BFS and Dijkstra's Algorithm

Lecture 17

Part I

A Brief Review

Whatever-first-search

Given $G = (V, E)$ a directed graph and vertex $u \in V$. Let $n = |V|$.

```
Explore(G, u):
    array Visited[1..n]
    Initialize:  Set Visited[i] = FALSE for 1 ≤ i ≤ n
    List:  ToExplore, S
    Add u to ToExplore and to S, Visited[u] = TRUE
    Make tree T with root as u
    while (ToExplore is non-empty) do
        Remove node x from ToExplore
        for each edge (x, y) in Adj(x) do
            if (Visited[y] == FALSE)
                Visited[y] = TRUE
                Add y to ToExplore
                Add y to S
                Add y to T with edge (x, y)
    Output S
```

Properties of Basic Search

DFS and **BFS** are special case of BasicSearch.

1. Depth First Search (**DFS**): use stack data structure to implement the list ***ToExplore***
2. Breadth First Search (**BFS**): use queue data structure to implementing the list ***ToExplore***

DFS with Visit Times

Keep track of when nodes are visited.

DFS(G)
    for `all` $u \in V(G)$ **do**
        `Mark` u `as unvisited`
    T `is set to` $\emptyset$
    $time = 0$
    while $\exists$`unvisited` u **do**
        DFS(u)
    `Output` T
	DFS(u)
	    `Mark` u `as visited`
	    $\mathrm{pre}(u) = {+}{+}time$
	    for `each` uv `in` $Out(u)$ **do**
	        if v `is not marked` **then**
	            `add edge` uv `to` T
	            DFS(v)
	    $\mathrm{post}(u) = {+}{+}time$

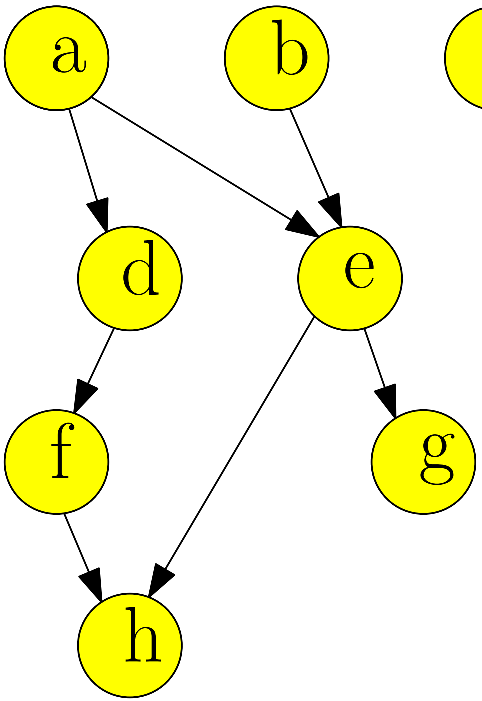
An Edge in DAG

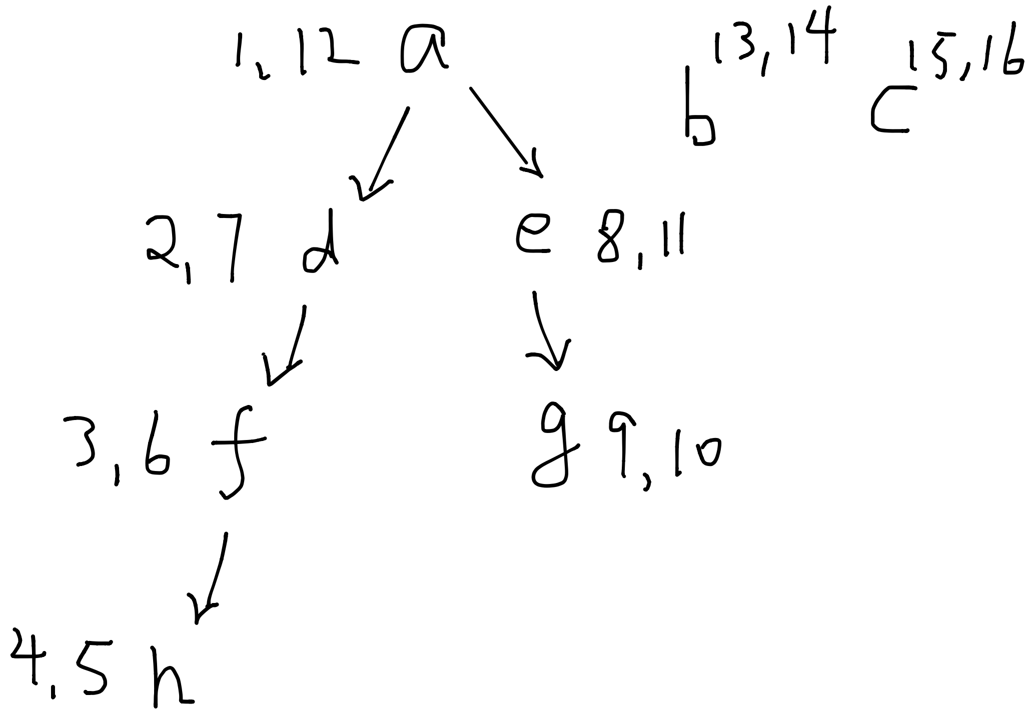
Proposition

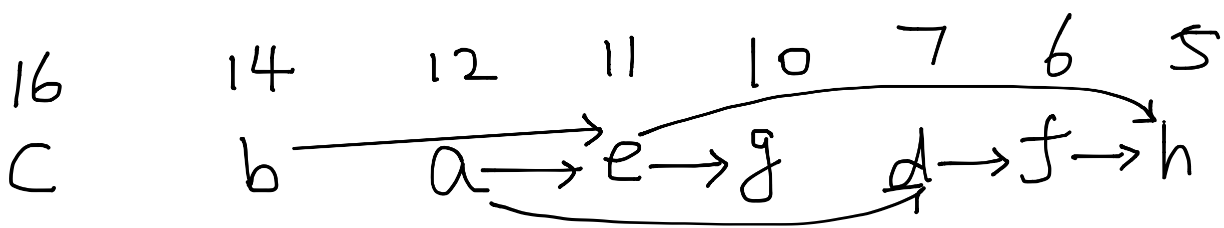
If G is a DAG *and* $\mathbf{post}(\boldsymbol{u}) < \mathbf{post}(\boldsymbol{v})$, *then* $(\boldsymbol{u}, \boldsymbol{v})$ *is not in* G.
i.e., for all edges $(\boldsymbol{u}, \boldsymbol{v})$ *in a* DAG, $\mathbf{post}(\boldsymbol{u}) > \mathbf{post}(\boldsymbol{v})$.

