Breadth First Search, Dijkstra's Algorithm for Shortest Paths

Lecture 3

August 30, 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 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

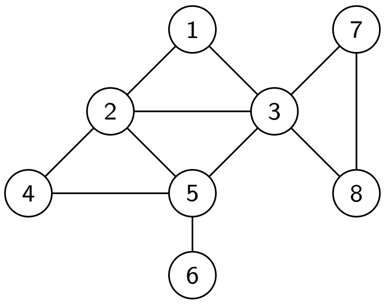
Given (undirected or directed) graph $G = (V, E)$ and node $s \in V$

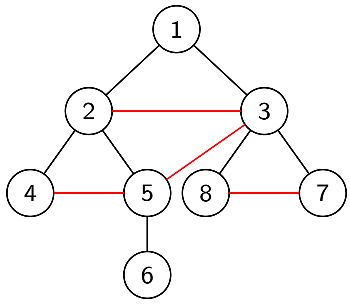
BFS(s)
```
Mark all vertices as unvisited
Initialize search tree T to be empty
Mark vertex s as visited
set Q to be the empty queue
enq(s)
while Q is nonempty do
    u = deq(Q)
    for each vertex v ∈ Adj(u)
        if v is not visited then
            add edge (u, v) to T
            Mark v as visited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

Proposition

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

BFS: An Example in Undirected Graphs

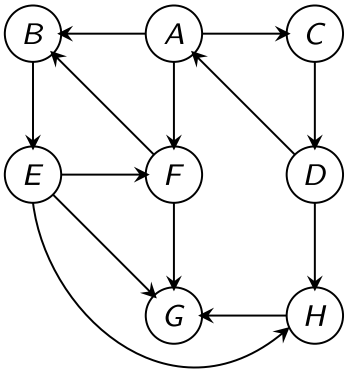




1. [1]
2. [2,3]
3. [3,4,5]
4. [4,5,7,8]
5. [5,7,8]
6. [7,8,6]
7. [8,6]
8. [6]
9. []

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 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

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

(C) *For every vertex* u, $\mathrm{dist}(u)$ *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* $|\mathrm{dist}(u) - \mathrm{dist}(v)| \leq 1$.

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

(C) *For every vertex* u, $\mathrm{dist}(u)$ *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* $\mathrm{dist}(v) - \mathrm{dist}(u) \leq 1$. *Not necessarily the case that* $\mathrm{dist}(u) - \mathrm{dist}(v) \leq 1$.

Proof.

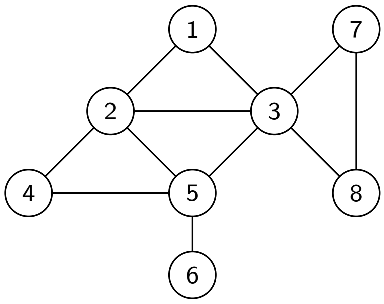
Exercise. □

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 $(u, v) \in \mathrm{Adj}(u)$ **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*

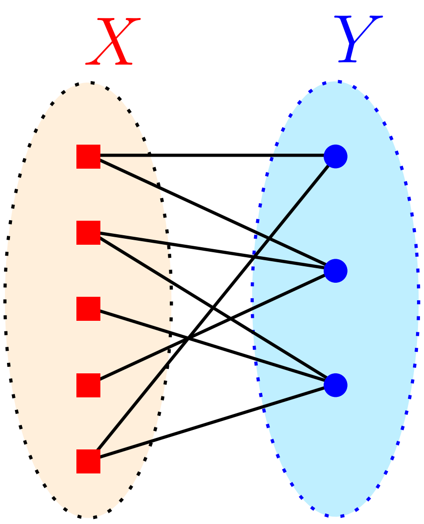
Part II

Bipartite Graphs and an application of BFS

Bipartite Graphs

Definition (Bipartite Graph)

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



Bipartite Graph Characterization

Question

When is a graph bipartite?

Proposition

Every tree is a bipartite graph.

Proof.

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

Proposition

An odd length cycle is not bipartite.

Odd Cycles are not Bipartite

Proposition

An odd length cycle is not bipartite.

Proof.

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

Subgraphs

Definition

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

Proposition

If G is bipartite then any subgraph H of G is also bipartite.

Proposition

A graph G is not bipartite if G has an odd cycle C as a subgraph.

Proof.

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

Bipartite Graph Characterization

Theorem

A graph G is bipartite if and only if it has no odd length cycle as subgraph.

Proof.

