Breadth First Search, Dijkstra's Algorithm for Shortest Paths

Lecture 3

January 25, 2011

Part I

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...

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

Queue Data Structure

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.

BFS Algorithm

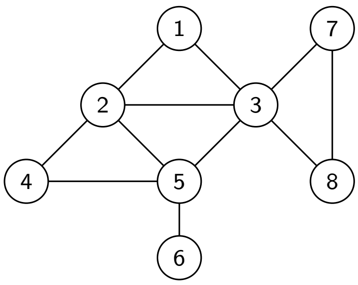
Given (undirected or directed) graph $G = (V, E)$ and node $s \in V$

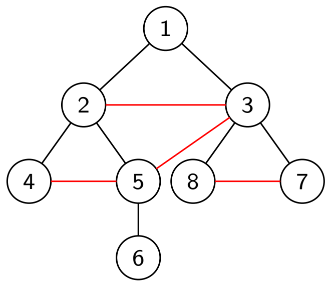
BFS(s)
```
Mark all vertices as unvisited
Initialize search tree T to be empty
Mark vertex s as visited
set Q to be the empty queue
enq(s)
while Q is nonempty do
    u = deq(Q)
    for each vertex v ∈ Adj(u)
        if v is not visited then
            add edge (u, v) to T
            Mark v as visited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

Proposition

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

BFS: An Example in Undirected Graphs

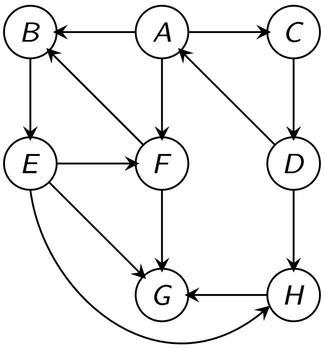




1. [1]
2. [2,3]
3. [3,4,5]
4. [4,5,7,8]
5. [5,7,8]
6. [7,8,6]
7. [8,6]
8. [6]
9. []

BFS tree is the set of black edges.

BFS: An Example in Directed Graphs



BFS with Distance

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

Properties of BFS: Undirected Graphs

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* $dist(u) < dist(v)$ *then* u *is visited before* v.

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

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

Proof.

Exercise. □

Properties of BFS: Directed Graphs

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* $dist(u) < dist(v)$ *then* u *is visited before* v

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

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

Proof.

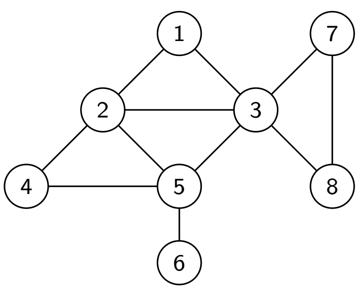
Exercise. □

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: $O(n + m)$

Example



BFS with Layers: Properties

Proposition

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

- **BFSLayers**(s) *outputs a* **BFS** *tree*
- L_i *is the set of vertices at distance exactly* i *from* s
- *If* G *is undirected, each edge* $e = \{u, v\}$ *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*
  - $\implies$ *Every edge in the graph is either between two vertices that are either (i) in the same layer, or (ii) in two consecutive layers.*

BFS with Layers: Properties

For directed graphs

Proposition

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

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

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