$u < v$

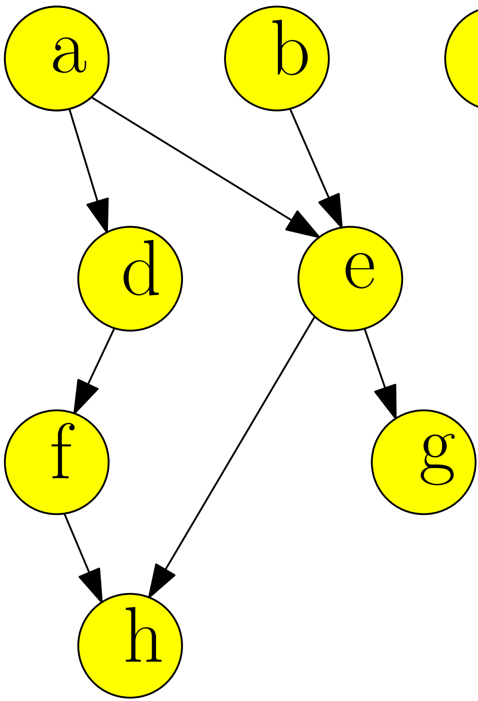
Reverse post-order is topological order

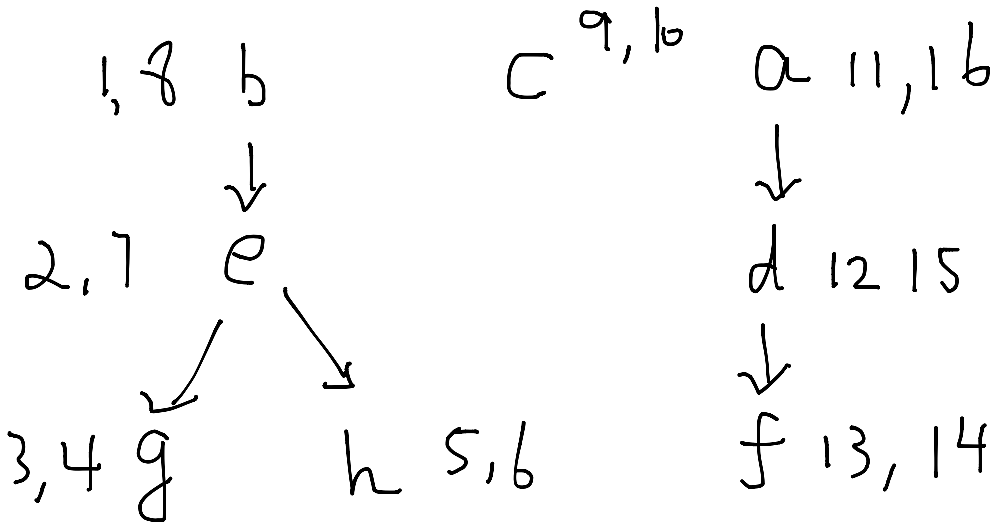


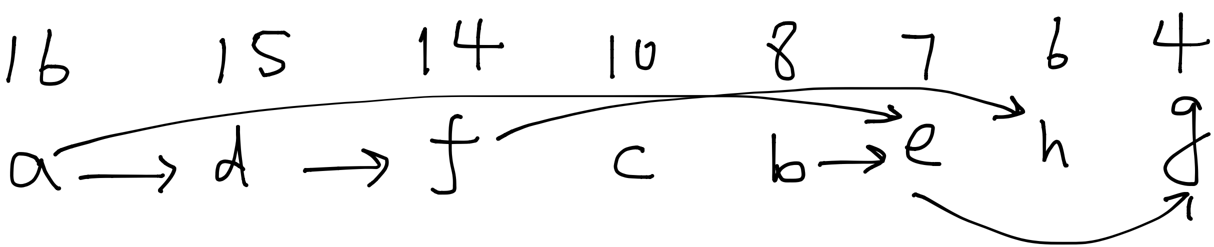




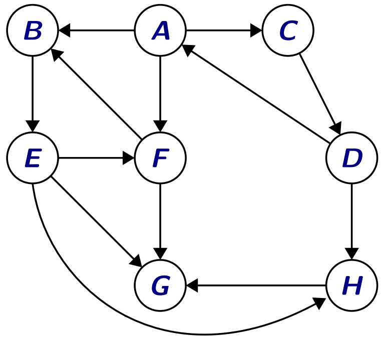
Reverse post-order is topological order



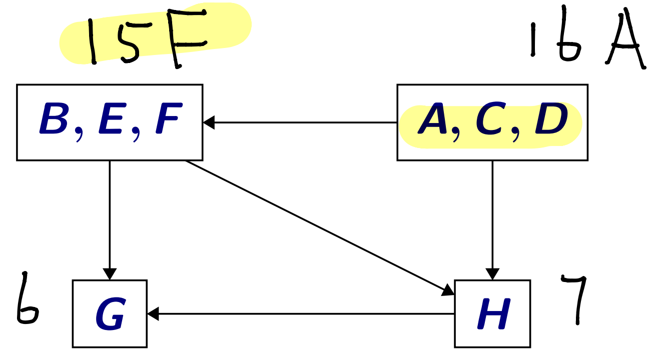




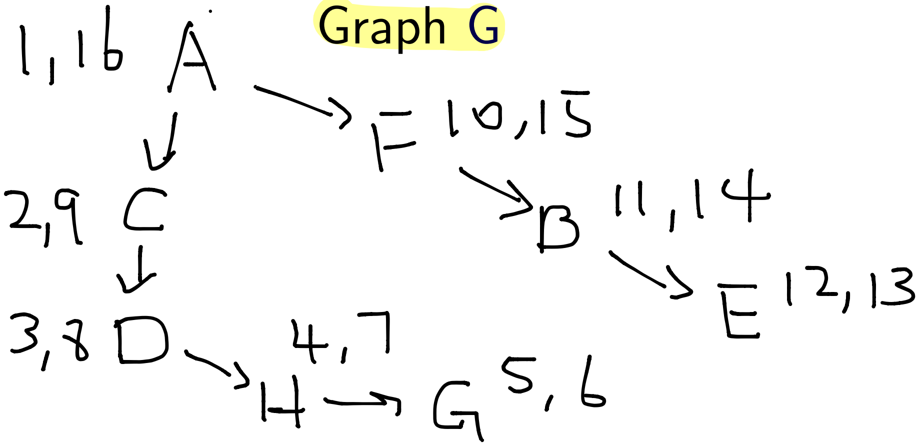
The SCCs are topologically sorted by arranging them in decreasing order of their highest post number.

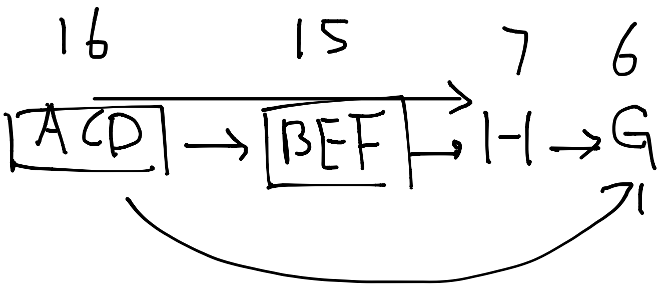


Graph G

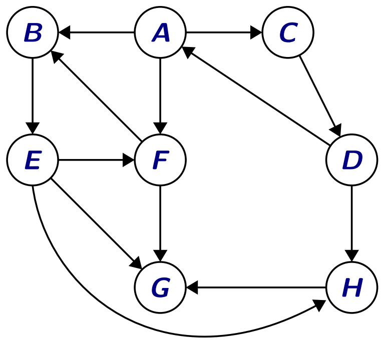


Graph of SCCs G^{SCC}

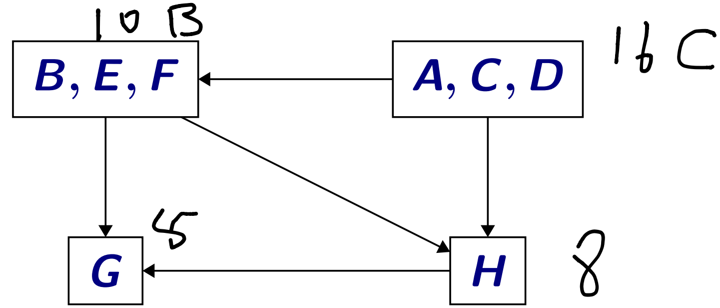




A Different DFS



Graph G



Graph of SCCs G^{SCC}

1,10 B
↓
2,9 E → H 7,8
↓
3,6 F
↓
4,5 G

C 11,16
↓
D 12,15
↓
A 13,14

16 10 8 5

$\boxed{ACD} \to \boxed{BEF} \to H \to G$

Part II

Breadth First Search

Breadth First Search (BFS)

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...

1. **DFS** good for exploring graph structure
2. **BFS** good for exploring *distances*

Queue Data Structure

Queues

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

1. **enqueue**: Adds an element to the end of the list
2. **dequeue**: Removes an element from the front of the list

