Introducing Graph Theory

- Graphs were introduced by Euler in 1736 with the Koenigsberg bridge problem (EAST PRUSSIA).

- Problem: Starting at some land area, is it possible to walk across all the bridges exactly once returning to the starting land area?

**Diagram:**

[Diagram of a graph with labeled vertices and edges, illustrating Euler's circuits and multi-graphs.]
Theorem

Multi-Graph $G$ has an euler circuit if and only if $G$ is connected and every node is of even degree.

Algorithm by Stephen Barnard. (Discuss Quickly)

function EULER(v:vertex) Returns Path
{
    path:=NULL;
    for all vertices w adjacent to v
        and edge(v,w) not yet used do
    {
        mark (v,w) used;
        path:={{(v,w)}} || EULER(w) || path;
        // concatenate represented by ||
    }
    return path;
}

C++ code appears elsewhere.
Possible Execution of Algorithm

The labels in the edges indicate the order in which they are used in the recursive calls. Next slide gives more details of the recursive calls.
Recursive Calls

<table>
<thead>
<tr>
<th>edge label</th>
<th>1st Iteration (v,w) Path</th>
<th>2nd Iteration (v,w) Path</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>(1,2) (1,3) (2,3) (3,4) (4,1)</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>(2,2) (2,3) (3,1) (4,3)</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>(3,4) (3,5) (5,3)</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>(4,3) (4,2)</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>(5,10) (10,9) (9,10)</td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>(6,4) (4,5) (5,6)</td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>(7,5) (5,6)</td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>(8,9) (9,4) (4,7) (7,8)</td>
<td></td>
</tr>
<tr>
<td>9</td>
<td>(9,8) (8,9) (9,10)</td>
<td></td>
</tr>
<tr>
<td>10</td>
<td>(10,8) (8,9)</td>
<td></td>
</tr>
<tr>
<td>11</td>
<td>(10,7) (7,8)</td>
<td></td>
</tr>
<tr>
<td>12</td>
<td>(7,8)</td>
<td></td>
</tr>
<tr>
<td>13</td>
<td>(8,8) (8,9) (9,10)</td>
<td></td>
</tr>
<tr>
<td>14</td>
<td>(1,9) (9,10)</td>
<td></td>
</tr>
<tr>
<td>15</td>
<td>(9,9)</td>
<td></td>
</tr>
<tr>
<td>16</td>
<td>(9,10)</td>
<td></td>
</tr>
<tr>
<td>17</td>
<td>(10,2) (2,10) (10,7) (7,6) (6,10) (10,9)</td>
<td></td>
</tr>
<tr>
<td>18</td>
<td>(7,10) (2,10) (10,7) (7,6) (6,10) (10,9)</td>
<td></td>
</tr>
<tr>
<td>19</td>
<td>(10,7) (10,7) (7,6) (6,10) (10,9)</td>
<td></td>
</tr>
<tr>
<td>20</td>
<td>(7,6) (7,6) (6,10) (10,9)</td>
<td></td>
</tr>
<tr>
<td>21</td>
<td>(6,10) (6,10) (10,9)</td>
<td></td>
</tr>
<tr>
<td>22</td>
<td>(10,9)</td>
<td></td>
</tr>
</tbody>
</table>

These are used to reduce the amount of writing.

Let $P_1 = (10,2)(2,10)(10,7)(7,6)(6,10)(10,9)$
Let $P_2 = (8,1)(1,9)(9,10)P_1(9,1)$
Let $P_3 = (6,9)(9,4)(6,7)(7,8)P_2$
let $P_4 = (3,10)(10,4)(4,5)(5,6)P_3$
**GRAPHS**

Set of nodes (points or vertices)
Set of edges (lines or arcs)

**UNDIRECTED**

\[ V = \{1, 2, 3, 4, 5\} \]
\[ E = \{(1,2),(2,3),(2,4),(2,5),(3,4),(4,5)\} \]

**DIRECTED**

\[ V = \{1, 2, 3, 4, 5\} \]
\[ E = \{(1,2),(2,3),(4,3),(4,5),(5,2),(1,5),(2,1)\} \]
Definition

Graph $G = (V, E)$

$V$: Set of vertices

$E$: Set of edges

- $\{i, j\}$: Undirected
  - $i$ and $j$ are adjacent
  - $\{i, j\}$ is incident on vertices $i$ and $j$

- $(i, j)$: Directed
  - $(i, j)$ is incident to vertex $j$ and incident from $i$
  - $i$ is adjacent to vertex $j$
  - $j$ is adjacent from vertex $i$

No multiple copies of edges
No self edge.
A sequence of vertices $P = i_1, i_2, \ldots, i_k$ is an $i_1$ to $i_k$ path if and only if $(i_j, i_{j+1}) \in E$ for every $1 \leq j < k$.

- Simple path: All vertices, except possibly for the $1^{st}$ and last, are different.
- Length of a path: # of edges in the path.
Adjacency Matrix

- \((v_i, v_j)\) if and only if \(A_{i,j} = 1\)
- Space \(n^2\) bits

**Directed Case**

\[
\begin{array}{cccc}
1 & \rightarrow & 2 \\
\downarrow & & \downarrow \\
4 & \rightarrow & 3
\end{array}
\]

**Bit Matrix**

<table>
<thead>
<tr>
<th></th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>3</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>4</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

**Undirected Case**

\[
\begin{array}{cccc}
1 & \rightarrow & 2 \\
\downarrow & & \downarrow \\
4 & \rightarrow & 3
\end{array}
\]

**Bit Matrix**

<table>
<thead>
<tr>
<th></th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>3</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>4</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

Adjacency Lists

- Space \(O(n + e)\)
**Minimum Cost Spanning Tree (MCST)**

- **Definition of Spanning Tree:** Let $G = (V, E)$ be an undirected connected graph. A subgraph $T = (V, E')$ of $G$ is a **spanning tree** if and only if $T$ is a tree.

- **Definition of Minimum Cost Spanning Tree:** Let $G = (V, E)$ be an undirected connected graph and $w : E \rightarrow I^+$. A subgraph $T = (V, E')$ of $G$ is a **minimum cost spanning tree** if $\sum_{(i,j) \in E} w(i, j)$ is minimum and $T$ is a tree.

- **Kruskal Algorithm:** Greedy method. Add lowest cost edge that does not create a cycle.
\{d, e\} ✓
\{b, e\} ✓
\{c, e\} ✓
\{b, c\} ✗
\{c, d\} ✗
\{a, b\} ✓
Kruskal’s Algorithm

n <- |V|; T <- NULL; E <- Set of edge in G;
while |T|<n-1 do
  e <- Deletemin(E);
  add e to T if it does not create a cycle
endwhile

More Details

Initialize Priority Queue(Q)
ADD all edges in G to priority queue
  Q(i,j,w(i,j));
T:=Empty;
while |T|<n-1 do
  {i,j} <- Deletemin(Q)
  if {T plus {i,j}} is not a cycle
    then add {i,j} to T
endwhile
Kruskal’s Algorithm (refined)

/* G is connected*/
Initialize Union-Find(1..n):
    Priority Queue(Q)
Add all edges in G to priority queue Q
    as triplets (i,j,W(i,j))
    /* W(i,j) is the key for comparison */
T <- NULL;
while |T|<n-1 do
    {i,j} <- Deltemin(Q);
    I <- Find(i);
    J <- Find(j);
    if I!=J then Add {i,j} to T;
        Union(I,J);
endif
endwhile

Time Complexity \( O(e \log e) \rightarrow O(e \log n) \).