CS 473: Fundamental Algorithms, Spring 2013

Breadth First Search, Dijkstra's Algorithm for Shortest Paths

Lecture 3

January 24, 2013

Part I

Breadth First Search

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...

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 picked in the order in which they were inserted.

Algorithm

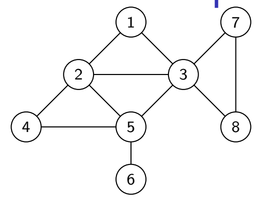
Given (undirected or directed) graph $G = (V, E)$ and node $s \in V$

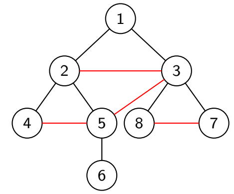
```
BFS(s)
    Mark all vertices as unvisited
    Initialize search tree T to be empty
    Mark vertex s as visited
    set Q to be the empty queue
    enq(s)
    while Q is nonempty do
        u = deq(Q)
        for each vertex v ∈ Adj(u)
            if v is not visited then
                add edge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

Proposition

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

: An Example in Undirected Graphs

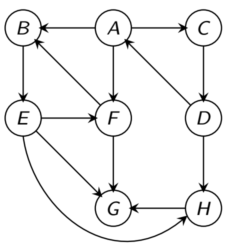




1. [1]
2. [2,3]
3. [3,4,5]
4. [4,5,7,8]
5. [5,7,8]
6. [7,8,6]
7. [8,6]
8. [6]
9. []

BFS tree is the set of black edges.

: An Example in Directed Graphs



with Distance