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

BFS Algorithm

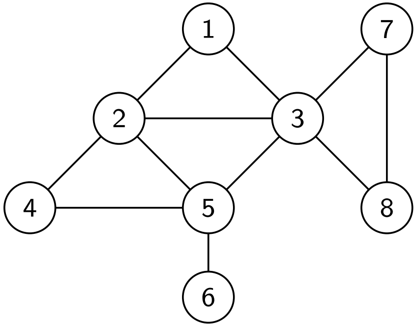
Given (undirected or directed) graph $G = (V, E)$ and node $s \in V$

```
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

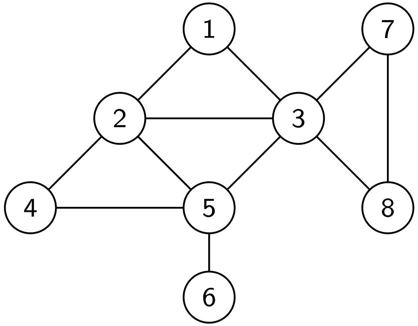
BFS(s) *runs in* $O(n + m)$ *time.*

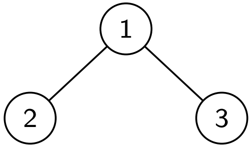
BFS: An Example in Undirected Graphs



1. [1]

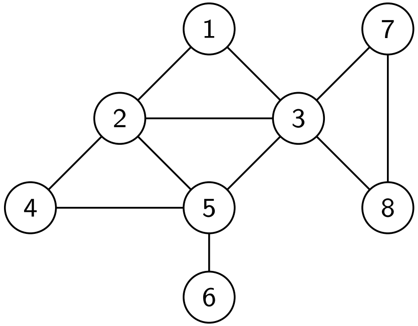
BFS: An Example in Undirected Graphs

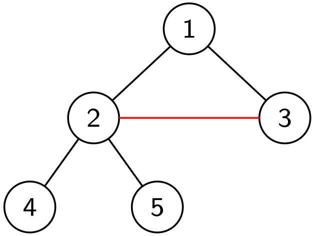




1. [1]
2. [2,3]

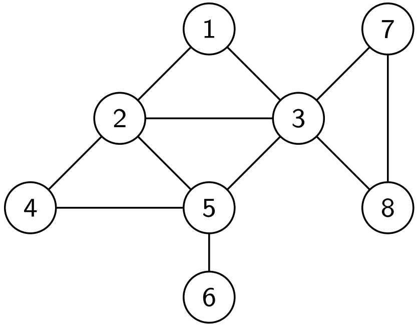
BFS: An Example in Undirected Graphs

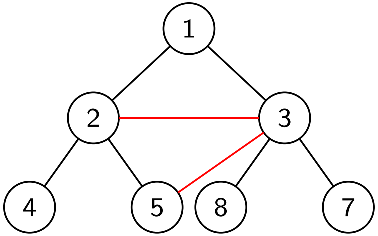




1. [1]
2. [2,3]
3. [3,4,5]

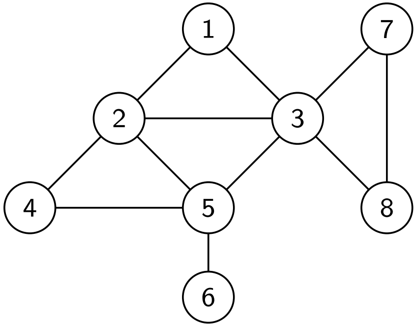
BFS: An Example in Undirected Graphs

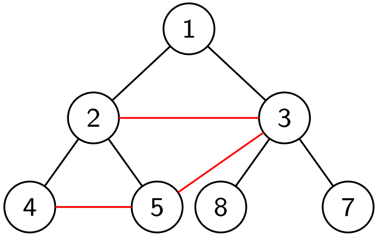




1. [1]
2. [2,3]
3. [3,4,5]
4. [4,5,7,8]

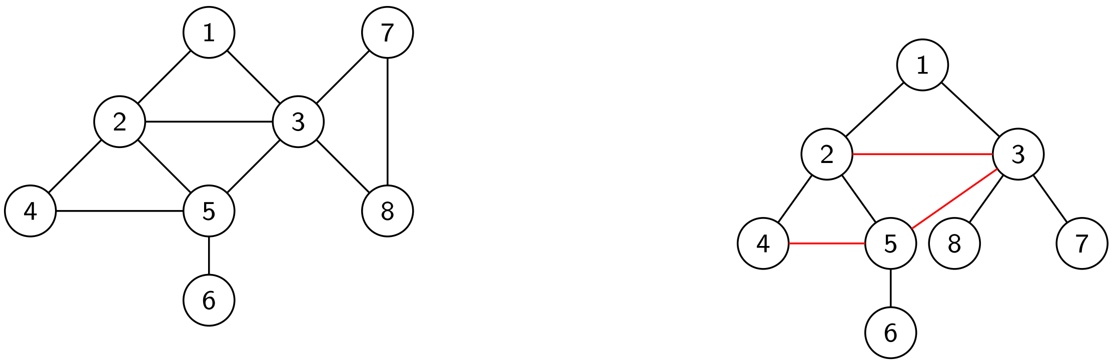
BFS: An Example in Undirected Graphs





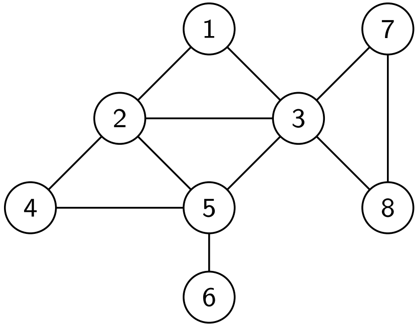
1. [1]
2. [2,3]
3. [3,4,5]
4. [4,5,7,8]
5. [5,7,8]

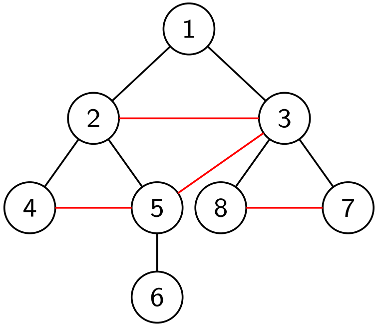
BFS: An Example in Undirected Graphs



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

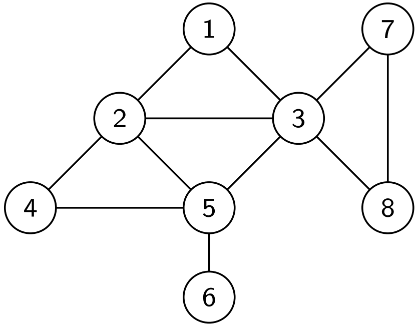
BFS: An Example in Undirected Graphs

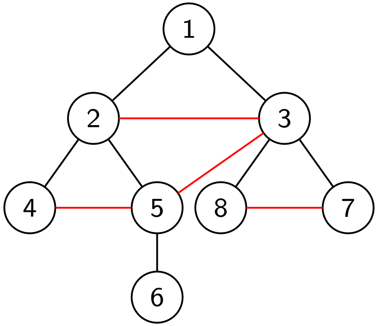




1. [1]
2. [2,3]
3. [3,4,5]
4. [4,5,7,8]
5. [5,7,8]
6. [7,8,6]
7. [8,6]

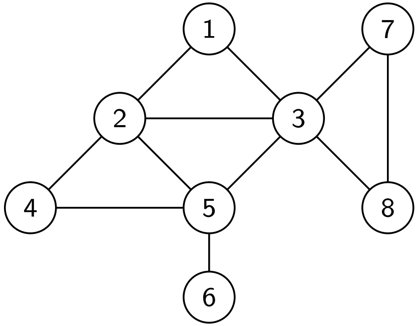
BFS: An Example in Undirected Graphs

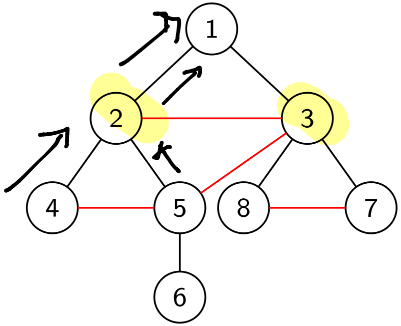




1. [1]
2. [2,3]
3. [3,4,5]
4. [4,5,7,8]
5. [5,7,8]
6. [7,8,6]
7. [8,6]
8. [6]

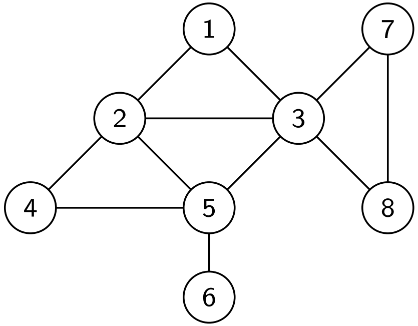
BFS: An Example in Undirected Graphs

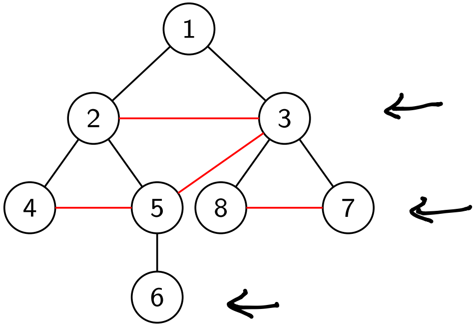




1. [1]
2. [2,3]
3. [3,4,5]
4. [4,5,7,8]
5. [5,7,8]
6. [7,8,6]
7. [8,6]
8. [6]
9. []

BFS: An Example in Undirected Graphs

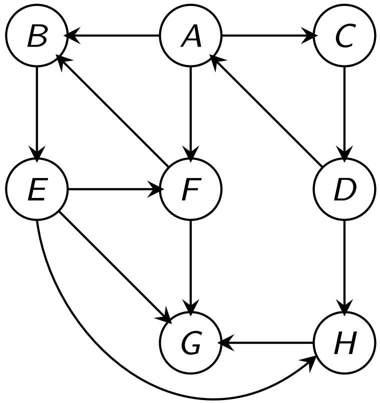


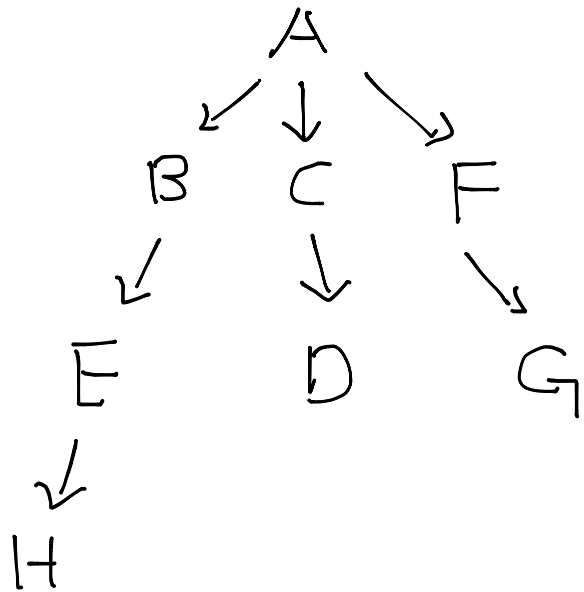


1. [1]
2. [2,3]
3. [3,4,5]
4. [4,5,7,8]
5. [5,7,8]
6. [7,8,6]
7. [8,6]
8. [6]
9. []

BFS tree is the set of black edges.

BFS: An Example in Directed Graphs





BFS with Distance

```
BFS(s)
    Mark all vertices as unvisited; 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
```

Properties of BFS: Undirected Graphs

Theorem

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

1. *The search tree contains exactly the set of vertices in the connected component of* s.
2. *If* $\mathrm{dist}(u) < \mathrm{dist}(v)$ *then* u *is visited before* v.
3. *For every vertex* u, $\mathrm{dist}(u)$ *is the length of a shortest path (in terms of number of edges) from* s *to* u.
4. *If* u, v *are in connected component of* s *and* $e = \{u, v\}$ *is an edge of* G, *then* $|\mathrm{dist}(u) - \mathrm{dist}(v)| \leq 1$.

Properties of BFS: Directed Graphs

Theorem

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

- Ⓐ *The search tree contains exactly the set of vertices reachable from* s
- Ⓑ *If* $\mathrm{dist}(u) < \mathrm{dist}(v)$ *then* u *is visited before* v
- Ⓒ *For every vertex* u, $\mathrm{dist}(u)$ *is the length of shortest path from* s *to* u
- Ⓓ *If* u *is reachable from* s *and* $e = (u, v)$ *is an edge of* G*, then* $\mathrm{dist}(v) - \mathrm{dist}(u) \leq 1$.
  Not necessarily the case that $\mathrm{dist}(u) - \mathrm{dist}(v) \leq 1$.

BFS with Layers

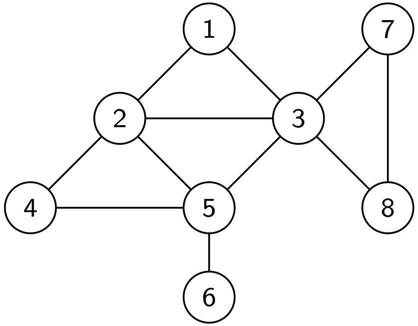
```
BFSLayers(s):
    Mark all vertices as unvisited and initialize T to be empty
    Mark s as visited and set L_0 = {s}
    i = 0
    while L_i is not empty do
            initialize L_{i+1} to be an empty list
            for each u in L_i do
                for each edge (u, v) ∈ Adj(u) do
     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

