Chapter 1

Background

In this chapter, we review basic graph definitions. Then we recall algorithmic aspects, in particular how to store graphs and how to find vertices of small degrees efficiently. We also review the potential function method. The material in this chapter is taken from a variety of sources, among others [CLR90, Gib85, NC88].

1.1 Graph definitions

A *graph* $G = (V, E)$ consists of a set V of *vertices* and and a set E of *edges*. Each edge e is a tuple from $V \times V$, i.e., $e = (v, w)$ with $v, w, \in V$.

In terms of notation, the variables v, w, u, x, and y are usually used for vertices. Edges are usually represented by $e, e', e_1, \ldots$. The number of vertices is $|V| = n$, and $|E| = m$ is the number of edges. If the graph in question is not clear from the context, we write $V(G), E(G), n(G)$ and $m(G)$ instead.

Cycles and paths

A *path* in a graph is a sequence $v_1, e_1, v_2, e_2, \ldots, v_k$ of alternatingly vertices and edges, beginning and ending at vertices, such that the endpoints of any edge in the sequence are exactly the vertex before and after it. Whenever the used edges are clear from the vertices alone, we will omit the edges, and just denote the path as $v_1, v_2, \ldots, v_k$. We say that the path *connects* the vertices v_1 and v_k. The *length* of a path is the number of edges on it, which is $k - 1$ in this example.

A path is called a *simple path* if no vertex appears on it twice. It is easy to show that if there is a path connecting v and w, $v \neq w$, then there is also a simple path connecting v and w.

A *cycle* is a path that begins and ends at the same vertex. Sometimes we write *k-cycle* for a cycle with k edges on it. A 3-cycle is also called a *triangle*.

A *simple cycle* is a cycle on which every vertex appears only once (we consider the begin and endpoint, which are the same, as only one appearance). If we want to emphasize that a cycle may or may not be simple, we will use the term *circuit*, rather than cycle.

Multigraphs

An edge e may be a *loop*, which means $e = (v, v)$, $v \in V$. Multiple edges are also allowed: an edge $e = (v, w)$ is a *multi-edge* if there exists some other edge $e' \in E$ such that $e' = (v, w)$. If there are exactly k edges (v, w), then (v, w) is called a *k-fold edges*. For $k = 1, 2, 3$, (v, w) is called a *simple,*

double and *triple edge*, respectively. A *simple* graph is a graph that contains no loops or multiple edges. If we want to emphasize that a graph G may or may not be simple, we will use the term *multigraph* for it.

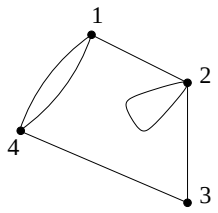

Figure 1.1: A graph, with loops and multiple edges.

There are times when multiple edges and loops unnecessarily complicate matters. A straightforward solution, often employed, is to use the *underlying simple graph* G', which is obtained by deleting every edge from G that makes G non-simple. So remove all loops, and replace each k-fold edge (v, w) with $k > 1$ by exactly one edge (v, w). We will show how to do this efficiently in Section 1.3.1.

Degrees

The degree of a vertex v is the number of times that v is an endpoint of an edge; it is denoted as *deg(v)*. Every loop counts twice towards the degree of its vertex. For example, every vertex in Figure 1.1 has degree 4. We write $\deg_G(v)$ for the degree of vertex v in graph G if the graph is not clear from the context. A vertex is called *even* if it has an even degree, and *odd* otherwise. A graph is called *k-regular* if every vertex has degree k.

Various things can be said about the degrees of vertices; two of the most useful observations are stated here.

Lemma 1.1 *For any graph,* $2m = \sum_{v \in V} \deg(v)$.

Proof: Every edge has two endpoints, and thus counts twice to the degree of some vertex. □

Lemma 1.2 *For any graph, the number of odd vertices is even.*

Proof: Consider that

$$2m = \sum_{v \in V} deg(v) = \sum_{\substack{v \in V \\ v \text{ even}}} deg(v) + \sum_{\substack{v \in V \\ v \text{ odd}}} (deg(v) - 1) + \#\{\text{odd vertices}\}$$

Since the left-hand side is an even number, and all entries in the sums on the right-hand side are even numbers, the number of odd vertices must also be even. □

Special graph classes

The following special graph classes will be needed occasionally.

