CSCI 104
Graph Representation and Traversals

Mark Redekopp
David Kempe
Sandra Batista
GRAPH REPRESENTATIONS
Graph Notation

- A **graph** is a collection of vertices (or nodes) and edges that connect vertices.

<table>
<thead>
<tr>
<th>V</th>
<th>E</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>(a,c)</td>
</tr>
<tr>
<td>b</td>
<td>(a,e)</td>
</tr>
<tr>
<td>c</td>
<td>(b,h)</td>
</tr>
<tr>
<td>d</td>
<td>(b,c)</td>
</tr>
<tr>
<td>e</td>
<td>(c,e)</td>
</tr>
<tr>
<td>f</td>
<td>(c,d)</td>
</tr>
<tr>
<td>g</td>
<td>(c,g)</td>
</tr>
<tr>
<td>h</td>
<td>(d,f)</td>
</tr>
<tr>
<td></td>
<td>(e,f)</td>
</tr>
<tr>
<td></td>
<td>(f,g)</td>
</tr>
<tr>
<td></td>
<td>(g,h)</td>
</tr>
</tbody>
</table>

- Let $V$ be the set of vertices
- Let $E$ be the set of edges
- Let $|V|$ or $n$ refer to the number of vertices
- Let $|E|$ or $m$ refer to the number of edges

$|V|=n=8$  \hspace{1cm} $|E|=m=11$
Graphs in the Real World

- Social networks
- Computer networks / Internet
- Path planning
- Interaction diagrams
- Bioinformatics
Basic Graph Representation

- Can simply store edges in a list
  - Unsorted
  - Sorted

\[ V = \{a, b, c, d, e, f, g, h\} \]
\[ E = \{\{(a, c), (a, e), (b, h), (b, c), (c, e), (c, d), (c, g), (d, f), (e, f), (f, g), (g, h)\} \]

\[ |V| = n = 8 \]
\[ |E| = m = 11 \]
Graph ADT

• What operations would you want to perform on a graph?
  • addVertex() : Vertex
  • addEdge(v1, v2)
  • getAdjacencies(v1) : List<Vertices>
    – Returns any vertex with an edge from v1 to itself
  • removeVertex(v)
  • removeEdge(v1, v2)
  • edgeExists(v1, v2) : bool

```cpp
#include<iostream>
using namespace std;

template<typename V, typename E>
class Graph{
  // Perfect for templating the data associated with a vertex and edge as V and E
};
```
More Common Graph Representations

- Graphs are really just a list of lists
  - List of vertices each having their own list of adjacent vertices
- Alternatively, sometimes graphs are also represented with an adjacency matrix
  - Entry at \((i,j)\) = 1 if there is an edge between vertex \(i\) and \(j\), 0 otherwise

### Adjacency Lists

<table>
<thead>
<tr>
<th>Vertex</th>
<th>Adjacent Vertices</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>c, e</td>
</tr>
<tr>
<td>b</td>
<td>c, h</td>
</tr>
<tr>
<td>c</td>
<td>a, b, d, e, g</td>
</tr>
<tr>
<td>d</td>
<td>c, f</td>
</tr>
<tr>
<td>e</td>
<td>a, c, f</td>
</tr>
<tr>
<td>f</td>
<td>d, e, g</td>
</tr>
<tr>
<td>g</td>
<td>c, f, h</td>
</tr>
<tr>
<td>h</td>
<td>b, g</td>
</tr>
</tbody>
</table>

### Adjacency Matrix Representation

<table>
<thead>
<tr>
<th></th>
<th>a</th>
<th>b</th>
<th>c</th>
<th>d</th>
<th>e</th>
<th>f</th>
<th>g</th>
<th>h</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>b</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>c</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>d</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>e</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>f</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>g</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>h</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>
Graph Representations

- Let $|V| = n = \# \text{ of vertices}$ and $|E| = m = \# \text{ of edges}$
- Adjacency List Representation
  - $O(_______________)$ memory storage
  - Existence of an edge requires $O(_______________)$ time
- Adjacency Matrix Representation
  - $O(_______________)$ storage
  - Existence of an edge requires $O(_________)$ lookup

How would you express this using the ADTs you've learned?

<table>
<thead>
<tr>
<th>List of Vertices</th>
<th>Adjacency Lists</th>
</tr>
</thead>
<tbody>
<tr>
<td>a, b, c, d, e, g</td>
<td>a, b, c, d, e, g</td>
</tr>
<tr>
<td>c, e</td>
<td>c, e</td>
</tr>
<tr>
<td>c, h</td>
<td>c, h</td>
</tr>
<tr>
<td>a, b, d, e, g</td>
<td>a, b, d, e, g</td>
</tr>
<tr>
<td>c, f</td>
<td>c, f</td>
</tr>
<tr>
<td>a, c, f</td>
<td>a, c, f</td>
</tr>
<tr>
<td>d, e, g</td>
<td>d, e, g</td>
</tr>
<tr>
<td>c, f, h</td>
<td>c, f, h</td>
</tr>
<tr>
<td>b, g</td>
<td>b, g</td>
</tr>
</tbody>
</table>

Adjacency Matrix Representation

<table>
<thead>
<tr>
<th></th>
<th>a</th>
<th>b</th>
<th>c</th>
<th>d</th>
<th>e</th>
<th>f</th>
<th>g</th>
<th>h</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>b</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>c</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>d</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>e</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>f</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>g</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>h</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>
Graph Representations

- Let $|V| = n = \# \text{ of vertices}$ and $|E| = m = \# \text{ of edges}$

- **Adjacency List Representation**
  - $O(|V| + |E|)$ memory storage
  - Define **degree** to be the number of edges incident on a vertex (deg(a) = 2, deg(c) = 5, etc.)
  - Existence of an edge requires searching the adjacency list in $O(\text{deg}(v))$

- **Adjacency Matrix Representation**
  - $O(|V|^2)$ storage
  - Existence of an edge requires $O(1)$ lookup (e.g. matrix[i][j] == 1)