BFS with Layers

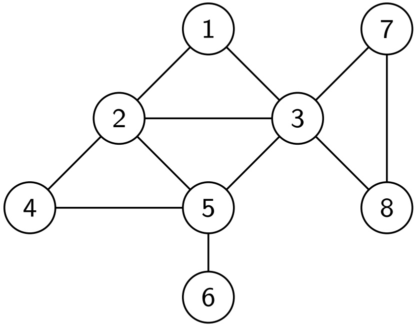
```
BFSLayers(s):
    Mark all vertices as unvisited and initialize T to be empty
    Mark s as visited and set L_0 = {s}
    i = 0
    while L_i is not empty do
            initialize L_{i+1} to be an empty list
            for each u in L_i do
                for each edge (u, v) ∈ Adj(u) do
     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

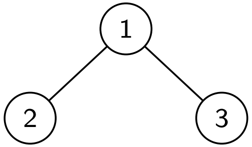
Running time: $O(n + m)$

BFS: An Example in Undirected Graphs

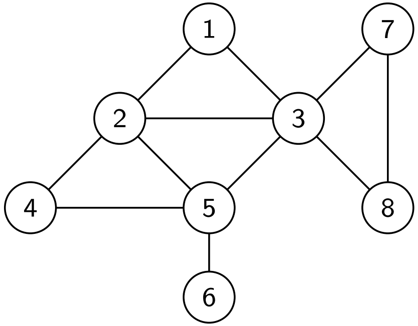


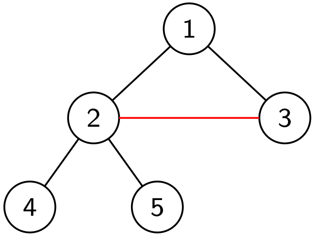
BFS: An Example in Undirected Graphs



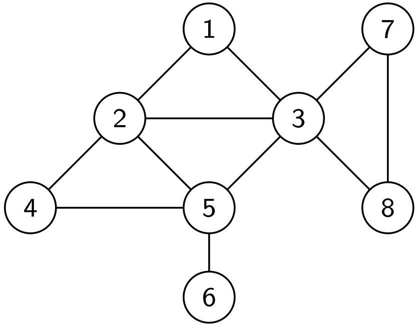


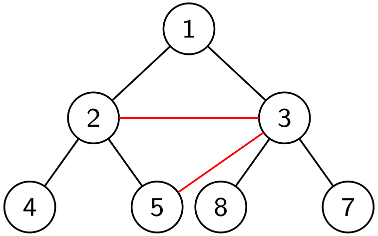
BFS: An Example in Undirected Graphs



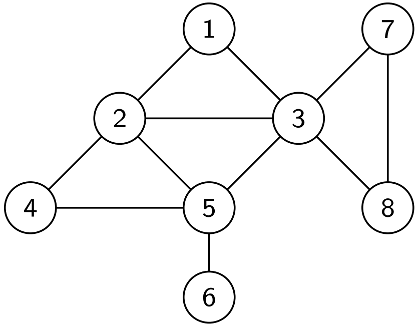


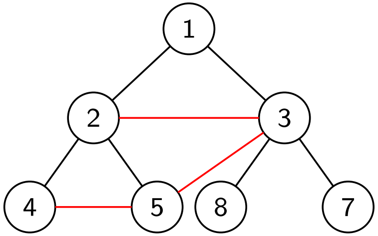
BFS: An Example in Undirected Graphs



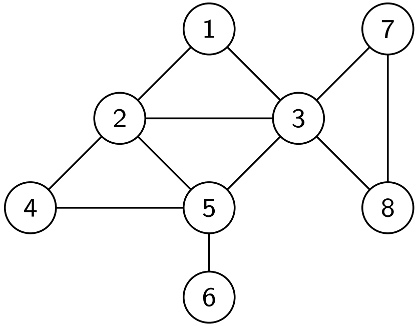


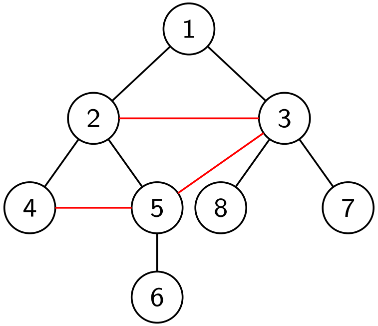
BFS: An Example in Undirected Graphs



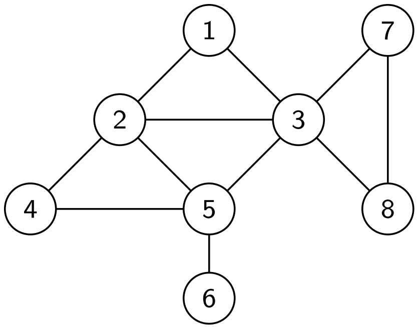


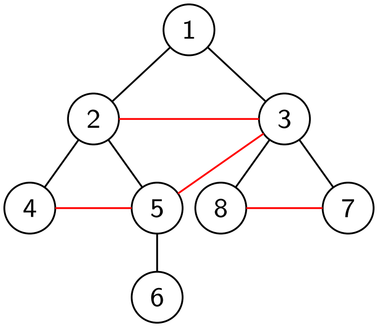
BFS: An Example in Undirected Graphs





BFS: An Example in Undirected Graphs





Part III

Shortest Paths and Dijkstra's Algorithm

Shortest Path Problems

Shortest Path Problems

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

1. Given nodes s, t find shortest path from s to t.
2. Given node s find shortest path from s to all other nodes.
3. Find shortest paths for all pairs of nodes.

Many applications!

Single-Source Shortest Paths:

Non-Negative Edge Lengths

Single-Source Shortest Path Problems

1. Input: A (undirected or directed) graph $G = (V, E)$ with non-negative edge lengths. For edge $e = (u, v)$, $\ell(e) = \ell(u, v)$ is its length.
2. Given nodes s, t find shortest path from s to t.
3. Given node s find shortest path from s to all other nodes.

Single-Source Shortest Paths:

Non-Negative Edge Lengths

Single-Source Shortest Path Problems

1. Input: A (undirected or directed) graph $G = (V, E)$ with non-negative edge lengths. For edge $e = (u, v)$, $\ell(e) = \ell(u, v)$ is its length.
2. Given nodes s, t find shortest path from s to t.
3. Given node s find shortest path from s to all other nodes.

1. Restrict attention to directed graphs
2. Undirected graph problem can be reduced to directed graph problem

