Chapter 2

Searches in graphs and digraphs

2.1 Searches and connectivity in graphs

Finding the connected component of a vertex v in a graph is not difficult. It suffices to compute a spanning tree of this component starting from v. For this purpose, we need a list of edges to be explored, initially containing all edges, and a list of the explored vertices, initially containing only v. At each step, a new edge ab is explored with a already explored and b is added in the list of explored vertices if it had not been explored yet. When all edges have been explored, we get a spanning tree of the connected component of v.

Given a graph G and a vertex v the following algorithm return a spanning tree T of the connected component of G.

Algorithm 2.1 (Search)**.**

```
1. Mark $v$ and initialize $L$ to the set of all $\{v,u\} \in E$ , $V(T) := \{v\}$, $E(T) := \emptyset$.
2. If $L = \emptyset$, then return $T$; else let $ab \in L$. $L := L \setminus \{ab\}$.
3. If $b$ is not marked, then mark it; $V(T) := V(T) \cup \{b\}$; $E(T) := E(T) \cup \{a,b\}$; add all the
   elements of $\{bu \mid u \in E(G)\}$ to $L$.
4. Goto 2.
```

There are two well-known searches which correspond to two different orderings of the edges:

- the breadth-first search (BFS) (Algorithm 2.2) explores first the neighbours of v, then the neighbours of its children;
- the depth-first search (DFS) (Algorithm 2.3) explores first all the vertices of a branch pending in v.

The difference between these two approaches is that the vertices are stored either in a queue (FIFO) or in a stack (LIFO). A *queue* is just a list which is updated by either adding a new element to one end (its *tail*) or removing an element from the other end (its *head*). A *stack* is simply a list, one end of which is identified as its *top*; it may be updated either by adding a new element as its top or else by removing its top element.

The following algorithms also compute the connected component C of v and a spanning tree of C.

Algorithm 2.2 (Breadth-First Search)**.**

```
1. Mark $v$, $V(T) := \{v\}$, $E(T) := \emptyset$ and initialize a queue $Q$ to $v$.
2. If $F = \emptyset$ then return $T$. Else, remove the first vertex $u$ of $Q$.
3. For every unmarked vertex $w$ adjacent to $u$, $V(T) := V(T) \cup \{w\}$; $E(T) = E(T) \cup \{uw\}$;
   add $w$ to $Q$ and mark $w$.
4. Go to 2.
```

Algorithm 2.3 (Depth-First Search)**.**

```
0. For every vertex, $L(u) := N(u)$.
1. Mark $v$; $V(T) := \{v\}$; $E(T) := \emptyset$ and initialize a stack $P$ to $v$.
2. If $P = \emptyset$, then return $T$. Else, let $u$ be the top of the stack.
3. If $L(u) = \emptyset$, then remove $u$ from $P$ and go to 2.
4. Else, remove a vertex $w$ from $L(u)$.
5. If $w$ is marked go to 3. Else, $V(T) := V(T) \cup \{w\}$; $E(T) = E(T) \cup \{uw\}$; add $w$ to $P$ and
   mark $w$.
6. Go to 2.
```

A tree obtained by running a breadth-first search is called a *breadth-first search tree* or *BFS-tree*. Similarly, a tree obtained by running a depth-first search is called a *depth-first search tree* or *DFS-tree*. If the search is run from vertex v, this vertex is called the *root* of the search tree.

Observe that in Algorithms 2.1, 2.2 and 2.3 every edge is examined at most twice (once per endvertex). These algorithms can be modified in order to compute all the connected components of a graph in time so that every edge is examined at most twice. For this purpose, while some vertex does not belong to a connected component (i.e., has not been marked), it is sufficient to compute its connected component.

