    Update $\mathbf{dist(s, v)}$
    $S = S \cup \{v\}$

How can we implement the step in the for loop?

- **S** contains the **i − 1** closest nodes to **s**
- Want to find the **i**th closest node from **V − S**.

What do we know about the **i**th closest node?

Claim

*Let **P** be a shortest path from **s** to **v** where **v** is the **i**th closest node. Then, all intermediate nodes in **P** belong to **S**.*

Proof.

If **P** had an intermediate node **u** not in **S** then **u** will be closer to **s** than **v**. Implies **v** is not the **i**th closest node to **s** - recall that **S** already has the **i − 1** closest nodes. □

- **S** contains the **i** − **1** closest nodes to **s**
- Want to find the **i**th closest node from **V** − **S**.

What do we know about the **i**th closest node?

Claim

Let **P** *be a shortest path from* **s** *to* **v** *where* **v** *is the* **i***th closest node. Then, all intermediate nodes in* **P** *belong to* **S**.

Proof.

If **P** had an intermediate node **u** not in **S** then **u** will be closer to **s** than **v**. Implies **v** is not the **i**th closest node to **s** - recall that **S** already has the **i** − **1** closest nodes. □

- **S** contains the **i** $-$ **1** closest nodes to **s**
- Want to find the **i**th closest node from **V** $-$ **S**.

What do we know about the **i**th closest node?

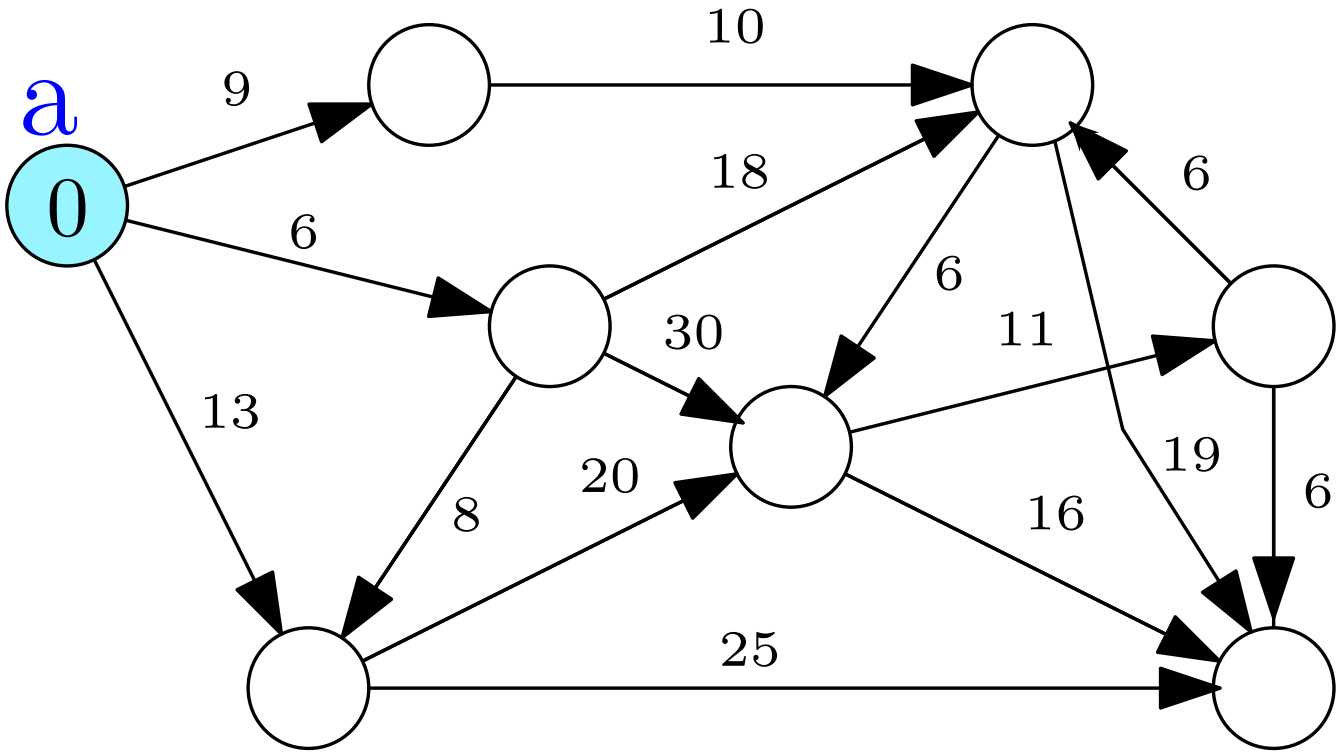
Claim

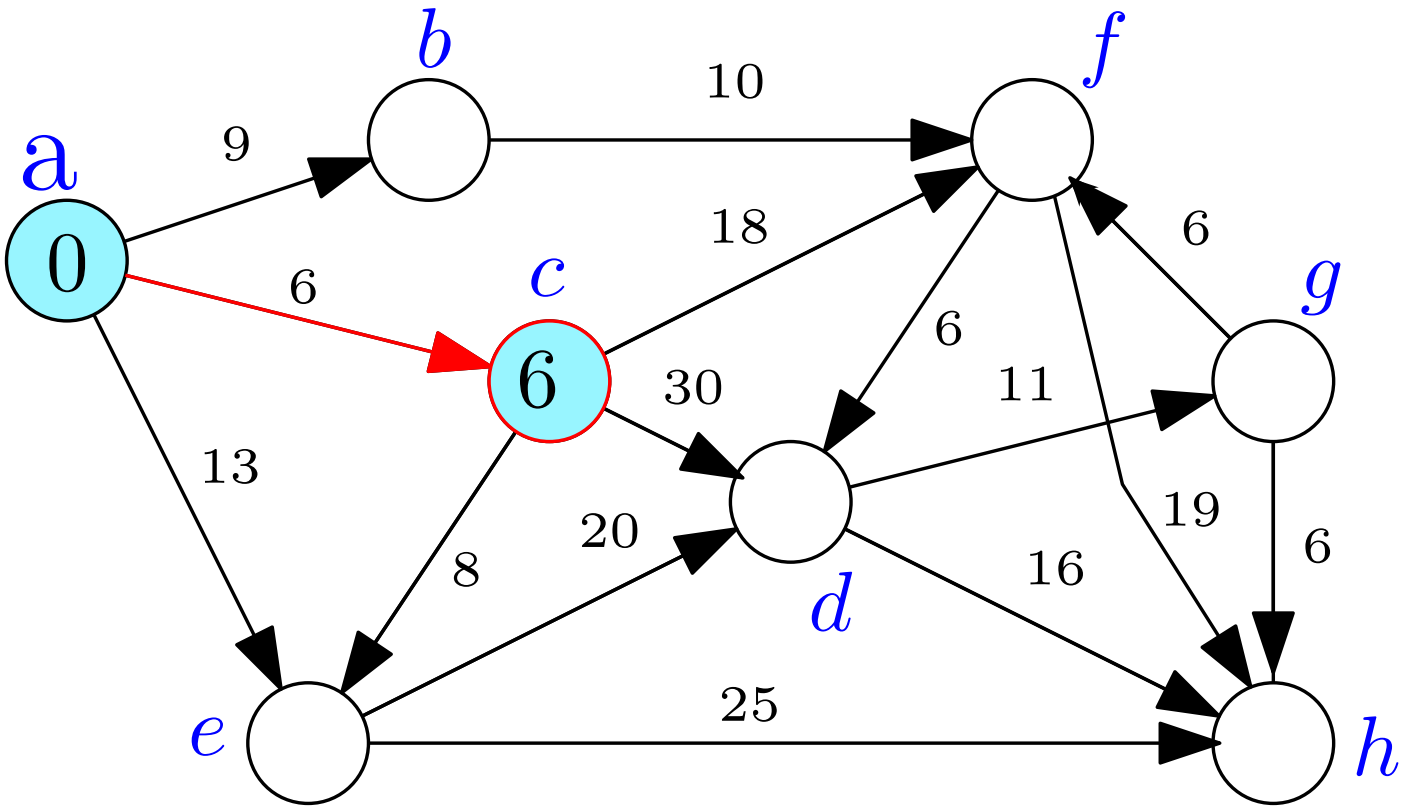
Let **P** *be a shortest path from* **s** *to* **v** *where* **v** *is the* **i***th closest node. Then, all intermediate nodes in* **P** *belong to* **S**.

