# Introduction to graph theory

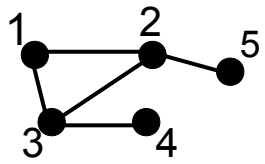
Loh Bo Huai Victor

November 30, 2009

## Graph terminology

A **graph** $G$ consists of a set of **vertices**, usually known as $V$ and a set of **edges**, usually known as $E$ which is a set of pairs of members of $V$. Vertrices are sometimes referred to as nodes and edges are sometimes referred to as arcs.

The diagram below is a graph – the dots represent the vertices of the graph and the lines connecting the dots represent the edges of the graph. A vertex is said to be **incident** to an edge if it is one of the end-points of the edge. Two different vertices are said to be **adjacent** if both vertices are **incident** to the same edge. For example, in the diagram below, vertex 1 is **adjacent** to vertex 5. The **degree** of a vertex is the number of edges that are **incident** to that vertex. For example, vertex 5 has a **degree** of 1.




In the above graph, $V = \{1, 2, 3, 4\}$ and $E = \{(1, 2), (1, 3), (2, 3), (2, 5), (3, 4)\}$.

**Lemma (Handshaking Lemma)**

$$\sum_{v_i \in V(G)} d(v_i) = 2E(G)$$

$|V(G)|$ is called **order** of $G$, while $|E(G)|$ is called **size** of $G$. In describing the runtime complexity of a graph algorithm on a given graph $G = (V, E)$, we usually measure the input

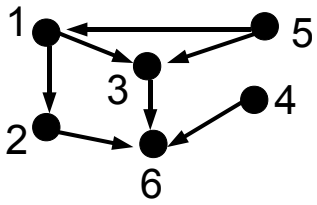
in terms of $|V|$ and $|E|$. Inside asymptoic notation, the symbol $V$ denotes $|V|$ and the symbol $E$ denotes $|E|$. For example, we might say that "BFS has a runtime complexity of $O(V + E)$" and this means that BFS runs in $O(|V| + |E|)$. This convention is adopted as it makes runtime complexity formulas easier to read.

An edge is a **self-loop** if both the end-points of the edge are the same. A graph is **simple** if it does not contain **self-loops** or an edge that is repeated in E. The graph in the diagram above is a **simple** graph as it does not contain **self-loops** or an edge that is repeated in E. A graph is called a **multigraph** if it contains a given edge more than once or contain **self-loops**.

A graph can be **weighted** or **unweighted**. In a **weighted** graph, each edge in the graph has a weight associated with it. In an **unweighted** graph, each edge in the graph does not have a weight associated with it.

A graph is said to be **complete** if there is an edge between every pair of vertices. A **complete** graph has $\frac{|V|(|V|-1)}{2}$ edges. A graph is said to be **sparse** if $|E|$ is much less than $|V|^2$. On the other hand, if $|E|$ is close to $|V|^2$, the graph is said to be **dense**.

A graph can also be **undirected** or **directed**. In an **undirected** graph, one can travel from one vertex to another in any direction as long as the 2 vertices are adjacent. However, for a **directed** graph, one can only travel from one vertex to another in one direction if the 2 vertices are adjacent. Taking the below diagram as an example of a directed graph, one can only travel from vertex 3 to vertex 6, but not from vertex 6 to vertex 3.




The **out-degree** of a vertex is the number of arcs which begin from that vertex. The **in-degree** of a vertex is the number of arcs which end at that vertex. For example, in the above diagram, vertex 5 has **in-degree** 0 and **out-degree** 2.

A **path** from vertex $u$ to vertex $v$ is a sequence of vertices ($v_0$, $v_1$, ..., $v_k$) such that $v_0 = u$ and $v_k = v$ and $(v_0, v_1), (v_1, v_2), ..., (v_{k-1}, v_k) \in E$ (that is to say there is an edge from $v_0$ to $v_1$ and so on). The length of such a path is $k$. Vertex $v$ is **reachable** from vertex $u$ if there exists a **path** from $u$ to $v$.

A path is a **cycle** if it is a path from a vertex to the same vertex. A **cycle** is **simple** if it contains no vertex more than once, except the start (and end) vertex, which only appears as the first and last vertex in the path. A graph is **acyclic** if and only if it contains no **cycles**. These definitions extend similarly to directed graphs.

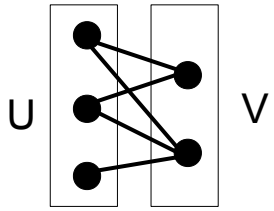
An undirected graph is **connected** if there is a path from every vertex to every other vertex. A directed graph is **strongly connected** if there is a path from every vertex to every other vertex. A **strongly connected component** (or SCC) of a directed graph is a vertex $u$ and the collection of all vertices $v$ such that there is a path from $u$ to $v$ and a path from $v$ to $u$.

# Special graph

A **tree** is an undirected graph that is connected but contain no cycles. A **tree** has $|V| - 1$ edges. An undirected graph which has no cycles, but is not connected, is called a **forest**.

A **DAG** is short-form for directed acyclic graph and this term is commonly used. Every directed graph is a DAG of its strongly connected components (ie every directed graph can be decomposed into a dag where each strongly connected component forms a vertex).

A graph is said to be **bipartite** if the vertices can be split into two disjoint sets $U$ and $V$ ($U$ and $V$ are called bipartition) such there are no edges between any two vertices of $U$ or any two vertices of $V$. Equivalently, a **bipartite** graph is a graph that does not contain any odd-length cycles. The diagram below shows a bipartite graph.




A graph is **planar** if it is possible to draw the graph on a plane without any edge crossing. If a graph is planar, then for $|V| \geq 3, |E| \leq 3|V| - 6$. According to the famous **Four Color Theorem**, all planar graphs can be colored with 4 colors, which means $\chi(G) \leq 4$ for planar graphs.

