Introduction to Graphs

1. Introduction

• Definition: A graph G consists of two sets V and E, where V is a set of vertices (or nodes) and E is a set of edges. G is denoted as G=(V, E).

• In an undirected graph, the pair of vertices representing an edge is unordered, i.e. (u,v) is the same as (v,u).

• In a directed graph, the pair of vertices representing an edge is ordered, i.e. <u,v> is different from <v,u>.

• Example:
  - undirected graph \( G = (V, E) = (\{1,2,3,4\},\{(1,2) (1,3) (1,4) (2,3) (2,4)\}) \)
  - directed graph \( G' = (V',E') = (\{1,2,3\}, \{<1,2> <2,1> <2,3>\}) \)
  - E may be an empty set, i.e. no edges.
  - Undirected graph may not have self-loops. Also, assume no multi-edges from a node u to another node v

• Note: Multigraph may have self-loops and multi-edges

• Maximum number of edges of an undirected graph with n nodes is \( n \cdot (n-1)/2 \)

• Maximum number of edges of a directed graph with n nodes is \( n \cdot (n-1) \) // assume no self loops
• Graphs with maximum number of edges are called **complete graphs**

• In an undirected graph, if \((u,v)\) is an edge, then we say that \(u\) and \(v\) are **adjacent** and the edge \((u,v)\) is **incident** on vertices \(u\) and \(v\)

• In a directed graph, if \(<u,v>\) is an edge, then we say that \(v\) is **adjacent to** \(u\) and \(u\) is **adjacent from** \(v\) (Alternatively, \(v\) is a **successor** of \(u\) and \(u\) is a **predecessor** of \(v\)). The edge \(<u,v>\) is **incident** to both vertices \(u\) and \(v\).

• If \(G=(V,E)\) is a graph, then \(G'=(V',E')\) is a **subgraph** of \(G\) if \(V' \subseteq V\) and \(E' \subseteq E\) and every edges in \(E'\) is incident to vertices in \(V'\). Example:

  ![Graph](image)

  - If \(G\) is an undirected graph, a **path** from a vertex \(v_i\) to \(v_j\) is a sequence of vertices \(v_i, v_{k1}, v_{k2}, \ldots, v_{km}, v_j\) such that \((v_i, v_{k1}), (v_{k1}, v_{k2}), \ldots, (v_{km}, v_j)\) are edges in \(G\). Example: \(v_1, v_3, v_2, v_4\)

  - If \(G\) is a directed graph, a **path** from a vertex \(v_i\) to \(v_j\) is a sequence of vertices \(v_i, v_{k1}, v_{k2}, \ldots, v_{km}, v_j\) such that \(<v_i, v_{k1}>, <v_{k1}, v_{k2}>, \ldots, <v_{km}, v_j>\) are edges in \(G\).

  - The **length** of a path is the number of edges in it. Example: a path \(v_1, v_2, v_4\) has length 2

  - A **simple path** is a path in which all vertices except possibly the first and the last are distinct

  - A **cycle** is a simple path in which the first and the last vertices are the same. Example: \(v_1, v_2, v_4, v_1\) is a cycle

![Diagram](image)
In an undirected graph $G$, two vertices $u$ and $v$ are said to be connected if there is a path from $u$ to $v$. An undirected graph $G$ is said to be connected if every pair of vertices is connected. A connected component of a graph $G$ is a maximally connected subgraph of $G$.

An undirected graph $G$ is said to be a tree if it is connected and has no cycle (acyclic). An undirected graph is said to be a forest if it consists of several trees.

$G'$ is said to be a spanning tree (spanning forest) of an undirected graph if

- $G'$ is a subgraph of $G$
- $G'$ consists of all vertices if $G$
- $G'$ is a tree (forest)
• A directed graph is said to be strongly connected if for every pair of vertices u and v, there is a directed path from u to v and from v to u.

• Strongly connected component is a maximally subgraph that is strongly connected.

\[ G : \text{strongly connected} \]

\[ G : \quad V_1 \xrightarrow{} V_2 \xrightarrow{} V_3 \xrightarrow{} V_4 \xrightarrow{} V_5 \]

\[ \text{Strongly connected components of } G : \]

\[ V_1 \xrightarrow{} V_2 \quad V_3 \quad V_4 \xrightarrow{} V_5 \]

• In an undirected graph, the degree of a vertex is the number of edges incident to the vertex.

• In a directed graph, the in-degree of a vertex is the number of edges coming into the vertex. The out-degree of a vertex is the number of edges leaving the vertex.

