Graphs

A graph is a non-linear data structure consisting of vertices (also known as nodes) and edges (links between vertices). Usually this is denoted as $G = <V, E>$, where $V$ is a set of vertices, and $E$ is a set of edges. Each edge is a pair $(x, y)$, where $x$ and $y$ belongs to $V$. If the edge pair is ordered, the edge is called directed and thus the graph is directed graph. Otherwise, the graph is called undirected. Sometimes, an edge has a component called edge cost (or weight).

Graphs have applications in many areas: a city map, airports, computer networks, electrical circuits, and so on.

Here is the directed weighted graph

![Directed Weighted Graph](image)

The graph has 5 vertices

$V = \{v_1, v_2, v_3, v_4, v_5\}$

and 6 edges

$E = \{(v_1,v_2), \ (v_2,v_5), \ (v_5,v_3), \ (v_4,v_5), \ (v_4,v_3), \ (v_4,v_1)\}$

A graph is simple if there is no more than one edge between any two vertices. Otherwise, a graph is called multigraph. We also assume that a graph has no loops - an edge connecting the same vertex. In this course we will consider only simple graphs, without self-loops or multiple edges.
In a directed graph, an edge leaves its **origin** and ends in the **destination**. The **out-degree** of a vertex is the number of edges leaving the vertex. The **in-degree** of a vertex is the number of edges entering the vertex. The **degree** of a vertex is the number of edges entering and leaving the vertex. In the above example, the vertex $v_5$ has the in-degree $= 2$, and the out-degree $= 1$, thus the degree is 3. The **sink** is a node, which has out-degree 0. A **path** is a sequence of distinctive vertices connected by edges. The **path length** is the number of edges on the path. In the weighted graph, the length is the sum of the costs along the path. A **cycle** in a directed graph is a path that begins and ends at the same vertex. A **forest** is a graph without cycles. A graph is **connected**, if there is a path between any two vertices. A **tree** is a connected graph without cycles. An alternative definition, a tree is a connected forest.

**Proposition 1.**

In a directed graph, the sum of in-degrees (or out-degrees) is equal the number of edges.

*Reasoning:* In a directed graph, any edge $(x, y)$ contributes one unit to all in-degrees and one unit to all out-degrees.

**Proposition 2.**

In a connected graph with $m$ edges the sum of degrees of all vertices is $2*m$.

*Reasoning:* The proposition follows from the previous one.

**Exercise.** Suppose a simple graph has 15 edges, 3 vertices of degree 4, and all others of degree 3. How many vertices does the graph have? $3*4 + (x-3)*3 = 30$

**Exercise.** Given a graph with 7 vertices; 3 of them of degree two and 4 of degree one. Is this graph is connected? No, $3*2 + 4*1=10$ must have $10/2 = 5$ edges

**Proposition 3.**

In an undirected graph with $n$ vertices, there are at most $n*(n-1)/2$ edges.
In a directed graph with $n$ vertices, there are at most $n*(n-1)$ edges.

*Reasoning:* Consider an undirected graph with $n$ vertices and choose any vertex. The possible number of edges leaving this vertex is $n-1$. Take another vertex. The possible number of edges leaving this vertex is $n-2$ (don't count the edge between these vertices twice!), and so on. We have

$$(n-1) + (n-2) + ... + 1 = n(n-1)/2$$
For all graphs, the number of edges and vertices satisfies the following inequality
\(|E| < |V|^2\), meaning that the number of edges is always less than the number of vertices squared. If the number of edges is close to \(|V|^2\), we say that this is a dense graph, it has a large number of edges. Otherwise, this is a sparse graph. In most cases, the graph is relatively sparse.

**Proposition 4.**

*Let \(G\) be an undirected graph.*

- If \(G\) is connected, then \(|E| \geq |V| - 1\)
- If \(G\) is a tree, then \(|E| = |V| - 1\)
- If \(G\) is a forest, then \(|E| \leq |V| - 1\)

A complete graph is a graph in which every vertex is adjacent (i.e., connected) to every other vertex.