- A *forest* is a graph without a cycle.
- A *tree* is a graph that is a forest and that has exactly $n - 1$ edges. Many properties of trees are known, we will come back to this in Section 2.1.3.

- A *Eulerian* graph is a graph that has a *Eulerian circuit*, i.e., a cycle (not necessarily simple) that passes through every edge exactly once. It is known that a graph is Eulerian if and only if all vertices have even degree (see for example [PS82]).

- The *complete graph* with n vertices, denoted K_n, is the graph with n vertices and all possible edges that can exist such that the graph remains simple. Thus, any pair of vertices is connected by exactly one edge, which in particular means that K_n has $\binom{n}{2}$ edges.

- A *bipartite graph* is a graph for which the vertices can be partitioned into sets A and B (i.e., $V = A \cup B$) such that any edge conncts a vertex in A with a vertex in B. Since any cycle in a bipartite graph must alternate between vertices in A and B, any cycle in a bipartite graph has even length. The converse also holds: any graph for which all cycles have even length is bipartite.

- The *complete bipartite graph* on $n_1 + n_2$ vertices, denoted K_{n_1,n_2}, is the simple bipartite graph with $|A| = n_1$ and $|B| = n_2$ that has as many edges as possible. Thus, any vertex in A is connected to any vertex in B by exactly one edge, which in particular means that K_{n_1,n_2} has $n_1 \cdot n_2$ edges.

Directed graphs

Sometimes, the edges have a sense of where they are coming from and where they are going to. We then call the graph a *directed graph* or *digraph*. A directed edge is denoted as $e = v \to w$, where w is called the *head* and v the *tail* of e. We denote by $indeg(v)$ the number of edges for which v is the head; this is called the *in-degree*, and these edges are called *incoming at* v. We denote by $outdeg(v)$ the number of edges for which v is the tail; this is called the *out-degree*, and these edges are called *outgoing at* v. We write $indeg_G(v)$ and $outdeg_G(v)$ if the graph is not clear from the context.

Lemma 1.3 *For any digraph,* $m = \sum_{v \in V} indeg(v) = \sum_{v \in V} outdeg(v)$.

Proof: Every edge has exactly one head and exactly one tail, hence counts exactly once to an in-degree, and exactly once to an out-degree. □

1.2 Operations on graphs

Subgraphs

Let $G = (V, E)$ be a graph. A *subgraph* of G is a graph $G' = (V', E')$ such that $V' \subseteq V$ and $E' \subseteq E$. Since G' is a graph, all edges in E' must have both endpoints in V'.

An *induced subgraph* of G is a subgraph where all possible edges are taken, so a subgraph $G' = (V', E')$ where E' are all those edges in G for which both endpoints are in V'. An induced subgraph thus can be specified by just giving its vertex set, and we say that G' is the graph *induced by* V'.

We can also specify a subgraph by giving an edge set. If E' is a set of edges, then the graph *formed by* E' is the graph whose vertices are the endpoints of edges in E', and whose edges are exactly E'.

Deletion, contraction, subdivision

Various operations can be performed on a graph $G = (V, E)$ that result in another graph $G' = (V', E')$. We mentioned a few of them here:

- Deletion of an edge e: This simply means that e is removed from the edge set.

  If G' results from G by deleting the edges in E', then we denote $G' = G - E'$.

- Deletion of a vertex v: This means that first we delete all incident edges of v, and then the vertex v itself.

  If G' results from G by deleting the vertices in V', then we denote $G' = G - V'$.

- Contraction of two vertices v and w: This means that we create one new vertex x. Then, for any edge (u, v) incident to v, we delete this edge, and add a new edge (u, x). For any edge (w, y) incident to w, we delete this edge and add a new edge (x, y). Finally, we delete vertices v and w.

  If G' results from G by contracting the vertices in V', then we denote $G' = G\backslash V'$. Even if G was simple, G' might very well have loops and multiple edges. See Figure 1.2.

