Chapter 1

Introduction

This chapter is a general introduction to graph theory. First we present some basic graph vocabulary and conventions that will be used throughout the notes. The chapter ends with the two most common data structures to represent graphs in programs.

1.1 Basic definitions

1.1.1 Edges, vertices and degrees

Geometrically we represent graphs as a set of points in space (which are called *vertices*) and lines that connect them (the *edges*). We denote the set of vertices by V and the set of edges by E. The graph is then written as $G = (V, E)$. Figure 1.1 shows a graph, $G = (\{u_1, u_2, \ldots, u_6\}, \{e_1, e_2, \ldots, e_{11}\})$.

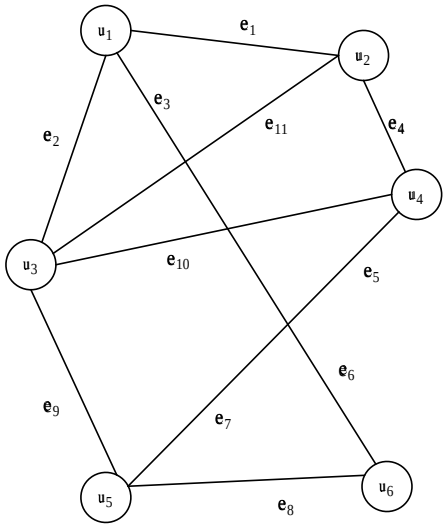


Figure 1.1: A simple graph with 6 vertices and 11 edges

The number of vertices is denoted by $|V|$ and the number of edges by $|E|$ [1]. Each edge is defined by the two vertices that it connects. These vertices are called the *end-points* of

[1] Usually we write $|V| = n$ and $|E| = m$

the edge. For example, in Figure 1.1 $e_1 = (u_1, u_2),\ e_2 = (u_1, u_3),\ \ldots, e_{11} = (u_2, u_3)$. If an edge e has u as an end point, we say that e is *incident* with u. Also if $(u, v) \in E$ we say that u is *adjacent* to v. In Figure 1.1, $e_1,\ e_2$ and e_3 are incident with u_1 which is adjacent to u_2, u_3 and u_6.

Continuing with the definitions we can define the *self-loop* and the *parallel edge*. A self loop is an edge (u, v) for which $u = v$. A parallel edge cannot be uniquely identified by its end points. In Figure 1.2, edge e_1 is a self loop and e_3 is parallel to e_4.

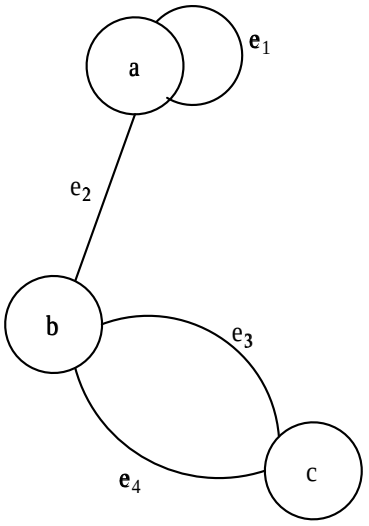


Figure 1.2: An example of a graph containing a self loop and parallel edges

The degree of a vertex u, written $d(u)$, is defined as the number of edges incident with u. In Figure 1.1 $d(u_1) = 3,\ d(u_2) = 3,\ \ldots, d(u_6) = 2$. Theorem 1.1 relates the degrees of the vertices with the number of edges in a graph.

Theorem 1.1 *The sum of the degrees of all the vertices in a graph is twice the number of the edges.*

Proof. Each edge contributes one to the degree of each edge that is incident to. Because each edge is incident to two vertices, if we add up all the degrees of a graph the result will be twice the number of the edges:

$$\sum_{i=1}^{|V|} d(u_i) = 2|E| \tag{1.1}$$

■

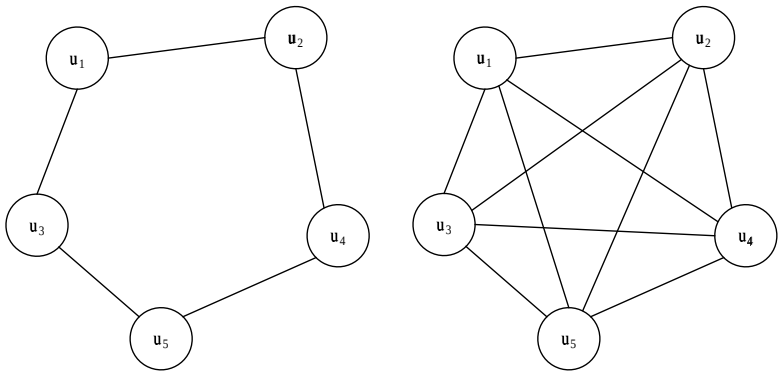
An important corollary of theorem 1.1 is the following.

Corollary 1.1 *The number of vertices of odd degree in a finite graph is even.*

Proof. The right hand side of equation (1.1) is an even number as is the contribution to the left hand side from vertices of even degree. Therefore, the sum of the degrees of odd degree vertices is even which establishes the corollary.

■

A graph is called *b-regular* if $d(u) = b,\ \forall u \in V$. Such a graph has $\frac{n \cdot b}{2}$ edges. A graph is called *complete* if it is $(n-1)$-regular. This graph has $\frac{n \cdot (n-1)}{2}$ edges. An arbitrary graph has at most $\frac{n \cdot (n-1)}{2}$ edges. In Figure 1.3(a), we see a 2-regular graph and in Figure 1.3(b), a complete graph.



(a) A 2-regular graph (b) A complete graph

Figure 1.3: Regular and complete graphs

1.1.2 Directed graphs

In some applications it is natural to assign a *direction* to each edge of the graph. Thus in a diagram of a graph each edge is represented by an arrow (see Figure 1.4). A graph augmented in this way is called a *directed graph* or a *digraph*. If $e = (u_i, u_j)$, is an edge of a digraph then the order of u_i and u_j becomes significant. The edge e is understood to be directed from the first vertex u_i to the second vertex u_j. Thus, if a digraph contains the edge (u_i, u_j) then it may or it may not contain the edge (u_j, u_i). The directed edge (u_i, u_j) is said to be *incident from* u_i and *incident to* u_j. For the vertex u the *out-degree* $d^+(u)$ and the *in-degree* $d^-(u)$ are, respectively, the number of edges incident from u and the number of edges incident to u. A *symmetric* digraph is a digraph in which for every edge (u_i, u_j) there is an edge (u_j, u_i). A digraph is *balanced* if for every vertex $u,\ d^+(u) = d^-(u)$.

1.1.3 Paths and connectivity

A *path* from u_1 to u_k is a sequence $P = u_1,\ u_2,\ u_3,\ \ldots,\ u_{k-1},\ u_k$, for which $(u_i, u_{i+1}) \in E$ for $i = 1,\ \ldots,\ k-1$. If $u_1 = u_k$, then we say that P is a *cycle*. An example of a path and a cycle is shown in Figure 1.5. We say that u_i is *connected* to u_j if there is a path from u_i to u_j. By convention every vertex is connected to itself. *Connection* is an equivalence relation[2] on the vertex set of a graph which partitions it into subsets $V_1,\ V_2,\ \ldots,\ V_k$. A pair of vertices is connected if and only if they belong to the same subset of the partition. These subsets are called *connected components*.

[2]See appendix A for more information on equivalence relations and classes

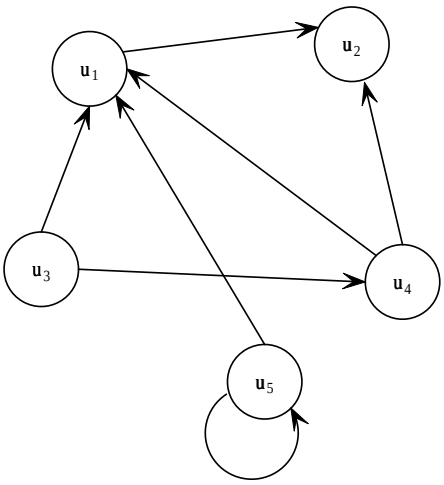


Figure 1.4: An example of a directed graph

A graph is called *connected* if from any vertex there is a path to any other vertex (see Figure 1.6). Another equivalent definition is that a graph is connected if it has only one component. A graph with more components is called *disconnected*.

1.1.4 Trees

A *tree* is a connected acyclic graph (see Figure 1.7(a)). A *forest* is a graph whose components are trees. A tree in which a vertex, the *root*, is distinguished, is called a *rooted tree*. In a rooted tree any vertex of degree 1, unless it is the root, is called a *leaf*. The *depth* of a vertex in a rooted tree is the number of edges in the path from the root to the vertex. If (u, v) is an edge of a rooted tree such that u lies on the path from the root to v, then u is said to be the *father* of v and v is the *son* of u. An *ancestor* of u is any vertex of the path from u to the root of the tree. Similarly, if u is an ancestor of v, then v is a *descendant* of u. A *binary* tree is a rooted tree in which every vertex, unless it is a leaf, has at most two sons (see figure 1.7(b)).

Trees have some very important properties, that are stated by Theorem 1.2.

Theorem 1.2 *If T is a tree with n vertices, then*

1. *Any two vertices of T are connected by precisely one path.*
2. *T has $(n-1)$ edges.*

Proof.

1. T is connected and so there exists at least one path between any two vertices u and v. Suppose that two distinct paths, P_1 and P_2 exist between u and v. Following these paths from u to v, let them first diverge at u' and first converge at v'. That section of P_1 from u' to v' followed by that section of P_2 from v' to u' must form a cycle. By definition, T contains no cycles and so we have a contradiction.
2. Proof is by induction on the number of vertices n in T. If $|V| = 1$ or $|V| = 2$ then, trivially, the number of edges in T is $(n-1)$. We assume that the statement is true

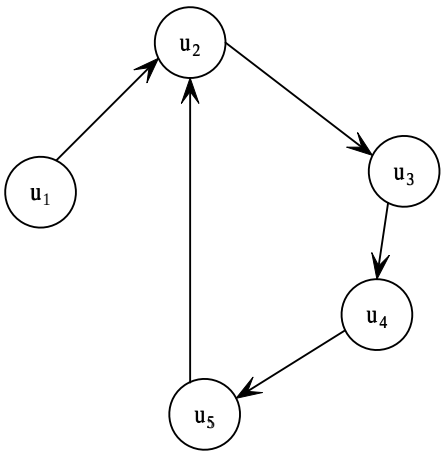


Figure 1.5: An example of a cycle and a path

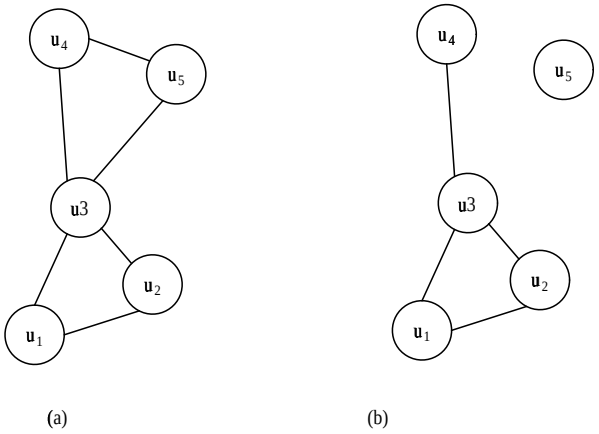


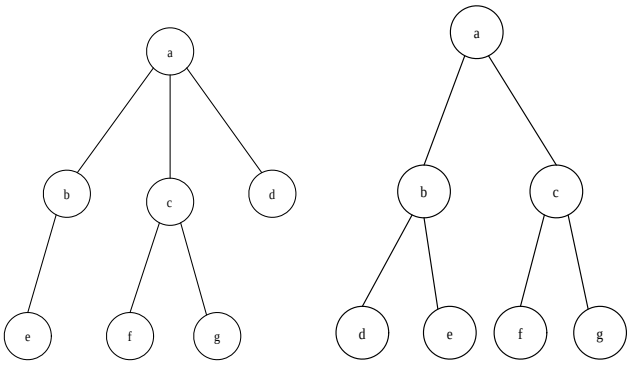
Figure 1.6: An example of a connected and a disconnected graph

for all trees with less than n vertices. Let T have n vertices. There must be a vertex of degree 1 contained in T, otherwise we could trace a cycle by following any path from vertex to vertex entering each vertex by one edge and leaving by another. If we remove a vertex of degree 1, v, from T we neither disconnect T nor create a cycle. Hence $(T - v)$ is a tree with $(|V| - 1)$ vertices. By the induction hypothesis $(T - v)$ has $(|V| - 2)$ edges. Hence replacing v provides T with $(n - 1)$ edges.

■

1.1.5 Weighted graphs

Usually with each vertex or edge of a graph, we keep some satellite data, useful for the applications at hand. For example when modeling a computer network, each vertex might represent a router and each edge the optical fibers that connect the routers. If we are interested in connecting all the routers minimizing the cost of the network, we will



(a) A tree (b) A binary tree

Figure 1.7: Trees

assign a number to each edge that represents the length of the optical fiber required for the connection. Two efficient algorithms for this task will be given in Section 2.4.