![Complete graph diagram](image)

A subgraph of a graph \(G\) is a graph whose vertices and edges are subsets of the vertices and edges of \(G\). If a graph is not connected, its maximal connected subgraphs are called connected components.

![Subgraph diagram](image)

A spanning tree of a graph is a subgraph, which is a tree and contains all vertices of the graph. An example,

![Spanning tree diagram](image)
Representation:

In a directed graph, vertex \( x \) is **adjacent** to vertex \( y \) if and only if there is an edge \((x, y)\) between them.

There are two standard ways to represent a graph:

- as a collection of adjacency lists
- or as an adjacency matrix

The adjacency-list representation is used for representation of the sparse graphs. An adjacency-matrix representation may be preferred when the graph is dense.

The adjacency-list representation of a graph \( G = \langle V, E \rangle \) consists of an array of linked lists, one for each vertices. Each such list contains all vertices adjacent to a chosen one. Here is an adjacency-list representation:

![Adjacency List Representation](image)

Vertices in an adjacency list are stored in an arbitrary order. A potential disadvantage of the adjacency-list representation is that there is no quicker way to determine if there is an edge between two given vertices. This disadvantage is eliminated by an adjacency-matrix representation.

For the adjacency-matrix representation we assume that the vertices are numbered \( 0, 1, 2, \ldots, N-1 \). The adjacency-matrix representation of a directed graph \( G = \langle V, E \rangle \) then consists of a \(|V| \times |V|\) matrix, such that

\[
\text{matrix}[i][j] = \begin{cases} 
\text{cost} \ (\text{or binary } 1), & \text{if there is an edge between } i \text{ and } j \\
\text{INFINITY} \ (\text{or binary } 0), & \text{otherwise}
\end{cases}
\]

Here is the adjacency-matrix for the graph on the first page:
The adjacency-matrix implementation:

It consists of two classes. The matrix class

```java
public class Matrix {
    private Double[][] M;
    protected final static Double INFINITY = Double.MAX_VALUE;

    public Matrix(int n) {
        M = new Double[n][n];
        for (int i = 0; i < n; i++)
            for (int j = 0; j < n; j++)
                M[i][j] = INFINITY;
    }

    //methods: get, set, toString
}
```

and the Graph class

```java
public class Graph extends Matrix {
    private int numEdges;
    private int numVertices;

    public Graph(int n) {
        super(n);
        numEdges = 0;
        numVertices = n;
    }

    //methods: addEdge, removeEdge, ...
}
```
Exercise 1.

Implement the method `inDegree`, which computes the in-degree of a given vertex.

Exercise 2.

Implement the method `removeEdge`, which removes an edge from a graph.

Exercise 3.

How many graphs are there which have \( n \) vertices? Don't count graphs with vertices connected to themselves.

Comparison of both representations:

<table>
<thead>
<tr>
<th>Adjacency List</th>
<th>Adjacency Matrix</th>
</tr>
</thead>
<tbody>
<tr>
<td>sparse (</td>
<td>E</td>
</tr>
<tr>
<td>Space</td>
<td>(O(</td>
</tr>
<tr>
<td>Edge access</td>
<td>(O(</td>
</tr>
</tbody>
</table>

Graph Traversal

To search a graph \( G \), we need to visit all vertices in a systematic order. We can choose any vertex as a starting point. Then we will systematically enumerate all vertices accessible from it. Because of a graph might contain cycles, we need some way for marking a vertex as having been visited. To do so we will have a boolean array `visited` with all elements initially set to false. We will set a correspondent element to true as soon as we visit a particular vertex. Also, we need to keep in mind that the graph might be disconnected. We need to have a container where we will store all unvisited adjacent vertices.

Here is an algorithm:

- remove a vertex from a container;
- print it;
- find all adjacent vertices;
- mark them as visited;
- enter them into a container;
- repeated as long as the container is not empty.

