Graphs
Goals

- Introduction and Motivation
- Representations
Why do we care about graphs?
Many Applications

- Social Networks – Facebook
- Video Games - Motion Graphs
- Machine Learning/AI
- Delivery Networks/Scheduling – UPS?
- Computer Vision – Image Segmentation
- ...

...
Graphs

• Graphs represent relationships or connections
• Superset of trees (i.e., a tree is a restricted form of a graph):
  – A tree has more restrictive relationships and topology:
    • Each node has a single predecessor—its parent
    • There is a single, unique path from the root to any node
    • No cycles
    • Example: less than or greater than in a binary search tree
  – A graph represents general relationships:
    • Each node may have many predecessors
    • There may be multiple paths (or no path) from one node to another
    • Can have cycles or loops
    • Examples: airline flight connections, friends, algorithmic flow, etc.
A graph is composed of \textit{vertices} and \textit{edges}.

\textbf{Vertices (also called nodes):}
- Represent objects, states (i.e., conditions or configurations), positions, or simply just place holders
- Set \{v_1, v_2, ..., v_n\}: each vertex is unique \(\rightarrow\) no two vertices represent the same object/state

\textbf{Edges (also called arcs):}
- An edge \((v_i, v_j)\) between two vertices indicates that they are directly related, connected, etc.
- Can be either \textit{directed} or \textit{undirected}
- Can be either \textit{weighted} (or labeled) or \textit{unweighted}
- If there is an edge from \(v_i\) to \(v_j\), then \(v_j\) is a \textit{neighbor} of \(V_j\) (if the edge is undirected then \(v_i\) and \(v_j\) are neighbors or each other)
## Graphs: Types of Edges

<table>
<thead>
<tr>
<th></th>
<th>Undirected</th>
<th>Directed</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Unweighted</strong></td>
<td><img src="image" alt="Unweighted Undirected" /></td>
<td><img src="image" alt="Unweighted Directed" /></td>
</tr>
<tr>
<td><strong>Weighted</strong></td>
<td><img src="image" alt="Weighted Undirected" /></td>
<td><img src="image" alt="Weighted Directed" /></td>
</tr>
<tr>
<td></td>
<td><img src="image" alt="Weighted Undirected" /></td>
<td><img src="image" alt="Weighted Directed" /></td>
</tr>
</tbody>
</table>
Graphs: What kinds of questions can we ask?

• Is A reachable from B?
• What nodes are reachable from A?
• What’s the shortest path from A to B?
• Is the graph completely connected?
• How many arcs are between A & B?

etc...
Graphs: How do we represent/store them?
### Adjacency Matrix

**O(v^2) space**

By convention, a vertex is usually connected to itself (though, this is not always the case).

Stores only the edges → more space efficient for sparse graph: **O(V + E)**

where sparse means relatively few edges

### Edge List

**O(v+e) space**

Pendleton:  \{Pueblo, Phoenix\}

Pensacola:  \{Phoenix\}

Peoria: \{Pueblo, Pittsburgh\}

Phoenix: \{Pueblo, Peoria, Pittsburgh\}

Pierre: \{Pendleton\}

Pittsburgh: \{Pensacola\}

Princeton: \{Pittsburgh\}

Pueblo: \{Pierre\}
### Adjacency Matrix

\[ O(v^2) \text{ space} \]

<table>
<thead>
<tr>
<th>City</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
</tr>
</thead>
<tbody>
<tr>
<td>0: Pendleton</td>
<td>?</td>
<td>0</td>
<td>0</td>
<td>13</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>22</td>
</tr>
<tr>
<td>1: Pensacola</td>
<td>0</td>
<td>?</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>2: Peoria</td>
<td>0</td>
<td>0</td>
<td>?</td>
<td>0</td>
<td>0</td>
<td>8</td>
<td>0</td>
<td>13</td>
</tr>
<tr>
<td>3: Phoenix</td>
<td>0</td>
<td>0</td>
<td>43</td>
<td>?</td>
<td>0</td>
<td>16</td>
<td>0</td>
<td>90</td>
</tr>
<tr>
<td>4: Pierre</td>
<td>7</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>?</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>5: Pittsburgh</td>
<td>0</td>
<td>10</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>?</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>6: Princeton</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>5</td>
<td>?</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>7: Pueblo</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>22</td>
<td>0</td>
<td>0</td>
<td>?</td>
</tr>
</tbody>
</table>

