Section 6.1. Tree-Search

Note. In Theorem 4.6 we saw that a graph is connected if and only if it has a spanning tree. The proof of this is based on removing edges from cycles in the graph until the resulting subgraph is acyclic. One might argue that this is an algorithm for finding a spanning subtree of a connected graph, but it does not tell us how to find cycles in the graph. In this section we give two algorithms to find a spanning tree in a given connected graph.

Note. Recall that the edge cut (or coboundary) of a set of vertices X in a graph in the set of edges in the graph with one end in X and one end in $V \setminus X$, denoted $\partial(X)$ (see Section 2.5. Edge Cuts and Bonds). In this section, when F is a subgraph of G we denote $\partial(V(F))$ simply as $\partial(F)$ and call it the *edge cut* associated with F.

Note 6.1.A. For T a tree which is a subgraph of graph G, we have $V(T) \subseteq V(G)$. If $V(T) = V(G)$ then T is a spanning tree of G, and we have by Theorem 4.6 that G is connected. Not suppose $V(T) \subset V(G)$ (that is, $V(t) \subsetneq V(G)$) then either $\partial(T) = \varnothing$ or $\partial(T) \neq \varnothing$. If $\partial(T) = \varnothing$ then G is disconnected since there are vertices in $V(G) \setminus V(T)$ are <u>not</u> adjacent to the vertices in $V(T)$. If $\partial(T) \neq \varnothing$ then there is $x \in V(T)$ and $y \in V(G) \setminus V(T)$ such that $xy \in \partial(T)$. In this case we can add vertex y and edge xy to T to create a new tree in G that is larger than T. See Figure 6.1. This is the basic idea in "growing" a spanning tree of a graph G.

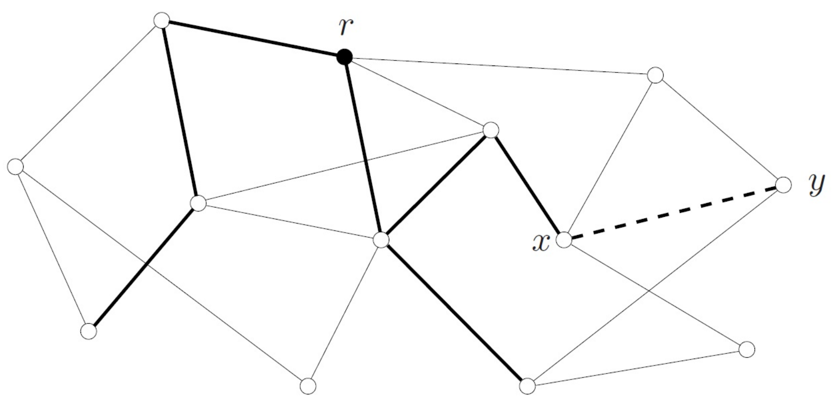

Fig. 6.1. Growing a tree in a graph

Note/Definition. Recall from Section 4.1. Forests and Trees that a *rooted tree* is a tree T with a specified vertex x called the *root* of the tree; such a tree is denoted $T(x)$. Using the technique described in Note 6.1.A, we can start with a vertex r of graph G, declaring it a root, and then construct a sequence of rooted trees, each with root r, ending with either spanning tree with root r or ending with a nonspanning tree with root r (this second case occurring when graph G is not connected). Such a procedure is a *tree-search* and the final tree is a *search tree.*

Note. We present two tree-search algorithms. The Breadth-First Search allows us to create a rooted tree in which the distance of each vertex from the root in the tree is the same as the distance from the root in the graph G. The Depth-First Search allows us to find cut vertices in graph G. First, we introduce some new terminology.

Definition. Let T be a tree with root r. The *level* of a vertex v in T is the length of hte path rTv. Additional terms are inspired by genealogy. Each vertex on the path rTv (including v itself) is an *ancestor* of v. Each vertex of which v is an ancestor (which is determined by treating other vertices on rTv as roots of T) is a *descendant* of v. An ancestor or descendant of a vertex is *proper* if it is not the vertex itself. Two vertices are *related* in T if one is an ancestor of the other. The immediate proper ancestor of a vertex (where the immediacy is determined by the level of the vertices), other than root r, is the vertex's *predecessor* or *parent*, denoted $p(v)$; the vertices whose immediate predecessor is v are its *successors* or *children*.

