Part IV

Common Reduction Targets
Chapter 13

Depth-First Search

As we have seen in earlier chapters, many problems can be characterized as graph problems. Graph problems often require the searching of the graph for certain structural characteristics. For example, one problem which we will examine in this chapter is searching a connected undirected graph for vertices whose removal would disconnect the graph. Such vertices are called articulation points.

In order to find articulation points, an algorithm must extract a great deal of information involving the paths connecting the various vertices in the graph. One way of organizing this information is by constructing a certain kind of rooted spanning tree, called a depth-first spanning tree. One advantage to processing a rooted spanning tree as opposed to a graph is that a rooted tree fits more naturally with the top-down approach. Furthermore, as we will see later in this chapter, a depth-first spanning tree has properties that are helpful for extracting connectivity information. We therefore find that many problems can be reduced to finding one or more depth-first spanning trees. These spanning trees are found using a technique called depth-first search.

Because algorithms using depth-first search operate on rooted trees, we begin by studying the problem of determining ancestry in rooted trees. The technique we use to solve this problem will motivate the depth-first search technique. We will then show how depth-first search can be used to find articulation points in a connected undirected graph. Finally, we will show how the technique can be extended to problems for directed graphs.
13.1 Ancestry in Rooted Trees

Suppose we are given two nodes, \( x \) and \( y \), in a rooted tree \( T \). We wish to determine whether \( x \) is an ancestor of \( y \). By traversing the subtree rooted at \( x \) (see Section 6.1), we can determine whether \( y \) is in that subtree. In the worst case, \( y \) is not in the subtree, so we have to traverse the entire subtree. If the subtree contains most of the nodes in \( T \), the traversal will take \( \Theta(n) \) time, where \( n \) is the number of nodes in the tree.

It seems unlikely that we would be able to solve this problem in \( o(n) \) time. However, note that we can traverse the entire tree in \( \Theta(n) \) time. This fact might help us to solve efficiently the problem of determining ancestry for several pairs of nodes. Perhaps we can do a single traversal in \( \Theta(n) \) time, and save enough information that only constant additional time is needed to decide ancestry for any pair of nodes in the tree.

We can get some insight into how to accumulate the necessary information by reviewing preorder and postorder traversals, as outlined in Section 6.1. A preorder traversal visits a node \( x \) before visiting any of its proper descendants, whereas a postorder traversal visits \( x \) after visiting all of its proper descendants. Therefore, if \( x \) is a proper ancestor of \( y \), a preorder traversal will visit \( x \) before visiting \( y \), and a postorder traversal will visit \( x \) after visiting \( y \). If we could combine a preorder traversal with a postorder traversal and keep track of the order in which the visits were made, we could then efficiently check a necessary condition for \( x \) being a proper ancestor of \( y \).

We will show that the above condition is also sufficient for \( x \) being a proper ancestor of \( y \). First, however, let us present an algorithm for calculating this information in order to be able to reason about it more precisely. In order to describe the algorithm, we need a simple data structure called a \textsc{VisitCounter}, whose definition is shown in Figure 13.1. It has two representation variables, an integer \( \text{count} \) and an array \( \text{num}[0..n-1] \). Its structural invariant is that for \( 0 \leq i < n \), \( \text{num}[i] \) is a natural number, and that

\[
\text{count} = \max_{0 \leq i < n} \text{num}[i].
\]

We interpret the size of the structure to be the size of \( \text{num} \), and we interpret the value of \( \text{num}[i] \) as the value associated with \( i \). Clearly, the constructor runs in \( \Theta(n) \) time, and the operations all run in \( \Theta(1) \) time.

The algorithm shown in Figure 13.2 combines the preorder and postorder traversals of the tree \( T \). We use a (directed) \textsc{Graph} to represent \( T \). \textit{pre} is a \textsc{VisitCounter} that records the order in which nodes are visited in the
Figure 13.1 The data structure VisitCounter

**Structural Invariant:** For $0 \leq i < n$, $num[i]$ is a NAT, and

\[ count = \max_{0 \leq i < n} num[i]. \]

**Precondition:** $n$ is a NAT.

**Postcondition:** Constructs a VisitCounter of size $n$, all of whose values are 0.

VisitCounter($n$)

```plaintext
count ← 0; num ← new Array[0..n - 1]
for $i ← 0$ to $n - 1$
    num[$i$] ← 0
```

**Precondition:** $i$ is a NAT strictly less than Size().

**Postcondition:** Associates with $i$ a value of $m + 1$, where $m$ is the largest value initially associated with any $j$, $0 \leq j < \text{Size}()$.

VisitCounter.Visit($i$)

```plaintext
count ← count + 1; num[$i$] ← count
```

**Precondition:** $i$ is a NAT strictly less than Size().

**Postcondition:** Returns the value associated with $i$.

VisitCounter.Num($i$)

```plaintext
return num[$i$]
```

**Precondition:** true.

**Postcondition:** Returns the size of this VisitCounter.

VisitCounter.Size()

```plaintext
return SizeOf(num)
```
Precondition: $T$ is a Graph representing a rooted tree with edges directed from parents to children. $pre$ and $post$ are VisitCounters whose size $n$ is the number of nodes in $T$, and $i$ is a Nat strictly less than $n$.

Postcondition: Let $S$ be the set of descendants of $i$. For every $j \in S$ and every node $k \notin S$, $pre.\text{Num}(j) > pre.\text{Num}(k)$ and $post.\text{Num}(j) > post.\text{Num}(k)$. For any $j, k \in S$, $j$ is a proper ancestor of $k$ iff $pre.\text{Num}(j) < pre.\text{Num}(k)$ and $post.\text{Num}(j) > post.\text{Num}(k)$. If node $k \notin S$, then $pre.\text{Num}(k)$ and $post.\text{Num}(k)$ are unchanged.

PrePostTraverse($T, i, pre, post$)

1. $pre.\text{Visit}(i) \leftarrow T.\text{AllFrom}(i)$
   
   // Invariant: The postcondition holds with $S$ denoting the set of proper descendants of $i$ that are not descendants of any node in $L$, except that $pre.\text{Num}(i)$ has been changed to be larger than $pre.\text{Num}(k)$ for every $k \notin S \cup \{i\}$.

2. While not $L.\text{IsEmpty}()
   
   next $\leftarrow L.\text{Head}().\text{Dest}(); \quad L \leftarrow L.\text{Tail}()

   PrePostTraverse($T, next, pre, post$)

   $post.\text{Visit}(i)$

preorder traversal, and $post$ is a VisitCounter that records the order in which nodes are visited in the postorder traversal.

Note that PrePostTraverse is different from most recursive algorithms in that there is no explicit base case. However, a base case does exist — the case in which $i$ has no outgoing edges. In this case, the loop will not iterate, and no recursive call will be made. The lack of an explicit base case is reflected in the following correctness proof, which likewise does not contain a separate base case.

**Theorem 13.1** PrePostTraverse satisfies its specification.

**Proof:** By induction on $n$, the size of $T$.

**Induction Hypothesis:** Assume that PrePostTraverse satisfies its specification for every tree with strictly fewer than $n$ nodes.
**Induction Step:** Assume the precondition is satisfied. We must show the correctness of the invariant.

**Initialization:** The call to `pre.Visit(i)` makes `pre.Num(i)` larger than any other values in `pre`. Otherwise, no other values in `pre` or `post` have been changed from their initial values. At the beginning of the loop, `L` contains the children of `i` and `S = ∅`. The invariant is therefore satisfied.

**Maintenance:** Suppose the invariant holds at the beginning of an iteration. Clearly, the precondition holds for the recursive call. Because the subtree rooted at `next` has strictly fewer nodes than does `T`, from the Induction Hypothesis, the recursive call satisfies the postcondition with `S` denoting the set of descendants of `next`. Let `R` be the set of descendants of `next`. Let `S′` denote the value of `S` at the end of the iteration; i.e., `S′ = S ∪ R`. We must show that the invariant holds for `S′` at the end of the iteration.