```
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
```

Properties of : 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* $\mathrm{dist}(\mathbf{u}) < \mathrm{dist}(\mathbf{v})$ *then* **u** *is visited before* **v**.

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

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

Proof.

Exercise. □

Properties of : 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* $\mathrm{dist}(\mathbf{u}) < \mathrm{dist}(\mathbf{v})$ *then* **u** *is visited before* **v**

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

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

Proof.

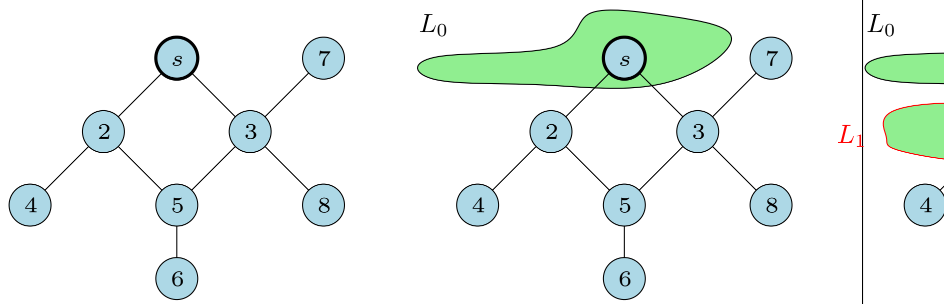
Exercise. □

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



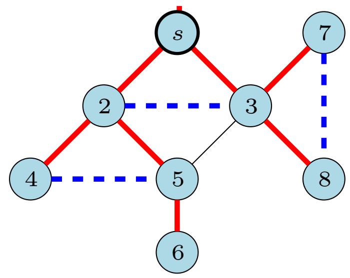
with Layers: Properties

Proposition

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

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

Example



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:*

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

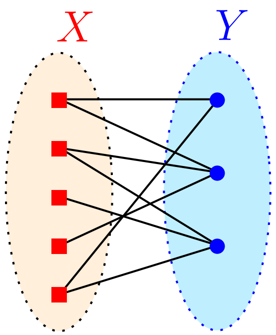
Part II

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

Subgraphs

Definition

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

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.

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

□

Proof of Claim

Claim

In $\mathbf{BFS}(\mathbf{u})$ if $\mathbf{a}, \mathbf{b} \in \mathbf{L_i}$ 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 $\mathbf{BFS}$ 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 $\mathbf{2(j - i) + 1}$.

□

Proof of Claim: Figure

Another tidbit

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 $G = (V, E)$ 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.

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

Single-Source Shortest Paths via

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**? 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 $\mathbf{s}$

Lemma

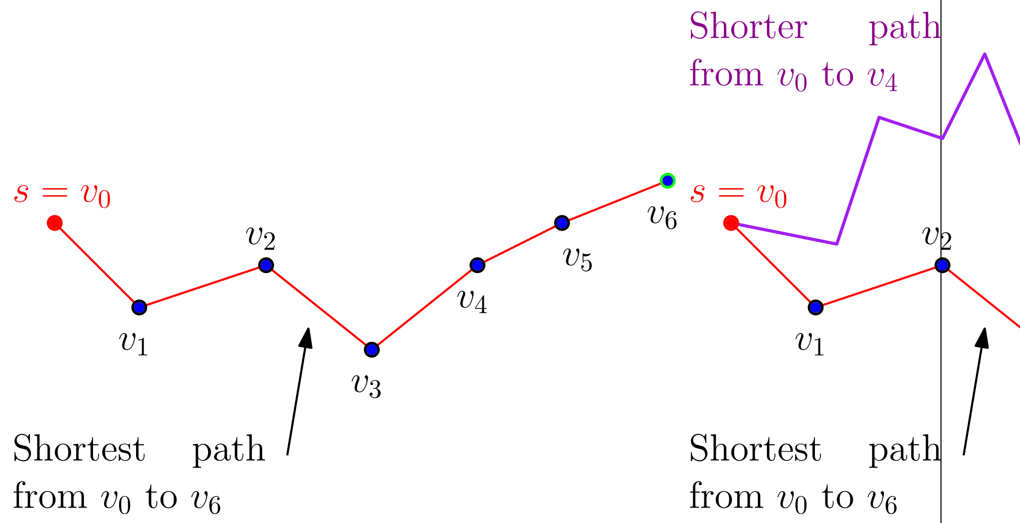
Let $\mathbf{G}$ *be a directed graph with non-negative edge lengths. Let* $\mathrm{dist}(\mathbf{s},\mathbf{v})$ *denote the shortest path length from* $\mathbf{s}$ *to* $\mathbf{v}$*. If* $\mathbf{s} = \mathbf{v}_0 \rightarrow \mathbf{v}_1 \rightarrow \mathbf{v}_2 \rightarrow \ldots \rightarrow \mathbf{v}_k$ *is a shortest path from* $\mathbf{s}$ *to* $\mathbf{v}_k$ *then for* $1 \leq \mathbf{i} < \mathbf{k}$:

1. $\mathbf{s} = \mathbf{v}_0 \rightarrow \mathbf{v}_1 \rightarrow \mathbf{v}_2 \rightarrow \ldots \rightarrow \mathbf{v}_i$ *is a shortest path from* $\mathbf{s}$ *to* $\mathbf{v}_i$
2. $\mathrm{dist}(\mathbf{s},\mathbf{v}_i) \leq \mathrm{dist}(\mathbf{s},\mathbf{v}_k)$.

Proof.

Suppose not. Then for some $\mathbf{i} < \mathbf{k}$ there is a path $\mathbf{P}'$ from $\mathbf{s}$ to $\mathbf{v}_i$ of length strictly less than that of $\mathbf{s} = \mathbf{v}_0 \rightarrow \mathbf{v}_1 \rightarrow \ldots \rightarrow \mathbf{v}_i$. Then $\mathbf{P}'$ concatenated with $\mathbf{v}_i \rightarrow \mathbf{v}_{i+1} \ldots \rightarrow \mathbf{v}_k$ contains a strictly shorter path to $\mathbf{v}_k$ than $\mathbf{s} = \mathbf{v}_0 \rightarrow \mathbf{v}_1 \ldots \rightarrow \mathbf{v}_k$. □

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

1. $\mathbf{S}$ contains the $\mathbf{i} - 1$ closest nodes to $\mathbf{s}$
2. Want to find the $\mathbf{i}$th closest node from $\mathbf{V} - \mathbf{S}$.

What do we know about the $\mathbf{i}$th closest node?

Claim

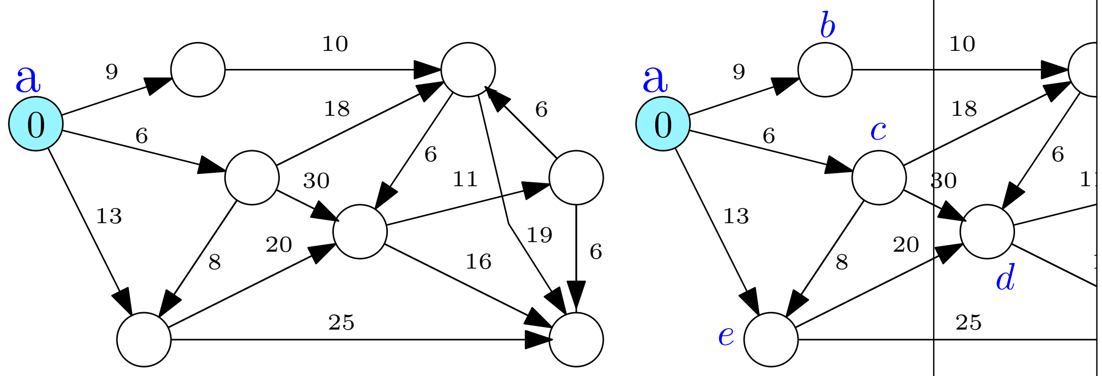
Let $\mathbf{P}$ *be a shortest path from* $\mathbf{s}$ *to* $\mathbf{v}$ *where* $\mathbf{v}$ *is the* $\mathbf{i}$*th closest node. Then, all intermediate nodes in* $\mathbf{P}$ *belong to* $\mathbf{S}$.

Proof.

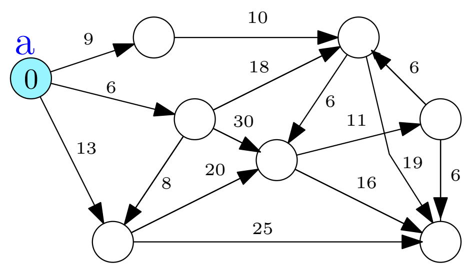
If $\mathbf{P}$ had an intermediate node $\mathbf{u}$ not in $\mathbf{S}$ then $\mathbf{u}$ will be closer to $\mathbf{s}$ than $\mathbf{v}$. Implies $\mathbf{v}$ is not the $\mathbf{i}$th closest node to $\mathbf{s}$ - recall that $\mathbf{S}$ already has the $\mathbf{i} - 1$ closest nodes. □

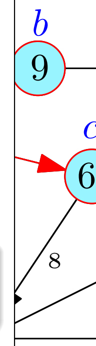
Finding the ith closest node repeatedly

An example



Finding the ith closest node





Corollary

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

Finding the ith closest node

1. **S** contains the $\mathbf{i - 1}$ closest nodes to **s**
2. Want to find the **i**th closest node from $\mathbf{V - S}$.

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

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

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

Lemma

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

Finding the ith closest node

Lemma

Given:

1. **S**: *Set of* $\mathbf{i - 1}$ *closest nodes to* **s**.
2. $\mathbf{d'(s, u)} = \mathbf{min}_{\mathbf{x \in S}}(\mathrm{dist}(\mathbf{s}, \mathbf{x}) + \ell(\mathbf{x}, \mathbf{u}))$

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

Proof.

Let **v** be the **i**th 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 $\mathbf{d'(s, v)} = \mathrm{dist}(\mathbf{s}, \mathbf{v})$. □

Finding the ith closest node

Lemma

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

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

Algorithm

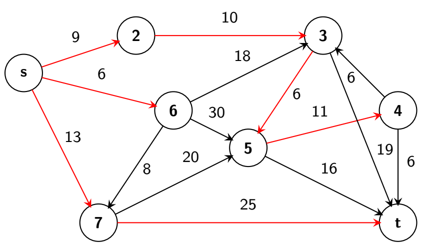
```
Initialize for each node v:  dist(s,v) = ∞
Initialize S = ∅,  d'(s,s) = 0
for i = 1 to |V| do
    (* Invariant:  S contains the i-1 closest nodes to s *)
    (* Invariant:  d'(s,u) is shortest path distance from u to s
     using only S as intermediate nodes*)
    Let v be such that d'(s,v) = min_{u∈V−S}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