Note. Notice that we can orient the edges of rooted tree $T(r)$ using the predecessor function p as

$$E(T(r)) = \{(p(v), v) \mid v \in V(T(r)) \setminus \{r\}\}.$$

The rooted tree is then determined by its vertex set and predecessor function p.

Note. In the rest of this chapter, we assume that all graphs and digraphs are connected and loopless.

Note/Definition. In the Breadth-First Search algorithm, we introduce an ordered list Q, called the *queue*. One end of the queue is the *head* and the other is the *tail*. As the algorithm runs, the queue Q contains all vertices from which the current tree could potentially be grown. The algorithm starts (at $t = 0$) with $Q = \varnothing$. As the spanning rooted tree $T(r)$ is created, a vertex added to $T(r)$ is added to Q. At each step in the creation of $T(r)$, the adjacency list of the vertex at the head of A is searched for a neighbor to add to $T(r)$. If every neighbor is already in the tree, then the vertex at the head of Q is removed from Q. The new head of Q is then processed and the algorithm terminates when Q is again empty. The Breadth-First Search algorithm outputs the predecessor function (which determines the spanning rooted tree $T(r)$) and a function $\ell : V \to \mathbb{N}$ which gives the level of each vertex in the tree (and hence the distance of each vertex from the root r in graph G). We can also keep track of when (that is, the value of parameter t) each vertex was added to $T(r)$ with function $t : V \to \mathbb{N}$. When we illustrate the algorithm, vertices in $T(r)$ are coloured black (i.e., represented by a solid disk). We now state the Breadth-First Search (BFS) algorithm.

Algorithm 6.1. BREADTH-FIRST SEARCH (BFS).

INPUT: a connected graph $G(r)$

OUTPUT: an r-tree T in G with predecessor function p, a level function ℓ such that $\ell(v) = d_G(r, v)$ for all $v \in V$, and a time function t

```
1. set $i := 0$ and $Q := \varnothing$
2. increment $i$ by 1
3. colour $r$ black
4. set $\ell(r) := 0$ and $t(r) := i$
5. append $r$ to $Q$
6. while $Q$ is nonempty do
7.     consider the head $x$ of $Q$
```

```
8.        **if** $x$ has an uncoloured neighbour $y$ then
9.            increment $i$ by 1
10.           colour $y$ black
11.           set $p(y) := x$, $\ell(y) := \ell(x) + 1$ and $t(y) := i$
12.           append $y$ to $Q$
13.       **else**
14.           remove $x$ from $Q$
15.       **end if**
16. **end while**
17. return $(p, \ell, t)$.
```

Definition. The spanning r-tree T returned by the BFS algorithm is a *breadth-first search tree*, or *BFS-tree*, of G.

Example 6.1.A. We now apply the BFS algorithm to the graph of Figure 6.2. The labels of the vertices in Figure 6.2(a) represent the times at which they were added to the tree. The distance function (that is, the "level" of a vertex) is given in Figure 6.2(b).

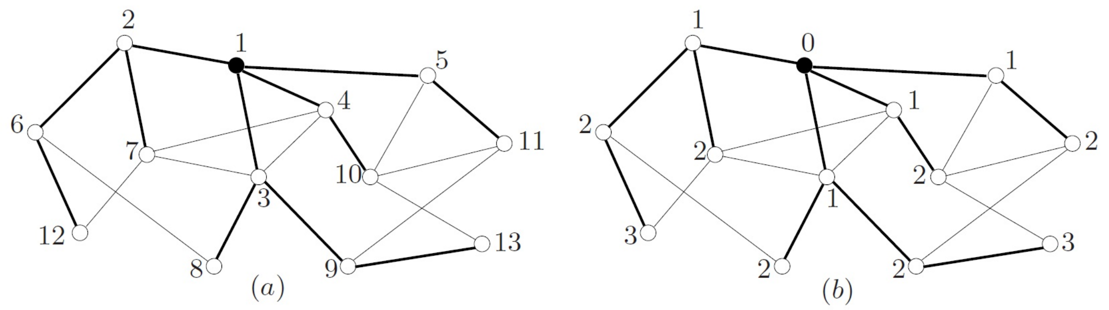

Fig. 6.2. A breadth-first search tree in a connected graph: (a) the time function t, and (b) the level function ℓ

