Review of Priority Queues and Graph Searches

T. M. Murali

August 27, 29 2018
Motivation: Sort a List of Numbers

Sort

INSTANCE: Nonempty list $x_1, x_2, \ldots, x_n$ of integers.

SOLUTION: A permutation $y_1, y_2, \ldots, y_n$ of $x_1, x_2, \ldots, x_n$ such that $y_i \leq y_{i+1}$, for all $1 \leq i < n$. 
Motivation: Sort a List of Numbers

Sort

**INSTANCE:** Nonempty list $x_1, x_2, \ldots, x_n$ of integers.

**SOLUTION:** A permutation $y_1, y_2, \ldots, y_n$ of $x_1, x_2, \ldots, x_n$ such that $y_i \leq y_{i+1}$, for all $1 \leq i < n$.

Possible algorithm:
- Store all the numbers in a data structure $D$.
- Repeatedly find the smallest number in $D$, output it, and remove it.
Sort

**INSTANCE:** Nonempty list \( x_1, x_2, \ldots, x_n \) of integers.

**SOLUTION:** A permutation \( y_1, y_2, \ldots, y_n \) of \( x_1, x_2, \ldots, x_n \) such that 
\[ y_i \leq y_{i+1}, \text{ for all } 1 \leq i < n. \]

- Possible algorithm:
  - Store all the numbers in a data structure \( D \).
  - Repeatedly find the smallest number in \( D \), output it, and remove it.

- To get \( O(n \log n) \) running time, each “find minimum” step and each “remove” step must take \( O(\log n) \) time.
Candidate Data Structures for Sorting

- Possible algorithm:
  - Store all the numbers in a data structure $D$.
  - Repeatedly find the smallest number in $D$, output it, and remove it.
- Data structure must support three operations:
Candidate Data Structures for Sorting

- Possible algorithm:
  - Store all the numbers in a data structure $D$.
  - Repeatedly find the smallest number in $D$, output it, and remove it.
- Data structure must support three operations: insertion of a number, finding minimum, and deleting minimum in $O(\log n)$ time.
Candidate Data Structures for Sorting

- Possible algorithm:
  - Store all the numbers in a data structure $D$.
  - Repeatedly find the smallest number in $D$, output it, and remove it.
- Data structure must support three operations: insertion of a number, finding minimum, and deleting minimum in $O(\log n)$ time.
Priority Queue

- Store a set $S$ of elements, where each element $v$ has a priority value $\text{key}(v)$.
- Smaller key values $\equiv$ higher priorities.
- Operations supported:
  - find the element with smallest key
  - remove the smallest element
  - insert an element
  - delete an element
  - update the key of an element

Element deletion and key update require knowledge of the position of the element in the priority queue.
Heaps

- Combine benefits of both lists and sorted arrays.
- Conceptually, a heap is a balanced binary tree.
- **Heap order**: For every element $v$ at a node $i$, the element $w$ at $i$’s parent satisfies $\text{key}(w) \leq \text{key}(v)$.
Heaps

- Combine benefits of both lists and sorted arrays.
- Conceptually, a heap is a balanced binary tree.
- **Heap order**: For every element \( v \) at a node \( i \), the element \( w \) at \( i \)'s parent satisfies \( \text{key}(w) \leq \text{key}(v) \).
- We can implement a heap in a pointer-based data structure.
Heaps

- Combine benefits of both lists and sorted arrays.
- Conceptually, a heap is a balanced binary tree.
- **Heap order**: For every element \( v \) at a node \( i \), the element \( w \) at \( i \)'s parent satisfies \( \text{key}(w) \leq \text{key}(v) \).
- We can implement a heap in a pointer-based data structure.
- Alternatively, assume maximum number \( N \) of elements is known in advance.
- Store nodes of the heap in an array.
  - Node at index \( i \) has children at indices \( 2i \) and \( 2i + 1 \) and parent at index \( \lfloor i/2 \rfloor \).
  - Index 1 is the root.
  - How do you know that a node at index \( i \) is a leaf?
Heaps

- Combine benefits of both lists and sorted arrays.
- Conceptually, a heap is a balanced binary tree.
- **Heap order**: For every element \( v \) at a node \( i \), the element \( w \) at \( i \)’s parent satisfies \( \text{key}(w) \leq \text{key}(v) \).
- We can implement a heap in a pointer-based data structure.
- Alternatively, assume maximum number \( N \) of elements is known in advance.
- Store nodes of the heap in an array.
  - Node at index \( i \) has children at indices \( 2i \) and \( 2i + 1 \) and parent at index \( \lfloor i/2 \rfloor \).
  - Index 1 is the root.
  - How do you know that a node at index \( i \) is a leaf? If \( 2i > n \), where \( n \) is the current number of elements in the heap.
Example of a Heap

Figure 2.3 Values in a heap shown as a binary tree on the left, and represented as an array on the right. The arrows show the children for the top three nodes in the tree.
Inserting an Element: Heapify-up

1. Insert new element at index \( n + 1 \).
2. Fix heap order using \( \text{Heapify-up}(H, n + 1) \).

---

\text{Heapify-up}(H,i):

\begin{align*}
\text{If } i &> 1 \text{ then} \\
\text{let } j &= \text{parent}(i) = \lfloor i/2 \rfloor \\
\text{If key}[H[i]] &< \text{key}[H[j]] \text{ then} \\
\text{swap the array entries } &H[i] \text{ and } H[j] \\
\text{Heapify-up}(H,j) & \\
\text{Endif} \end{align*}

\text{Endif}
Inserting an Element: Heapify-up

1. Insert new element at index $n + 1$.
2. Fix heap order using $\text{Heapify-up}(H, n + 1)$.

\begin{align*}
\text{Heapify-up}(H, i) : \\
\text{If } i > 1 \text{ then} \\
\text{let } j = \text{parent}(i) = \lfloor i/2 \rfloor \\
\text{If } \text{key}[H[i]] < \text{key}[H[j]] \text{ then} \\
\quad \text{swap the array entries } H[i] \text{ and } H[j] \\
\quad \text{Heapify-up}(H, j) \\
\text{Endif} \\
\text{Endif}
\end{align*}

- Proof of correctness: read pages 61–62 of your textbook.
**Example of Heapify-up**

The Heapify-up process is moving element \(v\) toward the root.

**Figure 2.4** The Heapify-up process. Key 3 (at position 16) is too small (on the left). After swapping keys 3 and 11, the heap violation moves one step closer to the root of the tree (on the right).
Running time of Heapify-up

Heapify-up(H,i):

If \( i > 1 \) then

let \( j = \text{parent}(i) = \lfloor i/2 \rfloor \)

If \( \text{key}[H[i]] < \text{key}[H[j]] \) then

swap the array entries \( H[i] \) and \( H[j] \)

Heapify-up(H,j)

Endif

Endif

- Running time of Heapify-up(i)
Running time of Heapify-up

Heapify-up(H,i):
  If $i > 1$ then
    let $j = \text{parent}(i) = \lfloor i/2 \rfloor$
    If $\text{key}[H[i]] < \text{key}[H[j]]$ then
      swap the array entries $H[i]$ and $H[j]$
      Heapify-up(H,j)
    Endif
  Endif
Endif

- Running time of Heapify-up($i$) is $O(\log i)$.
  - Each invocation decreases the second argument by a factor of at least 2.
  - After $k$ invocations, argument is at most $i/2^k$.
  - Therefore $i/2^k \geq 1$, which implies that $k \leq \log_2 i$. 
Deleting an Element: **Heapify-down**

1. Suppose $H$ has $n + 1$ elements.
3. If element at $H[i]$ is too small, fix heap order using Heapify-up($H, i$).
4. If element at $H[i]$ is too large, fix heap order using Heapify-down($H, i$).

**Heapify-down($H, i$):**

Let $n = \text{length}(H)$

If $2i > n$ then
   Terminate with $H$ unchanged
Else if $2i < n$ then
   Let $\text{left} = 2i$, and $\text{right} = 2i + 1$
   Let $j$ be the index that minimizes $\text{key}[H[\text{left}]]$ and $\text{key}[H[\text{right}]]$
Else if $2i = n$ then
   Let $j = 2i$
Endif

If $\text{key}[H[j]] < \text{key}[H[i]]$ then
   swap the array entries $H[i]$ and $H[j]$
   Heapify-down($H, j$)
Endif
Deleting an Element: Heapify-down

- Suppose $H$ has $n + 1$ elements.