Let us first determine the values in `pre` and `post` that have changed from their initial values by the time the iteration completes. From the invariant, only `pre.Num(i)`, `pre.Num(j)`, and `post.Num(j)` such that `j ∈ S` have changed prior to the beginning of the iteration. From the Induction Hypothesis, the recursive call only changes the values of `pre.Num(j)` and `post.Num(j)` for `j ∈ R`. Thus, the only values to have changed from their initial values are `pre.Num(i)`, `pre.Num(j)`, and `post.Num(j)` such that `j ∈ S′`. Furthermore because only values for `j ∈ R` are changed by the iteration, it is still the case that `pre.Num(i) > pre.Num(k)` for all `k /∈ S′ ∪ {i}`.

Let `j ∈ S′` and `k /∈ S′`. If `j /∈ R`, then `pre.Num(j)`, `post.Num(j)`, `pre.Num(k)` and `post.Num(k)` are unchanged by the iteration. Therefore, because `j ∈ S`, from the invariant, it is still the case that `pre.Num(j) > pre.Num(k)` and `post.Num(j) > post.Num(k)`. On the other hand, suppose `j ∈ R`. Because `k /∈ R`, by the Induction Hypothesis, `pre.Num(j) > pre.Num(k)` and `post.Num(j) > post.Num(k)` at the end of the iteration.

Now let `j, k ∈ S′`. We must show that `j` is a proper ancestor of `k` iff `pre.Num(j) < pre.Num(k)` and `post.Num(j) > post.Num(k).

⇒: Suppose `j` is a proper ancestor of `k`. Then it is either `j` and `k` are both in `R` or neither is in `R`. If neither `j` nor `k` is in `R`, then the iteration changes none of their `pre` or `post` values. Hence, from the invariant, `pre.Num(j) < pre.Num(k)` and `post.Num(j) > post.Num(k)`. On the other hand, if `j, k ∈ R` then from the Induction Hypothesis, `pre.Num(j) < pre.Num(k)` and `post.Num(j) > post.Num(k).`
\[ S \text{ ancestor of every node in } T \]

\[ R \text{ only values to have been changed by the algorithm are } j \]

\[ \text{Termination: Because } L \text{ contains finitely many elements and each iteration removes one element from } L, \text{ the loop must eventually terminate.} \]

\[ \text{Correctness: Assume the invariant holds and that } L \text{ is empty when the loop terminates. We need to show that the postcondition holds when the algorithm finishes. Let } S \text{ denote the set of descendants of } i. \]

\[ \text{Let us first consider which values in } pre \text{ and } post \text{ have been changed by the algorithm. From the invariant, only } pre(i), \text{ } pre(j), \text{ and } post(j), \text{ where } j \in S \setminus \{i\}, \text{ have changed by the time the loop terminates. The final call to } post.\text{Visit}(i) \text{ changes } post(i). \text{ Therefore, the only values to have been changed by the algorithm are } pre(j) \text{ and } post(j) \text{ such that } j \in S. \]

\[ \text{Let } j \in S \text{ and } k \notin S. \text{ If } j \neq i, \text{ then from the invariant, } pre(j) > pre(k) \text{ and } post(j) > post(k). \text{ If } j = i, \text{ then from the invariant } pre(j) > pre(k). \text{ Furthermore, the call to } post.\text{Visit}(i) \text{ makes } post(j) > post(k). \]

\[ \text{Now let } j, k \in S. \text{ We must show that } j \text{ is a proper ancestor of } k \iff \text{pre}(j) < \text{pre}(k), \text{ and } \text{post}(j) > \text{post}(k). \]

\[ \Rightarrow: \text{ Suppose } j \text{ is a proper ancestor of } k. \text{ Then } k \neq i, \text{ because } i \text{ is an ancestor of every node in } S. \text{ If } j \neq i, \text{ then it follows from the invariant that } \text{pre}(j) < \text{pre}(k) \text{ and } \text{post}(j) > \text{post}(k). \text{ If } j = i, \text{ then from the invariant } \text{pre}(j) < \text{pre}(k), \text{ and the final call to } post.\text{Visit}(i) \text{ makes } \text{post}(j) > \text{post}(k). \]

\[ \Leftarrow: \text{ Suppose } \text{pre}(j) < \text{pre}(k) \text{ and } \text{post}(j) > \text{post}(k). \text{ If neither } j = i \text{ nor } k = i, \text{ then from the invariant, } j \text{ is a proper ancestor of } k. \text{ If } j = i, \text{ then clearly } j \text{ is a proper ancestor of } k. \text{ Finally, } k \neq i \text{ because the final call to } post.\text{Visit}(i) \text{ would then make } \text{post}(k) > \text{post}(j). \]

\[ \square \]

In order to analyze the running time of PrePostTraverse, let us assume that } T \text{ is represented as a ListGraph. Then the call to AllFrom
Figure 13.3 Algorithm for testing ancestry for multiple pairs of nodes in a rooted tree

**Precondition:** $T$ refers to a Graph representing a tree rooted at $i$ with edges directed from parents to children, and $x[1..m]$ and $y[1..m]$ are arrays of NATS less than $T$.Size().

**Postcondition:** Returns an array $ancestor[1..m]$ of BOOLs such that $ancestor[i]$ is true iff $x[i]$ is a proper ancestor of $y[i]$.

Ancestors($T$, $i$, $x[1..m]$, $y[1..m]$)

$n \leftarrow T$.Size();
ancestor $\leftarrow$ new ARRAY[1..m]
pre $\leftarrow$ new VISIT_COUNTER(n); post $\leftarrow$ new VISIT_COUNTER(n)
PrePostTraverse($T$, $i$, pre, post)

for $j \leftarrow 1$ to $m$

condPre $\leftarrow$ pre.NUM($x[j]$) < pre.NUM($y[j]$)
condPost $\leftarrow$ post.NUM($x[j]$) > post.NUM($y[j]$)
ancestor[$j$] $\leftarrow$ condPre and condPost

return ancestor

runs in $\Theta(1)$ time. Furthermore, the while loop iterates exactly $m$ times, where $m$ is the number of children of $i$. Because each iteration of the while loop results in one recursive call, it is easily seen that the running time is proportional to the total number of calls to PrePostTraverse. It is easily shown by induction on $n$, the number of nodes in the subtree rooted at $i$, that a call in which the second parameter is $i$ results in exactly $n$ total calls. The running time is therefore in $\Theta(n)$.

The algorithm for testing ancestry for multiple pairs of nodes is given in Figure 13.3. The initialization prior to the call to PrePostTraverse clearly runs in $\Theta(n)$ time, as does the call to PrePostTraverse. The body of the loop runs in $\Theta(1)$ time. Because the loop iterates $m$ times, the entire algorithm runs in $\Theta(n + m)$ time.

### 13.2 Reachability in a Graph

We will now show how the technique used in the last section can be applied to graph problems. Consider the problem of determining whether there is a path from a given vertex $i$ to a given vertex $j$ in an undirected graph.
CHAPTER 13. DEPTH-FIRST SEARCH

Viewing the problem top-down, we first note that there is a path if \( i = j \). Otherwise, we can retrieve all of the vertices adjacent to \( i \) and remove \( i \) from the graph. For each vertex \( k \) that was adjacent to \( i \), we can then determine whether there is a path from \( k \) to \( j \) in the resulting graph. We must be careful, however, because each of these tests is destructive — it removes all of the vertices it reaches. As a result, we must be sure a node \( k \) is still in the graph before we solve that subproblem. Note that if it has been removed, then it must have been reachable from one of the other vertices adjacent to \( i \); hence all nodes reachable from \( k \) have been removed. Thus, if \( j \) were reachable from \( k \), we would have already found that it was reachable from \( i \).

