Overview

• HW7 due in one week
• Last lecture: Graph Theory
  – N-Cube, Bipartite Graphs, Complete Bipartite Graphs
  – Operations: Subgraphs, Union of Graphs
  – Adjacency Matrix, Adjacency Lists
  – Isomorphism
  – Path, Circuit, Cycle: Hamiltonian Cycle, Eulerian Circuit
• Today’s lecture
  – Connectivity, Connected Components, Planar Graphs
  – Shortest path problem: Dijkstra algorithm
  – Traveling salesman problem
  – Trees: Spanning trees, Rooted trees, Binary trees
  – Tree traversal problem

Connectivity

• Definition: An undirected graph is called connected if there is a path between every pair of distinct vertices in the graph.
• For example, any two computers in a network can communicate if and only if the graph of this network is connected.
• Note: A graph consisting of only one vertex is always connected, because it does not contain any pair of distinct vertices.
• Examples:

```plaintext
Yes.  

No.  

Yes.  

No.  
```
• **Definition:** A graph that is not connected is the union of two or more connected subgraphs, each pair of which has no vertex in common. These disjoint connected subgraphs are called the **connected components** of the graph.

• **Definition:** A connected component of a graph G is a maximal connected subgraph of G.

• **Example:**

![Graph Image]

The connected components are the graphs with vertices \{a, b, c, d\}, \{e\}, \{f\}, \{i, g, h, j\}.

• **Definition:** An directed graph is **strongly connected** if there is a path from a to b and from b to a whenever a and b are vertices in the graph.

• **Definition:** An directed graph is **weakly connected** if there is a path between any two vertices in the underlying undirected graph.

**Weakly connected**, because, for example, there is no path from b to d. But underlying undirected graph is connected.

**Strongly connected**, because there are paths between all possible pairs of vertices.
A **planar graph** is one that can be represented (on a sheet of paper, that is, in the plane) so that its edges intersect only at endpoints.

Right graph = Left graph. It is clearly a planar graph.

what about $K_4$ and $K_5$?

**NO**

---

**Application: Shortest Path Problems**

- We can assign **weights** to the edges of graphs, for example, to represent the distance between cities in a railway network:

  ![Graph](image)

- One of the most interesting questions that we can investigate with such graphs is:

  What is the **shortest path** between two vertices in the graph, that is, the path with the **minimal sum of weights** along the way?

  This corresponds to the shortest train connection or the fastest connection in a computer network (edge weight = response time).
Dijkstra’s Algorithm

- Dijkstra’s algorithm is an iterative procedure that finds the shortest path between two vertices (e.g., a to z) in a weighted graph.
- It proceeds by finding the length of the shortest path from a to successive vertices.
- The algorithm terminates once it reaches the vertex z.

The final shortest path is then back-tracked from z.

Example:

Answer: a, c, b, d, e, z  
keep min cost value on each vertex

```
1 function Dijkstra(Graph, source(a), target(z)):
2   for each vertex v in Graph: // Initializations
3       dist[v] := infinity  // Unknown distance function from source to v
4       prev[v] := undefined // Previous node in optimal path from source
5       dist[source] := 0  // Distance from source to source
6   Q := the set of all nodes in Graph
7   while Q is not empty: // The main loop
8       u := node in Q with smallest dist[]
9       remove u from Q
10      if u=target, exit, done!
11      for each neighbor v of u and v still in Q.
12          alt := dist[u] + dist_between(u, v)
13          if alt < dist[v] // Relax (u,v)
14             dist[v] := alt  // update v’s data
15             prev[v] := u // update v’s previous node to u
```

\[ \text{dist}(V=a, \ldots, z) = \infty \]
\[ \text{prev}(V=0, \ldots, z) = \text{undefined} \]

1. dist(a) = 0 @ source
2. Pop node w/ smallest d(C) → A
3. Look at all neighbors q(a) → b, c
   - b) alt = d(a) + 4 = 4
   - c) alt = d(a) + 3 = 3

\[ d(C)=4, \text{prev}(C)=a \]