1. Delete element at $H[i]$ by moving element at $H[n + 1]$ to $H[i]$.
2. If element at $H[i]$ is too small, fix heap order using Heapify-up($H, i$).
3. If element at $H[i]$ is too large, fix heap order using Heapify-down($H, i$).

Heapify-down($H, i$):

Let $n = \text{length}(H)$

If $2i > n$ then
   Terminate with $H$ unchanged
Else if $2i < n$ then
   Let $\text{left} = 2i$, and $\text{right} = 2i + 1$
   Let $j$ be the index that minimizes $\text{key}[H[\text{left}]]$ and $\text{key}[H[\text{right}]]$
Else if $2i = n$ then
   Let $j = 2i$
Endif

If $\text{key}[H[j]] < \text{key}[H[i]]$ then
   swap the array entries $H[i]$ and $H[j]$  
   Heapify-down($H, j$)
Endif
Example of Heapify-down

The Heapify-down process is moving element $w$ down, toward the leaves.

Figure 2.5 The Heapify-down process: Key 21 (at position 3) is too big (on the left). After swapping keys 21 and 7, the heap violation moves one step closer to the bottom of the tree (on the right).
Running time of Heapify-down

Heapify-down(H,i):

Let n = length(H)
If 2i > n then
  Terminate with H unchanged
Else if 2i < n then
  Let left = 2i, and right = 2i + 1
  Let j be the index that minimizes key[H[left]] and key[H[right]]
Else if 2i = n then
  Let j = i
Endif
If key[H[j]] < key[H[i]] then
  swap the array entries H[i] and H[j]
  Heapify-down(H,j)
Endif

Every invocation of Heapify-down increases its second argument by a factor of at least two.
## Running time of Heapify-down

The running time of Heapify-down is $O\left(\log_2 \frac{n}{i}\right)$.

### Heapify-down(H,i):

1. Let $n = \text{length}(H)$
2. If $2i > n$ then
   - Terminate with $H$ unchanged
3. Else if $2i < n$ then
   - Let $\text{left} = 2i$, and $\text{right} = 2i + 1$
   - Let $j$ be the index that minimizes $\text{key}[H[\text{left}]]$ and $\text{key}[H[\text{right}]]$.
4. Else if $2i = n$ then
   - Let $j = 2i$
5. Endif

- If $\text{key}[H[j]] < \text{key}[H[i]]$ then
  - swap the array entries $H[i]$ and $H[j]$
  - Heapify-down($H,j$)
- Endif

- **Every invocation of Heapify-down increases its second argument by a factor of at least two.**
- After $k$ invocations argument must be at least
Running time of Heapify-down

Heapify-down(H,i):
    Let $n = \text{length}(H)$
    If $2i > n$ then
        Terminate with $H$ unchanged
    Else if $2i < n$ then
        Let left = $2i$, and right = $2i + 1$
        Let $j$ be the index that minimizes $\text{key}[H[\text{left}]]$ and $\text{key}[H[\text{right}]]$
    Else if $2i = n$ then
        Let $j = 2i$
    Endif
    If $\text{key}[H[j]] < \text{key}[H[i]]$ then
        swap the array entries $H[i]$ and $H[j]$
        Heapify-down($H,j$)
    Endif

- Every invocation of Heapify-down increases its second argument by a factor of at least two.
- After $k$ invocations argument must be at least $i 2^k \leq n$, which implies that $k \leq \log_2 \frac{n}{i}$. Therefore running time is $O(\log_2 \frac{n}{i})$. 
Sort

**INSTANCE:** Nonempty list $x_1, x_2, \ldots, x_n$ of integers.

**SOLUTION:** A permutation $y_1, y_2, \ldots, y_n$ of $x_1, x_2, \ldots, x_n$ such that $y_i \leq y_{i+1}$, for all $1 \leq i < n$. 

---

"Final algorithm:

1. Insert each number in a priority queue $H$.
2. Repeatedly find the smallest number in $H$, output it, and delete it from $H$.

Each insertion and deletion takes $O(\log n)$ time for a total running time of $O(n \log n)$.

---

T. M. Murali August 27, 29 2018 Review of Priority Queues and Graph Searches
Sort

**INSTANCE:** Nonempty list $x_1, x_2, \ldots, x_n$ of integers.

**SOLUTION:** A permutation $y_1, y_2, \ldots, y_n$ of $x_1, x_2, \ldots, x_n$ such that $y_i \leq y_{i+1}$, for all $1 \leq i < n$.

- **Final algorithm:**
  - Insert each number in a priority queue $H$.
  - Repeatedly find the smallest number in $H$, output it, and delete it from $H$. 
Sort

**INSTANCE:** Nonempty list $x_1, x_2, \ldots, x_n$ of integers.

**SOLUTION:** A permutation $y_1, y_2, \ldots, y_n$ of $x_1, x_2, \ldots, x_n$ such that $y_i \leq y_{i+1}$, for all $1 \leq i < n$.

- **Final algorithm:**
  - Insert each number in a priority queue $H$.
  - Repeatedly find the smallest number in $H$, output it, and delete it from $H$.

- Each insertion and deletion takes $O(\log n)$ time for a total running time of $O(n \log n)$.
The Oracle of Bacon
Review of Priority Queues and Graph Searches
Priorities

Queues

Graph Definitions

Graph Traversal

BFS

DFS

Implementations

T. M. Murali
August 27, 29 2018

Review of Priorities and Graph Searches
Review of Priority Queues and Graph Searches

T. M. Murali
August 27, 29 2018
Graphs

- Model pairwise relationships (edges) between objects (nodes).
Graphs

- Model pairwise relationships (edges) between objects (nodes).
- Useful in a large number of applications:
Graphs

- Model pairwise relationships (edges) between objects (nodes).
- Useful in a large number of applications: computer networks, the World Wide Web, ecology (food webs), social networks, software systems, job scheduling, VLSI circuits, cellular networks, ...
- Other examples: gene and protein networks, our bodies (nervous and circulatory systems, brains), buildings, transportation networks, ...
Graphs

- Model pairwise relationships (edges) between objects (nodes).
- Useful in a large number of applications: computer networks, the World Wide Web, ecology (food webs), social networks, software systems, job scheduling, VLSI circuits, cellular networks, ... 
- Other examples: gene and protein networks, our bodies (nervous and circulatory systems, brains), buildings, transportation networks, ...
Euler and Graphs

Devise a walk through the city that crosses each of the seven bridges exactly once.
Euler and Graphs
Definition of a Graph

- **Undirected graph** \( G = (V, E) \): set \( V \) of nodes and set \( E \) of edges, where \( E \subseteq V \times V \).
  - Elements of \( E \) are unordered pairs.
  - Edge \((u, v)\) is *incident* on \( u, v \); \( u \) and \( v \) are *neighbours* of each other.
  - Exactly one edge between any pair of nodes.
  - \( G \) contains no self loops, i.e., no edges of the form \((u, u)\).
Definition of a Graph

- Directed graph $G = (V, E)$: set $V$ of nodes and set $E$ of edges, where $E \subseteq V \times V$.
  - Elements of $E$ are ordered pairs.
  - $e = (u, v)$: $u$ is the tail of the edge $e$, $v$ is its head; $e$ is directed from $u$ to $v$.
  - A pair of nodes may be connected by two directed edges: $(u, v)$ and $(v, u)$.
  - $G$ contains no self loops.
A $v_1$-$v_k$ path in an undirected graph $G = (V, E)$ is a sequence $P$ of nodes $v_1, v_2, \ldots, v_{k-1}, v_k \in V$ such that every consecutive pair of nodes $v_i, v_{i+1}, 1 \leq i < k$ is connected by an edge in $E$. 
A \( v_1-v_k \) path in an undirected graph \( G = (V, E) \) is a sequence \( P \) of nodes \( v_1, v_2, \ldots, v_{k-1}, v_k \in V \) such that every consecutive pair of nodes \( v_i, v_{i+1}, 1 \leq i < k \) is connected by an edge in \( E \).
A $v_1$-$v_k$ path in an undirected graph $G = (V, E)$ is a sequence $P$ of nodes $v_1, v_2, \ldots, v_{k-1}, v_k \in V$ such that every consecutive pair of nodes $v_i, v_{i+1}, 1 \leq i < k$ is connected by an edge in $E$. 
A \( v_1 - v_k \) path in an undirected graph \( G = (V, E) \) is a sequence \( P \) of nodes \( v_1, v_2, \ldots, v_{k-1}, v_k \in V \) such that every consecutive pair of nodes \( v_i, v_{i+1}, 1 \leq i < k \) is connected by an edge in \( E \).