The queue Q evolves as follows. Each time a vertex is added to Q (as the head) we give the new elements of Q in blue. Each time a vertex is removed from (the tail of) Q we give the new elements of Q in red. Notice that graph G has 13 vertices. Once BFS starts we add a vertex to Q 13 times and remove a vertex from Q 13 times. This is reflected in the 13 red versions of Q and the 13 blue versions of Q. This requires a total of 26 steps, represented by the 26 arrows.

LEVEL	STRUCTURE OF Q
0	$Q = \varnothing \to 1$
1	$\to 1\ 2 \to 1\ 2\ 3 \to 1\ 2\ 3\ 4 \to 1\ 2\ 3\ 4\ 5 \to 2\ 3\ 4\ 5$
2	$\to 2\ 3\ 4\ 5\ 6 \to 2\ 3\ 4\ 5\ 6\ 7 \to 3\ 4\ 5\ 6\ 7 \to 3\ 4\ 5\ 6\ 7\ 8$
	$\to 3\ 4\ 5\ 6\ 7\ 8\ 9 \to 4\ 5\ 6\ 7\ 8\ 9 \to 4\ 5\ 6\ 7\ 8\ 9\ 10$
	$\to 5\ 6\ 7\ 8\ 9\ 10 \to 5\ 6\ 7\ 8\ 9\ 10\ 11 \to 6\ 7\ 8\ 9\ 10\ 11$
3	$\to 6\ 7\ 8\ 9\ 10\ 11\ 12 \to 7\ 8\ 9\ 10\ 11\ 12 \to 8\ 9\ 10\ 11\ 12$
	$\to 9\ 10\ 11\ 12 \to 9\ 10\ 11\ 12\ 13 \to 10\ 11\ 12\ 13$
	$\to 11\ 12\ 13 \to 12\ 13 \to 13 \to \varnothing = Q.$

Theorem 6.2. Let T be a BFS-tree of a connected graph G, with root r. Then:

(a) for every vertex v of G, $\ell(v) = d_T(r, v)$, the level of v in T, and

(b) every edge of G joins vertices on the same or consecutive levels of T; that is, $|\ell(u) - \ell(v)| \leq 1$ for all $uv \in E$.

Note. In Theorem 6.2(a), we saw that the level function is the same as the distance from the root in the r-tree: $\ell(v) = d_T(r, v)$. Next we show that this also equals the distance from root r to v in the original graph G.

Theorem 6.3. Let G be a connected graph. Then the values of the level function ℓ returned by BFS are the distances in G from the root r: $\ell(v) = d_G(r, v)$ for all $v \in V$.

Note. Exercise 6.1.2 outlines an alternative inductive proof of Theorems 6.2 and 6.3.

Note. Next, we consider the "Depth-First Search" algorithm. In this algorithm, new vertices are added to the spanning tree by connections to the most recently added vertices. Informally, you might think of the Breadth-First Search as "scooping up" all nearby neighbors of vertices already in T. Depth-First runs along a path until it can be extended no more, then one backtracks along the existing tree until new vertices can be added and a new path constructed. So Breadth-First scoops widely to add new vertices (notice the "wide" collections of vertices added at the different levels in Figure 6.2(b)), whereas Depth-First runs away from the root, backtracks, and runs again. Thus, the names of these two algorithms!

Note/Definition. In Depth-First, the vertices of T whose adjacency lists have yet to be fully scanned are stored in a stack. A *stack* is a list in which one end is designated as the *top*. As the algorithm runs, new vertices are either added to the top of the stack, or the existing top is removed. The stack is initially empty. When a new vertex is added to T, it is added to the top of S. The neighbors of the top vertex are scanned for possible inclusion in T. If all neighbors of the top vertex are already in T, then the top vertex is removed from the stack. This process is iterated until stack S is again empty and the algorithm ends.