### Adjacency Lists

<table>
<thead>
<tr>
<th>List of Vertices</th>
<th>Adjacency Lists</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>c, e</td>
</tr>
<tr>
<td>b</td>
<td>c, h</td>
</tr>
<tr>
<td>c</td>
<td>a, b, d, e, g</td>
</tr>
<tr>
<td>d</td>
<td>c, f</td>
</tr>
<tr>
<td>e</td>
<td>a, c, f</td>
</tr>
<tr>
<td>f</td>
<td>d, e, g</td>
</tr>
<tr>
<td>g</td>
<td>c, f, h</td>
</tr>
<tr>
<td>h</td>
<td>b, g</td>
</tr>
</tbody>
</table>

### Adjacency Matrix Representation

<table>
<thead>
<tr>
<th></th>
<th>a</th>
<th>b</th>
<th>c</th>
<th>d</th>
<th>e</th>
<th>f</th>
<th>g</th>
<th>h</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>b</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>c</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>d</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>e</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>f</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>g</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>h</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>
Graph Representations

- Can 'a' get to 'b' in two hops?
- Adjacency List
  - For each neighbor of a...
  - Search that neighbor's list for b
- Adjacency Matrix
  - Take the dot product of row a & column b

Adjacency List

<table>
<thead>
<tr>
<th></th>
<th>a</th>
<th>b</th>
<th>c</th>
<th>d</th>
<th>e</th>
<th>f</th>
<th>g</th>
<th>h</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>c,e</td>
<td>c,h</td>
<td>a,b,d,e,g</td>
<td>c,f</td>
<td>a,c,f</td>
<td>d,e,g</td>
<td>c,f,h</td>
<td>b,g</td>
</tr>
<tr>
<td>b</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>c</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>d</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>e</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>f</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>g</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>h</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Adjacency Matrix Representation

<table>
<thead>
<tr>
<th></th>
<th>a</th>
<th>b</th>
<th>c</th>
<th>d</th>
<th>e</th>
<th>f</th>
<th>g</th>
<th>h</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>b</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>c</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>d</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>e</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>f</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>g</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>h</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>
Graph Representations

- Can 'a' get to 'b' in two hops?
- Adjacency List
  - For each neighbor of a...
  - Search that neighbor's list for b
- Adjacency Matrix
  - Take the dot product of row a & column b

```java
int sum = 0;
for(int i=0; i < n; i++){
    sum += adj[src][i]*adj[i][dst];
}
if(sum > 0) // two-hop path exists
```

<table>
<thead>
<tr>
<th></th>
<th>a</th>
<th>b</th>
<th>c</th>
<th>d</th>
<th>e</th>
<th>f</th>
<th>g</th>
<th>h</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>b</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>c</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>d</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>e</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>f</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>g</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>h</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>
Directed vs. Undirected Graphs

- In the previous graphs, edges were **undirected** (meaning edges are 'bidirectional' or 'reflexive')
  - An edge \((u,v)\) implies \((v,u)\)
- In **directed** graphs, links are unidirectional
  - An edge \((u,v)\) does not imply \((v,u)\)
  - For Edge \((u,v)\): the **source** is \(u\), **target** is \(v\)
- For adjacency list form, you may need 2 lists per vertex for both predecessors and successors

Adjacency Matrix Representation

<table>
<thead>
<tr>
<th></th>
<th>a</th>
<th>b</th>
<th>c</th>
<th>d</th>
<th>e</th>
<th>f</th>
<th>g</th>
<th>h</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>b</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>c</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>d</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>e</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>f</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>g</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>h</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>
Directed vs. Undirected Graphs

- In directed graph with edge (src, tgt) we define
  - Successor(src) = tgt
  - Predecessor(tgt) = src

- Using an adjacency list representation may warrant two lists predecessors and successors

```
Adjacency Matrix Representation
Source       a b c d e f g h
Target       a 0 0 1 0 1 0 0 0
             b 0 0 0 0 0 0 0 1
             c 0 1 0 1 1 0 1 0
             d 0 0 0 0 0 1 0 0
             e 0 0 0 0 0 1 0 0
             f 0 0 0 0 0 0 0 0
             g 0 0 0 0 1 0 0 0
             h 0 0 0 0 0 0 1 0
```

List of Vertices

- a: c, e
- b: h
- c: b, d, e, g
- d: f
- e: f
- f: c, h
- g: f
- h: g

Succs (Outgoing) | Preds (Incoming)
--- | ---
Succs | Preds
Graph Runtime, \(|V| = n, |E| = m\)

<table>
<thead>
<tr>
<th>Operation vs Implementation for Edges</th>
<th>Add edge</th>
<th>Delete Edge</th>
<th>Test Edge</th>
<th>Enumerate edges for single vertex</th>
</tr>
</thead>
<tbody>
<tr>
<td>Unsorted array or Linked List</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Sorted array</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Adjacency List</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Adjacency Matrix</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
# Graph Runtime, $|V| = n, |E| = m$

<table>
<thead>
<tr>
<th>Operation vs Implementation for Edges</th>
<th>Add edge</th>
<th>Delete Edge</th>
<th>Test Edge</th>
<th>Enumerate edges for single vertex</th>
</tr>
</thead>
<tbody>
<tr>
<td>Unsorted array or Linked List</td>
<td>$\Theta(1)$</td>
<td>$\Theta(m)$</td>
<td>$\Theta(m)$</td>
<td>$\Theta(m)$</td>
</tr>
<tr>
<td>Sorted array</td>
<td>$\Theta(m)$</td>
<td>$\Theta(m)$</td>
<td>$\Theta(\log m)$ [if binary search used]</td>
<td>$\Theta(\log m) + \Theta(\deg(v))$ [if binary search used]</td>
</tr>
<tr>
<td>Adjacency List</td>
<td>Time to find List for a given vertex + $\Theta(1)$</td>
<td>Time to find List for a given vertex + $\Theta(\deg(v))$</td>
<td>Time to find List for a given vertex + $\Theta(\deg(v))$</td>
<td>Time to find List for a given vertex + $\Theta(\deg(v))$</td>
</tr>
<tr>
<td>Adjacency Matrix</td>
<td>$\Theta(1)$</td>
<td>$\Theta(1)$</td>
<td>$\Theta(1)$</td>
<td>$\Theta(v)$</td>
</tr>
</tbody>
</table>
Graph Algorithms
PAGERANK ALGORITHM
PageRank

