OLD CS 473: Fundamental Algorithms, Spring 2015

Breadth First Search, Dijkstra's Algorithm for Shortest Paths

Lecture 4

January 29, 2015

Part I

Breadth First Search

Breadth First Search (BFS)

Overview

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

(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

queue: 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.

BFS Algorithm

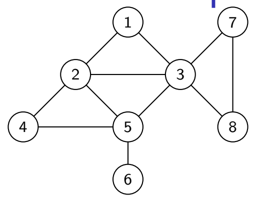
Given (undirected or directed) graph $G = (V, E)$ and node $s \in V$

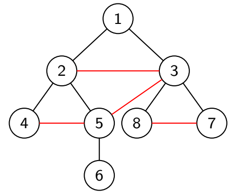
BFS(s)
  Mark all vertices as unvisited
  Initialize search tree T to be empty
  Mark vertex s as visited
  set Q to be the empty queue
  enq(s)
  while Q is nonempty **do**
    $u =$ **deq**(Q)
    for each vertex $v \in \mathrm{Adj}(u)$
      if v is not visited **then**
        add edge (u, v) to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

Proposition

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

BFS: An Example in Undirected Graphs

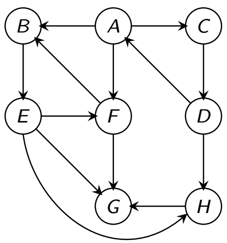




1. [1]
2. [2,3]
3. [3,4,5]
4. [4,5,7,8]
5. [5,7,8]
6. [7,8,6]
7. [8,6]
8. [6]
9. []

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)

1. $V($ **BFS** *tree comp.*$)$ = *set vertices in connected component* s.
2. *If* $\mathrm{dist}(u) < \mathrm{dist}(v)$ *then* u *is visited before* v.
3. $\forall u \in V$, $\mathrm{dist}(u)$ = *the length of shortest path from* s *to* u.
4. *If* $u, v \in$ *connected component of* s, *and* $e = uv$ *is an edge of* G, *then either* $e \in$ **BFS** *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 $T \leftarrow$ **BFS**(s):

1. *For search tree* T. $V(T)$ = *set of vertices reachable from* s
2. *If* $\mathrm{dist}(u) < \mathrm{dist}(v)$ *then* u *is visited before* v
3. $\forall u \in V(T)$: $\mathrm{dist}(u)$ = *length of shortest path from* s *to* u
4. *If* u *is reachable from* s, $e = (u \to v) \in E(G)$. *Then either (i)* e *is an edge in the search tree, or (ii)* $\mathrm{dist}(v) - \mathrm{dist}(u) \leq 1$.
   Not necessarily the case that $\mathrm{dist}(u) - \mathrm{dist}(v) \leq 1$.

Proof.

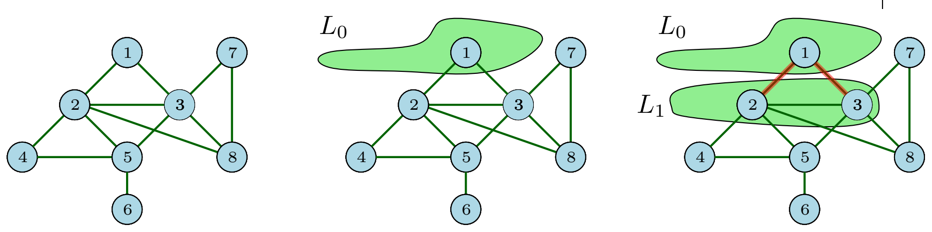
Exercise. □

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) ∈ 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



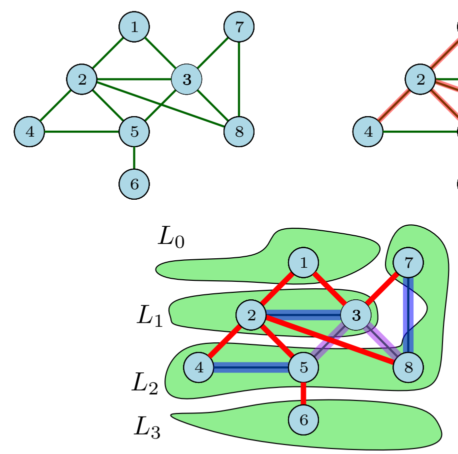
BFS with Layers: Properties

Proposition

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

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* $e = uv$ *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: Tree/cross/forward (backward) edges



BFS with Layers: Properties

For directed graphs

Proposition

The following properties hold on termination of **BFSLayers**(s), *if G is directed.*
For each edge $e = (u \to v)$ *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*

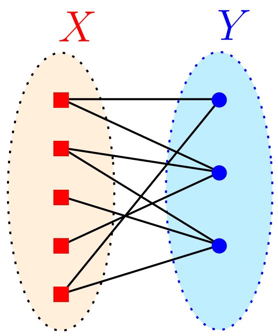
Part II

Bipartite Graphs and an application of BFS

Bipartite Graphs

Definition (Bipartite Graph)

Undirected graph $G = (V, E)$ 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. □

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! □

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 an undirected G is bipartite then any subgraph H of G is also bipartite.