Graphs with value assigned edges are called *weighted graphs*. Formally we define a function $w : E \mapsto \mathbb{R}$ defined in the set of edges E and taking values from the real numbers. The value $w(e)$ is called the *weight of the edge* e. The weight of a whole graph is defined to be the sum of the weights of its edges. In Figure 1.8, we see an example of a weighted graph. In this graph the weight of the edge (a, c) is 10, the weight of the edge (f, d) is 5 and so forth.

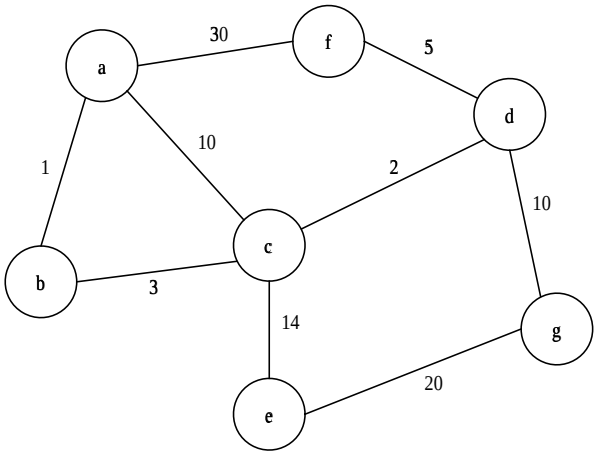


Figure 1.8: An example of a weighted graph

1.1.6 Subgraphs

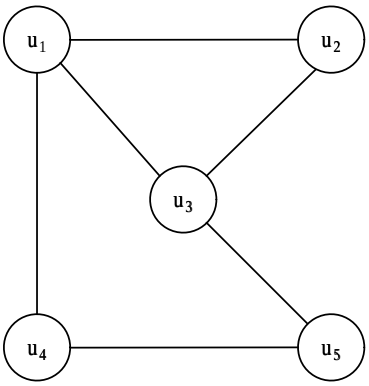
A *subgraph* of G is a graph which is taken from G by the removal of a number of edges and/or vertices of G. If we remove a vertex we must remove all the edges that are incident with it. If H is a subgraph of G then G is called a supergraph of H and we write $H \subseteq G$.

A subgraph of G *induced* by a subset of the vertices, $V' \subset V$, is the graph consisting of V' and those edges of G with both end points in V'.

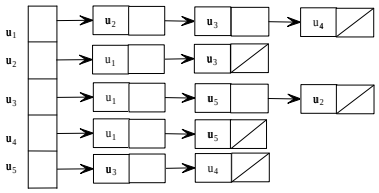
1.2 Graph representations

There are two standard ways to represent a graph in computer programs: as an array of *adjacency lists* or as an *adjacency matrix*.

The adjacency list representation of a graph $G = (V, E)$ is an array Adj with $|V|$ elements each representing one vertex $u \in V$. Every element of this array is a linked list, with each node being a vertex adjacent to u (see figures 1.9(b), 1.10(b)). It is obvious that in an undirected graph if v is a node in the adjacency list of the vertex u, then u is a node in the adjacency list of v. On the other hand, this symmetry does not exist in the representation of directed graphs. Because of that the sum of the lengths of all the adjacency lists in a directed graph is $|E|$, and in an undirected graph is $2|E|$. In both cases, the adjacency list representation requires $O(max(|V|, |E|)) = O(n + m)$ memory.



(a) An undirected graph



(b) The adjacency list representation

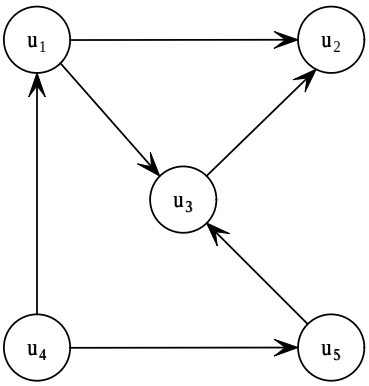
	u_1	u_2	u_3	u_4	u_5
u_1	0	1	1	1	0
u_2	1	0	1	0	0
u_3	1	1	0	0	1
u_4	1	0	0	0	1
u_5	0	0	1	1	0

(c) The adjacency matrix representation

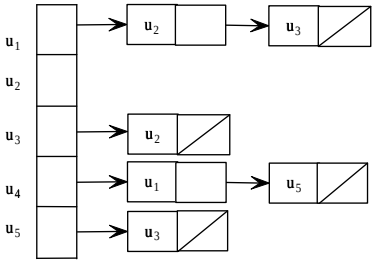
Figure 1.9: The two representations of an undirected graph

The adjacency list can be easily modified to represent weighted graphs by storing the weight $w(u, v)$ of the edge (u, v) with the vertex v in u's adjacency list. The adjacency list representation can also be easily modified in order to support many other graph variants.

The disadvantage of the adjacency list is that there is no quick way to tell if u is adjacent to v. To do that we have to search the list of u for the element v which requires $O(deg(u))$ steps. For the undirected case, we can do this in parallel for the lists of u and v reducing the cost to $O(min(deg(u), deg(v)))$. The other method to represent a graph is the adjacency



(a) A directed graph



(b) The adjacency list representation

	u_1	u_2	u_3	u_4	u_5
u_1	0	1	1	0	0
u_2	0	0	0	0	0
u_3	0	1	0	0	0
u_4	1	0	0	0	1
u_5	0	0	1	0	0

(c) The adjacency matrix representation

Figure 1.10: The two representations of a directed graph

matrix (see figures 1.9(c), 1.10(c)). We assume that the vertices are numbered 1, 2, ..., n in some arbitrary manner. The adjacency matrix representation is a $n \times n$ matrix $A = (a_{i,j})$ such that

$$a_{ij} = \begin{cases} 1 & \text{if } (i,j) \in E, \\ 0 & \text{otherwise.} \end{cases}$$

The adjacency matrix representation requires $O(n^2)$ memory, independently of the number of edges.

As with the case of adjacency lists, we can observe a kind of symmetry, for undirected graphs, in the adjacency matrix representation. The *transpose* of a matrix $A = (a_{ij})$ is defined to be the matrix $A^T = (a_{ij}^T)$ where $a_{ij}^T = a_{ji}$. For an undirected graph the adjacency matrix is symmetric along the main diagonal, i.e. $A^T = A$. This property is important because we can store only the entries on and above the main diagonal, thus the memory requirements are reduced almost by half.

The main advantage of the adjacency matrix representation is that we can determine if $(u, v) \in E$ in $O(1)$ time, by accesing the element a_{uv}. Therefore, this is the prefered way to represent reasonably small graphs or dense graphs for which $|E| \approx |V|^2$.

Like the adjacency list representation, the adjacency matrix representation can be used for weighted graphs. For example, if $G = (V, E)$ is a weighted graph with weight function w, the weight $w(u, v)$ of the edge $(u, v) \in E$ is stored as the entry in row u and column v. If an edge does not exist, a `NIL` value can be stored, although for many applications it is useful to use a value such as 0 or ∞.

Chapter 2

Breadth First Search (BFS)

Breadth First Search is an algorithm for searching graphs. Many important graph algorithms are based on BFS, like Dijkstra's single-source shortest-paths algorithm and Prim's minimum spanning tree algorithm. Given a directed or undirected graph $G = (V, E)$ and a distinguished vertex s, which we name *source vertex*, breadth first search explores all the vertices of G that are reachable from s. This is done in a systematic way that is analyzed in the next sections.

2.1 BFS in general

Breadth First Search computes the distance from s to all of its reachable vertices. This algorithm works on both directed and undirected graphs. The BFS algorithm is given below in pseudocode.

BREADTH FIRST SEARCH

Input: A graph $G = (V, E)$ and a source vertex $s \in V$.

Output: A breadth first tree that includes all the vertices from G that are reachable from s, the distance of each vertex from s and its parent.

```
BFS(G, s)
 1  for  each vertex v in V − {s}
 2       do color[v] ← white
 3          d[v] ← ∞
 4          π[v] ← NIL
 5  d[s] ← 0
 6  π[s] ← NIL
 7  color[s] ← gray
 8  Q ← s
 9  while Q ≠ ∅
10       do u ← DEQUEUE(Q)
11          for  each v in Adj[u]
12              do if color[v] = white
13                    then color[v] ← gray
```

14 $d[v] \leftarrow d[u] + 1$
15 $\pi[v] \leftarrow u$
16 ENQUEUE(Q, v)
17 $color[u] \leftarrow black$

2.1.1 Description of the algorithm

The algorithm takes as input a graph $G = (V, E)$ and a vertex $s \in V$, that is called the *source vertex*. After the execution of the algorithm for every vertex that is reachable from s we have its parent in the BFS tree (see Section 2.2)and its distance from the source. The distance between two vertices is computed in number of edges.

The algorithm uses a FIFO queue Q in order to keep track of the vertices it must visit. Also, each vertex has one of three colors: white that means that we have not yet visited the vertex, gray that means that we have visited the vertex itself but not all the vertices adjacent to it and black that means that we have visited the vertex and all its neighbors. The color, the parent and the distance from the source can be maintained as attributes in the adjacency list representation as for the case of a weighted graph(see Section 1.2).

The first four lines of the algorithm initialize the attributes for all the vertices except the source. Initially the color is white because we have not yet visited the vertex, the distance is set to ∞ and the parent is set to `NIL` because we do not know it. The next four lines are the initialization for the source vertex. The distance from itself is zero so we set $d[s] \leftarrow 0$. As will be described in the Section 2.2 the source vertex is the root of the BFS tree and therefore it has no father. Thus, $\pi[u] \leftarrow$ `NIL`. Finally, the source is the first vertex we visit and that is why it is painted gray in line 7 and it is inserted into the (initially empty) queue.

The loop in lines 9–17 is the core of the algorithm. The algorithm will enter the loop after the initialization because we have inserted s into Q and so $Q \neq \emptyset$. In line 10, the first element of the queue (the head) is removed and assigned to u and the **for** loop in line 11 examines all the vertices adjacent to it. If there is a vertex v adjacent to u such that its color is white (i.e. we have not visited it yet), it is painted gray. In line 13, its distance from s is set to be the distance of u plus one edge in line 14 and u becomes its parent in line 15. Finally, v is inserted at the end of the queue in line 16.

When all the vertices adjacent to u have been examined, i.e. the algorithm exits the **for** loop that begins in line 11, u is painted black. The algorithm ends when the queue, Q, is empty, that is when there are no more gray vertices to be examined.

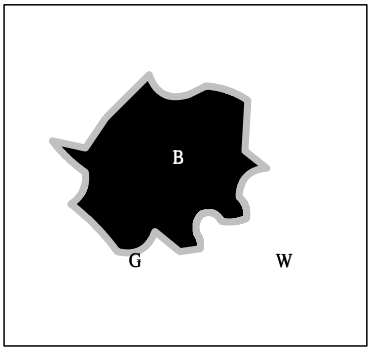


Figure 2.1: Gray vertices are the "frontier" between black and white vertices

Make the following notes on BFS:

- When a vertex is painted black, the algorithm will never encounter it again.
- The gray vertices as mentioned before are vertices that have been visited themselves but their adjacency lists are not yet fully examined. We can think of them as the frontier between discovered (i.e. black and gray) vertices and undiscovered (i.e. white) vertices. This fact is illustrated in Figure 2.1.
- If a vertex v is not reachable from s (i.e. there is not a path from s to v), it will not be discovered by the algorithm because it only examines vertices adjacent to already discovered vertices. For such a vertex the attributes will not be changed and therefore after the termination of the algorithm $d[v] = \infty$, $\pi[v] = \texttt{NIL}$ and $color[v] = white$.

The effect of BFS for a simple graph is shown in Figure 2.2.

In Figure 2.2(a), the source b has been visited (it is painted gray) and it is inserted into the queue. In Figure 2.2(b), the vertices adjacent to b i.e. e, f and c are painted gray, and are inserted into the queue. Because they are adjacent to s their distance from s is 1. Vertex s is painted black because all the vertices adjacent to have been painted gray, and it is removed from the queue. Next, in Figure 2.2(c), vertex a, that is adjacent to e is visited, and inserted into the queue. Its distance from b is $d[w] + 1 = 2$. The algorithm continues in the same manner until all the vertices are black and the queue is empty as shown in Figure 2.2(h).

2.1.2 BFS correctness

In this section we will prove that the distance $d[u]$, of a vertex $u \in V$ from the source s after the application of BFS on a graph $G = (V, E)$, is either the minimum distance $\delta(s, u)$ or ∞ if there is not a path from s to u. We begin with the definition of $\delta(s, u)$ for *unweighted* graphs.

The *sortest path distance* $\delta(s, u)$ from s to u is defined as the minimum number of edges in *any* path from the vertex s to the vertex u. If there is not a path from s to u then $\delta(s, u) = \infty$. A path of length $\delta(s, u)$ is called *shortest path* from s to u. Now, let's examine some useful properties of shortest-path distances, which will help us further down.

Lemma 2.1 *Let $G = (V, E)$ be a graph (directed or undirected) and $s \in V$ be an arbitrary vertex. Then for any edge $(u, v) \in E$:*

$$\delta(s, v) \leq \delta(s, u) + 1$$

Proof. If u is reachable from s, then v is also reachable with 1 more edge ((u, v)) at most. If u is not reachable then of course $\delta(s, v) \leq \infty$. Therefore, either way the inequality holds.

