BFS and Dijkstra’s Algorithm

Lecture 17
Part I

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

**Explore**($G,u$):

array $Visited[1..n]$

Initialize: Set $Visited[i] = \text{FALSE}$ for $1 \leq i \leq n$

List: $ToExplore$, $S$

Add $u$ to $ToExplore$ and to $S$, $Visited[u] = \text{TRUE}$

Make tree $T$ with root as $u$

while ($ToExplore$ is non-empty) do

Remove node $x$ from $ToExplore$

for each edge $(x, y)$ in $\text{Adj}(x)$ do

if ($Visited[y] == \text{FALSE}$)

$Visited[y] = \text{TRUE}$

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*
Keep track of when nodes are visited.

**DFS** with Visit Times

**DFS**\((G)\)

\[
\text{for all } u \in V(G) \text{ do}
\]

Mark \( u \) as unvisited

\( T \) is set to \( \emptyset \)

\( time = 0 \)

**DFS**\((u)\)

Mark \( u \) as visited

\( pre(u) = ++time \)

**DFS**\((u)\)

**DFS**\((v)\)

Output \( T \)

Output \( \emptyset \)

**DFS**\((u)\)

**DFS**\((v)\)

Output \( T \)

Output \( \emptyset \)

Mark \( u \) as visited

**DFS**\((u)\)

**DFS**\((v)\)

Output \( T \)

Output \( \emptyset \)

Mark \( u \) as visited

**DFS**\((u)\)

**DFS**\((v)\)

Output \( T \)

Output \( \emptyset \)
An Edge in DAG

Proposition

If $G$ is a DAG and $\text{post}(u) < \text{post}(v)$, then $(u, v)$ is not in $G$. i.e., for all edges $(u, v)$ in a DAG, $\text{post}(u) > \text{post}(v)$. 
Reverse post-order is topological order

a \rightarrow b \rightarrow c

\rightarrow d \rightarrow e

\rightarrow f \rightarrow g

\rightarrow h
Reverse post-order is topological order
Sort SCCs

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 of $G^{\text{SCC}}$
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 $s \in V$

<table>
<thead>
<tr>
<th><strong>BFS</strong>($s$)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Mark all vertices as unvisited</td>
</tr>
<tr>
<td>Initialize search tree $T$ to be empty</td>
</tr>
<tr>
<td>Mark vertex $s$ as visited</td>
</tr>
<tr>
<td>set $Q$ to be the empty queue</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th><strong>enq</strong>($s$)</th>
</tr>
</thead>
<tbody>
<tr>
<td>while $Q$ is nonempty do</td>
</tr>
<tr>
<td>$u = \text{deq}(Q)$</td>
</tr>
<tr>
<td>for each vertex $v \in \text{Adj}(u)$</td>
</tr>
<tr>
<td>if $v$ is not visited then</td>
</tr>
<tr>
<td>add edge $(u, v)$ to $T$</td>
</tr>
<tr>
<td>Mark $v$ as visited and enq($v$)</td>
</tr>
</tbody>
</table>

**Proposition**

$\text{BFS}(s)$ 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 tree is the set of black edges.
BFS: An Example in Undirected Graphs

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

BFS tree is the set of black edges.
BFS: An Example in Undirected Graphs

1. [1]  4. [4,5,7,8]
2. [2,3]  5. [5,7,8]
3. [3,4,5]
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 tree is the set of black edges.
BFS: An Example in Undirected Graphs

2. [2,3]  5. [5,7,8]
3. [3,4,5]  6. [7,8,6]
BFS: An Example in Undirected Graphs

3. [3,4,5] 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]  
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

Definition

A directed graph (also called a digraph) is $G = (V, E)$, where

- $V$ is a set of vertices or nodes
- $E \subseteq V \times V$ is set of ordered pairs of vertices called edges

Viswanathan CS473ug
BFS with Distance

**BFS(s)**

Mark all vertices as unvisited; for each $v$ set $\text{dist}(v) = \infty$

Initialize search tree $T$ to be empty

Mark vertex $s$ as visited and set $\text{dist}(s) = 0$

set $Q$ to be the empty queue

$\text{enq}(s)$

while $Q$ is nonempty do

$u = \text{deq}(Q)$

for each vertex $v \in \text{Adj}(u)$ do

if $v$ is not visited do

add edge $(u, v)$ to $T$

Mark $v$ as visited, $\text{enq}(v)$

and set $\text{dist}(v) = \text{dist}(u) + 1$
Properties of BFS: Undirected Graphs

Theorem

The following properties hold upon termination of $\text{BFS}(s)$

A. The search tree contains exactly the set of vertices in the connected component of $s$.
B. If $\text{dist}(u) < \text{dist}(v)$ then $u$ is visited before $v$.
C. For every vertex $u$, $\text{dist}(u)$ is the length of a shortest path (in terms of number of edges) from $s$ to $u$.
D. If $u, v$ are in connected component of $s$ and $e = \{u, v\}$ is an edge of $G$, then $|\text{dist}(u) - \text{dist}(v)| \leq 1$. 
Properties of BFS: Directed Graphs

Theorem

The following properties hold upon termination of $\text{BFS}(s)$:

A. The search tree contains exactly the set of vertices reachable from $s$

B. If $\text{dist}(u) < \text{dist}(v)$ then $u$ is visited before $v$

C. For every vertex $u$, $\text{dist}(u)$ is 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 $\text{dist}(v) - \text{dist}(u) \leq 1$.

