Pre-lecture brain teaser

Given a directed graph (G), propose an algorithm that finds a vertex that is contained within the source SCC of the meta-graph of G.
ECE-374-B: Lecture 17 - Shortest Paths [BFS, Djikstra]

Instructor: Nickvash Kani

March 23, 2022

University of Illinois at Urbana-Champaign
Given a directed graph \((G)\), propose an algorithm that finds a vertex that is contained within the source SCC of the meta-graph of \(G\).
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...

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

**BFS Algorithm**

- **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
  - **enqueue**$(Q, s)$
  - while $Q$ is nonempty do
    - $u = \text{dequeue}(Q)$
    - for each vertex $v \in \text{Adj}(u)$
      - if $v$ is not visited then
        - add edge $(u, v)$ to $T$
        - Mark $v$ as visited and **enqueue**$(v)$

**Proposition**

$\text{BFS}(s)$ runs in $O(n + m)$ time.
T1. [1]
BFS: An Example in Undirected Graphs

T1. [1]
T2. [2,3]
T1. [1]
T2. [2,3]
BFS: An Example in Undirected Graphs

T1. [1]
T2. [2,3]
T3. [3,4,5]
BFS: An Example in Undirected Graphs

T1. [1]  T4. [4,5,7,8]
T2. [2,3]  T5. [5,7,8]
BFS: An Example in Undirected Graphs

T1. [1]  
T2. [2,3]  
T3. [3,4,5]  
T4. [4,5,7,8]  
T5. [5,7,8]  
T6. [7,8,6]

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

T1. [1]
T2. [2,3]
T3. [3,4,5]
T4. [4,5,7,8]
T5. [5,7,8]
T6. [7,8,6]
BFS: An Example in Undirected Graphs

T1. [1]  
T2. [2,3]  
T3. [3,4,5]

T4. [4,5,7,8]  
T5. [5,7,8]  
T6. [7,8,6]  
T7. [8,6]
BFS: An Example in Undirected Graphs

T1. [1]
T2. [2,3]
T3. [3,4,5]
T4. [4,5,7,8]
T5. [5,7,8]
T6. [7,8,6]
T7. [8,6]
T8. [6]
BFS: An Example in Undirected Graphs

T1. [1]
T2. [2,3]
T3. [3,4,5]
T4. [4,5,7,8]
T5. [5,7,8]
T6. [7,8,6]
T7. [8,6]
T8. [6]
T9. []

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

BFS tree is the set of purple edges.

T1. [1]
T2. [2,3]
T3. [3,4,5]
T4. [4,5,7,8]
T5. [5,7,8]
T6. [7,8,6]
T7. [8,6]
T8. [6]
T9. []
BFS: An Example in Undirected Graphs

T1. [1]  
T2. [2,3]  
T3. [3,4,5]  
T4. [4,5,7,8]  
T5. [5,7,8]  
T6. [7,8,6]  
T7. [8,6]  
T8. [6]  
T9. []

**BFS** tree is the set of purple edges.
BFS: An Example in Directed Graphs
BFS: An Example in Directed Graphs

T1. [A]
BFS: An Example in Directed Graphs

T1. [A]
T2. [B,C,F]
BFS: An Example in Directed Graphs

T1. [A]
T2. [B, C, F]
BFS: An Example in Directed Graphs

T1. [A]
T2. [B,C,F]
T3. [C,F,E]
BFS: An Example in Directed Graphs

T1. [A]  
T2. [B,C,F]  
T3. [C,F,E]  
T4. [F,E,D]
BFS: An Example in Directed Graphs

T1. [A]
T2. [B,C,F]
T3. [C,F,E]
T4. [F,E,D]
T5. [E,D,G]
BFS: An Example in Directed Graphs

T1. [A]
T2. [B,C,F]
T3. [C,F,E]
T4. [F,E,D]
T5. [E,D,G]
T6. [D,G,H]
BFS: An Example in Directed Graphs

T1. [A]
T2. [B,C,F]
T3. [C,F,E]
T4. [F,E,D]
T5. [E,D,G]
T6. [D,G,H]
T7. [G,H]
BFS: An Example in Directed Graphs

T1. [A]  
T2. [B,C,F]  
T3. [C,F,E]  
T4. [F,E,D]  
T5. [E,D,G]  
T6. [D,G,H]  
T7. [G,H]  
T8. [H]
BFS: An Example in Directed Graphs

T1. [A]
T2. [B,C,F]
T3. [C,F,E]
T4. [F,E,D]
T5. [E,D,G]
T6. [D,G,H]
T7. [G,H]
T8. [H]
T9. []
BFS with distances and layers
**BFS with distances**

**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

enqueue(s)
while \( Q \) is nonempty do
  \( u = \text{dequeue}(Q) \)
  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, enqueue(v)
      and set \( \text{dist}(v) = \text{dist}(u) + 1 \)
The following properties hold upon termination of BFS(s):

(A) Search tree contains exactly the set of vertices in the connected component of s.

