Graphs

- Definition: A graph consists of a set \( N \) of nodes (or vertices) and a set \( E \) of edges (or arcs). Each edge connects two vertices.
  - A directed graph is a graph in which each edge has a direction
  - A weighted graph: an edge label represents the cost of traversing the edge
    - edge is written \((v,w)\) where \( v \) and \( w \) are vertices
      - \((v,w)\) is incident on \( v \) and \( w \)
    - parallel edges: incident on the same vertices
    - an edge \((v,v)\) is called a loop
  - a path of length \( n \) between two vertices is a sequence of edges which connect them, written
    \((v_0,v_1,v_1,...,v_{n-1},v_n)\)
    - each edge in the path is incident on \( v(i-1) \) and \( v(i) \).
  - connected graph: there is a path between every pair of vertices
  - a cycle is a path of nonzero length from \( v \) to \( v \) with no repeated edges.

Representing Graphs

- Adjacency matrix
  - n by n matrix, where \( n \) is number of vertices
  - \( A(m,n) = 1 \) iff \((m,n)\) is an edge
  - for weighted graph: \( A(m,n) = w \) (weight of edge)
  - only works if there are no parallel edges
  - Example: adjacency matrix for a graph

\[
\begin{bmatrix}
0 & 2 & 0 & 1 & 0 & 0 & 0 \\
0 & 0 & 0 & 3 & 10 & 0 & 0 \\
4 & 0 & 0 & 0 & 0 & 5 & 0 \\
0 & 0 & 2 & 0 & 2 & 8 & 4 \\
0 & 0 & 0 & 0 & 0 & 6 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 1 & 0 \\
\end{bmatrix}
\]

- Adjacency list
  - each vertex has linked list of edges
  - edge stores destination and label
    - better if adjacency matrix would be sparse
  - can work even if there are parallel edges
- **Example:**

- **Graph Traversal**
  - Depth-first: follow a path until it ends, or until a cycle
  - Breadth-first: follow all paths in parallel
  - Example: start from v0
    - assume edges with low-numbered destinations are chosen first
      - Depth-first:
      - Breadth-first:

- **Task: Find shortest path between 2 nodes**

- **Dijkstra's algorithm**
  - Best-first: sort paths visited so far by cost
    - Priority queue is an efficient way to do this
  - If reach same node, only continue if it's a cheaper path than found before
  - Can stop when all queued paths are more expensive than known paths

- **Dijkstra Implementation**
  
  Find cheapest path \((v, \ldots, w)\)
  
  - Maintain a table of length \(V\) (number of vertices in the graph)
  
  - The \(i^{th}\) entry in the table contains:
    - Cost of best (cheapest) known path \((v, \ldots, v_i)\)
    - Immediate predecessor to \(v_i\) on the best known path
    - Whether or not the known path is definitely the cheapest to \(v_i\)
  
  - Initialize the table so that entry \(v\) has a cost of 0, and no predecessor
  - Also maintain a priority queue of vertices (prioritized by cheapest known path).

  Initialize the queue with \(v\).
  
  - At each iteration:
    1. Remove the minimum item \(v_{\text{min}}\) on the priority queue
    2. Mark the table to indicate that the known path to \(v_{\text{min}}\) is definitely the cheapest
    3. "Expand" the vertex, by exploring edges originating at \(v_{\text{min}}\). For each edge \((v_{\text{min}}, w_i)\), add \(w_i\) to the queue. Update table information for \(w_i\) (cost, predecessor)
4. Continue until any known path to \( w \) is definitely the cheapest

- Example graph

Another example: Find the shortest path from Chicago to Peoria
- \textbf{A}^*  
  - Extension of Dijkstra  
  - Utilizes an "estimator" function, which estimates the cost of \((v, \ldots, w)\)  
  - Modification of Dijkstra: compute \(c'\) (real cost so far + estimated remaining cost),  
    and use that to order the priority queue

- \textbf{A}^* \textbf{behavior}  
  - Still guaranteed to find cheapest path as long as \(c' \leq c\) in all cases

- \textbf{Dynamic creation of graphs}  
  - In many applications, it is not feasible to represent an entire graph
Example: the Web