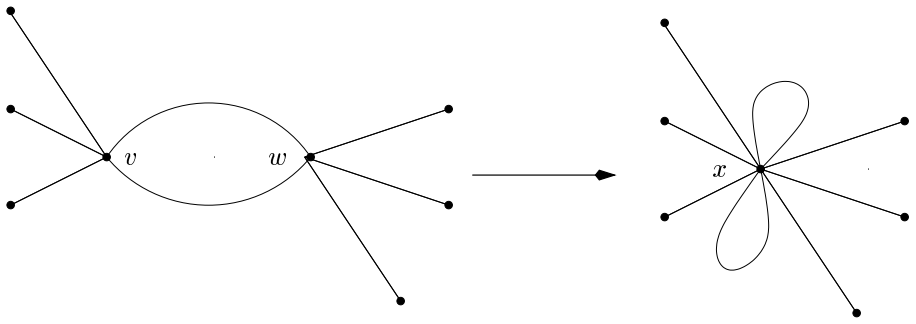

Figure 1.2: The contraction of $\{v, w\}$ creates multiple edges and loops.

- Contraction of an edge $e = (v, w)$: This means that we delete the edge e, and then contract v and w as described above.

  If G' results from G by contracting the edges in E', then we denote $G' = G\backslash E'$.

- Removing a vertex v of degree 2: Let (u, v) and (v, w) be the two incident edges of v. This operation means that we delete (u, v) and (v, w), and add the edge (u, w). Then we delete v.

- Subdividing an edge $e = (u, w)$: This operation is the reverse of the previous operation. We add a new vertex v, add edges (u, v) and (v, w), and delete the edge (u, w).

- A graph G is called a *subdivision* of a graph H, if G can be obtained from H by a number of edge subdivisions.

1.3 Algorithmic aspects of graphs

There is a variety of ways how graphs can be stored. We review here only one, which is probably the most complicated, but in exchange also the most versatile, in that it also supports multiple

edges and loops, makes it easily possible to store extra information with vertices and edges, allows fast graph updates and uses only linear space. Namely, we use incidence lists, i.e., every vertex has a list of incident edges. More precisely:

- Every vertex has a list of incident edges. (Without further mentioning, we assume that any list is a doubly-linked lists with pointers to the first and last element.)
- Every vertex v knows its degree $\deg(v)$.
- Every edge $e = (v, w)$ knows its endpoints $v, w \in V$, and where e is referred to in the incidence lists of v and w.

See Figure 1.3 for an example of this data structure.

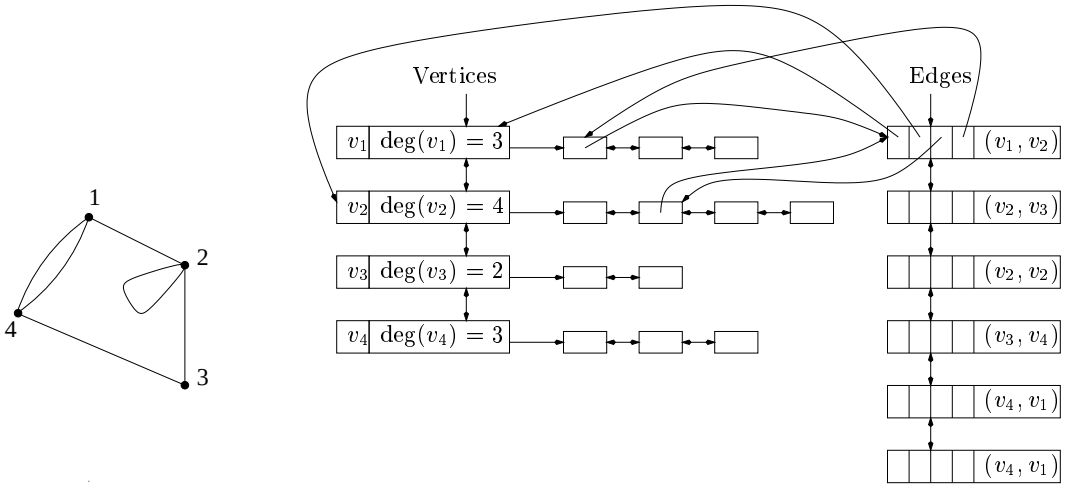

Figure 1.3: A very simple graph, and excerpts of the data structure to store it.

With this data structures, the following holds:

- Insertion or deletion of vertices or edges can be done in constant ($O(1)$) time.
- The adjacency query "is v adjacent to w" can be answered in $O(\min\{deg(v), deg(w)\})$ time, by performing a linear search through the incidence list of whichever of the two vertices has the smaller degree.
- Contracting two vertices v, w takes $O(\min\{deg(v), deg(w)\})$ time, by re-connecting the edges at the vertex with the smaller degree to the other vertex.