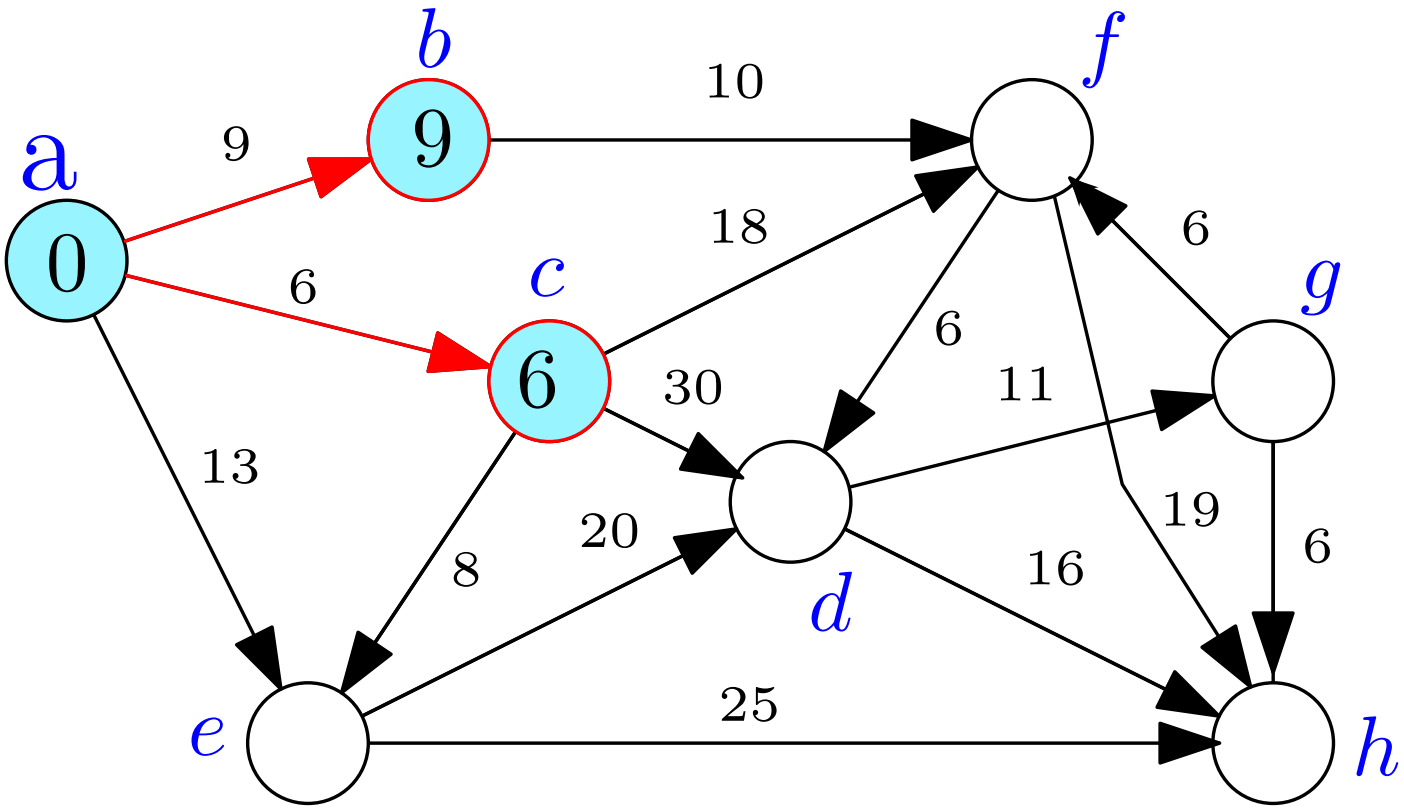
Proof.

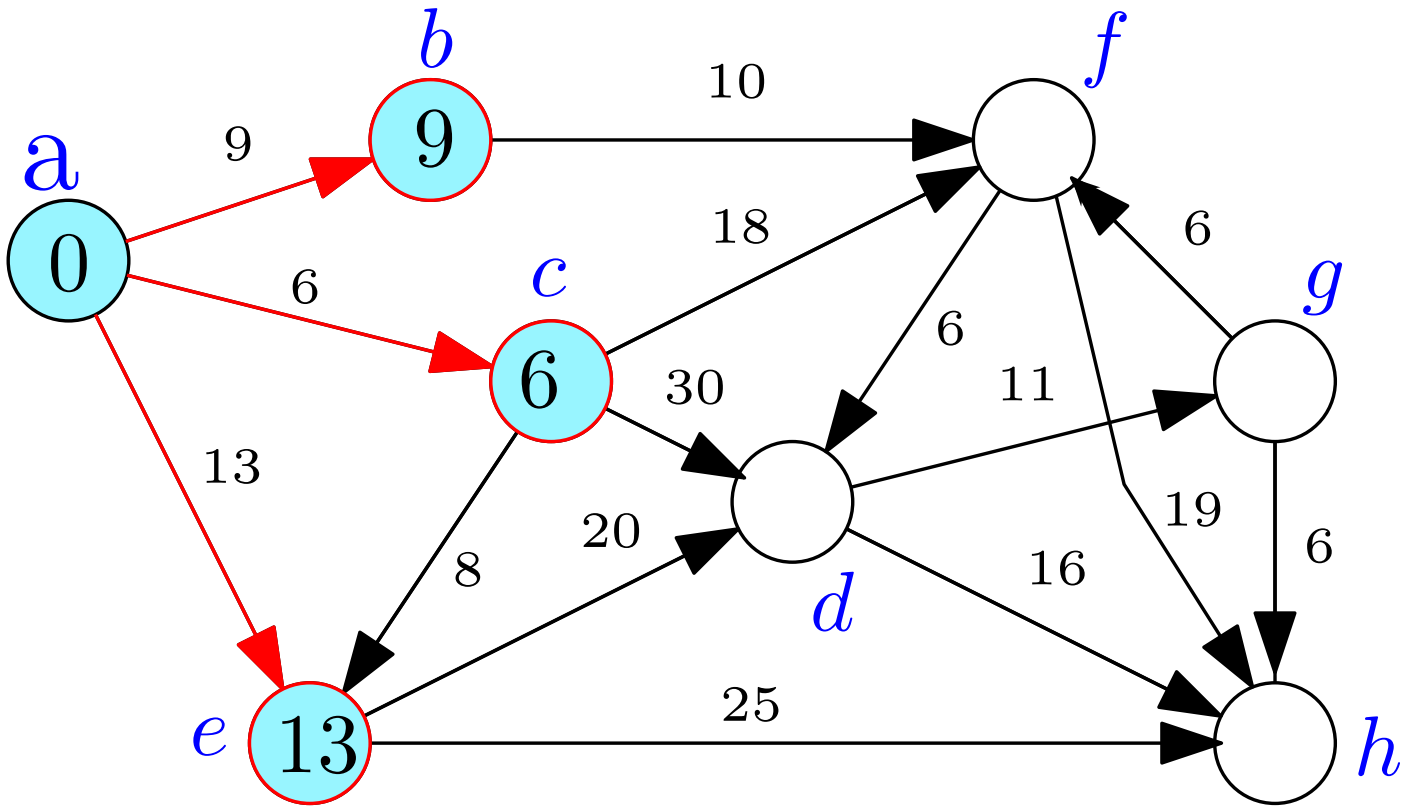
If **P** had an intermediate node **u** not in **S** then **u** will be closer to **s** than **v**. Implies **v** is not the **i**th closest node to **s** - recall that **S** already has the **i** $-$ **1** closest nodes. □

