Overview

• Q&A on Graphs at the end of next lecture: Bring questions!

• Last Lecture
  – Graph Theory: Definitions using Sets, Undirected/Directed Graphs
  – Vertex degree, Handshaking theorem for Undirected Graph
  – Vertex terms, Vertex degree, & Handshaking theorem for Directed Graph
  – Complete Graph, n-Cube, Bipartite Graph, Complete bipartite graph
  – Operations: Subgraph, Union

• This Lecture: More on graph theory
  – Adjacency matrix, Adjacency list
  – Isomorphism (equivalency of graphs)
  – Path/Circuit/Cycle (Hamilton & Euler)
  – Connectivity, connected graphs, strong/weak connected
  – Planar graphs

Representing Graphs

• Definition: Let $G = (V, E)$ be a graph with $|V| = n$. Suppose that the vertices of $G$ are listed in arbitrary order as $v_1, v_2, ..., v_n$.

• The adjacency matrix $A_G$ of $G$, with respect to this listing of the vertices, is the $n \times n$ zero-one-valued matrix with 1 as its $(i, j)$-th entry when $v_i$ and $v_j$ are adjacent, and 0 otherwise.

• In other words, for an adjacency matrix $A = [a_{ij}]$, $a_{ij} = 1$ if $\{v_i, v_j\}$ is an edge of $G$, $a_{ij} = 0$ otherwise.

• Example: What is the adjacency matrix $A_G$ for the following graph $G$ based on the order of vertices $a, b, c, d$?

Solution: $A_G = \begin{bmatrix}
0 & 1 & 1 & 1 \\
1 & 0 & 0 & 1 \\
1 & 0 & 0 & 1 \\
1 & 1 & 1 & 0
\end{bmatrix}$
Definition: **Adjacency list** representation consists of a *list* of its vertices together with a separate *list* for each vertex that contains all the vertices adjacent to that vertex.

Example: let $1 = a$, $2 = b$, $3 = c$ and $4 = d$

<table>
<thead>
<tr>
<th>Vertex</th>
<th>List of adjacencies</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2, 3, 4</td>
</tr>
<tr>
<td>2</td>
<td>1, 4</td>
</tr>
<tr>
<td>3</td>
<td>1, 4</td>
</tr>
<tr>
<td>4</td>
<td>1, 2, 3</td>
</tr>
</tbody>
</table>

Note: Adjacency Matrix and Adjacency Lists can be used to defined directed graphs also.

---

**Isomorphism of Graphs**

- **Definition:** The graphs $G_1 = (V_1, E_1)$ and $G_2 = (V_2, E_2)$ are **isomorphic** if there is a **bijection** (an one-to-one and onto function) $f$ from $V_1$ to $V_2$ with the property that $a$ and $b$ are adjacent in $G_1$ if and only if $f(a)$ and $f(b)$ are adjacent in $G_2$, for all $a$ and $b$ in $V_1$. Such a function $f$ is called an **isomorphism**.

- In other words, $G_1$ and $G_2$ are isomorphic if their vertices can be **ordered** in such a way that the adjacency matrices $A_{G_1}$ and $A_{G_2}$ are identical.

- From a visual standpoint, $G_1$ and $G_2$ are isomorphic if they can be rearranged in such a way that their **displays** are identical.
• Unfortunately, for two graphs, each with \( n \) vertices, there are \( n! \) possible isomorphisms that we have to check in order to show that these graphs are isomorphic.

• One way to show that two graphs are not isomorphic is to check invariants, that is, properties that two isomorphic simple graphs must both have. For example, they must have

- the same number of vertices,
- the same number of edges, and
- the same degrees of vertices (there are more….)

Note that two graphs that differ in any of these invariants must be “not isomorphic”, but two graphs that match in all of them are not necessarily isomorphic.

→ so check the invariants first to see if they are NOT isomorphic!

---

• Example I: Are the following two graphs isomorphic?

\[ \text{Solution: Yes, they are isomorphic, because they can be arranged to look identical, i.e. } f(a) = e, f(b) = a, f(c) = b, f(d) = c, f(e) = d. \]

• Example II: How about these two graphs?

\[ \text{Solution: No, they are not isomorphic, because they differ in the degrees of their vertices. Vertex } d \text{ in right graph is of degree one, but there is no such vertex in the left graph.} \]
Path, Circuit, Cycle

• **Definition:** A path of length \( n \) from \( u \) to \( v \), where \( n \) is a positive integer, in an undirected graph, is a sequence of edges \( e_1, e_2, ..., e_n \) of the graph such that \( e_1 = \{x_0, x_1\}, e_2 = \{x_1, x_2\}, ..., e_n = \{x_{n-1}, x_n\} \), where \( x_0 = u \) and \( x_n = v \). We can also denote this path by its vertex sequence \( x_0, x_1, ..., x_n \). The path or circuit is said to pass through or traverse \( x_0, x_1, ..., x_n \).

• The path is a **Circuit** if it begins and ends at the same vertex, that is, if \( u = v \).

• A path or circuit is **simple** if it does not contain the same edge more than once. A **simple circuit** is also called a **Cycle**.

• A cycle that contains all the vertices of a graph is called a **Hamiltonian (Hamilton) Cycle**.

• A circuit that includes each edge of a graph exactly once is called an **Eulerian Circuit**.

**Theorem:** Let \( G \) be a (connected) graph. \( G \) has an Eulerian circuit if and only if each vertex is even degree.

**Example:** The Königsberg Bridges graph.

This graph is not Eulerian, therefore, a solution does not exist.

**Example:**

1-2-3-1-4-5-1-6-7-8-6-2-9-1
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:

  ![Graph Example 1](image1.png)

  Yes.

  ![Graph Example 2](image2.png)

  Yes.

  ![Graph Example 3](image3.png)

  No.

  ![Graph Example 4](image4.png)

  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 Example 5](image5.png)

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

\[K_4\] and \([K_5]\) isomorphic?

1. Check invariants:
   - # of vertices
   - # of edges
   - distribution of degrees
2. Try making one graph nonplanar so that it looks to save us time!
3. See a bijection \( f \):
   - \( V_1 \to V_2 \)
   - adjacency in \( G_1 \) remain true in \( G_2 \)
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:

- 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 destination 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 // 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
```