Today – More about Trees

- Spanning trees
  - Prim’s algorithm
  - Kruskal’s algorithm

- Generic search algorithm
  - Depth-first search example
  - Handling cycles
  - Breadth-first search example
Trees

- A tree is a connected graph without cycles
- A connected graph is a tree iff it has $N$ vertices and $N-1$ edges
- A graph is a tree iff there is one and only one path joining any two of its vertices

Spanning Trees

- A Spanning tree of a graph $G$, is a tree that includes all the vertices from $G$. The resulting spanning tree is not unique

Minimum Spanning Tree

- Prim’s Algorithm
  - Finds a subset of the edges (that form a tree) including every vertex and the total weight of all the edges in tree is minimized
    - Choose starting vertex
    - Create the Fringe Set
    - Loop until the MST contains all the vertices in the graph
      - Remove edge with minimum weight from Fringe Set
      - Add the edge to MST
      - Update the Fringe Set

Prim – Initialization

- Pick any vertex $x$ as the starting vertex
- Place $x$ in the Minimum Spanning Tree (MST)
- For each vertex $y$ in the graph that is adjacent to $x$
  - Add $y$ to the Fringe Set
- For each vertex $y$ in the Fringe Set
  - Set weight of $y$ to weight of the edge connecting $y$ to $x$
  - Set $x$ to be parent of $y$
**Prim – Body**

While number of vertices in MST < vertices in the graph

Find vertex \( y \) with minimum weight in the Fringe Set

Add vertex and the edge \( x,y \) to the MST

Remove \( y \) from the Fringe Set

For all vertices \( z \) adjacent to \( y \)

If \( z \) is not in the Fringe Set

Add \( z \) to the Fringe Set

Set parent to \( y \)

Set weight of \( z \) to weight of the edge connecting \( z \) to \( y \)

Else

If Weight(\( y,z \)) < Weight(\( z \)) then

Set parent to \( y \)

Set weight of \( z \) to weight of the edge connecting \( z \) to \( y \)

---

**Minimum Spanning Tree**

- **Kruskal’s Algorithm**
  - Finds a minimum spanning tree for a connected weighted graph

  - Create a set of trees, where each vertex in the graph is a separate tree
  - Create set \( S \) containing all edges in the graph
  - While \( S \) not empty
    - Remove edge with minimum weight from \( S \)
    - if that edge connects two different trees, then add it to the forest, combining two trees into a single tree
    - Otherwise discard that edge
More about Trees

• Spanning trees
  – Prim’s algorithm
  – Kruskal’s algorithm

• Generic search algorithm
  – Depth-first search example
  – Handling cycles
  – Breadth-first search example

Depth First Search (DFS)

Idea:
• Explore descendants before siblings
• Explore siblings left to right

Where do we place the children on the queue?
• Assume we pick first element of Q
• Add path extensions to the Q

Simple Search Algorithm

Let Q be a list of partial paths,
Let S be the start node and
Let G be the Goal node.

1. Initialize Q with partial path (S)
2. If Q is empty, fail. Else, pick a partial path N from Q
3. If head(N) = G, return N (goal reached!)
4. Else:
   a) Remove N from Q
   b) Find all children of head(N) and create all the one-step extensions of N to each child.
   c) Add all extended paths to Q
   d) Go to step 2.

Depth-First

Pick first element of Q; Add path extensions to front of Q
**Simple Search Algorithm**

Let $Q$ be a list of partial paths, 
Let $S$ be the start node and 
Let $G$ be the Goal node.

1. Initialize $Q$ with partial path $(S)$
2. If $Q$ is empty, fail. Else, pick a partial path $N$ from $Q$
3. If head($N$) = $G$, return $N$ (goal reached!)
4. Else:
   a) Remove $N$ from $Q$
   b) Find all children of head($N$) and create all the one-step extensions of $N$ to each child.
   c) Add all extended paths to $Q$
   d) Go to step 2.

**Depth-First**

Pick first element of $Q$; Add path extensions to front of $Q$

<table>
<thead>
<tr>
<th>$Q$</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
</tr>
<tr>
<td>2</td>
</tr>
<tr>
<td>3</td>
</tr>
<tr>
<td>4</td>
</tr>
<tr>
<td>5</td>
</tr>
</tbody>
</table>

*Added paths in blue*
Simple Search Algorithm

Let Q be a list of partial paths,
Let S be the start node and
Let G be the Goal node.

1. Initialize Q with partial path (S)
2. If Q is empty, fail. Else, pick a partial path N from Q
3. If head(N) = G, return N      (goal reached!)
4. Else:
   a) Remove N from Q
   b) Find all children of head(N) and create all the one-step extensions of N to each child.
   c) Add all extended paths to Q
   d) Go to step 2.

Depth-First

Pick first element of Q; Add path extensions to front of Q

<table>
<thead>
<tr>
<th>Q</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
</tr>
<tr>
<td>(S)</td>
</tr>
<tr>
<td>2</td>
</tr>
<tr>
<td>(A S) (B S)</td>
</tr>
<tr>
<td>3</td>
</tr>
<tr>
<td>(C A S) (D A S) (B S)</td>
</tr>
<tr>
<td>4</td>
</tr>
<tr>
<td>5</td>
</tr>
</tbody>
</table>

**Added paths in blue**
**Depth-First**

Pick first element of Q; Add path extensions to front of Q

```
<table>
<thead>
<tr>
<th>Q</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>1 (S)</td>
<td></td>
</tr>
<tr>
<td>2 (A S) [B S]</td>
<td></td>
</tr>
<tr>
<td>3 (C A S) [D A S] [B S]</td>
<td></td>
</tr>
<tr>
<td>4 (D A S) [B S]</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td></td>
</tr>
</tbody>
</table>
```

*Added paths in blue*

---

**Depth-First**

Pick first element of Q; Add path extensions to front of Q

```
<table>
<thead>
<tr>
<th>Q</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>1 (S)</td>
<td></td>
</tr>
<tr>
<td>2 (A S) [B S]</td>
<td></td>
</tr>
<tr>
<td>3 (C A S) [D A S] [B S]</td>
<td></td>
</tr>
<tr>
<td>4 (D A S) [B S]</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td></td>
</tr>
</tbody>
</table>
```

*Added paths in blue*
**Simple Search Algorithm**

Let Q be a list of partial paths, Let S be the start node and Let G be the Goal node.