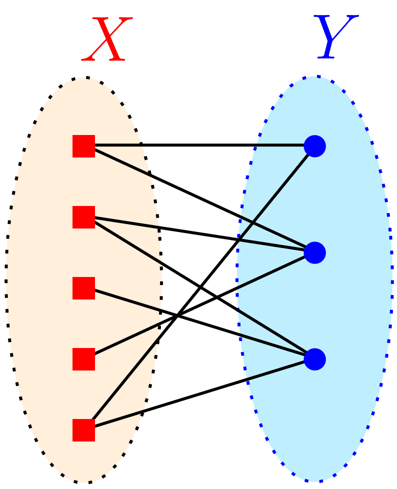
Part II

Bipartite Graphs and an application of BFS

Bipartite Graphs

Definition (Bipartite Graph)

Undirected graph $\mathbf{G = (V, E)}$ is a **bipartite graph** if $\mathbf{V}$ can be partitioned into $\mathbf{X}$ and $\mathbf{Y}$ s.t. all edges in $\mathbf{E}$ are between $\mathbf{X}$ and $\mathbf{Y}$.



Bipartite Graph Characterization

Question

When is a graph bipartite?

Proposition

Every tree is a bipartite graph.

Proof.

Root tree $\mathbf{T}$ at some node $\mathbf{r}$. Let $\mathbf{L_i}$ be all nodes at level $\mathbf{i}$, that is, $\mathbf{L_i}$ is all nodes at distance $\mathbf{i}$ from root $\mathbf{r}$. Now define $\mathbf{X}$ to be all nodes at even levels and $\mathbf{Y}$ to be all nodes at odd level. Only edges in $\mathbf{T}$ are between levels. □

Proposition

An odd length cycle is not bipartite.

Odd Cycles are not Bipartite

Proposition

An odd length cycle is not bipartite.

Proof.

Let $\mathbf{C} = \mathbf{u_1}, \mathbf{u_2}, \ldots, \mathbf{u_{2k+1}}, \mathbf{u_1}$ be an odd cycle. Suppose $\mathbf{C}$ is a bipartite graph and let $\mathbf{X}, \mathbf{Y}$ be the bipartition. Without loss of generality $\mathbf{u_1} \in \mathbf{X}$. Implies $\mathbf{u_2} \in \mathbf{Y}$. Implies $\mathbf{u_3} \in \mathbf{X}$. Inductively, $\mathbf{u_i} \in \mathbf{X}$ if $\mathbf{i}$ is odd $\mathbf{u_i} \in \mathbf{Y}$ if $\mathbf{i}$ is even. But $\{\mathbf{u_1}, \mathbf{u_{2k+1}}\}$ is an edge and both belong to $\mathbf{X}$! □

Subgraphs

Definition

Given a graph $\mathbf{G} = (\mathbf{V}, \mathbf{E})$ a **subgraph** of $\mathbf{G}$ is another graph $\mathbf{H} = (\mathbf{V}', \mathbf{E}')$ where $\mathbf{V}' \subseteq \mathbf{V}$ and $\mathbf{E}' \subseteq \mathbf{E}$.

Proposition

If $\mathbf{G}$ is bipartite then any subgraph $\mathbf{H}$ of $\mathbf{G}$ is also bipartite.

Proposition

A graph $\mathbf{G}$ is not bipartite if $\mathbf{G}$ has an odd cycle $\mathbf{C}$ as a subgraph.

Proof.

If $\mathbf{G}$ is bipartite then since $\mathbf{C}$ is a subgraph, $\mathbf{C}$ is also bipartite (by above proposition). However, $\mathbf{C}$ is not bipartite! □

Bipartite Graph Characterization

Theorem

A graph $\mathbf{G}$ is bipartite if and only if it has no odd length cycle as subgraph.

Proof.

Only If: $\mathbf{G}$ has an odd cycle implies $\mathbf{G}$ is not bipartite.
If: $\mathbf{G}$ has no odd length cycle. Assume without loss of generality that $\mathbf{G}$ is connected.

- Pick $\mathbf{u}$ arbitrarily and do **BFS**$(\mathbf{u})$