2.1.1 Distance in graphs

A nice property of a breadth-first search tree is that is can give the distance from the root r to all other vertices. Therefore we just have to have a value $l(u)$ to every vertex called *level* of u which corresponds to $dist_T(r,u)$ and as we will show later also to $dist_G(r,u)$. Hence the following algorithm is the following:

Algorithm 2.4 (Distance from r)**.**

```
1. Mark $r$, $l(r) := 0$ and initialize a queue $Q$ to $v$.     [[ $V(T) := \{v\}$; $E(T) := \emptyset$; ]]
2. If $F = \emptyset$ then return $l$. Else, remove the first vertex $u$ of $Q$.
3. For every unmarked vertex $w$ adjacent to $u$, $l(w) := l(u)+1$; add $w$ to $Q$ and mark $w$.
           [[$V(T) := V(T) \cup \{w\}$; $E(T) = E(T) \cup \{uw\}$; ]]
4. Go to 2.
```

Observe that the construction of the tree T (operation between brackets at Step 1 and 3) is practically useless. It just help us to show that the function l has the properties we announced. The first ones justifies our referring to l as the level function.

Theorem 2.1. *Let T be a BFS-tree of a connected graph G, with root r. Then:*

(i) for every vertex v of G, $l(v) = dist_T(r,v)$;

(ii) every edge of G joins vertices on the same or consecutive levels of T; that is

$$|l(u) - l(v)| \leq 1, \quad \textit{for all } uv \in E(G).$$

Proof. The proof of (i) is left to the reader in Exercise 2.1. To establish (ii), it suffices to prove that if $uv \in E(G)$ and $l(u) < l(v)$, then $l(u) = l(v) - 1$.

we first establish, by induction on $l(u)$, that if u and v are any two vertices such that $l(u) < l(v)$, then u joined Q before v. This evident if $l(u) = 0$, because u is then the root r of T. Suppose that the assertion is true whenever $l(u) < k$, and consider the case $l(u) = k$, where $k > 0$. Let x be the predecessor of u, that is the vertex which is explored when we add u to Q. Then it follows from line 3 of Algorithm 2.4 that $l(x) = l(u) - 1$. Similarly, if y is the predecessor of v then $l(y) = l(v) - 1$. By induction, x joined Q before y. Therefore u being a neighbour of x, joined Q before v.

Now suppose that $uv \in E(G)$ and $l(u) < l(v)$. If u is the predecessor of v, then $l(u) = l(v) - 1$, again by line 3 of Algorithm 2.4. If not, let y be the predecessor of v. Because v was added to T by the edge yv, and not the edge uv, the vertex y joined Q before u, hence $l(y) \leq l(u)$ by the claim established above. Therefore $l(v) - 1 = ly) \leq l(u) \leq l(v) \leq l(v) - 1$, which implies $l(u) = l(v) - 1$. $\square$

The following theorem shows that Algorithm 2.4 runs correctly.

Theorem 2.2. *Let T be a BFS-tree of a connected graph G, with root r. Then:*

$$dist_T(r,v) = dist_G(r,v), \quad \text{for all } v \in V(G).$$

Proof. Clearly, $dist_T(r,v) \geq dist_G(r,v)$ because T is a subgraph of G.

Let us establish the opposite inequality by induction on the length of a shortest (r,v)-path, the proposition holding trivially when the length is 0.

Let P be a shortest (r,v)-path in G, where $v \neq r$, and let u be the predecessor of v on P. The (r,u)-subpath of P is a shortest (r,u)-path, and $d_G(r,u) = d_G(r,v) - 1$. By induction, $l(u) \leq d_G(r,u)$, and by Theorem 2.1-(ii), $l(v) = l(u) \leq 1$. Therefore

$$dist_T(r,v) = l(v) \leq l(u) + 1 \leq d_G(r,u) + 1 = d_G(r,v).$$

□

2.2 Searches and strong connectivity in directed graphs

One can explore digraphs in much the same way as graphs, but by growing arborescences rather than (rooted trees). An *arboresence* is an orientation of a rooted tree in which all the arcs are directed from the root to the leaves. It can be seen as a digraph in which every vertex has indegree 1 except one called the root which has indegree 0. As with search in graph, search in digraph may be refined by restricting the choice of the arc to be added at each stage. In this way, we obtain directed versions of breadth-first search and depth-first search. We now discuss how search can be applied to find the strongly connected components of a digraph.

