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...

1. DFS good for exploring graph structure
2. BFS good for exploring distances
xkcd take on DFS

**PREPARING FOR A DATE:**

What situations might I prepare for?
1) Medical emergency
2) Dancing
3) Food too expensive

**OKAY, WHAT KINDS OF EMERGENCIES CAN HAPPEN?**

- a) Snakebite
- b) Lightning strike
- c) Fall from chair
- d) Diagonal dancing

**HMM, WHICH SNakes ARE DANGEROUS? LET'S SEE...**

- a) J) Corn snake
- b) Garter snake
- c) Copperhead

**THE RESEARCH COMPARING SNAKE VENOMS IS SCATTERED AND INCONSISTENT. I'LL MAKE A SPREADSHEET TO ORGANIZE IT.**

**I'M HERE TO PICK YOU UP. YOU'RE NOT DRESSED?**

**BY LOR, THE INLAND TAIPAN HAS THE DEADIEST VENOM OF ANY SNAKE!**

I really need to stop using depth-first searches.
Distances in Graphs

Given a graph $G = (V, E)$ and two nodes $s, t$ the distance $\text{dist}(s, t)$ is the length of the shortest path from $s$ to $t$ in $G$. 

Triangle inequality: $\text{dist}(u, v) + \text{dist}(v, w) \geq \text{dist}(u, w)$ for all $u, v, w \in V$. 

Distances in Graphs

Given a graph $G = (V, E)$ and two nodes $s, t$ the distance $\text{dist}(s, t)$ is the length of the shortest path from $s$ to $t$ in $G$

- $\text{dist}(s, t) = \text{dist}(t, s)$ in undirected graphs while $\text{dist}(s, t)$ and $\text{dist}(t, s)$ may be different in directed graphs

Triangle inequality: $\text{dist}(u, v) + \text{dist}(v, w) \geq \text{dist}(u, w)$ for all $u, v, w \in V$
Distances in Graphs

Given a graph $G = (V, E)$ and two nodes $s, t$ the distance $\text{dist}(s, t)$ is the length of the shortest path from $s$ to $t$ in $G$

- $\text{dist}(s, t) = \text{dist}(t, s)$ in undirected graphs while $\text{dist}(s, t)$ and $\text{dist}(t, s)$ may be different in directed graphs
- Triangle inequality: $\text{dist}(u, v) + \text{dist}(v, w) \geq \text{dist}(u, w)$ for all $u, v, w \in V$
## Shortest Path Problems

**Input** A (undirected or directed) graph $G = (V, E)$

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!
Shortest Path Problems

Input A (undirected or directed) graph $G = (V, E)$

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!

These are *unweighted* problems. More general problem when edges have lengths which can potentially be negative! We will see them soon.
Single-Source Shortest Paths

Single-Source Shortest Path Problems

Input
A (undirected or directed) graph $G = (V, E)$

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.
Single-Source Shortest Paths

**Input** A (undirected or directed) graph \( G = (V, E) \)

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.

**Notation:** If \( s \) is clear from context we may use \( \text{dist}(u) \) as short hand for \( \text{dist}(s, u) \).
Single-Source Shortest Path Problems

**Input**  A (undirected or directed) graph  \( G = (V, E) \)

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.

**Notation:** If  \( s \) is clear from context we may use  \( \text{dist}(u) \) as shorthand for  \( \text{dist}(s, u) \).

- **BFS** solves single-source shortest path problems in unweighted graphs (both undirected and directed) in  \( O(n + m) \) time.
- **BFS** is obtained from Basic Search by using a Queue data structure.
## Queue Data Structure

### Queues

A **queue** is a list of elements which supports the operations:

1. **enqueue**: Adds an element to the end of the list
2. **dequeue**: Removes an element from the front of the list

Elements are extracted in **first-in first-out (FIFO)** order, i.e., elements are picked in the order in which they were inserted.
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
- \( \text{enq}(s) \)

**while** \( Q \) is nonempty **do**
- \( u = \text{deq}(Q) \)
- **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 \( \text{enq}(v) \)

**Proposition**

**BFS\( (s) \) runs in** \( O(n + m) \) **time.**
BFS: An Example in Undirected Graphs

1. [1]