In order to avoid deleting vertices from the graph, we need a mechanism for selecting a subgraph based on a given subset of the vertices. More precisely, let \( G = (V, E) \) be a (directed or undirected) graph, and let \( V' \subseteq V \). We define the subgraph of \( G \) induced by \( V' \) to be \( G' = (V', E') \), where \( E' \) is the set of edges connecting vertices from \( V' \). We therefore need a mechanism for selecting a subset of the vertices in the graph.

For this purpose, we define the data structure Selector. A Selector represents a set of \( n \) elements numbered \( 0, \ldots, n - 1 \), each of which is either selected or unselected. The constructor and operations for Selector are specified in Figure 13.4. It is a straightforward matter to implement Selector using an array of booleans so that the constructor, SelectAll, and UnSelectAll run in \( \Theta(n) \) time, where \( n \) is the number of elements represented, and so that the remaining operations run in \( \Theta(1) \) time.

We can now traverse the graph using almost the same algorithm as PrePostTraverse — the only differences are that pre and post are not needed, and we must check that a vertex has not already been visited before we traverse it. We call this traversal a depth-first search (DFS). The entire algorithm is shown in Figure 13.5. We retain pre and post in order to maintain a close relationship between ReachDFS and PrePostTraverse.

Let \( G \) be an undirected graph, and let \( i \in \mathbb{N} \) such that \( i < G.\text{Size()} \). Further let \( sel \) be a Selector of size \( G.\text{Size()} \) in which all elements are selected, and let \( pre \) and \( post \) be VisitCounters of size \( G.\text{Size()} \) in which all values are 0. Suppose we invoke ReachDFS\((G, i, sel, pre, post)\). We define a directed graph \( G' \) as follows, based on the behavior of this invocation:

- \( G' \) has the same vertices as \( G \);
- \( G' \) has the edge \((j, k)\) iff a call ReachDFS\((G, j, sel, pre, post)\) is made, which in turn calls ReachDFS\((G, k, sel, pre, post)\).
Let us consider the structure of \( G' \). We first observe that for each vertex \( k \neq i \), there is some edge \((j,k)\) in \( G' \) iff \( k \) is reachable from \( i \) in \( G \). Furthermore, a call to \( \text{ReachDFS}(G, k, \text{sel}, \text{pre}, \text{post}) \) can be made only if \( \text{sel}.\text{IsSelected}(k) = \text{true} \). Because this call immediately unselects \( k \), and because the algorithm never selects a vertex, it follows that \( \text{ReachDFS}(G, k, \text{sel}, \text{pre}, \text{post}) \) can be called at most once. Hence, each vertex in \( G' \) has at most one incoming edge. Finally, \( i \) can have no incoming edges. Therefore, \( G' \) forms a tree rooted at \( i \). The vertices of \( G' \) are exactly the vertices reachable from \( i \) in the subgraph of \( G \) induced by the selected vertices. \( G' \) is therefore a rooted spanning tree of the connected component.
**Precondition:** $G$ refers to an undirected graph, and $i$ and $j$ are Nats strictly less than $G$.Size().

**Postcondition:** Returns true iff there is a path in $G$ from $i$ to $j$.

\[
\text{Reachable}(G, i, j) = n \leftarrow G.\text{Size}(); \text{sel} \leftarrow \text{new Selector}(n) \hspace{1cm} \text{pre} \leftarrow \text{new VisitCounter}(n); \text{post} \leftarrow \text{new VisitCounter}(n) \hspace{1cm} \text{ReachDFS}(G, i, \text{sel}, \text{pre}, \text{post}) \hspace{1cm} \text{return not } \text{sel}.\text{IsSelected}(j)
\]

**Precondition:** $G$ refers to an undirected graph, $i$ is a Nat such that $i < G$.Size(), $\text{sel}$ refers to a Selector of size $G$.Size(), $\text{pre}$ and $\text{post}$ refer to VisitCounters of size $G$.Size(), and $\text{sel}.\text{IsSelected}(i) = \text{true}$.

**Postcondition:** Unselects each $j$ such that $j$ is reachable from $i$ in $G'$, where $G'$ denotes the subgraph of $G$ induced by the set of selected vertices.

\[
\text{ReachDFS}(G, i, \text{sel}, \text{pre}, \text{post}) = \text{sel}.\text{Unselect}(i); \text{pre}.\text{Visit}(i); L \leftarrow G.\text{AllFrom}(i) \hspace{1cm} \text{while not } L.\text{IsEmpty}()
\]

\[
\hspace{1cm} \text{next} \leftarrow L.\text{Head}().\text{Dest}(); L \leftarrow L.\text{Tail}() \hspace{1cm} \text{if } \text{sel}.\text{IsSelected}(next)
\]

\[
\hspace{1cm} \text{ReachDFS}(G, \text{next}, \text{sel}, \text{pre}, \text{post}) \hspace{1cm} \text{post}.\text{Visit}(i)
\]

It should now be clear that the calls to ReachDFS$(G, i, \text{sel}, \text{pre}, \text{post})$ and PrePostTraverse$(G', i, \text{pre}, \text{post})$ produce exactly the same values in $\text{pre}$ and $\text{post}$. In essence, ReachDFS performs both a preorder traversal and a postorder traversal on a rooted tree. This rooted tree is a spanning tree of a particular connected component of a given graph. The given graph is the subgraph of the input graph $G$ induced by the selected vertices, and the connected component is specified by the input vertex $i$. The spanning tree is not specified, but is implied by the behavior of ReachDFS. We call this spanning tree the depth-first spanning tree generated by the call to ReachDFS$(G, i, \text{sel}, \text{pre}, \text{post})$.

We can use the correspondence between ReachDFS and PrePostTra-
VERSE in order to analyze the running time of ReachDFS. Suppose $G$ is implemented as a ListGraph. Let $G'$ be the subgraph of $G$ induced by the selected vertices, and let $G''$ be the connected component of $G'$ containing $i$; thus, the vertices in $G''$ are the vertices visited by ReachDFS. Let $n$ be the number of vertices in $G''$. Certainly, ReachDFS runs in $\Omega(n)$ time. The only difference in the two algorithms is that in ReachDFS, the loop may iterate more times. Thus, if we ignore the iterations in which no recursive call is made, the running time is the same as that of PrePostTraverse: $\Theta(n)$.

In the call ReachDFS($G, j, sel, pre, post$), the loop iterates $m$ times, where $m$ is the number of vertices adjacent to $j$ in $G$. The total number of iterations in all recursive calls is therefore $2a_1 + a_2$, where $a_1$ is the number of edges in $G''$ and $a_2$ is the number of edges in $G$ from vertices in $G''$ to vertices not in $G''$. The time for a single iteration that does not make a recursive call is in $\Theta(1)$. Because $G''$ is connected, $a_1 \geq n - 1$; hence, the total running time of ReachDFS($G, i, sel, pre, post$) is in $\Theta(a)$, where $a$ is the number of edges in $G$ incident on vertices in $G''$.

### 13.3 A Generic Depth-First Search

Due to its hierarchical nature, a rooted tree is more amenable to the top-down approach to algorithm design than is a graph. Furthermore, as we will see shortly, a depth-first spanning tree has several additional properties that can prove useful for designing graph algorithms. For this reason, it makes sense to generalize ReachDFS to a general-purpose depth-first search algorithm. With such an algorithm, we can then design our algorithms as traversals of depth-first spanning trees.

In order to generalize this algorithm, we need an ADT for defining various ways of processing a depth-first spanning tree. Upon examining ReachDFS, we see that there are five places where processing might occur:

- Preorder processing of vertices can occur prior to the loop.
- Preorder processing of tree edges might occur prior to the recursive call.
- Postorder processing of tree edges might occur following the recursive call.
- Though the if statement in ReachDFS has no else-block, we might include an else-block for processing other edges.
• Postorder processing of vertices can occur following the loop.

We therefore have the ADT specified in Figure 13.6. The generic depth-first search is shown in Figure 13.7.