- $\mathbf{X} = \cup_{\mathbf{i} \text{ is even}} \mathbf{L_i}$ and $\mathbf{Y} = \cup_{\mathbf{i} \text{ is odd}} \mathbf{L_i}$
- **Claim:** $\mathbf{X}$ and $\mathbf{Y}$ is a valid bipartition if $\mathbf{G}$ has no odd length cycle.

□

Proof of Claim

Claim

In $\mathbf{BFS}(\mathbf{u})$ *if* $\mathbf{a}, \mathbf{b} \in \mathbf{L}_{\mathbf{i}}$ *and* $(\mathbf{a}, \mathbf{b})$ *is an edge then there is an odd length cycle containing* $(\mathbf{a}, \mathbf{b})$.

Proof.

Let $\mathbf{v}$ be least common ancestor of $\mathbf{a}, \mathbf{b}$ in **BFS** tree $\mathbf{T}$.
$\mathbf{v}$ is in some level $\mathbf{j} < \mathbf{i}$ (could be $\mathbf{u}$ itself).
Path from $\mathbf{v} \rightsquigarrow \mathbf{a}$ in $\mathbf{T}$ is of length $\mathbf{j} - \mathbf{i}$.
Path from $\mathbf{v} \rightsquigarrow \mathbf{b}$ in $\mathbf{T}$ is of length $\mathbf{j} - \mathbf{i}$.
These two paths plus $(\mathbf{a}, \mathbf{b})$ forms an odd cycle of length $\mathbf{2(j - i) + 1}$. □

Corollary

There is an $\mathbf{O(n + m)}$ *time algorithm to check if* $\mathbf{G}$ *is bipartite and output an odd cycle if it is not.*

Part III

Shortest Paths and Dijkstra's Algorithm

Shortest Path Problems

Shortest Path Problems

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

Single-Source Shortest Paths: Non-Negative Edge Lengths

Single-Source Shortest Path Problems

Input A (undirected or directed) graph $\mathbf{G = (V, E)}$ with non-negative edge lengths. For edge $\mathbf{e = (u, v)}$, $\ell(\mathbf{e}) = \ell(\mathbf{u, v})$ 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 $\ell(\mathbf{u, v}) = \ell(\mathbf{v, u}) = \ell(\{\mathbf{u, v}\})$
  - Exercise: show reduction works

Single-Source Shortest Paths via BFS

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.

Towards an algorithm

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

Lemma

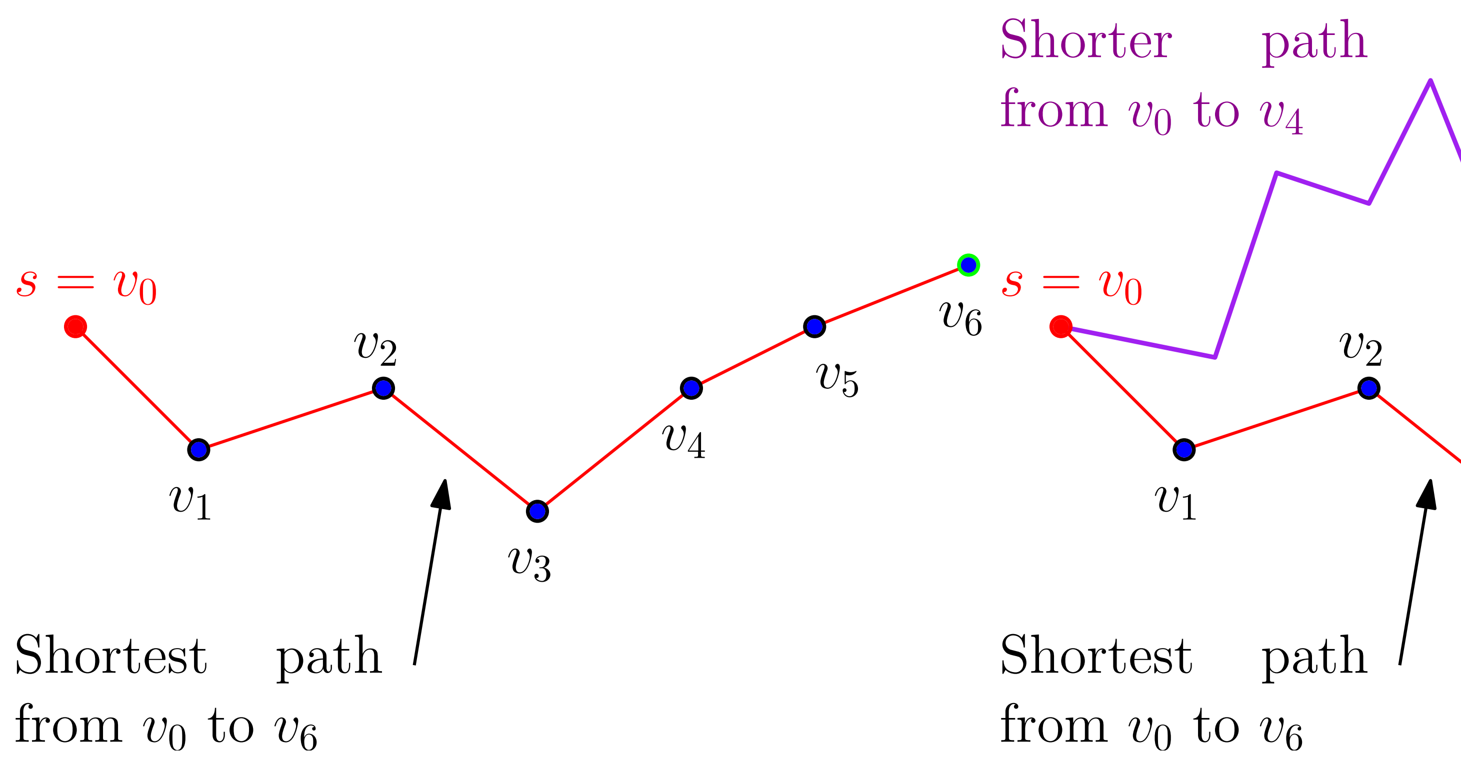
Let G be a directed graph with non-negative edge lengths. Let $\mathrm{dist}(s, v)$ denote the shortest path length from s to v. If $s = v_0 \to v_1 \to v_2 \to \dots \to v_k$ is a shortest path from s to v_k then for $1 \le i < k$:

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

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 $s = v_0 \to v_1 \to \dots \to v_i$. Then P' concatenated with $v_i \to v_{i+1} \dots \to v_k$ contains a strictly shorter path to v_k than $s = v_0 \to v_1 \dots \to v_k$.

A proof by picture



A Basic Strategy

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