4. Remove a from Q
5. Pop Q → c
6. Remove C from Q
7. Look at neighbors q(C) = b, d
   - b) alt = d(b) + 3 = 5
6. d(C)=3, \text{prev}(C)=a

\[ d(C)=3, \text{prev}(C)=a \]

8. d(C)=3, \text{prev}(C)=a
9. d(C)=3, \text{prev}(C)=a
10. d(C)=3, \text{prev}(C)=a

\[ Q = \{ a, b, c, d, e, z \} \]

1. alt = d(a) + 6 = 6
2. alt = d(a) + 6 = 6
3. alt = d(a) + 6 = 6

\[ d(C)=10, \text{prev}(C)=a \]

4. \[ d(C)=10, \text{prev}(C)=a \]
5. \[ d(C)=10, \text{prev}(C)=a \]
6. \[ d(C)=10, \text{prev}(C)=a \]
7. \[ d(C)=10, \text{prev}(C)=a \]
8. \[ d(C)=10, \text{prev}(C)=a \]
9. \[ d(C)=10, \text{prev}(C)=a \]
10. \[ d(C)=10, \text{prev}(C)=a \]

\[ d(C)=10, \text{prev}(C)=a \]
Application: The Traveling Salesman Problem

- The **traveling salesman problem** is one of the classical problems in computer science.

  A traveling salesman wants to visit all major cities and then return to his starting point (any city!). Of course he wants to save time and energy, so he wants to determine the **shortest path** for his trip.

  We can represent the cities and the distances between them by a **weighted, complete, undirected graph**.

  The problem then is to find a **Hamiltonian cycle of minimum total weight that visits each vertex exactly once**.

- **Example:** What path would the traveling salesman take to visit the following cities?

  ![Graph](attachment:image.png)

  **Solution:** The shortest path is Boston, New York, Chicago, Toronto, Boston (2,000 miles).

- **Question:** Given $n$ vertices, how many different cycles (with all $n$ vertices) can we form by connecting these vertices with edges?
**Solution:** We first choose a starting point. Then we have \((n - 1)\) choices for the second vertex in the cycle, \((n - 2)\) for the third one, and so on, so there are \((n - 1)!\) choices for the whole cycle.

However, this number includes identical cycles that were constructed in **opposite directions**. Therefore, the actual number of different cycles is \((n - 1)!/2\).

- Unfortunately, no algorithm solving the traveling salesman problem with polynomial worst-case time complexity has been devised yet.

- This means that for large numbers of vertices, solving the traveling salesman problem is **not tractable (impractical)**.

- In these cases, we can use efficient **approximation algorithms** that determine a path whose length may be slightly larger than the traveling salesman’s path.

### 6.2 Introduction to Trees
• Definition: A graph $G$ is said to be a **tree** if it is connected and has no cycle (**acyclic**).

• $G$ is said to be a **forest** if it consists of several trees.

\[ G \text{ is acyclic but not connected.} \]

\[ G \quad \text{Several Spanning trees } G' \text{ of } G \]

\[ G = (V, E) \]

\[ G' = (V', E') \]
**Kruskal Algorithm** (to find a spanning tree):

**Input:** a connected graph \( G = (V,E) \)

**Output:** a spanning tree \((V,T)\) of \( G \)

\[
T = \emptyset \\
\text{for each } e \in E \{ \\
\quad \text{if } (V, \{e\} \cup T) \text{ is acyclic then} \\
\quad \quad T = T \cup \{e\} \\
\} \\
\text{return } (V,T)
\]

Note: above algorithm can be modified to obtain minimum cost spanning tree.
Definition: A **rooted tree** is a connected acyclic graph with one node designated as the **root** of the tree.