Not necessarily the case that $\text{dist}(u) - \text{dist}(v) \leq 1$. 
BFS with Layers

**BFS Layers**($s$):

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

![Undirected Graphs Diagram]

(UIUC) CS/ECE 374 21 March 30, 2021 21 / 45
BFS: An Example in Undirected Graphs
BFS: An Example in Undirected Graphs
BFS: An Example in Undirected Graphs

![Undirected Graph Example](image-url)
BFS: An Example in Undirected Graphs
Part III

Shortest Paths and Dijkstra’s Algorithm
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
Special case: All edge lengths are 1.
**Special case:** All edge lengths are 1.

1. Run \( \text{BFS}(s) \) 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**?
**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 \(e\)
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? 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?

**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 $s = v_0 \rightarrow v_1 \rightarrow v_2 \rightarrow \ldots \rightarrow v_k$ is a shortest path from $s$ to $v_k$, then for $1 \leq i < k$:

$$s = v_0 \rightarrow v_1 \rightarrow v_2 \rightarrow \ldots \rightarrow v_i$$

is a shortest path from $s$ to $v_i$.

Relies on non-negative edge lengths.
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 $s = v_0 \rightarrow v_1 \rightarrow v_2 \rightarrow \ldots \rightarrow v_k$ 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. $\text{dist}(s, v_i) \leq \text{dist}(s, v_k)$. Relies on non-neg edge lengths.
A proof by picture

\[ s = v_0 \]

Shortest path from \( v_0 \) to \( v_6 \)
A proof by picture

Shorter path from $v_0$ to $v_4$

$s = v_0$

Shortest path from $v_0$ to $v_6$
A proof by picture

Shortest path from $v_0$ to $v_6$. A contradiction.