(B) If dist(u) < dist(v) then u is visited before v.

(C) For every vertex u, dist(u) is 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 |dist(u) − dist(v)| ≤ 1.
Properties of **BFS**: Directed Graphs

**Theorem**
The following properties hold upon termination of **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 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 $\text{dist}(v) - \text{dist}(u) \leq 1$. *Not necessarily the case that $\text{dist}(u) - \text{dist}(v) \leq 1$.**
BFS with Layers

**BFS Layers**

1. **Mark all vertices as unvisited and initialize $T$ to be empty**
2. **Mark $s$ as visited and set $L_0 = \{s\}**
3. **Initialize $i = 0$**
4. **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

**BFS\text{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:** $O(n + m)$
Layer 0: 1
Layer 1: 2, 3
Layer 2: 4, 5, 7, 8
Layer 3: 6
BFS with Layers: Properties

Proposition
The following properties hold on termination of $\text{BFSLayers}(s)$.

- $\text{BFSLayers}(s)$ 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.
Layer 0: A
Layer 1: B, F, C
Layer 2: E, G, D
Layer 3: H
Proposition
The following properties hold on termination of BFS_Layers(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
Shortest Paths and Dijkstra’s Algorithm
Problem definition
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!
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 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.
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. Relies on non-negativity!
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. Relies on non-negativity!
Shortest path in the weighted case using BFS
• Special case: All edge lengths are 1.
• **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.
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**?
• **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 \(e\).
Example of edge refinement
Example of edge refinement
Example of edge refinement
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.
On the hereditary nature of shortest paths
You can not shortcut a shortest path

**Lemma**

$G$: directed graph with non-negative edge lengths.

$\text{dist}(s, v)$: shortest path length from $s$ to $v$.

If $s = v_0 \rightarrow v_1 \rightarrow v_2 \rightarrow \ldots \rightarrow v_k$ shortest path from $s$ to $v_k$ then for any $0 \leq i < j \leq k$:

$v_i \rightarrow v_{i+1} \rightarrow \ldots \rightarrow v_j$ is shortest path from $v_i$ to $v_j$
A proof by picture

$s = v_0$

Shortest path from $v_0$ to $v_{10}$
A proof by picture

Shorter path from $v_2$ to $v_8$

Shortest path from $v_0$ to $v_{10}$
A proof by picture

A shorter path from $v_0$ to $v_{10}$. A contradiction.

Shortest path from $v_0$ to $v_{10}$.
Corollary

$G$: directed graph with non-negative edge lengths.

dist$(s, v)$: shortest path length from $s$ to $v$.

If $s = v_0 \rightarrow v_1 \rightarrow v_2 \rightarrow \ldots \rightarrow v_k$ shortest path from $s$ to $v_k$ then for any $0 \leq i \leq k$:

- $s = v_0 \rightarrow v_1 \rightarrow v_2 \rightarrow \ldots \rightarrow v_i$ is shortest path from $s$ to $v_i$
- dist$(s, v_i) \leq$ dist$(s, v_k)$. Relies on non-neg edge lengths.
The basic algorithm: Find the $i^{th}$ closest vertex
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 $v$, $\text{dist}(s, v) = \infty$
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 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

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

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

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

**Claim**

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

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

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

**Claim**

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

**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
Finding the $i^{th}$ closest node repeatedly
Finding the $i^{th}$ closest node repeatedly
Finding the $i^{th}$ closest node repeatedly
Finding the $i^{th}$ closest node repeatedly
Finding the $i^{th}$ closest node repeatedly
Finding the $i^{th}$ closest node repeatedly
Finding the $i^{th}$ closest node repeatedly
Finding the $i^{th}$ closest node repeatedly
Finding the $i^{th}$ closest node

Corollary
The $i^{th}$ closest node is adjacent to $X$. 
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

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

Initialize for each node \( v \): \( \text{dist}(s,v) = \infty \)
Initialize \( X = \emptyset \), \( \text{d}'(s,s) = 0 \)
for \( i = 1 \) to \( |V| \) do

(* Invariant: \( X \) contains the \( i-1 \) closest nodes to \( s \) *)
(* Invariant: \( \text{d}'(s,u) \) is shortest path distance from \( u \) to \( s \)
   using only \( X \) as intermediate nodes*)

Let \( v \) be such that \( \text{d}'(s,v) = \min_{u \in V - X} \text{d}'(s,u) \)
\( \text{dist}(s,v) = \text{d}'(s,v) \)
\( X = X \cup \{v\} \)
for each node \( u \) in \( V - X \) do