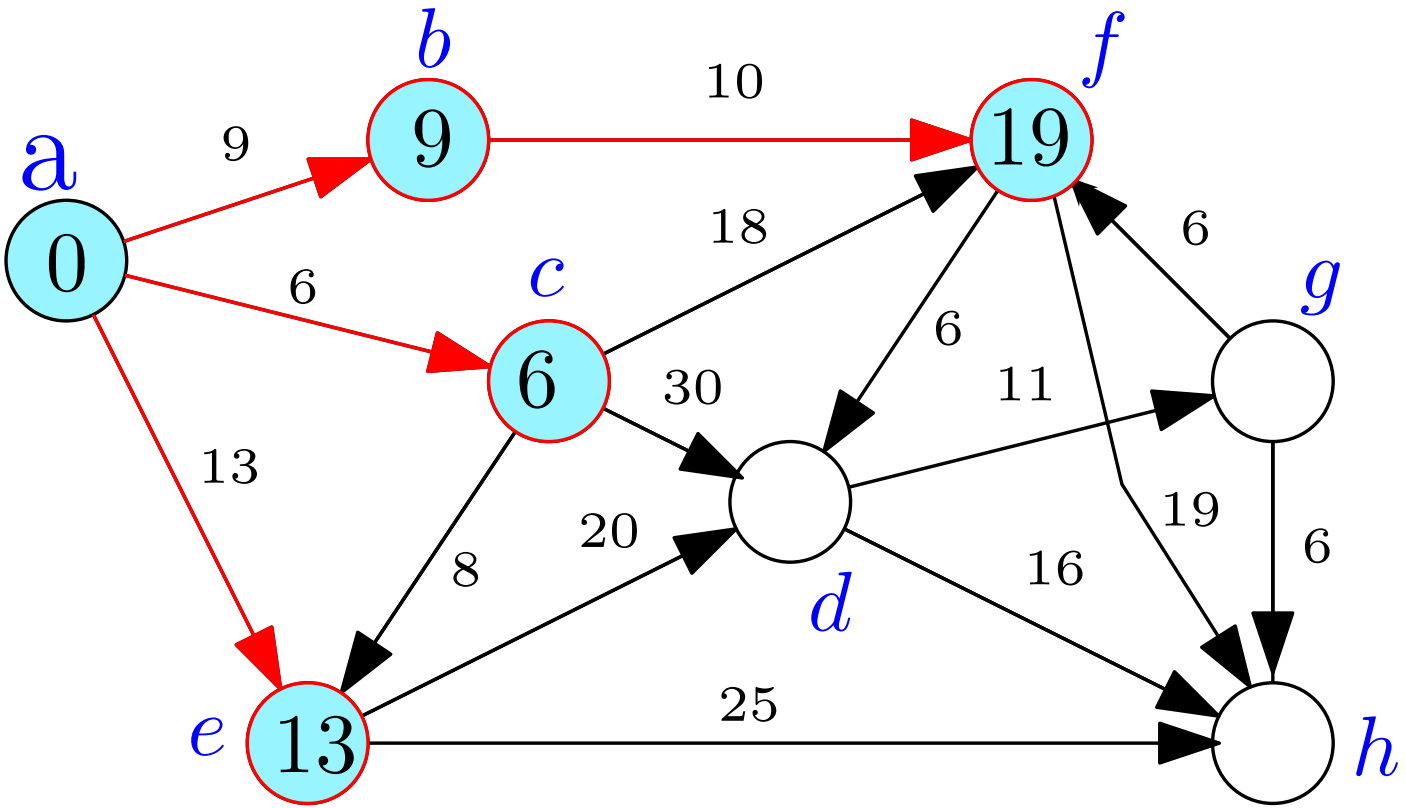
An example

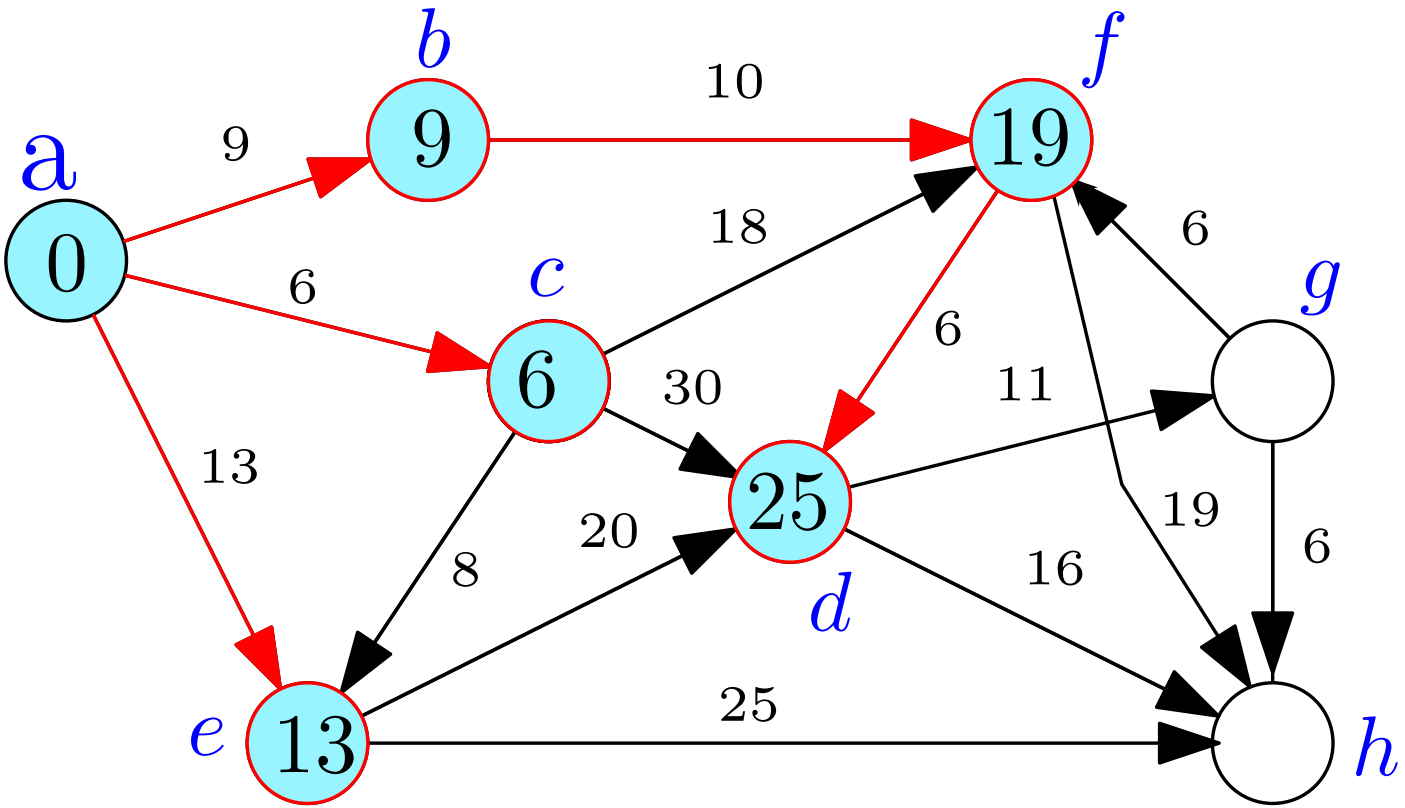


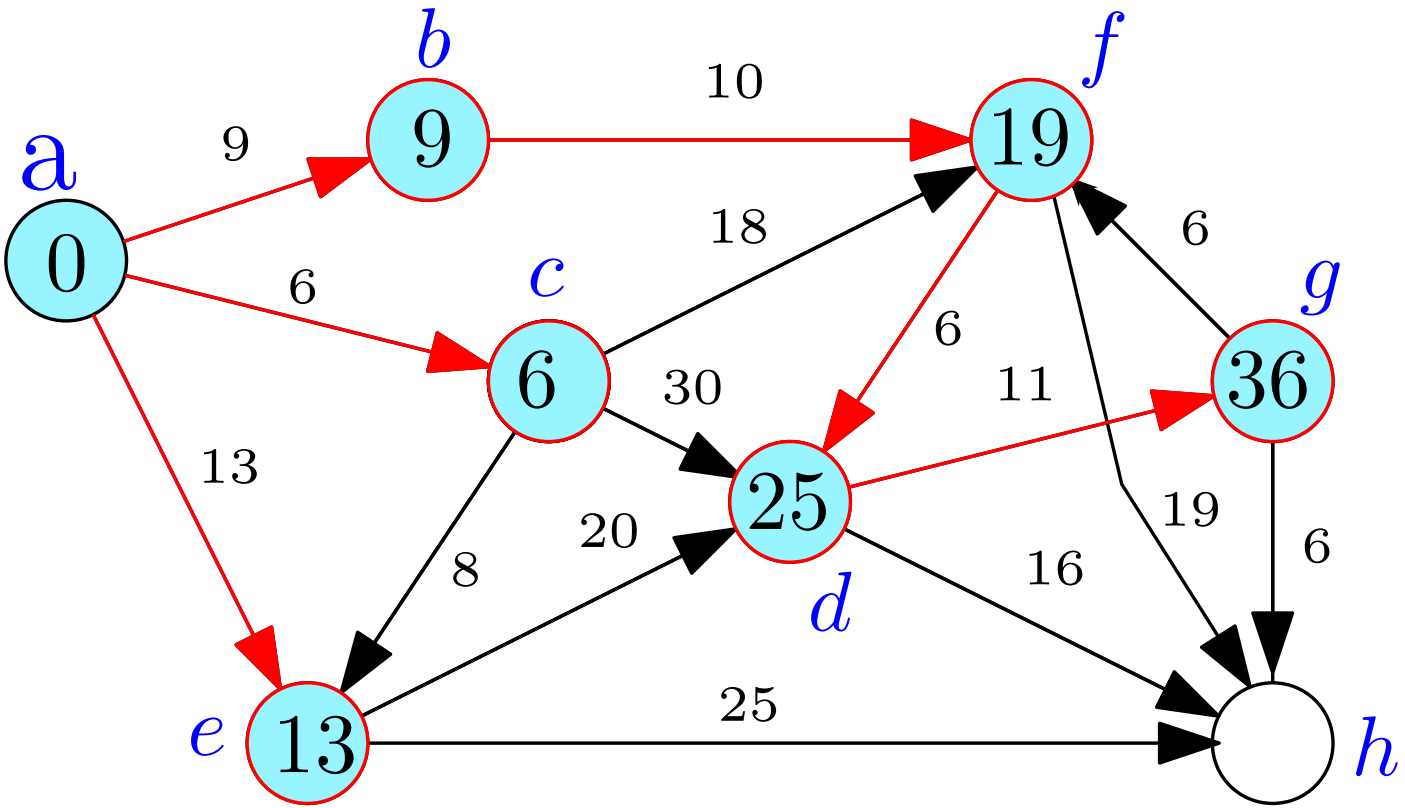

a
0
b
c
6
d
e
f
g
h
9
10
18
6
6
6
30
11
13
20
8
19
16
6
25


a
0
b
9
c
6
d
e
f
g
h
9
6
10
18
13
30
8
20
6
11
16
19
6
25

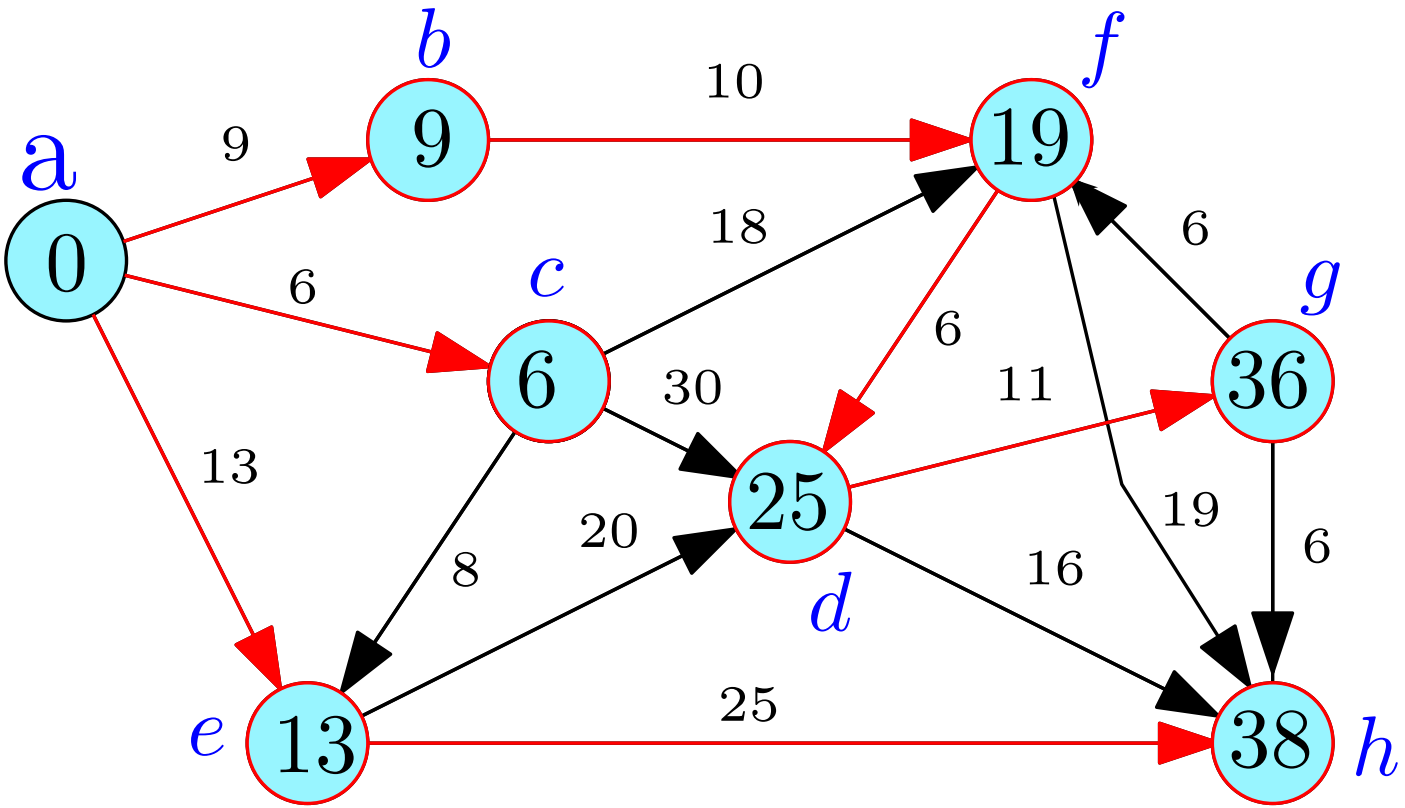

a
0
b
9
c
6
d
e
13
f
g
h
9
10
6
18
6
30
6
11
13
20
8
19
16
6
25

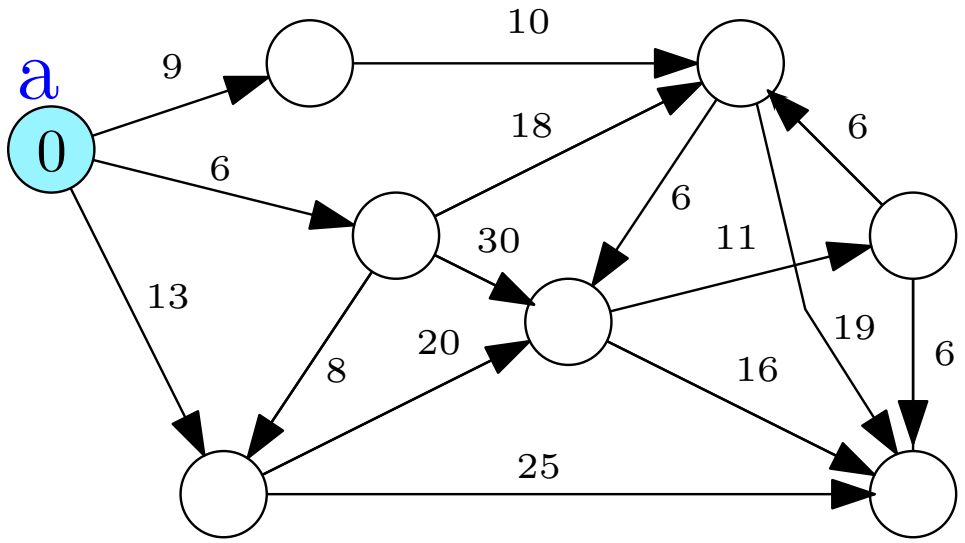

a
0
b
9
c
6
d
e
13
f
19
g
h
9
10
6
18
6
30
11
6
13
20
8
19
6
16
25


a
0
b
9
c
6
d
25
e
13
f
19
g
h
9
10
6
18
6
30
6
11
13
20
8
19
6
16
25


a
0
b
9
c
6
d
25
e
13
f
19
g
36
h
9
10
6
18
6
30
13
8
20
11
6
19
16
6
25

An example





Corollary

The **i***th closest node is adjacent to* **S***.*

- S contains the $i-1$ closest nodes to s
- Want to find the ith closest node from $V-S$.

- For each $u \in V-S$ let $P(s,u,S)$ be a shortest path from s to u using only nodes in S as intermediate vertices.
- Let $d'(s,u)$ be the length of $P(s,u,S)$

Observations: for each $u \in V-S$,

- $dist(s,u) \leq d'(s,u)$ since we are constraining the paths
- $d'(s,u) = \min_{a \in S}(dist(s,a) + \ell(a,u))$ - Why?

Lemma

If v is the ith closest node to s, then $d'(s,v) = dist(s,v)$.

- S contains the $i-1$ closest nodes to s
- Want to find the ith closest node from $V-S$.

- For each $u \in V-S$ let $P(s,u,S)$ be a shortest path from s to u using only nodes in S as intermediate vertices.
- Let $d'(s,u)$ be the length of $P(s,u,S)$

Observations: for each $u \in V-S$,

- $\mathrm{dist}(s,u) \leq d'(s,u)$ since we are constraining the paths
- $d'(s,u) = \min_{a \in S}(\mathrm{dist}(s,a) + \ell(a,u))$ - Why?

Lemma

If v is the ith closest node to s, then $d'(s,v) = \mathrm{dist}(s,v)$.

- S contains the $i-1$ closest nodes to s
- Want to find the ith closest node from $V-S$.

- For each $u \in V-S$ let $P(s,u,S)$ be a shortest path from s to u using only nodes in S as intermediate vertices.
- Let $d'(s,u)$ be the length of $P(s,u,S)$

Observations: for each $u \in V-S$,

- $\mathrm{dist}(s,u) \leq d'(s,u)$ since we are constraining the paths
- $d'(s,u) = \min_{a \in S}(\mathrm{dist}(s,a) + \ell(a,u))$ - Why?

Lemma

If v is the ith closest node to s, then $d'(s,v) = \mathrm{dist}(s,v)$.

Lemma

If $\mathbf{v}$ *is an* $\mathbf{i}$*th closest node to* $\mathbf{s}$*, then* $\mathbf{d'(s, v) = \mathrm{dist}(s, v)}$.

Proof.