Shortest path from $v_0$ to $v_6$
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 \( v \), \( \text{dist}(s,v) = \infty \)
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 \( \text{dist}(s,v) \)
\( X = X \cup \{v\} \)
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 $v$, $\text{dist}(s, v) = \infty$
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 $\text{dist}(s, v)$
    $X = X \cup \{v\}$
```

How can we implement the step in the for loop?
Finding the $i$th closest node

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

What do we know about the $i$th closest node?
Finding the $i$th closest node

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

What do we know about the $i$th closest node?

**Corollary**

*The $i$th closest node is adjacent to $X$.*
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 $X$. 
Finding 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 $X$.

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 $i$th closest node repeatedly

An example
Finding the $i$th closest node repeatedly

An example
Finding the $i$th closest node repeatedly

An example
Finding the $i$th closest node repeatedly

An example
Finding the $i$th closest node repeatedly

An example
Finding the $i$th closest node repeatedly

An example
Finding the $i$th closest node repeatedly

An example
Finding the $i$th closest node repeatedly

An example
Finding the \( i \)th closest node repeatedly

An example

![Graph](image-url)
Finding the \(i\)th closest node

1. \(X\) contains the \(i - 1\) closest nodes to \(s\)
2. Want to find the \(i\)th closest node from \(V - X\).

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 $i$th closest node

1. $X$ contains the $i-1$ closest nodes to $s$

2. Want to find the $i$th closest node from $V - X$.

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.

Let $d'(s, u)$ be the length of $P(s, u, X)$

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

1. $\text{dist}(s, u) \leq d'(s, u)$ since we are constraining the paths

2. $d'(s, u) = \min_{t \in X}(\text{dist}(s, t) + \ell(t, u))$
Finding the $i$th closest node

1. $X$ contains the $i-1$ closest nodes to $s$

2. Want to find the $i$th closest node from $V - X$.

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.

Let $d'(s, u)$ be the length of $P(s, u, X)$

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

1. $\text{dist}(s, u) \leq d'(s, u)$ since we are constraining the paths

2. $d'(s, u) = \min_{t \in X} (\text{dist}(s, t) + \ell(t, u))$

Lemma

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

**Lemma**

Given:

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

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

**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 $X$ as intermediate nodes (see previous claim). Therefore $d'(s, v) = \text{dist}(s, v)$. □
Finding the $i$th closest node

Lemma

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

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 \( i \)th closest node

**Lemma**

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

**Corollary**

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

**Proof.**

For every node \( u \in V - X \), \( \text{dist}(s, u) \leq d'(s, u) \) and for the \( i \)th closest node \( v \), \( \text{dist}(s, v) = d'(s, v) \). Moreover, \( \text{dist}(s, u) \geq \text{dist}(s, v) \) for each \( u \in V - X \).
Algorithm

Initialize for each node $v$: $\text{dist}(s, v) = \infty$

Initialize $X = \emptyset$, $d'(s, s) = 0$

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 $d'(s, v) = \min_{u \in V - X} d'(s, u)$

$\text{dist}(s, v) = d'(s, v)$

$X = X \cup \{v\}$

for each node $u$ in $V - X$ do

$$d'(s, u) = \min_{t \in X} \left( \text{dist}(s, t) + \ell(t, u) \right)$$
Initialize for each node $v$: $\text{dist}(s, v) = \infty$

Initialize $X = \emptyset$, $d'(s, s) = 0$

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 $d'(s, v) = \min_{u \in V - X} d'(s, u)$

$\text{dist}(s, v) = d'(s, v)$

$X = X \cup \{v\}$

for each node $u$ in $V - X$ do

$$d'(s, u) = \min_{t \in X} \left( \text{dist}(s, t) + \ell(t, u) \right)$$

Correctness: By induction on $i$ using previous lemmas.
Algorithm

Initialize for each node $v$: $\text{dist}(s, v) = \infty$

Initialize $X = \emptyset$, $d'(s, s) = 0$

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 $d'(s, v) = \min_{u \in V - X} d'(s, u)$

$\text{dist}(s, v) = d'(s, v)$

$X = X \cup \{v\}$

for each node $u$ in $V - X$ do

$$d'(s, u) = \min_{t \in X} \left( \text{dist}(s, t) + \ell(t, u) \right)$$

Correctness: By induction on $i$ using previous lemmas.

Running time:

$O(n \cdot (n + m))$ time.
Initialization: For each node $v$, $\text{dist}(s, v) = \infty$

Initialize $X = \emptyset$, $d'(s, s) = 0$

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 $d'(s, v) = \min_{u \in V - X} d'(s, u)$

$\text{dist}(s, v) = d'(s, v)$

$X = X \cup \{v\}$

for each node $u$ in $V - X$ do

$$d'(s, u) = \min_{t \in X} \left( \text{dist}(s, t) + \ell(t, u) \right)$$

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 $X$; $O(m + n)$ 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 $i$. 
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 $X$ in iteration $i$.

Initialize for each node $v$, $\text{dist}(s, v) = d'(s, v) = \infty$

Initialize $X = \emptyset$, $d'(s, s) = 0$

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 $d'(s, v) = \min_{u \in V - X} d'(s, u)$

$\text{dist}(s, v) = d'(s, v)$

$X = X \cup \{v\}$

Update $d'(s, u)$ for each $u$ in $V - X$ as follows:

$$d'(s, u) = \min \left( d'(s, u), \text{dist}(s, v) + \ell(v, u) \right)$$

Running time:
Improved Algorithm

Initialize for each node $v$, $\text{dist}(s, v) = d'(s, v) = \infty$

Initialize $X = \emptyset$, $d'(s, s) = 0$

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 $d'(s, v) = \min_{u \in V - X} d'(s, u)$

$\text{dist}(s, v) = d'(s, v)$

$X = X \cup \{v\}$

Update $d'(s, u)$ for each $u$ in $V - X$ as follows:

$$d'(s, u) = \min \left( d'(s, u), \text{dist}(s, v) + \ell(v, u) \right)$$

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 \( \text{dist}(s, u) \) maintain it
2. update \( \text{dist} \) values after adding \( v \) by scanning edges out of \( v \)

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

Priority Queues to maintain \( \text{dist} \) values for faster running time
Dijkstra’s Algorithm

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

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

Priority Queues to maintain $\text{dist}$ values for faster running time

3. 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 $S$.

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

\[ Q \leftarrow \text{makePQ}() \]
\[ \text{insert}(Q, (s, 0)) \]
\[ \text{for each node } u \neq s \text{ do} \]
\[ \quad \text{insert}(Q, (u, \infty)) \]
\[ X \leftarrow \emptyset \]
\[ \text{for } i = 1 \text{ to } |V| \text{ do} \]
\[ \quad (v, \text{dist}(s, v)) = \text{extractMin}(Q) \]
\[ X = X \cup \{v\} \]
\[ \text{for each } u \text{ in } \text{Adj}(v) \text{ do} \]
\[ \quad \text{decreaseKey}(Q, (u, \min(\text{dist}(s, u), \text{dist}(s, v) + \ell(v, u)))) \]

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
### 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.
Fibonacci Heaps

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

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

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

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: \( \ell \) 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!)
Dijkstra’s algorithm finds the shortest path distances from s to $V$.

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

**Question:** How do we find the paths themselves?

```plaintext
Q = makePQ()
insert(Q, (s, 0))
prev(s) ← null

for each node $u \neq s$ do
    insert(Q, (u, ∞))
    prev(u) ← null

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

(UIUC) CS/ECE 374 43 / 45
March 30, 2021 43
Lemma

The edge set \((u, \text{prev}(u))\) 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 \(\{(u, \text{prev}(u)) \mid u \in V\}\) 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 \(V\).
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 $G^{\text{rev}}$!