Implementation

• How do we represent graphs with a computer?

• Adjacency Matrix:

\[
\begin{array}{ccccc}
A & B & C & D & E \\
\hline
A & 0 & 240 & 0 & 0 & 0 \\
B & 0 & 0 & 1100 & 698 & 0 \\
C & 1200 & 1100 & 0 & 0 & 0 \\
D & 0 & 0 & 0 & 0 & 743 \\
E & 0 & 0 & 0 & 743 & 0 \\
\end{array}
\]

\[
\begin{array}{ccccc}
A & B & C & D & E \\
\hline
A & \infty & 240 & \infty & \infty & \infty \\
B & \infty & \infty & 1100 & 698 & \infty \\
C & 1200 & 1100 & \infty & \infty & \infty \\
D & \infty & \infty & \infty & \infty & 743 \\
E & \infty & \infty & \infty & 743 & \infty \\
\end{array}
\]
Multidimensional Array based

- **Adjacency matrix:**
  - **Advantages:**
    - Easy to maintain.
    - Determination if an edge exists between two vertices is simple and efficient.
  - **Disadvantages:**
    - Might be difficult to avoid representing absent nodes
      - 0 might be a valid edge and infinity is not valid
    - Consume a lot of space. Especially for disconnected/non-complete or sparse graphs.
    - Might not be possible due to number of Nodes. Memory consumed is proportional to the square of the number of nodes. 100,000 nodes would require more memory than a 32-bit addressable machine could provide.
Reference based

• Adjacency List:

  - B 240
  - C 1100
  - A 1200
  - E 743
  - D 743
  - D 698
  - B 1100

• Disadvantages
  – Complicated to administer and maintain
  – Less efficient at determining if edges exist.

• Advantages
  – Great for sparsely connected graphs.
  – Efficient at determining which vertices are connected to a particular vertex.
Rules based

• Some problems present graphs that are too large to store in their entirety.
  – Chess and Go are two such examples

• Instead of storing actually edges we can store, code or otherwise implement rules that define what is connected to what.
  – In chess we know that one board state is adjacent to another if we can move from one to other with a single, well defined, valid move.

• Each graph node then is an object that contains:
  • A reference to a object that contains the label, state or information for the node.
  • A linked list of references to other nodes (these are the edges.

• In any case it is important to choose your implementation wisely on the basis of the project and goal’s requirements and your available resources.
Traversing Graphs

- It is often necessary to visit all the nodes in a graph. We therefore desire algorithms capable of traversing all the nodes of a graph.
- Similar to pre-order, in-order traversals of BSTs
- There are two distinct ways of thinking about graph traversal:
  - Depth first
  - Breadth first
Connected Components

• Only guaranteed to visit all the nodes if the graph is connected. (there exists a path from any vertex to any other vertex.)

• If a graph isn’t connected these traversals will only traverse those nodes that are connected to the starting node \( v \).

• More terminology: The subset visited is termed the “connected component” containing \( v \).
Cycles

• If a graph contains cycles then it is possible for simple traversal algorithms to loop indefinitely; visiting the same node(s) repeatedly.

• To prevent this nodes are “marked” when they are first visited and the traversal never visits a marked node.
  – Nodes can be marked by setting true in an array or by the presence of the Node as a key in some Hash or Tree map.
Depth First

- The concept behind depth first search is to visit nodes as “deeply” into the graph as quickly as possible:

- Nodes in this graph are labeled in the order they are visited by a Depth First Traversal
Recursive DFS

• Algorithm:
  • Dfs(v)
    {
      mark v as visited;
      for (each unvisited vertex u adjacent to v)
        dfs(u)
    }
Iterative DFS

- $Dfs(v)$
  
  ```
  Stack s = new Stack();
  s.push(v);
  mark v as visited;
  while (!s.isEmpty())
    if (s.peek() has no unvisited, adjacent nodes)
      s.pop(); // done with node on top
    else {
      u = select unvisited, adjacent node to s.peek();
      s.push(u);
      mark u as visited;
    }
  ```
Breadth First Search

• The concept behind breadth first search is that nodes are visited as soon as they are found, or can be reached:

• Nodes are now labeled with the order visited and the minimum distance from the start.

![Breadth First Search Diagram]

10 21 31 41 52 62 73 83 94
Iterative BFS

- Bfs(v)
  {
    Queue q = new Queue();
    s.enqueue(v);
    mark v as visited;
    while (!s.isEmpty()) {
      w = q.dequeue(); // working on nodes reachable
      // from node w
      // do any processing of w needed.
      for (each unvisited node u adjacent to w) {
        mark u as visited
        q.enqueue(u);
      }
    }
  }