- Consider the graph at the right
  - These could be webpages with links shown in the corresponding direction
  - These could be neighboring cities
- PageRank generally tries to answer the question:
  - If we let a bunch of people randomly "walk" the graph, what is the probability that they end up at a certain location (page, city, etc.) in the "steady-state"
- We could solve this problem through Monte-Carlo simulation (essentially the CS 103 PA5 or PA1 Coin-flipping or Zombie assignment...depending on semester)
  - Simulate a large number of random walkers and record where each one ends to build up an answer of the probabilities for each vertex
- But there are more efficient ways of doing it
PageRank

- Let us write out the adjacency matrix for this graph
- Now let us make a weighted version by normalizing based on the out-degree of each node
  - Ex. If you're at node B we have a 50-50 chance of going to A or E
- From this you could write a system of linear equations (i.e. what are the chances you end up at vertex I at the next time step, given you are at some vertex J now)
  - \( p_A = 0.5*p_B \)
  - \( p_B = p_C \)
  - \( p_C = p_A + p_D + 0.5*p_E \)
  - \( p_D = 0.5*p_E \)
  - \( p_E = 0.5*p_B \)
  - We also know: \( p_A + p_B + p_C + p_D + p_E = 1 \)
PageRank

- **System of Linear Equations**
  - \( pA = 0.5*pB \)
  - \( pB = pC \)
  - \( pC = pA + pD + 0.5*pE \)
  - \( pD = 0.5*pE \)
  - \( pE = 0.5*pB \)
  - We also know: \( pA + pB + pC + pD + pE = 1 \)

- If you know something about linear algebra, you know we can write these equations in matrix form as a linear system
  - \( Ax = y \)

---

**Weighted Adjacency Matrix**

<table>
<thead>
<tr>
<th></th>
<th>a</th>
<th>b</th>
<th>c</th>
<th>d</th>
<th>e</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>0</td>
<td>0.5</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>b</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>c</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0.5</td>
</tr>
<tr>
<td>d</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0.5</td>
</tr>
<tr>
<td>e</td>
<td>0</td>
<td>0.5</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

*Divide by \((a_{i,j})/\text{degree}(j)\)*

- \( pA = 0.5PB \)
- \( pB = pC \)
- \( pC = pA+pD+0.5*pE \)
- \( pD = 0.5*pE \)
- \( pE = 0.5*pB \)
PageRank

But remember we want the steady state solution
  - The solution where the probabilities don't change from one step to the next

So we want a solution to: \( \mathbf{A} \mathbf{p} = \mathbf{p} \)

We can:
  - Use a linear system solver (Gaussian elimination)
  - Or we can just seed the problem with some probabilities and then just iterate until the solution settles down

\[
\begin{pmatrix}
0 & 0.5 & 0 & 0 & 0 \\
0 & 0 & 1 & 0 & 0 \\
1 & 0 & 0 & 1 & 0.5 \\
0 & 0 & 0 & 0 & 0.5 \\
0 & 0.5 & 0 & 0 & 0 \\
\end{pmatrix}
\begin{pmatrix}
pA \\
pB \\
pC \\
pD \\
pE \\
\end{pmatrix}
= \begin{pmatrix}
pA \\
pB \\
pC \\
pD \\
pE \\
\end{pmatrix}
\]

Weighted Adjacency Matrix
[Divide by \((a_{i,j})/\text{degree}(j)\)]
Iterative PageRank

- But remember we want the steady state solution
  - The solution where the probabilities don't change from one step to the next
- So we want a solution to: \( Ap = p \)
- We can:
  - Use a linear system solver (Gaussian elimination)
  - Or we can just seed the problem with some probabilities and then just iterate until the solution settles down

\[
\begin{array}{cccc}
0 & 0.5 & 0 & 0 \\
0 & 0 & 1 & 0 \\
1 & 0 & 0 & 1 \ 0.5 \\
0 & 0 & 0 & 0 \ 0.5 \\
0 & 0 & 0 & 0 \\
\end{array}
\begin{array}{c}
\cdot \ 2 \\
\cdot \ 2 \\
\cdot \ 2 \\
\cdot \ 2 \\
\cdot \ 2 \\
\end{array}
\begin{array}{c}
0.2 \\
0.2 \\
0.2 \\
0.2 \\
0.2 \\
\end{array}
\begin{array}{c}
0.1 \\
0.2 \\
0.5 \\
0.1 \\
0.1 \\
\end{array}
\begin{array}{cccc}
0 & 0.5 & 0 & 0 \\
0 & 0 & 1 & 0 \\
1 & 0 & 0 & 1 \ 0.5 \\
0 & 0 & 0 & 0 \ 0.5 \\
0 & 0 & 0 & 0 \\
\end{array}
\begin{array}{c}
? \\
? \\
? \\
? \\
? \\
\end{array}
\begin{array}{c}
0.1507 \\
0.3078 \\
0.3126 \\
0.0783 \\
0.1507 \\
\end{array}
\]


\[
\begin{array}{cccc}
0 & 0.5 & 0 & 0 \\
0 & 0 & 1 & 0 \\
1 & 0 & 0 & 1 \ 0.5 \\
0 & 0 & 0 & 0 \ 0.5 \\
0 & 0 & 0 & 0 \\
\end{array}
\begin{array}{c}
\cdot \ 2 \\
\cdot \ 2 \\
\cdot \ 2 \\
\cdot \ 2 \\
\cdot \ 2 \\
\end{array}
\begin{array}{c}
0.1538 \\
0.3077 \\
0.3077 \\
0.0769 \\
0.1538 \\
\end{array}
\]

Step 1 Sol.  Step 2 Sol.  Actual PageRank Solution from solving linear system:
Additional Notes

