# Chapter 16 – Graphs

<table>
<thead>
<tr>
<th>Graph Categories</th>
<th>Strong Components</th>
</tr>
</thead>
<tbody>
<tr>
<td>Example of Digraph</td>
<td>Graph G and Its Transpose $G^T$</td>
</tr>
<tr>
<td>Connectedness of Digraph</td>
<td>Shortest-Path Example (2 slides)</td>
</tr>
<tr>
<td>Adjacency Matrix</td>
<td>Dijkstra Minimum-Path Algorithm (2 slides)</td>
</tr>
<tr>
<td>Adjacency Set</td>
<td>Minimum Spanning Tree Example</td>
</tr>
<tr>
<td>vertexInfo Object</td>
<td>Minimum Spanning Tree: vertices A and B</td>
</tr>
<tr>
<td>Vertex Map and Vector vInfo</td>
<td>Completing the Minimum Spanning-Tree with Vertices C and D</td>
</tr>
<tr>
<td>VtxMap and Vinfo Example</td>
<td>Summary Slides (4 slides)</td>
</tr>
<tr>
<td>Breadth-First Search Algorithm (2 slides)</td>
<td></td>
</tr>
<tr>
<td>Dfs()</td>
<td></td>
</tr>
</tbody>
</table>
Graph Categories

- A graph is *connected* if each pair of vertices have a path between them.
- A *complete graph* is a connected graph in which each pair of vertices are linked by an edge.

![Graph Categories Diagram]

(a: Connected)  (b: Disconnected)  (c: Complete)
Example of Digraph

- Graph with ordered edges are called *directed graphs* or *digraphs*

Vertices \( V = \{ A, B, C, D, E \} \)
Edges \( E = \{(A,B), (A,C), (A,D), (B,D), (B,E), (C,A), (D,E)\} \)
Connectedness of Digraph

- **Strongly connected** if there is a path from any vertex to any other vertex.
- **Weakly connected** if, for each pair of vertices $v_i$ and $v_j$, there is either a path $P(v_i, v_j)$ or a path $P(v_i, v_j)$.

![Diagram of connectedness types](image)

- **(a)** Not Strongly or Weakly Connected
  (No path $E$ to $D$ or $D$ to $E$)

- **(b)** Strongly Connected

- **(c)** Weakly Connected
  (No path from $D$ to a vertex)
An m by m matrix, called an *adjacency matrix*, identifies the edges. An entry in row i and column j corresponds to the edge e = (vᵢ, vⱼ). Its value is the weight of the edge, or -1 if the edge does not exist.
Adjacency Set

(a) Vertices: A, B, C, D, E
Set of Neighbors:
- A: B
- B: C, D
- C: B, E
- D: E
- E: B

(b) Vertices: A, B, C, D, E
Set of Neighbors:
- A: B, C, D, E
- B: A, C
- C: B, D
- D: C, E
- E: C, D
A vertexInfo object consists of seven data members. The first two members, called vtxMapLoc and edges, identify the vertex in the map and its adjacency set.
To store the vertices in a graph, we provide a `map<T,int>` container, called `vtxMap`, where a vertex name is the key of type `T`. The int field of a map object is an index into a vector of `vertexInfo` objects, called `vInfo`. The size of the vector is initially the number of vertices in the graph, and there is a 1-1 correspondence between an entry in the map and a `vertexInfo` entry in the vector.
VtxMap and Vinfo Example

- VtxMap

  - A: 0
  - B: 1
  - C: 2
  - D: 3

- Vinfo

  - 0(A): 1
  - 1(B): 1
  - 2(C): 1
  - 3(D): 1

- VtxMap

  - locA: edges, inDegree = 1
  - locB: edges, inDegree = 1
  - locC: edges, inDegree = 2
  - locD: edges, inDegree = 1
Breadth-First Search Algorithm

VisitQueue: B C G
VisitSet: A

VisitQueue: C G D
VisitSet: A B

VisitQueue: G D
VisitSet: A B C
Breadth-First Search... (Cont.)

VisitQueue: D
VisitSet: A, B, C

VisitQueue: F
VisitSet: A, B, C

VisitQueue: E, F
VisitSet: A, B, C

VisitQueue: G, D, E
VisitSet: A, B, C

VisitQueue: G, D, E
VisitSet: A, B, C, F
 dfs() 

(a) v = A, w = B
First call to dfsVisit() starts at D.
Second call to dfsVisit() starts at A.
dfsList: A D C B

(b) v_S = A, v = B, w = C
dfsVisit() starts at B, discovers neighbor D, and then neighbor C.
dfsList: A B D C

(c) v_S = A, v = B, w = C
dfsVisit() starts at A, discovers C, discovers D, and finally discovers B. The neighbor of B (C) is GRAY (a back edge).
dfsList: <empty>
path C D B C is a cycle

(d) v_S = A, v = B, w = C
dfsVisit() starts at A and proceeds along the path from B to C.
dfsList: A B C
A *strongly connected component* of a graph $G$ is a maximal set of vertices $SC$ in $G$ that are mutually accessible.
The transpose has the same set of vertices $V$ as graph $G$ but a new edge set $E^T$ consisting of the edges of $G$ but with the opposite direction.
The shortest-path algorithm includes a queue that indirectly stores the vertices, using the corresponding vInfo index. Each iterative step removes a vertex from the queue and searches its adjacency set to locate all of the unvisited neighbors and add them to the queue.
Example: Find the shortest path for the previous graph from F to C.
Dijkstra Minimum-Path Algorithm From A to D Example

minInfo(B,4)  minInfo(C,11)  minInfo(E,4)

priority queue
Dijkstra Minimum-Path Algorithm From... (Cont...)

minInfo(C,10)  minInfo(C,11)  minInfo(E,4)  minInfo(D,12)

priority queue

minInfo(C,10)  minInfo(C,11)  minInfo(D,12)

priority queue

minInfo(D,12)

priority queue
Minimum Spanning Tree Example

Network of Hubs

Minimum spanning tree

Minimum amount of cable = 241
Minimum Spanning Tree: Vertices A and B

Spanning tree with vertices A, B
\( \text{minSpanTreeSize} = 2, \text{minTreeWeight} = 2 \)
Completing the Minimum Spanning-Tree Algorithm with Vertices C and D

Spanning tree with vertices A, B, D
minSpanTreeSize = 3, minTreeWeight = 7

Spanning tree with vertices A, B, D, C
minSpanTreeSize = 4, minTreeWeight = 14
§- Undirected and Directed Graph (digraph)

- Both types of graphs can be either weighted or nonweighted.
§- Breadth-First, bfs()

- locates all vertices reachable from a starting vertex
- can be used to find the minimum distance from a starting vertex to an ending vertex in a graph.
§- Depth-First search, dfs()

- produces a list of all graph vertices in the reverse order of their finishing times.

- supported by a recursive depth-first visit function, dfsVisit()

- an algorithm can check to see whether a graph is acyclic (has no cycles) and can perform a topological sort of a directed acyclic graph (DAG)

- forms the basis for an efficient algorithm that finds the strong components of a graph
§- Dijkstra's algorithm

- if weights, uses a priority queue to determine a path from a starting to an ending vertex, of minimum weight

- This idea can be extended to Prim's algorithm, which computes the minimum spanning tree in an undirected, connected graph.