Let $\mathbf{v}$ be the $\mathbf{i}$th closest node to $\mathbf{s}$. Then there is a shortest path $\mathbf{P}$ from $\mathbf{s}$ to $\mathbf{v}$ that contains only nodes in $\mathbf{S}$ as intermediate nodes (see previous claim). Therefore $\mathbf{d'(s, v) = \mathrm{dist}(s, v)}$. □

Lemma

If $\mathbf{v}$ *is an* $\mathbf{i}$*th closest node to* $\mathbf{s}$*, then* $\mathbf{d'(s, v) = dist(s, v)}$.

Corollary

The $\mathbf{i}$*th closest node to* $\mathbf{s}$ *is the node* $\mathbf{v \in V - S}$ *such that* $\mathbf{d'(s, v) = \min_{u \in V - S} d'(s, u)}$.

Proof.

For every node $\mathbf{u \in V - S}$, $\mathbf{dist(s, u) \leq d'(s, u)}$ and for the $\mathbf{i}$th closest node $\mathbf{v}$, $\mathbf{dist(s, v) = d'(s, v)}$. Moreover, $\mathbf{dist(s, u) \geq dist(s, v)}$ for each $\mathbf{u \in V - S}$. □

Initialize for each node v: $\mathbf{dist(s,v) = \infty}$
Initialize S = $\emptyset$, $\mathbf{d'(s,s) = 0}$
for $\mathbf{i = 1}$ to $\mathbf{|V|}$ **do**
    (* Invariant: S contains the i-1 closest nodes to s *)
    (* Invariant: d'(s,u) is shortest path distance from u to s using only S as intermediate nodes*)
    Let v be such that d'(s,v) = $\mathbf{\min_{u \in V-S}}$ d'(s,u)
    $\mathbf{dist(s,v) = d'(s,v)}$
    $\mathbf{S = S \cup \{v\}}$
    for each node u in $\mathbf{V \setminus S}$
        compute d'(s,u) = $\mathbf{\min_{a \in S}}$ $(\mathbf{dist(s,a) + \ell(a,u)})$

Correctness: By induction on i using previous lemmas.
Running time: $\mathbf{O(n \cdot (n+m))}$ time.

- n outer iterations. In each iteration, $\mathbf{d'(s,u)}$ for each u by scanning all edges out of nodes in S; $\mathbf{O(m+n)}$ time/iteration.

Initialize for each node v: $\mathbf{dist(s,v) = \infty}$
Initialize S = $\emptyset$, $\mathbf{d'(s,s) = 0}$
for $\mathbf{i = 1}$ to $\mathbf{|V|}$ **do**
    (* Invariant: S contains the i-1 closest nodes to s *)
    (* Invariant: d'(s,u) is shortest path distance from u to s using only S as intermediate nodes*)
    Let v be such that d'(s,v) = $\mathbf{\min_{u \in V-S}}$ d'(s,u)
    $\mathbf{dist(s,v) = d'(s,v)}$
    $\mathbf{S = S \cup \{v\}}$
    for each node $\mathbf{u}$ in $\mathbf{V \setminus S}$
        compute d'(s,u) = $\mathbf{\min_{a \in S}}$ $(\mathbf{dist(s,a) + \ell(a,u)})$

Correctness: By induction on $\mathbf{i}$ using previous lemmas.

Running time: $\mathbf{O(n \cdot (n+m))}$ time.

- $\mathbf{n}$ outer iterations. In each iteration, $\mathbf{d'(s,u)}$ for each $\mathbf{u}$ by scanning all edges out of nodes in $\mathbf{S}$; $\mathbf{O(m+n)}$ time/iteration.