- What if we change the graph and now D has no incoming links...what is its PageRank?
  - 0
- Most PR algorithms add a probability that someone just enters that URL (i.e. enters the graph at that node)
  - Usually define something called the damping factor, $\alpha$ (often chosen around 0.15)
  - Probability of randomly starting or jumping somewhere = $1-\alpha$
- So at each time step the next PR value for node $i$ is given as:
  
  $$Pr(i) = \frac{\alpha}{N} + (1 - \alpha) \sum_{j \in Pred(i)} \frac{Pr(j)}{OutDeg(j)}$$

- $N$ is the total number of vertices
- Usually run 30 or so update steps
- Start each $Pr(i) = 1/N$
In a Web Search Setting

• Given some search keywords we could find the pages that have that matching keywords
• We often expand that set of pages by including all successors and predecessors of those pages
  – Include all pages that are within a radius of 1 of the pages that actually have the keyword
• Now consider that set of pages and the subgraph that it induces
• Run PageRank on that subgraph

Full WebGraph

Page Hits (Contain keyword)

Expanded (Preds & Succs)

Induced Subgraph to run PageRank
Dijkstra's Algorithm

SINGLE-SOURCE SHORTEST PATH (SSSP)
SSSP

• Let us associate a 'weight' with each edge
  – Could be physical distance, cost of using the link, etc.

• Find the shortest path from a source node, 'a' to all other nodes

<table>
<thead>
<tr>
<th>List of Vertices</th>
<th>Adjacency Lists</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>(c,13),(e,4)</td>
</tr>
<tr>
<td>b</td>
<td>(c,5),(h,6)</td>
</tr>
<tr>
<td>c</td>
<td>(a,13),(b,5),(d,2),(e,8),(g,7)</td>
</tr>
<tr>
<td>d</td>
<td>(c,2),(f,1)</td>
</tr>
<tr>
<td>e</td>
<td>(a,4),(c,8),(f,3)</td>
</tr>
<tr>
<td>f</td>
<td>(d,1),(e,3),(g,4)</td>
</tr>
<tr>
<td>g</td>
<td>(c,7),(f,4),(h,14)</td>
</tr>
<tr>
<td>h</td>
<td>(b,6),(g,14)</td>
</tr>
</tbody>
</table>
SSSP

• What is the shortest distance from 'a' to all other vertices?
• How would you go about computing those distances?

![Graph with distances and adjacency lists]

<table>
<thead>
<tr>
<th>List of Vertices</th>
<th>Adjacency Lists</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>(c, 13), (e, 4)</td>
</tr>
<tr>
<td>b</td>
<td>(c, 5), (h, 6)</td>
</tr>
<tr>
<td>c</td>
<td>(a, 13), (b, 5), (d, 2), (e, 8), (g, 7)</td>
</tr>
<tr>
<td>d</td>
<td>(c, 2), (f, 1)</td>
</tr>
<tr>
<td>e</td>
<td>(a, 4), (c, 8), (f, 3)</td>
</tr>
<tr>
<td>f</td>
<td>(d, 1), (e, 3), (g, 4)</td>
</tr>
<tr>
<td>g</td>
<td>(c, 7), (f, 4), (h, 14)</td>
</tr>
<tr>
<td>h</td>
<td>(b, 6), (g, 14)</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Vert</th>
<th>Dist</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>0</td>
</tr>
<tr>
<td>b</td>
<td></td>
</tr>
<tr>
<td>c</td>
<td></td>
</tr>
<tr>
<td>d</td>
<td></td>
</tr>
<tr>
<td>e</td>
<td></td>
</tr>
<tr>
<td>f</td>
<td></td>
</tr>
<tr>
<td>g</td>
<td></td>
</tr>
<tr>
<td>h</td>
<td></td>
</tr>
</tbody>
</table>
Dijkstra's Algorithm

- Dijkstra's algorithm is similar to a BFS but pulls out the smallest distance vertex (from the source) rather than pulling vertices out in FIFO order (as in BFS).

- Maintain a data structure that you can identify shortly
  - We'll show it as a table of all vertices with their currently 'known' distance from the source.
    - Initially, a has dist=0
    - All others = infinite distance
Dijkstra's Algorithm

1. SSSP(G, s)
2. PQ = empty PQ
3. s.dist = 0; s.pred = NULL
4. PQ.insert(s)
5. For all v in vertices
6. if v != s then v.dist = inf; PQ.insert(v)
7. while PQ is not empty
8. v = min(); PQ.remove_min()
9. for u in neighbors(v)
10. w = weight(v, u)
11. if(v.dist + w < u.dist)
12. u.pred = v
13. u.dist = v.dist + w;
14. PQ.decreaseKey(u, u.dist)
Dijkstra's Algorithm

1. SSSP(G, s)
2. PQ = empty PQ
3. s.dist = 0; s.pred = NULL
4. PQ.insert(s)
5. For all v in vertices
6. if v != s then v.dist = inf; PQ.insert(v)
7. while PQ is not empty
8. v = min(); PQ.remove_min()
9. for u in neighbors(v)
10. w = weight(v,u)
11. if(v.dist + w < u.dist)
12. u.pred = v
13. u.dist = v.dist + w;
14. PQ.decreaseKey(u, u.dist)
Dijkstra's Algorithm

1. SSSP(G, s)
2. PQ = empty PQ
3. s.dist = 0; s.pred = NULL
4. PQ.insert(s)
5. For all v in vertices
6. if v !≠ s then v.dist = inf; PQ.insert(v)
7. while PQ is not empty
8. v = min(); PQ.remove_min()
9. for u in neighbors(v)
10. w = weight(v,u)
11. if(v.dist + w < u.dist)
12. u.pred = v
13. u.dist = v.dist + w;
14. PQ.decreaseKey(u, u.dist)
Dijkstra's Algorithm

1. SSSP(G, s)
2. PQ = empty PQ
3. s.dist = 0; s.pred = NULL
4. PQ.insert(s)
5. For all v in vertices
6. if v != s then v.dist = inf; PQ.insert(v)
7. while PQ is not empty
8. v = min(); PQ.remove_min()
9. for u in neighbors(v)
10. w = weight(v, u)
11. if(v.dist + w < u.dist)
12. u.pred = v
13. u.dist = v.dist + w;
14. PQ.decreaseKey(u, u.dist)
Dijkstra's Algorithm