### Edge List

\[ O(v+e) \text{ space} \]

- **Pendleton**: \{Pueblo,22\}, \{Phoenix,13\}
- **Pensacola**: \{Phoenix,1\}
- **Peoria**: \{Pueblo,13\}, \{Pittsburgh,8\}
- **Phoenix**: \{Pueblo,90\}, \{Peoria,43\}, \{Pittsburgh,16\}
- **Pierre**: \{Pendleton,7\}
- **Pittsburgh**: \{Pensacola,10\}
- **Princeton**: \{Pittsburgh,5\}
- **Pueblo**: \{Pierre,22\}
O(V+E)

Dominated by vertices

Dominated by edges
Your Turn

• Complete Worksheet #40 Graph Representations (on your own!)
Single Source Reachability - Edge List Representation
Question

- What nodes are reachable from Peoria?
- Let’s assume an edge-list representation
Exhaustive Search Algorithm

- Technique in which the algorithm systematically processes all vertices/edges of the graph in order to answer a question
findReachable (graph g, vertex start) {
    create a set of reachable vertices, initially empty. call this r.
    create a container for vertices known to be reachable (but not yet explored). call this c
    add start vertex to container c
    while the container c is not empty {
        remove first entry from the container c, assign to v
        if v is not already in the set of reachable vertices r {
            add v to the reachable set r
            add the neighbors of v, not already reachable, to the container c
        }
    }
    return r
• Let’s use a Stack as our container

• Basic algorithm:
  
  Initialize set of *reachable* vertices and add $v_i$ to a stack
  
  While stack is not empty
    
    Get and remove (pop) last vertex $v$ from stack
    
    if vertex $v$ is not in reachable,
      
      add it to reachable
    
    For all neighbors, $v_j$, of $v$, *if $v_j$ is NOT in reachable*
      
      add to stack
What cities are reachable from peoria? [Just for repeatability, when I push neighbors on the stack, I do so in alphabetical order)

<table>
<thead>
<tr>
<th>Stack(top-&gt;bot)</th>
<th>Reachable</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 Peoria</td>
<td>{}</td>
</tr>
</tbody>
</table>

![Diagram showing the reachability from Peoria to other cities]
What cities are reachable from Peoria?

<table>
<thead>
<tr>
<th>Stack(top-&gt;bot)</th>
<th>Reachable</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Peoria</td>
</tr>
<tr>
<td></td>
<td>{}</td>
</tr>
<tr>
<td>1</td>
<td>Pueblo, Pittsburgh Peoria</td>
</tr>
</tbody>
</table>

Diagram:
- Peoria
- Pittsburgh
- Princeton
- Pensacola
- Phoenix
- Pueblo
- Pierre
- Pendleton
- Peoria
<table>
<thead>
<tr>
<th>Stack(top-&gt;bot)</th>
<th>Reachable</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>peoria, {}</td>
</tr>
<tr>
<td>1</td>
<td>pueblo, pittsburgh, peoria</td>
</tr>
<tr>
<td>2</td>
<td>pierre, pittsburgh, peoria, pueblo</td>
</tr>
</tbody>
</table>

The diagram illustrates the reachability from the single source Stack. For each step in the stack, the reachable cities are shown in the table. The arrows in the diagram indicate the connections between cities.
Single-Source Reachability: Stack

<table>
<thead>
<tr>
<th>Stack(top-&gt;bot)</th>
<th>Reachable</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>peoria</td>
</tr>
<tr>
<td>1</td>
<td>pueblo, pittsburgh</td>
</tr>
<tr>
<td>2</td>
<td>pierre, pittsburgh</td>
</tr>
<tr>
<td>3</td>
<td>pendleton, pittsburgh</td>
</tr>
</tbody>
</table>

Diagram:
- Peoria
- Pueblo
- Pierre
- Pittsburgh
- Phoenix
- Pensacola
- Princeton
- Pendleton

Reachable:
- 0: peoria
- 1: pueblo, pittsburgh
- 2: pierre, pittsburgh
- 3: pendleton, pittsburgh

Reachable:
- 0: {}
- 1: peoria
- 2: peoria, pueblo
- 3: peoria, pueblo, pierre
## Single-Source Reachability: Stack