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

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

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.
Dijkstra’s algorithm
Example: Dijkstra algorithm in action
Example: Dijkstra algorithm in action
Example: Dijkstra algorithm in action
Example: Dijkstra algorithm in action
Example: Dijkstra algorithm in action
Example: Dijkstra algorithm in action
Example: Dijkstra algorithm in action
Example: Dijkstra algorithm in action
Example: Dijkstra algorithm in action
Example: Dijkstra algorithm in action
Example: Dijkstra algorithm in action
Example: Dijkstra algorithm in action
Example: Dijkstra algorithm in action
Example: Dijkstra algorithm in action
Example: Dijkstra algorithm in action
Example: Dijkstra algorithm in action
Example: Dijkstra algorithm in action
Example: Dijkstra algorithm in action
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 $X$ in iteration $i$. 

\[
\begin{align*}
\text{Initialize for each node } v, & \quad \text{dist}(s, v) = d'(s, v) = \infty \\
\text{Initialize } X = \emptyset, & \quad d'(s, s) = 0 \\
\text{for } i = 1 \text{ to } |V| \text{ do} & \\
& \quad \text{// $X$ contains the } i-1 \text{ closest nodes to } s, \text{ and the values of } d'(s, u) \text{ are current} \\
& \quad \text{Let } v \text{ be node realizing } d'(s, v) = \min_{u \in V - X} d'(s, u) \\
& \quad \text{dist}(s, v) = d'(s, v) \\
& \quad X = X \cup \{v\} \\
& \quad \text{Update } d'(s, u) \text{ for each } u \text{ in } V - X \text{ as follows:} \\
& \quad d'(s, u) = \min(d'(s, u), \text{dist}(s, v) + \ell(v, u)) \\
\end{align*}
\]

Running time: $O(m + n^2)$ time.

- $n$ outer iterations and in each iteration following steps
- 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
- Finding $v$ from $d'(s, u)$ values is $O(n)$ time
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 $X$ 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:
$O(m + n^2)$ 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.

- $n$ outer iterations and in each iteration following steps
- updating $d'(s,u)$ after $v$ is added takes $O(\text{deg}(v))$ time so total work is $O(m)$ since a node enters $X$ only once
- Finding $v$ from $d'(s,u)$ values is $O(n)$ time
Dijkstra’s Algorithm

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

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

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

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

\[
\begin{align*}
\text{Initialize for each node } v, & \quad \text{dist}(s, v) = \infty \\
\text{Initialize } X = \emptyset, & \quad \text{dist}(s, s) = 0 \\
\text{for } i = 1 \text{ to } |V| \text{ do} & \\
\text{Let } v \text{ be such that } \text{dist}(s, v) = \min_{u \in V - X} \text{dist}(s, u) \\
X = X \cup \{v\} & \\
\text{for each } u \text{ in Adj}(v) \text{ do} & \\
\text{dist}(s, u) = \min\left(\text{dist}(s, u), \text{dist}(s, v) + \ell(v, u)\right)
\end{align*}
\]

**Priority Queues** to maintain \(\text{dist}\) values for faster running time
- Using heaps and standard priority queues: \(O((m + n) \log n)\)
- Using Fibonacci heaps: \(O(m + n \log n)\).
Dijkstra using priority queues
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 $insert$. 
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.
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

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

Priority Queue operations:

\begin{itemize}
  \item $O(n)$ \textbf{insert} operations
  \item $O(n)$ \textbf{extractMin} operations
  \item $O(m)$ \textbf{decreaseKey} operations
\end{itemize}
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
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

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

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

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

- **extractMin, insert, delete, meld** in $O(\log n)$ time
- **decreaseKey** in $O(1)$ amortized time: $\ell$ 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, ....
- Boost library implements both Fibonacci heaps and rank-pairing heaps.
Shortest path trees and variants
Dijkstra’s alg. 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 ≠ s do 
  insert(Q, (u, ∞)) 
  prev(u) ← null 
X = ∅ 
for i = 1 to |V| do 
  (v, dist(s, v)) = extractMin(Q) 
  X = X ∪ {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
```
Dijkstra’s alg. 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 ≠ s do
    insert(Q, (u, ∞))
    prev(u) ← null

X = ∅
for i = 1 to |V| do
    (v, dist(s, v)) = extractMin(Q)
    X = X ∪ {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
```
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.

- The edge set \(\{(u, \text{prev}(u)) \mid u \in V\}\) induces a directed in-tree rooted at \(s\) (Why?)

- Use induction on \(|X|\) to argue that the tree is a shortest path tree for nodes in \(V\).
Dijkstra’s alg. gives shortest paths from $s$ to all nodes in $V$.

How do we find shortest paths from all of $V$ to $s$?
Dijkstra’s alg. 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 $G^{rev}$!