Note. We have two functions mapping $V \to \mathbb{N}$ which we refer to as "times." The time $f(v)$ is when v is added to T (and so when v is added to stack S), and the time $l(v)$ when v is removed from S (because all neighbors of v are already in T). At time $l(v)$ the algorithm backtracks to the predecessor $p(v)$. The time functions increment by 1 each time the stack S changes. With n vertices in G, we have $f(r) = 1$ and $l(r) = 2n$ (since each vertex is added to the stack once and removed from the stack once). At each leaf v of T we have $l(v) = f(v) + 1$ (the leaf is added to T, it then has no neighbors not already in T, and then it is removed). We now state the Depth-First Search algorithm.

Algorithm 6.4. Depth-First Search (DFS).
Input: a connected graph G
Output: a rooted spanning tree of G with predecessor function p, and two time functions f and l

```
1. set $i := 0$ and $S := \varnothing$
2. hoose any vertex $r$ (as the root)
3. increment $i$ by 1
4. colour $r$ black
5. set $f(r) := i$
6. add root $r$ to $S$
```

```
 7. while $S$ is nonempty do
 8.     consider the top vertex $x$ of $S$
 9.     increment $i$ by 1
10.     if $x$ has an uncoloured neighbour $y$ then
11.         colour $y$ black
12.         set $p(y) := x$ and $f(y) = i$
13.         add $y$ to the top of $S$
14.     else
15.         set $l(x) := i$
16.         remove $x$ from $S$
17.     end if
18. end while
19. return $(p, f, l)$.
```

Example 6.1.B. We now apply the DFS algorithm to the graph of Figure 6.2. The labels of the vertices in Figure 6.23(a) represent the ordered pair $(f(v), l(v))$ and represent the times at which the vertex was made to top of stack S (with $f(v)$) and the time at which the vertex was removed from the top of the stack (with $l(v)$). A different drawing of the tree, along with the values of f and l, is given in Figure 6.3(b) below.

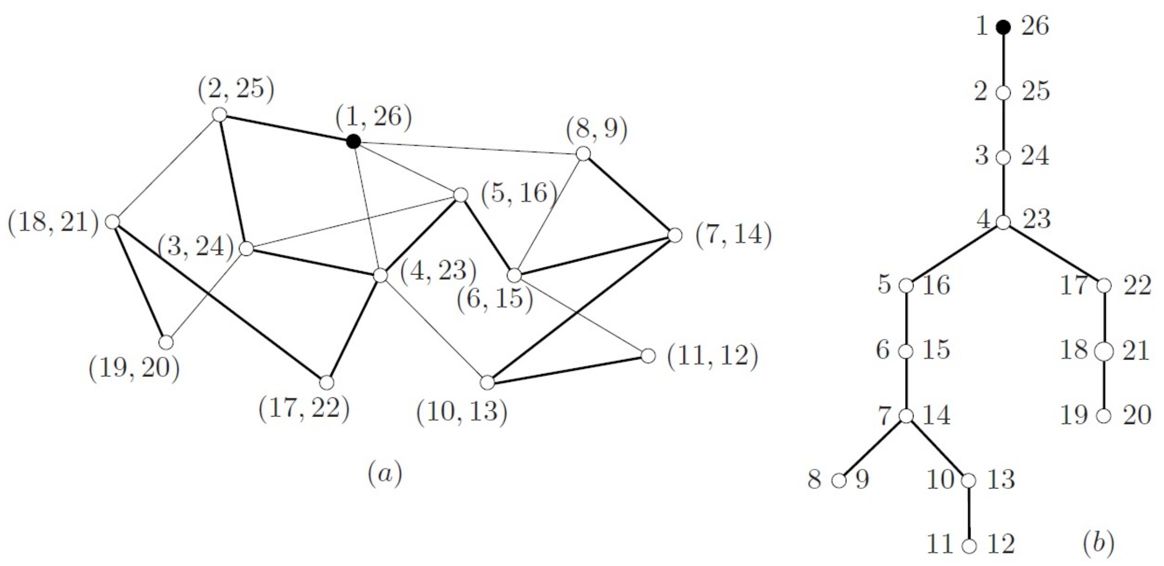

Fig. 6.3. (a) A depth-first search tree of a connected graph, and (b) another drawing of this tree