Some not-so-obvious claims will be shown in the next subsections.

1.3.1 Computing the underlying simple graph

We claim that the underlying simple graph can be computed in linear ($O(n + m)$) time. This is obvious for removing loops: just scan the list of edges, and remove any edge that has the same endpoint twice. This takes $O(m)$ time.

To remove multiple edges, we have to work a little harder. The crucial idea is to sort the edges in such a way that any two edges that have the form (v, w) are guaranteed to be consecutive. This can be achieved with a double bucket sort (also known as *radix sort*) as follows:

Chapter 2

Graph k-Connectivity

In this chapter, we introduce the concept of connectivity of graphs, and show how to find connected and biconnected components in linear time.

2.1 Connected graphs and components

2.1.1 Definitions

A graph is *connected* if there exists a path from any vertex to any other vertex in the graph. A *connected component* of a graph G is a maximal subgraph G' of G that is connected.

No vertex v can belong to two connected components C_1 and C_2, for otherwise if w_1, w_2 are two vertices in $C_1 \backslash C_2$ and $C_2 \backslash C_1$, then there would be a path from w_1 to v and a path from v to w_2. Combining these two paths we get a path from w_1 to w_2, so w_1 and w_2 belong to the same connected component.

Taking the union of all the connected components of a graph will result in the original graph.

2.1.2 Computing connected components

The connected components can be found easily with any graph traversal method, for example depth-first search or breadth-first search. Here, by a *graph traversal*, we mean any method that starts at an arbitrary vertex r (called the *root* of the traversal), and proceeds to visit all vertices that can be reached along a path from r. Therefore, the graph traversal will find the connected component containing r.

Presuming the graph traversal works in linear time, we can find all connected components in linear time. Namely, we scan all vertices of the graph. If the vertex in consideration has been already visited during a graph traversal, we ignore it and proceed with the next vertex in the list. If the vertex v in consideration has not yet been visited during a graph traversal, then we start a new graph traversal with v as the root. This will visit all vertices in the connected component C that contains v, and takes $O(n(C) + m(C))$ time. We can store this connected component, for example, by keeping track of the number of found components, storing with each vertex v this number when we find v; then two vertices are in the same connected components if and only if they have the same number.

The overall time for this algorithm is $O(n)$ for scanning the vertex list, and $\sum(n(C) + m(C))$ for the graph traversals, where the sum is over all connected components. Since every vertex and edge belongs to exactly one connected component, the total time is $O(n + m)$ as desired.

2.1.3 Properties of connected graphs and trees

If a graph is connected, then it cannot have too few edges.

Lemma 2.1 *Any connected graph G with $n \geq 2$ vertices has at least $n-1$ edges.*

Proof: Let r be an arbitrary vertex of the graph. For each vertex $v \neq r$, there exists a path P_v connecting r and v by connectivity. If there is more than one path, pick an arbitrary one. For the purpose of the following description, it will be convenient to think of P_v as directed from r to v.

We construct a subgraph G_T by scanning the vertices $v \neq r$. We initialize G_T as just vertex r. When we look at a vertex $v \neq r$, then we first look whether v already belongs to G_T; if so, do nothing. If v does not yet belong to G_T, then let x be the last vertex on P_v that does belong to G_T. Add all edges and vertices on P_v between x and v to G_T.

By induction on the number of vertices in G_T, one can verify the following the following properties of G_T:

- For any vertex $v \neq r$ in G_T, there exists exactly one path from r to v in G_T.
- v has exactly one incoming edge in G_T.
- r has no incoming edge in G_T.

Also, eventually all vertices will belong to G_T. Since every edge in G_T belongs (in its undirected version) also to G, we have

$$m(G) \geq m(G_T) = \sum_{v \in V} indeg_{G_T}(v) = n - 1,$$

which proves the claim. □

A lot of interesting results follow from this lemma, and in particular its proof. For example, recall that a tree is a graph without cycles with $n-1$ edges. The re-occurrence of the bound of $n-1$ in Lemma 2.1 suggests that maybe equality holds if and only if G is a tree? This indeed is the case. In fact, the following characterization of tree holds:

Lemma 2.2 *Any tree has the following three properties:*

(a) It has $n-1$ edges.

(b) It has no cycle.

(c) It is connected.

Conversely, any graph that satisfies any two of the above properties is a tree.

For a proof, see for example [PS82].