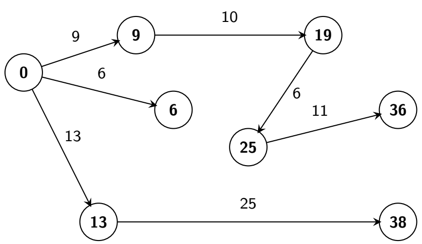
Correctness: By induction on $\mathbf{i}$ using previous lemmas.

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

1. $\mathbf{n}$ 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.

Example





Improved Algorithm

1. Main work is to compute the $\mathbf{d'(s,u)}$ values in each iteration
2. $\mathbf{d'(s,u)}$ 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
    v be node realizing 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.

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

3. Finding $\mathbf{v}$ from $\mathbf{d'(s,u)}$ values is $\mathbf{O(n)}$ time

Dijkstra's Algorithm

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

```
Initialize for each node v, dist(s,v) = ∞
Initialize 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

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

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.
6. **decreaseKey**$(v, k'(v))$: *decrease* key of v from $k(v)$ (current key) to $k'(v)$ (new key). Assumption: $k'(v) \le k(v)$.
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 ⇐ 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:

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

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: 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?

```
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.

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 $|S|$ to argue that the tree is a shortest path tree for nodes in **V**.

□

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}!