1. SSSP(G, s)
2. PQ = empty PQ
3. s.dist = 0; s.pred = NULL
4. PQ.insert(s)
5. For all v in vertices
6. if v != s then v.dist = inf; PQ.insert(v)
7. while PQ is not empty
8. v = min(); PQ.remove_min()
9. for u in neighbors(v)
10. w = weight(v,u)
11. if(v.dist + w < u.dist)
12. u.pred = v
13. u.dist = v.dist + w;
14. PQ.decreaseKey(u, u.dist)
Dijkstra's Algorithm

1. SSSP(G, s)
2. PQ = empty PQ
3. s.dist = 0; s.pred = NULL
4. PQ.insert(s)
5. For all v in vertices
6. if v != s then v.dist = inf; PQ.insert(v)
7. while PQ is not empty
8. v = min(); PQ.remove_min()
9. for u in neighbors(v)
10. w = weight(v, u)
11. if(v.dist + w < u.dist)
12. u.pred = v
13. u.dist = v.dist + w;
14. PQ.decreaseKey(u, u.dist)
Dijkstra's Algorithm

1. SSSP(G, s)
2. PQ = empty PQ
3. s.dist = 0; s.pred = NULL
4. PQ.insert(s)
5. For all v in vertices
6. if v != s then v.dist = inf; PQ.insert(v)
7. while PQ is not empty
8. v = min(); PQ.remove_min()
9. for u in neighbors(v)
10. w = weight(v,u)
11. if(v.dist + w < u.dist)
12. u.pred = v
13. u.dist = v.dist + w;
14. PQ.decreaseKey(u, u.dist)
Dijkstra's Algorithm

1. SSSP(G, s)
2. PQ = empty PQ
3. s.dist = 0; s.pred = NULL
4. PQ.insert(s)
5. For all v in vertices
6. if v != s then v.dist = inf; PQ.insert(v)
7. while PQ is not empty
8. v = min(); PQ.remove_min()
9. for u in neighbors(v)
10. w = weight(v,u)
11. if(v.dist + w < u.dist)
12. u.pred = v
13. u.dist = v.dist + w;
14. PQ.decreaseKey(u, u.dist)
Dijkstra's Algorithm

1. SSSP(G, s)
2. PQ = empty PQ
3. s.dist = 0; s.pred = NULL
4. PQ.insert(s)
5. For all v in vertices
6. if v != s then v.dist = inf; PQ.insert(v)
7. while PQ is not empty
8. v = min(); PQ.remove_min()
9. for u in neighbors(v)
10. w = weight(v,u)
11. if(v.dist + w < u.dist)
12. u.pred = v
13. u.dist = v.dist + w;
14. PQ.decreaseKey(u, u.dist)
Another Example

• Try another example of Dijkstra's
Analysis

- What is the loop invariant? What can I say about the vertex I pull out from the PQ?
  - It is guaranteed that there is no shorter path to that vertex
  - UNLESS: negative edge weights

- Could use induction to prove
  - When I pull the first node out (it is the start node) it's weight has to be 0 and that is definitely the shortest path to itself
  - I then "relax" (i.e. decrease) the distance to neighbors it connects to and the next node I pull out would be the neighbor with the shortest distance from the start
    - Could there be shorter path to that node?
    - No, because any other path would use some other edge from the start which would have to have a larger weight
Dijkstra's Run-time Analysis

- What is the run-time of Dijkstra's algorithm?
- How many times do you execute the while loop on 8?
- How many total times do you execute the for loop on 10?

1. SSSP(G, s)
2. PQ = empty PQ
3. s.dist = 0; s.pred = NULL
4. PQ.insert(s)
5. For all v in vertices
6. if v != s then v.dist = inf;
7. PQ.insert(v)
8. while PQ is not empty
9. v = min(); PQ.remove_min()
10. for u in neighbors(v)
11. w = weight(v,u)
12. if(v.dist + w < u.dist)
13. u.pred = v
14. u.dist = v.dist + w;
15. PQ.decreaseKey(u, u.dist)
**Dijkstra's Run-time Analysis**

- What is the run-time of Dijkstra's algorithm?
- How many times do you execute the while loop on 8?
  - V total times because once you pull a node out each iteration that node's distance is guaranteed to be the shortest distance and will never be put back in the PQ
  - What does each call to `remove_min()` cost...
  - ...\(\log(V)\) [at most V items in PQ]
- How many total times do you execute the for loop on 10?
  - E total times: Visit each vertex's neighbors
  - Each iteration may call `decreaseKey()` which is \(\log(V)\)
- Total runtime = \(V\log(V) + E\log(V) = (V+E)\log(V)\)
  - This is usually dominated by \(E\log(V)\)
Tangent on Heaps/PQs

• Suppose min-heaps
  – Though everything we're about to say is true for max heaps but for increasing values
• We know insert/remove is log(n) for a heap
• What if we want to decrease a value already in the heap...
  – Example: Decrease 26 to 9
  – Could we find 26 easily?
    • No requires a linear search through the array/heap => O(n)
  – Once we find it could we adjust it easily?
    • Yes, just promote it until it is in the right location => O(log n)
• So currently decrease-key() would cost O(n) + O(log n) = O(n)
• Can we do better?
Tangent on Heaps/PQs

- Can we provide a decrease-key() that runs in \(O(\log n)\) and not \(O(n)\)?
  - Remember we’d have to first find then promote

- We need to know where items sit in the heap
  - Essentially we want to quickly know the location given the key/priority (i.e. Map key => location)
  - Unfortunately storing the heap as an array does just the opposite (maps location => key)

- What if we assigned each node a unique index [0 to n-1] and maintained an alternative map that did provide the reverse indexing
  - Then I could find where the key sits and then promote it

- If I used std::map to maintain the id => heap index map I still cannot achieve \(O(\log n)\) decreaseKey() time?
  - No! each promotion swap requires update your location and your parents
  - \(O(\log n)\) swaps each requiring lookup(s) in the location map \([O(\log n)]\) yielding \(O(\log^2(n))\)
Tangent on Heaps/PQs