■

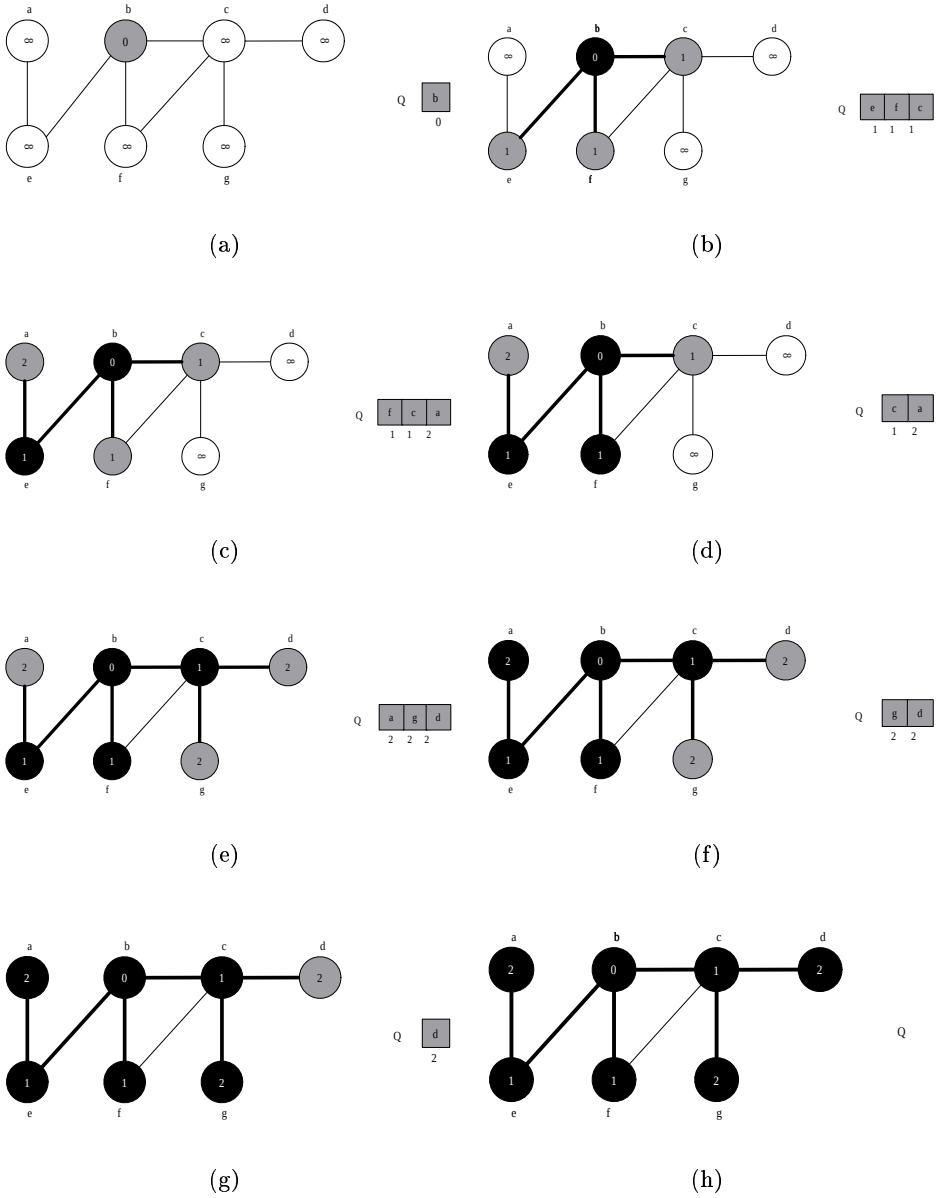


Figure 2.2: The effect of BFS algorithm on a simple graph

Lemma 2.2 *Let $G = (V, E)$ be a graph (directed or undirected) and BFS is applied on G with source $s \in V$ being an arbitrary vertex. When finished for each vertex $v \in V$ the value $d[v]$ satisfies $d[v] \geq \delta(s, v)$.*

Proof. We use induction for each time a vertex is placed in the queue Q. The hypothesis is that $d[v] \geq \delta(s, v)$. The hypothesis holds at the beginning, when s is placed in Q: $d[s] = 0 = \delta(s, s)$ and $d[v] = \infty \geq \delta(s, v),\ \forall v \in V - s$.

The inductive step is the situation when a white vertex v is discovered from its parent vertex u. According to the hypothesis $d[u] \geq \delta(s, u)$. From the assignment of BFS in line 14 and Lemma 2.1 we obtain:

$$d[v] = d[u] + 1 \geq \delta(s, u) + 1 \geq \delta(s, v)$$

After that, $d[v]$ never changes because v is colored gray, so the if-statement that changes $d[v]$ is never again true. Thus, the inductive hypothesis is maintained.

■

Lemma 2.3 *During the execution of BFS on a graph $G = (V, E)$ the queue Q contains the vertices $\langle v_1,\ v_2,\ \ldots,\ v_r \rangle$ where v_1 is the head of Q and v_r its tail. Then*

$$\begin{aligned} &d[v_r] \leq d[v_1] + 1 \text{ and} \\ &d[v_i] \leq d[v_{i+1}],\ \text{for } i = 1,\ 2,\ \ldots,\ r - 1 \end{aligned}$$

Proof. We use induction on the times that a vertex is enqueued in Q. At the beginning the hypothesis holds. For the inductive step, the lemma must hold after a vertex is dequeued and after it is enqueued. When dequeuing the head vertex v_1, v_2 becomes the head. We have $d[v_r] \leq d[v_1] + 1 \leq d[v_2] + 1$ (the rest inequalities don't change) and the inequality (and the lemma) holds for the new head v_2 during the dequeuing of v_1.

When enqueuing a vertex v it becomes v_{r+1}. At that time, we have already removed vertex u, whose adjacency list is currently being scanned, from the queue Q, and by the inductive hypothesis, the new head v_1 has $d[v_1] \geq d[u]$. Thus, $d[v_{r+1}] = d[v] = d[u] + 1 \leq d[v_1] + 1$. From the inductive hypothesis, we also have that $d[v_r] \leq d[u] + 1$ and so $d[v_r] \leq d[u] + 1 = d[v] = d[v_{r+1}]$, and the remaining inequalities are unaffected. Thus, the lemma follows when v is enqueued.

■

We can now use Lemmata 2.1, 2.2 and 2.3 in order to prove that BFS correctly finds shortest path distances.

Theorem 2.1 (BFS correctness) *Let $G = (V, E)$ be a graph (directed or undirected) and suppose that BFS is run on G from a given source vertex $s \in V$. Then, during its execution, BFS discovers every vertex $v \in V$ that is reachable from the source s, and upon termination, $d[v] = \delta(s, v), \forall v \in V$. Moreover, for any vertex $v \neq s$ that is reachable from s, one of the shortest paths from s to v is the shortest path from s to $\pi[v]$ followed by the edge $(\pi[v], v)$.*

Proof. First we examine the case in which vertex v is unreachable from s. From lemma 2.2 we have $d[v] \geq \delta(s, v) = \infty$. Line 14 is never reached for vertex v: If it was reached, there should exist an execution of this line (the current execution or a previous, of an ancestor of

v) in which d was set to ∞, which, by induction, cannot stand. Line 14 is therefore executed only for vertices with finite d values. Thus, if v is unreachable, it is never discovered.

Now we examine vertices reachable from s. Let V_k represent the set of vertices at distance k from s ($V_k = \{v \in V : \delta(s, v) = k\}$). We shall use induction on k. Our inductive hypothesis will be the assumption that for each vertex $v \in V_k$, there is exactly one point during the execution of BFS at which:

- v is grayed,
- $d[v]$ is set to k,
- if $v \neq s$ then $\pi[v]$ is set to u for some $u \in V_{k-1}$ and,
- v is inserted into the queue Q.

As we have noted before there is certainly at most one such point.

The basis is for $k = 0$. We have $V_0 = \{s\}$, since the source s is the only vertex at distance 0 from s. During the initialization, s is grayed, $d[s]$ is set to 0, and s is placed into Q, so the inductive hypothesis holds.

For the inductive step, we notice that the queue Q is never empty until the algorithm terminates and that, once a vertex u is inserted into the queue, neither $d[u]$ nor $\pi[u]$ ever changes. By lemma 2.3, therefore, if vertices are inserted into the queue over the course of the algorithm in the order $v_1,\ v_2,\ \ldots,\ v_r$, then the sequence of distances is monotonically increasing: $d[v_i] \leq d[v_i + 1]$ for $i = 1,\ 2,\ \ldots,\ r - 1$.

Now let us consider an arbitrary vertex $v \in V_k$, where $k \geq 1$. The monotonicity property, combined with $d[v] \geq k$ (by lemma 2.2) and the inductive hypothesis, implies that v must be discovered after all vertices in V_{k-1} are enqueued, if it is discovered at all.

Since $\delta(s, v) = k$, there is a path of k edges from s to v, and thus there exists a vertex $u \in V_{k-1}$ such that $(u, v) \in E$. Without loss of generality, let u be the first such vertex grayed, which must happen since, by induction, all vertices in V_{k-1} are grayed. The code for BFS enqueues every grayed vertex, and hence u must ultimately be set to the value of the head of the queue in line 10. When u acquires the value of the head, its adjacency list is scanned and v is discovered. (The vertex v could not have been discovered earlier, since it is not adjacent to any vertex in V_j for $j < k - 1$ -otherwise, v could not belong to V_k - and by assumption, u is the first vertex discovered in V_{k-1} to which v is adjacent.) Line 13 grays v, line 14 establishes $d[v] = d[u] + 1 = k$, line 15 sets $\pi[v]$ to u, and line 16 inserts v into the queue. Since v is an arbitrary vertex in V_k, the inductive hypothesis is proved. Finally we observe that if $v \in V_k$, then by what we have just seen, $\pi[v] \in V_{k-1}$. Thus, we can obtain a shortest path from s to v by taking a shortest path from s to $\pi[v]$ and then traversing the edge $(\pi[v], v)$.

■

2.1.3 Analysis of the BFS algorithm

In this section we analyze the running time of the BFS algorithm. The initialization in lines 1–4 takes $O(n)$ time since the loop is executed once for each vertex of the graph. The **for** loop in lines 11–16 will be executed one time for each edge of the graph, because each vertex is painted white once during the initialization and therefore takes $O(m)$ time. Both ENQUEUE and DEQUEUE operations run in constant time $O(1)$. Finally lines 10 and 17 will

be executed one time for each vertex because every vertex is enqueued and dequeued only one time. Thus they take $O(n)$ time. Therefore the total running time of the algorithm will be: $O(n) + O(n) + O(m) = O(m + n)$.

2.1.4 BFS and disconnected graphs

As mentioned in the beginning of this chapter, given a graph G and a vertex s BFS finds all the vertices that are reachable from the source s. If the graph is disconnected, we can identify all the connected components, by running successively the BFS algorithm on the (proper) subgraph $G' \subset G$ that is induced by the set $V' \subset V$ of the vertices that are still white.

The following algorithm takes as input a graph $G = (V, E)$ and a source vertex s and assigns a number to each vertex that corresponds to the connected component that it belongs to.

BFS-Connected Components

Input: A graph $G = (V, E)$ and a source vertex $s \in V$.

Output: A number assigned to each vertex $u \in V$ that represents the connected component that it belongs to.

```
BFS-Connected-Components(G, s)
 1  c ← 0
 2  G' ← G
 3  repeat
 4          s ← v ∈ V'
 5          BFS(G', s)
 6          c ← c + 1
 7          for  each u in V'
 8              do if color[u] = black
 9                    then component[u] ← c
10          G' ← Remove-Black-Vertices(G')
11     until V' = ∅
```

The algorithm works as follows: Lines 1–2 are the initialization. One counter, that counts the connected components is set to zero and we make a copy of the graph. Note that this copy should contain pointers to the original vertices of the graph because we want to assign a number in each of them.

The core of the algorithm, in lines 3–11, is a repeat-loop that applies the BFS to the remaining graph and then assigns the component number to the vertices colored black by it and removes them. The black vertices must be removed because the initialization of BFS colors every vertex white, and this can cause an infinite loop.

The procedure Remove-Black-Vertices takes as input a graph colored by BFS and returns another graph that contains all the vertices of the original graph except those colored black. Formally, if $G = (V, E)$ is the input graph and $G' = (V', E')$ is the output graph then $u \in V - \{V' \cap V\}$ if and only if $color[u] = black$. The pseudo-code is given below.

Remove-Black-Vertices

Input: A graph $G = (V, E)$ after it has been colored by BFS.

Output: The original graph without the vertices colored black.

```
REMOVE-BLACK-VERTICES(G)
1  for  each v in V
2      do if color[v] = black
3            then V ← V − {v}
4  return G
```

The BFS-CONNECTED-COMPONENTS runs in $O(c(m+n))$ time, where c is the number of the connected components of the graph. The repeat-loop, in lines 3–11, will be executed exactly c times because each time BFS finds a connected component, it will be removed by REMOVE-BLACK-VERTICES. BFS runs in $O(m+n)$ time.