The stack S evolves as follows. The value of the time functions f or l is given over each arrow. When a vertex is added to S, the time and arrow are blue and when a vertex is removed from S the time and arrow are red. Notice that there are 13 blue arrows and 13 red arrows.

$$\varnothing \xrightarrow{1} 1 \xrightarrow{2} 1\,2 \xrightarrow{3} 1\,2\,3 \xrightarrow{4} 1\,2\,3\,4 \xrightarrow{5} 1\,2\,3\,4\,5 \xrightarrow{6} 1\,2\,3\,4\,5\,6 \xrightarrow{7} 1\,2\,3\,4\,5\,6\,7$$

$$\xrightarrow{8} 1\,2\,3\,4\,5\,6\,7\,8 \xrightarrow{9} 1\,2\,3\,4\,5\,6\,7 \xrightarrow{10} 1\,2\,3\,4\,5\,6\,7\,10 \xrightarrow{11} 1\,2\,3\,4\,5\,6\,7\,10\,11$$

$$\xrightarrow{12} 1\,2\,3\,4\,5\,6\,7\,10 \xrightarrow{13} 1\,2\,3\,4\,5\,6\,7 \xrightarrow{14} 1\,2\,3\,4\,5\,6 \xrightarrow{15} 1\,2\,3\,4\,5 \xrightarrow{16} 1\,2\,3\,4 \xrightarrow{17} 1\,2\,3\,4\,17$$

$$\xrightarrow{18} 1\,2\,3\,4\,17\,18 \xrightarrow{19} 1\,2\,3\,4\,17\,18\,19 \xrightarrow{20} 1\,2\,3\,4\,17\,18 \xrightarrow{21} 1\,2\,3\,4\,17 \xrightarrow{22} 1\,2\,3\,4 \xrightarrow{23} 1\,2\,3 \xrightarrow{24} 1\,2 \xrightarrow{25} 1 \xrightarrow{26} \varnothing.$$

Proposition 6.5. Let u and v be two vertices of G, with $f(u) < f(v)$.

(a) If u and v are adjacent in G, then $l(u) < l(v)$.

(b) u is an ancestor of v in T if and only if $l(v) < l(u)$.

Note. Proposition 6.5 gives the Depth-First Search tree T and the time functions f and l. The next theorem gives the "quintessential property of DFS-trees" (Bondy and Murty page 142).

Proposition 6.6. Let T be a DFS-tree of a graph G. Then every edge of G joins vertices which are related in T.

Definition. Let T be a Depth-First Search tree T in graph G. Orient the edges of T from parent to child. For the nontree edges of G (whose ends are related in T, as shown in Theorem 6.6) from descendant to ancestor. These latter oriented edges are *back edges*.

Note. The next theorem allows us to find the cut vertices of a graph G using an oriented DFS-tree. An application of cut vertices involves graphs which represent communication networks. A cut vertex of such a graph corresponds to a "station" whose breakdown would disrupt communication for part of the network.

Theorem 6.7. Let T be a DFS-tree of a connected graph G. The root of T is a cut vertex of G if and only if it has at least two children. Any other vertex of T is a cut vertex of G if and only if it has a child no descendant of which dominates (by a directed back edge) a proper ancestor of the vertex.

Note. In Theorem 6.7, if the root has at least two children, then the DFS algorithm will build T in a way that includes all descendants of the first child, then backtrack all the way back to the root and next build the tree that includes all descendants of the second child, etc. So the root is a cut vertex in this case and the descendants of the multiple children represents the vertices of the components of G minus the root. In the case of the "other vertex of T," the vertex is in a cycle of G if and only if the described back edge exists. The cut vertices claim then follows from Theorem 5.1.