```
Initialize for each node v, dist(s, v) = ∞
Initialize S = ∅,
for i = 1 to |V| do
    (* Invariant: S contains the i − 1 closest nodes to s *)
    Among nodes in V \ S, find the node v that is the
            ith closest to s
    Update dist(s, v)
    S = S ∪ {v}
```

How can we implement the step in the for loop?

Finding the ith closest node

- **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

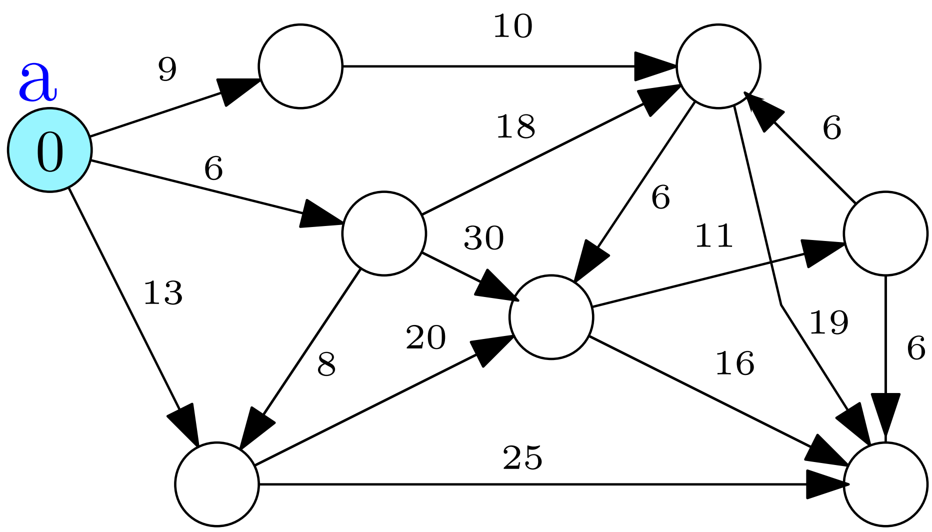
Let **P** *be a shortest path from* **s** *to* **v** *where* **v** *is the* **i***th closest node. Then, all intermediate nodes in* **P** *belong to* **S***.*

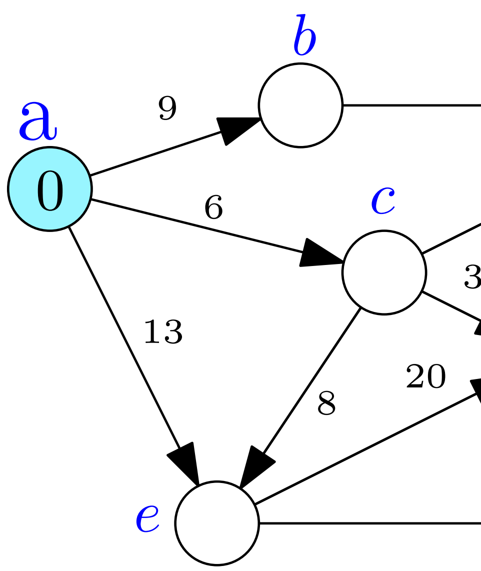
Proof.

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. □

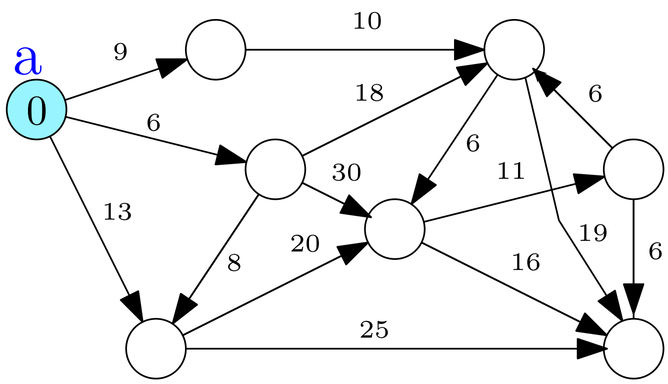
Finding the ith closest node repeatedly

An example





Finding the ith closest node



Corollary

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

Finding the ith closest node

- $\mathbf{S}$ contains the $\mathbf{i-1}$ closest nodes to $\mathbf{s}$
- Want to find the $\mathbf{i}$th closest node from $\mathbf{V-S}$.
- For each $\mathbf{u \in V - S}$ let $\mathbf{P(s,u,S)}$ be a shortest path from $\mathbf{s}$ to $\mathbf{u}$ using only nodes in $\mathbf{S}$ as intermediate vertices.
- Let $\mathbf{d'(s,u)}$ be the length of $\mathbf{P(s,u,S)}$

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

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

Lemma

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

Finding the ith closest node

Lemma

If v is an ith 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 S as intermediate nodes (see previous claim). Therefore $d'(s, v) = \mathrm{dist}(s, v)$. □

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 $v \in V - S$ such that $d'(s, v) = \min_{u \in V - S} d'(s, u)$.

Proof.

For every node $u \in V - S$, $\mathrm{dist}(s, u) \leq d'(s, u)$ 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 $u \in V - S$. □