Finally, REMOVE-BLACK-VERTICES needs $O(n)$ time to be executed. If we assume an adjacency list representation of G', then we only need to examine if the vertex in the position i in the adjacency list array is black. If it is, this means that this vertex was reachable from the source s when BFS was executed in line 5. Therefore, all the vertices adjacent to it were reachable and by the theorem 2.1 BFS visited them and painted them black. Thus, BFS-CONNECTED-COMPONENTS runs in $O(c((m+n)+n)) = O(c(m+n))$.

2.2 Breadth First Trees

One of the ways to depict the results of the application of BFS on a graph, is the Breadth First Tree, that the algorithm produces. Note that the tree is not directly accessible as a result of BFS; after the execution of the algorithm for each vertex we have its father and not its sons. Note that the root of the tree is the source s that is given as an input to BFS.

Breadth first trees have the nice property that every edge of the graph can be classified into one of three groups. Some edges are in the tree themselves, some edges connect two vertices at the same level of the tree and finally some edges connect two vertices on two adjacent levels of the tree. For example, in figure 2.3(a) edges $(1,2)$, $(1,3)$, $(1,4)$, $(2,5)$, $(2,6)$ and $(5,7)$ are edges of the first category, $(2,3)$ is of the second category and $(5,3)$, $(5,4)$, $(6,4)$ and $(7,6)$ are of the third category. The edges that are not in the tree (i.e. those of the two last categories) are called *crossedges*. In the Figure 2.3(b), the crossedges are shown with dashed lines.

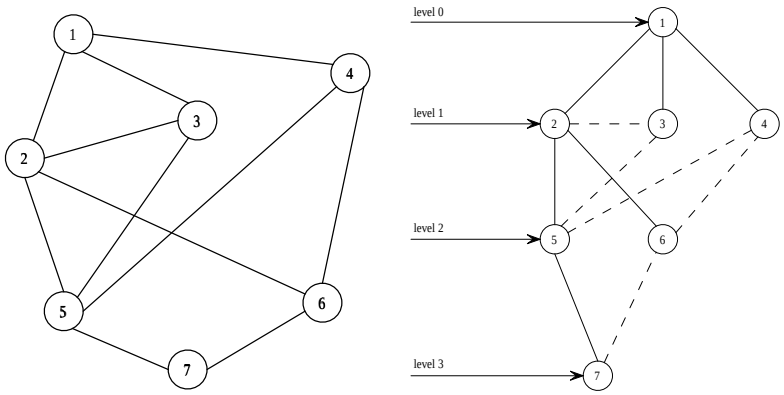
Formally, for a graph $G = (V, E)$ with source s, we define the *predecessor subgraph* of G as $G_\pi = (V_\pi, E_\pi)$, where

$$V_\pi = \{u \in V : \pi[u] \neq \texttt{NIL}\} \cup \{s\}$$

and

$$E_\pi = \{(\pi[u], u) \in E : u \in V_\pi - \{s\}\}$$

In simple words the predecessor subgraph is the connected component of the graph that includes the source s and does not include the crossedges. The predecessor graph G_π is a breadth first tree if V_π consists of the vertices reachable from s and, for all $u \in V_\pi$ there is a unique simple path from s to u in G_π that is also a shortest path from s to u in G. Lemma 2.4 shows that the predecessor subgraph that results from applying BFS on a graph is in fact a breadth first tree.



(a) A graph

(b) The Breadth First Tree that results from the graph in 2.3(a)

Figure 2.3: A graph and the resulting Breadth First Tree

Lemma 2.4 *When applied to a graph $G = (V, E)$ (directed or undirected), procedure BFS constructs π so that the predecessor subgraph $G_\pi = (V_\pi, E_\pi)$ is a breadth first tree.*

Proof. Line 15 of BFS sets $\pi[v] = u$ only if $(u, v) \in E$ and $\delta(s, u) < \infty$ - i.e. if v is reachable from s. Therefore V_π consists of the vertices in V reachable from u. Since G_π forms a tree, it contains a unique path from s to each vertex in V_π. By applying Theorem 2.1 inductively, we conclude that every such path is a shortest path.

■

There is an easy way to print the path from s to u, assuming that BFS has already been executed to compute the breadth first tree.

PRINT-PATH

Input: A graph $G = (V, E)$ after BFS has been executed on it, a source vertex s and a destination vertex u.

Output: Prints the path from s to u.

```
PRINT-PATH(G, s, u)
1  if u = s
2     then PRINT(s)
3     else  if π[u] = NIL
4              then PRINT( "no path from " s " to " u " exists." )
5              else  PRINT-PATH(G, s, π[u])
6                    PRINT(u)
```

This procedure runs in linear time in the number of vertices in the path printed, since each recursive call is for a path shorter by one vertex.

2.3 Single Source Shortest Paths (Dijkstra's Algorithm)

Assume we have a weighted directed graph $G = (V, E)$ such that every edge $(u, v) \in E$ has a nonnegative weight ($w(u, v) \geq 0$). There are some applications where we want to find the shortest path from a source s to every other vertex u of G. Dijkstra's algorithm solves this problem.

DIJKSTRA

Input: A graph $G = (V, E)$, a weight function $w : E \mapsto \mathbb{R}$ and a source vertex s.

Output: For each vertex u other than s the shortest path from s to u.

```
DIJKSTRA(G, w, s)
 1  for each vertex v in V − {s}
 2      do d[v] ← ∞
 3         π[v] ← NIL
 4  d[s] ← 0
 5  S ← s
 6  Q ← V − S
 7  while Q ≠ ∅
 8      do u ← EXTRACT-MIN(Q)
 9         S ← S ∪ {u}
10         for each v in Adj[u]
11             do if d[v] > d[u] + w(u, v)
12                   then d[v] ← d[u] + w(u, v)
13                        π[v] ← u
```

2.3.1 Description of the algorithm

The input of Dijkstra's algorithm, is a graph G, a source vertex s and a function w that assigns a weight to every edge $(u, v) \in E$. The output of the algorithm is the shortest path from s to every vertex $u \in V$, if there is a path from s to u or ∞ otherwise. For all vertices $v \in S$ we have $d[v] = \delta(v, s)$ where

$$\delta(u, v) = \begin{cases} \min\{w(p) : u \overset{p}{\rightsquigarrow} v\} & \text{if there is a path from } u \text{ to } v \\ \infty & \text{otherwise} \end{cases}$$

We can see that lines 1-4 of the algorithm are for the initialization of $d[\cdot]$ and $\pi[\cdot]$. $d[u]$ corresponds to the distance of u from s, while $\pi[u]$ corresponds to the vertex v that is the "parent" of u, meaning that v is the previous vertex from u in the shortest path from s to u.

Lines 5-6 perform the initialization of the two sets S and Q that we are going to use at the next steps of the algorithm. Lines 7-13 are the basic segment of the algorithm. Each time the loop is executed, a vertex u is removed from Q and it is added to the set S. Following the relaxation takes place in order to adjust the $d[\cdot]$ and $\pi[\cdot]$ attributes of the vertices that are adjacent to u. Since every vertex is being removed from Q exactly once, this while loop is executed exactly $|v|$ times.

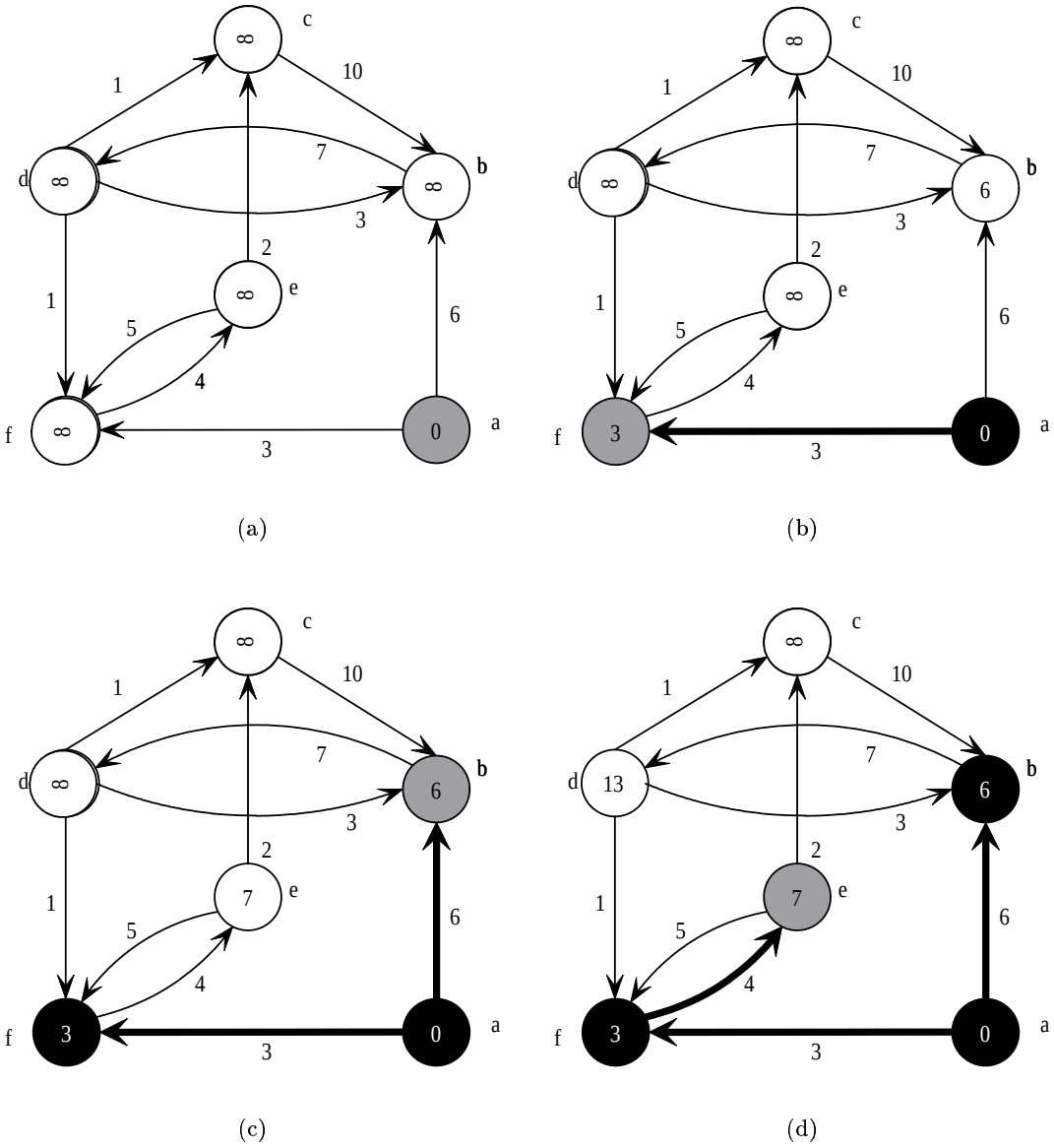
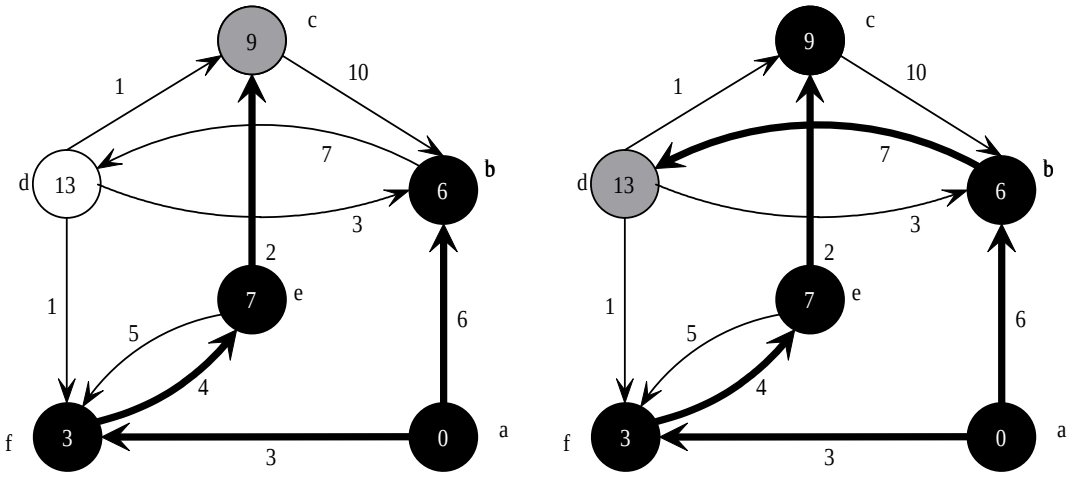
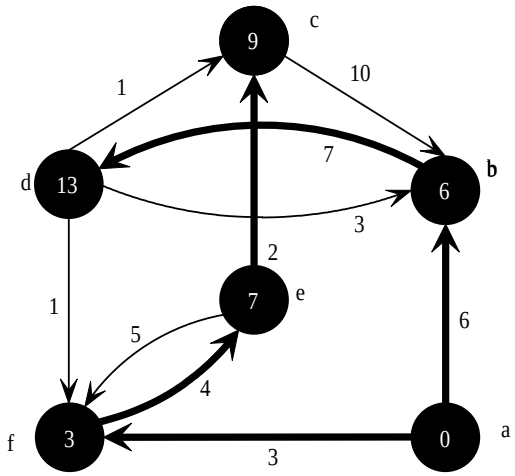


Figure 2.4: The operation of Dijkstra's algorithm (part 1)



(e)

(f)



(g)

Figure 2.4: The operation of Dijkstra's algorithm (part 2)

In Figure 2.4, we can see an example of the operation of Dijkstra's algorithm. In Figure 2.4(a), vertex a is under examination. We compute the value d of every vertex adjacent to a, we paint the vertex a black and the algorithm continues examining the next minimum vertex in Q, which in our example is f (see Figure 2.4(b)). In this way we continue until all the vertices of the graph are painted black and until all the edges of the desired minimum spanning tree are painted black too (see Figure 2.4(d)).