- Am I out of luck then?
- No, try an array / hash map
  - $O(1)$ lookup
- Now each swap/promotion up the heap only costs $O(1)$ and thus I have:
  - Find => $O(1)$
    - Using an array (or hashmap)
  - Promote => $O(\log n)$
    - Bubble up at most $\log(n)$ levels with each level incurring $O(1)$ updates of locations in the hashmap
- Decrease-key() is an important operation in the next algorithm we'll look at
Tangent on Heaps/PQs - old

- Can we provide a decrease-key() that runs in $O(\log n)$ and not $O(n)$
  - Remember we'd have to first find then promote
- We need to know where items sit in the heap
  - Essentially we want to quickly know the location given the key/priority (i.e. Map key => location)
  - Unfortunately storing the heap as an array does just the opposite (maps location => key)
- What if we maintained an alternative map that did provide the reverse indexing
  - Then I could find where the key sits and then promote it
- If I keep that map as a balanced BST can I achieve $O(\log n)$ decreaseKey() time?
  - No! each promotion swap requires update your location and your parents
  - $O(\log n)$ swaps each requiring lookup(s) in the location map $[O(\log n)]$ yielding $O(\log^2(n))$
Tangent on Heaps/PQs -old

• Am I out of luck then?
• No, try a hash map
  – O(1) lookup
• Now each swap/promotion up the heap only costs O(1) and thus I have:
  – Find => O(1)
    • Using the hashmap
  – Promote => O(log n)
    • Bubble up at most log(n) levels with each level incurring O(1) updates of locations in the hashmap
• Decrease-key() is an important operation in the next algorithm we'll look at
A* Search Algorithm

**ALGORITHM HIGHLIGHT**
Search Methods

• Many systems require searching for goal states

  – Path Planning
    • Roomba Vacuum
    • Mapquest/Google Maps
    • Games!!

  – Optimization Problems
    • Find the optimal solution to a problem with many constraints
Search Applied to 8-Tile Game

• 8-Tile Puzzle
  – 3x3 grid with one blank space
  – With a series of moves, get the tiles in sequential order
  – Goal state:

HW6 Goal State

Goal State for these slides
Search Methods

• **Brute-Force Search**: When you don’t know where the answer is, just search all possibilities until you find it.

• **Heuristic Search**: A heuristic is a “rule of thumb”. An example is in a chess game, to decide which move to make, count the values of the pieces left for your opponent. Use that value to “score” the possible moves you can make.
  
  – Heuristics are not perfect measures, they are quick computations to give an approximation (e.g. may not take into account “delayed gratification” or “setting up an opponent”)
Brute Force Search

• Brute Force Search Tree
  – Generate all possible moves
  – Explore each move despite its proximity to the goal node
Heuristics

• Heuristics are “scores” of how close a state is to the goal (usually, lower = better)

• These scores must be easy to compute (i.e. simpler than solving the problem)

• Heuristics can usually be developed by simplifying the constraints on a problem

• Heuristics for 8-tile puzzle
  – # of tiles out of place
    • Simplified problem: If we could just pick a tile up and put it in its correct place
  – Total x-, y- distance of each tile from its correct location (Manhattan distance)
    • Simplified problem if tiles could stack on top of each other / hop over each other

<table>
<thead>
<tr>
<th># of Tiles out of Place = 3</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 8 3</td>
</tr>
<tr>
<td>4 5 6</td>
</tr>
<tr>
<td>2 7</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Total x-/y- distance = 6</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 8 3</td>
</tr>
<tr>
<td>4 5 6</td>
</tr>
<tr>
<td>2 7</td>
</tr>
</tbody>
</table>
Heuristic Search

• Heuristic Search Tree
  – Use total x-/y-distance (Manhattan distance) heuristic
  – Explore the lowest scored states
Caution About Heuristics

• Heuristics are just estimates and thus could be wrong
• Sometimes pursuing lowest heuristic score leads to a less-than optimal solution or even no solution
• Solution
  – Take # of moves from start (depth) into account
A-Star Algorithm

- Use a new metric to decide which state to explore/expand

Define
- \( h = \) heuristic score (same as always)
- \( g = \) number of moves from start it took to get to current state
- \( f = g + h \)

- As we explore states and their successors, assign each state its f-score and always explore the state with lowest f-score

- Heuristics should always underestimate the distance to the goal
  - If they do, A* guarantees optimal solutions
A-Star Algorithm

• Maintain 2 lists
  – Open list = Nodes to be explored (chosen from)
  – Closed list = Nodes already explored (already chosen)

• General A* Pseudocode

```python
open_list.push(Start State)
while(open_list is not empty)
  1. s ← remove min. f-value state from open_list
     (if tie in f-values, select one w/ larger g-value)
  2. Add s to closed list
  3a. if s = goal node then trace path back to start; STOP!
  3b. Generate successors/neighbors of s, compute their f values, and add them to open_list if they are not in the closed_list (so we don’t re-explore), or if they are already in the open list, update them if they have a smaller f value
```
Path-Planning w/ A* Algorithm

• Find optimal path from S to G using A*
  – Use heuristic of Manhattan (x-/y-) distance

```python
open_list.push(Start State)
while(open_list is not empty)
    1. s ← remove min. f-value state from open_list (if tie in f-values, select one w/ larger g-value)
    2. Add s to closed list
    3a. if s = goal node then
        trace path back to start; STOP!
    3b. else
        Generate successors/neighbors of s, compute their f-values, and add them to open_list if they are not in the closed_list (so we don’t re-explore), or if they are already in the open list, update them if they have a smaller f value
```

**If implementing this for a programming assignment, please see the slide at the end about alternate closed-list implementation**
Path-Planning w/ A* Algorithm

• Find optimal path from S to G using A*
  – Use heuristic of Manhattan (x-/y-) distance

**If implementing this for a programming assignment, please see the slide at the end about alternate closed-list implementation**

```plaintext
open_list.push(Start State)
while(open_list is not empty)
  1. s ← remove min. f-value state from open_list (if tie in f-values, select one w/ larger g-value)
  2. Add s to closed list
  3a. if s = goal node then
      trace path back to start; STOP!
  3b. else
      Generate successors/neighbors of s, compute their f-values, and add them to open_list if they are not in the closed_list (so we don’t re-explore), or if they are already in the open list, update them if they have a smaller f value
```
Path-Planning w/ A* Algorithm

