Graph Theory

Definitions:

A graph is a pair $G = (V, E)$, where V is the set of vertices and E is a set of edges. For convenience we denote $|V| = n, |E| = m$, and:

- $V = \{v_1, \dots, v_n\}$
- $E \subset V \times V, E = \{(u, v) | u, v \in V(G)\}$

A **degree** $\boxed{\deg(v)} \leq n - 1$ is the number of edges connected to vertex v. The total: $\sum_{v \in V(G)} \deg(v) = 2m$ – you count each edge twice (once for each end of it).

A **weight function** $\boxed{w: E \to \mathbb{R}^+}$ can be defined over the edges (it doesn't have to be to $\mathbb{R}$, could be a set of labels or $\{0,1\}$ etc.).

A graph can be either **directed** or **undirected**. A directed graph has directed edges, that is: $(u, v) \neq (v, u)$.

A **path** $\boxed{p =< v_1, v_2, \dots, v_k >}$ is a valid path if and only if $(v_1, v_2), (v_2, v_3), \dots, (v_{k-1}, v_k) \in E$. A **simple path** is a path that goes through any vertex in it only **once**.

A **cycle** is a closed path (starts and ends at some vertex). Cycles can be even / odd depending on the number of nodes in it.

A graph $\tilde{G} \subset G$ is a **sub-graph** iff $\tilde{G} = \left(\tilde{V}, \tilde{E}\right), \tilde{V} \subset V, \tilde{E} \subset E$.

The **maximum degree of the graph** is $max - deg = \max_{v \in V} \deg(v)$.

The **minimum degree of the graph** is $min - deg = \min_{v \in V} \deg(v)$.

The **averaged degree of the graph** is $acg - deg = \frac{1}{n} \sum_{v \in V} \deg(v) = \frac{2m}{n}$

According to the definition of deg we now that $\sum_{v \in V} \deg(v) \leq n(n-1) \Rightarrow 2m \leq n(n-1) \Rightarrow m = O(n^2)$

A graph $K_n = (V, E)$ is a **complete graph** if $|V| = n, |E| = \binom{n}{2} = \frac{n!}{2!(n-2)!} = \frac{n(n-1)}{2}$ – a graph with all possible edges (undirected).

A **connected component** of a graph is a subgraph in which there's a path between every two nodes in it.

Connectivity in directed graph is called a **strong connectivity**.

A **shortest path** in a weighted graph between nodes u, v where $P = \{p_i | \, p_i \; is \; a \; path \; between \; u \; and \; v\}$ is $argmin_{p_i \in P} w(p_i)$.

A **tree** is simply a connected graph with no cycles – **acyclic** graph. A tree is a minimal structure, since it establishes connectivity with a minimal number of edges.

Statement**:**

For $T = (V, E), |V| = n$ where T is a tree, we have $|E| = n - 1$

Proof:

For $|V| = 1$ we have no edges (0), for $|V| = 2$ we have 1 edge exactly.

Now assume correctness for $|V| = n - 1$ ($|E| = n - 2$). Now we will add another vertex v, so we have to connect it to one of the nodes in the tree to keep the connectivity. However we cannot add more than one edge, since the tree is connected – thus adding more than one edge will result in a cycle.

A graph $G = (V, E)$ is **bipartite graph** if $V = A \cup B$ such that $\forall (u, v) \in E: u \in A, v \in B$.

Note that an **undirected tree is also a bipartite** graph – that's because there are no cycles.

Theorem:

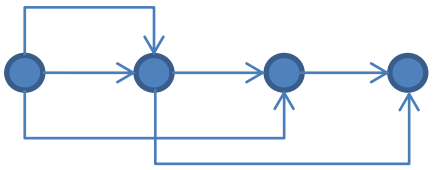
A graph is bipartite iff it **doesn't** have an odd cycle.

Is a bipartite graph dense or sparse? It is **dense**, example: $|A| = |B| = \frac{n}{2}$, connect all vertices in A to those in B, and we get $|E| = \frac{n}{2} \times \frac{n}{2} = \Theta(n^2)$.

A **rooted tree** is a directed tree; a **forest** is a set of trees.

For a forest with c trees with a total $|V| = n$, the number of edges is $|E| - c$ (by induction).

A **directed acyclic graph (DAG)** is a directed graph with no cycles, like a forward connected graph:



A DAG can also be a dense (extremely dense) graph, up to degree of $\frac{n(n-1)}{2}$ since it is $\sum_{i=0}^{n-1} i$ – each node has maximum number of forward edges.

The maximum number of edges in a directed graph can be two times the maximum in an undirected graph, which is $2 \cdot \frac{n(n-1)}{2} = n(n-1)$.

A graph can be represented in a $n \times n$ matrix A where $A_{ij} = \begin{cases} 1, & (v_i, v_j) \in E \\ 0, & otherwise \end{cases}$

In an undirected graph there is symmetry with respect to the diagonal - $\forall i, j: A_{ij} = A_{ji}$.