Let us now consider the useful properties of depth-first spanning trees. These properties concern the non-tree edges. First, we show the following theorem regarding undirected graphs.

**Theorem 13.2** Let $G$ be a connected undirected graph with $n$ vertices, and let $sel$ be a Selector of size $n$ in which all elements are selected. Suppose we call $\text{Dfs}(G, i, sel, s)$, where $s$ is a Searcher of size $n$. Then for every edge $\{j, k\}$ processed as a non-tree edge, either $j$ is an ancestor of $k$ or $k$ is an ancestor of $j$.

**Proof:** Without loss of generality, assume $j$ is unselected before $k$ is. Consider the call to $\text{Dfs}$ on vertex $j$. Initially, $j$ is preorder processed while $k$ is still selected. We consider two cases.

**Case 1:** $\{j, k\}$ is processed as a non-tree edge in the call to $\text{Dfs}$ on $j$. Then when this happens, $k$ must be unselected. There must therefore have been a call to $\text{Dfs}$ on $k$ which unselected $k$. This call resulted in $k$ being both preorder processed and postorder processed after $j$ was preorder processed, but before $j$ was postorder processed. $j$ is therefore a proper ancestor of $k$.

**Case 2:** $\{j, k\}$ is processed as a tree edge in the call to $\text{Dfs}$ on $j$, but is processed as a non-tree edge in the call to $\text{Dfs}$ on $k$. In this case, $k$ is by definition a child of $j$. □

The above theorem gives the property of depth-first spanning trees that makes depth-first search so useful for connected undirected graphs. Given a connected undirected graph $G$ and a depth-first spanning tree $T$ of $G$, let us refer to edges of $G$ that correspond to edges in $T$ as *tree edges*. We will call all other edges *back edges*. By definition, tree edges connect parents with children. Theorem 13.2 tells us that back edges connect ancestors with descendants.

However, Theorem 13.2 does not apply to depth-first search on a directed graph. To see why, consider the graph shown in Figure 13.8. The solid edges in part (b) show a depth-first search tree for the graph in part (a); the remaining edges of the graph are shown with dashed lines in part (b). Because 0 is the root and all other vertices are reachable from 0, all other
Precondition: \( n \) is a Nat.
Postcondition: Constructs a new Searcher of size \( n \).

**Searcher\((n)\)**

Precondition: \( i \) is a Nat less than the size of this Searcher.
Postcondition: true.

**Searcher.PreProc\((i)\)**

Precondition: \( i \) is a Nat less than the size of this Searcher.
Postcondition: true.

**Searcher.PostProc\((i)\)**

Precondition: \( e \) is an Edge whose vertices are less than the size of this Searcher.
Postcondition: true.

**Searcher.TreePreProc\((e)\)**

Precondition: \( e \) is an Edge whose vertices are less than the size of this Searcher.
Postcondition: true.

**Searcher.TreePostProc\((e)\)**

Precondition: \( e \) is an Edge whose vertices are less than the size of this Searcher.
Postcondition: true.

**Searcher.OtherEdgeProc\((e)\)**
**Precondition:** $G$ refers to a Graph, $i$ is a Nat less than $G$.Size(), $sel$ refers to a Selector with size $G$.Size() such that $i$ is selected, and $s$ refers to a Searcher with size $G$.Size().

**Postcondition:** Traverses a depth-first spanning tree rooted at $i$ on the connected component containing $i$ in the subgraph of $G$ induced by the selected vertices. Each vertex $j$ in this tree is processed by calling $s$.PreProc($j$) before any of $j$’s proper descendants are processed and by calling $s$.PostProc($j$) after all of $j$’s descendants are processed. Each edge $(j, k)$ in the tree is processed by calling $s$.TreePreProc($j, k$) before $k$ is processed and by calling $s$.TreePostProc($j, k$) after $k$ is processed. All other edges $e$ from $j$ to any node in $G$ are processed by calling $s$.OtherEdgeProc($e$).

Dfs($G, i, sel, s$)

  sel.Unselect($i$); $s$.PreProc($i$); $L \leftarrow G$.AllFrom($i$)
  
  while not $L$.IsEmpty()
      
      edge $\leftarrow L$.Head(); next $\leftarrow edge$.Dest(); $L \leftarrow L$.Tail()
      
      if sel.IsSelected(next)
          $s$.TreePreProc(edge)
          Dfs($G, next, sel, s$)
          $s$.TreePostProc(edge)
      else
          $s$.OtherEdgeProc(edge)
      
      $s$.PostProc($i$)

vertices are descendants of 0. Suppose $(0, 1)$ is the first edge from 0 to be processed. Because 2 is the only vertex reachable from 1, it is the only proper descendant of 1. Of the remaining edges from 0, only $(0, 3)$ leads to a vertex that has not yet been reached, so 3 is the only other child of 0. Finally, because 4 is reachable from 3, 3 is the parent of 4. In the resulting depth-first spanning tree, 3 is neither an ancestor nor a descendant of 1, but there is an edge $(3, 1)$ in the graph.

The point at which the proof of Theorem 13.2 fails for directed graphs is the initial assumption that $j$ is unselected before $k$ is. For an undirected graph, one of the endpoints of the edge will be unselected first, and it doesn’t
matter which endpoint we call $j$. However, with a directed edge, either the source or the destination may be unselected first, and we must consider both cases. Given the assumption that the source is unselected first, the remainder of the proof follows. We therefore have the following theorem.

**Theorem 13.3** Let $G$ be a directed graph with $n$ vertices such that all vertices are reachable from $i$, and let $sel$ be a SELECTOR of size $n$ in which all elements are selected. Suppose we call $\text{Dfs}(G, i, sel, s)$, where $s$ is a SEARCHER of size $n$. Then for every edge $(j, k)$ processed as a non-tree edge, if $j$ is unselected before $k$ is, then $j$ is an ancestor of $k$.

Thus, if we draw a depth-first spanning tree with subtrees listed from left to right in the order we unselect them (as in Figure 13.8), there will be no edges leading from left to right. As we can see from Figure 13.8, all three remaining possibilities can occur, namely:

- edges from ancestors to descendants (we call these *forward edges* if they are not in the tree);
- edges from descendants to ancestors (we call these *back edges*); and
- edges from right to left (we call these *cross edges*).

Theorem 13.3 gives us the property we need to make use of depth-first search with directed graphs.

As a final observation, we note that back edges in directed graphs always form cycles, because there is always a path along the tree edges from a vertex to any of its descendants. Hence, a directed acyclic graph cannot have back edges.
In the next three sections, we will show how to use depth-first search to design algorithms for connected undirected graphs, directed acyclic graphs, and directed graphs.

### 13.4 Articulation Points

As we mentioned at the beginning of this chapter, an **articulation point** in a connected undirected graph is any vertex whose removal yields a disconnected graph. If, for example, a given graph represents a communications network, then an articulation point is a node whose failure would partition the network. It would therefore be desirable to know if a given network contains any articulation points.

Let $G$ be a connected undirected graph, and let $T$ be a depth-first spanning tree for $G$. We first note that it is easy to tell if the root of $T$ is an articulation point. If $T$ has only one child, then the removal of its root from $G$ cannot disconnect $G$ — the tree edges continue to connect the graph. On the other hand, from Theorem 13.2, $G$ can have no edges between different subtrees of $T$. Thus, if $T$ has more than one child, its root must be an articulation point of $G$. We therefore conclude that the root of $T$ is an articulation point of $G$ if and only if it has more than one child.

The above property suggests the following algorithm. Let $n$ be the number of vertices and $a$ be the number of edges in $G$. We do $n$ separate depth-first searches, using a different vertex as the root for each search. As we process the tree edges (either in preorder or in postorder), we count the number of children of the root. After the search completes, we then determine whether the root is an articulation point by examining the number of children it has. It is not hard to see that we could construct such an algorithm with running time in $\Theta(na)$, provided $G$ is implemented as a `ListGraph`.

