Graph Theory

Goal: To plan the most efficient route.

Vocabulary:

- A **graph** is a finite set of dots and connectors.
- A **vertex** is a dot on the graph where edges meet, representing an intersection of streets, a land mass, or a fixed general location. A vertex can only occur when a dot is explicitly placed, not whenever two edges intersect. Vertices are usually labelled with capital letters.
- **Edges** are connectors between pairs of vertices that may represent a physical connection or a route.
- **Weights** can represent distance, travel time, travel cost, etc. and are sometimes assigned to edges depending on the problem.
- **Loops** are a special type of edge that connect a vertex to itself.
- The **degree of a vertex** is the number of edges meeting at that particular vertex. It is possible for a vertex to have a degree of zero or larger.

<table>
<thead>
<tr>
<th>Degree 0</th>
<th>Degree 1</th>
<th>Degree 2</th>
<th>Degree 3</th>
<th>Degree 4</th>
</tr>
</thead>
<tbody>
<tr>
<td><img src="image1.png" alt="Image" /></td>
<td><img src="image2.png" alt="Image" /></td>
<td><img src="image3.png" alt="Image" /></td>
<td><img src="image4.png" alt="Image" /></td>
<td><img src="image5.png" alt="Image" /></td>
</tr>
</tbody>
</table>

- A **path** is a sequence of vertices using the edges, usually used to connect two vertices.

- A **circuit** is a path that begins and ends at the same vertex.

- **Connected** is when there is a path from any vertex to any other vertex on a graph.
- **Disconnected** is when there is no way to get from a vertex to a different vertex on a graph.

- A **complete graph** is a graph where every vertex is connected to every other vertex.
• For $n$ vertices in a complete graph, there are $\frac{(n-1)!}{2}$ unique circuits possible.

• **Algorithms** are a step-by-step procedure for solving a problem.

• An **optimal algorithm** always produces the actual shortest path, provided one exists.

• An **efficient algorithm** can be implemented in a reasonable amount of time.

• A **Hamiltonian circuit** is a circuit that visits every vertex once, with no repeats.

• A Hamiltonian path is a path that visits every vertex once, without repeats. The path does not necessarily have to start and end at the same vertex.

• An **Euler circuit** is a circuit that uses every edge without repeats. A graph will contain an Euler circuit if all vertices have even degree.

• An **Euler path** is a path that uses every edge in a graph with no repeats. The path does not necessarily have to start and end at the same vertex. A graph will contain an Euler path if it contains at most two vertices of odd degree.

• Eulerization is the process of adding edges to a graph to create an Euler circuit on a graph.

• A **spanning tree** is a graph that contains a path from any vertex to any other vertex, but has no circuits.

• **Subgraphs** are new graphs formed from using all of the vertices, but only some of the edges from the original graph.

• The **minimum cost spanning tree (MCST)** is the spanning tree with the smallest total edge weight.

**Dijkstra’s Algorithm**

• Finds the shortest path between two vertices and is both optimal and efficient.

• **Step 1**: Mark the ending vertex with a distance of zero. Designate this vertex as current.

• **Step 2**: Find all vertices leading to the current vertex. Calculate their distances to the end. Don’t record this distance if it’s longer than a previously recorded distance.

• **Step 3**: Mark the current vertex as visited. We will never look at this vertex again.

• **Step 4**: Mark the vertex with the smallest distance as current and repeat from Step 2.

• Example on page 4.

**Fleury’s Algorithm**

• Determines if a graph has an Euler circuit.

• **Step 1**: Start at any vertex.

• **Step 2**: Choose any edge leaving your current vertex, providing that deleting that edge will not separate the graph into two disconnected sets of edges.
• **Step 3**: Add that edge to your circuit and delete it from the graph.
• **Step 4**: Continue until you’re done.
• **Example on page 6.**

**Brute Force Algorithm (exhaustive search)**
- Finds the lowest cost Hamiltonian circuit and is optimal but not always efficient.
- **Step 1**: List all possible Hamiltonian circuits.
- **Step 2**: Find the length of each circuit by adding the edge weights.
- **Step 3**: Select the circuit with the minimal total weights.
- **Example on page 7.**

**Nearest Neighbor Algorithm (NNA)**
- It is a heuristic algorithm, meaning it is an efficient algorithm that gives approximate solutions.
- It is also a greedy algorithm, which means it only looks at immediate decisions without considering future consequences.
- **Step 1**: Select a starting point.
- **Step 2**: Move to the nearest unvisited vertex (the edge with the smallest weight).
- **Step 3**: Repeat until the circuit is complete.
- **Example on page 7.**

**Repeated Nearest Neighbor Algorithm (RNNA)**
- Better than the basic NNA, but is still greedy and will produce bad results for some graphs.
- **Step 1**: Do the Nearest Neighbor Algorithm starting at each vertex.
- **Step 2**: Choose the circuit produced with minimal total weight.
- **Example on page 7.**

**Sorted Edges Algorithm (Cheapest Link Algorithm)**
- Overcomes some of the shortcomings of NNA, but still does not guarantee the optimal circuit.
- **Step 1**: Select the cheapest unused edge in the graph.
- **Step 2**: Repeat Step 1, adding the cheapest unused edge to the circuit, unless:
  - Adding the edge would create a circuit that does not contain all vertices, or
  - Adding the edge would give a vertex with degree 3.
- **Step 3**: Repeat until a circuit containing all vertices is formed.
- **Example on page 8.**

**Kruskal’s Algorithm**
- Finds the minimum cost spanning tree and is both optimal and efficient.
- **Step 1**: Select the cheapest unused edge in the graph.
- **Step 2**: Repeat Step 1, adding the cheapest unused edge, unless adding the edge would create a circuit.
- **Step 3**: Repeat until a spanning tree is formed.
- **Example on page 9.**
**Examples**

- **Dijkstra’s Algorithm**

  Consider the map below. Determine the shortest path between vertex T and vertex Y using Dijkstra’s Algorithm.