<table>
<thead>
<tr>
<th>Stack(top-&gt;bot)</th>
<th>Reachable</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>peoria {}</td>
</tr>
<tr>
<td>1</td>
<td>pueblo, pittsburgh peoria</td>
</tr>
<tr>
<td>2</td>
<td>pierre, pittsburgh peoria, pueblo</td>
</tr>
<tr>
<td>3</td>
<td>pendleton, pittsburgh peoria, pueblo, pierre</td>
</tr>
<tr>
<td>4</td>
<td>phoenix, pittsburgh peoria, pueblo, pierre, pendleton</td>
</tr>
</tbody>
</table>
## Single-Source Reachability: Stack

<table>
<thead>
<tr>
<th>Stack(top-&gt;bot)</th>
<th>Reachable</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 peoria</td>
<td>{}</td>
</tr>
<tr>
<td>1 pueblo, pittsburgh</td>
<td>peoria</td>
</tr>
<tr>
<td>2 pierre, pittsburgh</td>
<td>peoria, pueblo</td>
</tr>
<tr>
<td>3 pendleton, pittsburgh</td>
<td>peoria, pueblo, pierre</td>
</tr>
<tr>
<td>4 phoenix, pittsburgh</td>
<td>peoria, pueblo, pierre, pendleton</td>
</tr>
<tr>
<td>5 pittsburgh, pittsburgh</td>
<td>peoria, pueblo, pierre, pendleton, phoenix</td>
</tr>
</tbody>
</table>

![Graph showing reachability between cities]
### Single-Source Reachability: Stack

<table>
<thead>
<tr>
<th>Stack(top-&gt;bot)</th>
<th>Reachable</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>peoria</td>
</tr>
<tr>
<td>1</td>
<td>pueblo, pittsburgh</td>
</tr>
<tr>
<td>2</td>
<td>pierre, pittsburgh</td>
</tr>
<tr>
<td>3</td>
<td>pendleton, pittsburgh</td>
</tr>
<tr>
<td>4</td>
<td>phoenix, pittsburgh</td>
</tr>
<tr>
<td>5</td>
<td>pittsburgh, pittsburgh</td>
</tr>
<tr>
<td>6</td>
<td>pensacola, pittsburgh</td>
</tr>
</tbody>
</table>

The table shows the reachable cities for each stack operation. The diagram visualizes the connection between cities.
## Single-Source Reachability: Stack

<table>
<thead>
<tr>
<th>Stack(top→bot)</th>
<th>Reachable</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>peoria</td>
</tr>
<tr>
<td>1</td>
<td>pueblo, pittsburgh</td>
</tr>
<tr>
<td>2</td>
<td>pierre, pittsburgh</td>
</tr>
<tr>
<td>3</td>
<td>pendleton, pittsburgh</td>
</tr>
<tr>
<td>4</td>
<td>phoenix, pittsburgh</td>
</tr>
<tr>
<td>5</td>
<td>pittsburgh, pittsburgh</td>
</tr>
<tr>
<td>6</td>
<td>pensacola, pittsburgh</td>
</tr>
<tr>
<td>7</td>
<td>pittsburgh</td>
</tr>
</tbody>
</table>

![Graph of reachability](graph.png)
## Single-Source Reachability: Stack

<table>
<thead>
<tr>
<th>Stack(top-&gt;bot)</th>
<th>Reachable</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td><code>peoria</code></td>
</tr>
<tr>
<td>1</td>
<td><code>pueblo, pittsburgh</code> <code>peoria</code></td>
</tr>
<tr>
<td>2</td>
<td><code>pierre, pittsburgh</code> <code>peoria, pueblo</code></td>
</tr>
<tr>
<td>3</td>
<td><code>pendleton, pittsburgh</code> <code>peoria, pueblo, pierre</code></td>
</tr>
<tr>
<td>4</td>
<td><code>phoenix, pittsburgh</code> <code>peoria, pueblo, pierre, pendleton</code></td>
</tr>
<tr>
<td>5</td>
<td><code>pittsburgh, pittsburgh</code> <code>peoria, pueblo, pierre, pendleton, phoenix</code></td>
</tr>
<tr>
<td>6</td>
<td><code>pensacola, pittsburgh</code> <code>peoria, pueblo, pierre, pendleton, phoenix, pittsburgh</code></td>
</tr>
<tr>
<td>7</td>
<td><code>pittsburgh</code> <code>peoria, pueblo, pierre, pendleton, phoenix, pittsburgh, pensacola</code></td>
</tr>
<tr>
<td>8</td>
<td><code>{}</code></td>
</tr>
</tbody>
</table>

