Graph Algorithms

A graph \( G = (V, E) \) contains a set of \( \overline{\text{vertices}} \) \( V \), a set of \( \overline{\text{edges}} \) \( E \). Each edge \( e \in E \) is a pair of vertices \( (v_i, v_j) \) where \( v_i \in V \), \( v_j \in V \). The edge \( e \) "connects" \( v_i \) and \( v_j \). If the pair \( (v_i, v_j) \) is an ordered pair, \( G \) is said to be a directed graph, otherwise it is an undirected graph.

In some graphs (directed or undirected), the edges carry some weight or cost. Such graphs are called \( \overline{\text{weighted}} \) graphs. (*vertices are also called nodes*)

Example:

1. Let people with Facebook accounts be the nodes (or vertices) of a graph. Two nodes are connected by an edge if they represent people who are Facebook friends.

2. Let each city be a node. Two nodes are connected by an edge if there is a direct flight that runs between the two cities.

3. Let every course at DAICT be represented by the node of a graph. We can put a directed edge from course A to course B.
A path in a graph is a sequence of vertices $V_1, V_2, \ldots, V_n$, such that $(V_i, V_{i+1}) \in E$ for all $i$, such that $1 \leq i < n$. The length of a path is the number of edges on the path.

$G = (V, E)$
A **path** in a graph $G = (V, E)$ is a sequence of vertices $W_1, W_2, \ldots, W_n$ such that $(W_i, W_{i+1}) \in E$ for all $i$, such that $1 \leq i < N$. The **length** of the path is the number of edges on the path.

In some cases, we can have a path from a vertex to itself. Such a path has length 0. A path is said to be **simple** if all vertices on that path are distinct (except for possibly the first and last ones, which could be the same).
G = (V, E) where V = \{A, B, C, D, E, F, G\}

E = \{(A, B), (A, D), (B, D), (B, C), (C, D), (D, E), (C, E), (C, F), (E, F), (F, G), (B, G)\}

(simple path)

example of a path: A, B, C OR

A, B, C, E, F, G, B, A (not a simple path)

In a directed graph, the vertex sequence must follow the edge directions.

eg.: Graph 2

A, B, D, E, F, G is a valid path, but G, F, C, B, A is not.
A path of length >1 whose first and last vertices are the same is called a cycle.

eg: ABDA in graph 1, ADCBA in graph 2.

In an undirected graph, a cycle must also contain strictly distinct edges.

eg: ABA is not a valid cycle in Graph 1 as AB and BA are considered the same edge.

ABA is a valid cycle in graph 2 as AB and BA are DISTINCT edges in Graph 2.

A graph with no cycles is said to be acyclic. (Any tree is an acyclic graph!)

Connected graphs

An graph is said to be connected if there exists a path between any given pair of vertices in the graph.

eg: Graph 1 is connected. Here is a graph that is not connected.
A directed graph which has a path from any vertex to every other vertex is said to be strongly connected. A directed graph that is not strongly connected but whose underlying undirected graph (obtained by removal of the edge directions) is connected, is "weakly connected".

E.g.: graph 2 is not strongly connected (why?) but it is weakly connected.

Here is a graph that is strongly connected nor weakly connected.

Here is a graph that is strongly connected (and hence weakly connected).

Degree of a vertex

The number of edges incident upon a vertex in an undirected graph is called the degree of the vertex.
eg. The degree of the vertices in graph 1 are as follows:
\[ d(A) = 2, \ d(B) = 4, \ d(C) = 4, \ d(D) = 4, \ d(E) = 3, \ d(F) = 3, \ d(G) = 2. \]

For a directed graph, the in-degree of a vertex is the number of edges entering that vertex. The out-degree is the number of edges emanating from the vertex.

eg: Graph 2
\[ id(A) = 1, \ id(B) = 2, \ id(C) = 2, \ id(D) = 2, \ id(E) = 2, \ id(F) = 2, \ id(G) = 2. \]
\[ od(A) = 2, \ od(B) = 2, \ od(C) = 3, \ od(D) = 2, \ od(E) = 1, \ od(F) = 1, \ od(G) = 0. \]

**Graph Representation:**

**Adjacency matrix**

A graph \( G = (V, E) \) can be represented as a matrix of size \(|V| \times |V|\) — denoted as ‘A’ where \( A[i][j] = 1 \) if \((Vi, Vj) \in E\)
\[ = 0 \quad \text{if} \quad (Vi, Vj) \notin E. \]

This is a simple representation, but can be
very expensive in terms of memory requirements especially if the graph is "sparse". For instance, the graph of people on Facebook is very sparse - most people have fewer than 10000 friends whereas there are millions of Facebook users.