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

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

$\square$

Proof of Claim

Claim

In **BFS**(u) *if* $a, b \in L_i$ *and* (a, b) *is an edge then there is an odd length cycle containing* (a, b).

Proof.

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

Another tidbit

Corollary

There is an $O(n + m)$ *time algorithm to check if* 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.

- Given nodes s, t find shortest path from s to t.
- Given node s find shortest path from 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 $G = (V, E)$ with non-negative edge lengths. For edge $e = (u, v)$, $\ell(e) = \ell(u, v)$ is its length.
- Given nodes s, t find shortest path from s to t.
- Given node s find shortest path from s to all other nodes.

- Restrict attention to directed graphs
- Undirected graph problem can be reduced to directed graph problem - how?
  - 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'.
  - set $\ell(u, v) = \ell(v, u) = \ell(\{u, v\})$
  - 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

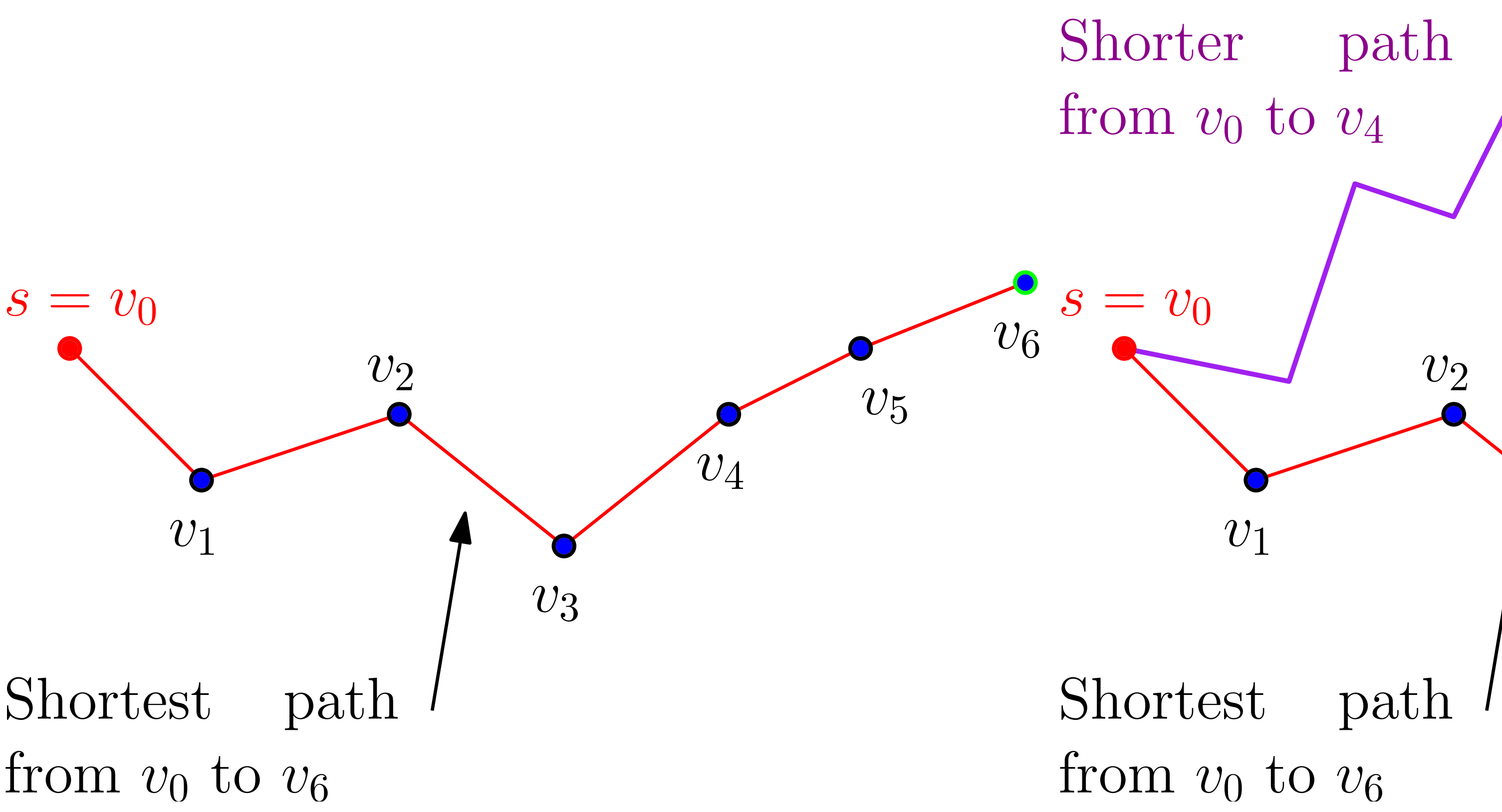
Let G be a directed graph with non-negative edge lengths. Let $\mathrm{dist}(s, v)$ denote the shortest path length from s to v. If $s = v_0 \to v_1 \to v_2 \to \ldots \to v_k$ is a shortest path from s to v_k then for $1 \leq i < k$:

- *$s = v_0 \to v_1 \to v_2 \to \ldots \to v_i$ 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

A proof by picture



A Basic Strategy

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, 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 ith closest node from $V - S$.