Dijkstra's algorithm is a greedy algorithm, because it always chooses the "closest" vertex in $V - S$ to be inserted into the set S. Greedy algorithms do not always yield optimal results in general, but as it will be shown by the next theorem, Dijkstra's algorithm does indeed compute shortest paths.

Theorem 2.2 *(Correctness of Dijkstra's algorithm) If we run Dijkstra's algorithm on a weighted, directed graph $G = (V, E)$ with nonnegative weight function w and source s, then at termination, $d[u] = \delta(s, u)$ for all vertices $u \in V$.*

Proof. We shall show that for each vertex $u \in V$, we have $d[u] = \delta(s, u)$ at the time when u is inserted into set S and that this equality is maintained thereafter.

For the purpose of contradiction, let u be the first vertex for which $d[u] \neq \delta(s, u)$ when it is inserted into set S. We shall check the situation at the beginning of the iteration of the while loop in which u is inserted into S and derive the contradiction that $d[u] = \delta(s, u)$ at that time by examining a shortest path from s to u. We must have $u \neq s$ because s is the first vertex inserted into set S and $d[s] = \delta(s, s) = 0$. Because $u \neq s$, we also have that $S \neq \emptyset$ just before u is inserted into S. There must also be some path from s to u, otherwise $d[u] = \delta(s, u) = \infty$, which would violate our assumption that $d[u] \neq \delta(s, u)$. Because there is at least one path from u to s, there is a shortest path p from s to u. Path p connects a vertex in S, namely s, to a vertex in $V - S$, namely u. Let us consider the first vertex y along p such that $y \in V - S$, and let $x \in V$ be y's predecessor. Thus, as shown in Figure 2.5, path p can be decomposed as $s \overset{p_1}{\rightsquigarrow} x \rightarrow y \overset{p_2}{\rightsquigarrow} u$

We claim that $d[y] = \delta(s, y)$ when u is inserted into S. To prove this claim observe that $x \in S$. Then, because u is chosen as the first vertex for which $d[u] \neq \delta(s, u)$ when it is inserted into S, we had $d[x] = \delta(s, x)$ when x is inserted into S. Edge (x, y) was relaxed at that time, so the claim can be proved.

We can now obtain a contradiction to prove the theorem. Because y occurs before u on a shortest path from s to u and all edge weights are nonnegative (notably those on path p_2), we have $\delta(s, y) \leq \delta(s, u)$, and thus

$$\begin{aligned} d[y] &= \delta(s, y) \\ &\leq \delta(s, u) \\ &\leq d[u] \end{aligned} \tag{2.1}$$

But because both vertices u and y were in $V - S$ when u was chosen in line 5, we have $d[u] \leq d[y]$. Thus, the two inequalities in equation (2.1) are in fact equalities, giving $d[y] = \delta(s, y) = \delta(s, u) = d[u]$.

Consequently, $d[u] = \delta(s, u)$, which contradicts our choice of u. We conclude that at the time each vertex $u \in V$ is inserted into set S, we have $d[u] = \delta(s, u)$, and it is being proved that this equality holds thereafter.

■

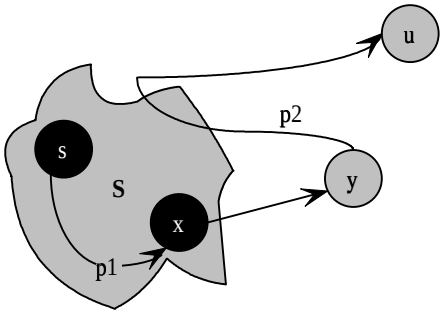


Figure 2.5: The proof of theorem 2.2

2.3.2 Analysis

The running time of Dijkstra's algorithm depends on the implementation of the priority queue Q. In this section we will examine two implementations:

- A naive implementation with a linear array
- An implementation with a priority queue using a Min-Heap (see appendix B)

In both cases the initialization in lines 1–3 takes $O(n)$ time because it examines all the vertices but the source exactly once. Each vertex is extracted exactly once from the queue since it is inserted only once in line 6, therefore the **while** loop in lines 7–13 is executed $O(n)$ times. The loop in lines 10–13 is executed once for every edge in the graph, that is m times totally.

If we implement the queue with a linear array, the operation EXTRACT-MIN takes $O(n)$ time because we need to examine all the elements of the array in order to find the minimum. Thus the total time spent in the operations EXTRACT-MIN will be $O(n^2)$. The total time of the algorithm will be

$$O(\underbrace{n^2}_{\text{EXTRACT-MIN}} + \underbrace{n}_{\text{initialization}} + \underbrace{m}_{\text{loop in lines 10–13}}) = O(n^2)$$

On the other hand, if we implement the queue using a Min-Heap data structure, the operation EXTRACT-MIN takes $O(\log n)$ time as shown in Appendix B.1.2 and there are n such operations. Moreover the assignment in line 12 must be changed to the operation DECREASE-KEY that also takes $O(\log n)$ time. This operation will be executed m times totally and so we have:

$$\begin{aligned} O(\underbrace{n \log n}_{\text{EXTRACT-MIN}} + \underbrace{n}_{\text{initialization}} + \underbrace{m \log n}_{\text{loop in lines 10–13}}) &= O((m+n)\log n) \\ &= O(m \log n) \end{aligned}$$

2.4 Minimum Spanning Trees

Given a weighted graph $G = (V, E)$, its weighted function $w : E \mapsto \mathbb{R}$, the acyclic subset $T \subseteq E$ that connects all the vertices of G whose total weight,

$$w(T) = \sum_{(u,v) \in T} w(u, v)$$

is minimized, then it is called a minimum spanning tree. The problem of finding this spanning tree is called the minimum spanning tree problem.

There are two proposed algorithms for solving this problem: Kruskal's algorithm and Prim's algorithm. Both algorithms are based on BFS and they both use a "greedy" strategy in order to solve the problem. A greedy algorithm, when it gets to a point where it has to choose between several alternatives, it chooses the one that is best at the moment. This strategy does not necessarily find the globally optimal solution, but it is proven that for the minimum spanning tree problem, these algorithms produce a spanning tree with minimum weight.

Let us now give two definitions that will be useful in the next sections:

- A *cut* $(S, V-S)$ of an undirected graph $G = (V, E)$ is a partition of V into two subsets S and $V - S$.
- We say that an edge $(u, v) \in E$ *crosses* the cut $(S, V - S)$ if one of its endpoints is in S and the other in $V - S$.
- *Light edge* is called an edge that crosses a cut of a weighted graph and has the minimum weight of all the edges that cross the cut.

Note that there might be more than one light edges, if more than one edges that cross a given cut have the same (minimum) weight.

2.4.1 General approach

Here we analyze a first general approach in order to solve the minimum spanning tree problem. We assume that we have a connected undirected graph $G = (V, E)$ with a weight function $w : E \mapsto \mathbb{R}$ and we want to find a minimum spanning tree for G. The pseudo-code for this first approach is given below.

MST-Generic

Input: A graph $G = (V, E)$ and a weight function $w : E \mapsto \mathbb{R}$

Output: A set A containing all the edges of a minimum spanning tree.

```
MST-Generic(G, w)
1  A ← ∅
2  while A does not form a minimum spanning tree
3      do find a safe edge (u,v)
4          A ← A ∪ {(u, v)}
5  return A
```

The algorithm takes as input the graph G and the function w and produces as an output a set A of edges that is a minimum spanning tree. This tree is not unique for a given graph and a given weight function. During the execution of the algorithm the set A is a subset of the tree.

In the first line, A is initialized to contain no elements. In lines 2–4, we have the main loop of the algorithm which is executed until A forms a spanning tree. Inside the loop we find a safe edge (u, v) and we insert it into A. By the term *safe edge* we mean an edge (u, v)

that does not violate the condition that $A \cup \{(u, v)\}$ is still a subset of a minimum spanning tree. Finally the algorithm returns the set A.

This algorithm faces the problem of finding a safe edge to include into the set A. We know that this problem can be solved due to the following fact. There must be a spanning tree T, such that $A \subseteq T$, and if there is an edge $(u, v) \in T$ such that $(u, v) \notin A$, then (u, v) is safe for A. The problem is to identify this safe edge. Both Kruskal's and Prim's algorithms determine the rule of finding this safe edge for the set A.

2.4.2 Kruskal's algorithm

In Kruskal's algorithm the safe edge (u, v) added to A is always a least weight edge that connects two distinct components. It can be proved that (u, v) is a safe edge. The pseudo-code of Kruskal's algorithm is shown below.

MST-KRUSKAL

Input: A graph $G = (V, E)$ and a weight function $w : E \mapsto \mathbb{R}$.

Output: A set A containing all the edges of a minimum spanning tree.

```
MST-KRUSKAL(G, w)
1  A ← ∅
2  for each vertex u in V[G]
3      do MAKE-SET(u)
4   sort the edges of E by nondecreasing weight w
5  for each edge (u, v) in E, in order by nondecreasing weight
6      do if FIND-SET(u) ≠ FIND-SET(v)
7            then A ← A ∪ {(u, v)}
8                 UNION(u, v)
9  return A
```

The implementation of the algorithm uses a disjoint set data structure (see Appendix B) to maintain the several disjoint sets of elements. Each of these sets contains the vertices in a tree of the current forest.

In the first line of the algorithm, A is initialized to contain no elements. In lines 2–3 n trees are created, one for each vertex $v \in V$. In line 4 all the edges in E are sorted into nondecreasing order according to their weight. The main **for** loop in lines 5–8 is executed for every edge of E in order of nondecreasing weight. Inside the loop we determine if two vertices u and v belong to the same tree by testing if FIND-SET(u) and FIND-SET(v) are equal. If that happens then (u, v) cannot be added to A because this will create a cycle, so the edge is discarded. If this is not the case then (u, v) is a safe edge and it is inserted into A. Also, we merge the trees that contain u and v by calling UNION(u, v). Finally, the algorithm returns the set A.

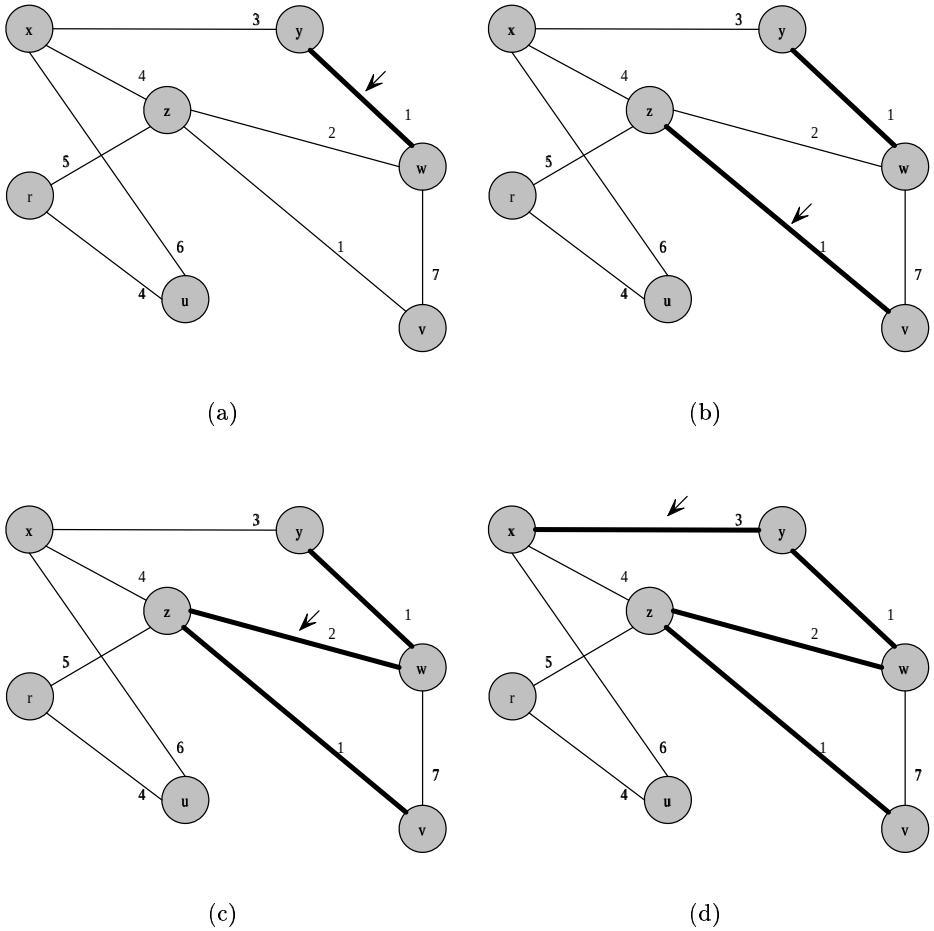
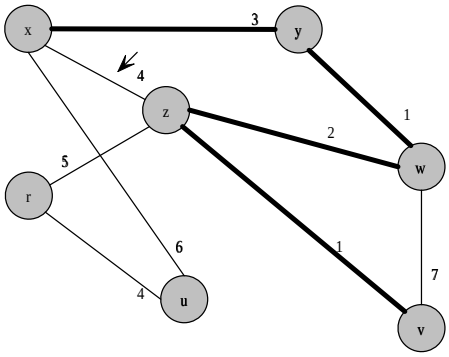
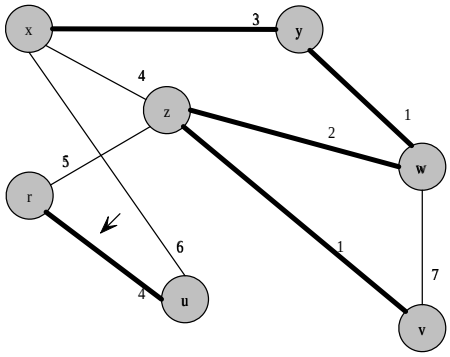