To test if a digraph $D = (V,E)$ is strongly connected, one has to check for every pair u, v of vertices if there is a (u,v)-dipath. Checking if such a path exists can be done by performing a search, so running $\binom{|V(D)|}{2}$ searches will do the job. However running a search from a vertex u finds all the vertices v that can be reached from u. So, in fact, one just need to run at most $|V|$ searches (one per vertex) yielding a total time $O(|V||E|)$.

The following proposition will yield an algorithm that test if a digraph is strong by running only two searches.

Proposition 2.3. *Let D be a digraph and v a vertex of D. D is strongly connected if and only if, for every $u \neq v$, there are a (u,v)-path and a (v,u)-path.*

Proof. If D is strongly connected, by definition, for every $u \neq v$, there are a (u,v)-path and a (v,u)-path.

Let us assume now that for every $u \neq v$, there are a (u,v)-path and a (v,u)-path. Let us show that D is strongly connected. Let u and w be two distinct vertices of D. There are a (u,v)-path P and a (v,w) path Q the concatenation of which is a (u,w)-walk. By Proposition 1.3, there is a (u,w)-path. □

We now describe an algorithm that computes the strongly connected components of a vertex v of a digraph. It is based on two searches starting from v, the first one in D and the reverse $\bar{D}$ of D. During the first search, the vertices u reachable from v are marked 1. During the second search the vertices u from which v can be reached marked 2 and included in the strongly connected component of v if they are already marked 1 (See Figure 2.1).

Algorithm 2.5 (Strongly connected component)**.**

1. Search D starting from v marking the vertices with 1.
2. Search $\bar{D}$ starting from v marking the vertices with 2.
3. Return the vertices marked with 1 and 2.

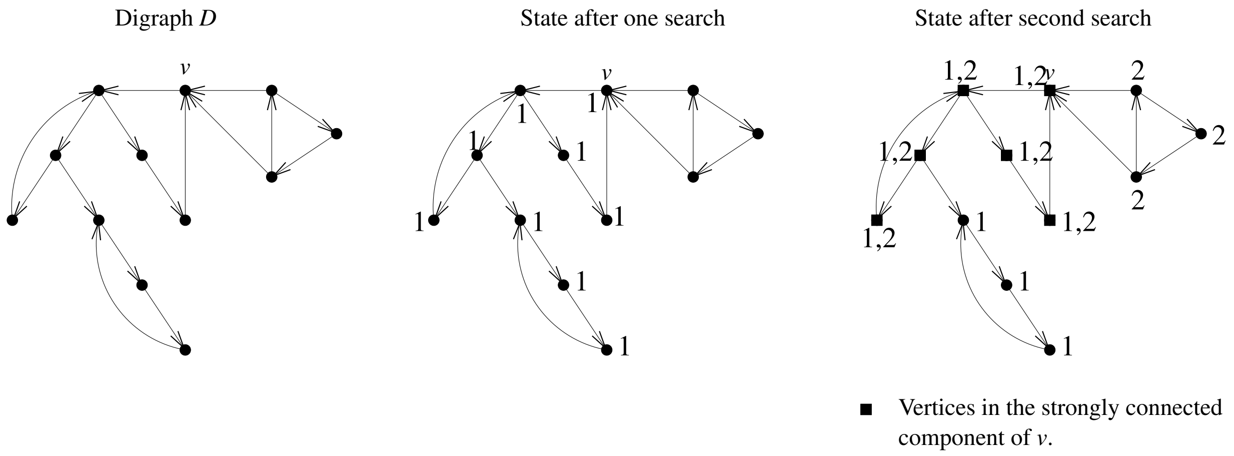
Digraph D
v
State after one search
v
1
1
1
1
1
1
1
1
1
State after second search
1,2
1,2
v
2
1,2
1,2
2
2
1,2
1,2
1
1
1
■ Vertices in the strongly connected component of v.