A path is \textit{simple} if all its nodes are distinct.
A \( v_1 - v_k \) path in an undirected graph \( G = (V, E) \) is a sequence \( P \) of nodes \( v_1, v_2, \ldots, v_{k-1}, v_k \in V \) such that every consecutive pair of nodes \( v_i, v_{i+1}, 1 \leq i < k \) is connected by an edge in \( E \).

- A path is simple if all its nodes are distinct.
- A cycle is a path where \( k > 2 \), the first \( k - 1 \) nodes are distinct, and \( v_1 = v_k \).
A $v_1$-$v_k$ path in an undirected graph $G = (V, E)$ is a sequence $P$ of nodes $v_1, v_2, \ldots, v_{k-1}, v_k \in V$ such that every consecutive pair of nodes $v_i, v_{i+1}, 1 \leq i < k$ is connected by an edge in $E$.

A path is simple if all its nodes are distinct.

A cycle is a path where $k > 2$, the first $k-1$ nodes are distinct, and $v_1 = v_k$.

Similar definitions carry over to directed graphs as well.
A v₁-vₖ path in an undirected graph G = (V, E) is a sequence P of nodes v₁, v₂, ..., vₖ−1, vₖ ∈ V such that every consecutive pair of nodes vᵢ, vᵢ₊₁, 1 ≤ i < k is connected by an edge in E.

A path is simple if all its nodes are distinct.

A cycle is a path where k > 2, the first k − 1 nodes are distinct, and v₁ = vₖ.

Similar definitions carry over to directed graphs as well.

An undirected graph G is connected if for every pair of nodes u, v ∈ V, there is a path from u to v in G.
Paths and Connectivity

- A \(v_1-v_k\) path in an undirected graph \(G = (V, E)\) is a sequence \(P\) of nodes \(v_1, v_2, \ldots, v_{k-1}, v_k \in V\) such that every consecutive pair of nodes \(v_i, v_{i+1}, 1 \leq i < k\) is connected by an edge in \(E\).
- A path is simple if all its nodes are distinct.
- A cycle is a path where \(k > 2\), the first \(k - 1\) nodes are distinct, and \(v_1 = v_k\).
- Similar definitions carry over to directed graphs as well.
- An undirected graph \(G\) is connected if for every pair of nodes \(u, v \in V\), there is a path from \(u\) to \(v\) in \(G\).
A $v_1$-$v_k$ path in an undirected graph $G = (V, E)$ is a sequence $P$ of nodes $v_1, v_2, \ldots, v_{k-1}, v_k \in V$ such that every consecutive pair of nodes $v_i, v_{i+1}, 1 \leq i < k$ is connected by an edge in $E$.

A path is simple if all its nodes are distinct.

A cycle is a path where $k > 2$, the first $k - 1$ nodes are distinct, and $v_1 = v_k$.

Similar definitions carry over to directed graphs as well.

An undirected graph $G$ is connected if for every pair of nodes $u, v \in V$, there is a path from $u$ to $v$ in $G$.

$Distance\ d(u, v)$ between two nodes $u$ and $v$ is the minimum number of edges in any $u$-$v$ path.
**s-t Connectivity**

**INSTANCE:** An undirected graph $G = (V, E)$ and two nodes $s, t \in V$.

**QUESTION:** Is there an $s$-$t$ path in $G$?
$s-t$ Connectivity

**INSTANCE:** An undirected graph $G = (V, E)$ and two nodes $s, t \in V$.

**QUESTION:** Is there an $s-t$ path in $G$?

- The *connected component of $G$ containing $s$* is the set of all nodes $u$ such that there is an $s-u$ path in $G$. 
**s–t Connectivity**

**INSTANCE:** An undirected graph $G = (V, E)$ and two nodes $s, t \in V$.

**QUESTION:** Is there an $s$-$t$ path in $G$?

- The *connected component of $G$ containing $s$* is the set of all nodes $u$ such that there is an $s$-$u$ path in $G$.
- Algorithm for the $s$-$t$ Connectivity problem: compute the connected component of $G$ that contains $s$ and check if $t$ is in that component.
Computing Connected Components

- “Explore” $G$ starting from $s$ and maintain set $R$ of visited nodes.

---

$R$ will consist of nodes to which $s$ has a path.

Initially $R = \{s\}$

While there is an edge $(u, v)$ where $u \in R$ and $v \notin R$
  - Add $v$ to $R$

Endwhile
Computing Connected Components

“Explore” \( G \) starting from \( s \) and maintain set \( R \) of visited nodes.

\[ R \text{ will consist of nodes to which } s \text{ has a path} \]
Initially \( R = \{s\} \)
While there is an edge \((u, v)\) where \( u \in R \) and \( v \notin R \)
   Add \( v \) to \( R \)
Endwhile
Computing Connected Components

- “Explore” $G$ starting from $s$ and maintain set $R$ of visited nodes.

$R$ will consist of nodes to which $s$ has a path
Initially $R = \{s\}$
While there is an edge $(u, v)$ where $u \in R$ and $v \notin R$
    Add $v$ to $R$
Endwhile
Computing Connected Components

- “Explore” $G$ starting from $s$ and maintain set $R$ of visited nodes.

$R$ will consist of nodes to which $s$ has a path
Initially $R = \{s\}$
While there is an edge $(u, v)$ where $u \in R$ and $v \notin R$
  Add $v$ to $R$
Endwhile
Computing Connected Components

- “Explore” $G$ starting from $s$ and maintain set $R$ of visited nodes.

$R$ will consist of nodes to which $s$ has a path

Initially $R = \{s\}$

While there is an edge $(u, v)$ where $u \in R$ and $v \notin R$
  Add $v$ to $R$

Endwhile
Computing Connected Components

“Explore” $G$ starting from $s$ and maintain set $R$ of visited nodes.

$R$ will consist of nodes to which $s$ has a path.
Initially $R = \{s\}$
While there is an edge $(u, v)$ where $u \in R$ and $v \notin R$
    Add $v$ to $R$
Endwhile
Computing Connected Components

- “Explore” $G$ starting from $s$ and maintain set $R$ of visited nodes.

$R$ will consist of nodes to which $s$ has a path
Initially $R = \{s\}$
While there is an edge $(u, v)$ where $u \in R$ and $v \notin R$
  Add $v$ to $R$
Endwhile
Computing Connected Components

• “Explore” $G$ starting from $s$ and maintain set $R$ of visited nodes.

$R$ will consist of nodes to which $s$ has a path
Initially $R = \{s\}$
While there is an edge $(u, v)$ where $u \in R$ and $v \notin R$
  Add $v$ to $R$
Endwhile
Computing Connected Components

- “Explore” $G$ starting from $s$ and maintain set $R$ of visited nodes.

$R$ will consist of nodes to which $s$ has a path
Initially $R = \{s\}$
While there is an edge $(u, v)$ where $u \in R$ and $v \notin R$
  Add $v$ to $R$
Endwhile
Issues in Computing Connected Components

\( R \) will consist of nodes to which \( s \) has a path.
Initially \( R = \{s\} \)
While there is an edge \((u, v)\) where \( u \in R \) and \( v \notin R \)
  Add \( v \) to \( R \)
Endwhile

- Why does the algorithm terminate?
- Does the algorithm truly compute connected component of \( G \) containing \( s \)?
- What is the running time of the algorithm?
Why does the algorithm terminate? Each iteration adds a new node to \( R \).

Does the algorithm truly compute connected component of \( G \) containing \( s \)?

What is the running time of the algorithm?

\[
R \text{ will consist of nodes to which } s \text{ has a path} \\
\text{Initially } R = \{s\} \\
\text{While there is an edge } (u, v) \text{ where } u \in R \text{ and } v \notin R \\
\quad \text{Add } v \text{ to } R \]
\text{Endwhile}
Correctness of the Algorithm

Claim: at the end of the algorithm, the set \( R \) is exactly the connected component of \( G \) containing \( s \).

---

\( R \) will consist of nodes to which \( s \) has a path
Initially \( R = \{s\} \)
While there is an edge \((u, v)\) where \( u \in R \) and \( v \notin R \)
    Add \( v \) to \( R \)
Endwhile

---

T. M. Murali August 27, 29 2018 Review of Priority Queues and Graph Searches
Correctness of the Algorithm

Claim: at the end of the algorithm, the set $R$ is exactly the connected component of $G$ containing $s$.

Proof: At termination, suppose $w \not\in R$ but there is an $s$-$w$ path $P$ in $G$.

