Last Time

- Introduced Graphs
Today

- Traversing a Graph
- A shortest path algorithm
We will use this graph as an example throughout today’s class.
Graph Operations

- for $v \in V$ – iterate over all vertices
- for $e \in E$ – iterate over all edges
- for $e \in \text{edges}(v)$ – iterate over all edges with $v$ as an endpoint
- for $v_j \in \text{adjacent}(v_i)$ – iterate over all vertices that are adjacent to $v_i$
Reachability

Input: a directed or undirected graph \( G = (V, E) \), and a source node \( s \).
Output: the set of nodes in \( G \) reachable from \( s \)

Search(G,s)

\[
R = \{s\} \\
\textbf{while} \text{ there is an edge } e \text{ from } R \text{ to } V-R \text{ do} \\
\quad \text{let } e = (u,v) \\
\quad R = R \cup \{v\} \\
\quad \text{parent}[v] = u \\
\textbf{end while} \\
\textbf{return} \quad R
\]
Given a Graph $G = (V,E)$.
At any time during a search algorithm we can partition the set of vertices into three sets.

- $R$ – the set of visited nodes
- $V-R$ – the set of nodes that haven’t been visited yet
- active or fringe vertices – those nodes that have edges from $R$ to $V-R$

Choosing which active node to expand, and which edge to follow, differentiates different search algorithms.

The book colors these sets of nodes do differentiate them.

- processed nodes - black
- active nodes - grey
- unreached nodes - white
Generic Search Algorithms
Depth-First Traversal

**Depth-First Search** – select the active node for processing by selecting the “most-recent” node first. Use a Stack.
**Depth-First Search** – select the active node for processing by selecting the “most-recent” node first. Use a Stack.

**DFS(G,s)**

```latex
\text{for } u \in V \text{ do} \\
\quad \text{mark}[u] = 0; \text{parent}[u] = \emptyset \\
\text{end for} \\
\text{Recursive-DFS}(s)
```

**Recursive-DFS(u)**

```latex
\text{mark}[u] = 1 \\
\text{for } v \in \text{adjacent}(u) \text{ do} \\
\quad \text{if } \text{mark}[v] = 1 \text{ then} \\
\quad\quad \text{parent}[v] = u; \\
\quad\quad \text{DFS}(v) \\
\quad\text{end if} \\
\text{end for}
```
Depth-First Search runtime

DFS runtime
Depth-First Search runtime

DFS runtime

Initialization Runtime?
Recursion Runtime?
Depth-First Search runtime

DFS runtime

Initialization Runtime?
Recursion Runtime?

Initialization = $O(V)$
Recursion = $O(E)$
Total DFS Runtime = $O(V+E)$
Breadth-First Search – select the active node for processing by selecting the “earliest” node first. Use a Queue.
Breadth-First Search – select the active node for processing by selecting the “earliest” node first. Use a Queue.

BFS(G,s)

for v ∈ V-{s} do
    parent[v] = ∅; mark[v] = 0; d[v] = 0
end for
parent[s] = ∅; mark[s] = 1; d[s] = 0
Q = {s}
while Q ≠ ∅ do
    u = Dequeue(Q)
    for v ∈ adjacent(u) do
        if mark[v] = 0 then
            mark[v] = 1; parent[v] = u; d[v] = d[u] + 1;
            Enqueue(Q,v)
        end if
    end for
end while
Runtime of Breadth-First-Search

BFS(G,s)

<table>
<thead>
<tr>
<th>Code Block</th>
</tr>
</thead>
<tbody>
<tr>
<td>for v ∈ V-{s} do</td>
</tr>
<tr>
<td>parent[v] = ∅; mark[v] = 0; d[v] = 0</td>
</tr>
<tr>
<td>end for</td>
</tr>
<tr>
<td>parent[s] = ∅; mark[s] = 1; d[s] = 0</td>
</tr>
<tr>
<td>Q = {s}</td>
</tr>
<tr>
<td>while Q ≠ ∅ do</td>
</tr>
<tr>
<td>u = Dequeue(Q)</td>
</tr>
<tr>
<td>for v ∈ adjacent(u) do</td>
</tr>
<tr>
<td>if mark[v] = 0 then</td>
</tr>
<tr>
<td>mark[v] = 1; parent[v] = u; d[v] = d[u] + 1;</td>
</tr>
<tr>
<td>Enqueue(Q,v)</td>
</tr>
<tr>
<td>end if</td>
</tr>
<tr>
<td>end for</td>
</tr>
<tr>
<td>end while</td>
</tr>
</tbody>
</table>
Runtime of Breadth-First-Search