[Diagram of an undirected graph with nodes labeled 1 to 8 and edges connecting them. The BFS tree is indicated with black edges.]
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: An Example in Undirected Graphs

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

1. \([1]\)
2. \([2,3]\)
3. \([3,4,5]\)
4. \([4,5,7,8]\)
5. \([5,7,8]\)

\[\text{BFS tree is the set of black edges.}\]
BFS: An Example in Undirected Graphs

1. [1]
2. [2,3]
3. [3,4,5]
4. [4,5,7,8]
5. [5,7,8]
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. \([8,6]\)

BFS tree is the set of black edges.
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]
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

![Directed Graph Example](image-url)

**Definition**

A directed graph (also called a digraph) is denoted as $G = (V, E)$, where $V$ is a set of vertices or nodes and $E \subseteq V \times V$ is a set of ordered pairs of vertices called edges.
**BFS with Distance**

**BFS**

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

**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 $\text{BFS}(s)$

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>A</strong></td>
<td>The search tree contains exactly the set of vertices in the connected component of $s$.</td>
</tr>
<tr>
<td><strong>B</strong></td>
<td>If $\text{dist}(u) &lt; \text{dist}(v)$ then $u$ is visited before $v$.</td>
</tr>
<tr>
<td><strong>C</strong></td>
<td>For every vertex $u$, $\text{dist}(u)$ is the length of a shortest path (in terms of number of edges) from $s$ to $u$.</td>
</tr>
<tr>
<td><strong>D</strong></td>
<td>If $u, v$ are in connected component of $s$ and $e = {u, v}$ is an edge of $G$, then $</td>
</tr>
</tbody>
</table>
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 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 is a simple algorithm but proving its properties formally is not straightforward.

BFS explores graph in increasing order of distance from source $s$.

There is a simpler variant that makes BFS exploration transparent and easier to understand.

Given $G$ and $s \in V$, define $L_i = \{ v | \text{dist}(s, v) = i \}$. The "layer" of all vertices at exactly distance $i$ from $s$ is $L_0 = \{ s \}$, and can be found from $L_0, L_2, \ldots, L_{i-1}$ inductively and easily.
BFS with Layers

**BFS** is a simple algorithm but proving its properties formally is not straightforward.

**BFS** explores graph in increasing order of distance from source $s$.

There is a simpler variant that makes **BFS** exploration transparent and easier to understand.

Given $G$ and $s \in V$ define $L_i = \{v \mid \text{dist}(s, v) = i\}$. The “layer” of all vertices at exactly distance $i$ from $s$.
**BFS with Layers**

**BFS** is a simple algorithm but proving its properties formally is not straightforward.

**BFS** explores graph in increasing order of distance from source $s$.

There is a simpler variant that makes **BFS** exploration transparent and easier to understand.

- Given $G$ and $s \in V$ define $L_i = \{v \mid \text{dist}(s, v) = i\}$. The “layer” of all vertices at exactly distance $i$ from $s$.
- $L_0 = \{s\}$
BFS with Layers

BFS is a simple algorithm but proving its properties formally is not straightforward.

BFS explores graph in increasing order of distance from source \( s \).

There is a simpler variant that makes BFS exploration transparent and easier to understand.

- Given \( G \) and \( s \in V \) define \( L_i = \{ v \mid \text{dist}(s, v) = i \} \). The “layer” of all vertices at exactly distance \( i \) from \( s \).
- \( L_0 = \{ s \} \)
- Can find \( L_i \) from \( L_0, L_2, \ldots, L_{i-1} \) inductively and easily.
**BFS with Layers**

**BFS\textunderscore 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 \(L_{i+1}\)
  \(i = i + 1\)
**BFS with Layers**

\[ \text{BFS Layers}(s): \]

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 \( L_{i+1} \)

\( i = i + 1 \)

\[ \text{Running time: } O(n + m) \]
**BFS with Layers: Properties**

**Proposition**

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

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

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 Layers: Properties

For directed graphs

**Proposition**

The following properties hold on termination of \textbf{BFSLayers}(s), if \( G \) is directed.

For each edge \( e = (u, v) \) is one of four types:

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

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.
Shortest Walk Problems

Given a graph $G = (V, E)$:

1. **A path** is a sequence of distinct vertices $v_1, v_2, \ldots, v_k$ such that $(v_i, v_{i+1}) \in E$ for $1 \leq i \leq k - 1$.