The algorithm runs in \(O(|V| + |E|)\) time.
There are two the most common traversals: depth-first order (DFT) and breadth-first order (BFT) traversals. On each step of DFT you go deeper and deeper into the graph. On each step of BFT you first visit all siblings and then their children. In case of BFT, we will use a queue and for DFT we will use a stack.

**The stack-container version (depth-first order).** Let us trace the algorithm using this graph:

![Graph Diagram]

We start with pushing 1 into the stack \( S = \{1\} \). Set \( \text{visited}[1] = \text{true} \). Then we pop 1 and push all unvisited adjacent vertices to \( S \). We also mark those vertices as visited: \( \text{visited}[2] = \text{true}; \text{visited}[3] = \text{true}; \text{visited}[4] = \text{true} \). The stack is \( S = \{4,3,2\} \), where the top is 4 and the bottom is 2. Observe that the order of pushing vertices to \( S \) is not important, however we chose an increasing order.

Since \( S \) is not empty, we pop 4 and then push all unvisited vertices to \( S \), which gives \( S = \{8,7,3,2\} \). We also mark them as visited \( \text{visited}[7] = \text{true}; \text{visited}[8] = \text{true} \).

We pop 8 and push all unvisited vertices (\( \text{visited}[6] = \text{true} \)) to \( S \), which gives \( S = \{6,7,3,2\} \).

We pop 6 and push all unvisited vertices (\( \text{visited}[5] = \text{true} \)) to \( S \), which gives \( S = \{5,7,3,2\} \).

We pop 5 and push all unvisited vertices (there are none) to \( S \), which gives \( S = \{7,3,2\} \).

We pop 7 and push all unvisited vertices (there are none) to \( S \), which gives \( S = \{3,2\} \).

We pop 3 and push all unvisited vertices (there are none) to \( S \), which gives \( S = \{2\} \).
We pop 2 and push all unvisited vertices (there are none) to S, which gives S = {}. The order in which we traverse the graph is 1, 4, 8, 6, 5, 7, 3, 2.

This is called a **depth-first order** traversal. Depth-first search (DFS) in an undirected graph is analogous to wandering in a labyrinth with a string and a can of paint.

**Remark.** DFT in the above description will visit all vertices in the **connected component** of a graph.

**Exercise.** Perform a DFS on the following graph (starting at vertex 1):

```
1------2------3------4
| \     |      |     /|
|   \   |      |   /  |
|     \ |      | /    |
5------6      7      8
|     /      / | \
|   /      /   |   \
| /      /     |     \
9------10-----11-----12
| \          / |      |
|   \      /   |      |
|     \  /     |      |
13-----14     15-----16
```

Observe that the result of traversal gives a spanning tree:
1 2 3 4 7 10 9 5 6 13 14 11 15 16 12 8

**backtracking from 6 to 9**

See *Graph.java* for the iterative and recursive implementations of the DFT.
The queue-container version (breadth-first order). We replace the container by the queue. We will add unvisited vertices at the end and remove them from the front. We again start with \( Q = \{1\} \), set \( \text{visited}[1] = \text{true} \). Then we pop 1, and enqueue all unvisited adjacent vertices to \( Q \). We also mark those vertices as visited: \( \text{visited}[2] = \text{true} \); \( \text{visited}[3] = \text{true} \); \( \text{visited}[4] = \text{true} \). The queue is \( Q = \{2, 3, 4\} \), where 2 is the front and 4 is the back. We enqueue vertices in an increasing order.

Since \( Q \) is not empty, we dequeue 2 and enqueue all unvisited vertices (\( \text{visited}[5] = \text{true} \); \( \text{visited}[6] = \text{true} \)) to \( Q \), which gives \( Q = \{3, 4, 5, 6\} \).

We dequeue 3 and enqueue all unvisited vertices (there are none) to \( Q \), which gives \( Q = \{4, 5, 6\} \).

We dequeue 4 and enqueue all unvisited vertices (\( \text{visited}[7] = \text{true} \); \( \text{visited}[8] = \text{true} \)) to \( Q \), which gives \( Q = \{5, 6, 7, 8\} \).