In order to obtain a more efficient algorithm, let us consider some vertex $i$ other than the root of $T$. If there are no back edges in $G$, then the removal of $i$ disconnects $G$ if $i$ has at least one child. In this case, $G$ is partitioned into one connected component for each child of $i$, plus a connected component containing all vertices that are not descendants of $i$. If, however, there are back edges, these back edges may connect all of these partitions together. Note, however, that from Theorem 13.2, no back edge can connect partitions containing two different children of $i$. In particular, if a back edge does connect a partition containing a child of $i$ with another partition, it leads from a proper descendant of $i$ to a proper ancestor of $i$. We therefore
conclude that \( i \) is an articulation point iff \( i \) has at least one child \( j \) in \( T \) such that no descendant of \( j \) is adjacent to a proper ancestor of \( i \).

If we can efficiently test the above property, then we should be able to find all articulation points with a single depth-first search. Note that it is sufficient to know, for each vertex \( j \) other than the root, the highest ancestor \( k \) of \( j \) that is adjacent to some descendant of \( j \). The parent \( i \) of \( j \) is an articulation point if \( k = i \). On the other hand, if for each child \( j \) of \( i \), the highest ancestor \( k \) adjacent to some descendant of \( j \) is a proper ancestor of \( i \), then \( i \) is not an articulation point. Because a vertex is preorder processed before all of its descendants, we can determine which of a given set of ancestors of a vertex is the closest to the root by determining which was preorder processed first. Thus, let us use a \texttt{VisitCounter} \( \text{pre} \) to keep track of the order the vertices are preorder processed. We then need to compute the following value for each vertex \( i \) other than the root:

\[
\text{highest}[i] = \min\{ \text{pre}.\text{Num}(k) \mid k \text{ is adjacent to a descendant of } i \}. \quad (13.1)
\]

Let us consider a top-down approach to computing \texttt{highest}[\( i \)]. We first observe that if \( j \) is a child of \( i \), then \( \text{highest}[i] \leq \text{highest}[j] \). Furthermore, the vertex \( k \) determining \texttt{highest}[\( i \)] is adjacent to either \( i \) or some proper descendant of \( i \). If \( k \) is adjacent to a proper descendant of \( i \), then \( \text{highest}[i] = \text{highest}[j] \) for some child \( j \) of \( i \). Finally, because \( i \) is adjacent to its parent, which has a smaller value in \( \text{pre} \) than does any child of \( i \), we can ignore the \( \text{pre} \) values of the children of \( i \) in computing \texttt{highest}[\( i \)]. We therefore conclude that

\[
\text{highest}[i] = \min( \{ \text{pre}.\text{Num}(k) \mid \{i, k\} \text{ is a back edge} \} \cup \{ \text{highest}[j] \mid j \text{ is a child of } i \}). \quad (13.2)
\]

We can now build a \texttt{Searcher} \( s \) so that \texttt{Dfs}(\( G, 0, \text{sel}, s \)) will find the articulation points of \( G \), where \( \text{sel} \) is an appropriate \texttt{Selector}. (Note that it doesn’t matter which node is used as the root of the depth-first search, so we will arbitrarily use \( 0 \).) Let \( n \) be the number of vertices in \( G \). We need as representation variables a \texttt{VisitCounter} \( \text{pre} \) of size \( n \), an array \texttt{highest}[\( 0..n-1 \)], a readable array \texttt{artPoints}[\( 0..n-1 \)] of booleans to store the results, and a natural number \texttt{rootChildren} to record the number of children of the root. Note that making \texttt{artPoints} readable makes this data structure insecure, because code that can read the reference to the array can change values in the array. We will discuss this issue in more detail shortly.

To implement the \texttt{Searcher} operations, we only need to determine when the various calculations need to be done. Initialization should go in the
constructor; however, because the elements of the arrays are not needed until the corresponding vertices are processed, we can initialize these elements in 
PreProc. We want the processing of a vertex \( i \) to compute \( \text{highest}[i] \). In order to use recurrence (13.2), we need \( \text{pre.num}(k) \) for each back edge \( \{i, k\} \) and \( \text{highest}[j] \) for each child \( j \) of \( i \). We therefore include the code to compute \( \text{highest}[i] \) in OtherEdgeProc and TreePostProc. The determination of whether a vertex \( i \) other than the root is an articulation point needs to occur once we have computed \( \text{highest}[j] \) for each child \( j \) of \( i \); hence, we include this code in TreePostProc. To be able to determine whether the root is an articulation point, we count its children in TreePostProc. We can then make the determination once all of the processing is complete, i.e., in the call to PostProc for the root.

The implementation of ArtSearcher is shown in Figure 13.9. We have not given an implementation of the TreePreProc operation — it does nothing. We have also not specified any preconditions or postconditions for the constructor or any of the operations. The reason for this is that we are only interested in what happens when we use this structure with a depth-first search. It therefore doesn’t make sense to prove its correctness in every context. As a result, we don’t need to make this structure secure. Furthermore, the code in each of the operations is so simple that specifying preconditions and postconditions is more trouble than it is worth. As we will see, it will be a straightforward matter to prove that the algorithm that uses this structure is correct.

We can now construct an algorithm that uses depth-first search to find the articulation points in a connected undirected Graph \( G \). The algorithm is shown in Figure 13.10. Let \( n \) be the number of vertices and \( a \) be the number of edges in \( G \), and suppose \( G \) is implemented as a ListGraph. Because each of the operations in ArtSearcher runs in \( \Theta(1) \) time, it is easily seen that the call to Dfs runs in \( \Theta(a) \) time, the same as ReachDFS. The remaining statements run in \( \Theta(n) \) time. Because \( G \) is connected, \( n \in O(a) \), so the entire algorithm runs in \( \Theta(a) \) time.

To prove that ArtPts is correct, we need to show that the call to Dfs results in \( s.\text{artPoints} \) containing the correct boolean values. In order to prove this, it is helpful to prove first that \( s.\text{highest} \) contains the correct values. This proof uses the fact that Dfs performs a traversal of a depth-first spanning tree of \( G \).

Lemma 13.4 Let \( G \) be a connected undirected Graph with \( n \) vertices. Let \( \text{sel} \) be a Selector of size \( n \) in which all elements are selected, and let \( s \) be a newly-constructed ArtSearcher of size \( n \). Then Dfs\((G, 0, \text{sel}, s)\) results
CHAPTER 13. DEPTH-FIRST SEARCH

**Figure 13.9 ArtSearcher implementation of Searcher**

ArtSearcher\((n)\)
\[
\text{artPoints} \leftarrow \textbf{new} \text{ Array}[0..n-1]; \text{highest} \leftarrow \textbf{new} \text{ Array}[0..n-1] \\
\text{pre} \leftarrow \textbf{new} \text{ VisitCounter}(n); \text{rootChildren} \leftarrow 0
\]

ArtSearcher\(\text{.PreProc}(i)\)
\[
\text{pre.Visit}(i); \text{artPoints}[i] \leftarrow \text{false}; \text{highest}[i] \leftarrow \infty
\]

ArtSearcher\(\text{.TreePostProc}(e)\)
\[
i \leftarrow e.\text{Source}(); j \leftarrow e.\text{Dest}(); \text{highest}[i] \leftarrow \text{Min}(\text{highest}[i], \text{highest}[j]) \\
\text{if } i = 0 \\
\quad \text{rootChildren} \leftarrow \text{rootChildren} + 1 \\
\text{else if } \text{highest}[j] = \text{pre.\text{Num}}(i) \\
\quad \text{artPoints}[i] \leftarrow \text{true}
\]

ArtSearcher\(\text{.OtherEdgeProc}(e)\)
\[
i \leftarrow e.\text{Source}(); k \leftarrow e.\text{Dest}() \\
\text{highest}[i] \leftarrow \text{Min}(\text{highest}[i], \text{pre.\text{Num}}(k))
\]

ArtSearcher\(\text{.PostProc}(i)\)
\[
\text{if } i = 0 \text{ and rootChildren} > 1 \\
\quad \text{artPoints}[i] \leftarrow \text{true}
\]

in \(s.\text{highest}[i]\) having the value specified in Equation (13.1) for \(1 \leq i < n\).