BFS(G,s)

\[
\text{for } v \in V-\{s\} \text{ do}
\]
\[
\text{parent}[v] = \emptyset; \text{mark}[v] = 0; d[v] = 0
\]
\[
\text{end for}
\]
\[
\text{parent}[s] = \emptyset; \text{mark}[s] = 1; d[s] = 0
\]
\[
Q = \{s\}
\]
\[
\text{while } Q \neq \emptyset \text{ do}
\]
\[
\text{u = Dequeue}(Q)
\]
\[
\text{for } v \in \text{adjacent}(u) \text{ do}
\]
\[
\text{if } \text{mark}[v] = 0 \text{ then}
\]
\[
\text{mark}[v] = 1; \text{parent}[v] = u; d[v] = d[u] + 1;
\]
\[
\text{Enqueue}(Q,v)
\]
\[
\text{end if}
\]
\[
\text{end for}
\]
\[
\text{end while}
\]

\[O(V+E)\]
Search Trees

DFS and BFS create **Search Trees**

Prove that DFS and BFS construct **Directed Acyclic Graphs**. Show that

- The resulting graph is a directed Graph
- The search traversal cannot have cycles

These are **Spanning Trees** or **Spanning Forests** if there are multiple connected components.
Define distance\((u,v)\) as the minimum length (in edges) of a path from \(u\) to \(v\).

Claim: \(d[v] = \text{distance}(s,v)\) for all nodes \(v \in V\)

Proof: Need to show that

- \(d[v] \leq \text{distance}(s,v)\)
- \(d[v] \geq \text{distance}(s,v)\)
Proof: Case 1) \( d[v] \geq \text{distance}(s,v) \)

A path with length \( d[v] \) can be reconstructed by traversing the parents of \( v \) until \( s \) is reached. Since \( \text{distance}(s,v) \) is the length of the minimum path, \( d[v] \) must be at least as large as \( \text{distance}(s,v) \). This logic can be demonstrated with greater rigor using induction.

- Base Case: \( d[s] \geq \text{distance}(s,s) \).
- Inductive step: for some vertex \( u \) adjacent to a fringe node \( v \).
  - \( d[v] = d[u] + 1 \)
  - \( d[v] \geq \text{distance}(s,u) + 1 \)
  - \( \text{distance}(s,v) \leq \text{distance}(s,u) + 1 \)
  - \( d[v] \geq \text{distance}(s,v) \)
Distance in a Graph

**Proof:** Case 2) \( d[v] \leq \text{distance}(s,v) \)
Assume not. Assume \( d(v) > \text{distance}(s,v) \). Show that this can never be true. Let \( u \) be the node immediately preceding \( v \) on the shortest path from \( s \) to \( v \)

\[
d[v] > \text{distance}(s,v) = \text{distance}(s,u) + 1 = d[u] + 1
\]

Consider what happens when \( u \) is dequeued from \( Q \).

1. \( v \) was **unvisited** – Then \( d[v] \) is set to \( d[u] + 1 \). Contradiction
2. \( v \) was visited – Then \( v \) was removed from \( Q \) earlier, and \( d[v] < d[u] \). Contradiction.
3. \( v \) was a fringe node, in the queue – Then it was enqueued by \( w \), **before** \( u \) where \( d[w] \leq d[u] \), and \( d[v] \) was set to \( d[w] + 1 \). Thus \( d[v] = d[w] + 1 \leq d[u] + 1 \). Contradiction

Therefore, \( d[v] \leq \text{distance}(s,v) \)

- Since \( d[v] \leq \text{distance}(s,v) \) **and** \( d[v] \geq \text{distance}(s,v) \), \( d[v] = \text{distance}(s,v) \)
We can use the same structure of the BFS to calculate the shortest path between two points in an undirected graph. Now distance(u,v) is $\sum$ weights of edges on the shortest path from u to v. Rather than expanding the nodes in order in a queue. We will expand the closest node first.
Example of Dijkstra’s Algorithm

Find the minimum length from source $s$ to any node $v \in V$. 
Find the minimum length from source $s$ to any node $v \in V$. 
Example of Dijkstra’s Algorithm

Find the minimum length from source $s$ to any node $v \in V$.
Find the minimum length from source $s$ to any node $v \in V$. 
Find the minimum length from source $s$ to any node $v \in V$. 