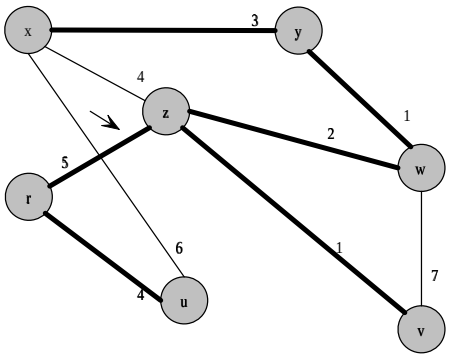
Figure 2.6: The operation of Kruskal's algorithm (part 1)



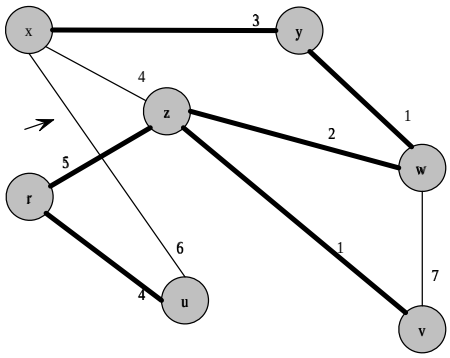
(g)



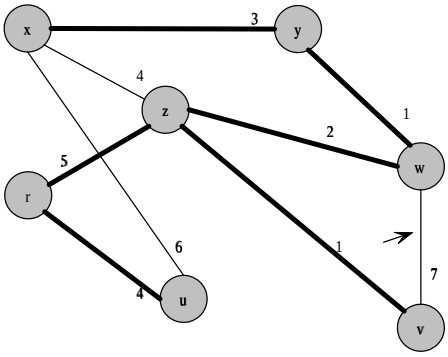
(h)



(i)



(j)



(k)

Figure 2.6: The operation of Kruskal's algorithm (part 2)

In Figure 2.6, we can see an example of the operation of Kruskal's algorithm. In Figure 2.6(a), we can see that the edge with the minimum weight has been selected. In the next steps, the edge with the smallest weight among the non-selected vertices are selected. For each edge, if its adjacent vertices belong to different trees, we paint it black, e.g. edge (z, w) in Figure 2.6(c), else it remains white, e.g. edge (x, z) in Figure 2.6(g). Finally the black edges construct the desirable minimum spanning tree.

The running time of Kruskal's algorithm depends on the implementation of the disjoint-set data structure (see Appendix B.2). Initialization takes time $O(n)$ and the time to sort the edges in line 4 is $O(m \log m)$. There are $O(m)$ operations of the disjoint set forest, that take $O(m \log m)$ time in total, so the overall time is $O(m \log m)$.

2.4.3 Prim's algorithm

Like Kruskal's algorithm, Prim's algorithm is a special case of the generic minimum spanning tree algorithm. Prim's algorithm operates with a similar way to Dijkstra's algorithm. The Prim's algorithm is given below in pseudo-code.

MST-PRIM

Input: A graph $G = (V, E)$, its weight function $w : E \mapsto \mathbb{R}$ and a source vertex r.

Output: A minimum spanning tree containing all the vertices of G and the key value and the parent of each vertex.

```
MST-PRIM(G, w, r)
 1  Q ← V[G]
 2  for each u in Q
 3      do key[u] ← ∞
 4  key[r] ← 0
 5  π[r] ← NIL
 6  while Q ≠ ∅
 7      do u ← EXTRACT-MIN(Q)
 8          for each v in Adj[u]
 9              do if v in Q and w(u, v) < key[v]
10                    then π[v] ← u
11                         key[v] ← w(u, v)
```

The connected graph G, its weight function w and the root r of the minimum spanning tree are given as input to the algorithm. For each vertex $v \in V$ the algorithm holds an attribute $key[v]$, which contains the minimum weight of any edge connecting v to a vertex in a tree, and an attribute $\pi[v]$, which contains the parent of the vertex v. When the algorithm is executed, all the vertices that are not included in the tree are stored in a priority queue Q based on their key fields.

In lines 1–3, the algorithm initializes the priority queue Q to contain all the vertices of G, and sets the key of each vertex to ∞. In lines 4–5, the attributes concerning the root vertex r are initialized. Lines 6–11 contain the main **while** loop of the algorithm which is executed until Q is empty. In every iteration the minimum vertex (according to each key value) is extracted from the queue and its adjacent vertices are examined. So if u is the minimum

vertex, the algorithm examines for each $v \in Adj[u]$ if $v \in Q$ and if $w(u,v) < key[v]$. If these conditions are met, then we assign to the attributes $key[v]$ and $\pi[v]$ the values $w(u,v)$ and u respectively. Throughout the execution of the algorithm $V - Q$ contains the vertices in the tree being grown and Q contains the remaining vertices that need to be examined. By extracting the minimum vertex from Q we identify a vertex u incident on a light edge crossing the cut $(V - Q, Q)$. Also by removing it from Q we add u to the set $V - Q$ of vertices in the tree. When the algorithm terminates the priority queue Q is empty.

At the end of the algorithm the minimum spanning tree A can be constructed by the information which was obtained through the execution of the algorithm. So the minimum spanning tree A for G is

$$A = \{(v, \pi[v]) : v \in V - \{r\}\}$$

The performance of the Prim's algorithm depends on the implementation of the priority queue Q. There are three possible implementations:

- With a simple implementation of the priority queue Q (i.e. an array), the algorithm requires $O(n^2)$ time.
- If Q is implemented as a binary heap we can replace the initialization in lines 1–4 with the procedure of the creation of the heap which takes $O(n)$ time. The while-loop, in lines 6–11, is executed n times, and since each EXTRACT-MIN operation takes $O(\log n)$ time, the total time for all calls of this operation is $O(n \log n)$. The for-loop in lines 8–11 is executed $O(m)$ times because the sum of the lengths of all adjacency lists is $2m$. In line 9 we can test the membership of v in Q in constant time by keeping a bit for each vertex that tells whether or not $v \in Q$, and updating this bit when the vertex is removed. In line 11 we can replace the assignment with a DECREASE-KEY operation on the heap, which can be implemented in $O(logn)$ time. Hence, the total time for Prim's algorithm is $O(n \log n + m \log n) = O(m \log n)$.
- We can achieve $O(m + n \log n)$ time by implementing the priority queue Q with a Fibonacci heap.

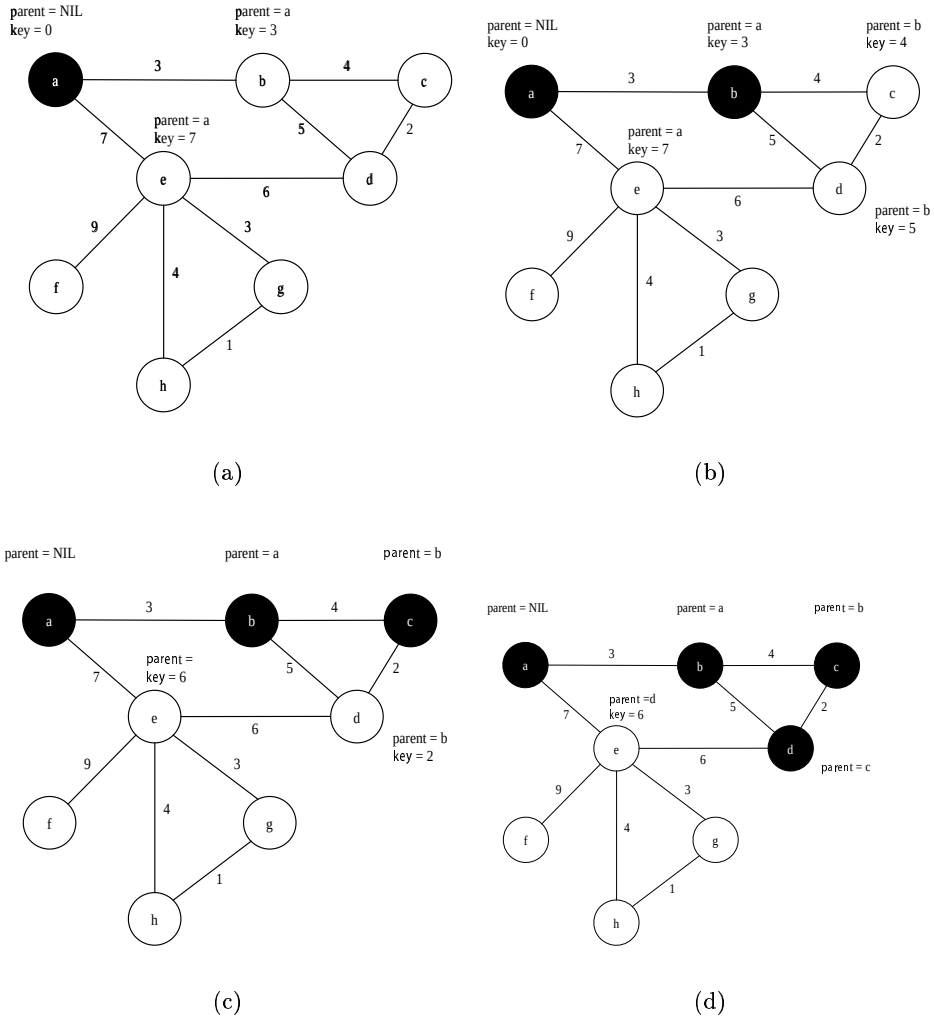
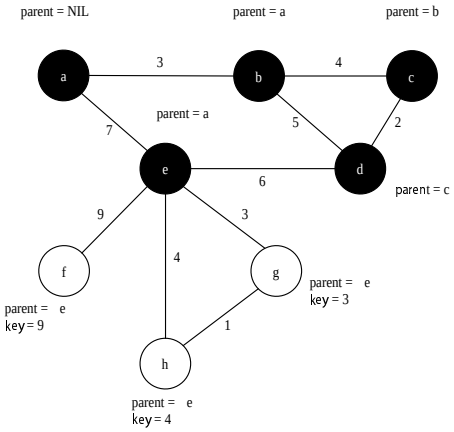
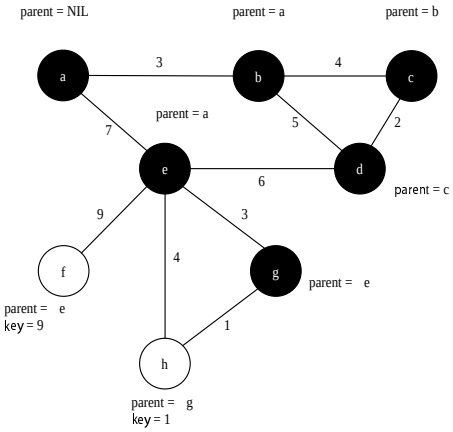


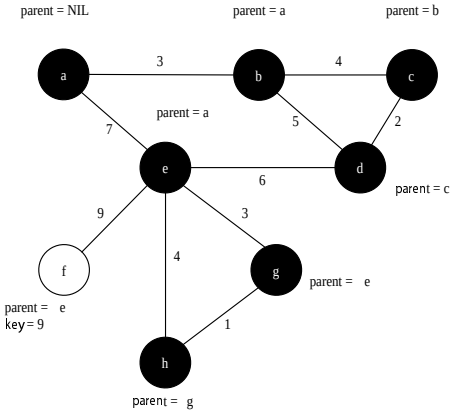
Figure 2.7: The execution of Prim's algorithm (part 1)



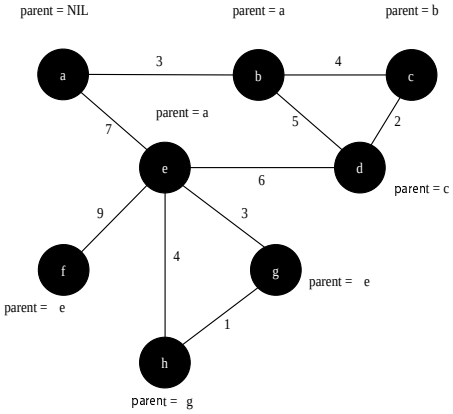
(e)



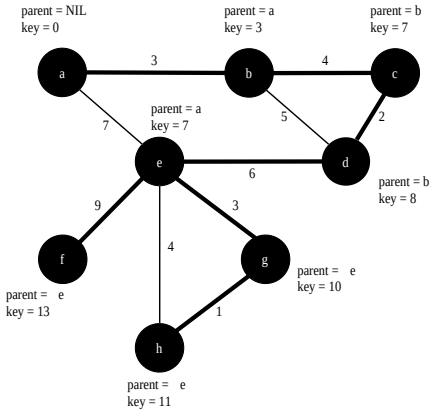
(f)



(g)



(h)



(i)

Figure 2.7: The execution of Prim's algorithm (part 2)

In Figure 2.7 we can see an example of the operation of Prim's algorithm. We start from the source edge a and for each edge examined we can see its key value and its parent value. As the algorithm proceeds these values can change. In Figure 2.7(i) we can see the final step of the algorithm and we can construct the minimum spanning tree by selecting the edges, which we have painted black so far.