2. **A walk** is a sequence of vertices $v_1, v_2, \ldots, v_k$ such that $(v_i, v_{i+1}) \in E$ for $1 \leq i \leq k - 1$. Vertices are allowed to repeat.

Finding walks is often easier and more natural than finding paths. Why? Concatenating two walks gives a walk while concatenating two paths may give a walk, and not necessarily a path.

When edges have non-negative lengths, finding a shortest $s$-$t$ walk is the same as finding a shortest $s$-$t$ path. Why? In more general settings walks are easier to work with.
Given a graph \( G = (V, E) \):

1. A path is a sequence of distinct vertices \( v_1, v_2, \ldots, v_k \) such that \((v_i, v_{i+1}) \in E\) for \(1 \leq i \leq k - 1\).

2. A walk is a sequence of vertices \( v_1, v_2, \ldots, v_k \) such that \((v_i, v_{i+1}) \in E\) for \(1 \leq i \leq k - 1\). Vertices are allowed to repeat.

Finding walks is often easier and more natural than finding paths. Why?
Shortest Walk Problems

Given a graph \( G = (V, E) \):

1. A path is a sequence of distinct vertices \( v_1, v_2, \ldots, v_k \) such that \( (v_i, v_{i+1}) \in E \) for \( 1 \leq i \leq k - 1 \).

2. A walk is a sequence of vertices \( v_1, v_2, \ldots, v_k \) such that \( (v_i, v_{i+1}) \in E \) for \( 1 \leq i \leq k - 1 \). Vertices are allowed to repeat.

Finding walks is often easier and more natural than finding paths. Why? Concatenating two walks gives a walk while concatenating two paths may give a walk, and not necessarily a path.
Shortest Walk Problems

Given a graph $G = (V, E)$:

1. A **path** is a sequence of distinct vertices $v_1, v_2, \ldots, v_k$ such that $(v_i, v_{i+1}) \in E$ for $1 \leq i \leq k - 1$.

2. A **walk** is a sequence of vertices $v_1, v_2, \ldots, v_k$ such that $(v_i, v_{i+1}) \in E$ for $1 \leq i \leq k - 1$. Vertices are allowed to repeat.

Finding walks is often easier and more natural than finding paths. Why? Concatenating two walks gives a walk while concatenating two paths may give a walk, and not necessarily a path.

When edges have non-negative lengths, finding a shortest $s$-$t$ walk is the same as finding a shortest $s$-$t$ path. Why?
Shortest Walk Problems

Given a graph $G = (V, E)$:

1. A path is a sequence of distinct vertices $v_1, v_2, \ldots, v_k$ such that $(v_i, v_{i+1}) \in E$ for $1 \leq i \leq k - 1$.

2. A walk is a sequence of vertices $v_1, v_2, \ldots, v_k$ such that $(v_i, v_{i+1}) \in E$ for $1 \leq i \leq k - 1$. Vertices are allowed to repeat.

Finding walks is often easier and more natural than finding paths. Why? Concatenating two walks gives a walk while concatenating two paths may give a walk, and not necessarily a path.

When edges have non-negative lengths, finding a shortest $s$-$t$ walk is the same as finding a shortest $s$-$t$ path. Why?

In more general settings walks are easier to work with.
**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 - 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 $\ell(u, v) = \ell(v, u) = \ell({u, v})$.
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.

Restrict attention to directed graphs

Undirected graph problem can be reduced to directed graph problem - how?

1. Given undirected graph $G$, create a new directed graph $G'$ by replacing each edge $\{u, v\}$ in $G$ by $(u, v)$ and $(v, u)$ in $G'$.

2. set $\ell(u, v) = \ell(v, u) = \ell(\{u, v\})$

Exercise: show reduction works. Relies on non-negativity!
Special case: All edge lengths are 1.
**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.
**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**?
**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*
**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. If \( s = v_0 \rightarrow v_1 \rightarrow v_2 \rightarrow \ldots \rightarrow v_i \) is a shortest path from \( s \) to \( v_i \) then for \( 1 \leq j < i \):

\[ \text{dist}(s, v_j) \leq \text{dist}(s, v_i). \]

Relies on non-neg edge lengths.