1. Consider first node $v$ in $P$ not in $R$ ($v \neq s$).
2. Let $u$ be the predecessor of $v$ in $P$.

$R$ will consist of nodes to which $s$ has a path
Initially $R = \{s\}$
While there is an edge $(u, v)$ where $u \in R$ and $v \not\in R$
    Add $v$ to $R$
Endwhile

T. M. Murali August 27, 29 2018 Review of Priority Queues and Graph Searches
Correctness of the Algorithm

- **Claim:** at the end of the algorithm, the set $R$ is exactly the connected component of $G$ containing $s$.

- **Proof:** At termination, suppose $w \notin R$ but there is an $s$-$w$ path $P$ in $G$.
  - Consider first node $v$ in $P$ not in $R$ ($v \neq s$).
  - Let $u$ be the predecessor of $v$ in $P$: $u$ is in $R$.
  - $(u, v)$ is an edge with $u \in R$ but $v \notin R$, contradicting the stopping rule.

\[ R \text{ will consist of nodes to which } s \text{ has a path} \]
\[ \text{Initially } R = \{s\} \]
\[ \text{While there is an edge } (u, v) \text{ where } u \in R \text{ and } v \notin R \]
\[ \quad \text{Add } v \text{ to } R \]
\[ \text{Endwhile} \]
Correctness of the Algorithm

Claim: at the end of the algorithm, the set $R$ is exactly the connected component of $G$ containing $s$.

Proof: At termination, suppose $w \notin R$ but there is an $s$-$w$ path $P$ in $G$.

- Consider first node $v$ in $P$ not in $R$ ($v \neq s$).
- Let $u$ be the predecessor of $v$ in $P$: $u$ is in $R$.
- $(u, v)$ is an edge with $u \in R$ but $v \notin R$, contradicting the stopping rule.
- Note: wrong to assume that predecessor of $w$ in $P$ is not in $R$. 

---

$R$ will consist of nodes to which $s$ has a path.
Initially $R=\{s\}$
While there is an edge $(u, v)$ where $u \in R$ and $v \notin R$
  Add $v$ to $R$
Endwhile
Running Time of the Algorithm

$R$ will consist of nodes to which $s$ has a path

Initially $R = \{s\}$

While there is an edge $(u,v)$ where $u \in R$ and $v \notin R$
   Add $v$ to $R$

Endwhile

Analyse algorithm in terms of two parameters: the number of nodes $n$ and the number of edges $m$.

Implement the while loop by examining each edge in $E$. Running time of each loop is $O(m)$.

How many while loops does the algorithm execute?

At most $n$.

The running time is $O(mn)$.

Can we improve the running time by processing edges more carefully?
Running Time of the Algorithm

R will consist of nodes to which s has a path
Initially $R = \{s\}$
While there is an edge $(u, v)$ where $u \in R$ and $v \notin R$
  Add $v$ to $R$
Endwhile

- Analyse algorithm in terms of two parameters: the number of nodes $n$ and the number of edges $m$.
- Implement the while loop by examining each edge in $E$. Running time of each loop is
Running Time of the Algorithm

$R$ will consist of nodes to which $s$ has a path
Initially $R = \{s\}$
While there is an edge $(u,v)$ where $u \in R$ and $v \notin R$
  Add $v$ to $R$
Endwhile

- Analyse algorithm in terms of two parameters: the number of nodes $n$ and the number of edges $m$.
- Implement the while loop by examining each edge in $E$. Running time of each loop is $O(m)$.
- How many while loops does the algorithm execute?
Running Time of the Algorithm

R will consist of nodes to which s has a path
Initially $R = \{s\}$
While there is an edge $(u, v)$ where $u \in R$ and $v \notin R$
    Add $v$ to $R$
Endwhile

- Analyse algorithm in terms of two parameters: the number of nodes $n$ and the number of edges $m$.
- Implement the while loop by examining each edge in $E$. Running time of each loop is $O(m)$.
- How many while loops does the algorithm execute? At most $n$.
- The running time is
Running Time of the Algorithm

$R$ will consist of nodes to which $s$ has a path.
Initially $R = \{s\}$
While there is an edge $(u, v)$ where $u \in R$ and $v \notin R$
    Add $v$ to $R$
Endwhile

- Analyse algorithm in terms of two parameters: the number of nodes $n$ and the number of edges $m$.
- Implement the while loop by examining each edge in $E$. Running time of each loop is $O(m)$.
- How many while loops does the algorithm execute? At most $n$.
- The running time is $O(mn)$. 

Running Time of the Algorithm

$R$ will consist of nodes to which $s$ has a path
Initially $R = \{s\}$
While there is an edge $(u, v)$ where $u \in R$ and $v \notin R$
    Add $v$ to $R$
Endwhile

- Analyse algorithm in terms of two parameters: the number of nodes $n$ and the number of edges $m$.
- Implement the while loop by examining each edge in $E$. Running time of each loop is $O(m)$.
- How many while loops does the algorithm execute? At most $n$.
- The running time is $O(mn)$.
- Can we improve the running time by processing edges more carefully?
Breadth-First Search (BFS)

- Idea: explore $G$ starting at $s$ and going “outward” in all directions, adding nodes one layer at a time.
Breadth-First Search (BFS)

- Idea: explore $G$ starting at $s$ and going “outward” in all directions, adding nodes one layer at a time.
- Layer $L_0$ contains only $s$. 
Breadth-First Search (BFS)

- Idea: explore $G$ starting at $s$ and going “outward” in all directions, adding nodes one layer at a time.
- Layer $L_0$ contains only $s$.
- Layer $L_1$ contains all neighbours of $s$. 
Breadth-First Search (BFS)

Idea: explore $G$ starting at $s$ and going “outward” in all directions, adding nodes one layer at a time.

Layer $L_0$ contains only $s$.

Layer $L_1$ contains all neighbours of $s$.

Given layers $L_0, L_1, \ldots, L_j$, layer $L_{j+1}$ contains all nodes that

1. do not belong to an earlier layer and
2. are connected by an edge to a node in layer $L_j$. 
Breadth-First Search (BFS)

- Idea: explore $G$ starting at $s$ and going “outward” in all directions, adding nodes one layer at a time.
- Layer $L_0$ contains only $s$.
- Layer $L_1$ contains all neighbours of $s$.
- Given layers $L_0, L_1, \ldots, L_j$, layer $L_{j+1}$ contains all nodes that
  1. do not belong to an earlier layer and
  2. are connected by an edge to a node in layer $L_j$. 
We have not yet described how to compute these layers.

Claim: For each $j \geq 1$, layer $L_j$ consists of all nodes
We have not yet described how to compute these layers.

Claim: For each $j \geq 1$, layer $L_j$ consists of all nodes exactly at distance $j$ from $S$. Proof

Properties of BFS
We have not yet described how to compute these layers.

Claim: For each $j \geq 1$, layer $L_j$ consists of all nodes exactly at distance $j$ from $S$. Proof by induction on $j$.

Claim: There is a path from $s$ to $t$ if and only if $t$ is a member of some layer.
We have not yet described how to compute these layers.

Claim: For each \( j \geq 1 \), layer \( L_j \) consists of all nodes exactly at distance \( j \) from \( S \). Proof by induction on \( j \).

Claim: There is a path from \( s \) to \( t \) if and only if \( t \) is a member of some layer.

Let \( v \) be a node in layer \( L_{j+1} \) and \( u \) be the “first” node in \( L_j \) such that \((u, v)\) is an edge in \( G \). Consider the graph \( T \) formed by all such edges, directed from \( u \) to \( v \).
We have not yet described how to compute these layers.

Claim: For each $j \geq 1$, layer $L_j$ consists of all nodes exactly at distance $j$ from $S$. Proof by induction on $j$.

Claim: There is a path from $s$ to $t$ if and only if $t$ is a member of some layer.

Let $v$ be a node in layer $L_{j+1}$ and $u$ be the “first” node in $L_j$ such that $(u, v)$ is an edge in $G$. Consider the graph $T$ formed by all such edges, directed from $u$ to $v$.

Why is $T$ a tree?
Properties of BFS

- We have not yet described how to compute these layers.
- Claim: For each \( j \geq 1 \), layer \( L_j \) consists of all nodes exactly at distance \( j \) from \( S \). Proof by induction on \( j \).
- Claim: There is a path from \( s \) to \( t \) if and only if \( t \) is a member of some layer.
- Let \( v \) be a node in layer \( L_{j+1} \) and \( u \) be the “first” node in \( L_j \) such that \((u, v)\) is an edge in \( G \). Consider the graph \( T \) formed by all such edges, directed from \( u \) to \( v \).
  - Why is \( T \) a tree? It is connected. The number of edges in \( T \) is the number of nodes in all the layers minus 1.
  - \( T \) is called the breadth-first search tree.
**Non-tree edge**: an edge of $G$ that does not belong to the BFS tree $T$.