Appendix A

Partitions and equivalence relations

A.1 Mathematical definitions

We know that $\mathbb{Q}$ is the set of all rational numbers, that is the numbers that can be expressed as quotients m/n of integers, where $n \neq 0$. It would be a mistake to describe $\mathbb{Q}$ as the set S of all "fractional expressions" m/n where m and n are integers and $n \neq 0$. That is because $\frac{2}{3}$ and $\frac{4}{6}$ are of course different fractional expressions, but we know that they both represent the *same* rational number. In reality, every element of $\mathbb{Q}$ is represented by infinite different elements of S. When we use the rational numbers in arithmetic, we see as *equivalent* all the elements of S that represent the same rational number in $\mathbb{Q}$.

The previous example is typical of situations that we consider different elements of a set to be arithmetically or algebraically equivalent. In these situations our set is *partitioned* in subsets and each of these subsets is a different entity. If b is an element of a such partitioned set, we usually write $\overline{b}$ to mean the subset of all the elements that are equivalent with b.

Example A.1 *Let S the set of all the fractional expressions m/n, where $m, n \in \mathbb{Z}$ and $n \neq 0$. The subset $\overline{\frac{2}{3}}$ of all the elements of S that are equivalent with the number $\frac{2}{3} \in \mathbb{Q}$ is*

$$\begin{aligned}\overline{2/3} &= \left\{ \frac{2}{3}, \frac{-2}{-3}, \frac{4}{6}, \frac{-4}{-6}, \dots \right\} \\ &= \left\{ \left. \frac{2n}{3n} \right| n \in \mathbb{Z} \text{ and } n \neq 0 \right\}\end{aligned}$$

□

Let us now give an exact definition of the partition of a set.

Definition A.1 (Partition) Partition of a set *is called an analysis of the set into subsets such that, every element of the set belongs to exactly one of the subsets. Those are the* partition subsets.

Example A.2 *Let $S = \{1, 2, 3, 4, 5, 6\}$. A partition of S is given by the subsets*

$$\{1, 6\},\ \{3\},\ \{2, 4, 5\}.$$

The subsets {1,2,3,4} and {4,5,6} are not a partition of S because 4 is in both subsets. The subsets {1,2,3} and {5,6} are not a partition because 4 is not in any of them.

□

How can we see if two fractional expressions m/n and r/s of the partitioned set S of the example A.1 belong in the same subset (i.e. they represent the same rational number)? One way is to reduce the fractions. This might not be easy; for example both 1909/4897 and 1403/3599 represent the same rational number because:

$$\frac{1909}{4897} = \frac{23 \cdot 83}{59 \cdot 83} \text{ and } \frac{1403}{3599} = \frac{23 \cdot 61}{59 \cdot 61}$$

This procedure is a difficult task. But we know that $m/n = r/s$ if and only if $ms = nr$. This gives us a more efficient way to determine if $m/n = r/s$:

$$(1909)(3599) = (4897)(1403) = 6870491$$

Let the relation $a \odot b$ state that a and b belong to the same subset of a given partition of a set S that contains a and b. The following properties are valid:

- $a \odot a$. Element a belongs in the same subset with itself.
- If $a \odot b$ then $b \odot a$. If element a is in the same subset with element b, then element b is in the same subset with element a.
- If $a \odot b$ and $b \odot c$ then $a \odot c$. If element a is in the same subset with element b and element b is in the same subset with element c, then element a is in the same subset with element c.

The following theorem is basic. It states that a relation $\odot$ among the elements of a set, that satisfies the above properties defines naturally a partition of the set.

Theorem A.1 *Let S be a non-empty set and $\odot$ a relation among the elements of S that satisfies the following properties for all $a, b, c \in S$.*

1. **reflexive:** $a \odot a, \ \forall a \in S,$
2. **symmetric:** $a \odot b \Rightarrow b \odot a, \ \forall a, b \in S$ *and*
3. **transitive:** $a \odot b,$ *and* $b \odot c \Rightarrow a \odot c$

Then $\odot$ defines naturally a partition of S, where

$$\overline{a} = \{x \in S | x \odot a\}$$

is the subset that contains a, for all $a \in S$. Conversely, every partition of S gives naturally a relation $\odot$ that satisfies the reflexive, symmetric and transitive property, if $a \odot b$ is defined to mean $a \in \overline{b}$.

Proof. The "converse" part of the theorem is already proven.

For the "straight" part we must prove that the subsets that are defined by $\overline{a} = \{x \in S | x \odot a\}$ are indeed a partition of S, that is, every element of S belongs to *exactly one subset*. Let $a \in S$. Then $a \in \overline{a}$ from the reflexive property (1), consequently a belongs to *at least one* subset.

If we assume now that a belongs also to the subset $\overline{b}$, we must prove that $\overline{a} = \overline{b}$. This means that a belongs to *at most one* subset of S. The standard way to prove that two sets are equal is to prove that each one is a subset of the other.

First we prove that $\overline{a} \subseteq \overline{b}$. Let $x \in \overline{a}$. Then $x \odot a$. Because $a \in \overline{b}$, $a \odot b$. From the transitive property (3) we get that $x \odot b \Rightarrow x \in \overline{b}$. Thus $\forall x \in \overline{a}, x \in \overline{b} \Rightarrow \overline{a} \subseteq \overline{b}$.

Now we prove that $\overline{b} \subseteq \overline{a}$. Let $y \in \overline{b}$. Then $y \odot b$. We also know that $a \in \overline{b}$ consequently $a \odot b$, and from the symmetric property (2), $b \odot a$. Then using the transitive property (3), $y \odot a \Rightarrow y \in \overline{a}$. Thus $\overline{b} \subseteq \overline{a} \Rightarrow \overline{a} = \overline{b}$.

■

Definition A.2 (Equivalence relation and equivalence class) *A relation $\odot$ on a set S that satisfies the reflexive, the symmetric and the transitive properties, as described in theorem A.1, is called* equivalence relation on S. *Every subset $\overline{a}$ of the natural partition that is defined by an equivalence relation, is called an* equivalence class.

Example A.3 *Let us verify that $m/n \odot r/s$ if and only if $ms = nr$ is an equivalence relation on the set S of the fractional expressions.*

Reflexive. *$m/n \odot m/n$ because $mn = mn$*

Symmetric. *If $m/n \odot r/s$ then $ms = nr$. Therefore $nr = ms$ and $r/s \odot m/n$.*

Transitive. *If $m/n \odot r/s$ and $r/s \odot u/v$ then $ms = nr$ and $rv = su$. Therefore $mvs = vms = vnr = nrv = nsu = nus$. Because $s \neq 0$ we have $mv = nu$ thus $m/n \odot u/v$.*

Every equivalence class of S is considered to be a rational number.

□

A.2 Equivalence relations and graphs

In this section we prove that the relations 'connected to' in an undirected graph and 'strongly connected to' in a directed graph are equivalence relations on the vertex set.

We first consider the undirected case. As we saw in section 1.1 we say that vertex u_i is connected to vertex u_j (we write for conciseness $u_i \odot u_j$)if there is a path from u_i to u_j. We must prove that the three properties of the equivalence relations hold for the relation $u_i \odot u_j$:

Reflexive. As mentioned in Section 1.1 by convention every vertex is connected to itself. Thus $u_i \odot u_i$.

Symmetric. If $u_i \odot u_j$ then there is a path $P = \{u_i,\ u_{i+1},\ u_{i+2},\ \ldots,\ u_{j-1},\ u_j\}$ from u_i to u_j. The inverse path $P' = \{u_j,\ u_{j-1},\ \ldots,\ u_{i+2},\ u_{i+1}, u_i\}$ connects the vertex u_j to the vertex u_i. Thus if $u_i \odot u_j$ then $u_j \odot u_i$.

Transitive. If $u_i \odot u_j$ and $u_j \odot u_k$ then there are paths $P_1 = \{u_i,\ u_{i+1},\ \ldots,\ u_{j-1},\ u_j\}$ from u_i to u_j and $P_2 = \{u_j,\ u_{j+1},\ \ldots,\ u_{k-1},\ u_k\}$ from u_j to u_k. We now define the path $P_3 = \{u_i,\ u_{i+1},\ \ldots,\ u_{j-1},\ u_j,\ u_{j+1},\ \ldots,\ u_{k-1},\ u_k\}$. This path connects u_i with u_k. Thus relation $\odot$ has all the three properties of an equivalence relation.

■

We now consider the directed case. We say that u_i is strongly connected to u_j if there is a (directed) path from u_i to u_j *and* a directed path from u_2 to u_1. Let us examine if this relation has the three properties of an equivalence relation. If u_i is strongly connected to u_j we write $u_i \otimes u_j$

Reflexive. By convention every vertex is strongly connected to itself. Thus $u_i \otimes u_i$.

Symmetric. If $u_i \otimes u_j$ then there are paths $P = u_i,\ u_{i+1},\ u_{i+2},\ \ldots,\ u_{j-1},\ u_j$ from u_i to u_j and $P' = u_j,\ u_{j+1},\ u_{j+2},\ \ldots,\ u_{i-1},\ u_i$ from u_j to u_i. Thus by definition $u_j \otimes u_i$.

Transitive. If $u_i \otimes u_j$ and $u_j \otimes u_k$ then there are (directed) paths

$$\begin{aligned}
P_1 &= \{u_i,\ u_{i+1},\ \ldots,\ u^1_{j-1},\ u_j\} \text{ from } u_i \text{ to } u_j \\
P'_1 &= \{u_j,\ u^1_{j+1},\ \ldots,\ u_{i-1},\ u_i\} \text{ from } u_j \text{ to } u_i \\
P_2 &= \{u_j,\ u^2_{j+1},\ \ldots,\ u_{k-1},\ u_k\} \text{ from } u_j \text{ to } u_k \\
P'_2 &= \{u_k,\ u_{k+1},\ \ldots,\ u^2_{j-1},\ u_j\} \text{ from } u_k \text{ to } u_j
\end{aligned}$$

Note that u_{j-1} is not necessarily the same vertex in paths P_1 and P'_2 and that is why we denote it by u^1_{j-1} in P_1 and u^2_{j-1} in P_2. The same is true for the vertex u_{j+1}.

Let us now define the paths

$$\begin{aligned}
P_3 &= \{u_i,\ u_{i+1},\ \ldots,\ u^1_{j-1},\ u_j,\ u^2_{j+1},\ \ldots,\ u_{k-1},\ u_k\} \text{ and} \\
P'_3 &= \{u_k,\ u_{k+1},\ \ldots,\ u^2_{j-1},\ u_j,\ u^1_{j+1},\ \ldots,\ u_{i-1},\ u_i\}
\end{aligned}$$

The existence of these paths proves that $u_i \otimes u_k$, therefore $\otimes$ is an equivalence relation.

■

The equivalence classes that are defined by the relation 'connected to' are called connected components of a graph and the equivalence classes that are defined by the relation 'strongly connected to' are called strongly connected components. We continue with one example.

Example A.4 *In Figure we see an undirected graph. Each of the vertices u_1, u_2, u_3 is connected to the others. Also u_4 is connected to u_5. In this graph there are two connected components (equivalence classes) one containing the vertices u_1, u_2 and u_3 and the other containing the remaining vertices.*

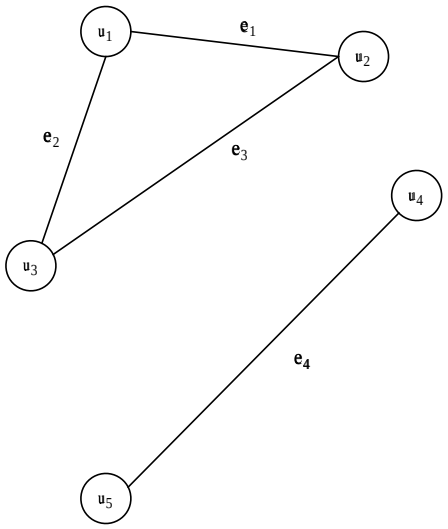


Figure A.1: An undirected graph with two connected components

Appendix B

Useful data structures

In this Appendix we present some very useful data structures for graph algorithms.

B.1 Heaps and Priority Queues

The *(binary) heap* is a data structure that can be represented as a complete binary tree (see Figure B.1(a)) with nodes that follow the same inequality to their parent node: for each node (except from the root that has not a parent) the value is less or equal to the value of its parent (Max-Heap) or greater or equal to the value of its parent (Min-Heap). This rule describes the *heap property* and leaves us with the root element having the greatest (in the former case) or the smallest (in the later) value of all.