Adjacency list

In this representation, we maintain an array of vertices in \( V \). Against each vertex \( v \), we keep a list of all those vertices \( w \in V \) \( (w \neq v) \) such that \( vw \in E \).

In an undirected graph, each edge \( e = (v, w) \) \( \in E \) appears in two lists - one for \( v \), one for \( w \). An adjacency list facilitates more efficient retrieval of all the immediate neighbors of any given vertex \( v \) and \( w \) are said to be "immediate neighbors" if \( vw \in E \) - \( O(d(v)) \) - as opposed to \( O(|V|) \) in case of an adjacency matrix.
For graph 1, the adjacency matrix representation is as follows:

<table>
<thead>
<tr>
<th></th>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
<th>E</th>
<th>F</th>
<th>G</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>B</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>C</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>D</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>E</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>F</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>G</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
</tbody>
</table>
template <class T>
class Graph - AM
{
  private:
    int nv, ne; // # of vertices/edges
    T ** am; // adj. matrix
}

public:
  Graph - AM (int V, int E)
  {
    nv = V; ne = E;
    alloc_2d_array (nv, nv, am,
      for (int i = 0; i < nv; i++)
        for (int j = 0; j < nv; j++)
          am[i][j] = 0;
  }

void alloc_2d_array (T ** &x, int r, int c)
{
  x = new T*[r]; // array of pointers
  for (int i = 0; i < r; i++)
    x[i] = new T[c]; // actual chunks
  // of datatypes "T".

  // X

  template <class T>
class Graph - AL
{
  private:
    int nv, ne;
    LinkedList<T> * al;
  }

  // PTO
Graph traversal

Given a graph, one of the key operations is to display the vertices in the graph, i.e., traverse the graph. There are two main methods for this: breadth-first search (BFS) and depth-first search (DFS).

BFS: In this method, you visit a vertex, print the data at the vertex (or the vertex number) and immediately push all its immediate neighbors into a queue. Then vertices are deleted one by one from the queue, their numbers are printed and their unvisited children are pushed into the queue. Each time the data of any node is printed, it is marked as "visited".
A BFS on Graph 1 is given as follows:

- Visit A — push B, D into a queue
- Visit B — push C, G into the queue
- Visit D — push E into the queue
- Visit C — push F into the queue
- Visit G — push nothing into the queue

BFS: A B D C G E F

Pseudocode:

```
void BFS(V)

1. V = set of vertices of the graph
2. V = first vertex from V; any vertex can be regarded as "first"
3. add V to Q; push V into a queue.

while (!empty(Q))

1. W = delete Q();
2. U = vertex adjacent to w in the graph;
3. while (U)
   1. if (U is not visited)
      1. add U to Q; mark U as visited;
      2. U = next vertex adjacent to w;
```
Note: the aforementioned pseudocode will print only those vertices that are reachable from the "first vertex" \( v_0 \). Thus, if the graph is not connected, it will leave some vertices unvisited. In fact, BFS is therefore often used as a test to check whether the given graph is connected.

**Complexity:**

**Time complexity** = \( O(|V|^2) \) for adjacency matrix

\[
= O\left(\sum_{i=1}^{V} d_i\right) \text{ (where } d_i \text{ = degree of } i^{th} \text{ vertex)}
\]

for an adjacency list.

**Space complexity** = \( O(1) \) for adj matrix

\[
= O(|V|) \text{ for adj list}
\]

Note: BFS is very similar to levelwise traversal in a binary tree.
Note: the aforementioned pseudocode will print only those vertices that are reachable from the "first vertex" \( v \). Thus if the graph is not connected, it will leave some vertices unvisited. In fact BFS is therefore often used as a test to check whether the given graph is connected.

**Complexity:**

Time complexity = \( O(|V|^2) \) for adjacency matrix

\[
= O\left(\sum_{i=1}^{\mid V\mid} d_i\right) \text{ (where } d_i = \text{ degree of } i^{th} \text{ vertex)}
\]

for an adjacency list.

Space complexity = \( O(1) \) for adj matrix

\[
= O(|V|) \text{ for adj list}
\]

Note: BFS is very similar to levelwise traversal in a binary tree.