Note. We now use a DFS-tree T to find cut vertices and blocks of a connected graph G. Let B be a block of G; recall that a block is a maximal nonseparable subgraph (see Figures 5.3 and 5.4(a) in Section 5.2. Separation and Blocks for a quick illustration of the idea). By Exercise 5.2.8(b), $T \cap B$ is a tree in G. Since T is a rooted tree, then we can define a root of the subtree $T \cap B$ (the vertex for which all other vertices in $T \cap B$ are descendants; by Proposition 6.5(b) this is the vertex of maximal l value in $T \cap B$). Define this vertex as the *root of clock* B with respect to T. We then have that the cut vertices of G are the roots of blocks (except for root r if it is a root of a single block; that is, if it has only one child); we are considering loopless graphs in this chapter so the cut vertices and separating vertices are the same by Note 5.2.A. So finding the cut vertices and blocks of G, we just need to find the roots of the blocks.

Note. Next we define $f^* : V \to \mathbb{N}$ which we use to find these roots. For $v \in V$, if some proper ancestor of v can be reached from v along a directed path consisting of tree (oriented) edges (possibly as few as 0 such tree edges) followed by one back edge then define $f^*(c)$ as the least f value of such an ancestor. If no proper ancestor of v can be reached in this process then set $f^*(v) := f(v)$. Bondy and Murty claim (page 143) that v is a root of a block if and only if it has a child w such that $f^*(w) \geq f(v)$, though this is not immediately clear. However, we now illustrate this with an example.

Note. A DFS-tree of a graph G is given in Figure 6.4, along with the values of g and l. We refer to the vertices here by their f values. We orient the edges of G as needed in Figure 6.4′ below.