Proof.
Suppose not. Then for some \( j < i \) there is a path \( P' \) from \( s \) to \( v_j \) of length strictly less than that of \( s = v_0 \rightarrow v_1 \rightarrow \ldots \rightarrow v_i \). Then \( P' \) concatenated with \( v_j \rightarrow v_{j+1} \rightarrow \ldots \rightarrow v_i \) contains a strictly shorter path to \( v_i \) than \( s = v_0 \rightarrow v_1 \rightarrow \ldots \rightarrow v_i \). For the second part, observe that edge lengths are non-negative.
Towards an algorithm

Why does **BFS** work?

**BFS** (s) explores nodes in increasing distance from s
Towards an algorithm

Why does \textbf{BFS} work?

\textbf{BFS}(s) explores nodes in increasing distance from \( s \).

\textbf{Lemma}

\textit{Let} \( G \) \textit{be a directed graph with non-negative edge lengths. If} \( s = v_0 \rightarrow v_1 \rightarrow v_2 \rightarrow \ldots \rightarrow v_i \) \textit{is a shortest path from} \( s \) \textit{to} \( v_i \) \textit{then for} \( 1 \leq j < i \):

1. \( s = v_0 \rightarrow v_1 \rightarrow \ldots \rightarrow v_j \) \textit{is a shortest path from} \( s \) \textit{to} \( v_j \)
2. \( \text{dist}(s, v_j) \leq \text{dist}(s, v_i) \). \textit{Relies on non-neg edge lengths.}
Towards an algorithm

Lemma

Let $G$ be a directed graph with non-negative edge lengths. If $s = v_0 \rightarrow v_1 \rightarrow v_2 \rightarrow \ldots \rightarrow v_i$ is a shortest path from $s$ to $v_i$ then for $1 \leq j < i$:

1. $s = v_0 \rightarrow v_1 \rightarrow \ldots \rightarrow v_j$ is a shortest path from $s$ to $v_j$
2. $\text{dist}(s, v_j) \leq \text{dist}(s, v_i)$. Relies on non-neg edge lengths.

Proof.

Suppose not. Then for some $j < i$ there is a path $P'$ from $s$ to $v_j$ of length strictly less than that of $s = v_0 \rightarrow v_1 \rightarrow \ldots \rightarrow v_j$. Then $P'$ concatenated with $v_j \rightarrow v_{j+1} \ldots \rightarrow v_i$ contains a strictly shorter path to $v_i$ than $s = v_0 \rightarrow v_1 \ldots \rightarrow v_i$. For the second part, observe that edge lengths are non-negative.
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

A shorter 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 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 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
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?

**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

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?

**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

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

1. 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 $V - X$.

1. 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 $P(s, u, X)$

Claim

For each $u \in V - X$, $d'(s, u) = \min_{t \in X}(\text{dist}(s, t) + \ell(t, u))$.
Understanding $d'(s, u)$ values
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 $V - X$.

1. 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 $P(s, u, X)$.

3. Can compute all $d'(s, u)$ values.

Main claim:

**Lemma**

The $i$th closest node to $s$ is the node $v \in V - X$ with the smallest $d'$ value, that is, $d'(s, v) = \min_{u \in V - X} d'(s, u)$.
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 $V - X$.
3. 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.
4. Let $d'(s, u)$ be the length of $P(s, u, X)$
5. Can compute all $d'(s, u)$ values

Main claim:

**Lemma**

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

Assuming claim, inductive algorithm follows.
Finding the \(i\)th closest node: proof

Auxiliary lemma:

**Lemma**

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 \(d'(s, v) = \text{dist}(s, v)\). \(\square\)
Finding the $i$th closest node: proof

**Lemma**

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

**Lemma**

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

**Proof.**

Assume distances are unique for simplicity. Let $v^* \in V - X$ be the $i'$th closest node to $s$. Implies for every other $u \in V - X$, $d'(s, u) \geq \text{dist}(s, u) > \text{dist}(s, v^*)$. But Lemma says $d'(s, v^*) = \text{dist}(s, v^*)$. Hence node $v$ that minimizes $d'(s, v)$ value must be $v^*$. 

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
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

$\hspace{1cm} d'(s, u) = \min_{t \in X} \left( \text{dist}(s, t) + \ell(t, u) \right)$
Algorithm

Initialize for each node $v$: $\text{dist}(s, v) = \infty$
Initialize $X = \emptyset$, $d'(s, s) = 0$

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:
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.