  - Mark the ending vertex with a distance of zero. The distances will be recorded in **[brackets]** after the vertex name.

  - For each vertex leading to Y, we calculate the distance to the end. For example, NB is a distance of 104 from the end, and MR is 96 from the end.

  - Mark Y as visited and mark the vertex with the smallest recorded distance as current. At this point, P will be current.

  - For each vertex leading to P (and not leading to a visited vertex), find the distance from the end.
    - Since E is 96 from P, and we’ve calculated P is 76 from Y, then E is 96+76 = 172 from Y.
    - Doing the same calculation for MR, 76+27 = 103. Since this is larger than the previously recorded distance from Y to MR, we will not replace it.
Mark P as visited and designate the vertex with the smallest recorded distance as current. At this point MR will be current.

For each vertex leading to MR (and not leading to a visited vertex), find the distance to the end.

- The only vertex to be considered is A, since we’ve already visited Y and P.
- Adding MR’s distance of 96 to the length from A to MR gives the distance 96 + 79 = 175 from A to Y.

Mark MR as visited and designate the vertex with the smallest recorded distance as current. At this point, NB will be current.

For each vertex leading to NB, find the distance to the end.

- We know the shortest distance from NB to Y is 104 and the distance from A to NB is 36, so the distance from A to Y through NB is 104 + 36 = 140. Since this distance is shorter than the previously calculated distance from Y to A through MR, we replace it.

Mark NB as visited and designate A as current, since it now has the shortest distance.

T is the only non-visited vertex leading to A, so we calculate the distance from T to Y through A: 20 + 140 = 160.
Mark A as visited and designate E as current.
The only non-visited vertex leading to E is T. Calculating the distance from T to Y through E, we compute $172 + 57 = 229$. Since this is longer than the existing marked time, we do not replace it.
Mark E as visited. Since all the vertices have now been visited, we are done.
From this, we know the shortest path from T to Y.

- **Fleury’s Algorithm**
  Find an Euler Circuit on this graph using Fleury’s algorithm, starting at vertex A.

  Choose edge AD.

  AD has been deleted and D is the current vertex.
  Can’t choose DC since that would disconnect the graph.
  Choose edge DE, making the circuit so far ADE.

  DE has been deleted and E is the current vertex.
  From here, there is only one option, so the rest of the circuit is determined.
  The final circuit is ADEBDCA.
MTH 170: Foundations in Contemporary Mathematics

- **Brute Force Algorithm (exhaustive search)**
  Apply the Brute Force algorithm to find the minimum cost Hamiltonian circuit on the graph below.

  ![Graph Image]

  - To apply the Brute Force Algorithm, list all the possible Hamiltonian circuits and calculate their weight:

    | Circuit   | Weight   |
    |-----------|----------|
    | ABCDA     | 4+13+8+1 = 26 |
    | ABDCDA    | 4+9+8+2   = 23 |
    | ACBDA     | 2+13+9+1   = 25 |

  - These are all the unique circuits on this graph. All other possible circuits are just the reverse of the listed circuits, or start at a different vertex, but result in the same weights.
  - From the table, the optimal circuit is ABDCDA, with a weight of 23.

- **Nearest Neighbor Algorithm (NNA)**
  Consider the graph below and apply the NNA starting at vertex A.

  ![Graph Image]

  - Starting at vertex A, the nearest neighbor is vertex D with a weight of 1.
  - From D, the nearest neighbor is C, with a weight of 8.
  - From C, the only option is B, which is the only unvisited vertex, with a weight of 13.
  - From B, return to A with a weight of 4.
  - The resulting circuit is ADCBA, with a total weight of 1+8+13+4 = 26.

- **Repeated Nearest Neighbor Algorithm (RNNA)**
  Consider the graph used in the Nearest Neighbor Algorithm example.

  - Starting at vertex A resulted in a circuit with weight 26.
  - Starting at vertex B, the nearest neighbor circuit is BADCB with a weight of 4+1+8+13 = 26. This is the same circuit as starting at A.
  - Starting at vertex C, the nearest neighbor circuit is CADBC with a weight of 2+1+9+13 = 25.
  - Starting at vertex D, the nearest neighbor circuit is DACBD with a weight of 1+2+13+9 = 25. This is the same circuit as starting at C, with a different starting vertex.
  - The circuit with the minimal total weight is CADBC or DACBD, with a weight of 25.
Sorted Edges Algorithm (Cheapest Link Algorithm)
Consider the graph below and apply the Sorted Edges Algorithm.

- The cheapest edge is AD, with a cost of 1.
- The next shortest edge is AC, with a weight of 2.

- For the third edge, we’d like to add AB, but that would give vertex A degree 3, which is not allowed in a Hamiltonian circuit.
- The next shortest edge is CD, but adding that would create a circuit ACDA that does not include vertex B, so we reject that edge.
- The next shortest edge is BD, so we add that to the graph.

- We then add the last edge, BC, to complete the circuit.

- The final circuit is ACBDA, with a weight of 25.
Kruskal’s Algorithm
Consider the graph below and find the minimum cost spanning tree.

- Begin adding edges:
  - AB $4 OK
  - AE $5 OK
  - BE $6 reject – closes circuit ABEA
  - DC $7 OK
  - AC $8 OK

- Since every vertex is now connected, stop at this point. We have formed a spanning tree with cost $24.