Initialize for each node v: $\mathbf{dist(s,v) = \infty}$
Initialize S = $\emptyset$, $\mathbf{d'(s,s) = 0}$
for $\mathbf{i = 1}$ to $\mathbf{|V|}$ **do**
    (* Invariant: S contains the i-1 closest nodes to s *)
    (* Invariant: d'(s,u) is shortest path distance from u to s using only S as intermediate nodes*)
    Let v be such that d'(s,v) = $\mathbf{\min_{u \in V-S}}$ d'(s,u)
    $\mathbf{dist(s,v) = d'(s,v)}$
    $\mathbf{S = S \cup \{v\}}$
    for each node $\mathbf{u}$ in $\mathbf{V \setminus S}$
        compute d'(s,u) = $\mathbf{\min_{a \in S}}$ $(\mathbf{dist(s,a) + \ell(a,u)})$

Correctness: By induction on $\mathbf{i}$ using previous lemmas.

Running time: $\mathbf{O(n \cdot (n+m))}$ time.

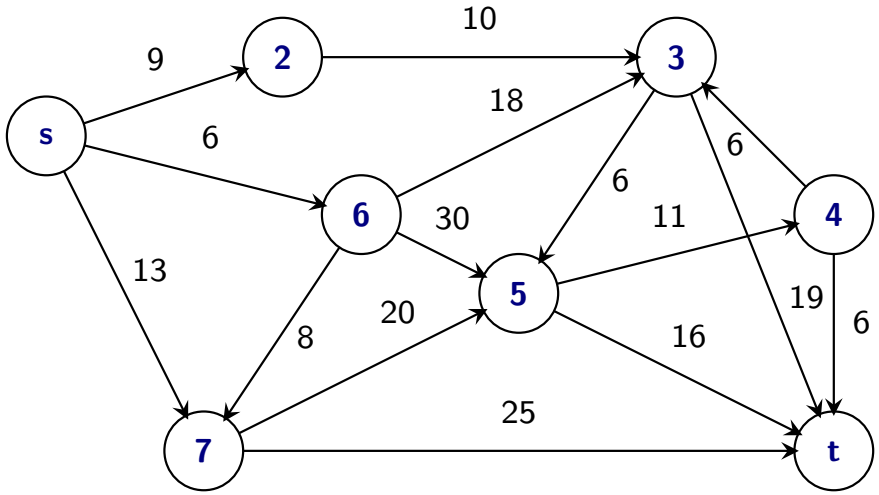
- $\mathbf{n}$ outer iterations. In each iteration, $\mathbf{d'(s,u)}$ for each $\mathbf{u}$ by scanning all edges out of nodes in $\mathbf{S}$; $\mathbf{O(m+n)}$ time/iteration.

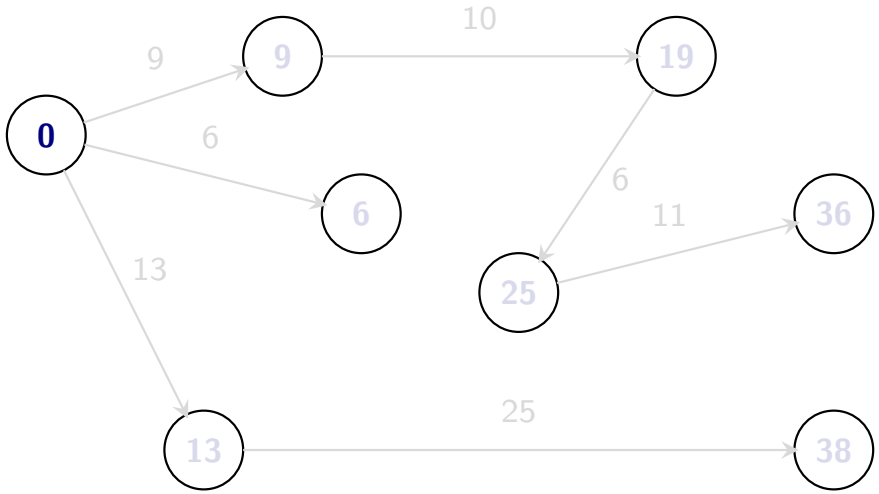
```
Initialize for each node v:  dist(s, v) = ∞
Initialize S = ∅, d'(s, s) = 0
for i = 1 to |V| do
    (* Invariant:  S contains the i-1 closest nodes to s *)
    (* Invariant:  d'(s,u) is shortest path distance from u to s
     using only S as intermediate nodes*)
    Let v be such that d'(s,v) = min_{u∈V−S} d'(s,u)
    dist(s, v) = d'(s, v)
    S = S ∪ {v}
    for each node u in V \ S
        compute d'(s,u) = min_{a∈S} (dist(s, a) + ℓ(a, u))
```

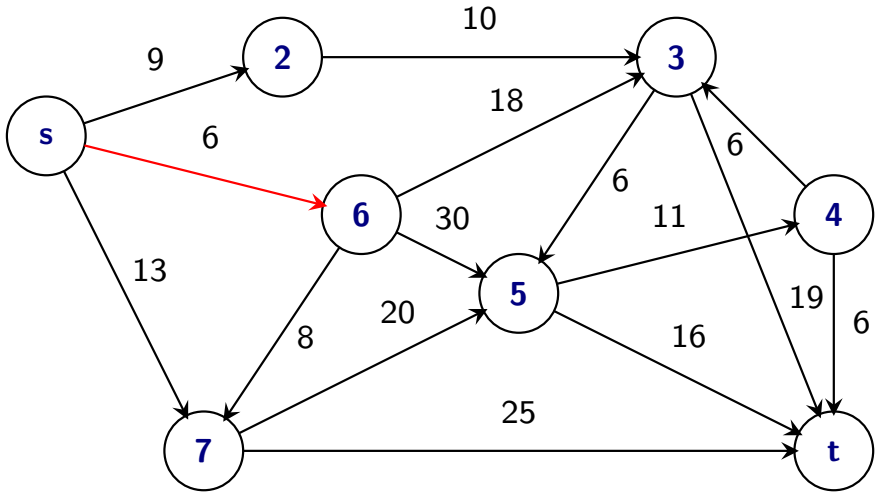
Correctness: By induction on i using previous lemmas.

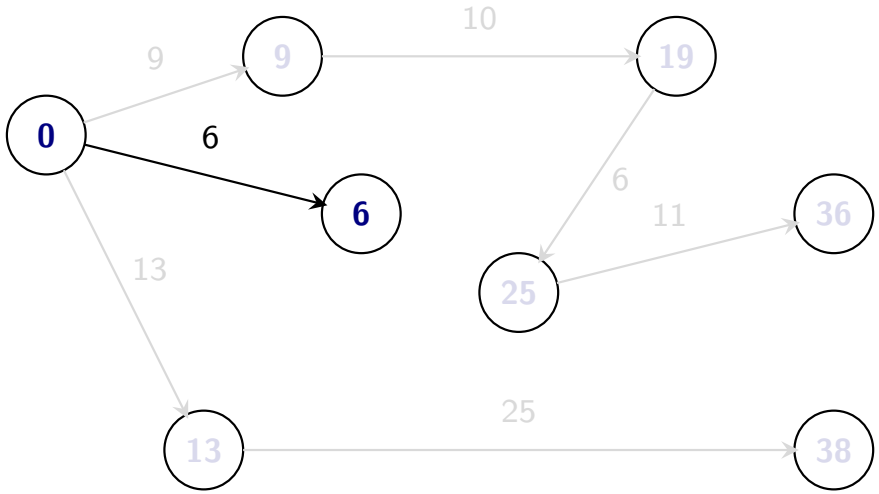
Running time: $O(n \cdot (n + m))$ time.

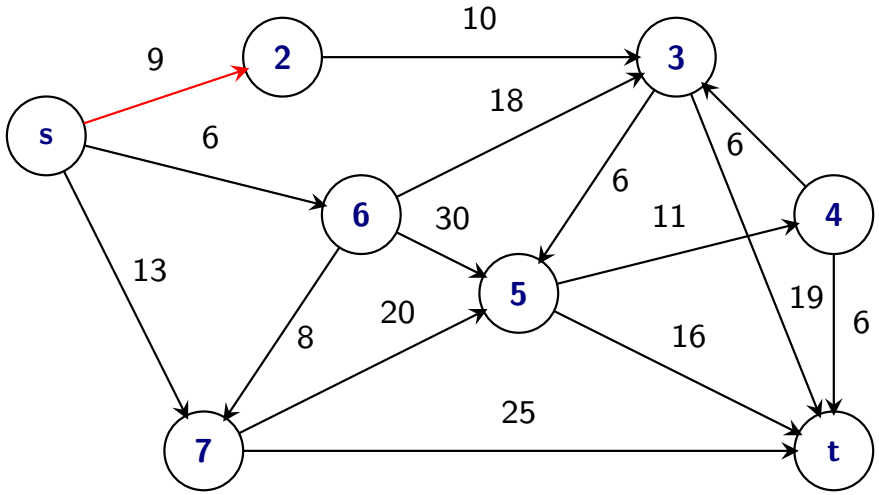
- n outer iterations. In each iteration, $d'(s, u)$ for each u by scanning all edges out of nodes in S; $O(m + n)$ time/iteration.