![Graph Diagram]
Implementation

- Reachable: Array of integers: \( R[i] = 0 / 1 \)
  - alternatively, use a hashmap
- Stack ADT: dynamic array, LL, etc.
- Graph Representation (edge list):
  - Dynamic array of LinkedLists (assuming nodes are mapped to integers)
  - alternatively...Map
    - key = name of node
    - value = linked list of neighbors
Your Turn

• Worksheet 41 – on your own
• Something to think about…
  – What happens if we use a Queue instead of a Stack to hold the unexplored neighbors?
Stack vs. Queue: Reachable from Alex

- Alex
  - Abner
    - Abigail
    - Adam
  - Adela
  - Angela
    - Alice
      - Agnes
    - Allen
    - Audrey
      - Arthur
DFS and BFS – Edge List Representation
Application: Maze Solving

• Can we reach F from S?
• Easily represent a maze as a graph
Application: Maze Path Finding Example

- Single-Source Reachability

For consistency (order in which neighbors are pushed onto the stack)
Application: Maze Path Finding Example
Application: Maze Path Finding Example

STACK

4d
3e
Application: Maze Path Finding Example

STACK

5d
3e
Application: Maze Path Finding Example

STACK
5c
3e
Application: Maze Path Finding Example

STACK

5b
4c
3e
Application: **Maze Path Finding Example**

STACK

- 5a
- 4b
- 4c
- 3e

**Diagram:**

- Maze grid labeled with numbers 1 to 5 horizontally and vertically.
- Red area labeled as 1.
- Numbered cells: 6, 5, 4, 1.
- Directions indicated by arrows: 1, 2, 3, 4.
Application: Maze Path Finding Example

STACK

4a
4b
4c
3e
Application: Maze Path Finding Example

STACK

4b
4c
3e

DEAD END!!
Application: Maze Path Finding Example

STACK

```
4c
3e
```
Application: Maze Path Finding Example

STACK

3c
3e
Application: Maze Path Finding Example

STACK
3b
2c
3e
Application: Maze Path Finding Example
What happens if we use a Queue?
Application: Maze Path Finding Example
Application: Maze Path Finding Example

```
<table>
<thead>
<tr>
<th>a</th>
<th>b</th>
<th>c</th>
<th>d</th>
<th>e</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
</tr>
</tbody>
</table>
```

```
QUEUE
```

```
4e
```

Diagram showing a maze with walls and a queue.
Application: Maze Path Finding Example
Application: Maze Path Finding Example

- Maze Path Finding Example

Queue:
- 3d
- 2e
- 4d
Application: Maze Path Finding Example
Application: Maze Path Finding Example
Application: Maze Path Finding Example

Diagram of a maze with numbered paths and a queue with items 1e and 5d.
Application: Maze Path Finding Example

![Maze Diagram]

- **.Queue:**
  - 5c
  - 1e

- Maze Path Finding Example

- **QUEUE:**
  - 5c
  - 1e
Application: Maze Path Finding Example

QUEUE
1d
5c
Application: Maze Path Finding Example

[Diagram of a maze with numbers and letters, showing a path from a to e and a queue with 5b, 4c, 1d]
Application: Maze Path Finding

Depth-First (Stack)

Breadth-First (Queue)
DFS vs. BFS

• DFS like a single person working a maze
• BFS like a wave flowing through a maze
• DFS can take an unfortunate route and have to backtrack a long way, and multiple times
• DFS can get lucky and find the solution very quickly
• BFS may not find it as quickly, but will *always* find it
• Because BFS first checks all paths of length 1, then of length 2, then of length 3, etc....it’s guaranteed to find a path containing the least steps from start to goal (if it exists)
• What if there’s one infinite path....DFS may go down it...but BFS will not get stuck in it
Time Complexity: DFS/BFS

- O(V+E) time in both cases
  - Key observations: Init is O(v) & Edge list scanned once for each vertex, so scans E edges
  - Initialize set of reachable vertices and add v_i to a stack
  - While stack is not empty
    - Get and remove (pop) last vertex v from stack
    - if vertex v is not in reachable, add it to reachable
    - For all neighbors, v_j, of v, if v_j is NOT in reachable add to stack
Space Complexity: DFS/BFS