Looking at A_G the **adjacency matrix** of graph G, note that:

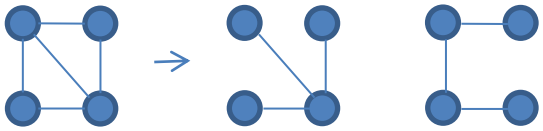
- The total number of 1's is $|E|$.
- The number of 1's in a row is the **out degree** of that row's node.
- The number of 1's in a column is the **in degree** of that column's node.

The adjacency list $Adj[i] = \{j\}$ is the set of nodes to which i has an edge coming out of it.

Another representation is a matrix of size $|V| \times |E| = n \times m$, where each column has two 1's representing the out and in nodes. This is a very sparse matrix, with $2m$ 1's. If we place 1 in the out and -1 in the in, summing up the rows will give the difference between in degree and out degree of the row's node.

Minimum spanning trees

A MST is a spanning subgraph over a graph with weights which is a tree. There could be more several spanning trees in a graph, for instance:



And we want to get the one with total minimal weight.

To build a tree we pick edges, and an algorithm to construct MST will start with an empty graph, and each phase it will add the next best edge – the selection of an edge by locally optimal condition is the **greedy** way.

At each step we are restricted to select only edges that do not create a cycle.

Greedy MST:

First sort the set of edges E by weight such that $w(e_1) \leq w(e_2) \leq \cdots \leq w(e_m)$ – that will take $O(m \lg m) \cong O(m \lg n^2) = O(m \lg n)$.

Each step we check all edges and don't pick the ones which create a cycle. How do we know if a cycle is created for a certain edge? If the end points of the new edge are already part of a connected component, adding that edge will create a cycle. To do that we will use the following data structure:

Disjoint union sets:

$S = (U, \{S_1, \dots, S_m\})$, $U = \{v_1, \dots, v_n\}$ where:

I. $S_i \subset U$

II. $i \neq j \Rightarrow S_i \cap S_j = \emptyset$

III. $\cup_i S_i = U$

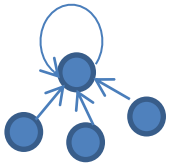
For instance:

$U = \{1,2,3,4,5,6\}, S_1 = \{1,4,6\}, S_2 = \{2,5\}, S_3 = \{3\}$

The data structure will support the following:

- Make-set(v): create a set containing v: $\{v\}$.
- Find-set(v): returns the set to which v belongs.
- Union(u, v): creates a set which is the union of the set containing v and the set containing u.
  $S_1 = \{v_1, \dots, v_l\}, S_2 = \{u_1, \dots, u_m\} \Rightarrow union(S_1, S_2) = \{v_1, \dots, v_l, u_1, \dots, u_m\}$

For a given $S_i = \{u, v, w, t\}$, there will be a **canonical** element of the set, say u, such that $\forall v \in S_i$ the *find-set* procedure will return that canonical element – the root.



The make-set and find-set are $O(1)$, and the union is $O(|S_2|)$, when we change the pointers of all $v \in S_1$ to point $find - set(S_1)$.

Back to MST:

We initially start with a union set $\left(V, \{\{v_1\}, \dots, \{v_n\}\}\right)$, and each edge we check if its two tips are in the same connectivity component.

Worst case, we will have $2m$ find operations, when we have to look at every tip of every edge. The complexity of find operation is constant, so it would be $O(1) \cdot 2m$ for that.

How many union operations: at most $n-1$ – in the worst case you have to add $n-1$ bridges between connectivity components, in order to build the tree. The total cost of all union operations (worst case):

- Initially every vertex is in its own set: $\{v_1\}, \dots, \{v_n\}$
- After $union(v_{n-1}, v_n)$: $\{v_1\}, \{v_2\}, \dots \{v_{n-1}, v_n\}$ – total 1 pointer change (the one of v_n)
- After $union(v_{n-2}, v_{n-1})$: $\{v_1\}, \dots \{v_{n-2}, v_{n-1}, v_n\}$ – total 2 pointers change (the last two).
- The last union will cost $n-1$ pointer changes.

The total is $\frac{n(n-1)}{2}$, and amortized each union will cost us n.

The cost is in general how many pointer redirection we will have to do.

When we encounter union of $S_1, S_2, |S_1| > |S_2|$ we will pointer S_2 to S_2.

The following holds: $|S_1| \geq 2|S_2|$ – this means a total of at most $\lg n$ pointer changes per pointer. For the total n pointers that'd $n \lg n$.

Kriskal's algorithm for finding MST (greedy):

1. Start with $A = \emptyset$
2. For every $v \in V$ do $make - set(v)$
3. Sort edges in E
4. For each $(u, v) \in E$ in increasing order:
5. If $find - set(u) \neq find - set(v)$:
    a. $A \cup \{(u, v)\} \rightarrow A$
    b. $union(u, v)$
6. Return A

Make set takes $O(n)$; sorting takes $O(m \lg n)$; 5-6 takes $O(n \lg n)$.

The running time of this algorithm is $O(m \lg n)$. This is not the best, as the best can be $O(n \lg n + m)$.