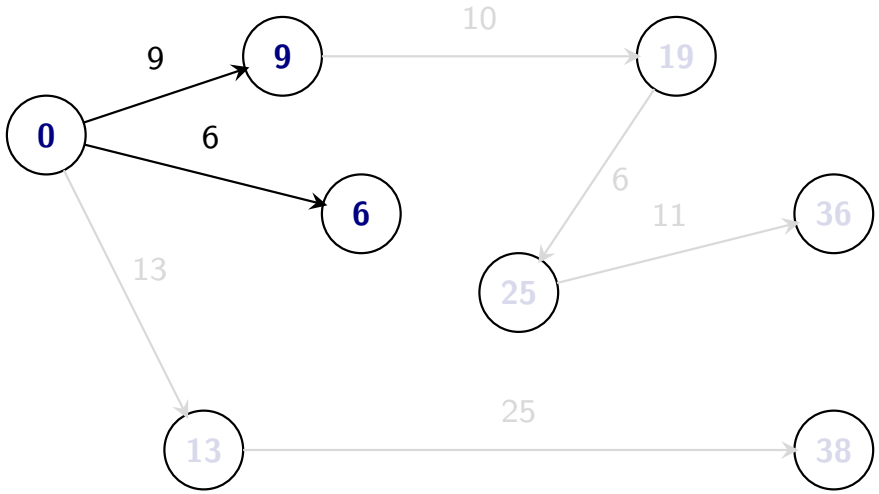

s
2
3
6
4
5
7
t
9
10
6
18
6
6
30
11
13
20
8
19
6
16
25

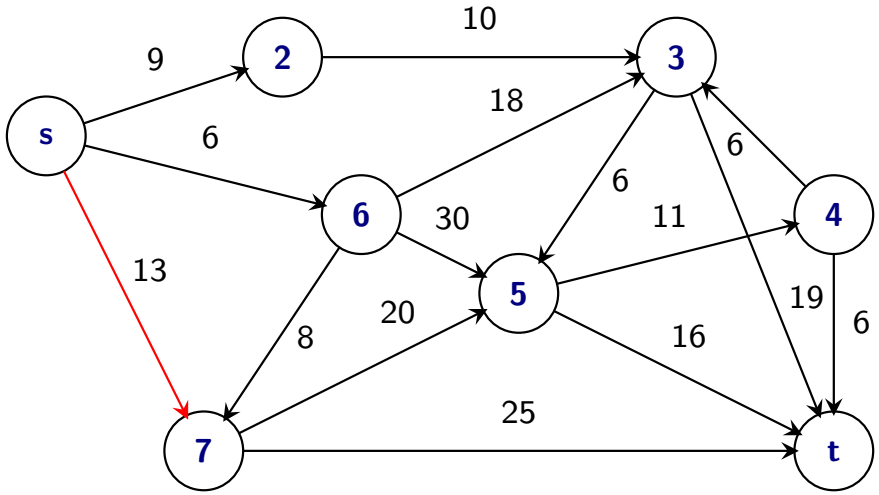

0
9
19
6
36
25
13
38
9
10
6
6
11
13
25

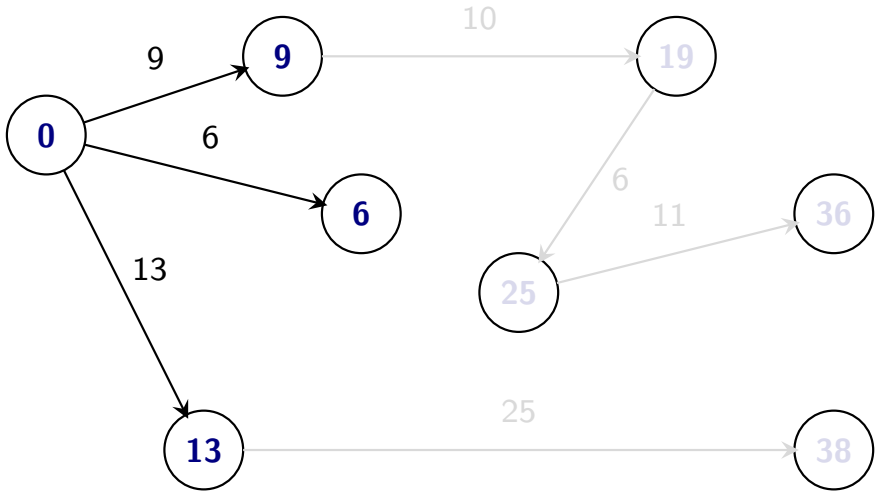

s
2
3
6
4
5
7
t
9
10
6
18
6
6
30
11
13
20
8
19
6
16
25

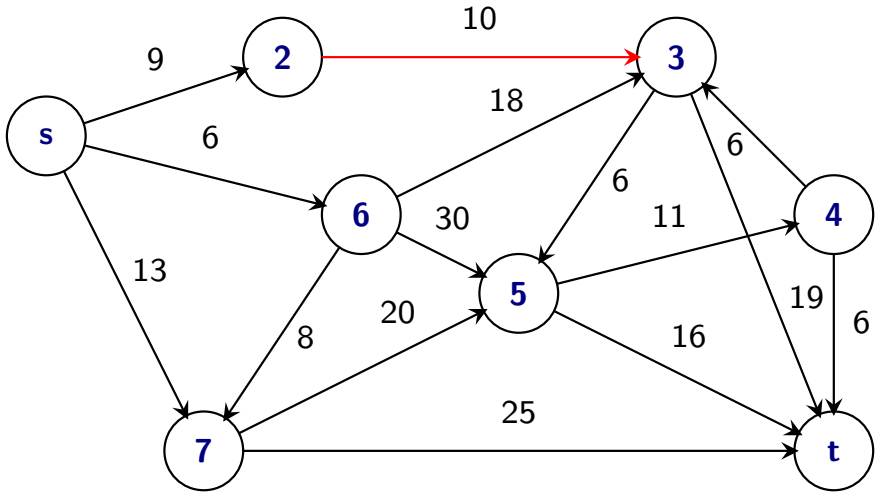

0
9
19
6
36
25
13
38
9
10
6
6
11
13
25

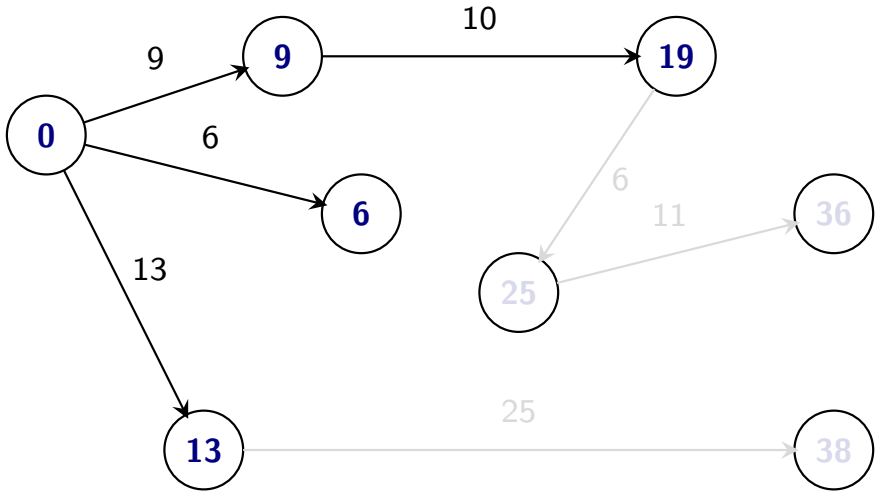

s
2
3
6
4
5
7
t
9
10
18
6
6
6
30
11
13
20
8
19
6
16
25

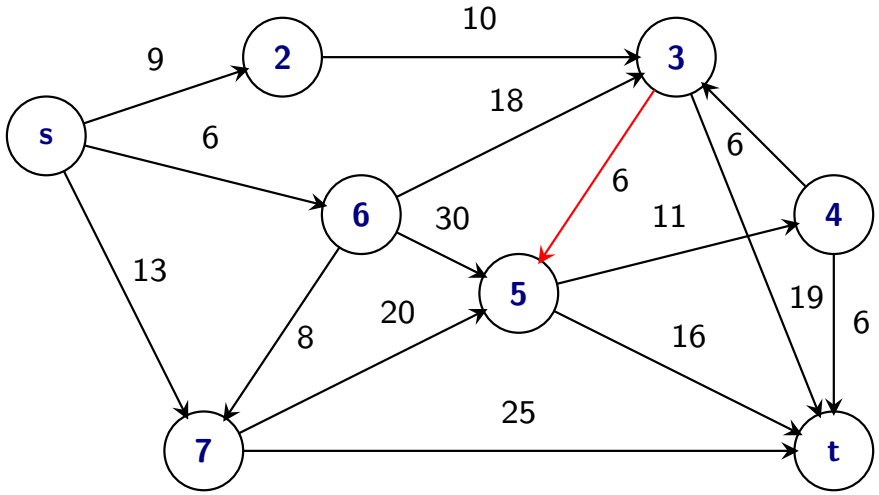

0
9
19
6
36
25
13
38
9
10
6
6
11
13
25

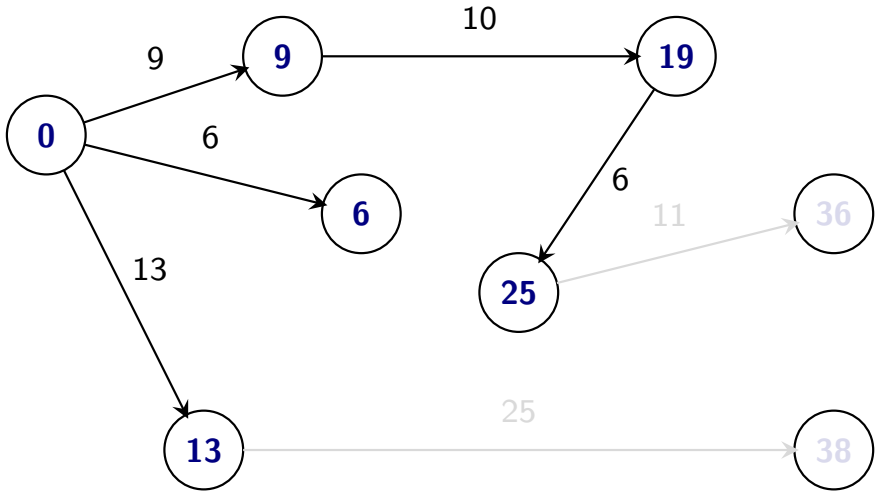

s
2
3
4
5
6
7
t
9
10
18
6
6
6
30
11
13
20
8
19
6
16
25

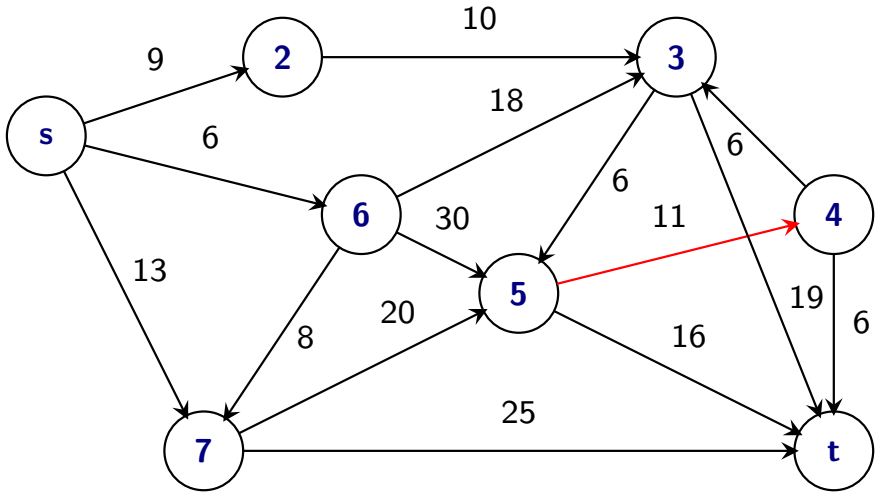

0
9
19
6
36
25
13
38
9
10
6
6
11
25

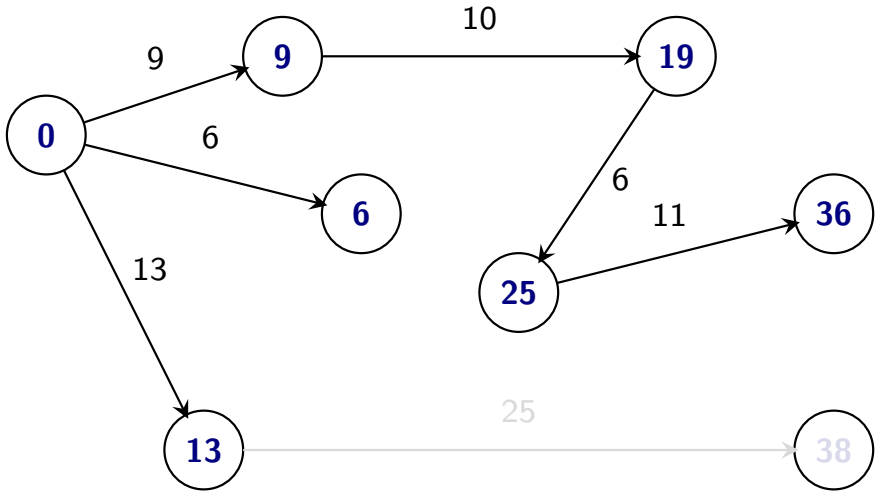

s
2
3
6
4
5
7
t
9
10
18
6
6
6
30
11
13
20
8
19
6
16
25

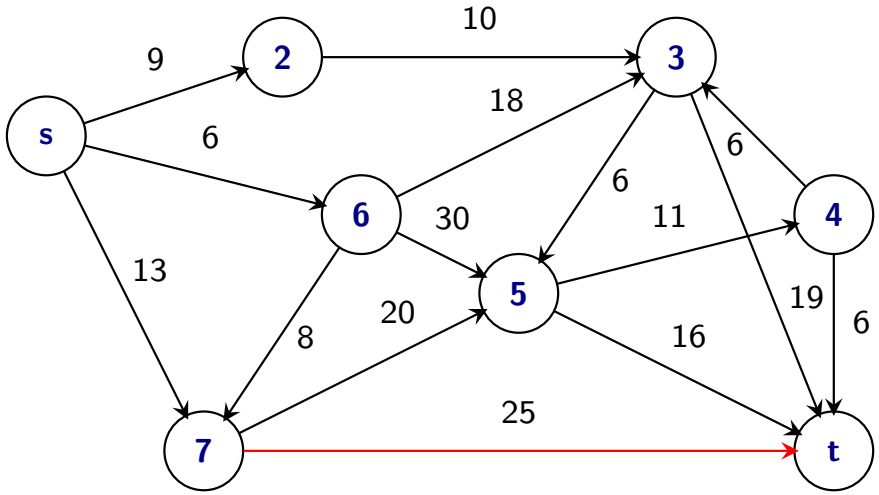

0
9
19
6
6
36
25
13
38
9
10
6
6
11
13
25

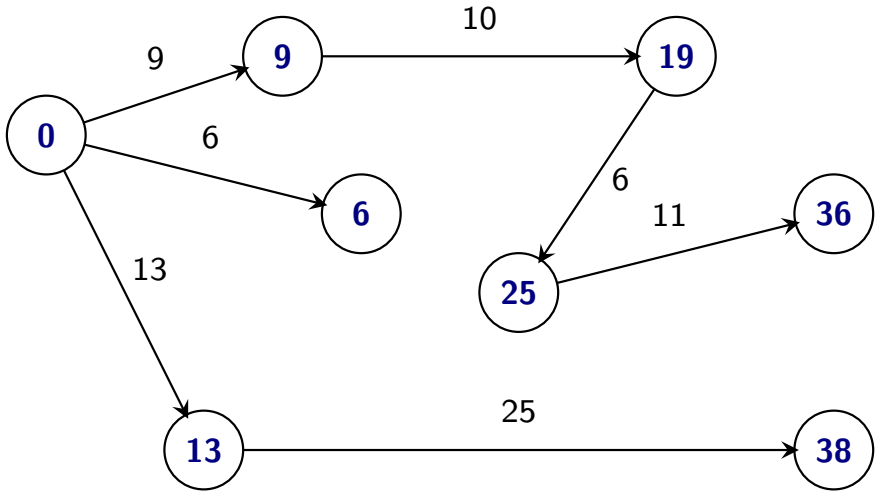

s
2
3
6
4
5
7
t
9
10
18
6
6
6
30
11
13
20
8
19
6
16
25


0
9
19
6
36
25
13
38
9
10
6
6
11
25


s
2
3
6
4
5
7
t
9
10
18
6
6
6
30
11
13
20
8
19
6
16
25


0
9
19
6
36
25
13
38
9
10
6
6
11
13
25


s
2
3
6
4
5
7
t
9
10
18
6
6
6
30
11
13
20
8
19
6
16
25


0
9
19
6
36
25
13
38
9
10
6
6
11
13
25

- Main work is to compute the $d'(s,u)$ values in each iteration
- $d'(s,u)$ changes from iteration i to $i+1$ only because of the node v that is added to S in iteration i.

```
Initialize for each node v, dist(s,v) = d'(s,v) = ∞
Initialize S = ∅, d'(s,s) = 0
for i = 1 to |V| do
    // S contains the i − 1 closest nodes to s,
    //        and the values of d'(s,u) are current
    Let v be such that d'(s,v) = min_{u∈V−S} d'(s,u)
    dist(s,v) = d'(s,v)
    S = S ∪ {v}
    Update d'(s,u) for each u in V-S as follows:
        d'(s,u) = min(d'(s,u), dist(s,v) + ℓ(v,u))
```

Running time: $O(m+n^2)$ time.

- n outer iterations and in each iteration following steps
- updating $d'(s,u)$ after v added takes $O(\deg(v))$ time so total

- Main work is to compute the $d'(s,u)$ values in each iteration