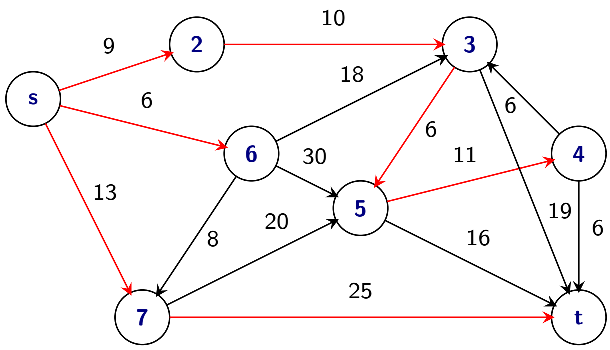
Algorithm

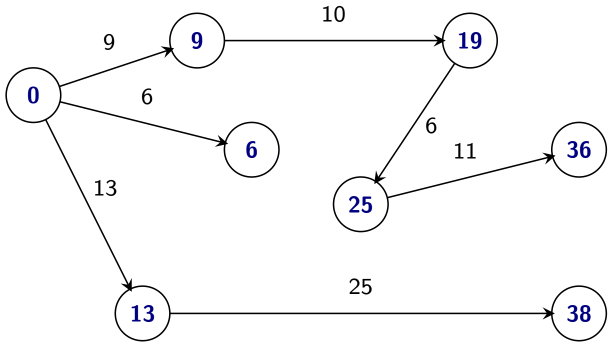
Initialize for each node **v**: $\text{dist}(s, v) = \infty$
Initialize S = $\emptyset$, $d'(s, s) = 0$
for $i = 1$ to $|V|$ **do**
    (* Invariant: S contains the i-1 closest nodes to s *)
    (* Invariant: d'(s,u) is shortest path distance from u to s
     using only S as intermediate nodes*)
   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

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.

Example





Improved Algorithm

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

```
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: $\mathbf{O(m+n^2)}$ time.

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

Dijkstra's Algorithm

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

```
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: $\mathbf{O((m+n)\log n)}$
- Using Fibonacci heaps: $\mathbf{O(m+n\log n)}$.

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

- `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`

Dijkstra's Algorithm using Priority Queues

```
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

Implementing Priority Queues via Heaps

Using Heaps

Store elements in a heap based on the key value

- 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

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

— 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.
— 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?

```
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
```


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.

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

□

Shortest paths to $\mathbf{s}$

Dijkstra's algorithm gives shortest paths from $\mathbf{s}$ to all nodes in $\mathbf{V}$.

How do we find shortest paths from all of $\mathbf{V}$ to $\mathbf{s}$?

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