- Find optimal path from S to G using A*
  - Use heuristic of Manhattan (x-/y-) distance

open_list.push(Start State)
while(open_list is not empty)
  1. s ← remove min. f-value state from
     open_list (if tie in f-values, select one w/ larger g-value)
  2. Add s to closed list
  3a. if s = goal node then
      trace path back to start; STOP!
  3b. else
      Generate successors/neighbors of s, compute their f-values, and add them to
      open_list if they are not in the closed_list (so we don’t re-explore), or if they are
      already in the open list, update them if they have a smaller f value
Path-Planning w/ A* Algorithm

• Find optimal path from S to G using A*
  – Use heuristic of Manhattan (x-/y-) distance

```
open_list.push(Start State)
while(open_list is not empty)
  1. s ← remove min. f-value state from
     open_list (if tie in f-values, select one w/ larger g-value)
  2. Add s to closed list
  3a. if s = goal node then
       trace path back to start; STOP!
  3b. else
       Generate successors/neighbors of s,
       compute their f-values, and add them to open_list if they are not in the closed_list
       (so we don’t re-explore), or if they are already in the open list, update them if they have a smaller f value
```
Path-Planning w/ A* Algorithm

- Find optimal path from S to G using A*
  - Use heuristic of Manhattan (x-/y-) distance

open_list.push(Start State)
while(open_list is not empty)
  1. s ← remove min. f-value state from open_list (if tie in f-values, select one w/ larger g-value)
  2. Add s to closed list
  3a. if s = goal node then
      trace path back to start; STOP!
  3b. else
      Generate successors/neighbors of s, compute their f-values, and add them to open_list if they are not in the closed_list (so we don’t re-explore), or if they are already in the open list, update them if they have a smaller f value
Path-Planning w/ A* Algorithm

- Find optimal path from S to G using A*
  - Use heuristic of Manhattan (x-/y-) distance

open_list.push(Start State)
while(open_list is not empty)
  1. $s \leftarrow$ remove min. $f$-value state from open_list (if tie in $f$-values, select one w/ larger $g$-value)
  2. Add $s$ to closed list
  3a. if $s$ = goal node then
      trace path back to start; STOP!
  3b. else
      Generate successors/neighbors of $s$, compute their $f$-values, and add them to open_list if they are not in the closed_list (so we don’t re-explore), or if they are already in the open list, update them if they have a smaller $f$ value
Path-Planning w/ A* Algorithm

- Find optimal path from S to G using A*
  - Use heuristic of Manhattan (x-/y-) distance

open_list.push(Start State)
while(open_list is not empty)

1. s ← remove min. f-value state from open_list (if tie in f-values, select one w/ larger g-value)
2. Add s to closed list
3a. if s = goal node then
    trace path back to start; STOP!
3b. else
    Generate successors/neighbors of s, compute their f-values, and add them to open_list if they are not in the closed_list (so we don’t re-explore), or if they are already in the open list, update them if they have a smaller f value
Path-Planning w/ A* Algorithm

• Find optimal path from S to G using A*
  – Use heuristic of Manhattan (x-/y-) distance

```plaintext
open_list.push(Start State)
while(open_list is not empty)
  1. s ← remove min. f-value state from open_list (if tie in f-values, select one w/ larger g-value)
  2. Add s to closed list
  3a. if s = goal node then trace path back to start; STOP!
  3b. else
      Generate successors/neighbors of s, compute their f-values, and add them to open_list if they are not in the closed_list (so we don’t re-explore), or if they are already in the open list, update them if they have a smaller f value
```
Path-Planning w/ A* Algorithm

• Find optimal path from S to G using A*
  – Use heuristic of Manhattan ($x$-$y$-) distance

open_list.push(Start State)
while(open_list is not empty)
  1. $s \leftarrow$ remove min. f-value state from open_list (if tie in f-values, select one w/ larger g-value)
  2. Add $s$ to closed list
  3a. if $s$ = goal node then
      trace path back to start; STOP!
  3b. else
      Generate successors/neighbors of $s$, compute their f-values, and add them to open_list if they are not in the closed_list (so we don’t re-explore), or if they are already in the open list, update them if they have a smaller f value
Path-Planning w/ A* Algorithm

• Find optimal path from S to G using A*
  – Use heuristic of Manhattan (x-/y-) distance

open_list.push(Start State)
while(open_list is not empty)
  1. s ← remove min. f-value state from open_list (if tie in f-values, select one w/ larger g-value)
  2. Add s to closed list
  3a. if s = goal node then trace path back to start; STOP!
  3b. else
     Generate successors/neighbors of s, compute their f-values, and add them to open_list if they are not in the closed_list (so we don’t re-explore), or if they are already in the open list, update them if they have a smaller f value
Path-Planning w/ A* Algorithm

- Find optimal path from S to G using A*
  - Use heuristic of Manhattan (x-/y-) distance

```
open_list.push(Start State)
while(open_list is not empty)
    1. s ← remove min. f-value state from
       open_list (if tie in f-values, select one w/ larger g-value)
    2. Add s to closed list
    3a. if s = goal node then
        trace path back to start; STOP!
    3b. else
        Generate successors/neighbors of s, compute their f-values, and add them to open_list if they are not in the closed_list (so we don’t re-explore), or if they are already in the open list, update them if they have a smaller f value
```
Path-Planning w/ A* Algorithm

• Find optimal path from S to G using A*
  – Use heuristic of Manhattan (x-/y-) distance

open_list.push(Start State)
while(open_list is not empty)
  1. s ← remove min. f-value state from open_list (if tie in f-values, select one w/ larger g-value)
  2. Add s to closed list
  3a. if s = goal node then
      trace path back to start; STOP!
  3b. else
      Generate successors/neighbors of s, compute their f-values, and add them to open_list if they are not in the closed_list (so we don’t re-explore), or if they are already in the open list, update them if they have a smaller f value
Path-Planning w/ A* Algorithm