Single-Source Shortest Paths via BFS

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

Single-Source Shortest Paths via BFS

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

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

Single-Source Shortest Paths via BFS

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

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

Special case: Suppose $\ell(e)$ is an integer for all e?
Can we use **BFS**?

Single-Source Shortest Paths via BFS

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

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

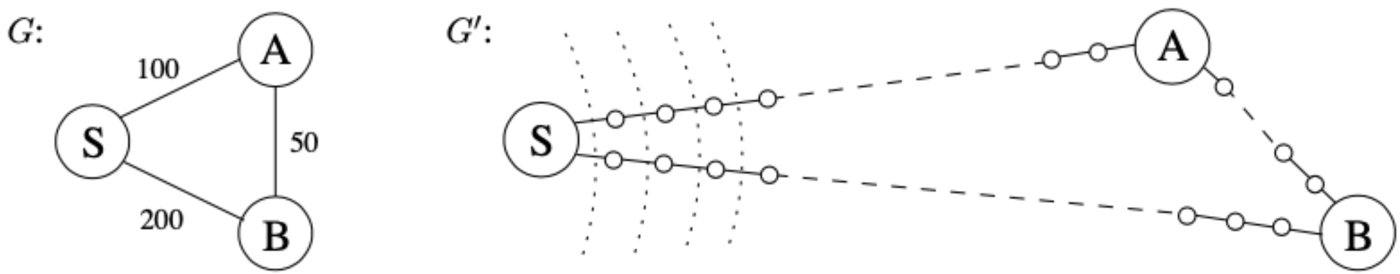
Special case: Suppose $\ell(e)$ is an integer for all e?
Can we use **BFS**? Reduce to unit edge-length problem by placing $\ell(e) - 1$ dummy nodes on e

Single-Source Shortest Paths via BFS

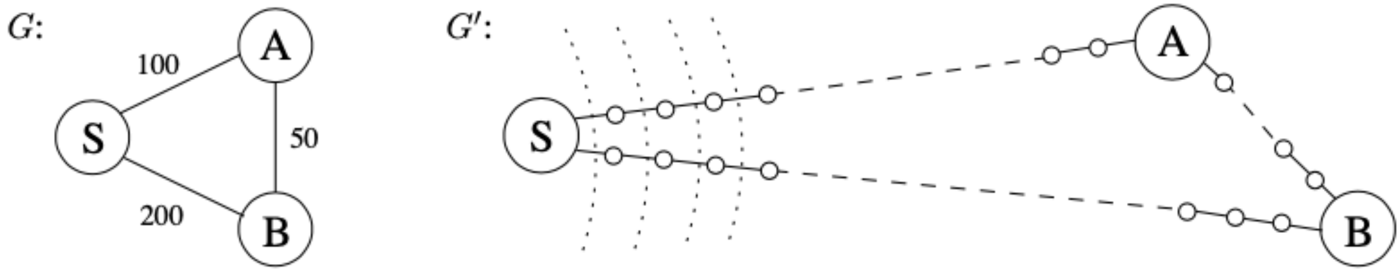
Special case: All edge lengths are $\mathbf{1}$.

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



Single-Source Shortest Paths via BFS



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.

Towards an algorithm

Why does **BFS** work?

Towards an algorithm

Why does **BFS** work?

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

Towards an algorithm

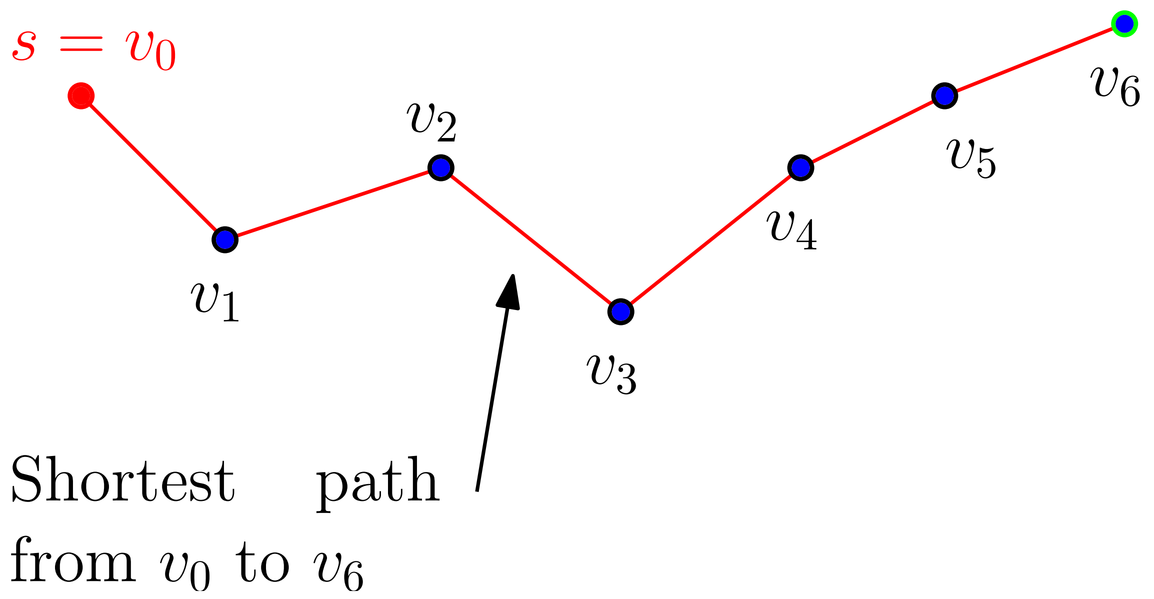
Why does **BFS** work?
BFS(s) explores nodes in increasing (shortest) distance from s

Lemma

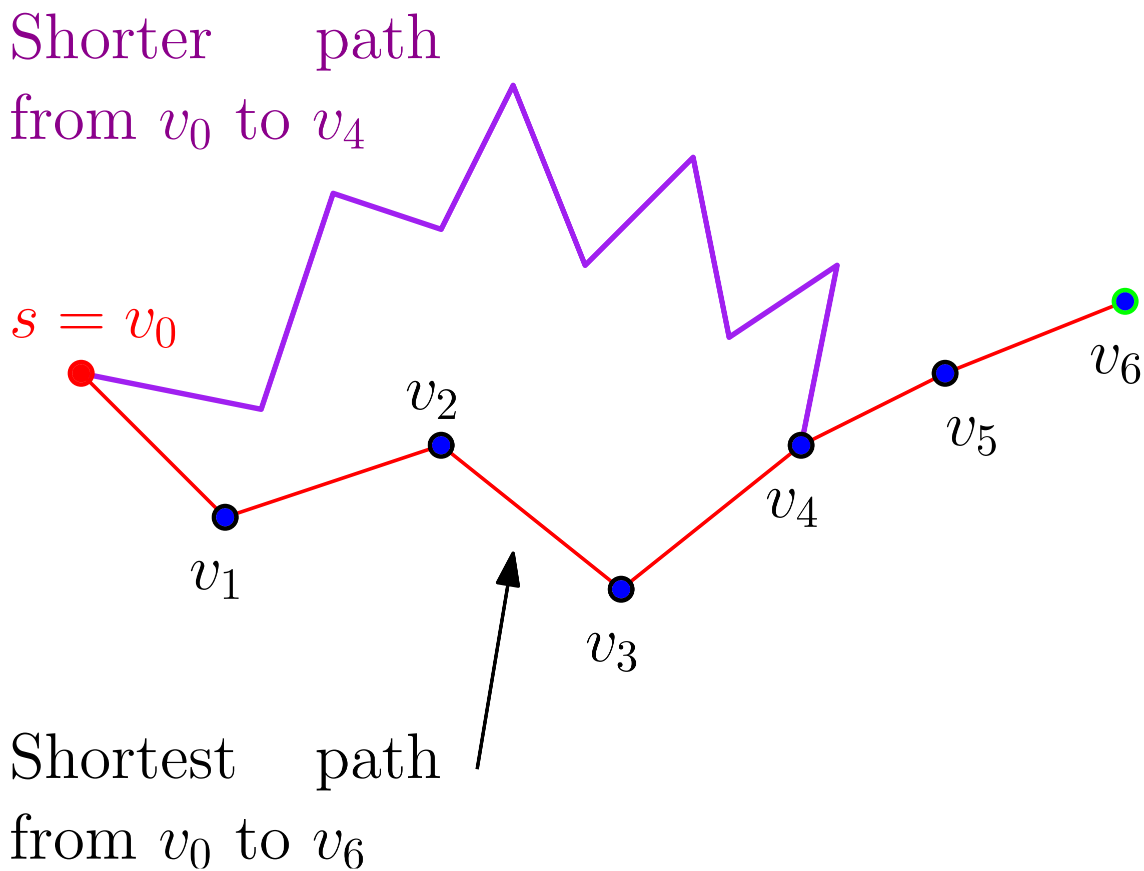
Let G be a directed graph with non-negative edge lengths. Let $\text{dist}(s, v)$ denote the shortest path length from s to v. If $s = v_0 \to v_1 \to v_2 \to \ldots \to v_k$ is a shortest path from s to v_k then for $1 \leq i < k$:

1. $s = v_0 \to v_1 \to v_2 \to \ldots \to v_i$ *is a shortest path from s to v_i*
2. $\text{dist}(s, v_i) \leq \text{dist}(s, v_k)$. *Relies on non-neg edge lengths.*

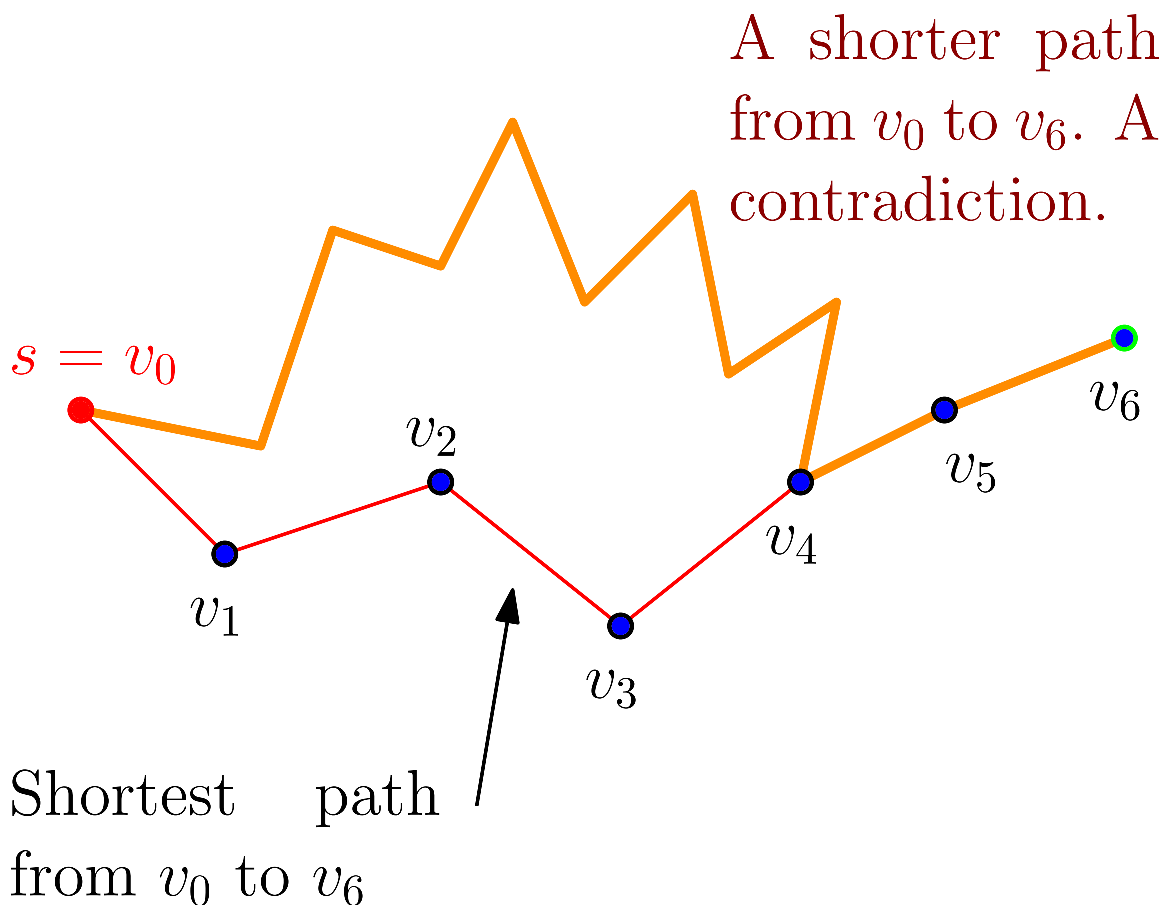
A proof by picture



A proof by picture



A proof by picture



A Basic Strategy

Explore vertices in increasing order of (shortest) 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, dist(s, v) = ∞
Initialize X = {s},
for i = 2 to |V| do
    (* Invariant: X contains the i − 1 closest nodes to s *)
    Among nodes in V − X, find the node v that is the
            i'th closest to s
    Update dist(s, v)
    X = X ∪ {v}
```