What do we know about the ith closest node?

Claim

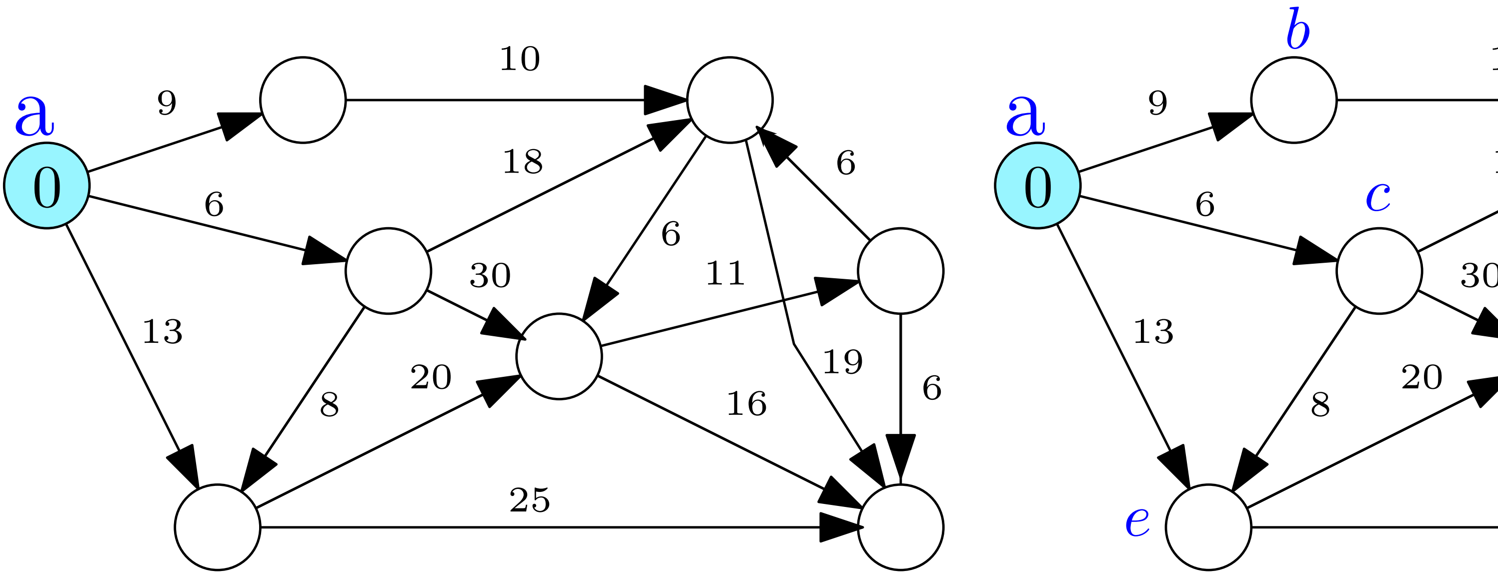
Let P be a shortest path from s to v where v is the ith closest node. Then, all intermediate nodes in P belong to S.

Proof.

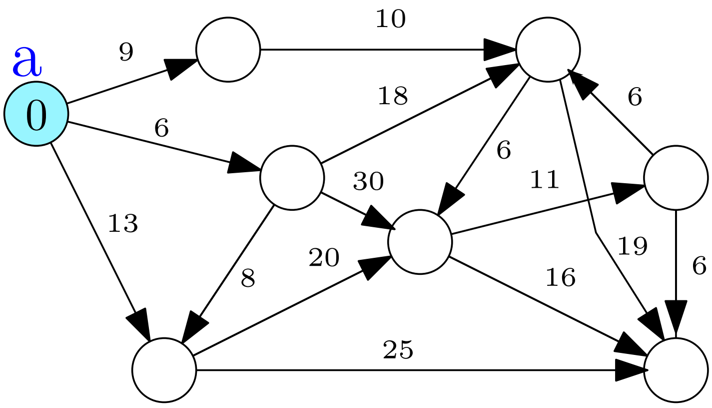
If P had an intermediate node u not in S then u will be closer to s than v. Implies v is not the ith closest node to s - recall that S already has the $i - 1$ closest nodes. $\square$

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

- 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 $u \in V-S$,

- $\mathrm{dist}(s,u) \le d'(s,u)$ 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 $d'(s,v) = \mathrm{dist}(s,v)$.