• What about space?
  – BFS must store all vertices on a Queue at least once
  – DFS uses a Stack and stores all vertices on the stack at least once
  – In both cases, O(V) space best case
  – In practice, BFS may take up more space because it looks at all paths of a specific length at once. e.g. if search a deep tree, BFS will store lots of long potential paths
DFS vs. BFS: In practice

- Depends on the problem
  - If there are some very deep paths, DFS could spend a lot of time going down them
  - If it’s a very broad/wide tree, BFS could require a lot of memory on the queue
  - If you need to find a shortest path, BFS guarantees is
  - Are solutions near top of the tree?
    - BFS may find it more quickly
    - e.g. Search a family tree for distant ancestor who was alive a long time ago
  - Are solutions at the leaves
    - DFS can find it more quickly
    - e.g. Search a family tree for someone who’s still alive
Implementation Variations

• Can easily do DFS recursively
• Can avoid “Reachable” in both DFS/BFS by instead, adding a **color** field to each node
  – white: unvisited
  – gray: considered (on queue, stack)
  – black: reachable
• Store additional information to use in solving other important graph problems
Dijkstra’s Algorithm – Edge List Representation
Weighted Graphs Representation: Edge List

What’s reachable AND what is the cheapest cost to get there?

Pendleton: \{\text{Pueblo:8, Phoenix:4}\}
Pensacola: \{\text{Phoenix:5}\}
Peoria: \{\text{Pueblo:3, Pittsburgh:5}\}
Phoenix: \{\text{Pueblo:3, Peoria:4, Pittsburgh:10}\}
Pierre: \{\text{Pendleton:2}\}
Pittsburgh: \{\text{Pensacola:4}\}
Princeton: \{\text{Pittsburgh:2}\}
Pueblo: \{\text{Pierre:3}\}
Dijkstra’s Algorithm

Initialize map of reachable vertices, and add source vertex, $v_i$, to a priority queue with distance zero.

While priority queue is not empty:

- Get min from priority queue and assign to $v$.
- If $v$ is not in reachable:
  - add $v$ with given cost to map of reachable vertices.
- For all neighbors, $v_j$, of $v$
  - If $v_j$ is not in set of reachable vertices, combine cost of reaching $v$ with cost to travel from $v$ to $v_j$, and add to priority queue.
Example: What is the distance from Pierre?

<table>
<thead>
<tr>
<th>Pqueue</th>
<th>Reachable</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>pierre(0)</td>
</tr>
</tbody>
</table>
Example: What is the distance from Pierre?

- Pierre:
  - Pendleton: 2
  - Pueblo: 3
  - Pensacola: 5
  - Phoenix: 10
  - Princeton: 4
  - Pittsburgh: 5

- Pendleton:
  - Pierre: 2
  - Pueblo: 8

- Pueblo:
  - Pierre: 3
  - Piesburgh: 3
  - Peoria: 10

- Pensacola:
  - Pueblo: 5
  - Peoria: 4

- Phoenix:
  - Pueblo: 3

- Princeton:
  - Pittsburgh: 4

- Pittsburgh:
  - Peoria: 5
  - Princeton: 2

Pqueue | Reachable
-------|-----------
0      | pierre(0) |
1      | pendleton(2) | pierre(0)
Example: What is the distance from Pierre?

<table>
<thead>
<tr>
<th>Pqueue</th>
<th>Reachable</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>pierre(0)</td>
</tr>
<tr>
<td>1</td>
<td>pendleton(2)</td>
</tr>
<tr>
<td>2</td>
<td>phoenix(6), pueblo(10)</td>
</tr>
</tbody>
</table>

Diagram:
- Pierre
- Pendleton
- Pueblo
- Phoenix
- Peoria
- Pittsburgh
- Princeton
- Pensacola