Instead, generate graph dynamically; assume a `successors` method, which takes the place of the explicit representation of edges

**Example application: game playing**

- Single person game
- Deterministic
- Example: 8-puzzle

<table>
<thead>
<tr>
<th>Start</th>
<th>Win</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 3 4</td>
<td></td>
</tr>
<tr>
<td>8 2</td>
<td></td>
</tr>
<tr>
<td>7 6 5</td>
<td></td>
</tr>
<tr>
<td>1 2 3</td>
<td></td>
</tr>
<tr>
<td>8 4</td>
<td></td>
</tr>
<tr>
<td>7 6 5</td>
<td></td>
</tr>
</tbody>
</table>

- Graph representation
  - vertices are puzzle positions
  - edges represent possible moves
  - Problem: 9! vertices, more than $2^{9!}$ edges
  - Requires dynamic generation of the graph

- Implementation
  - Define class to represent puzzle states, and a `successor` function
  - Define class to represent $c'$, $p$, and $v$ for a board state
  - Instead of an array, use a HashMap
  - First, try with no estimator, then using $A^*$
    - Estimator: Manhattan distance between current and winning position of each tile

**Minimum spanning trees**

- Definition: A minimum spanning tree $T$ of a graph $G = (N,E)$ has the following properties
  - Every node in $G$ is also in $T$
  - The edges in $T$ are a subset of $E$ such that
    - $T$ is connected
      - The sum of the weights of the edges of $T$ are minimized

**MST application example**

- Airline: Where to schedule flights?
  - The cities to be connected are vertices
• Edges represent pairs of cities (perhaps not all pairs), and miles between
• MST = least-cost set of flights to connect all cities (not all non-stop)
• Might want to supplement MST to save time
  o Roads: Where to build interstates?

• Example

Graph

\[\begin{array}{c}
v_1 \quad 2 \quad v_3 \\
\quad \downarrow 1 \quad \downarrow 3 \quad \downarrow 10 \\
v_2 \quad 2 \quad v_4 \quad 7 \quad v_5 \\
\quad \downarrow 5 \quad \downarrow 8 \quad \downarrow 4 \quad \downarrow 6 \\
v_6 \quad v_7 \\
1
\end{array}\]

Minimum spanning tree

\[\begin{array}{c}
v_1 \quad 2 \quad v_3 \\
\quad \downarrow 1 \\
v_2 \quad 2 \quad v_4 \quad v_5 \\
\quad \downarrow 4 \quad \downarrow 6 \\
v_6 \quad v_7 \\
1
\end{array}\]

• Prim's algorithm
  o Begin by arbitrarily picking a root
  o Pick the edge incident at the root with the lowest weight
    • This adds another vertex to the tree as well
    • Repeatedly pick the edge with the smallest cost incident on one vertex in the tree, and incident on one vertex not in the tree
    • Terminate when graph is connected
  o Prim's algorithm = Dijkstra's algorithm, but find cheapest path from \(v\) to all other vertices in the graph

• Kruskal's algorithm
  o Start with empty minimum spanning tree
  o Place the edges into a binary heap
  o Select the edge with the lowest weight;
- add it to the tree if it does not result in a cycle
  - Union-find algorithm can be used to detect cycles
- Stop when the graph is connected

**Union-find**

- Data structure for keeping track of graph connectivity
- Operations:
  - addEdge(v,w): add an edge between v and w; update data about graph
  - isConnected(v,w): determine if there is a path from v to w
- Does not explicitly keep track of edges; only the connectivity of vertices/subgraphs
- Simple version:
  - Connectivity representation is an array; begins with all unique numbers
  - Each number represents a subset of the vertices which are connected
  - Example: Graph contains 10 vertices, no edges initially

Consider the following sequence of operations:

```plaintext
addEdge(7,8)  
addEdge(8,9)  
addEdge(1,2)  
isConnected(7,9)  
addEdge(2,4)  
addEdge(0,3)  
addEdge(3,4)  
isConnected(1,4)  
isConnected(0,9)  
addEdge(3,5)  
addEdge(1,3)  
addEdge(5,6)  
addEdge(5,7)
```