Graphs

> Definition
>
> A *directed, simple graph* is a pair (V,E) where
>
> - V is a set of *vertices*
> - $E \subset V\times V$ is a set of *edges*

Terminology:

- We write the relationship E as $\rightarrow$, i.e. $(v,w) \in E$ is denoted as $v \rightarrow w$.
- If $(v,w) = e$, we call $\alpha(e) = v$ the *source* of e
- For this e, we call $\omega(e) = w$ the *target* of e.
- We call v and w *adjacent*.

This definition is that of a *directed*, *simple* graph.

- It is directed in the sense that (v,w) and (w,v) are different edges.
- It is simple in the sense that we can either have that v w are adjacent or not. But we cannot model that there exist multiple connections from v to w.
- We do allow loops $v\rightarrow v$, but we can

> Definition
>
> An *undirected, simple* graph is a pair (V,E), where
>
> - V is a set of vertices.
> - E os a subset of the restricted powerset $P_{\leq 2}(V) \backslash {\empty}$, which is the set of all non-empty subsets of V that have at most two members.

We can also add weights to vertices:

> Definition
>
> A *weighted graph* is a graph (directed or undirected) (V,E) with a function $c : E \rightarrow \R$. This function is often called the *cost function*.

We will later discuss *acyclic graphs* (acyclicity is a property only directed graphs can have, that is why we usually call them *directed acyclic graphs* or DAGs).

Density of graphs

> Definition
>
> A graph is called *dense* if $\lvert E \rvert \approx \lvert V \rvert ^2$

> Definition
>
> A graph is called *sparse* if $\lvert E \rvert << \lvert V \rvert ^2$

Note that

- in a directed graph, we always have $\lvert E \rvert \leq \lvert V \rvert ^2$.
- in an undirected graph, we always have $\lvert E \rvert \leq \binom {\lvert V \rvert}{2} = \frac {\lvert V \rvert (\lvert V \rvert - 1)}{2}$

In both cases, $\lvert E \rvert \in \mathcal O (\lvert V \rvert^2)$

Directed Accyclic Graphs (DAGs)

> Definition
>
> For a directed graph (V,E), a *cycle* is a list of vertices $v_1, v_2, ..., v_n$ s.t. $v_1 = v_n$ and $v_i \rightarrow v_{i+1}$ for all $i = 1, ... , n-1$.

> Definition
>
> A *DAG* is a directed graph for which no cycles exist. (We also say that it "has no cycles")

Representing a graph

As a first question:

- What operations does an interface for a graph need to support?

The following operations for **retrieving** information would be useful:

- Get the set V of vertices.
- Given two vertices $v,w \in V$, query if there is an edge between them.
- Given a vertex, get all the vertices adjacent to it.

We could also support **modifying** operations:

- Add/remove a vertex v from V: this also requires removing edges from E which either have $\alpha(e) = v$ or $\omega(e)= v$.
- Add/remove an edge.

We now list various representations.

Representation 1: Adjacency lists

The key idea is that for each vertex v in V, we store a list of vertices w for which $v \rightarrow w$. In each list element, we can store sattelite information, such as $c(v , w)$.

Note that

- For directed graphs, there is no duplicate information: for every $(v,w) \in E$ we have precisely one element w in the list of v that represents this.
- For undirected graphs, every connection (that is not a loop ${v} \in E$) is represented twice: for ${v,w}\in E$ we have an occurrence w in the adjacency list of v and an occurence of v in the adjacency list of v.
- Note that this requires an invariant, namely that v occurs in the adjacency list of w iff. w occurs in the adjacency list of v.

We can implement an adjacency list using arrays, vectors, single linked or double linked lists. If we can totally order the vertices, we can also use an implementation of a *sorted array* (i.e. heap, BST, red-black tree) for storing the elements, with possibly different access times for searching elements, inserting and deleting elements. The type of datastructure we need depends on what operations on the graph will be most frequently used.

The number of nodes and list elements that need to be stored is $\mathcal O (\lvert V \rvert + \lvert E \rvert)$. We will compare access efficiency later.

Representation 2: Adjacency matrix

The key idea is that for each pair of vertices (v,w) we store whether there is a connection $v\rightarrow w$. We can store this in a $\lvert V \rvert \times \lvert V \rvert$ matrix.

Note that:

- For directed graphs, we have no duplicate entries in the matrix.
- For undirected graphs, the matrix should either always be symmetric (invariant) or we ignore/don't implement the strictly lower triangular portion of the matrix.

In either case, we need $\mathcal O (\lvert V \rvert ^2)$ entries in the adjacency matrix, it being $\frac {\lvert V \rvert (\lvert V \rvert - 1)}{2}$ for undirected graphs or $\lvert V \rvert ^2$ for directed ones. We could use the adjacency matrix directly to store all edge costs, in which case the adjacency matrix holds $c(i,j)$ at entry ij. In that case, a sentinel value is used to represent "no connection".

Comparison of representation

Suppose we implement unsorted lists as resizable arrays, i.e. a `vector` in `c++`. Their access is constant time, search is linear time, insertion is $\mathcal O (n)$ and appending is $\mathcal O(1)$ when the array has spare space and otherwise we need to reallocate an array, leading to $\mathcal O (n)$ complexity.

We could also use a linked list to implement the unsorted list, which would give a $\mathcal O (1)$ time insertion in the head and $\mathcal O (n)$ anywhere else. Search is again $\mathcal O (n)$. Access is $\mathcal O (n)$

However, we can always number vertices `0` to `|V| - 1` and use them as indices into the vector of stored adjacencies. That makes access to a vertex constant time in the case of adjacency lists, and this is certainly better than storing the vertices themselves in a list.

Next, we do not store the matrix as one contiguous array, because that would make insertion very expensive (we would have to reallocate $\lvert V \rvert^2$ elements). Instead, we can number the vertices as indices into a resizable array, i.e. a `vector`, where a row of the matrix is stored as a `vector`. Undirected graphs only require each row i to list all connections for indices $\geq i$, meaning we only store the upper triangular part.

Operation	**Adjacency List representation**	**Adjacency Matrix representation**
Acces vertex $v \in V$	$\mathcal O(1)$	$\mathcal O (1)$

Operation	**Adjacency List representation**	**Adjacency Matrix representation**
query connection of a pair of vertices (v,w)	$\mathcal O (\deg (v))$	$\mathcal O(1)$
List the connections from $v \in V$	$\mathcal O (\deg (v))$	$\mathcal O (\deg (v))$
Storage	$\mathcal O (\lvert E \rvert + \lvert V \rvert)$	$\mathcal O (\lvert V \rvert ^2)$

For the **modifying** operations (assuming the adjacency list is a linked list, since we can insert at the head when adding a connection, and listing the vertices as indices into dynamically resizable array)

Operation	**Adjacency List representation**	**Adjacency Matrix representation**
Add a vertex v to V	$\mathcal O (1)$ or $\mathcal O (\lvert V \rvert)$	$\mathcal O (1)$ or $\mathcal O (\lvert V \rvert)$
Remove a vertex v from V (also requires removing edges from E which either have $\alpha(e) = v$ or $\omega(e)= v$, and renumbering indices)	$\mathcal O (\lvert V \rvert + \lvert E \rvert)$	$\mathcal O (\lvert V \rvert + \lvert E \rvert)$
Add an edge	$\mathcal O (1)$	$\mathcal O (1)$
Remove an edge	$\mathcal O (\deg (v))$	$\mathcal O (1)$