We dequeue 5 and enqueue all unvisited vertices (there are none) to \( Q \), which gives \( Q = \{6, 7, 8\} \).

We dequeue 6 and enqueue all unvisited vertices (there are none) to \( Q \), which gives \( Q = \{7, 8\} \).

We dequeue 7 and enqueue all unvisited vertices (3 is visited) to \( Q \), which gives \( Q = \{8\} \).

We dequeue 8 and enqueue all unvisited vertices (6 is visited) to \( Q \), which gives \( Q = \{\} \).

The order in which we traverse the graph is

\[ 1, 2, 3, 4, 5, 6, 7, 8 \]

This is called a **breadth-first order** traversal.

The following algorithms are based on the depth-first and breadth-first searches:

- Computing the connected component;
- Reporting that a graph has no cycles;
- Computing a path between two given vertices;
- Computing a spanning tree (the BFS tree is typically "short and bushy", the DFS tree is typically "long and stringy")
Topological sorting

A topological sort of a DAG is a listing of its vertices in such an order that if vertex \( W \) reachable from vertex \( V \) (it means there is a path \( V \rightarrow W \)), then \( W \) is listed after \( V \).

Algorithm: count in-degree of each vertex and repeatedly number and remove in-degree 0 vertex along with its out-going edges:

```java
public ArrayList topologicalSort() throws GraphException
{
    ArrayList V = new ArrayList();
    Graph clone = (Graph) clone();
    boolean[] marked = new boolean[numVertices];

    int count = 0;
    while(V.size() < numVertices && count < numVertices) {
        for(int i = 0; i < numVertices; i++) {
            if(!marked[i] && clone.inDegree(i) == 0) {
                // remove outgoing edges
            }
        }
        count++;
    }
    if(V.size() != numVertices) {
        throw new GraphException("Graph has a cycle.
    }
    return V;
}
```

Dijkstra's Algorithm

Consider directed or undirected graph where each edge has a nonnegative weight. One of the nodes is designated as a source \( s \). The length of the path is the sum of the edges' weights. The problem is to find the shortest existing path between \( s \) and any of the other vertices in the graph. By shortest path we mean a set of edges with the minimum possible sum of their weights.

How to tackle this problem? The first idea that comes to mind is to use the BFS. As we know the BFS gives the shortest path between any two vertices of an unweighted graph. We can easily adapt this idea to weighted graphs whose edges' weights are all positive integers. All that is required is to replace each edge of weight \( W \) by \( (W-1) \) vertices of weight 1.
We cannot apply this idea if weights are not integer. In such cases we will use Dijkstra's algorithm. The basic idea behind the algorithm is as follows:

All vertices are divided into two sets:
  D (discovered vertices) and U (undiscovered yet vertices).
At the very beginning, D contains only the source. At each step we choose a vertex from U whose weight to the source is least.

Let $L(x)$ denote the length of a shortest path from $s$ to a vertex $x$. $L(x)$ will always store the length of the best path (so far discovered) from $s$ to $x$. Procedure in pseudo-code

- Set $L(s) = 0$
- Set $L(x) = \infty$ for all vertices $x \neq s$
- Let $T$ be the set of all vertices in graph including $s$
- while $T$ is not empty
  1. choose vertex $v$ in $T$ with minimum length $L(v)$ from $s$ to $v$
  2. **Edge Relaxation**: for each $x$ in $T$ and adjacent to $v$
     
     $L(x) = \min(L(x), L(v) + w(v,x))$
  3. remove $v$ from $T$:
     
     $T = T - \{v\}$

Example:

```
    b  5  d
   o--------o
  /|      /|\
 4 / |    8/ | \ 6
  / |      / | \
 a o    / |   2| o z
  \ |  /    | / \
 2 \ | /     | / 3
   \|/      |/
   o--------o
   c 10    e
```