Distances:
- Pendleton to Pierre: 2
- Pierre to Peoria: 3
- Peoria to Pittsburgh: 5
- Pittsburgh to Princeton: 2
- Princeton to Pensacola: 4
- Pensacola to Phoenix: 5
- Phoenix to Pierre: 6
- Pierre to Pueblo: 3
- Pueblo to Phoenix: 4
- Phoenix to Pittsburgh: 10
Example: **What is the distance from Pierre?**

```
<table>
<thead>
<tr>
<th>Pqueue</th>
<th>Reachable</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>pierre(0)</td>
</tr>
<tr>
<td>1</td>
<td>pendleton(2) pierre(0)</td>
</tr>
<tr>
<td>2</td>
<td>phoenix(6), pueblo(10)</td>
</tr>
<tr>
<td>3</td>
<td>pueblo(9), peoria(10),</td>
</tr>
<tr>
<td></td>
<td>pittsburgh(16), pueblo(10)</td>
</tr>
</tbody>
</table>
```

**NOTE:** Reachable is only showing the latest node added to the collection!
Example: **What is the distance from Pierre?**

<table>
<thead>
<tr>
<th>Pqueue</th>
<th>Reachable</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>pierre(0)</td>
</tr>
<tr>
<td>1</td>
<td>pendleton(2)</td>
</tr>
<tr>
<td>2</td>
<td>phoenix(6),</td>
</tr>
<tr>
<td></td>
<td>pueblo(10)</td>
</tr>
<tr>
<td>3</td>
<td>pueblo(9),</td>
</tr>
<tr>
<td></td>
<td>peoria(10),</td>
</tr>
<tr>
<td></td>
<td>pittsburgh(16),</td>
</tr>
<tr>
<td></td>
<td>pueblo(10)</td>
</tr>
<tr>
<td>4</td>
<td>peoria(10),</td>
</tr>
<tr>
<td></td>
<td>pittsburgh(16),</td>
</tr>
<tr>
<td></td>
<td>pueblo(10)</td>
</tr>
</tbody>
</table>
Example: What is the distance from Pierre?

<table>
<thead>
<tr>
<th>Pqueue</th>
<th>Reachable</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>pierre(0)</td>
</tr>
<tr>
<td>1</td>
<td>pendleton(2)</td>
</tr>
<tr>
<td>2</td>
<td>phoenix(6), pueblo(10)</td>
</tr>
<tr>
<td>3</td>
<td>pueblo(9), peoria(10), pittsburgh(16), pueblo(10)</td>
</tr>
<tr>
<td>4</td>
<td>peoria(10), pittsburgh(16), pueblo(10)</td>
</tr>
<tr>
<td>5</td>
<td>pueblo(10), pittsburgh(15), pittsburgh(16), peoria(10)</td>
</tr>
</tbody>
</table>
Example: What is the distance from Pierre?

<table>
<thead>
<tr>
<th>Pqueue</th>
<th>Reachable</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>pierre(0)</td>
</tr>
<tr>
<td>1</td>
<td>pendleton(2)</td>
</tr>
<tr>
<td>2</td>
<td>phoenix(6), pueblo(10)</td>
</tr>
<tr>
<td>3</td>
<td>pueblo(9), peoria(10), pittsburgh(16), pueblo(10)</td>
</tr>
<tr>
<td>4</td>
<td>peoria(10), pittsburgh(16), pueblo(10)</td>
</tr>
<tr>
<td>5</td>
<td>pueblo(10), pittsburgh(15), pittsburgh(16), pueblo(10)</td>
</tr>
<tr>
<td>6</td>
<td>pittsburgh(15), pittsburgh(16)</td>
</tr>
</tbody>
</table>
Example: **What is the distance from Pierre?**

<table>
<thead>
<tr>
<th>Pqueue</th>
<th>Reachable</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>pierre(0)</td>
</tr>
<tr>
<td>1</td>
<td>pendleton(2) pierre(0)</td>
</tr>
<tr>
<td>2</td>
<td>phoenix(6), pueble(10) pendleton(2)</td>
</tr>
<tr>
<td>3</td>
<td>pueblo(9), peoria(10), pittsburgh(16), pueblo(10) phoenix(6)</td>
</tr>
<tr>
<td>4</td>
<td>peoria(10), pittsburgh(16), pueblo(10) pueblo(9)</td>
</tr>
<tr>
<td>5</td>
<td>pueblo(10) pittsburgh(15), pittsburgh(16), pueblo(10) peoria(10)</td>
</tr>
<tr>
<td>6</td>
<td>pittsburgh(15), pittsburgh(16) --</td>
</tr>
<tr>
<td>7</td>
<td>pittsburgh(16), pensacola(19) pittsburgh(15)</td>
</tr>
</tbody>
</table>
Example: What is the distance from Pierre?