1. Initialize Q with partial path (S)
2. If Q is empty, fail. Else, pick a partial path N from Q
3. If head(N) = G, return N (goal reached!)
4. Else:
   a) Remove N from Q
   b) Find all children of head(N) and create all the one-step extensions of N to each child.
   c) Add all extended paths to Q
   d) Go to step 2.

---

**Depth-First**

Pick first element of Q; Add path extensions to front of Q

---

**Depth-First**

Pick first element of Q; Add path extensions to front of Q

---

**Depth-First**

Pick first element of Q; Add path extensions to front of Q
More about Trees

- Spanning trees
  - Prim’s algorithm
  - Kruskal’s algorithm

- Generic search algorithm
  - Depth-first search example
  - Handling cycles
  - Breadth-first search example

**Issue:** Starting at S and moving top to bottom, will depth-first search ever reach G?

Depth-First

Effort can be wasted in more mild cases

<table>
<thead>
<tr>
<th>Q</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>(S)</td>
</tr>
<tr>
<td>2</td>
<td>(A S) (B S)</td>
</tr>
<tr>
<td>3</td>
<td>(C A S) (D A S) (B S)</td>
</tr>
<tr>
<td>4</td>
<td>(D A S) (B S)</td>
</tr>
<tr>
<td>5</td>
<td>(C D A S) (G D A S) (B S)</td>
</tr>
<tr>
<td>6</td>
<td>(G D A S) (B S)</td>
</tr>
</tbody>
</table>

- C visited multiple times
- Multiple paths to C, D & G

How Do We Avoid Repeat Visits?

**Idea:**

- Keep track of nodes already visited.
- Do not place visited nodes on Q.

**Does this maintain correctness?**

- Any goal reachable from a node that was visited a second time would be reachable from that node the first time.

**Does it always improve efficiency?**

- Guarantees each node appears at most once at the head of a path in Q.
Simple Search Algorithm

Let Q be a list of partial paths, 
Let S be the start node and 
Let G be the Goal node.

1. Initialize Q with partial path (S) as only entry; set Visited = ( )
2. If Q is empty, fail. Else, pick some partial path N from Q
3. If head(N) = G, return N (goal reached!)
4. Else
   a) Remove N from Q
   b) Find all children of head(N) not in Visited and create all the one-step extensions of N to each child.
   c) Add to Q all the extended paths;
   d) Add children of head(N) to Visited
   e) Go to step 2.

More about Trees

• Spanning trees
  – Prim’s algorithm
  – Kruskal’s algorithm

• Generic search algorithm
  – Depth-first search example
  – Handling cycles
  – Breadth-first search example

Breadth First Search (BFS)

Idea:
• Explore relatives at same level before their children
• Explore relatives left to right

Where do we place the children on the queue?
• Assume we pick first element of Q
• Add path extensions to ? of Q

Breadth-First

Pick first element of Q; Add path extensions to end of Q
Breadth-First
Pick first element of Q; Add path extensions to end of Q

<table>
<thead>
<tr>
<th>Q</th>
<th>Visited</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>(S)</td>
</tr>
<tr>
<td>2</td>
<td>(A S) (B S)</td>
</tr>
<tr>
<td>3</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td></td>
</tr>
<tr>
<td>6</td>
<td></td>
</tr>
</tbody>
</table>

Breadth-First
Pick first element of Q; Add path extensions to end of Q

<table>
<thead>
<tr>
<th>Q</th>
<th>Visited</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>(S)</td>
</tr>
<tr>
<td>2</td>
<td>(A S) (B S)</td>
</tr>
<tr>
<td>3</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td></td>
</tr>
<tr>
<td>6</td>
<td></td>
</tr>
</tbody>
</table>
**Breadth-First**
Pick first element of Q; Add path extensions to end of Q

<table>
<thead>
<tr>
<th>Q</th>
<th>Visited</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>(S) S</td>
</tr>
<tr>
<td>2</td>
<td>(A S) (B S) A,B,S</td>
</tr>
<tr>
<td>3</td>
<td>(B S) (C A S) (D A S) C,D,B,A,S</td>
</tr>
<tr>
<td>4</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td></td>
</tr>
<tr>
<td>6</td>
<td></td>
</tr>
</tbody>
</table>

**Depth First Search (DFS)**
Depth-first: Add path extensions to front of Q
Pick first element of Q

**Breadth First Search (BFS)**
Breadth-first: Add path extensions to back of Q
Pick first element of Q

**Summary**
- Most problem solving tasks may be formulated as state space search.
- Mathematical representations for search are graphs and search trees.
- Depth-first and breadth-first search may be framed, among others, as instances of a generic search strategy.
- Cycle detection is required to achieve efficiency and completeness.
• Document code
  – What it is doing
  – How it is doing it
  – What it is not doing (detailed status)

• Test run code

• Zip code