Proposition

An undirected 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! □

Bipartite Graph Characterization

Theorem

An undirected graph G is bipartite $\iff$ 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.

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

□

Proof of Claim

Claim

In $\mathbf{BFS}(u)$ if $a, b \in L_i$ and $ab \in E(G)$ then there is an odd length cycle containing ab.

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 $2(j - i) + 1$.

□

Proof of Claim: Figure

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 $e = (u \to v)$, $\ell(e) = \ell(u \to v)$ 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 $e = (u \to v)$, $\ell(e) = \ell(u \to v)$ 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 $(u \to v)$ and (v, u) in G'.
   2. set $\ell(u \to v) = \ell(v, u) = \ell(\{u, v\})$
   3. Exercise: show reduction works

Single-Source Shortest Paths via BFS

1. **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.
2. **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.
3. 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

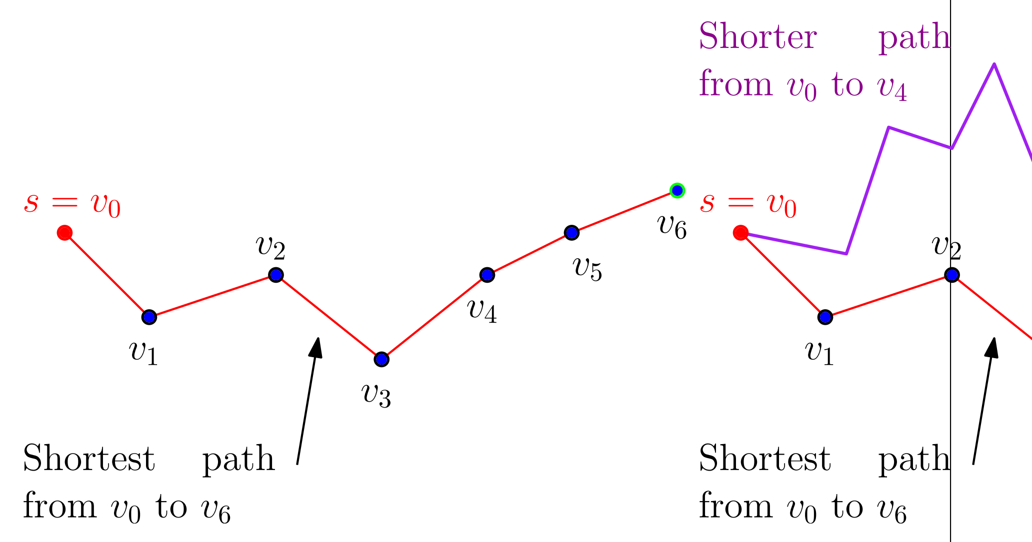
Let G be a directed graph with non-negative edge lengths. Let $\mathrm{dist}(s, v)$ denote the shortest path length from s to v. If $s = v_0 \rightarrow v_1 \rightarrow v_2 \rightarrow \ldots \rightarrow v_k$ is a shortest path from s to v_k then for $1 \leq i < k$:

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

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

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

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

What do we know about the ith closest node?

Claim

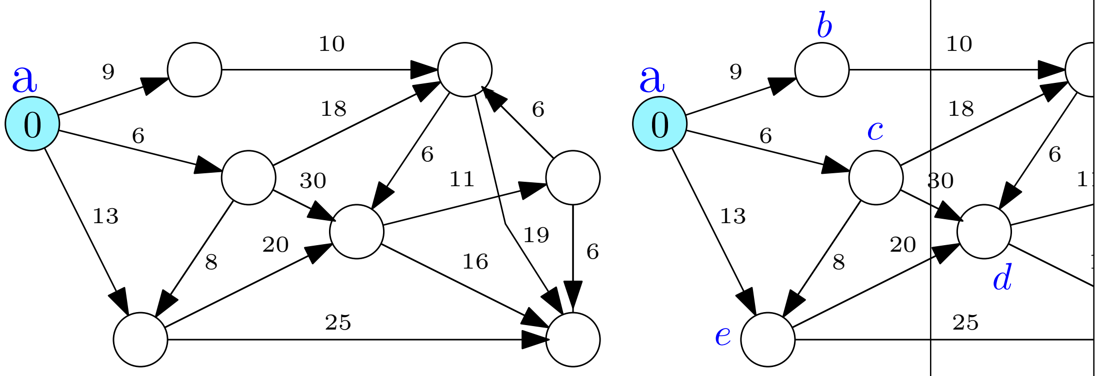
Let P be a shortest path from s to v where v is the ith closest node. Then, all intermediate nodes in P belong to S.

Proof.