- The nodes $r_1, r_2, \ldots, r_t$ are **children** of $r$, and $r$ is a **parent** of $r_1, r_2, \ldots, r_t$.
- A node with no children is called a **leaf**; all nonleaves are **internal nodes**.
- The **depth of a node** in a tree is the length of the path from the root to the node. The **height of the tree** is the maximum depth of any node in the tree.
- Branches: subtrees/subgraphs

**Binary Trees** is a rooted tree where each node has **at most** 2 children.

- **A full binary tree** occurs when all internal nodes have 2 children and all leaves are the same depth.
- **A complete binary tree** is an almost-full binary tree except for the deepest depth.

Note that a complete tree is not a complete graph!
Application of Trees: Counting problem

- A child can choose one jellybean out of two jellybeans (red, black), and one gummy bear out of three gummy bears (yellow, green, white). How many different sets of candy can the child have?

\[
\begin{align*}
\text{choose} & \quad \text{choose} \\
\text{jellybean} & \quad \text{gummy bear} \\
R & \quad Y: \ R, \ Y \\
\quad & \quad G: \ R, \ G \\
\quad & \quad W: \ R, \ W \\
B & \quad Y: \ B, \ Y \\
\quad & \quad G: \ B, \ G \\
\quad & \quad W: \ B, \ W \\
\therefore \quad 6 \text{ outcomes}
\end{align*}
\]

- Family tree - not only interesting but also useful for research in medical genetics.
- Files on your computer are organized in a hierarchical (treelike) structure (nested folders).
- Algebraic expression involving binary operations can be represented by a labeled binary tree.

\[
(2 + x) - (y * 3)
\]

\[
\frac{2 + 4}{\frac{x - y}{y}}
\]

\[
2 + x \quad * \\
3 \quad y
\]
**Binary Tree Representation**

Binary trees have special characteristics: the identity of the left and right child.

![Binary Tree Diagram](image)

<table>
<thead>
<tr>
<th></th>
<th>Left child</th>
<th>Right child</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>2</td>
<td>4</td>
<td>5</td>
</tr>
<tr>
<td>3</td>
<td>0</td>
<td>6</td>
</tr>
<tr>
<td>4</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>5</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>6</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

**Tree Traversal Algorithms**

**Traversal of Tree**: visit every nodes of a tree in a systematic order

The 3 common tree traversal algorithms are **preorder**, **inorder**, and **postorder** traversal.

These terms refer to the order in which the root of a tree is visited compared to the subtree nodes.

In these traversal methods, it is helpful to use the recursive view of a tree, where the root of a tree is a parent of the roots of subtrees.
ALGORITHM Preorder

`PR(T)`

// Writes the nodes of a tree with root r in preorder

write(r)

for i = 1 to t do

PR(Tᵢ)

end for

diagram: r

ALGORITHM Inorder

`IN(tree T)`

// Writes the nodes of a tree with root r in inorder

IN(T₁)

write(r)

for i = 2 to t do

IN(Tᵢ)

end for

end

diagram: a, b, c, d, e, f, i, g, h

ALGORITHM Postorder

`PO(tree T)`

// Writes the nodes of a tree with root r in Postorder

for i = 1 to t do

PO(Tᵢ)

end for

write(r)

diagram: a, b, c, d, e, f, i, g, h

e.g. Do a preorder, inorder, and postorder traversal of the tree.

Preorder: a, b, e, f, c, d, g, i, h

Inorder: e, b, f, a, c, i, g, d, h

Postorder: e, f, b, c, i, g, h, d, a
Algebraic expressions represented as binary trees

Inorder traversal: \((2+x) \ast 4\)

Preorder traversal: \(* + 2 \times 4\)

Postorder traversal: \(2 \times 4 \ast\)

- **infix notation**
  - operation symbol appears between the 2 operands.

- **prefix notation**
  - operation symbol precedes its operands. (Lisp)

- **postfix notation**
  - operation symbol follows its operands. (PS)