<table>
<thead>
<tr>
<th>Pqueue</th>
<th>Reachable</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>pierre(0)</td>
</tr>
<tr>
<td>1</td>
<td>pendleton(2)</td>
</tr>
<tr>
<td>2</td>
<td>phoenix(6), pueblo(10)</td>
</tr>
<tr>
<td>3</td>
<td>pueblo(9), peoria(10), pittsburgh(16), pueblo(10)</td>
</tr>
<tr>
<td>4</td>
<td>peoria(10), pittsburgh(16), pueblo(10)</td>
</tr>
<tr>
<td>5</td>
<td>pueblo(10)</td>
</tr>
<tr>
<td>6</td>
<td>pittsburgh(15), pittsburgh(16)</td>
</tr>
<tr>
<td>7</td>
<td>pittsburgh(16), pensacola(19)</td>
</tr>
<tr>
<td>8</td>
<td>pensacola(19)</td>
</tr>
</tbody>
</table>
Example: **What is the distance from Pierre?**

<table>
<thead>
<tr>
<th>Pqueue</th>
<th>Reachable</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>pierre(0)</td>
</tr>
<tr>
<td>1</td>
<td>pendleton(2)</td>
</tr>
<tr>
<td>2</td>
<td>phoenix(6), pueblo(10)</td>
</tr>
<tr>
<td>3</td>
<td>pueblo(9), peoria(10), pittsburgh(16), pueblo(10)</td>
</tr>
<tr>
<td>4</td>
<td>peoria(10), pittsburgh(16), pueblo(10)</td>
</tr>
<tr>
<td>5</td>
<td>pueblo(10), pittsburgh(15), pittsburgh(16),</td>
</tr>
<tr>
<td>6</td>
<td>pittsburgh(15), pittsburgh(16)</td>
</tr>
<tr>
<td>7</td>
<td>pittsburgh(16), pensacola(19)</td>
</tr>
<tr>
<td>8</td>
<td>pensacola(19)</td>
</tr>
<tr>
<td>9</td>
<td>{}</td>
</tr>
</tbody>
</table>
Dijkstra’s

• Cost-first search
• Always explores the next node with the CUMULATIVE least cost
• Our implementation: $O(V+E \log E)$
  – Key observation: Inner loop runs at most $E$ times
  – Time to add/rem from pqueue is bounded by $\log E$
    since all neighbors, or edges, can potentially be on the pqueue
  – $V$ comes from the initialization of reachable
    • assume reachable is an array or hashTable
• $O(V + E \log E)$
  – Key observation: Inner loop runs at most $E$ times
  – Time to add/rem from priority queue is bounded by $\log E$ since all neighbors, or edges, can potentially be on the priority queue

Initialize map of reachable vertices, and add source vertex $v_i$, to a priority queue with distance zero
While priority queue is not empty
  Getmin from priority queue and assign to $v$
  If $v$ is not in reachable
    add $v$ with given cost to map of reachable vertices
  For all neighbors, $v_j$, of $v$
    If $v_j$ is not is set of reachable vertices, combine cost of reaching $v$ with cost to travel from $v$ to $v_j$, and add to priority queue
• Same code, three different ADTs result in three kinds of searches!!!
  – DFS (Stack)
  – BFS (Queue)
  – Dijkstra's Cost-First Search (Pqueue)

Initialize set of *reachable* vertices and add \( v_i \) to [stack, queue, pqueue]
While [stack, queue, pqueue] is not empty
  Get and remove [top, first, min] vertex \( v \) from [stack, queue, pqueue]
  if vertex \( v \) is not in reachable,
    add it to reachable
    For all neighbors, \( v_j \), of \( v \), not already in reachable
      add to [stack, queue, pqueue]
      (in case of pqueue, add with cumulative cost)
Implementation of Dijkstra’s

- Pqueue: dynamic array heap
- Reachable:
  - Array indexed by node num
  - map: name, distance
  - hashMap
- Graph Representation: edge list with weights [map of maps]
  - Key: Node name
  - Value: Map of Neighboring nodes
    - Key: node name of one of the neighbors
    - Value: weight to that neighbor
Your Turn

• Complete Worksheet #42: Dijkstra’s Algorithm