• Find optimal path from S to G using A*
  – Use heuristic of Manhattan (x-/y-) distance

Open List

Closed List

open_list.push(Start State)
while(open_list is not empty)
  1. s ← remove min. f-value state from open_list (if tie in f-values, select one w/larger g-value)
  2. Add s to closed list
  3a. if s = goal node then
      trace path back to start; STOP!
  3b. else
      Generate successors/neighbors of s, compute their f-values, and add them to open_list if they are not in the closed_list (so we don’t re-explore), or if they are already in the open list, update them if they have a smaller f value
Path-Planning w/ A* Algorithm

- Find optimal path from S to G using A*
  - Use heuristic of Manhattan (x-/y-) distance

```python
open_list.push(Start State)
while(open_list is not empty)
    1. s ← remove min. f-value state from open_list (if tie in f-values, select one w/ larger g-value)
    2. Add s to closed list
    3a. if s = goal node then
        trace path back to start; STOP!
    3b. else
        Generate successors/neighbors of s, compute their f-values, and add them to open_list if they are not in the closed_list (so we don’t re-explore), or if they are already in the open list, update them if they have a smaller f value
```
A* and BFS

• BFS explores all nodes at a shorter distance from the start (i.e. $g$ value)
A* and BFS

• BFS explores all nodes at a shorter distance from the start (i.e. g value)
A* and BFS

• BFS is A* using just the g value to choose which item to select and expand
A* Analysis

• What data structure should we use for the open-list?
• What data structure should we use for the closed-list?
• What is the run time?
• Run time is similar to Dijkstra's algorithm...
  – We pull out each node/state once from the open-list so that incurs N*O(remove-cost)
  – We then visit each successor which is like O(E) and perform an insert or decrease operation which is like E*max(O(insert), O(decrease))
  – E = Number of potential successors and this depends on the problem and the possible solution space
  – For the tile puzzle game, how many potential boards are there?

```python
open_list.push(Start State)
while(open_list is not empty)
  1. s ← remove min. f-value state from open_list
     (if tie in f-values, select one w/ larger g-value)
  2. Add s to closed list
  3a. if s = goal node then trace path back to start; STOP!
  3b. Generate successors/neighbors of s, compute their f values, and add them to open_list if they are not in the closed_list (so we don’t re-explore), or if they are already in the open_list, update them if they have a smaller f value
```
Implementation Note

• When the distance to a node/state/successor (i.e. g value) is uniform, we can greedily add a state to the closed-list at the same time as we add it to the open-list.

```cpp
open_list.push(Start State)
while(open_list is not empty)
    1. s ← remove min. f-value state from open_list
       (if tie in f-values, select one w/ larger g-value)
    2. Add s to closed list
    3a. if s = goal node then trace path back to start; STOP!
    3b. Generate successors/neighbors of s, compute their f values, and add them to open_list if they are not in the closed_list (so we don’t re-explore), or if they are already in the open list, update them if they have a smaller f value
```

Non-uniform g-values

Uniform g-values

open_list.push(Start State)
Closed_list.push(Start State)
while(open_list is not empty)
    1. s ← remove min. f-value state from open_list
       (if tie in f-values, select one w/ larger g-value)
    3a. if s = goal node then trace path back to start; STOP!
    3b. Generate successors/neighbors of s, compute their f values, and add them to open_list and closed_list if they are not in the closed_list.

The first occurrence of a board has to be on the shortest path to the solution.
If time allows...

BETWEENESS CENTRALITY
BC Algorithm Overview

• What's the most central vertex(es) in the graph below?

• How do we define "centrality"?

• Betweenness centrality defines "centrality" as the nodes that are between the most other pairs

Sample Graph

Graph 1

Graph 2
BC Algorithm Overview

- Betweenness centrality (BC) defines "centrality" as the nodes that are between (i.e. on the path between) the most other pairs of vertices.
- BC considers betweenness on only "shortest" paths!
- To compute centrality score for each vertex we need to find shortest paths between all pairs...
  - Use the Breadth-First Search (BFS) algorithm to do this.

Sample Graph

Are these gray nodes 'between' a and e?

No, a-c-d-e is the shortest path?
BC Algorithm Overview

- Betweenness-Centrality determines "centrality" as the number of shortest paths from all-pairs upon which a vertex lies
- Consider the sample graph below
  - Each external vertex (a, b, e, f) lies is a member of only the shortest paths between itself and each other vertex
  - Vertices c and d lie on greater number of shortest paths and thus will be scored higher
BC Implementation

Based on Brandes' formulation for unweighted graphs

- Perform $|V|$ Breadth-first traversals
- Traversals result in a subgraph consisting of shortest paths from root to all other vertices
- Messages are then sent back up the subgraph from "leaf" vertices to the root summing the percentage of shortest-paths each vertex is a member of
- Summing a vertex's score from each traversal yields overall BC result

Sample Graph with final BC scores

<table>
<thead>
<tr>
<th></th>
<th>a</th>
<th>b</th>
<th>c</th>
<th>d</th>
<th>e</th>
<th>f</th>
</tr>
</thead>
<tbody>
<tr>
<td>5</td>
<td>5</td>
<td>5</td>
<td>19</td>
<td>19</td>
<td>5</td>
<td></td>
</tr>
</tbody>
</table>

Traversals from selected roots and resulting partial BC scores (in this case, the number of descendants)

<table>
<thead>
<tr>
<th></th>
<th>a</th>
<th>b</th>
<th>c</th>
<th>d</th>
<th>e</th>
<th>f</th>
</tr>
</thead>
<tbody>
<tr>
<td>5</td>
<td>5</td>
<td>4</td>
<td>0</td>
<td>2</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>5</td>
<td>5</td>
<td>0</td>
<td>0</td>
<td>2</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>
BC Implementation

• As you work down, track # of shortest paths running through a vertex and its predecessor(s)

• On your way up, sum the nodes beneath

# of shortest paths thru the vertex, [List of predecessor]

Score on the way back up (if multiple shortest paths, split the score appropriately)

Traversals from selected roots and resulting partial BC scores (in this case, the number of descendants)