# Representation of graph

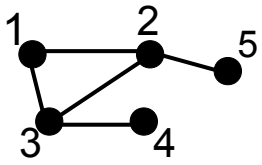
The choice of representation of graph is important, as different graph representations have different runtime complexity and space costs. It is important to choose the right representation when solving a graph problem, as using the wrong graph representation for a particular graph algorithm might change the runtime complexity of the algorithm. There is no such thing as the best kind of representation of graph as different graph algorithms might require

you to use different representation of graph. There are 4 main representation of graph – edge list, adjacency list, adjacency matrix and implicit representation.

## Edge List

The most obvious way to represent the graph is to store a list with a pair of vertex representing an edge. If the graph is weighted, then the weight of the edge has to be stored together with the pair of vertices.

This representation is simple to use as it is very easy to code and also very space efficient. It is also easy to add an edge to the list. However, it is difficult to determine whether 2 vertices are adjacent (as you would have to scan through the whole list) or to delete a specific edge (as you would have to scan through the whole list to find the edge to delete).




| | $u$ | $v$ |
|---|---|---|
| $e_1$ | 1 | 2 |
| $e_2$ | 1 | 3 |
| $e_3$ | 2 | 3 |
| $e_4$ | 2 | 5 |
| $e_5$ | 3 | 4 |

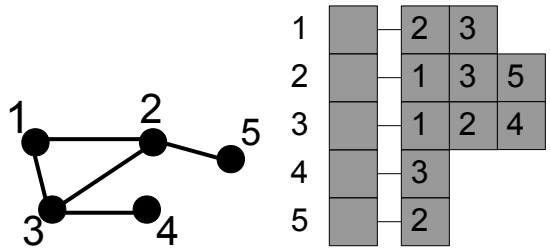
## Adjacency List

The second way to represent a graph is to keep track of all the edges that are incident to each vertex in the graph. This can be done by having an array of size $|V|$, where $|V|$ is the number of vertex in the graph, to store the list of vertices that are adjacent to each vertex in the graph.

Adjacency list can be quite hard to code when there is no bound to the number of edges adjacent to each vertex. In such cases, linked lists have to be used or the lists have to be allocated dynamically. This representation uses as much memory as edge list. Finding all vertices that are adjacent to a particular vertex is easy, but one would need to search through all the edges in the list to determine whether 2 vertices are adjacent. Furthermore, addition of edges is easy, just add to either the back or front of the list. To delete a edge, you need to determine the location of edges in the list first.

It is easy to extend this to weighted graph as all we need to do is to store the weight along with each adjacent vertex, instead of just the adjacent vertex, for each edge in the graph. Directed graph can also be easily maintained by either of these methods – storing

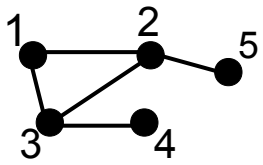
the edges only in one direction, keeping a list of incoming and outgoing arcs or denoting the direction for each edge.




## Adjacency Matrix

The third way of representation of a graph is by using adjacency matrix, which is actually a $|V|$ by $|V|$ matrix ($|V|$ is the number of vertices in the graph). For an unweighted graph, if there exist an edge between vertex i and vertex j, let the entry i,j be 1. Otherwise it is zero or set to infinity (ie a very huge number), depending on the algorithm you are using. For a weighted graph, if there exist an edge between vertex i and vertex j, let the entry i,j be the weight of that edge. The adjacency matrix is symmetrical for an undirected graph, while it is asymmetrical for a directed graph.

It is easy to insert edges, to delete edges and to check whether 2 vertices are adjacent using adjacency matrix. However, the problem with adjacency matrix is that it is not space efficient. This can be a problem if your graph is sparse and the adjacency matrix ends up with a lot of zeros or infinity. Another problem with adjacency matrix is that it does not allow 2 edges to connect to the same 2 vertices – you have to choose which one you would want to store.

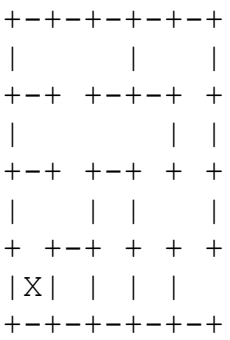



| | $V_1$ | $V_2$ | $V_3$ | $V_4$ | $V_5$ |
|---|---|---|---|---|---|
| $V_1$ | 1 | 1 | 1 | 0 | 0 |
| $V_2$ | 1 | 1 | 1 | 0 | 1 |
| $V_3$ | 1 | 1 | 1 | 1 | 0 |
| $V_4$ | 0 | 0 | 1 | 1 | 0 |
| $V_5$ | 0 | 1 | 0 | 0 | 1 |

## Implict Representation

For some graphs, it is not necessary to store the graph at all. This is because in such cases, it is easy to compute which vertex is adjacent to the current vertex. Examples would be the knights' tour or finding the shortest path out of the maze (Refer to diagram below). Using

implicit representation saves the hassle of trying to store all the edges or using adjacency list or adjacency matrix.



## Summary

| | Edge List | Adjacency List | Adjacency Matrix |
|---|---|---|---|
| Space | $O(E)$ | $O(V+E)$ | $O(V^2)$ |
| Checking adjaceny | $O(E)$ | $O(d_{max})$ | $O(1)$ |
| List of adjacent vertex | $O(E)$ | $O(d_{max})$ | $O(V)$ |
| Adding an edge | $O(1)$ | $O(1)$ | $O(1)$ |
| Deleting an edge | $O(E)$ | $O(d_{max})$ | $O(1)$ |

$d_{max}$ is the max degree of a vertex