![Diagram of Dijkstra's Algorithm](image-url)
Example of Dijkstra’s Algorithm

Find the minimum length from source $s$ to any node $v \in V$. 
Find the minimum length from source $s$ to any node $v \in V$. 
Find the minimum length from source $s$ to any node $v \in V$. 
Example of Dijkstra’s Algorithm

Find the minimum length from source $s$ to any node $v \in V$. 

\begin{array}{c}
\text{s:0} \quad 1 \quad 2 \quad 3 \\
\text{a:1} \quad \text{c:2} \quad \text{d:1} \quad \text{f:2} \quad \text{i:3} \\
\text{e:3} \quad \text{h:4} \quad \text{j:5}
\end{array}
Example of Dijkstra’s Algorithm

Find the minimum length from source $s$ to any node $v \in V$. 
Dijkstra's Algorithm pseudocode

Dijkstra(G, s)

for v ∈ V - {s} do
  mark[v] = 0; d[v] = ∞
end for

d[s] = 0
MinHeap H = {s}
while H ≠ ∅ do
  u = ExtractMin(H)
  mark[u] = 1
  for e ∈ edges(u) do
    (u, v) = e
    if mark[v] = 0 then
      if d[u] + weight[e] < d[v] then
        d[v] = d[u] + weight[e];
        parent[v] = u
      end if
    end if
    Insert(H, v)
  end for
end while
Proof of Dijkstra’s Algorithm

The Proof of Dijsktra’s algorithm has the same structure as the Proof that \( d[v] = \text{distance}(s,v) \).

Need to Show

- \( d[v] \geq \text{ShortestPath}(s, v) \)
- \( d[v] \leq \text{ShortestPath}(s, v) \)

Rather than incrementing by single values, you increment by edge weights.
Define Connected Components

Connected Components

All of the nodes within a connected component are **reachable** from every other node in the connected component.

\( \text{Connected}(u,v) \) if there exists a path from \( u \) to \( v \)

- Reflexive: \( \text{Connected}(u,u) \)
- Symmetric: \( \text{Connected}(u,v) = \text{Connected}(v,u) \)
- Transitive: \( \text{Connected}(u,v) \) and \( \text{Connected}(v,w) \) then \( \text{Connected}(u,w) \)
Identify Connected Components

Components(G)

\[
\text{numComps} = 0
\]
\[
\text{for } v \in V \text{ do}
\]
\[
\text{mark}[v] = 0; \text{parent}[v] = \emptyset
\]
\[
\text{end for}
\]
\[
\text{for } v \in V \text{ do}
\]
\[
\text{if } \text{mark}[v] = 0 \text{ then}
\]
\[
\text{numComps} = \text{numComps} + 1
\]
\[
\text{DFSC}(v, \text{numComps})
\]
\[
\text{end if}
\]
\[
\text{end for}
\]

DFSC(v, c)

\[
\text{mark}[v] = 1; \text{comp}[v] = c
\]
\[
\text{for } u \in \text{adjacent}(v) \text{ do}
\]
\[
\text{if } \text{mark}[u] = 0 \text{ then}
\]
\[
\text{parent}[u] = v
\]
\[
\text{DFSC}(u, c)
\]
\[
\text{end if}
\]
\[
\text{end for}
\]
To detect cycles using DFS.

- Construct a DFS spanning Tree.
- If there are any **back edges** in the Graph, it contains a cycle
  - A **back edge** connects a node at some depth $d$ (in the DFS tree), to a node at some depth $d' < d$. 
Cycle detection using DFS
Cycle detection using DFS

Graph of cycle detection using DFS:
- a:0 → e:3 → h:4
- c:2 → d:3 → f:2 → j:5
- b:1 → i:2

The graph shows a cycle as indicated by the arrows connecting the nodes.
Cycle detection using DFS

New Graph
Cycle detection using DFS
Cycle detection using DFS

Back edge from $f$ to $b$ indicates the presence of a cycle.
Homework 9 is Posted.

Next time (11/17)
- Strongly Connected Components.
- Greedy Algorithms for finding minimum spanning trees
  - Kruskal’s
  - Prim’s