Procedure:

```
L(a)=0  L(b)=\infty  L(c)=\infty  L(d)=\infty  L(e)=\infty  L(z)=\infty
T={a,b,c,d,e,z}

min is L(a)
L(a)=0  L(b)=4  L(c)=2  L(d)=\infty  L(e)=\infty  L(z)=\infty
Remove a. T={b,c,d,e,z}

min is L(c)
L(a)=0  L(b)=3  L(c)=2  L(d)=10  L(e)=12  L(z)=\infty
Remove c. T={b,d,e,z}

min is L(b)
L(a)=0  L(b)=3  L(c)=2  L(d)=8  L(e)=12  L(z)=\infty
Remove b. T={d,e,z}
```
min is $L(d)$
$L(a)=0$  $L(b)=3$  $L(c)=2$  $L(d)=8$  $L(e)=10$  $L(z)=14$
Remove d. $T=\{e, z\}$

min is $L(e)$
$L(a)=0$  $L(b)=3$  $L(c)=2$  $L(d)=8$  $L(e)=10$  $L(z)=13$
Remove e. $T=\{z\}$

$L(a)=0$  $L(b)=3$  $L(c)=2$  $L(d)=8$  $L(e)=10$  $L(z)=13$
$T=\{}$

Output: $L(z) = 13$

If we explore the whole graph we will find the minimal distances from $s$ to all other vertices. Dijkstra's algorithm is an example of greedy algorithms -- finding a locally optimal solution. The greedy method may not necessarily find the best solution.
We should be careful and not to introduce a negative weight cycles. Because in this case anyone can build an arbitrary low cost path! In the example below, (c, d, b) is a cycle with negative weight.

If we explore the whole graph we will find the minimal distances from $s$ to all other vertices. Dijkstra's algorithm is an example of greedy algorithms -- finding a locally optimal solution. The greedy method may not necessarily find the best solution.
We should be careful and not to introduce a negative weight cycles. Because in this case anyone can build an arbitrary low cost path! In the example below, (c, d, b) is a cycle with negative weight.

The solution which we found before (a,c,b,d,e,z) is not optimal now, (a,c,b,d,c,b,d,e,z) is much lower 11. If we add 8 to all edges and then apply the algorithm we will get a different path:
Exercise 2.

Perform Dijkstra’s shortest path algorithm starting from vertex A.

<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>
<th>H</th>
</tr>
</thead>
<tbody>
<tr>
<td>V</td>
<td>{A, B, C, D, E, F, G, H}</td>
<td>0</td>
<td>inf</td>
<td>inf</td>
<td>inf</td>
<td>inf</td>
<td>inf</td>
<td>inf</td>
</tr>
<tr>
<td>V</td>
<td>{B, C, D, E, F, G, H}</td>
<td>0</td>
<td>5</td>
<td>4</td>
<td>6</td>
<td>0</td>
<td>9</td>
<td>25</td>
</tr>
<tr>
<td>V</td>
<td>{B, D, E, F, G, H}</td>
<td>0</td>
<td>7</td>
<td>6</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>V</td>
<td>{D, E, F, G, H}</td>
<td>0</td>
<td></td>
<td></td>
<td>6</td>
<td>9</td>
<td></td>
<td></td>
</tr>
<tr>
<td>V</td>
<td>{D, F, G, H}</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td>40</td>
<td></td>
<td></td>
</tr>
<tr>
<td>V</td>
<td>{F, G, H}</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>11</td>
<td></td>
</tr>
<tr>
<td>V</td>
<td>{G, H}</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>8</td>
</tr>
<tr>
<td>V</td>
<td>{H}</td>
<td>5</td>
<td>4</td>
<td>7</td>
<td>6</td>
<td>6</td>
<td>8</td>
<td>9</td>
</tr>
</tbody>
</table>

Complexity:

- while T is not empty
  1. find min and remove O(|V|)
  2. for each x in T adjacent to v (this process is called relaxing the edge)
     \[ L(x) = \min( L(x), L(v) + w(v, x) ) \]  \(O(\text{Adj})\)

\[
> |V| + |\text{Adj}| = O(|V|^2 + |E|) = O(|V|^2)
\]

all vertices