• If each edge in a graph is labeled with numeric values, the graph is called weighted graph.

\[ V_3 \xrightarrow{5} V_4 \]

\[ V_2 \xrightarrow{7} V_3 \]

\[ V_2 \xrightarrow{8} V_4 \]
2. ADT Graph

- Example ADT graph operations:

  Note: assume each vertex contains a search key or value.

  | Create an empty graph |
  | Destroy a graph       |
  | Determine whether a graph is empty |
  | Determine the number of edges in a graph |
  | Determine the number of vertices in a graph |
  | Determine whether an edge exists between two given vertices |
  | Insert a new vertex (with new search key) into a graph |
  | Insert an edge between two given vertices in a graph |
  | Delete a particular vertex from a graph and any edges between the vertex and other vertices |
  | Delete an edge between two given vertices in a graph |
  | Retrieve from a graph the vertex that contains a given search key |

- Graph Representation

  - Given a graph, each vertices are usually represented by integers 0, 1, …, n-1; i.e. 0 corresponds to $v_1$, 1 corresponds with $v_2$, …, n-1 corresponds with $v_n$.

  - **Adjacency Matrix** $A$ is an $n$ by $n$ matrix such that $A(i, j) = 1$ if and only if $(v_i, v_j) \in E$

    For directed graph, $<v_i, v_j> \in E$.

    For undirected graph, $A(i,j) = 1$ if and only if $A(j,i) = 1$
• Example:

Matrix for directed graph $G$. Assume the 1st row is for $V_1$, 2nd row for $V_2$ etc:

\[
\begin{array}{cccc}
0 & 1 & 1 & 0 \\
1 & 0 & 0 & 0 \\
1 & 0 & 0 & 0 \\
1 & 0 & 0 & 0 \\
0 & 0 & 0 & 1 \\
\end{array}
\]

Matrix for undirected graph $G$:

\[
\begin{array}{cccc}
0 & 1 & 1 & 1 \\
1 & 0 & 1 & 1 \\
1 & 1 & 0 & 1 \\
1 & 0 & 1 & 1 \\
0 & 1 & 1 & 1 \\
\end{array}
\]

• For undirected graphs, adjacency matrix is symmetric about the diagonal.

• For undirected graphs, the row sum = the column sum is the degree of a vertex.

• For directed graphs, the row sum is the out-degree of a vertex and the column sum is the indegree of a vertex.

• Discussions:
  - To set-up a graph, it needs to set up all entries of the matrix $O(n^2)$.
  - Can quickly find out whether or not 2 vertices are adjacent $O(1)$.
  - To find out all the vertices that are adjacent to a particular vertex, need to look at whole row $O(n)$.
  - To look at all edges in the matrix, $O(n^2)$.
  - Not a good representation for sparse graph (graph with very few edges).
• Adjacency List: replace $n$ rows, i.e. $n$ vertices, of adjacency matrix with $n$ linked lists.

```c
typedef struct node {
    int vertex;
    struct node * next;
} ;
```

```c
struct node * graph[Max_Vertices];
```

• For an undirected graph with $n$ nodes and $e$ edges. It needs $n$ entries in array graph and $2e$ list nodes.

• Example: For the above undirected graph $G$.

For the above directed graph $G$ (store outgoing edges).
• May add a counter for each list to count the number of edges incident to the node

• For each edge (u,v) in an undirected graph, it appears in both list u and list v

• For a directed graph, may store edges in two ways, incoming edges or outgoing edges

• Discussions
  - To set-up a graph, it takes $O(n+e)$. If $e \ll n^2$, then it is faster
  - To examine all edges $O(e+n)$
  - To find out if (u,v) is an edge, may take $O(n)$
  - A good representation for sparse graphs, but not good for dense graphs
3. A Simple ADT Graph Implementation

// **************************************************************************************************
// An adjacency list representation of an undirected graph.
// Modify from the version from the text book
// **************************************************************************************************

#include <vector>
#include <list>
#include <set>
using namespace std;

// an edge is (v,w)
class Edge
{
public:
    int v, w;
    Edge(int firstVertex, int secondVertex)
    {
        v = firstVertex;
        w = secondVertex;
    }  // end constructor
};  // end Edge

// **************************************************************************************************
// undirected graph definition file
// **************************************************************************************************

class Graph
{
public:
    int numVertices;  // number of vertices in the graph
    int numEdges;     // number of edges in the graph

    // Adjacency list representation of the graph;
    // the set consists of the second vertex (key)
    vector<set<int> > adjList;

    // Postcondition: The graph is initialized to hold n vertices.
    Graph(int n);

    // Determines the number of vertices in the graph.
    int getNumVertices() const;