$n$ 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 $i$. 

$\text{Running time: } O(m + n^2)$ time.
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:

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

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.

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

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

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

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

Chandra (UIUC) CS/ECE 374 39 Spring 2021 39 / 48
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. \( O(n) \) \text{ insert} operations
2. \( O(n) \) \text{ extractMin} operations
3. \( O(m) \) \text{ 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

1. All operations can be done in $O(\log n)$ time

Dijkstra’s algorithm can be implemented in $O((n + m) \log n)$ time.
Priority Queues: Fibonacci Heaps/Relaxed Heaps

Fibonacci Heaps

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

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?

\[
\begin{align*}
Q &= \text{makePQ}() \\
\text{insert}(Q, (s, 0)) \\
\text{prev}(s) &\leftarrow \text{null} \\
\text{for each node } u &\neq s \text{ do} \\
&\hspace{1cm} \text{insert}(Q, (u, \infty)) \\
&\hspace{1cm} \text{prev}(u) &\leftarrow \text{null}
\end{align*}
\]

\[
X = \emptyset \\
\text{for } i = 1 \text{ to } |V| \text{ do} \\
&\hspace{1cm} (v, \text{dist}(s, v)) = \text{extractMin}(Q) \\
&\hspace{1cm} X = X \cup \{v\} \\
&\hspace{1cm} \text{for each } u \text{ in Adj}(v) \text{ do} \\
&\hspace{2cm} \text{if } (\text{dist}(s, v) + \ell(v, u) < \text{dist}(s, u)) \text{ then} \\
&\hspace{3cm} \text{decreaseKey}(Q, (u, \text{dist}(s, v) + \ell(v, u))) \\
&\hspace{3cm} \text{prev}(u) = v
\]
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 $G^{rev}$!
Shortest paths between sets of nodes

Suppose we are given $S \subset V$ and $T \subset V$. Want to find shortest path from $S$ to $T$ defined as:

$$\text{dist}(S, T) = \min_{s \in S, t \in T} \text{dist}(s, t)$$

How do we find $\text{dist}(S, T)$?
Example Problem

You want to go from your house to a friend’s house. Need to pick up some dessert along the way and hence need to stop at one of the many potential stores along the way. How do you calculate the “shortest” trip if you include this stop?
Example Problem

You want to go from your house to a friend’s house. Need to pick up some dessert along the way and hence need to stop at one of the many potential stores along the way. How do you calculate the “shortest” trip if you include this stop?

Given \( G = (V, E) \) and edge lengths \( \ell(e), e \in E \). Want to go from \( s \) to \( t \). A subset \( X \subset V \) that corresponds to stores. Want to find \( \min_{x \in X} d(s, x) + d(x, t) \).
Example Problem

You want to go from your house to a friend’s house. Need to pick up some dessert along the way and hence need to stop at one of the many potential stores along the way. How do you calculate the “shortest” trip if you include this stop?

Given $G = (V, E)$ and edge lengths $\ell(e), e \in E$. Want to go from $s$ to $t$. A subset $X \subset V$ that corresponds to stores. Want to find

$$\min_{x \in X} d(s, x) + d(x, t).$$

**Basic solution:** Compute for each $x \in X$, $d(s, x)$ and $d(x, t)$ and take minimum. $2|X|$ shortest path computations. $O(|X|(m + n \log n))$. 
Example Problem

You want to go from your house to a friend’s house. Need to pick up some dessert along the way and hence need to stop at one of the many potential stores along the way. How do you calculate the “shortest” trip if you include this stop?

Given $G = (V, E)$ and edge lengths $\ell(e), e \in E$. Want to go from $s$ to $t$. A subset $X \subset V$ that corresponds to stores. Want to find
\[
\min_{x \in X} d(s, x) + d(x, t).
\]

**Basic solution:** Compute for each $x \in X$, $d(s, x)$ and $d(x, t)$ and take minimum. $2|X|$ shortest path computations. $O(|X|(m + n \log n))$.

**Better solution:** Compute shortest path distances from $s$ to every node $v \in V$ with one Dijkstra. Compute from every node $v \in V$ shortest path distance to $t$ with one Dijkstra.