Claim: Let $T$ be a BFS tree, let $x$ and $y$ be nodes in $T$ belonging to layers $L_i$ and $L_j$, respectively, and let $(x, y)$ be an edge of $G$. Then $|i - j| \leq 1$. 
**Non-tree edge**: an edge of $G$ that does not belong to the BFS tree $T$.

**Claim**: Let $T$ be a BFS tree, let $x$ and $y$ be nodes in $T$ belonging to layers $L_i$ and $L_j$, respectively, and let $(x, y)$ be an edge of $G$. Then $|i - j| \leq 1$.

**Proof by contradiction**: Suppose $i < j - 1$. Node $x \in L_i$ $\Rightarrow$ all nodes adjacent to $x$ are in layers $L_1, L_2, \ldots L_{i+1}$. Hence $y$ must be in layer $L_{i+1}$ or earlier.
Non-tree edge: an edge of $G$ that does not belong to the BFS tree $T$.

Claim: Let $T$ be a BFS tree, let $x$ and $y$ be nodes in $T$ belonging to layers $L_i$ and $L_j$, respectively, and let $(x, y)$ be an edge of $G$. Then $|i - j| \leq 1$.

Proof by contradiction: Suppose $i < j - 1$. Node $x \in L_i \Rightarrow$ all nodes adjacent to $x$ are in layers $L_1, L_2, \ldots L_{i+1}$. Hence $y$ must be in layer $L_{i+1}$ or earlier.

Still unresolved: an efficient implementation of BFS.
Depth-First Search (DFS)

- Explore $G$ as if it were a maze: start from $s$, traverse first edge out (to node $v$), traverse first edge out of $v$, ... , reach a dead-end, backtrack, ....
Depth-First Search (DFS)

- Explore G as if it were a maze: start from s, traverse first edge out (to node v), traverse first edge out of v, ..., reach a dead-end, backtrack, ....

1. Mark all nodes as "Unexplored".
2. Invoke DFS(s).

---

DFS(u):
Mark u as "Explored" and add u to R
For each edge (u,v) incident to u
   If v is not marked "Explored" then
      Recursively invoke DFS(v)
   Endif
Endfor
Depth-First Search (DFS)

- Explore $G$ as if it were a maze: start from $s$, traverse first edge out (to node $v$), traverse first edge out of $v$, ..., reach a dead-end, backtrack, ....

1. Mark all nodes as “Unexplored”.
2. Invoke DFS($s$).

\[
\text{DFS}(u):
\]
- Mark $u$ as "Explored" and add $u$ to $R$
- For each edge $(u, v)$ incident to $u$
  - If $v$ is not marked "Explored" then
    - Recursively invoke DFS($v$)
  - Endif
- Endfor

- **Depth-first search tree** is a tree $T$: when DFS($v$) is invoked directly during the call to DFS($v$), add edge $(u, v)$ to $T$. 
Example of DFS
Example of DFS
Example of DFS

Graph

1 -> 2
2 -> 3
3 -> 4
3 -> 5
5 -> 6
4 -> 7
4 -> 8
5 -> 9
8 -> 10
8 -> 11
9 -> 12
12 -> 13

DFS Path:
1 -> 2 -> 3 -> 4 -> 7 -> 5 -> 6
1 -> 2 -> 3 -> 5 -> 8
1 -> 2 -> 3 -> 8
1 -> 2 -> 5
1 -> 2
1

T. M. Murali August 27, 29 2018 Review of Priority Queues and Graph Searches
Example of DFS
Example of DFS
Example of DFS
Example of DFS
Example of DFS
Both visit the same set of nodes but in a different order.
Both traverse all the edges in the connected component but in a different order.
BFS trees have root-to-leaf paths that look as short as possible while paths in DFS trees tend to be long and deep.
Non-tree edges
BFS within the same level or between adjacent levels.
Both visit the same set of nodes but in a different order.
Both traverse all the edges in the connected component but in a different order.
BFS trees have root-to-leaf paths that look as short as possible while paths in DFS trees tend to be long and deep.
Non-tree edges
- BFS within the same level or between adjacent levels.
- DFS connect ancestors to descendants.
Properties of DFS Trees

DFS\((u)\):
  Mark \(u\) as "Explored" and add \(u\) to \(R\)
  For each edge \((u,v)\) incident to \(u\)
    If \(v\) is not marked "Explored" then
      Recursively invoke DFS\((v)\)
    Endif
  Endfor

Observation: All nodes marked as “Explored” between the start of DFS\((u)\) and its end are descendants of \(u\) in the DFS tree \(T\).
Properties of DFS Trees

DFS(u):
Mark u as "Explored" and add u to R
For each edge (u,v) incident to u
  If v is not marked "Explored" then
    Recursively invoke DFS(v)
  Endif
Endfor

- Observation: All nodes marked as “Explored” between the start of DFS(u) and its end are descendants of u in the DFS tree T.
- Claim: Let x and y be nodes in a DFS tree T such that (x, y) is an edge of G but not of T. Then one of x or y is an ancestor of the other in T. Read proof on page 86 of your textbook.
Graph Definitions

Graph Traversal

BFS

DFS

Implementations

Representing Graphs

- Graph $G = (V, E)$ has two input parameters: $|V| = n, |E| = m$.
  - Size of the graph is defined to be $m + n$.
  - Strive for algorithms whose running time is linear in graph size, i.e., $O(m + n)$. 

T. M. Murali August 27, 29 2018 Review of Priority Queues and Graph Searches
Representing Graphs

- Graph $G = (V, E)$ has two input parameters: $|V| = n, |E| = m$.
  - Size of the graph is defined to be $m + n$.
  - Strive for algorithms whose running time is linear in graph size, i.e., $O(m + n)$.
- Assume $V = \{1, 2, \ldots, n − 1, n\}$.
- Adjacency matrix representation: $n \times n$ Boolean matrix, where the entry in row $i$ and column $j$ is $1$ iff the graph contains the edge $(i, j)$.
  - Space used is $\Theta(n^2)$.
Representing Graphs

- Graph $G = (V, E)$ has two input parameters: $|V| = n, |E| = m$.
  - Size of the graph is defined to be $m + n$.
  - Strive for algorithms whose running time is linear in graph size, i.e., $O(m + n)$.
- Assume $V = \{1, 2, \ldots, n - 1, n\}$.
- **Adjacency matrix** representation: $n \times n$ Boolean matrix, where the entry in row $i$ and column $j$ is 1 iff the graph contains the edge $(i, j)$.
  - Space used is $\Theta(n^2)$, which is optimal in the worst case.
  - Check if there is an edge between node $i$ and node $j$ in...
Representing Graphs

- Graph $G = (V, E)$ has two input parameters: $|V| = n, |E| = m$.
  - Size of the graph is defined to be $m + n$.
  - Strive for algorithms whose running time is linear in graph size, i.e., $O(m + n)$.

- Assume $V = \{1, 2, \ldots, n - 1, n\}$.