Figure 2.1: Execution of Algorithm 2.5

Contrary to Algorithm 2.1, Algoritm 2.5 does not give all strongly connected components of D in a single search examining each edges twice. Indeed, let us consider the digraph D with $V(D) = \{v_1, v_2, \ldots, v_n\}$ and$E(D) = \{(v_i, v_j) \mid i < j\}$. The components consist of each $\{v_i\}$. Hence, $|V|$ executions of Algoritm 2.5 must be done. Moreover, at each execution, all edges are considered.

2.2.1 Computing strongly connected components in one search

We now describe an algorithm that computes all strongly connected components of a digraph in time $O(|E|)$. It is a modified depth-first search in which two extra values are stored and updated. When a vertex is explored u it becomes *active* and is associated to two values $l(u)$ and $b(u)$. The first one $l(u)$ called *label* of u corresponds to the order of appearance of u during the search. It will never change. A vertex w such that $l(w) \geq l(u)$ is called a *successor* of u. A key ingredient of the algorithm is that as long as u is active, there is a (u,w)-path to each of its successors w. The second value $b(u)$ corresponds to the smallest label of a vertex reachable from u.

Algorithm 2.6 (All Strongly Connected Components)**.**

```
0. Initialize $i$ to 0.
1. If all vertices are marked, then terminate.
2. Let $s$ be an unmarked vertex.
3. $i := i+1$; initialize $l(s)$, $b(s)$ to $i$ and $u$ to $s$. $s$ becomes active.
4. If at least one arc leaving $u$, say $(u,v)$, is not marked, then do
    4.1 Mark $(u,v)$.
    4.2 If $v$ has already been explored and is active, then update $b(u)$ : $b(u) := \min(b(u),b(v))$.
    4.3 Else $v$ is a new vertex. $v$ becomes active; $i := i+1$; $l(v) := i$; $b(v) := l(v)$; $u := v$.
    4.4 Goto 4.
5. Else, all arcs leaving $u$ are marked, the exploration of $u$ is over:
    5.1 If $b(u) = l(u)$ then all active successors of $u$ induce a strongly connected component : return it and all its vertices become inactive; Goto 1.
    5.2 Else $b(u) < l(u)$. Let $w$ be the vertex from which $u$ has been explored. Update $b(w)$ : $b(w) := \min(b(w),b(u))$; $u := w$; Goto 4.
```

Correctness of Algorithm 2.6: We will show by induction the following three points.

1) If $l(u) < l(v)$, and if u and v are active, then v is a successor of u;

2) At every step, for any active vertex v, there is a path from v to the vertex w with label $l(w) = b(v)$.

3) When the exploration of u terminates (Step 5), all the active vertices of $S(u) = \{v \text{ active } \mid b(u) \leq l(v) \leq l(u)\}$ are in the same strongly connected component as u.

4) $b(u) = l(u)$ if and only if $S(u)$ is a strongly connected component.

1) and 2) Let to the reader.