The data structure can be implemented using an array whose elements are the nodes of the binary tree (see figure B.1(b)). Let A be such an array. A may contain more elements than the heap that it implements by using two attributes: the length of the array ($length[A]$) and the size of the heap implemented by A ($size[A]$). Obviously, $size[A] \leq length[A]$.

The root of the tree is the first element of A ($A[1]$). For an arbitrary node with index i, we can find the indices of its parent (PARENT(i)), and its two children (LEFT(i) and RIGHT(i)) as follows:

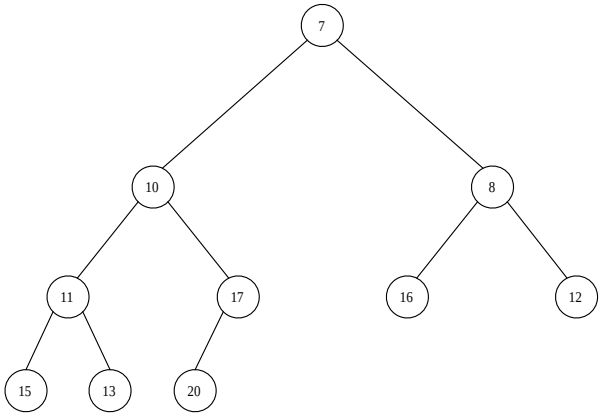
```
PARENT(i)
    return ⌊i/2⌋

LEFT(i)
    return 2i

RIGHT(i)
    return 2i + 1
```

In order to satisfy the heap property that was described earlier we must have:

$$A[\text{PARENT}(i)] \geq A[i] \text{ for the Max-Heap}$$

or

$$A[\text{PARENT}(i)] \leq A[i] \text{ for the Min-Heap}$$



(a) The tree representation of a heap

1	2	3	4	5	6	7	8	9	10
7	10	8	11	17	16	12	15	13	20

(b) The array implementation of a heap

Figure B.1: A heap represented by a binary tree and implemented by an array

The *height* of a node in a tree is called the number of edges of the longest downward path from the node to a leaf, and height of the tree is called the height of its root. Thus, a heap of n elements has height $\Theta(\log n)$ since it is a complete binary tree. The main operations on heaps run in time at most proportional to its height and therefore take $O(\log n)$ time

We will now discuss two basic procedures of the heaps and two accessory procedures that allow a heap to be used as a priority queue. We will be concerned with Min-Heaps, although the procedures for a Max-Heap are equivalent.

B.1.1 Heap procedures

The two basic procedures that are defined on a heap are:

- The HEAPIFY which runs in $O(\log n)$ time and maintains the heap property, and
- the BUILD-HEAP which runs in linear time and produces a heap from an unordered array.

The HEAPIFY procedure takes as input an array A and an index i. The procedure assumes that the sub-trees having roots the elements LEFT(i) and RIGHT(i) are already heaps, but $A[i]$ is greater than its children, thus violating the Min-Heap property. This procedure lets the element in $A[i]$ to "float down" in the heap so that the tree rooted at $A[i]$ becomes a heap.

HEAPIFY

Input: An array A and an index i. The heap property must hold for the subtrees rooted at LEFT(i) and RIGHT(i).

Output: An array A for which the heap property holds for the subtree rooted at i.

```
HEAPIFY(A, i)
 1  l ← LEFT(i)
 2  r ← RIGHT(i)
 3  if l ≤ size[A] and A[l] ≤ A[i]
 4     then smallest ← l
 5     else  smallest ← i
 6  if r ≤ size[A] and A[r] ≤ A[smallest]
 7     then smallest ← r
 8  if smallest ≠ i
 9     then  exchange  A[i] ↔ A[smallest]
10           HEAPIFY(A, smallest)
```

This procedure compares the values of the elements i LEFT(i) and RIGHT(i) and finds the index of the element with the smallest value of the three. If this element is other than $A[i]$, the procedure interchanges the elements $A[i]$ and $A[smallest]$ and calls recursively itself for the subtree rooted now at i

The HEAPIFY procedure takes $O(\log n)$ to run. That is because the operations in the function take $\Theta(1)$ time and the function is evoked $\log n$ times at most.

We can now use HEAPIFY to get BUILD-HEAP procedure, which builds a heap from a completely unordered array $A[1..n]$ where $n = length[A]$. This procedure is designed based on the observation that when A is a heap the elements $A[(n/2 + 1)..n]$ are the leaves of the tree and therefore are already heaps. We can start building the heap by applying the HEAPIFY procedure on the smallest (with the fewest elements) subtrees and then moving to the grater ones until reaching the root, which will give us a correct heap. This is achieved by calling HEAPIFY procedure for the subtrees rooted at $n/2 \ldots 1$ as follows:

BUILD-HEAP

Input: An unordered array A.

Output: An array A that represents a heap having as elements the elements of the input array.

```
BUILD-HEAP(A)
1  size[A] ← length[A]
2  for i ← ⌊length[A]/2⌋ downto 1
3     do HEAPIFY(A, i)
```

At first glance we note that the required time for the procedure is $O(n\ logn)$. However we can observe that the running time of HEAPIFY varies with the height of the node in the tree and that most nodes have small heights. In fact we can prove that in an n-element heap there are at most $\lceil n/2^{h+1} \rceil$ nodes of height h. As we saw the time required by HEAPIFY

on a subtree with n elements is $O(\log n) = O(h)$ where $h = \log n$ is the height of the node. Therefore the total time for BUILD-HEAP is:

$$\sum_{h=0}^{\lfloor \log n \rfloor} \lceil \frac{n}{2^{h+1}} \rceil O(h) = O\left(n \sum_{h=0}^{\lfloor \log n \rfloor} \frac{h}{2^h} \right)$$

We know that

$$\sum_{k=0}^{\infty} k x^k = \frac{x}{(1-x)^2}$$

if $|x| < 1$, therefore

$$\sum_{h=0}^{\infty} h(1/2)^h = \frac{1/2}{(1-1/2)^2} = 2$$

Thus, the running time of BUILD-HEAP can be bounded as

$$\begin{aligned} O\left(n \sum_{h=0}^{\lfloor \log n \rfloor} \frac{h}{2^h} \right) &= O\left(n \sum_{h=0}^{\infty} \frac{h}{2^h} \right) \\ &= O(n) \end{aligned}$$

B.1.2 Priority queue procedures

If we supply the heap with two more procedures, we can use it to build a priority queue. These procedures are:

- the INSERT procedure, that inserts an element in the queue and
- the EXTRACT-MIN that removes and returns the smallest element in the queue.

We can also define the procedure MINIMUM that returns the minimum value without removing it from the queue.

One natural way to implement a priority queue is using heaps. The operation HEAP-MINIMUM returns the first element of the array A in $\Theta(1)$ time. The HEAP-EXTRACT-MIN extracts the root element (i.e. the minimum element) of the heap and replaces it with a leaf (i.e. the last element of the array). Following, we have a tree with the two subtrees of the root being correct heaps. Therefore the HEAPIFY procedure is applied to the first element of the array in order to reform the tree to maintain the heap property.

HEAP-EXTRACT-MIN

Input: An array A representing a heap.

Output: The node at the root of the heap. The remaining elements still form a heap.

```
HEAP-EXTRACT-MIN(A)
1  if size[A] < 1
2     then error "heap underflow"
```

```
3  min ← A[1]
4  A[1] ← A[size[A]]
5  size[A] ← size[A] − 1
6  HEAPIFY(A, 1)
7  return min
```

The running time of HEAP-EXTRACT-MIN is $O(\log n)$ since it performs only a constant amount of work before the call to HEAPIFY that takes $O(\log n)$ time.

The HEAP-INSERT inserts a new element in the heap, by adding one more leaf and then traversing the path from this leaf to the root until it finds a proper place for the new element to be placed.

HEAP-INSERT

Input: An array A representing a heap and an element to be inserted.

Output: The heap containing the element.

```
HEAP-INSERT(A, element)
1  size[A] ← size[A] + 1
2  i ← size[A]
3  while i > 1 and A[PARENT(i)] < element
4      do A[i] ← A[PARENT(i)]
5          i ← PARENT(i)
```

The procedure runs in $O(\log n)$ time since the path traced from the leaf to the root of the heap has $O(\log n)$ length.

B.2 Disjoint Sets

A *disjoint-set data structure* is a data structure that contains disjoint dynamic sets. Let $\mathcal{S} = \{S_1,\ S_2,\ \ldots,\ S_n\}$ be a set of disjoint sets, that is

$$S_i \cap S_j = \emptyset,\ \forall i \neq j \text{ with } 1 \leq i \leq n \text{ and } 1 \leq j \leq n$$

Each set S_i has a *representative* which is an arbitrary member of the set but always the same one if the set is not modified. There are three basic operations on the data structure:

MAKE-SET(x) creates a new set, containing only the element x. The element x may not be a member of another set because that would prevent the sets from being disjoint.

UNION(x, y) unites the two sets to which x and y belong (S_x and S_y) into one new set $S_{x \cup y}$ with elements all the elements of both the initial sets. The initial sets may not belong to $\mathcal{S}$ any more (because $S_x \cap S_{x \cup y} \neq \emptyset$ and $S_y \cap S_{x \cup y} \neq \emptyset$) and thus, are destroyed. The representative of the resulting set can be any member of the new set. Actually one of the two initial representatives becomes the representative of the resulting set.

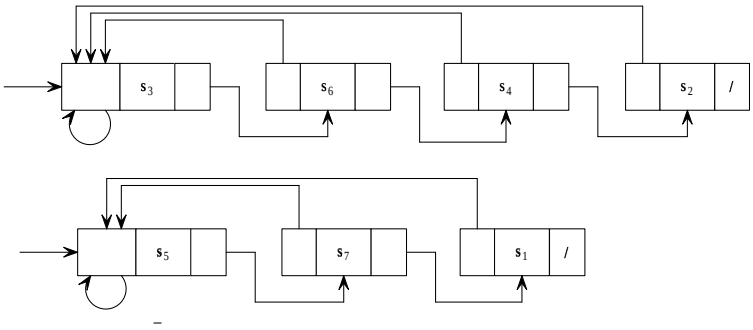
FIND-SET(x) returns a pointer to the representative of the set containing x.

B.2.1 Implementation of disjoint sets with linked lists

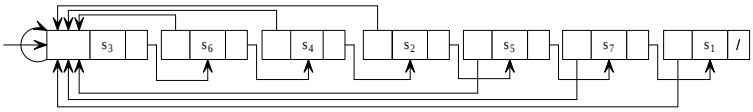
We can implement the disjoint-set data structure by using a collection of linked lists. Each list represents a set and its first element is the representative. Every element points back to the representative and has a pointer to the next element of the list. The order of appearance of the elements in the list is not important.

It is easy to observe that the MAKE-SET and FIND-SET operations are easily implemented and run in $O(1)$ time: MAKE-SET(x) creates a new list containing only one element, x. FIND-SET(x) returns the pointer from x to the representative of the list.

However UNION(x, y) has more things to do: this function not only has to connect S_x at the end of the list S_y (operation that requires $O(1)$ time) but it also has to change the representative pointers of all S_x's elements to point to the representative of S_y. Therefore this procedure takes time linear in the number of elements of S_x.



(a) Two disjoint sets represented by linked lists



(b) The union of the disjoint sets

Figure B.2: Two disjoint sets represented by linked lists and their union

An obvious change that we can make in the above implementation is to keep information about the length of each list in the representative. This way we can append the shorter list at the end of the longer, with ties broken arbitrarily. Using this technique, which is called *weighted union heuristic*, a single UNION still takes $\Omega(m)$ time if both sets have $\Omega(m)$ members. We can prove, however, that a sequence of m MAKE-SET, UNION and FIND-SET operations, n of which are MAKE-SET operations takes $O(m + n \log n)$ instead of $\Theta(m^2)$ time[1].

One of the many applications of the disjoint set data structure is to compute the connected components of a graph. The algorithm is given below.

CONNECTED-COMPONENTS

[1] See [1] chapter 22 for more information about the running time of the operations on a disjoint set data structure

Input: A graph $G = (V, E)$.

Output: A collection of disjoint sets, each one representing a connected component of G.

```
CONNECTED-COMPONENTS(G)
1  for each vertex v in V
2      do MAKE-SETu
3  for each edge (u, v) in E
4      do if FIND-SET(u) ≠ FIND-SET(v)
5            then UNION(u, v)
```

Initially the above procedure places each vertex v in a different set. Then, for each edge (u, v), it unites the sets containing u and v if they are different. We observe that there are n MAKE-SET operations, $2m$ FIND-SET operations, and at most m UNION operations. So in total there are 3m+n operation, n of which are MAKE-SET. As stated earlier, this sequence of operations takes $O(n + 3m + nlogn)$ time, thus the complexity of the algorithm is $O(m + nlogn)$.

Bibliography

[1] Thomas H. Cormen, Charles E. Leiserson, and Ronald L. Rivest. *Introduction to Algorithms.* MIT Press, Cambridge Massachusetts, 1990.

[2] Alan Gibbons. *Algorithmic Graph Theory.* Cambidge University Press, Cambridge, 1985.