A Basic Strategy

Explore vertices in increasing order of (shortest) 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, dist(s, v) = ∞
Initialize X = {s},
for i = 2 to |V| do
    (* Invariant: X contains the i − 1 closest nodes to s *)
    Among nodes in V − X, find the node v that is the
          i'th closest to s
    Update dist(s, v)
    X = X ∪ {v}
```

How can we implement the step in the for loop?

Finding the ith closest node

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

What do we know about the ith closest node?

Finding the ith closest node

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

What do we know about the ith closest node?

Corollary

The ith closest node is adjacent to X.

Finding the ith closest node

Claim

Let P be a shortest path from s to v where v is the ith closest node. Then, all intermediate nodes in P belong to X.

Finding the ith closest node

Claim

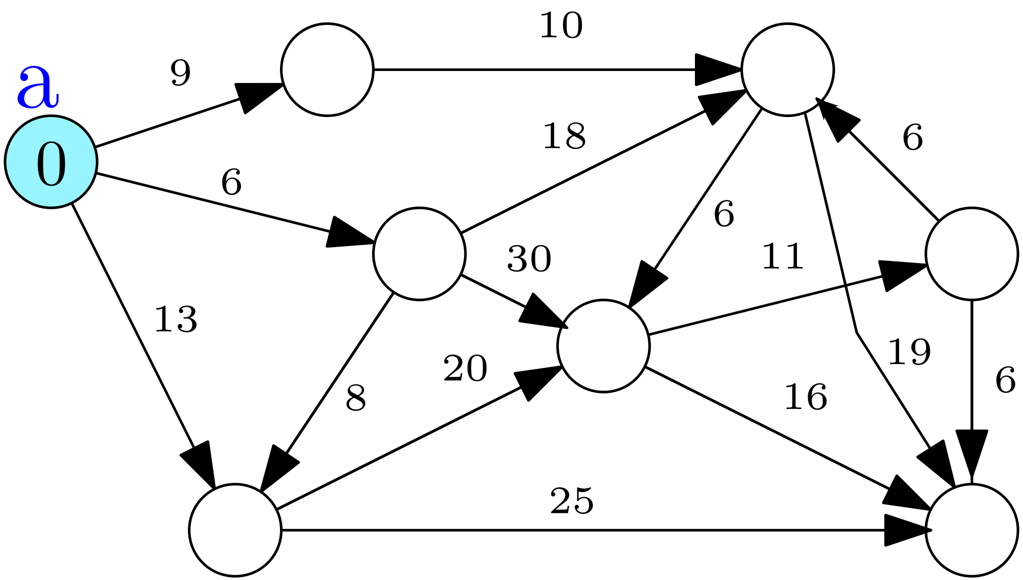
Let P be a shortest path from s to v where v is the ith closest node. Then, all intermediate nodes in P belong to X.

Proof.

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

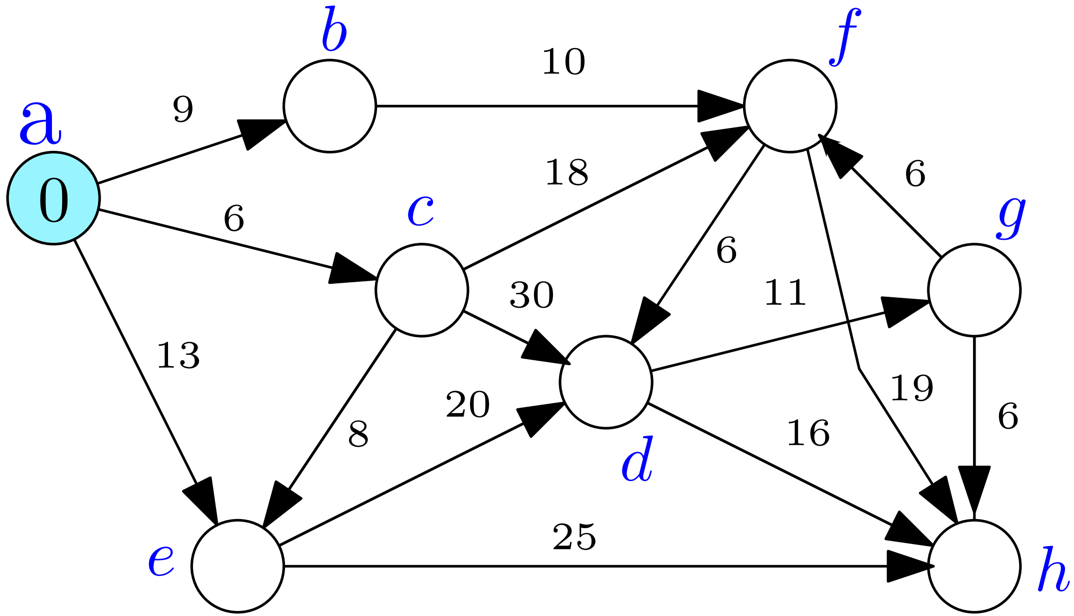
Finding the ith closest node repeatedly

An example



Finding the ith closest node repeatedly

An example



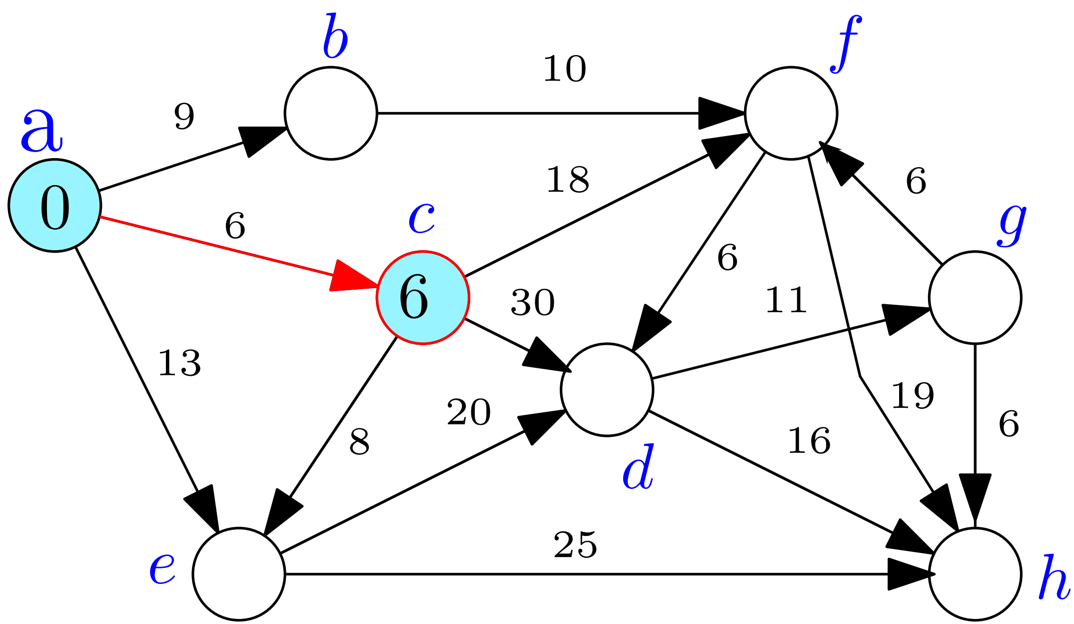
X

{ a }

b c e
9 6 13

Finding the ith closest node repeatedly

An example



a c

b e f d

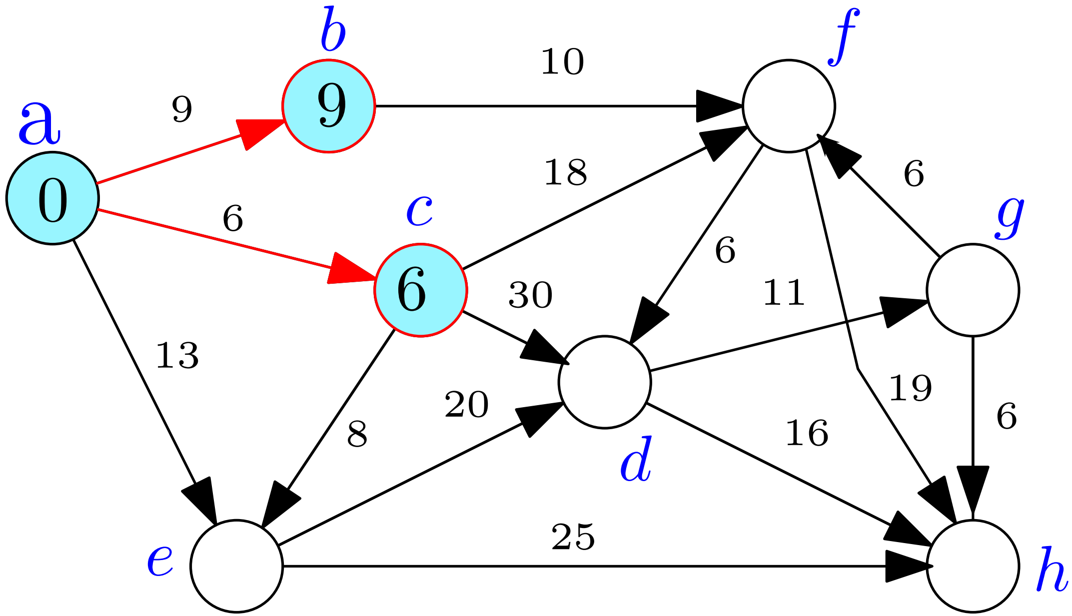
9 13 ? 36

24

a → c → f

Finding the ith closest node repeatedly

An example



a b c

e f d

13 19 36

$x \to f$

$a \to b \to f$

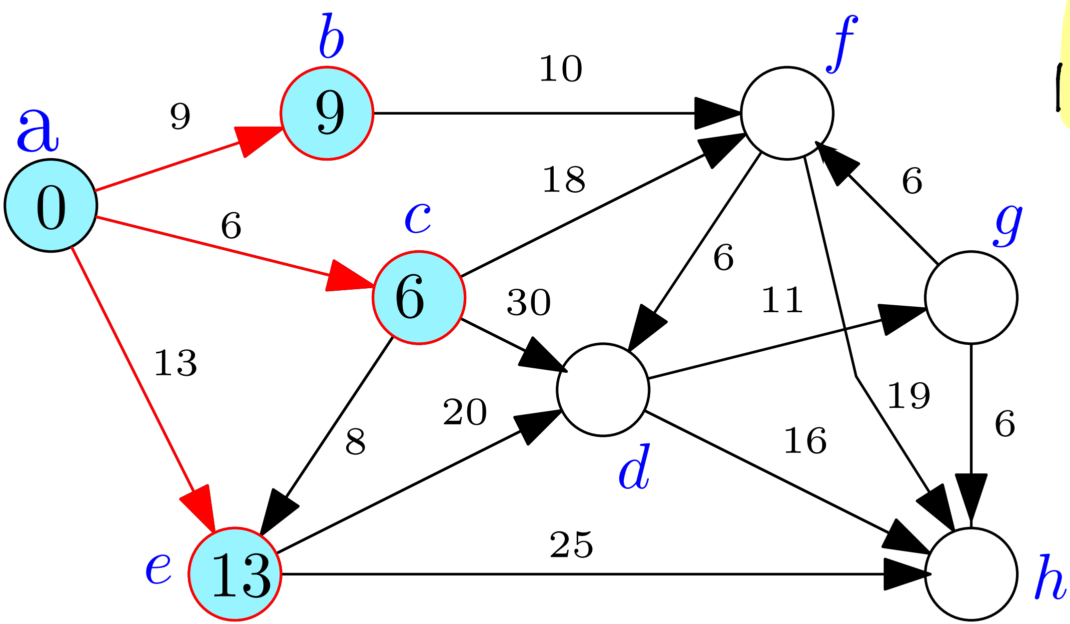
19

$a \to c \to f$

24

Finding the ith closest node repeatedly

An example



f d h
19 33 38

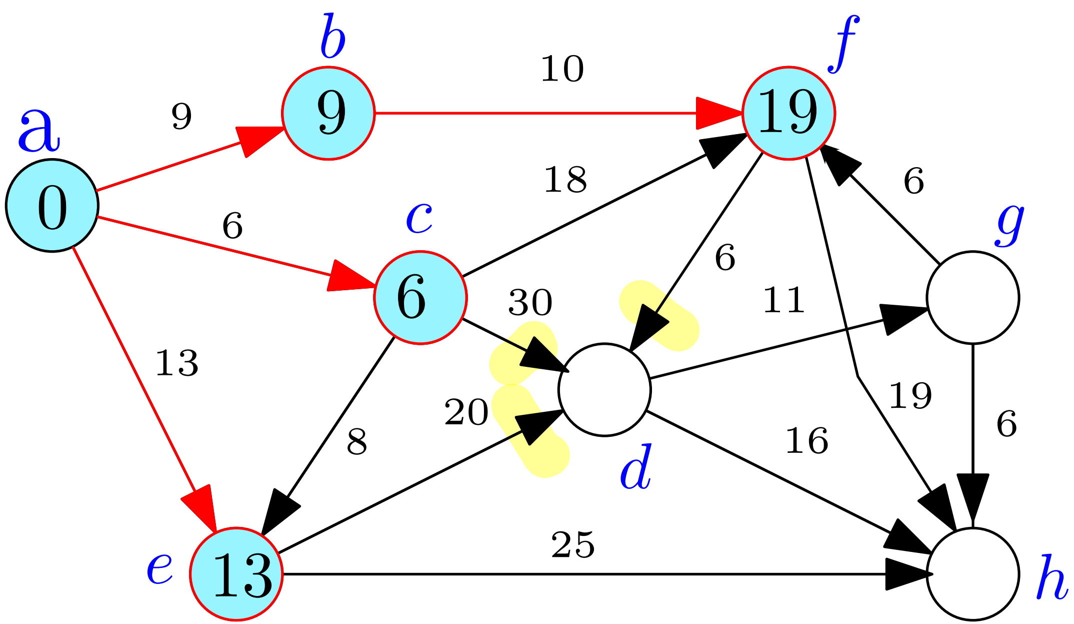
a→b→f
a→c→f

a→c→d
36

a→e→d
33

Finding the ith closest node repeatedly

An example

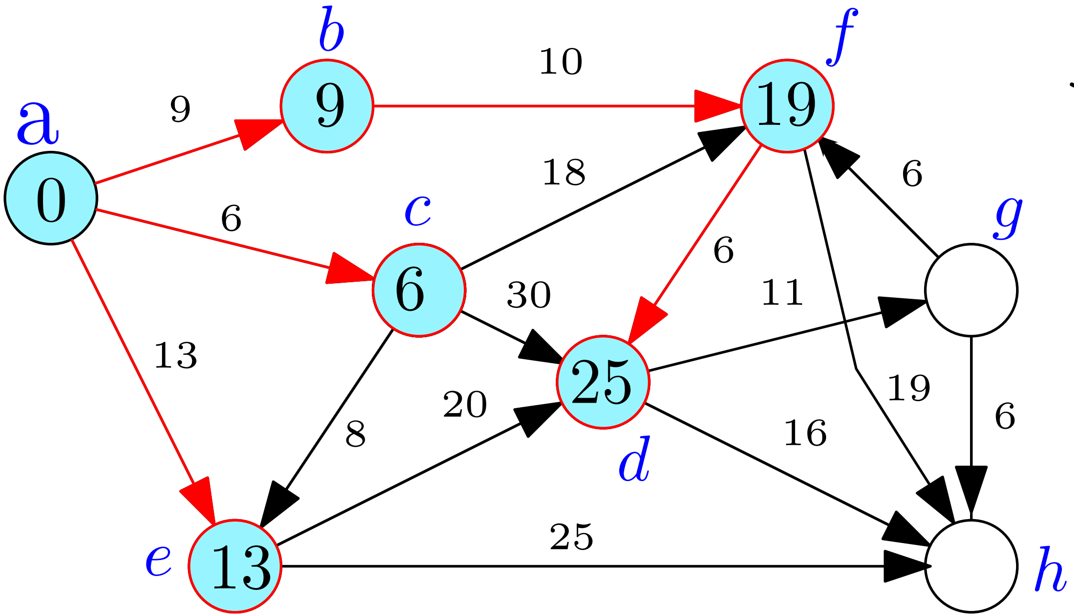


d h
25 38

$f \to d^{25}$
$c \to d^{36}$
$e \to d$ 33
$e \to h^{38}$
$f \to h$ 38