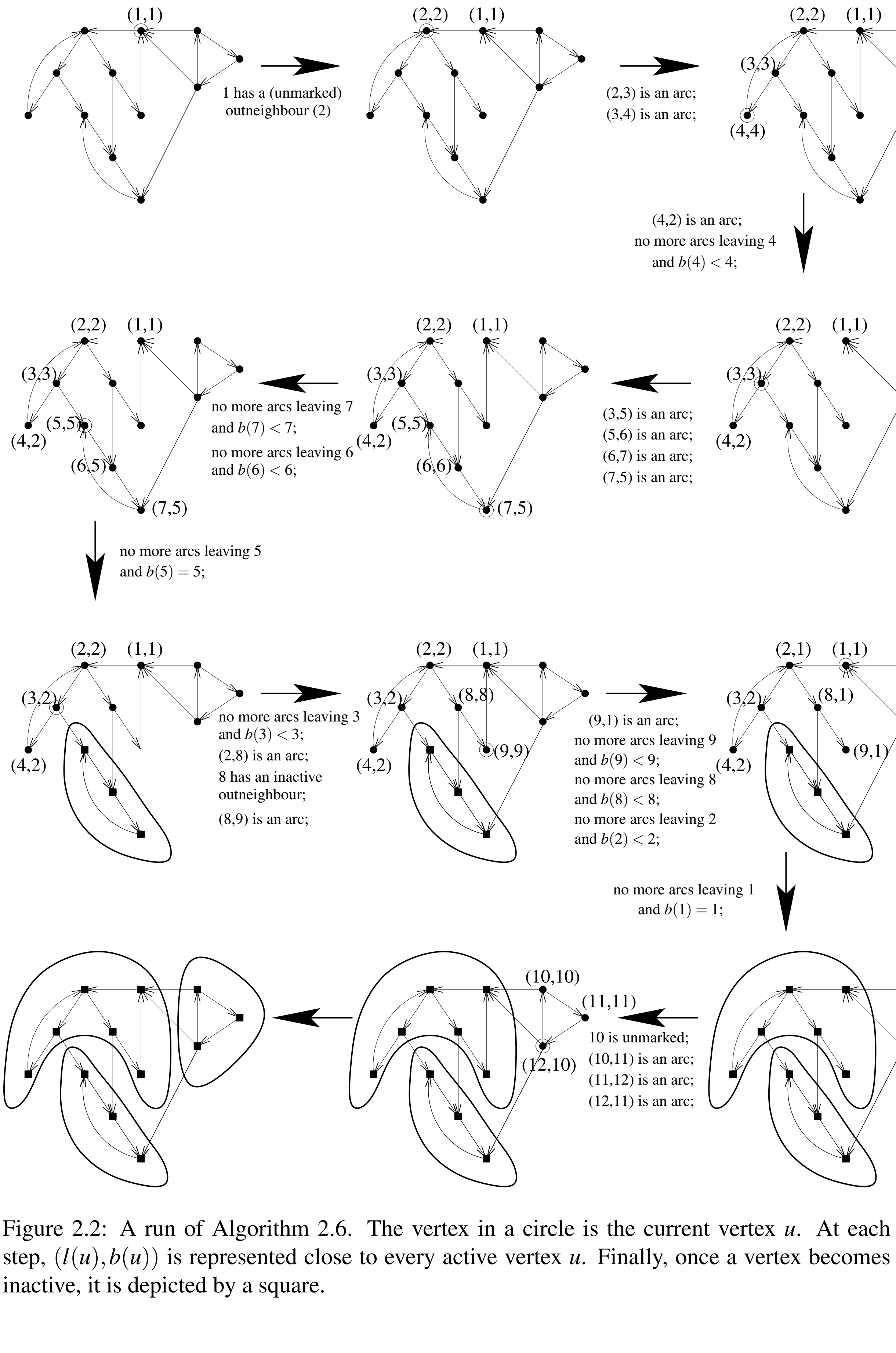

Figure 2.2: A run of Algorithm 2.6. The vertex in a circle is the current vertex u. At each step, $(l(u), b(u))$ is represented close to every active vertex u. Finally, once a vertex becomes inactive, it is depicted by a square.