**Proof:** Let \(1 \leq i < n\). We first observe that \(s.\text{highest}[i]\) is only changed by \(\text{PreProc}(i), \text{TreePostProc}(e), \text{and OtherEdgeProc}(e)\), where \(e\) is an edge from \(i\); i.e., \(s.\text{highest}[i]\) is only changed during the processing of vertex \(i\). We will show by induction on \(m\), the number of descendants of \(i\) in the depth-first spanning tree, that the processing of vertex \(i\) gives \(s.\text{highest}[i]\) the correct value.

**Induction Hypothesis:** Assume that for any \(j\) with fewer than \(m\) descen-
**Precondition:** $G$ refers to a connected undirected graph.

**Postcondition:** Returns an array $A[0..G.Size()-1]$ of booleans such that $A[i]$ is true iff $i$ is an articulation point in $G$.

\[
\text{ArtPts}(G) \\
\quad n \leftarrow G.Size(); \quad s \leftarrow \text{new ArtSearcher}(n); \quad \text{sel} \leftarrow \text{new Selector}(n) \\
\quad \text{DFS}(G, 0, \text{sel}, s) \\
\quad \text{return } s.\text{ArtPoints()}
\]

The proof of Lemma 13.4 relies heavily on known properties of depth-first search and depth-first spanning trees. As a result, it is quite straightforward. Often proofs of correctness for algorithms using depth-first search don’t even require induction. Such is the case for the proof of the following theorem.

**Theorem 13.5** \(\text{ArtPts}\) satisfies its specification.
**CHAPTER 13. DEPTH-FIRST SEARCH**

**Proof:** Let $0 \leq i < n$. We must show that the call to Dfs results in $s.artPoints[i]$ being true if $i$ is an articulation point, or false otherwise. We first note that $artPoints[i]$ is changed only during the processing of $i$. Furthermore, it is initialized to false in PreProc($i$). We must therefore show that it is set to true iff $i$ is an articulation point. We consider two cases.

**Case 1:** $i = 0$. Then $artPoints[i]$ is set to true iff $rootChildren > 1$ in the call to PostProc(0). $rootChildren$ is only changed during the processing of vertex 0. It is initialized in PreProc(0) to 0 and incremented by 1 in TreePostProc(e) for each tree edge $(0, j)$. When PostProc(0) is called, $rootChildren$ therefore contains the number of children of vertex 0, which is the root of the depth-first spanning tree. As we have observed earlier, the root is an articulation point iff it has more than one child.

**Case 2:** $i > 0$. Then $artPoints[i]$ is set to true iff $s.highest[j]$ has a value equal to $s.pre.Num(i)$ in the call to TreePostProc(e), where $e = (i, j)$ for some vertex $j$. Because $(i, j)$ is passed to TreePostProc, $j$ must be a child of $i$. From Lemma 13.4, the call to Dfs sets $s.highest[j]$ to the correct value, as defined in Equation (13.1). Furthermore, an examination of the proof of Lemma 13.4 reveals that this value is set by the processing of vertex $j$. This processing is done prior to the call to TreePostProc(e), so that $s.highest[j]$ has the correct value by the time it is used. Furthermore, $s.pre.Num(i)$ is set to its proper value by PreProc($i$), which is also called before TreePostProc(e). As we have already shown, $i$ is an articulation point iff $s.highest[j] = s.pre.Num(i)$ for some child $j$ of $i$. □

### 13.5 Topological Sort Revisited

In Section 9.2, we gave an algorithm for finding a topological sort of a directed acyclic graph. This algorithm ran in $\Theta(n+a)$ time for a ListGraph with $n$ vertices and $a$ edges. In this section, we give an alternative algorithm that illustrates the use of depth-first search on directed acyclic graphs.

We first note that a shortcoming of the Dfs algorithm is that it only processes vertices that are reachable from the root. In an arbitrary graph, there may be no vertex from which every vertex is reachable; hence it may be impossible to find a depth-first spanning tree (or any spanning tree) for the graph.

To remedy this shortcoming, we provide the algorithm DfsAll, shown...
Figure 13.11 Algorithm for processing an entire graph with depth-first search

Precondition: $G$ is a Graph and $s$ is a Searcher with size $G$.Size().

Postcondition: Traverses a depth-first spanning forest on $G$. Each vertex $j$ in this forest is processed by calling $s$.PrePROC($j$) before any of $j$’s proper descendants are processed and by calling $s$.PostPROC($j$) after all of $j$’s descendants are processed. Each edge ($j, k$) in the forest is processed by calling $s$.TreePrePROC(($j, k$)) before $k$ is processed and by calling $s$.TreePostPROC(($j, k$)) after $k$ is processed. All other edges $e$ from $j$ to any node in $G$ are processed by calling $s$.OtherEDGEPROC($e$).

DfsAll($G, s$)

\[
\begin{align*}
&n \leftarrow G.\text{Size}(); \quad \text{sel} \leftarrow \text{new} \ \text{Selector}(n) \\
&\text{for} \, i \leftarrow 0 \ \text{to} \ n - 1 \\
&\quad \text{if} \ \text{sel}.\text{IsSelected}(i) \\
&\quad \quad \text{Dfs}(G, i, \text{sel}, s)
\end{align*}
\]

in Figure 13.11. In order to understand its postcondition, it helps compare its behavior with that of Dfs. In particular, suppose $G$ has $n$ vertices, and let $G'$ be the graph obtained by adding a new vertex $n$ to $G$ and edges from $n$ to every other vertex in the graph. Notice that the behavior of the for loop in DfsAll($G, s$) is exactly the same as the behavior of Dfs($G', n, \text{sel}, s$), except that DfsAll only processes the vertices and edges in $G$. In particular, each time Dfs is called by DfsAll, it traverses a subtree of $G'$ rooted at a child of the root $n$. For this reason, we call the collection of trees traversed by DfsAll a depth-first spanning forest. In particular, note that either Theorem 13.2 or Theorem 13.3, depending on whether $G$ is undirected or directed, can be extended to apply to this forest.

Let $G$ be a ListGraph with $n$ vertices and $a$ edges. A graph $G'$ constructed by adding a new vertex and $n - 1$ new edges to $G$ then has $n + a - 1$ edges. Therefore, the running time of DfsAll($G, s$) is easily seen to be in $\Theta(n + a)$, provided each of the vertex and edge processing operations in $s$ runs in $\Theta(1)$ time.

Now consider the depth-first spanning forest for a directed acyclic graph. Because there are no cycles, the spanning forest can have no back edges. This leaves only tree edges, forward edges and cross edges. Furthermore, for each
Figure 13.12 TopSortSearcher implementation of Searcher

TopSortSearcher(n)
  order ← new Array[0..n − 1]; loc ← n

TopSortSearcher.PostProc(i)
  loc ← loc − 1; order[loc] ← i

Figure 13.13 Topological sort algorithm using depth-first search

Precondition: G is a directed acyclic graph.
Postcondition: Returns an array listing the vertices of G in topological order.

DfsTopSort(G)
  n ← G.Size(); s ← new TopSortSearcher(n)
  DfsAll(G, s)
  return s.Order()

of these types of edge \((i, j)\), \(j\) is postorder processed before \(i\). This property suggests a straightforward algorithm for topological sort, namely, to order the vertices in the reverse of the order in which they are postorder processed by a depth-first search.