Finding the ith closest node repeatedly

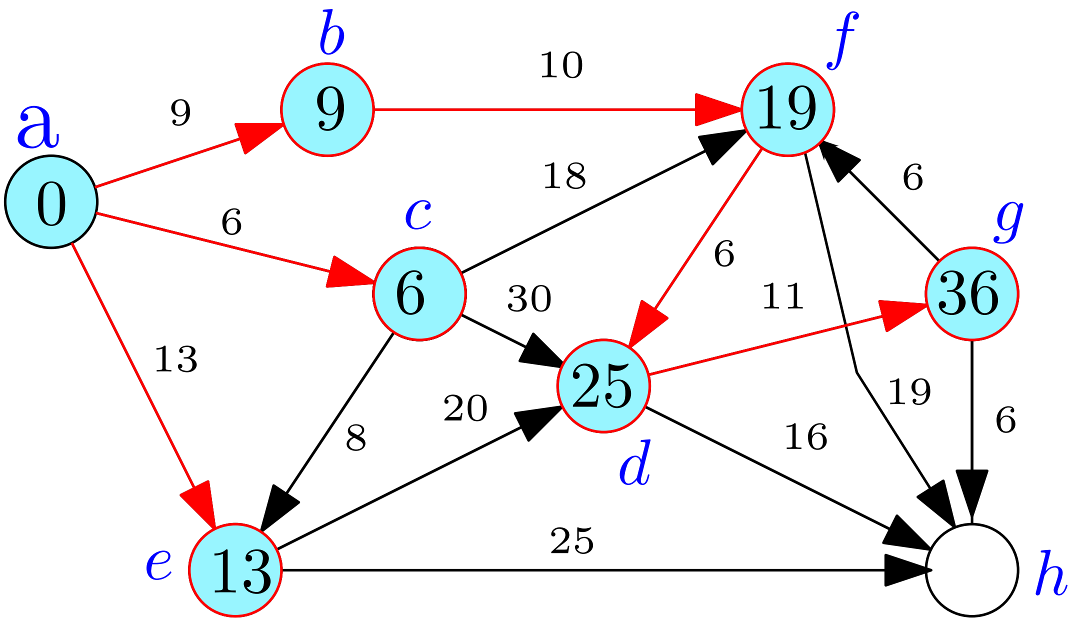
An example



g h
36 38

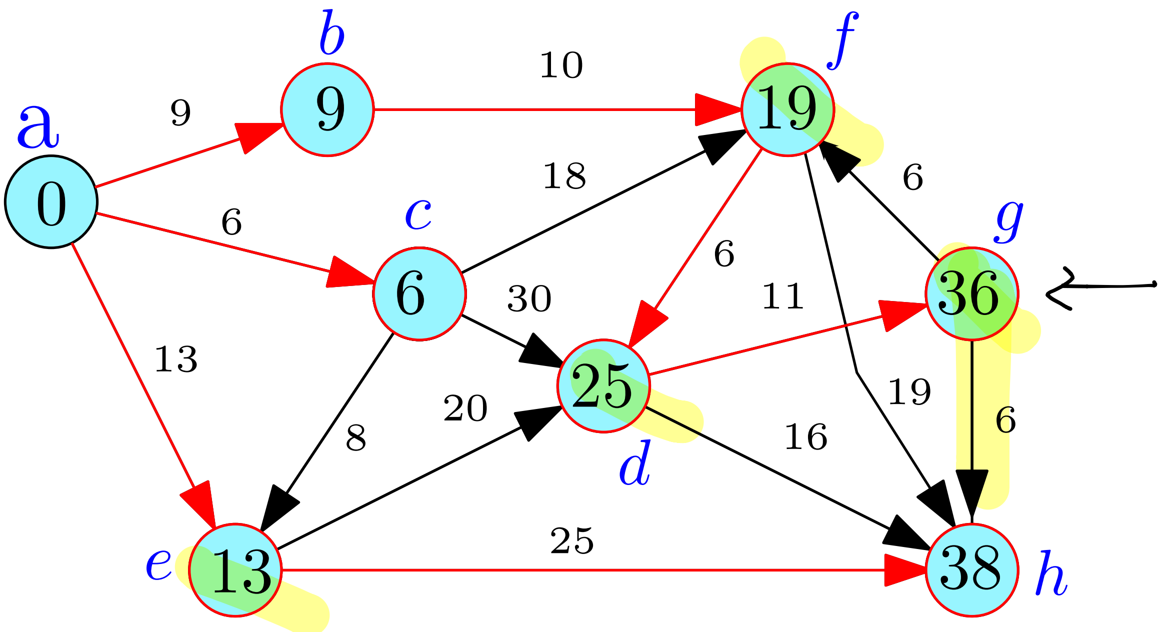
Finding the ith closest node repeatedly

An example



Finding the ith closest node repeatedly

An example



Finding the ith closest node

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

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

Finding the ith closest node

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

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

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

1. $\mathrm{dist}(s,u) \le d'(s,u)$ since we are constraining the paths
2. $d'(s,u) = \min_{t \in X}(\mathrm{dist}(s,t) + \ell(t,u))$

Finding the ith closest node

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

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

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

1. $\mathrm{dist}(s,u) \le d'(s,u)$ since we are constraining the paths
2. $d'(s,u) = \min_{t \in X}(\mathrm{dist}(s,t) + \ell(t,u))$

Lemma

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

Finding the ith closest node

Lemma

Given:

1. X: *Set of* $i-1$ *closest nodes to* s.
2. $d'(s,u) = \min_{t \in X}(\mathrm{dist}(s,t) + \ell(t,u))$

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

Proof.

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

Finding the ith closest node

Lemma

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

Corollary

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

Finding the ith closest node

Lemma

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

Corollary

The ith closest node to s is the node $v \in V - X$ such that $d'(s,v) = \min_{u \in V-X} d'(s,u)$.

Proof.

For every node $u \in V - X$, $\mathrm{dist}(s,u) \leq d'(s,u)$ and for the ith closest node v, $\mathrm{dist}(s,v) = d'(s,v)$. Moreover, $\mathrm{dist}(s,u) \geq \mathrm{dist}(s,v)$ for each $u \in V - X$. $\square$

Algorithm