3) From Proposition 2.3, it is sufficient to show that, for every v in $S(u)$, there is a (u,v)-path and a (v,u)-path. Let v be a vertex in $S(u)$.
Let us assume first that $l(v) < l(u)$. Then, by 1), there is a (v,u)-path. let w be the vertex such that $l(w) = b(u)$. By 2), there is a (u,w)-path, and by 1) there is a (w,v)-path. The concatenation of both these paths is a (u,v)-walk. By Proposition 1.3 (i), there is a (u,v)-path.
Let us now assume that $l(v) > l(u)$. By 1) there is a (u,v)-path. Moreover, $b(v) < l(v)$, otherwise, the vertex v would have become inactive at Step 5. Let v_1 be the vertex with label $l(v_1) = b(v)$. By 2), there is a (v,v_1)-path P_1. If $l(v_1) \leq l(u)$, then, by 1), there is a (v_1,u)-path, the concatenation of which with P_1 is a (v,u)-walk. Hence, a (v,u)-path exists from Proposition 1.3 (i). If $l(v_1) > l(u)$, then $b(v_1) < l(v_1)$, for otherwise, the vertex v_1 (and also v) would have become inactive at Step 5. Let v_2 be the vertex such that $l(v_2) = b(v_1)$. Using similar arguments, while $l(v_i) > l(u)$, we have $l(v_i) > b(v_i)$ and we set v_{i+1} such that $l(v_{i+1)} = b(v_i)$. Since the label of v_i strictly decreases, the sequence of the v_is is finite. Hence, $v = v_0, v_1, \dots, v_k$ et $l(v_k) \leq l(u)$. Using 2), for every $1 \leq i \leq k$, there is a (v_{i-1}, v_i)-path. Moreover, by 1), there is a (v_k,u)-path. The concatenation of all these paths is a (v,u)-walk, and then, by Proposition 1.3 (i), there is a (u,v)-path.

4) Now, if $b(u) = l(u)$, the arcs leaving $S(u)$ have their heads inactive (vertices in a distinct strongly connected component, by the induction hypothesis 3). Hence, $S(u)$ is a strongly connected component.
If $b(u) < l(u)$, the vertex labelled $b(u)$ is active. So it is a predecessor of u. Hence, u and v are in the same component.

2.3 Bipartite graphs

A *bipartition* of a graph G is a partition (A,B) of $V(G)$ into two stable sets. Hence, every edge of G has an endvertex in A and the other in B. We often write $G = ((A,B),E)$ for a bipartite graph with bipartition (A,B).

Bipartite graphs satisfy some properties.

Proposition 2.4. *Let $G = ((A,B),E)$ be a bipartite graph.*

(i) for any two vertices u and v, all the (u,v)-walks have the same length parity.

(ii) if G is connected then it has only two bipartitions (A,B) and (B,A).

Proof. (i) Without loss of generality, we may assume that u is in A. Let $(v_0, v_1, \dots, v_k)$ be a (u,v)-walk (so $v_0 = u$ and $v_k = v$). Since G is bipartite and $V_0 \in A$ then $v_1 \in B$ and so $v_2 \in A$. And so on by induction, if i is even then $v_i \in A$ and if i is odd $v_i \in B$. Thus if $v \in A$ then k is even and if $v \in B$ k is odd.

(ii) Let u be a vertex. Let A_0 (resp. A_1) be the set of vertices at even (resp. odd) distance to v_0 in G. By (i), in any bipartition of G, A_0 is included in the part containing u and A_1 in the other. $A_0 \cup A_1 = V(G)$ since the graph is connected then there are only two possible partitions of G: (A_0,A_1) and (A_1,A_0). □

There are graphs which are not bipartite. For example, the odd cycles. Indeed in the cycle $(v_0, v_1, \dots, v_{2k}, v_0)$ the path (v_0, v_{2k}) and $(v_0, v_1, \dots, v_{2k})$ are two (v_0, v_{2k})-paths of different length parity. Hence if a graph is bipartite, it contains no odd cycles. This easy necessary condition to be bipartite is in fact sufficient.

Theorem 2.5. *A graph G is bipartite if and only if it has no odd cycle.*

Proof. Clearly, it suffices to prove it for connected graphs. Let G be a connected graph. If G contains an odd cycle, it is not bipartite.