- **Adjacency matrix** representation: $n \times n$ Boolean matrix, where the entry in row $i$ and column $j$ is 1 iff the graph contains the edge $(i, j)$.
  - Space used is $\Theta(n^2)$, which is optimal in the worst case.
  - Check if there is an edge between node $i$ and node $j$ in $O(1)$ time.
  - Iterate over all the edges incident on node $i$ in
Representing Graphs

- **Graph** $G = (V, E)$ has two input parameters: $|V| = n, |E| = m$.
  - Size of the graph is defined to be $m + n$.
  - Strive for algorithms whose running time is linear in graph size, i.e., $O(m + n)$.
- Assume $V = \{1, 2, \ldots, n - 1, n\}$.
- **Adjacency matrix** representation: $n \times n$ Boolean matrix, where the entry in row $i$ and column $j$ is 1 iff the graph contains the edge $(i, j)$.
  - Space used is $\Theta(n^2)$, which is optimal in the worst case.
  - Check if there is an edge between node $i$ and node $j$ in $O(1)$ time.
  - Iterate over all the edges incident on node $i$ in $\Theta(n)$ time.
Representing Graphs

- Graph $G = (V, E)$ has two input parameters: $|V| = n, |E| = m$.
  - Size of the graph is defined to be $m + n$.
  - Strive for algorithms whose running time is linear in graph size, i.e., $O(m + n)$.
- Assume $V = \{1, 2, \ldots, n - 1, n\}$.
- **Adjacency matrix** representation: $n \times n$ Boolean matrix, where the entry in row $i$ and column $j$ is 1 iff the graph contains the edge $(i, j)$.
  - Space used is $\Theta(n^2)$, which is optimal in the worst case.
  - Check if there is an edge between node $i$ and node $j$ in $O(1)$ time.
  - Iterate over all the edges incident on node $i$ in $\Theta(n)$ time.
- **Adjacency list** representation: array $Adj$, where $Adj[v]$ stores the list of all nodes adjacent to $v$.
  - An edge $e = (u, v)$ appears twice: in $Adj[u]$ and $Adj[v]$. 
Representing Graphs

- **Graph** $G = (V, E)$ has two input parameters: $|V| = n, |E| = m$.
  - Size of the graph is defined to be $m + n$.
  - Strive for algorithms whose running time is linear in graph size, i.e., $O(m + n)$.
- Assume $V = \{1, 2, \ldots, n - 1, n\}$.
- **Adjacency matrix** representation: $n \times n$ Boolean matrix, where the entry in row $i$ and column $j$ is 1 iff the graph contains the edge $(i, j)$.
  - Space used is $\Theta(n^2)$, which is optimal in the worst case.
  - Check if there is an edge between node $i$ and node $j$ in $O(1)$ time.
  - Iterate over all the edges incident on node $i$ in $\Theta(n)$ time.
- **Adjacency list** representation: array $Adj$, where $Adj[v]$ stores the list of all nodes adjacent to $v$.
  - An edge $e = (u, v)$ appears twice: in $Adj[u]$ and $Adj[v]$.
  - $n_v$ = the number of neighbours of node $v$.
  - Space used is
Representing Graphs

- Graph \( G = (V, E) \) has two input parameters: \(|V| = n, |E| = m\).
  - Size of the graph is defined to be \( m + n \).
  - Strive for algorithms whose running time is linear in graph size, i.e., \( O(m + n) \).
- Assume \( V = \{1, 2, \ldots, n - 1, n\} \).
- **Adjacency matrix** representation: \( n \times n \) Boolean matrix, where the entry in row \( i \) and column \( j \) is 1 iff the graph contains the edge \((i, j)\).
  - Space used is \( \Theta(n^2) \), which is optimal in the worst case.
  - Check if there is an edge between node \( i \) and node \( j \) in \( O(1) \) time.
  - Iterate over all the edges incident on node \( i \) in \( \Theta(n) \) time.
- **Adjacency list** representation: array \( \text{Adj} \), where \( \text{Adj}[v] \) stores the list of all nodes adjacent to \( v \).
  - An edge \( e = (u, v) \) appears twice: in \( \text{Adj}[u] \) and \( \text{Adj}[v] \).
  - \( n_v \) = the number of neighbours of node \( v \).
  - Space used is \( O(n + \sum_{v \in G} n_v) = \)
Representing Graphs

- Graph $G = (V, E)$ has two input parameters: $|V| = n, |E| = m$.
  - Size of the graph is defined to be $m + n$.
  - Strive for algorithms whose running time is linear in graph size, i.e., $O(m + n)$.
- Assume $V = \{1, 2, \ldots, n-1, n\}$.
- **Adjacency matrix** representation: $n \times n$ Boolean matrix, where the entry in row $i$ and column $j$ is 1 iff the graph contains the edge $(i, j)$.
  - Space used is $\Theta(n^2)$, which is optimal in the worst case.
  - Check if there is an edge between node $i$ and node $j$ in $O(1)$ time.
  - Iterate over all the edges incident on node $i$ in $\Theta(n)$ time.
- **Adjacency list** representation: array Adj, where $\text{Adj}[v]$ stores the list of all nodes adjacent to $v$.
  - An edge $e = (u, v)$ appears twice: in $\text{Adj}[u]$ and $\text{Adj}[v]$.
  - $n_v$ = the number of neighbours of node $v$.
  - Space used is $O(n + \sum_{v \in G} n_v) = O(n + m)$, which is optimal for every graph.
  - Check if there is an edge between node $u$ and node $v$ in
Representing Graphs

- **Graph** $G = (V, E)$ has two input parameters: $|V| = n, |E| = m$.
  - Size of the graph is defined to be $m + n$.
  - Strive for algorithms whose running time is linear in graph size, i.e., $O(m + n)$.
- Assume $V = \{1, 2, \ldots, n - 1, n\}$.
- **Adjacency matrix** representation: $n \times n$ Boolean matrix, where the entry in row $i$ and column $j$ is 1 iff the graph contains the edge $(i, j)$.
  - Space used is $\Theta(n^2)$, which is optimal in the worst case.
  - Check if there is an edge between node $i$ and node $j$ in $O(1)$ time.
  - Iterate over all the edges incident on node $i$ in $\Theta(n)$ time.
- **Adjacency list** representation: array Adj, where Adj[v] stores the list of all nodes adjacent to v.
  - An edge $e = (u, v)$ appears twice: in Adj[u] and Adj[v].
  - $n_v =$ the number of neighbours of node $v$.
  - Space used is $O(n + \sum_{v \in G} n_v) = O(n + m)$, which is optimal for every graph.
  - Check if there is an edge between node $u$ and node $v$ in $O(n_u)$ time.
  - Iterate over all the edges incident on node $u$ in
Representing Graphs

- Graph $G = (V, E)$ has two input parameters: $|V| = n, |E| = m$.
  - Size of the graph is defined to be $m + n$.
  - Strive for algorithms whose running time is linear in graph size, i.e., $O(m + n)$.
- Assume $V = \{1, 2, \ldots, n - 1, n\}$.
- **Adjacency matrix** representation: $n \times n$ Boolean matrix, where the entry in row $i$ and column $j$ is 1 iff the graph contains the edge $(i, j)$.
  - Space used is $\Theta(n^2)$, which is optimal in the worst case.
  - Check if there is an edge between node $i$ and node $j$ in $O(1)$ time.
  - Iterate over all the edges incident on node $i$ in $\Theta(n)$ time.
- **Adjacency list** representation: array Adj, where Adj[$v$] stores the list of all nodes adjacent to $v$.
  - An edge $e = (u, v)$ appears twice: in Adj[$u$] and Adj[$v$].
  - $n_v =$ the number of neighbours of node $v$.
  - Space used is $O(n + \sum_{v \in G} n_v) = O(n + m)$, which is optimal for every graph.
  - Check if there is an edge between node $u$ and node $v$ in $O(n_u)$ time.
  - Iterate over all the edges incident on node $u$ in $\Theta(n_u)$ time.
Data Structures for Implementation

- “Implementation” of BFS and DFS: fully specify the algorithms and data structures so that we can obtain provably efficient times.
- Inner loop of both BFS and DFS: process the set of edges incident on a given node and the set of visited nodes.
- How do we store the set of visited nodes? Order in which we process the nodes is crucial.
Data Structures for Implementation

- “Implementation” of BFS and DFS: fully specify the algorithms and data structures so that we can obtain provably efficient times.
- Inner loop of both BFS and DFS: process the set of edges incident on a given node and the set of visited nodes.
- How do we store the set of visited nodes? Order in which we process the nodes is crucial.
  - BFS: store visited nodes in a queue (first-in, first-out).
  - DFS: store visited nodes in a stack (last-in, first-out)
Using a Queue in BFS

- Maintain an array Discovered and set Discovered[v] = true as soon as the algorithm sees v.
- Maintain all the layers in a single queue L.

BFS(s):
- Set Discovered[s] = true
- Set Discovered[v] = false, for all other nodes v
- Initialize L to consist of the single element s
- While L is not empty
  - Pop the node u at the head of L
  - Consider each edge (u, v) incident on u
  - If Discovered[v] = false then
    - Set Discovered[v] = true
    - Add edge (u, v) to the tree T
    - Push v to the back of L
  - Endif
- Endwhile
Using a Queue in BFS

- Maintain an array Discovered and set Discovered[v] = true as soon as the algorithm sees v.
- Maintain all the layers in a single queue L.

BFS(s):
- Set Discovered[s] = true
- Set Discovered[v] = false, for all other nodes v
- Initialize L to consist of the single element s
- While L is not empty
  - Pop the node u at the head of L
  - Consider each edge (u, v) incident on u
  - If Discovered[v] = false then
    - Set Discovered[v] = true
    - Add edge (u, v) to the tree T
    - Push v to the back of L
  - Endif
- Endwhile

Simple to modify this procedure to keep track of layer numbers as well.