- $d'(s,u)$ changes from iteration i to $i+1$ only because of the node v that is added to S in iteration i.

Initialize for each node v, $dist(s,v) = d'(s,v) = \infty$
Initialize $S = \emptyset$, d'(s,s) = 0
for $i = 1$ to $|V|$ **do**
    // S contains the $i-1$ closest nodes to s,
    // and the values of $d'(s,u)$ are current
    Let v be such that d'(s,v) = $\min_{u \in V-S}$ d'(s,u)
    $dist(s,v) = d'(s,v)$
    $S = S \cup \{v\}$
    Update d'(s,u) for each u in V-S as follows:
        $d'(s,u) = \min(d'(s,u), dist(s,v) + \ell(v,u))$

Running time: $O(m + n^2)$ time.

- n outer iterations and in each iteration following steps
- updating $d'(s,u)$ after v added takes $O(deg(v))$ time so total

Initialize for each node v, $\mathbf{dist(s,v) = d'(s,v) = \infty}$
Initialize S = $\emptyset$, d'(s,s) = 0
for $i = 1$ to $|V|$ **do**
    // S contains the $i-1$ closest nodes to s,
    // and the values of $d'(s,u)$ are current
    Let v be such that d'(s,v) = $\min_{u \in V-S}$ d'(s,u)
    $\mathbf{dist(s,v) = d'(s,v)}$
    $S = S \cup \{v\}$
    Update d'(s,u) for each u in V-S as follows:
        $d'(s,u) = \min(d'(s,u), \mathbf{dist(s,v)} + \ell(v,u))$

Running time: $O(m + n^2)$ time.

- n outer iterations and in each iteration following steps
- updating $d'(s,u)$ after v added takes $O(deg(v))$ time so total work is $O(m)$ since a node enters S only once
- Finding v from $d'(s,u)$ values is $O(n)$ time

- eliminate $d'(s,u)$ and let $dist(s,u)$ maintain it
- update **dist** values after adding **v** by scanning edges out of **v**

```
Initialize for each node v, dist(s,v) = ∞
Initialize S = {s}, dist(s,s) = 0
for i = 1 to |V| do
    Let v be such that dist(s,v) = min_{u∈V−S} dist(s,u)
    S = S ∪ {v}
    for each u in Adj(v) do
        dist(s,u) = min(dist(s,u), dist(s,v) + ℓ(v,u))
```

Priority Queues to maintain **dist** values for faster running time

- Using heaps and standard priority queues: $O((m+n)\log n)$
- Using Fibonacci heaps: $O(m + n\log n)$.