Now recall the graph G_T that we computed in the proof of Lemma 2.1. We showed that any vertex $v \neq r$ can be reached from r in G_T. Therefore, G_T is connected, because we can get from any v to any w by going from v to r, and then from r to w. We also showed that $m(G_T) = n-1$. This two observations together with the above characterization of trees says that G_T is a tree.

Thus, any connected graph G has a subgraph G_T that contains all vertices and is a tree; such a subgraph is sometimes also called a *spanning tree*.

Now let T be a tree, and let v be an arbitrary vertex. Apply the proof of Lemma 2.1 to T, setting $r = v$. The spanning tree that we obtain has $n - 1$ edges, and therefore must be again T. For any vertex $w \neq v$, we have a unique path from v to w in $G_T = T$. This proves that for any tree T and any two vertices v, w, there exists a unique path between v and w in T.

Trees have another useful property:

Lemma 2.3 *Any tree with $n \geq 2$ vertices has at least two* leaves, *i.e., vertices with degree 1.*

Proof: Let T be a tree with $n \geq 2$ vertices, and let w be a vertex of minimum degree in T. Because T is connected and $n \geq 2$, vertex w must have at least one neighbor, so $\deg(w) \geq 1$. Assume now that all vertex $\neq w$ have degree ≥ 2, then

$$2m = \sum_{v \in V} \deg(v) \geq 2(n-1) + 1.$$

But on the other hand, $2m = 2(n-1)$ because T is a tree, so this implies $0 \geq 1$, which is impossible. Therefore there must be at least one vertex $x \neq w$ with $\deg(x) \leq 1$. As above, one argues $\deg(x) \geq 1$, so $\deg(x) = 1$. Since w was the vertex of minimum degree, this means $\deg(w) = 1$ as well, which proves the claim. □

2.2 Higher connectivity

2.2.1 Definitions

A *cutting k-set* is a set V_k of k vertices such that $G - V_k$ has strictly more connected components than G. When $k = 1$, the vertex in V_k is known as a *cut-vertex* (sometimes also called *articulation point*), and when $k = 2$, the pair of vertices in V_k is known as a *cutting pair* (sometimes also called *separation pair*). A graph is *k-connected* if there is no cutting set with fewer than k vertices. In Figure 2.1 the leftmost graph is 1-connected, since the circled vertex is a cut-vertex. The middle graph is 2-connected (also known as *biconnected*), while the rightmost graph is 3-connected (also known as *triconnected*). There are no fancy names such as, say, penta-connected for graphs that are k-connected, for $k > 3$.

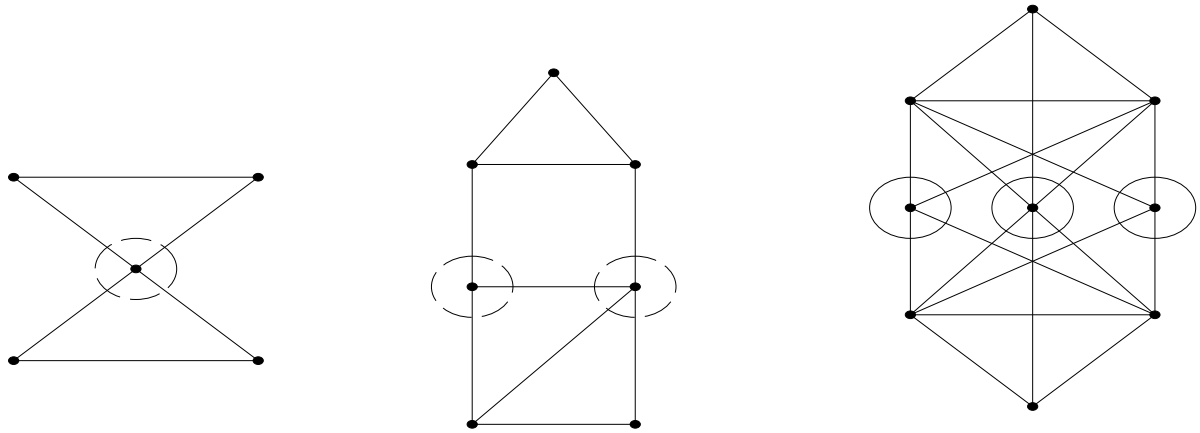

Figure 2.1: 1-connected, 2-connected, and 3-connected