Finding the ith closest node

Lemma

Given:

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

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 $u \in V - S$, $\mathrm{dist}(s,u) \le d'(s,u)$ and for the ith closest node v, $\mathrm{dist}(s,v) = d'(s,v)$. Moreover, $\mathrm{dist}(s,u) \ge \mathrm{dist}(s,v)$ for each $u \in V - S$. □

Algorithm

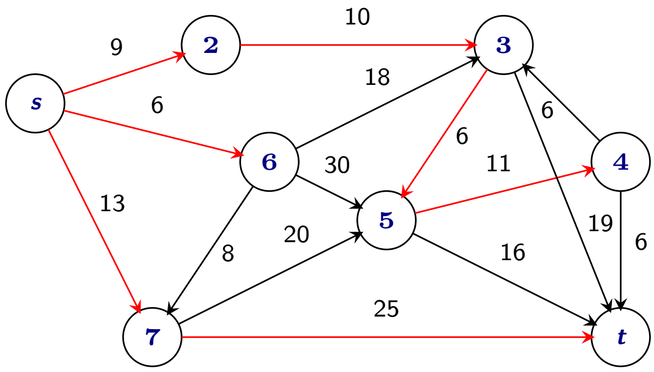
Initialize for each node v: $\mathrm{dist}(s,v) = \infty$
Initialize S = $\emptyset$, $d'(s,s) = 0$
for $i = 1$ to $|V|$ **do**
    (* Invariant: S contains the i-1 closest nodes to s *)
    (* Invariant: d'(s,u) is shortest path distance from u to s using only S as intermediate nodes*)
    Let v be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

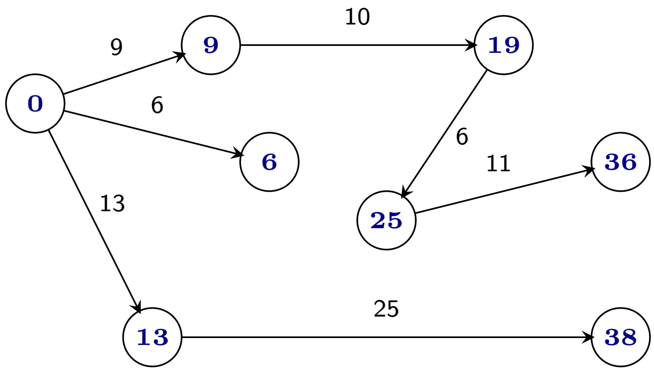
Correctness: By induction on i using previous lemmas.

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 $d'(s, u)$ 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, $\mathrm{dist}(s, v) = d'(s, v) = \infty$
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
  v be node realizing $d'(s, v) = \min_{u \in V - S} d'(s, u)$
  $\mathrm{dist}(s, v) = d'(s, v)$
  $S = S \cup \{v\}$
  Update $d'(s, u)$ for each u in $V - S$ as follows:
    $d'(s, u) = min\Big(d'(s, u),\ \mathrm{dist}(s, v) + \ell(v, u)\Big)$

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

Dijkstra's Algorithm

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

Initialize for each node v, $\mathrm{dist}(s, v) = \infty$
Initialize S = {s}, $\mathrm{dist}(s, s) = 0$
for $i = 1$ to $|V|$ do
    Let v be such that $\mathrm{dist}(s, v) = \min_{u \in V-S} \mathrm{dist}(s, u)$
    $S = S \cup \{v\}$
    for each u in $\mathrm{Adj}(v)$ do
        $\mathrm{dist}(s, u) = min\Big(\mathrm{dist}(s, u),\ \mathrm{dist}(s, v) + \ell(v, u)\Big)$

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

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

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:

- **makePQ**: create an empty queue.
- **findMin**: find the minimum key in S.
- **extractMin**: Remove $v \in S$ with smallest key and return it.
- **insert**$(v, k(v))$: 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.

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, ∞))
$S \Leftarrow \emptyset$
for $i = 1$ to $|V|$ **do**
    $(v, \mathrm{dist}(s, v)) = \mathit{extractMin}(Q)$
    $S = S \cup \{v\}$
    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:

- $O(n)$ **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**, **insert**, **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 edge set $\{(u, \mathrm{prev}(u)) \mid u \in V\}$ 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.

$\square$

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?

- 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 G^{rev}!