- eliminate $d'(s,u)$ and let $dist(s,u)$ maintain it
- update **dist** values after adding **v** by scanning edges out of **v**

```
Initialize for each node v, dist(s,v) = ∞
Initialize S = {s}, dist(s,s) = 0
for i = 1 to |V| do
    Let v be such that dist(s,v) = min_{u∈V−S} dist(s,u)
    S = S ∪ {v}
    for each u in Adj(v) do
        dist(s,u) = min(dist(s,u), dist(s,v) + ℓ(v,u))
```

Priority Queues to maintain **dist** values for faster running time

- Using heaps and standard priority queues: $O((m+n)\log n)$
- Using Fibonacci heaps: $O(m + n\log n)$.

Data structure to store a set **S** of **n** elements where each element **v** $\in$ **S** has an associated real/integer key **k(v)** such that the following operations

- `makeQ`: create an empty queue
- `findMin`: find the minimum key in **S**
- `extractMin`: Remove **v** $\in$ **S** with smallest key and return it
- `add(v, k(v))`: Add new element **v** with key **k(v)** to **S**
- `delete(v)`: Remove element **v** from **S**
- `decreaseKey(v, k'(v))`: *decrease* key of **v** from **k(v)** (current key) to **k′(v)** (new key). Assumption: **k′(v)** $\leq$ **k(v)**
- `meld`: merge two separate priority queues into one

can be performed in **O(log n)** time each.

`decreaseKey` via `delete` and `add`

Data structure to store a set S of n elements where each element $v \in S$ has an associated real/integer key $k(v)$ such that the following operations

- `makeQ`: create an empty queue
- `findMin`: find the minimum key in S
- `extractMin`: Remove $v \in S$ with smallest key and return it
- `add(v, k(v))`: Add new element v with key $k(v)$ to S
- `delete(v)`: Remove element v from S
- `decreaseKey(v, k'(v))`: *decrease* key of v from $k(v)$ (current key) to $k'(v)$ (new key). Assumption: $k'(v) \leq k(v)$
- `meld`: merge two separate priority queues into one

can be performed in $O(\log n)$ time each.

`decreaseKey` via `delete` and `add`

Data structure to store a set S of n elements where each element $v \in S$ has an associated real/integer key $k(v)$ such that the following operations

- `makeQ`: create an empty queue
- `findMin`: find the minimum key in S
- `extractMin`: Remove $v \in S$ with smallest key and return it
- `add(v, k(v))`: Add new element v with key $k(v)$ to S
- `delete(v)`: Remove element v from S
- `decreaseKey(v, k'(v))`: *decrease* key of v from $k(v)$ (current key) to $k'(v)$ (new key). Assumption: $k'(v) \leq k(v)$
- `meld`: merge two separate priority queues into one

can be performed in $O(\log n)$ time each.
`decreaseKey` via `delete` and `add`

```
Q = makePQ()
insert(Q, (s, 0))
for each node u ≠ s do
    insert(Q, (u, ∞))
S = ∅
for i = 1 to |V| do
    (v, dist(s, v)) = extractMin(Q)
    S = S ∪ {v}
    For each u in Adj(v) do
        decreaseKey(Q, (u, min(dist(s, u), dist(s, v) + ℓ(v, u))))
```

Priority Queue operations:

- **O(n)** `insert` operations
- **O(n)** `extractMin` operations
- **O(m)** `decreaseKey` operations

Using Heaps

Store elements in a heap based on the key value

- All operations can be done in $\mathbf{O(\log n)}$ time

Dijkstra's algorithm can be implemented in $\mathbf{O((n+m)\log n)}$ time.

Using Heaps

Store elements in a heap based on the key value

- All operations can be done in $\mathbf{O(\log n)}$ time

Dijkstra's algorithm can be implemented in $\mathbf{O((n + m) \log n)}$ time.

Fibonacci Heaps

- `extractMin, add, delete, meld` in $\mathbf{O(\log n)}$ time
- `decreaseKey` in $\mathbf{O(1)}$ *amortized* time: ℓ decreaseKey operations for $\ell \geq \mathbf{n}$ take *together* $\mathbf{O}(\ell)$ time
- Relaxed Heaps: `decreaseKey` in $\mathbf{O(1)}$ worst case time but at the expense of `meld` (not necessary for Dijkstra's algorithm)

— Dijkstra's algorithm can be implemented in $\mathbf{O(n \log n + m)}$ time. If $\mathbf{m = \Omega(n \log n)}$, running time is linear in input size.

— Data structures are complicated to analyze/implement. Recent work has obtained data structures that are easier to analyze and implement, and perform well in practice. Rank-Pairing Heaps (European Symposium on Algorithms, September 2009!)

Fibonacci Heaps

- `extractMin, add, delete, meld` in $\mathbf{O(\log n)}$ time
- `decreaseKey` in $\mathbf{O(1)}$ *amortized* time: ℓ decreaseKey operations for $\ell \geq \mathbf{n}$ take *together* $\mathbf{O(\ell)}$ time
- Relaxed Heaps: `decreaseKey` in $\mathbf{O(1)}$ worst case time but at the expense of `meld` (not necessary for Dijkstra's algorithm)

— Dijkstra's algorithm can be implemented in $\mathbf{O(n \log n + m)}$ time. If $\mathbf{m = \Omega(n \log n)}$, running time is linear in input size.

— Data structures are complicated to analyze/implement. Recent work has obtained data structures that are easier to analyze and implement, and perform well in practice. Rank-Pairing Heaps (European Symposium on Algorithms, September 2009!)

Fibonacci Heaps

- `extractMin`, `add`, `delete`, `meld` in $\mathbf{O(\log n)}$ time
- `decreaseKey` in $\mathbf{O(1)}$ *amortized* time: ℓ decreaseKey operations for $\ell \geq \mathbf{n}$ take *together* $\mathbf{O(\ell)}$ time
- Relaxed Heaps: `decreaseKey` in $\mathbf{O(1)}$ worst case time but at the expense of `meld` (not necessary for Dijkstra's algorithm)

— Dijkstra's algorithm can be implemented in $\mathbf{O(n \log n + m)}$ time. If $\mathbf{m = \Omega(n \log n)}$, running time is linear in input size.

— Data structures are complicated to analyze/implement. Recent work has obtained data structures that are easier to analyze and implement, and perform well in practice. Rank-Pairing Heaps (European Symposium on Algorithms, September 2009!)

Fibonacci Heaps

- `extractMin, add, delete, meld` in $\mathbf{O(\log n)}$ time
- `decreaseKey` in $\mathbf{O(1)}$ *amortized* time: ℓ decreaseKey operations for $\ell \geq \mathbf{n}$ take *together* $\mathbf{O(\ell)}$ time
- Relaxed Heaps: `decreaseKey` in $\mathbf{O(1)}$ worst case time but at the expense of `meld` (not necessary for Dijkstra's algorithm)

— Dijkstra's algorithm can be implemented in $\mathbf{O(n \log n + m)}$ time. If $\mathbf{m = \Omega(n \log n)}$, running time is linear in input size.

— Data structures are complicated to analyze/implement. Recent work has obtained data structures that are easier to analyze and implement, and perform well in practice. Rank-Pairing Heaps (European Symposium on Algorithms, September 2009!)

Dijkstra's algorithm finds the shortest path distances from s to V.

Question: How do we find the paths themselves?

```
Q = makePQ()
insert(Q, (s, 0))
prev(s) = null
for each node u ≠ s do
    insert(Q, (u, ∞) )
    prev(u) = null

S = ∅
for i = 1 to |V| do
    (v, dist(s, v)) = extractMin(Q)
    S = S ∪ {v}
    for each u in Adj(v) do
        if (dist(s, v) + ℓ(v, u) < dist(s, u) ) then
            decreaseKey(Q, (u, dist(s, v) + ℓ(v, u)) )
            prev(u) = v
```

Dijkstra's algorithm finds the shortest path distances from s to V.

Question: How do we find the paths themselves?

```
Q = makePQ()
insert(Q, (s, 0))
prev(s) = null
for each node u ≠ s do
    insert(Q, (u, ∞) )
    prev(u) = null

S = ∅
for i = 1 to |V| do
    (v, dist(s, v)) = extractMin(Q)
    S = S ∪ {v}
    for each u in Adj(v) do
        if (dist(s, v) + ℓ(v, u) < dist(s, u) ) then
            decreaseKey(Q, (u, dist(s, v) + ℓ(v, u)) )
            prev(u) = v
```

Lemma

The edge set $(\mathbf{u}, \mathbf{prev(u)})$ *is the* reverse *of a shortest path tree rooted at* **s**. *For each* **u**, *the reverse of the path from* **u** *to* **s** *in the tree is a shortest path from* **s** *to* **u**.

Proof Sketch.

- The edgeset $\{(\mathbf{u}, \mathbf{prev(u)}) \mid \mathbf{u} \in \mathbf{V}\}$ induces a directed in-tree rooted at **s** (Why?)
- Use induction on $|\mathbf{S}|$ to argue that the tree is a shortest path tree for nodes in **V**.

Dijkstra's algorithm gives shortest paths from **s** to all nodes in **V**.

How do we find shortest paths from all of **V** to **s**?

- In undirected graphs shortest path from **s** to **u** is a shortest path from **u** to **s** so there is no need to distinguish.
- In directed graphs, use Dijkstra's algorithm in $\mathbf{G^{rev}}$!

Dijkstra's algorithm gives shortest paths from **s** to all nodes in **V**.

How do we find shortest paths from all of **V** to **s**?

- In undirected graphs shortest path from **s** to **u** is a shortest path from **u** to **s** so there is no need to distinguish.
- In directed graphs, use Dijkstra's algorithm in $\mathbf{G^{rev}}$!