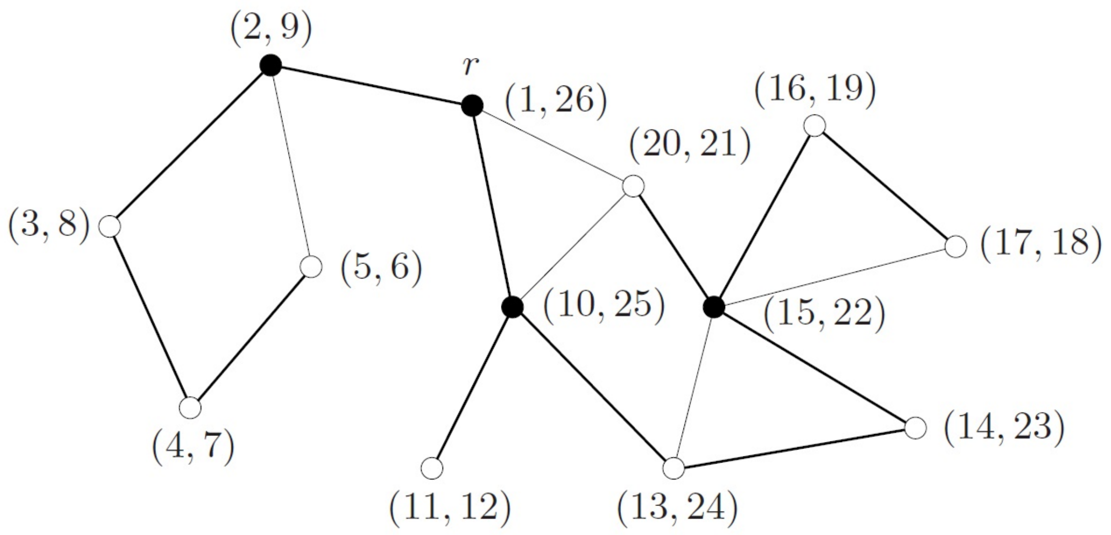

Fig. 6.4. Finding the cut vertices and blocks of a graph by depth-first search

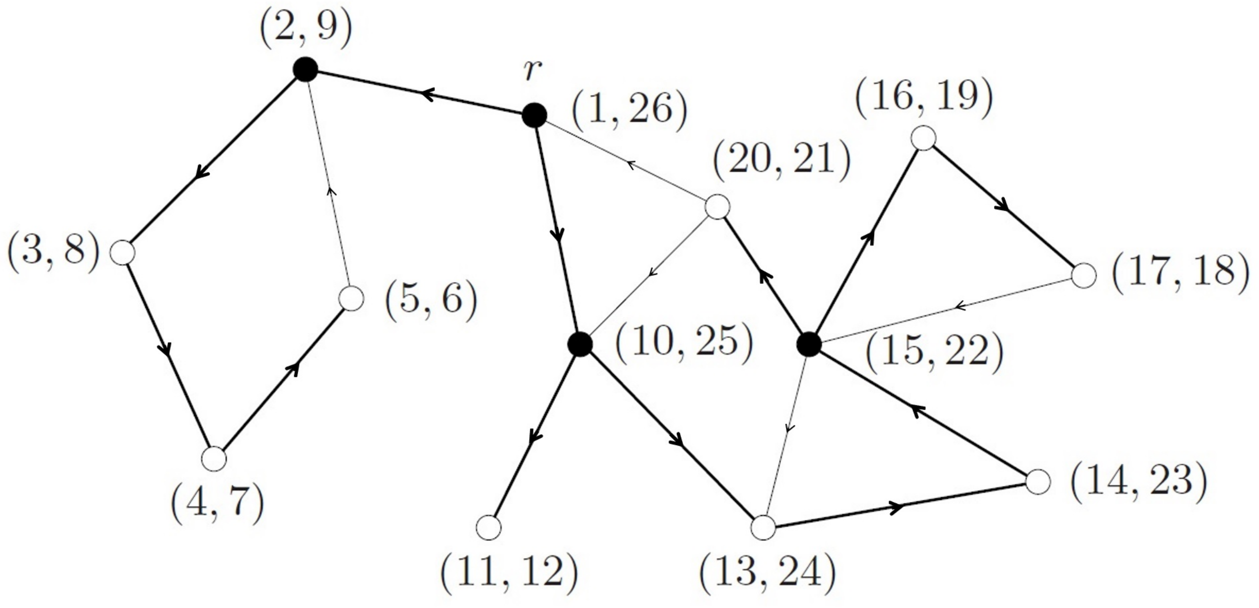

Figure 6.4′

For the root of T we have $f^*(f) = f(r) = f(1) = 1$ since the root $r = 1$ has no proper ancestor. For vertex 2 we have $f^*(2) = f(2) = 2$ since there is not directed path to the proper ancestor r. For vertex 3 there is a directed path as described from vertex 3 to ancestor vertex 2 so $f^*(3) = f(2) = 2$. Similarity for vertices 4 and 5 we have $f^*(4) = f(2) = 2$ and $f^*(5) = f(2) = 2$. For vertex 10 there is a directed path as described (the path $(10, 13, 14, 15, 20, 1)$) to the proper ancestor r so $f^*(1)) = f(r) = 1$, and similarly for vertices 13, 14, 15, and 20 we have $f^*(13) = f^*(14) = f^*(15) = f^*(20) = f^*(r) = 1$. Notice that for vertex 14 there is also a directed path as described to proper ancestor 13 for which $f(13) = 13$, but we take $f^*(14)$ as the lesser value $f(r) = 1$. There are no directed paths from vertex 11, so $f^*(11) = f(11) = 11$. For vertices 16 and 17 there are directed paths as described to proper ancestor 15, so $f^*(16) = f^*(17) = f(15) = 15$. So we have the following f^* values for the vertices v and their children w:

v	$f^*(v)$	w	$f^*(w)$
$r = 1$	1	2	2
		10	11
2	2	3	2
3	2	4	2
4	2	5	2
5	2	–	–
10	1	11	11
		13	1
11	11	–	–
13	1	14	1
14	1	15	1
15	1	16	15
		20	1
16	15	17	15
17	15	–	–
20	1	–	–

The event where $f^*(w) \geq f(v)$ is indicated in red fonts and so we see that the cut vertices are vertices $r = 1$, 2, 10, and 15, as indicated in Figure 6.4.

Revised: 6/11/2022