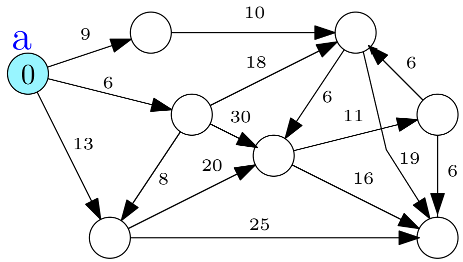
If P had an intermediate node u not in S then u will be closer to s than v. Implies v is not the ith closest node to s - recall that S already has the $i - 1$ closest nodes. □

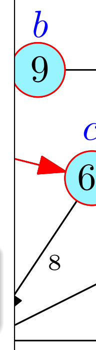
Finding the ith closest node repeatedly

An example



Finding the ith closest node





Corollary

The ith closest node is adjacent to S.

Finding the ith closest node

1. S contains the $i-1$ closest nodes to s
2. Want to find the ith closest node from $V-S$.
3. For each $u \in \mathrm{V} \setminus S$ let $P(s,u,S)$ be a shortest path from s to u using only nodes in S as intermediate vertices.
4. Let $d'(s,u)$ be the length of $P(s,u,S)$
5. Observations: for each $u \in V-S$,
    1. $\mathrm{dist}(s,u) \le d'(s,u)$ since we are constraining the paths
    2. $d'(s,u) = \min_{a \in S}(\mathrm{dist}(s,a) + \ell(a,u))$ - Why?
6. **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. S: *Set of* $i-1$ *closest nodes to* s.
2. $d'(s,u) = \min_{x \in S}(\mathrm{dist}(s,x) + \ell(x,u))$

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 $u \in V - S$. □

Candidate algorithm for shortest path

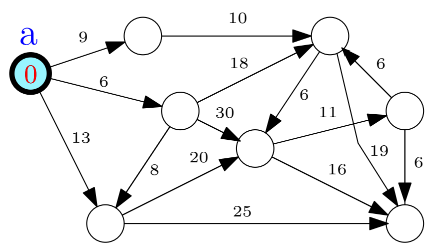
```
Initialize for each node v:  dist(s,v) = ∞
Initialize S = ∅,  d'(s,s) = 0
for i = 1 to |V| do
    (* Invariant:  S contains the i-1 closest nodes to s *)
    (* Invariant:  d'(s,u) is shortest path distance from u to s
     using only S as intermediate nodes*)
    Let v be such that d'(s,v) = min_{u∈V−S}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

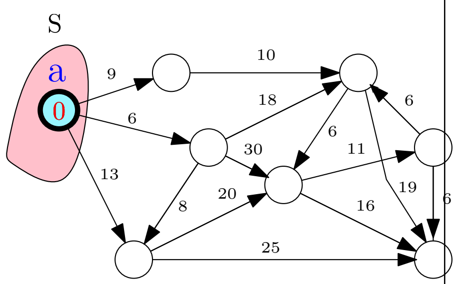
Correctness: By induction on i using previous lemmas.

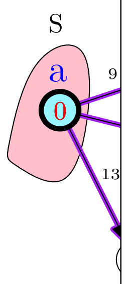
Running time: $O(n \cdot (n+m))$ time.

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

Example







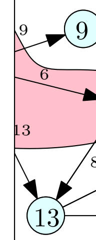
Improved Algorithm

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 S in iteration i.

```
Initialize for each node v, dist(s,v) = d'(s,v) = ∞
Initialize S = ∅, d'(s,s) = 0
for i = 1 to |V| do
    // S contains the i − 1 closest nodes to s,
    //       and the values of d'(s,u) are current
    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: $O(m + n^2)$ time.

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



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

Dijkstra's Algorithm

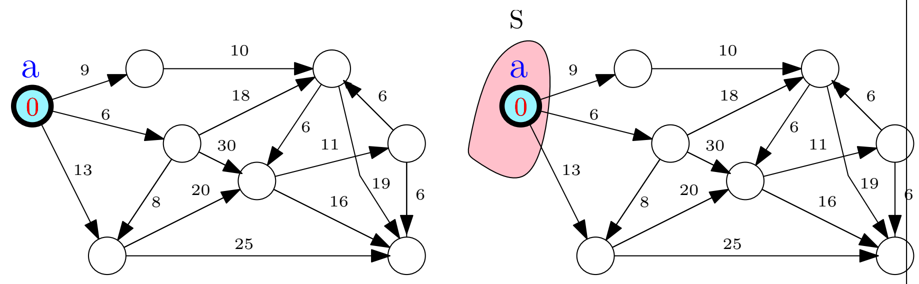
1. eliminate $d'(s,u)$ and let $\text{dist}(s,u)$ maintain it
2. update *dist* values after adding v by scanning edges out of v

```
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: $O(m + n\log n)$.

Example: Dijkstra algorithm in action



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**, **delete** in $O(\log n)$ time.
2. **insert** in $O(1)$ *amortized* time.
3. **decreaseKey** in $O(1)$ *amortized* time: ℓ **decreaseKey** operations for $\ell \geq n$ take *together* $O(\ell)$ time
4. 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}!