The Searcher for this algorithm needs as representation variables a readable array \(order[0..n − 1]\) for storing the listing of vertices in topological order and a natural number \(loc\) for storing the location in \(order\) of the last vertex to be inserted. Only the constructor and the PostProc operation are nonempty; these are shown in Figure 13.12. The topological sort algorithm is shown in Figure 13.13. If \(G\) is implemented as a ListGraph, the algorithm’s running time is clearly in \(\Theta(n + a)\), where \(n\) is the number of vertices and \(a\) is the number of edges in \(G\). We leave the proof of correctness as an exercise.
13.6 Strongly Connected Components

Let $G = (V, E)$ be a directed graph. We say that $G$ is strongly connected if for each pair of vertices $u$ and $v$, there is a path from $u$ to $v$. For an arbitrary directed graph $G$, let $S \subseteq V$. We say that $S$ is a strongly connected component of $G$ if

- the subgraph of $G$ induced by $S$ is strongly connected; and
- for any subset $S' \subseteq V$, if $S \subseteq S'$ and the subgraph of $G$ induced by $S'$ is strongly connected, then $S' = S$.

Thus, the strongly connected component containing a given vertex $i$ is the set of vertices $j$ such that there are paths from $i$ to $j$ and from $j$ to $i$.

It is easily seen that the strongly connected components of a directed graph $G = (V, E)$ partition the $V$ into disjoint subsets. We wish to design an algorithm to find this partition.

Let us consider a depth-first spanning forest of a directed graph $G$. We will begin by trying to find one strongly connected component. We first note that from any vertex $i$, we can reach all descendants of $i$. Depending on which back edges exist, we may be able to reach some ancestors of $i$. Depending on which cross edges exist, we may be able to reach some other vertices as well; however, we cannot reach any vertices in a tree to the right of the tree containing $i$ (i.e., vertices in a tree that is processed later).

This suggests that we might focus on either the first or the last tree processed. Consider the last tree processed. Let $i$ be the root, and let $j$ be any vertex such that there is a path from $j$ to $i$. Then $j$ must be in the same tree as $i$, and hence must be a descendant of $i$. There is therefore a path from $i$ to $j$. As a result, the strongly connected component containing $i$ is the set of vertices $j$ such that there is a path from $j$ to $i$. We generalize this fact with the following theorem.

**Theorem 13.6** Let $G$ be a directed graph, and let $F$ be a depth-first spanning forest of $G$. Let $S$ be a strongly connected component of $G$, and let $i$ be the vertex in $S$ that is postorder processed last. Let $G'$ be the subgraph of $G$ induced by set of vertices postorder processed no later than $i$. Then $S$ is the set of vertices $j$ such that there is a path from $j$ to $i$ in $G'$.

**Proof:** Clearly, for every vertex $j \in S$, there is a path from $j$ to $i$ that stays entirely within $S$. Because $i$ is postorder processed last of the vertices in $S$, this path stays within $G'$. Therefore, let $j$ be a vertex such that there is a
path from \( j \) to \( i \) in \( G' \). We will show that \( j \in S \). Specifically, we will show that \( j \) is a descendant of \( i \), so that there is a path from \( i \) to \( j \). The proof is by induction on \( n \), the length of the shortest path from \( j \) to \( i \) in \( G' \).

**Base:** \( n = 0 \). Then \( j = i \), so that \( j \) is a descendant of \( i \).

**Induction Hypothesis:** Let \( n > 0 \), and assume that for every \( m < n \), if there is a path of length \( m \) from \( k \) to \( i \) in \( G' \), then \( k \) is a descendant of \( i \).

**Induction Step:** Suppose there is a path of length \( n \) from \( j \) to \( i \) in \( G' \). Let \((j, k)\) be the first edge in this path. Then there is a path of length \( n - 1 \) from \( k \) to \( i \) in \( G' \). From the Induction Hypothesis, \( k \) is a descendant of \( i \).

We now have three cases.

**Case 1:** \((j, k)\) is a back edge. Then \( j \) is clearly a descendant of \( i \).

**Case 2:** \((j, k)\) is either a forward edge or a tree edge. Then \( i \) and \( j \) are both ancestors of \( k \). Because \( j \) is in \( G' \), it can be postorder processed no later than \( i \). Therefore, \( j \) cannot be a proper ancestor of \( i \). \( j \) must therefore be a descendant of \( i \).

**Case 3:** \((j, k)\) is a cross edge. Then \( k \) is postorder processed before \( j \) is. Because \( j \) is postorder processed between \( k \) and \( i \), and because \( k \) is a descendant of \( i \), \( j \) must also be a descendant of \( i \). \( \square \)

The above theorem suggests the following approach to finding the connected components of \( G \). We first do a depth-first search on the entire graph using a postorder VisitCounter post. We then select all of the vertices. To see how we might find an arbitrary strongly-connected component, suppose some of the components have been found and unselected. We find the selected vertex \( i \) that has maximum post.Num\((i)\). We then find all vertices \( j \) from among the selected vertices such that there is a path from \( j \) to \( i \) containing only selected vertices.

We have to be careful at this point because the set of selected vertices may not be exactly the set of vertices that are postorder processed no later than \( i \). Specifically, there may be a vertex \( j \) that belongs to one of the components that have already been found, but which is postorder processed before \( i \). However, Theorem 13.6 tells us that because \( j \) belongs to a different component than \( i \), there is no path from \( j \) to \( i \). Therefore, eliminating such nodes will not interfere with the correct identification of a strongly connected
component. We conclude that the vertices that we find comprise the strongly connected component containing \(i\).

In order to be able to implement this algorithm, we need to be able to find all vertices \(j\) from which \(i\) is reachable via selected vertices. This is almost the same as the reachability problem covered in Section 13.2, except that the edges are now directed, and we must follow the edges in the wrong direction. It is not hard to see that we can use depth-first search to find all vertices reachable from a given vertex \(i\) in a directed graph. In order to be able to use this algorithm to find all vertices \(j\) from which \(i\) is reachable, we must reverse the direction of the edges.

Because \(\text{DfsAll}\) processes all of the edges in the graph, we can use it to build a new graph in which all of the edges have been reversed. In fact, we can use the same depth-first search to record the order of the postorder processing of the vertices. We use three representation variables:

- a readable \text{ListMultigraph} \text{reverse} (recall from Section 9.5 that if we know we will not attempt to add parallel edges, it is more efficient to add edges to a \text{ListMultigraph} and construct a \text{ListGraph} from it);

- a readable array \text{order}[0..n-1]; and

- a natural number \text{loc}.

As we process each edge, we add its reverse to \text{reverse}. As we postorder process each vertex, we add it to \text{order} as we did for topological sort (Figure 13.12). The resulting \text{RevSearcher} is shown in Figure 13.14.

Once \(\text{DfsAll}\) is called with a \text{RevSearcher}, we need to perform a second depth-first search on the entire reversed graph. The \text{Searcher} we need for this search uses a readable array \text{components}[0..n-1] in which it will store values indicating the component to which a given vertex belongs, along with a natural number \text{count} to keep track of the number of strongly connected components completely found. It also includes an operation \text{NextComp}, used to indicate that a strongly connected component has been found completely. Its implementation is shown in Figure 13.15.

Because the second depth-first search must start each tree at a particular vertex, we need to modify \(\text{DfsAll}\) slightly. The resulting algorithm is shown in Figure 13.16. We leave it as an exercise to show that this algorithm runs in \(\Theta(n + a)\) time, where \(n\) is the number of vertices and \(a\) is the number of edges.
CHAPTER 13. DEPTH-FIRST SEARCH

Figure 13.14 RevSearcher implementation of Searcher

RevSearcher\(n\)
\[
\begin{align*}
\text{reverse} & \leftarrow \text{new} \ \text{ListMultigraph}(n) \\
\text{order} & \leftarrow \text{new} \ \text{Array}[0..n - 1]; \ \text{loc} \leftarrow n
\end{align*}
\]

RevSearcher.TreePreProc\(e\)
\[
\text{reverse}.\text{Put}(\text{e}.\text{Dest}(), \text{e}.\text{Source}(), \text{e}.\text{Data}())
\]

RevSearcher.OtherEdgeProc\(e\)
\[
\text{reverse}.\text{Put}(\text{e}.\text{Dest}(), \text{e}.\text{Source}(), \text{e}.\text{Data}())
\]

RevSearcher.PostProc\(i\)
\[
\text{loc} \leftarrow \text{loc} - 1; \ \text{order}[\text{loc}] \leftarrow i
\]

13.7 Summary

Many graph problems can be reduced to depth-first search. In performing the reduction, we focus on a depth-first spanning tree or a depth-first spanning forest. Because a rooted tree is more amenable to the top-down approach than is a graph, algorithmic design is made easier. Furthermore, depth-first spanning trees have structural properties that are often useful in designing graph algorithms.