Claim: More formally: If BFS(s) pops (v, lv) from L immediately after it pops (u, lu), then either lv = lu or lv = lu + 1.
Using a Queue in BFS

- Maintain an array `Discovered` and set `Discovered[v] = true` as soon as the algorithm sees `v`.
- Maintain all the layers in a single queue `L`.

BFS(s):
- Set `Discovered[s] = true`
- Set `Discovered[v] = false`, for all other nodes `v`
- Initialize `L` to consist of the single element `s`
- While `L` is not empty
  - Pop the node `u` at the head of `L`
  - Consider each edge `(u, v)` incident on `u`
  - If `Discovered[v] = false` then
    - Set `Discovered[v] = true`
    - Add edge `(u, v)` to the tree `T`
    - Push `v` to the back of `L`
  - Endif
- Endwhile

Simple to modify this procedure to keep track of layer numbers as well.

Store the pair `(u, l_u)`, where `l_u` is the index of the layer containing `u`.

Claim: More formally: If `BFS(s)` pops `(v, l_v)` from `L` immediately after it pops `(u, l_u)`, then either `l_v = l_u` or `l_v = l_u + 1`.
Using a Queue in BFS

- Maintain an array Discovered and set Discovered[v] = true as soon as the algorithm sees v.
- Maintain all the layers in a single queue L.

BFS(s):
- Set Discovered[s] = true
- Set Discovered[v] = false, for all other nodes v
- Initialize L to consist of the single element s
- While L is not empty
  - Pop the node u at the head of L
  - Consider each edge (u, v) incident on u
    - If Discovered[v] = false then
      - Set Discovered[v] = true
      - Add edge (u, v) to the tree T
      - Push v to the back of L
    - Endif
  - Endwhile

Claim: More formally: If BFS(s) pops (v, l_v) from L immediately after it pops (u, l_u), then either l_v = l_u or l_v = l_u + 1.
Using a Queue in BFS

- Maintain an array `Discovered` and set `Discovered[v] = true` as soon as the algorithm sees `v`.
- Maintain all the layers in a single queue `L`.

**BFS(s):**
- Set `Discovered[s] = true`
- Set `Discovered[v] = false`, for all other nodes `v`
- Initialize `L` to consist of the single element `s`
- While `L` is not empty
  - Pop the node `u` at the head of `L`
  - Consider each edge `(u, v)` incident on `u`
  - If `Discovered[v] = false` then
    - Set `Discovered[v] = true`
    - Add edge `(u, v)` to the tree `T`
    - Push `v` to the back of `L`
  - Endif
- Endwhile
Using a Queue in BFS

- Maintain an array Discovered and set $\text{Discovered}[v] = true$ as soon as the algorithm sees $v$.
- Maintain all the layers in a single queue $L$.

**BFS($s$):**

1. Set $\text{Discovered}[s] = true$.
2. Set $\text{Discovered}[v] = false$, for all other nodes $v$.
3. Initialize $L$ to consist of the single element $s$.
4. While $L$ is not empty
   - Pop the node $u$ at the head of $L$.
   - Consider each edge $(u, v)$ incident on $u$.
   - If $\text{Discovered}[v] = false$ then
     - Set $\text{Discovered}[v] = true$.
     - Add edge $(u, v)$ to the tree $T$.
     - Push $v$ to the back of $L$.
   - Endif
5. Endwhile
Using a Queue in BFS

- Maintain an array Discovered and set Discovered[\(v\)] = true as soon as the algorithm sees \(v\).
- Maintain all the layers in a single queue \(L\).

**BFS(s):**

1. Set Discovered[s] = true
2. Set Discovered[\(v\)] = false, for all other nodes \(v\)
3. Initialize \(L\) to consist of the single element \(s\)
4. While \(L\) is not empty
   a. Pop the node \(u\) at the head of \(L\)
   b. Consider each edge \((u, v)\) incident on \(u\)
   c. If \(\text{Discovered}[v] = \text{false}\) then
      i. Set \(\text{Discovered}[v] = \text{true}\)
      ii. Add edge \((u, v)\) to the tree \(T\)
      iii. Push \(v\) to the back of \(L\)
   d. Endif
5. Endwhile
Using a Queue in BFS

- Maintain an array Discovered and set $\text{Discovered}[v] = true$ as soon as the algorithm sees $v$.
- Maintain all the layers in a single queue $L$.

**BFS($s$):**
- Set $\text{Discovered}[s] = true$
- Set $\text{Discovered}[v] = false$, for all other nodes $v$
- Initialize $L$ to consist of the single element $s$
- While $L$ is not empty
  - Pop the node $u$ at the head of $L$
  - Consider each edge $(u, v)$ incident on $u$
  - If $\text{Discovered}[v] = false$ then
    - Set $\text{Discovered}[v] = true$
    - Add edge $(u, v)$ to the tree $T$
    - Push $v$ to the back of $L$
  - Endif
- Endwhile
Using a Queue in BFS

- Maintain an array Discovered and set Discovered[\(v\)] = \text{true} as soon as the algorithm sees \(v\).
- Maintain all the layers in a single queue \(L\).

**BFS(s):**

Set Discovered[\(s\)] = \text{true}

Set Discovered[\(v\)] = false, for all other nodes \(v\)

Initialize \(L\) to consist of the single element \(s\)

While \(L\) is not empty

- Pop the node \(u\) at the head of \(L\)
- Consider each edge \((u, v)\) incident on \(u\)
  - If Discovered[\(v\)] = false then
    - Set Discovered[\(v\)] = \text{true}
    - Add edge \((u, v)\) to the tree \(T\)
    - Push \(v\) to the back of \(L\)
  - Endif

Endwhile
Using a Queue in BFS

- Maintain an array Discovered and set Discovered[v] = true as soon as the algorithm sees v.
- Maintain all the layers in a single queue L.

BFS(s):
- Set Discovered[s] = true
- Set Discovered[v] = false, for all other nodes v
- Initialize L to consist of the single element s
- While L is not empty
  - Pop the node u at the head of L
  - Consider each edge (u, v) incident on u
    - If Discovered[v] = false then
      - Set Discovered[v] = true
      - Add edge (u, v) to the tree T
      - Push v to the back of L
  - Endif
- Endwhile
Using a Queue in BFS

- Maintain an array `Discovered` and set \( \text{Discovered}[v] = \text{true} \) as soon as the algorithm sees \( v \).
- Maintain all the layers in a single queue \( L \).

BFS(\( s \)):

1. Set `Discovered[s] = true`
2. Set `Discovered[v] = false`, for all other nodes \( v \)
3. Initialize \( L \) to consist of the single element \( s \)

While \( L \) is not empty:

   1. Pop the node \( u \) at the head of \( L \)
   2. Consider each edge \((u, v)\) incident on \( u \)
   3. If `Discovered[v] = false` then
      1. Set `Discovered[v] = true`
      2. Add edge \((u, v)\) to the tree \( T \)
      3. Push \( v \) to the back of \( L \)

Endif

Endwhile
Using a Queue in BFS

- Maintain an array Discovered and set $\text{Discovered}[v] = \text{true}$ as soon as the algorithm sees $v$.
- Maintain all the layers in a single queue $L$.

### BFS(s):

1. Set $\text{Discovered}[s] = \text{true}$
2. Set $\text{Discovered}[v] = \text{false}$, for all other nodes $v$
3. Initialize $L$ to consist of the single element $s$
4. While $L$ is not empty
   - Pop the node $u$ at the head of $L$
   - Consider each edge $(u, v)$ incident on $u$
   - If $\text{Discovered}[v] = \text{false}$ then
     - Set $\text{Discovered}[v] = \text{true}$
     - Add edge $(u, v)$ to the tree $T$
   - Push $v$ to the back of $L$
5. Endwhile
Using a Queue in BFS

- Maintain an array Discovered and set Discovered[v] = true as soon as the algorithm sees v.
- Maintain all the layers in a single queue L.

BFS(s):
Set Discovered[s] = true
Set Discovered[v] = false, for all other nodes v
Initialize L to consist of the single element s
While L is not empty
   Pop the node u at the head of L
   Consider each edge (u, v) incident on u
   If Discovered[v] = false then
      Set Discovered[v] = true
      Add edge (u, v) to the tree T
      Push v to the back of L
   Endif
Endwhile

- Simple to modify this procedure to keep track of layer numbers as well.
Using a Queue in BFS

- Maintain an array `Discovered` and set `Discovered[v] = true` as soon as the algorithm sees `v`.
- Maintain all the layers in a single queue `L`.