Conversely, assume that G contains no odd cycle. Let v_0 be a vertex of G. Let A_0 (resp. A_1) be the set of vertices at even (resp. odd) distance to v_0 in G. Let us now show that (A_0, A_1) is a bipartiton of G. Let uv be an edge of G and P_u (resp. P_v) be a shortest (u, v_0)-path (resp. (v, v_0)-path). The concatenation P_u, P_v and (v, u) is a closed walk. By Proposition 1.3, this walk has even length otherwise G would contain an odd cycle. Hence P_u and P_v have different length and so uv has an endvertex in each of the A_i, $i = 0, 1$. □

The above proof may be translated into an algorithm which, given a connected graph G, returns either a bipartition if G is bipartite or "G is not bipartite" otherwise. Basicallly, it runs a Breadth-First Search from a vertex and check if there is no edge between vertices of levels of differents parity. Hence instead of marking the vertices with their level number as for the distance (see Subsection 2.1.1), we mark them with the parity of their level and thus we just need two marks, 0 and 1.

Algorithm 2.1 (Finding a bipartition)**.**

```
1. Pick a vertex $x$ and mark it with $m(x) := 0$; $N := \{x\}$.
2. If $N$ is non-empty, then remove a vertex $v$ of $N$ and do the following.
        For all neighbour $w$ of $v$ do
          - If $m(w) = m(v)$, return "$G$ is not bipartite";
          - Otherwise if $w$ is unmarked, mark it with $m(v) + 1$ mod 2 and put $w$ in $N$; Go to 2.
3. If $N$ is empty, let $A_i, i = 0, 1$ be the set of vertices marked $i$ and return "$G$ is bipartite with bipartition" $(A_0, A_1)$.
```

Algorithm 2.1 may be easily modified to return an odd cycle when G is not bipartite. See Exercise 2.10.

2.4 Exercises

Exercise 2.1. Show Theorem 2.1-(i).

Exercise 2.2 (Entriger, Kleitman and Székely)**.**
For a connected graph G, define $\sigma(G) = \sum_{u,v \in V(G)} dist(u,v)$.

1) Let G be a connected graph. For $v \in V(G)$, let T_v be a BFS-tree of G rooted at v. Show that $\sum_{v \in V(G)} \sigma(T_v) = 2(n-1)\sigma(G)$.

2) Deduce that every connected graph G has a spanning tree T such that $\sigma(T) \leq 2(1 - \frac{1}{n})\sigma(G)$.

Exercise 2.3 (Tuza)**.** Let G be a connected graph, let x be a vertex of G, and let T be a spanning tree of G that maximizes the function $\sum_{v \in V(G)} dist_T(x,v)$. Show that T is a DFS-tree of G.

Exercise 2.4 (Chartrand and Kronk)**.** Let G be a connected graph in which every DFS-tree is a path (rooted at the start). Show that G is a cycle, a complete graph, or a complete bipartite graph in which both parts have the same number of vertices.

Exercise 2.5 (Pósa)**.** A *chord* of a cycle C in a graph G is an edge in $E(G) \setminus E(C)$ both of whose endvertices lie on C. Let G be a graph such that $|E(G)| \geq 2|V(G)| - 3$ and $|V(G)| \geq 4$. Show that G contains a cycle with at least one chord.

Exercise 2.6. Let a be a connected graph G. Prove that G is bipartite if and only if $dist(a,b) \neq dist(a,c)$ for all edge bc.

Exercise 2.7.
1) Show that every tree is bipartite.
2) Prove that every tree has a leaf in the largest part of its bipartition.

Exercise 2.8. Prove that a bipartite graph G has at most $|V(G)|^2/4$ edges and give a graph attaining this bound.

Exercise 2.9. Show that a graph is bipartite if and only if each of its subgraph H has a stable set of size at least $|V(H)|/2$.

Exercise 2.10. Give an algorithm that, given a connected graph G, returns either a bipartition if G is bipartite or an odd cycle if G is non-bipartite.

Exercise 2.11. Describe an algorithm based on a breadth-first search for finding a shortest odd cycle in a graph.

Exercise 2.12. Let $G = ((A,B),E)$ be a bipartite graph without isolated vertices such that $d(x) \geq d(b)$ for all $xy \in E$, where $a \in A$ and $b \in B$. Prove that $|A| \leq |B|$, with equality if and only if $d(a) = d(b)$ for all $ab \in E$.