The implementation of a reduction to depth-first search consists mainly of defining an implementation of the Searcher ADT. This data structure defines what processing will occur at the various stages of the traversal of the depth-first spanning tree. Proofs of correctness can then focus on the traversal, utilizing induction as necessary.

13.8 Exercises

Exercise 13.1 Analyze the worst-case running time of the algorithm PrePostTraverse, shown in Figure 13.2, assuming the tree \(T\) is implemented as a MatrixGraph.
Figure 13.15 SCCSearcher implementation of Searcher

```
SCCSearcher(n)
    components ← new Array[0..n − 1]; count ← 0

SCCSearcher.PreProc(i)
    components[i] ← count

SCCSearcher.NextComp()
    count ← count + 1
```

Figure 13.16 An algorithm for finding strongly connected components in a directed graph

**Precondition:** $G$ refers to a directed Graph.

**Postcondition:** Returns an array $C[0..n − 1]$, where $n$ is the number of vertices in $G$, such that $C[i] = C[j]$ iff $i$ and $j$ belong to the same strongly connected component.

```
StronglyConnComp(G)
    n ← G.Size(); rs ← new RevSearcher(n)
    DfsAll(G, rs)
    order ← rs.Order(); $G' ← new$ ListGraph(rs.Reverse())
    ss ← new SCCSearcher(n); sel ← new Selector(n)
    for $i ← 0$ to $n − 1$
        if sel.IsSelected(order[i])
            Dfs($G'$, order[i], sel, ss); ss.NextComp()
    return ss.Components()
```
Exercise 13.2  Prove that DFSTopSort, shown in Figures 13.12 and 13.13, meets its specification.

Exercise 13.3  Show that STRONGLYCONNCOMP, shown in Figures 13.14, 13.15, and 13.16, runs in $\Theta(n + a)$ time, where $n$ is the number of vertices and $a$ is the number of edges in the given graph, assuming the graph is implemented as a ListGraph.

Exercise 13.4  Prove that STRONGLYCONNCOMP, shown in Figures 13.14, 13.15, and 13.16, meets its specification.

Exercise 13.5  Give an algorithm that decides whether a given directed graph $G$ contains a cycle. Your algorithm should return a boolean value that is true iff $G$ has a cycle. Assuming $G$ is implemented as a ListGraph, your algorithm should run in $O(n + a)$ time, where $n$ is the number of vertices and $a$ is the number of edges in $G$.

Exercise 13.6  A bridge in a connected undirected graph is an edge whose removal disconnects the graph. Give an algorithm that returns a ConsList containing all bridges of a given connected undirected graph. Your algorithm should run in $O(a)$ time in the worst case, where $a$ is the number of edges in the graph, assuming the graph is implemented as a ListGraph.

Exercise 13.7  A connected undirected graph is said to be biconnected if it is impossible to disconnect the graph by removing a single vertex; i.e., it is biconnected iff it has no articulation points. A biconnected component of a connected undirected graph $G$ is a maximal biconnected subgraph $G'$ of $G$ (by “maximal”, we mean that there is no biconnected subgraph of $G$ that contains all of $G'$ plus other vertices and/or edges).

a.  Prove that each edge in a connected undirected graph $G$ belongs to exactly one biconnected component of $G$.

* b.  Give an algorithm to identify the biconnected components of a given connected undirected graph $G$. Specifically, your algorithm should set the data field of each Edge in $G$ to a natural number so that $e.data = f.data$ iff $e$ and $f$ belong to the same biconnected component. Your algorithm should run in $O(a)$ time in the worst case, where $a$ is the number of edges in the graph, assuming the graph is implemented as a ListGraph.
* Exercise 13.8 A directed graph is *semiconnected* if for each pair of vertices $i$ and $j$, there is either a path from $i$ to $j$ or a path from $j$ to $i$. Give an algorithm to decide whether a given directed graph $G$ is semiconnected. Your algorithm should return a boolean that is true iff $G$ is semiconnected. Your algorithm should run in $O(n + a)$ time, where $n$ is the number of vertices and $a$ is the number of edges in $G$.

* Exercise 13.9 An *arborescence* of a directed graph $G = (V, E)$ is a subset $E' \subseteq E$ such that $(V, E')$ is a rooted tree with edges directed from parents to children. Give an algorithm to determine whether a given directed graph $G$ contains an arborescence, and if so, returns one. If $G$ contains an arborescence, your algorithm should return an array $parent[0..n-1]$ such that $parent[i]$ gives the parent of vertex $i$ for all vertices other than the root, and such that $parent[i] = -1$ if $i$ is the root of the arborescence. If $G$ does not contain an arborescence, your algorithm should return nil. You algorithm should operate in $O(n + a)$ time, where $n$ is the number of vertices and $a$ is the number of edges in $G$.

* Exercise 13.10 Give an algorithm that takes a connected undirected graph $G = (V, E)$ as input and produces as output a strongly connected directed graph $G' = (V, E')$ such that

- if $\{i, j\} \in E$, then exactly one of $(i, j)$ and $(j, i)$ is in $E'$; and
- if $\{i, j\} \notin E$, then neither $(i, j)$ nor $(j, i)$ is in $E'$.

Thus, $G'$ is obtained from $G$ by assigning a direction to each edge of $G$. If no such $G'$ exists, your algorithm should return nil. Your algorithm should run in $O(a)$ time, where $a$ is the number of edges in the graph, assuming $G$ is implemented as a ListGraph.

* Exercise 13.11 A directed graph is *singly connected* if for each pair of vertices $i$ and $j$, there is at most one simple path from $i$ to $j$. Give an efficient algorithm to determine whether a given directed graph $G$ is singly connected. Your algorithm should return a boolean that is true iff $G$ is singly connected. Analyze the worst-case running time of your algorithm assuming that $G$ is implemented as a ListGraph.

* Exercise 13.12 A *coloring* of an undirected graph is an assignment of labels to the vertices such that no two adjacent vertices have the same label. A *$k$-coloring* is a coloring that uses no more than $k$ distinct labels. Give
an efficient algorithm to find a 3-coloring for a given connected undirected graph $G$ such that no vertex in $G$ is adjacent to more than 3 vertices, and at least one vertex is adjacent to strictly fewer than 3 vertices (a 3-coloring always exists for such a graph). Your algorithm should run in $O(n)$ time, where $n$ is the number of vertices in $G$, assuming $G$ is implemented as a ListGraph.

**Exercise 13.13** An undirected graph is said to be bipartite if its vertices can be partitioned into two disjoint sets such that no two vertices belonging to the same partition are adjacent. (Note that such a partitioning is a 2-coloring, as defined in Exercise 13.12.) Give an efficient algorithm to find such a partitioning if one exists. Your algorithm should run in $O(n + a)$ time, where $n$ is the number of vertices and $a$ is the number of edges in the graph.

### 13.9 Chapter Notes

The depth-first search technique was developed in the nineteenth century by Trémaux, as reported by Lucas [87]. Its properties were studied by Tarjan [102], who presented an algorithm he credits to Hopcroft for finding articulation points and biconnected components (Exercise 13.7); see also Hopcroft and Tarjan [62]. The algorithm given in Section 13.6 for finding strongly connected components is due to Sharir [97].