BFS(s):
- Set `Discovered[s] = true`
- Set `Discovered[v] = false`, for all other nodes `v`
- Initialize `L` to consist of the single element `s`
- While `L` is not empty
  - Pop the node `u` at the head of `L`
  - Consider each edge `(u, v)` incident on `u`
  - If `Discovered[v] = false`
    - Set `Discovered[v] = true`
    - Add edge `(u, v)` to the tree `T`
    - Push `v` to the back of `L`
- Endif
- Endwhile
- Simple to modify this procedure to keep track of layer numbers as well. Store the pair `(u, l_u)`, where `l_u` is the index of the layer containing `u`. 
Using a Queue in BFS

- Maintain an array Discovered and set $\text{Discovered}[v] = true$ as soon as the algorithm sees $v$.
- Maintain all the layers in a single queue $L$.

**BFS($s$):**
- Set $\text{Discovered}[s] = true$
- Set $\text{Discovered}[v] = false$, for all other nodes $v$
- Initialize $L$ to consist of the single element $s$
- While $L$ is not empty
  - Pop the node $u$ at the head of $L$
  - Consider each edge $(u, v)$ incident on $u$
    - If $\text{Discovered}[v] = false$ then
      - Set $\text{Discovered}[v] = true$
      - Add edge $(u, v)$ to the tree $T$
      - Push $v$ to the back of $L$
  - Endif
- Endwhile

Simple to modify this procedure to keep track of layer numbers as well. Store the pair $(u, l_u)$, where $l_u$ is the index of the layer containing $u$.

Claim: More formally: If BFS($s$) pops $(v, l_v)$ from $L$ immediately after it pops $(u, l_u)$, then either $l_v = l_u$ or $l_v = l_u + 1$. 

T. M. Murali August 27, 29 2018 Review of Priority Queues and Graph Searches
Analysis of BFS Implementation

BFS(s):
Set Discovered[s] = true
Set Discovered[v] = false, for all other nodes v
Initialize L to consist of the single element s
While L is not empty
    Pop the node u at the head of L
    Consider each edge (u, v) incident on u
    If Discovered[v] = false then
        Set Discovered[v] = true
        Add edge (u, v) to the tree T
        Push v to the back of L
    Endif
Endwhile

How many times is a node popped from L?
Analysis of BFS Implementation

BFS(s):

- Set Discovered[s] = true
- Set Discovered[v] = false, for all other nodes v
- Initialize L to consist of the single element s

While L is not empty

- Pop the node u at the head of L
- Consider each edge (u, v) incident on u
- If Discovered[v] = false then
  - Set Discovered[v] = true
  - Add edge (u, v) to the tree T
  - Push v to the back of L

Endif

Endwhile

- How many times is a node popped from L? Exactly once.
Analysis of BFS Implementation

BFS(s):
Set $\text{Discovered}[s] = \text{true}$
Set $\text{Discovered}[v] = \text{false}$, for all other nodes $v$
Initialize $L$ to consist of the single element $s$
While $L$ is not empty
  Pop the node $u$ at the head of $L$
  Consider each edge $(u, v)$ incident on $u$
  If $\text{Discovered}[v] = \text{false}$ then
    Set $\text{Discovered}[v] = \text{true}$
    Add edge $(u, v)$ to the tree $T$
    Push $v$ to the back of $L$
  Endif
Endwhile

- How many times is a node popped from $L$? Exactly once.
- Time used by for loop for a node $u$: $O(n_u)$ time.

Total time for all for loops: $\sum_{u \in G} O(n_u) = O(m) \text{ time.}$
Maintaining layer information: $O(1)$ time per node.
Total time is $O(n + m)$. 

T. M. Murali August 27, 29 2018
Analysis of BFS Implementation

BFS(s):

Set Discovered[s] = true
Set Discovered[v] = false, for all other nodes v
Initialize L to consist of the single element s
While L is not empty
   Pop the node u at the head of L
   Consider each edge (u, v) incident on u
   If Discovered[v] = false then
      Set Discovered[v] = true
      Add edge (u, v) to the tree T
      Push v to the back of L
   Endif
Endwhile

- How many times is a node popped from L? Exactly once.
- Time used by for loop for a node u: $O(n_u)$ time.
Analysis of BFS Implementation

BFS(s):

Set Discovered[s] = true
Set Discovered[v] = false, for all other nodes v
Initialize L to consist of the single element s
While L is not empty
    Pop the node u at the head of L
    Consider each edge (u, v) incident on u
    If Discovered[v] = false then
        Set Discovered[v] = true
        Add edge (u, v) to the tree T
        Push v to the back of L
    Endif
Endwhile

- How many times is a node popped from L? Exactly once.
- Time used by for loop for a node u: \(O(n_u)\) time.
- Total time for all for loops: \(\sum_{u \in G} O(n_u) = O(m)\) time.
- Maintaining layer information:
Analysis of BFS Implementation

BFS(s):
Set Discovered[s] = true
Set Discovered[v] = false, for all other nodes v
Initialize L to consist of the single element s
While L is not empty
    Pop the node u at the head of L
    Consider each edge (u, v) incident on u
    If Discovered[v] = false then
        Set Discovered[v] = true
        Add edge (u, v) to the tree T
        Push v to the back of L
    Endif
Endwhile

- How many times is a node popped from L? Exactly once.
- Time used by for loop for a node u: \(O(n_u)\) time.
- Total time for all for loops: \(\sum_{u \in G} O(n_u) = O(m)\) time.
- Maintaining layer information: \(O(1)\) time per node.
- Total time is \(O(n + m)\).
Recursive DFS to Stack-Based DFS

DFS(u):
   Mark u as "Explored" and add u to R
   For each edge (u, v) incident to u
     If v is not marked "Explored" then
       Recursively invoke DFS(v)
     Endif
   Endfor

- Procedure has “tail recursion”: recursive call is the last step.
Recursive DFS to Stack-Based DFS

DFS(u):

Mark u as "Explored" and add u to R

For each edge (u,v) incident to u

If v is not marked "Explored" then

  Recursively invoke DFS(v)

Endif

Endfor

- Procedure has “tail recursion”: recursive call is the last step.
- Can replace the recursion by an iteration: use a stack to explicitly implement the recursion.
Analysing DFS

`DFS(s):`

1. Initialize `S` to be a stack with one element `s`
2. While `S` is not empty
   1. Take a node `u` from `S`
   2. If `Explored[u] = false`
      1. Set `Explored[u] = true`
      2. For each edge `(u, v)` incident to `u`
         1. Add `v` to the stack `S`
   3. Endfor
3. Endif
4. Endwhile

- How many times is a node's adjacency list scanned?
Analysing DFS

DFS(s):
   Initialize S to be a stack with one element s
   While S is not empty
      Take a node u from S
      If Explored[u] = false then
         Set Explored[u] = true
         For each edge (u, v) incident to u
            Add v to the stack S
      Endfor
   Endif
Endwhile

- How many times is a node's adjacency list scanned? Exactly once.
Analysing DFS

DFS(s):
Initialize S to be a stack with one element s
While S is not empty
    Take a node u from S
    If Explored[u] = false then
        Set Explored[u] = true
        For each edge (u, v) incident to u
            Add v to the stack S
    Endfor
Endif
Endwhile

- How many times is a node’s adjacency list scanned? Exactly once.
- The total amount of time to process edges incident on node u’s is
Analysing DFS

DFS(s):

Initialize S to be a stack with one element s
While S is not empty
    Take a node u from S
    If Explored[u] = false then
        Set Explored[u] = true
        For each edge (u, v) incident to u
            Add v to the stack S
    Endfor
Endif
Endwhile

- How many times is a node’s adjacency list scanned? Exactly once.
- The total amount of time to process edges incident on node u’s is $O(n_u)$.
- The total running time of the algorithm is $O(n + m)$. 

T. M. Murali
August 27, 29 2018
Review of Priority Queues and Graph Searches
Analysing DFS

DFS(s):
  Initialize S to be a stack with one element s
  While S is not empty
    Take a node u from S
    If Explored[u] = false then
      Set Explored[u] = true
      For each edge (u, v) incident to u
        Add v to the stack S
    Endfor
  Endif
Endwhile

- How many times is a node’s adjacency list scanned? Exactly once.
- The total amount of time to process edges incident on node u’s is $O(n_u)$.
- The total running time of the algorithm is $O(n + m)$. 