```
Initialize for each node v: dist(s, v) = ∞
Initialize X = ∅, d'(s, s) = 0
for i = 1 to |V| do
    (* Invariant: X contains the i − 1 closest nodes to s *)
    (* Invariant: d'(s, u) is shortest path distance from u to s
     using only X as intermediate nodes*)
    Let v be such that d'(s, v) = min_{u∈V−X} d'(s, u)
    dist(s, v) = d'(s, v)
    X = X ∪ {v}
    for each node u in V − X do
        d'(s, u) = min_{t∈X} (dist(s, t) + ℓ(t, u))
```

Algorithm

```
Initialize for each node v:  dist(s, v) = ∞
Initialize X = ∅, d'(s, s) = 0
for i = 1 to |V| do
    (* Invariant:  X contains the i − 1 closest nodes to s *)
    (* Invariant:  d'(s, u) is shortest path distance from u to s
     using only X as intermediate nodes*)
    Let v be such that d'(s, v) = min_{u∈V−X} d'(s, u)
    dist(s, v) = d'(s, v)
    X = X ∪ {v}
    for each node u in V − X do
        d'(s, u) = min_{t∈X} (dist(s, t) + ℓ(t, u))
```

Correctness: By induction on i using previous lemmas.

Algorithm

```
Initialize for each node v: dist(s, v) = ∞
Initialize X = ∅, d'(s, s) = 0
for i = 1 to |V| do
    (* Invariant: X contains the i − 1 closest nodes to s *)
    (* Invariant: d'(s, u) is shortest path distance from u to s
     using only X as intermediate nodes*)
    Let v be such that d'(s, v) = min_{u∈V−X} d'(s, u)
    dist(s, v) = d'(s, v)
    X = X ∪ {v}
    for each node u in V − X do
        d'(s, u) = min_{t∈X} (dist(s, t) + ℓ(t, u))
```

Correctness: By induction on i using previous lemmas.

Running time:

Algorithm

```
Initialize for each node v:  dist(s, v) = ∞
Initialize X = ∅, d'(s, s) = 0
for i = 1 to |V| do
    (* Invariant: X contains the i − 1 closest nodes to s *)
    (* Invariant: d'(s, u) is shortest path distance from u to s
     using only X as intermediate nodes*)
    Let v be such that d'(s, v) = min_{u∈V−X} d'(s, u)
    dist(s, v) = d'(s, v)
    X = X ∪ {v}
    for each node u in V − X do
        d'(s, u) = min_{t∈X} (dist(s, t) + ℓ(t, u))
```

Correctness: By induction on i using previous lemmas.

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

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

Improved Algorithm

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

Improved Algorithm

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

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

Running time:

Improved Algorithm

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

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

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

Dijkstra's Algorithm

1. eliminate $d'(s, u)$ and let $\mathrm{dist}(s, u)$ maintain it
2. update *dist* values after adding v by scanning edges out of v

```
Initialize for each node v, dist(s, v) = ∞
Initialize X = ∅, dist(s, s) = 0
for i = 1 to |V| do
    Let v be such that dist(s, v) = min_{u∈V−X} dist(s, u)
    X = X ∪ {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

Dijkstra's Algorithm

1. eliminate $d'(s, u)$ and let $\mathrm{dist}(s, u)$ maintain it
2. update *dist* values after adding v by scanning edges out of v

```
Initialize for each node v, dist(s, v) = ∞
Initialize X = ∅, dist(s, s) = 0
for i = 1 to |V| do
    Let v be such that dist(s, v) = min_{u∈V−X} dist(s, u)
    X = X ∪ {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

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

Priority Queues

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:

1. **makePQ**: create an empty queue.
2. **findMin**: find the minimum key in S.
3. **extractMin**: Remove $v \in S$ with smallest key and return it.
4. **insert**$(v, k(v))$: Add new element v with key $k(v)$ to S.
5. **delete**(v): Remove element v from S.

Priority Queues

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:

1. **makePQ**: create an empty queue.
2. **findMin**: find the minimum key in S.
3. **extractMin**: Remove $v \in S$ with smallest key and return it.
4. **insert**$(v, k(v))$: Add new element v with key $k(v)$ to S.
5. **delete**(v): Remove element v from S.
6. **decreaseKey**$(v, k'(v))$: *decrease* key of v from $k(v)$ (current key) to $k'(v)$ (new key). Assumption: $k'(v) \leq k(v)$.
7. **meld**: merge two separate priority queues into one.

Priority Queues

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:

1. **makePQ**: create an empty queue.
2. **findMin**: find the minimum key in S.
3. **extractMin**: Remove $v \in S$ with smallest key and return it.
4. **insert**$(v, k(v))$: Add new element v with key $k(v)$ to S.
5. **delete**(v): Remove element v from S.
6. **decreaseKey**$(v, k'(v))$: *decrease* key of v from $k(v)$ (current key) to $k'(v)$ (new key). Assumption: $k'(v) \leq k(v)$.
7. **meld**: merge two separate priority queues into one.

All operations can be performed in $O(\log n)$ time.
decreaseKey is implemented via **delete** and **insert**.

Dijkstra's Algorithm using Priority Queues

$Q \leftarrow$ makePQ()
insert(Q, $(s, 0)$)
for each node $u \neq s$ **do**
    insert(Q, (u, ∞))
$X \leftarrow \emptyset$
for $i = 1$ to $|V|$ **do**
    $(v, \mathrm{dist}(s, v)) = \mathit{extractMin}(Q)$
    $X = X \cup \{v\}$
    for each u in $\mathrm{Adj}(v)$ **do**
        decreaseKey$\Big(Q, \big(u, \min\big(\mathrm{dist}(s, u),\ \mathrm{dist}(s, v) + \ell(v, u)\big)\big)\Big)$.

Priority Queue operations:

1. $O(n)$ **insert** operations
2. $O(n)$ **extractMin** operations
3. $O(m)$ **decreaseKey** operations

Implementing Priority Queues via Heaps

Using Heaps

Store elements in a heap based on the key value

1. All operations can be done in $O(\log n)$ time

Implementing Priority Queues via Heaps

Using Heaps

Store elements in a heap based on the key value

1. All operations can be done in $O(\log n)$ time

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

Priority Queues: Fibonacci Heaps/Relaxed Heaps

Fibonacci Heaps

1. **extractMin**, **insert**, **delete**, **meld** in $O(\log n)$ time
2. **decreaseKey** in $O(1)$ *amortized* time:

Priority Queues: Fibonacci Heaps/Relaxed Heaps

Fibonacci Heaps

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

Priority Queues: Fibonacci Heaps/Relaxed Heaps

Fibonacci Heaps

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

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

Priority Queues: Fibonacci Heaps/Relaxed Heaps

Fibonacci Heaps

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

1. Dijkstra's algorithm can be implemented in $O(n \log n + m)$ time. If $m = \Omega(n \log n)$, running time is linear in input size.
2. 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!)

Shortest Path Tree

Dijkstra's algorithm finds the shortest path distances from s to V.
Question: How do we find the paths themselves?

Shortest Path Tree

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

X = ∅
for i = 1 to |V| do
    (v, dist(s, v)) = extractMin(Q)
    X = X ∪ {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
```

Shortest Path Tree

Lemma

The edge set $(u, \mathrm{prev}(u))$ *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.

1. The edge set $\{(u, \mathrm{prev}(u)) \mid u \in V\}$ induces a directed in-tree rooted at s (Why?)
2. Use induction on $|X|$ to argue that the tree is a shortest path tree for nodes in V.

$\square$

Shortest paths to s

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?

Shortest paths to s

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?

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