    // Determines the number of edges in the graph.
    int getNumEdges() const;
};
// add an edge into graph
void add(Edge e);

// remove an existing edge from graph
void remove(Edge e);

// check if an edge is in graph
bool findEdge(Edge e) const;

};
// End of header file

// ****************************************
// Implementation file Graph.cpp
// ****************************************

Graph::Graph(int n)
{
    adjList = vector<set<int> >(n);
    numVertices = n;
    numEdges    = 0;
}  // end constructor

int Graph::getNumVertices() const
{
    return numVertices;
}  // end getNumVertices

int Graph::getNumEdges() const
{
    return numEdges;
}  // end getNumEdges

void Graph::add(Edge e)
{
    adjList[e.v].insert(e.w);
    adjList[e.w].insert(e.v);
    numEdges++;
}  // end add

void Graph::remove(Edge e)
{
    adjList[e.v].erase(e.w);
    adjList[e.w].erase(e.v);
    numEdges--;
}  // end remove
bool Graph::findEdge(Edge e) const
{
    set<int> m = adjList[e.v];
    set<int>::iterator iter = m.find(e.w);
    if (iter == m.end())
        return 0;
    else
        return 1;
}  // end findEdge
// end of implementation file

// simple test
int main()
{
    Graph g(5);
    Edge   e1(1,2), e2(2,3), e3(3,4), e4(1,4), e5(3,2);

    cout << "Add four edges (1,2), (2,3), (3,4) and (1,4)\n";
    g.add(e1); g.add(e2); g.add(e3); g.add(e4);

    cout << "Number of vertices in graph : " << g.getNumVertices() << endl;
    cout << "Number of edges in graph : " << g.getNumEdges() << endl;

    cout << "Remove an edge (3,2)\n";
    g.remove(e5);
    cout << "Number of edges in graph : " << g.getNumEdges() << endl;

    cout << "Is edge (3,2) in graph? " << g.findEdge(e5) << endl;
    cout << "Is edge (1,4) in graph? " << g.findEdge(e4) << endl;
}

OUTPUT :
Add four edges (1,2), (2,3), (3,4) and (1,4)
Number of vertices in graph : 5
Number of edges in graph : 4
Remove an edge (3,2)
Number of edges in graph : 3
Is edge (3,2) in graph? 0
Is edge (1,4) in graph? 1
4. Graph Traversals

- Recall: tree traversals: visit all nodes (preorder, inorder and postorder)
- Graph: also visit all nodes (vertices)

- **Depth-First Search** // visit all vertices that are reachable from v

  ```java
  dfs(in v:vertex)
  {
      // Traverses a graph beginning at vertex v using dfs
      mark v as visited // may use an array visited[] to indicate this
      for (each unvisited vertex u adjacent to v)
          dfs(u)
  }
  ```

Example:

Assume vertices in adjacency list are stores in alphabetical order.

The order of visiting is:

a, b, d, h, e, f, c, g (do not reach i and j)

Assume directed graph (edges point downward direction).
The order if visiting is:

a, b, d, h, e, c, f, g (do not reach i and j)

- Running time: \(O(n+e)\) if we are using adjacency list, \(O(n^2)\) if we are using adjacency matrix

- Note: dfs() algorithm can be used (with minor modification) to determine many problems such as “whether a graph has a cycle” and “connected components of a graph”
• **Breadth-First Search** // visit all vertices adjacent to a vertex before going forward

```
bfs(in v:vertex)
{
    q.createQueue()
    mark v as visited         // may use an array Visited[]
    q.enqueue(v)              // add v into q

    while(!q.isEmpty())
    {
        d.dequeue(w)   // delete 1st element w from Queue
        for (each unvisited vertex u adjacent to w)
        {
            mark u as visited
            q.enqueue(u)
        }
    }
}

• Example

Assume vertices in adjacency list are stores in alphabetical order.

The order of visiting is:
- a, b, c, d, e, f, g, h (do not reach i and j)

Assume directed graph (edges point downward direction).
The order if visiting is:
- a, b, c, d, e, f, g, h (do not reach i and j)

• Running time: O(n+e) if we are using adjacency list, O(n^2) if we are using adjacency matrix
• Note: Just like dfs(), bfs() algorithm can also be used (with minor modification) to determine many problems.
5. Application of Graphs

5.1 Connected Components

- Is an undirected graph connected?

  Just use DFS(v) or BFS(v) starts with any node v. At the end of DFS() or BFS(), check if there are any unvisited nodes. If yes → “not connected”; otherwise, all nodes are visited → “connected”

- Get all connected components of an undirected graph.

  Assume n nodes are labeled 0,1,2,…, n-1

  for (j=0; j<n, j++)
  if ( node j is not visited) DFS(j);

  Example:

  ![Graph Diagram]

  Note : Each time DFS() is called, it visits all nodes in the same connected component. The “for loop” checks to make sure all nodes are visited. The total time is O(n+e) for adjacency list.
5.2 Topological Sorting

- Assume that we represent the course structures in a school by a directed graph. Let a node represents a course and an edge \( <u,v> \) means “course \( u \) is prerequisite of course \( v \)”.

Goal: to list all courses in an order such that all prerequisites are satisfied. Such a linear order is called a topological order. Arranging the vertices into topological order is called topological sorting.

- Note: Only for the directed acyclic (no cycles) graph.

- Example:

```
<table>
<thead>
<tr>
<th></th>
<th>a</th>
<th>b</th>
<th>c</th>
<th>d</th>
<th>e</th>
<th>f</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td></td>
<td></td>
<td></td>
<td>&lt;d&gt;</td>
<td></td>
<td>&lt;f&gt;</td>
</tr>
<tr>
<td>b</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>c</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>d</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>e</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>f</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```

- a, g, d, b, e, c, f

- a, b, g, d, e, f, c
15. Introduction to Graphs

- Algorithm #1

  // Input a graph with n vertices
  // For each loop, include a new vertex without successors into the
  // beginning of the list
  // Output a topological order in the list

  for (j = 1 to n)
  {
    select a vertex v that has no successors
    insert v into the beginning of a list
    delete v and its edges from the graph
  }

  For example above, one possible solution:

  f c e b d g a ← insertList() // reverse order

- Algorithm #2

  // Input a graph with n vertices
  // for each loop, include a new vertex v without predecessors into a queue
  // delete v and its edges from the graph
  // Output a topological order in the queue, i.e. from the front to the
  // rear of the queue

  for (j = 1 to n)
  {
    select a vertex v that has no predecessor
    insert v into the queue
    delete v and its edges from the graph
  }

  For example above, one possible solution:

  g a d b e f c ← insertQueue()
5.3 Spanning Trees

- A connected graph (undirected graph only) with \( n \) vertices has at least \( n-1 \) edges.
- If a connected graph with \( n \) vertices has exactly \( n-1 \) edges, then it is a tree.
- If a connected graph with \( n \) vertices has more than \( n-1 \) edges, then it has a cycle.
- Spanning tree is a minimum subgraph \( G' = (V',E') \) of a connected graph \( G = (V,E) \) such that \( V' = V \) and \( G' \) is connected, i.e. \( |E'| = |V| - 1 \) edges.

- Example: two spanning trees of graph \( G \)

G :

- Algorithm: modify DFS() and BFS() to include edges in the spanning tree.

Note: Assume the input graph is a connected graph.

```plaintext
DFS_SpanningTree(v)
mark v as visited
for (each unvisited vertex u adjacent to v)
    mark (u,v) // add this statement
    DFS_SpanningTree (u)
```
BFS_SpanningTree(v)

mark v as visited
add v into Queue

while(Queue is not empty)
{
    delete 1st element w from Queue
    for (each unvisited vertex u adjacent to w)
    {
        mark u as visited
        mark (u,w) // add this statement
        insert u into Queue
    }
}

5.4 Minimum Spanning Trees

• Recall: there may be many spanning trees in a connected graph

• Assume that each edge has a weight (weighted graph). The cost of a spanning tree is the sum of the weights of all edges in the spanning tree.

• A minimum spanning tree is a spanning tree with the smallest cost.

• Kruskal’s algorithm
  // assume input graph is connected

start with a set S of n independent vertices
sort all edges in non-decreasing order of their costs
while (there are less than n-1 edges in S)
{
    delete the next minimum cost edge e
    if there is a cycle in S when e is included, discard e
    else add e into S
}
Note: CSC510 will cover this algorithm in detail.

Look at another algorithm, Prim’s algorithm in your text.
5.5 Other graph problems

- Shortest Paths: Given a weighted directed graph. There may be several paths between two vertices. The weight (or cost) of a path is the sum of the weights of all edges in the path. The shortest path between two vertices is the path that has the smallest weight.

- Is there a cycle in a given graph?

- Traveling salesperson problem: Given a weighted directed graph that represents a roadmap. The salesperson starts at a vertex v, visits every other vertex exactly once, and returns to vertex v. The cost of the path must be minimum.

- Graph coloring problem: Let G be an undirected graph and m be a positive integer. Is it possible to color nodes of G such that no two adjacent nodes have the same color and only m colors are used.

- There are many other graph problems…