SEARCHC(1), setting LOWLINK[1] to MIN(1, LOWLINK[2]) = 1. If we then consider edge (1, 4), we do nothing since (1, 4) is a forward edge and the condition of line 10 of SEARCHC is not met because $4 > 1$.

Next, we call SEARCHC(5), and cross edge (5, 4) causes us to set LOWLINK[5] to 4, since $4 < 5$ and 4 is on STACK. When we again return to SEARCHC(1), we set LOWLINK[1] to the minimum of its former value 1 and LOWLINK[5], which yields 1.

Then, since all edges out of 1 have been considered, and LOWLINK[1] = 1, we discover that 1 is the root of a strongly connected component. This component consists of 1 and all vertices above 1 on the stack. Since 1 was the first vertex visited, vertices 2, 3, 4, and 5 are all above 1, in that order. Thus the stack is emptied and the list of vertices 1, 2, 3, 4, 5 is printed as a strongly connected component of G.

The remaining strongly connected components are {7} and {6, 8}. We leave it to the reader to complete the calculation of LOWLINK and the strongly connected components, starting at vertex 6. Note that the roots of the strongly connected components were last encountered in the order 1, 7, 6. □

Theorem 5.4. Algorithm 5.4 correctly finds the strongly connected components of G in $O(\text{MAX}(n, e))$ time on an n-vertex, e-edge directed graph G.

Proof. It is easy to check that the time spent by one call of SEARCHC(v), exclusive of recursive calls to SEARCHC, is a constant plus time proportional to the number of edges leaving vertex v. Thus all calls to SEARCHC together require time proportional to the number of vertices plus the number of edges, as SEARCHC is called only once at any vertex. The portions of Algorithm 5.4 other than SEARCHC can clearly be implemented in time $O(n)$. Thus the time bound is proven.

To prove correctness it suffices to prove, by induction on the number of calls to SEARCHC that have terminated, that when SEARCH(v) terminates, v is LOWLINK[v] is correctly computed. By lines 12–16 of SEARCHC, v is made the root of a strongly connected component if and only if we have LOWLINK[v] = v. Moreover, the vertices printed out are exactly those descendants of v which are not in components whose roots have been found before v, as required by Lemma 5.8. That is, the vertices above v on the stack are descendants of v, and their roots have not been found before v since they are still on the stack.

To prove LOWLINK is computed correctly, note that there are two places in Fig. 5.15 where LOWLINK[v] could receive a value less than v, that is, at lines 9 and 11 of SEARCHC. In the first case, w is a son of v, and LOWLINK[w] $< v$. Then there is a vertex x = LOWLINK[w] that can be reached from a descendant y of w by a cross or back edge. Moreover, the root r of the strongly connected component containing x is an ancestor of w. Since $x < v$, we have $r < v$, so r is a proper ancestor of v. Thus we see LOWLINK[v] should be at least as small as LOWLINK[w].

In the second case, at line 11, there is a cross or back edge from v to a vertex $w < v$ whose strongly connected component C has not yet been found. The call of SEARCHC on the root r of C has not terminated, so r must be an ancestor of v. (Since $r \leq w < v$, either r is to the left of v or r is an ancestor of v. But if r were to the left of v, SEARCHC(r) would have terminated.) Again it follows that LOWLINK[v] should be at least as low as w.

We must still show that SEARCHC computes LOWLINK[v] to be as low as it should be. Suppose in contradiction that there is a descendant x of v with a cross or back edge from x to y, and the root r of the strongly connected component containing y is an ancestor of v. We must show that LOWLINK is set at least as low as y.

CASE 1. $x = v$. We may assume by the inductive hypothesis and Lemma 5.9 that all strongly connected components found so far are correct. Then y must still be on STACK, since SEARCH(r) has not terminated. Thus line 11 sets LOWLINK[v] to y or lower.

CASE 2. $x \neq v$. Let z be the son of v of which x is a descendant. Then by the inductive hypothesis, when SEARCHC(z) terminates, LOWLINK[z] has been set to y or lower. At line 9 LOWLINK[v] is set this low, if it is not already lower. □

5.6 PATH-FINDING PROBLEMS

In this section we consider two frequently occurring problems having to do with paths between vertices. In what follows let G be a directed graph. The graph G^* which has the same vertex set as G, but has an edge from v to w if and only if there is a path (of length 0 or more) from v to w in G, is called the (*reflexive* and) *transitive closure* of G.

A problem closely related to finding the transitive closure of a graph is the *shortest-path problem*. Associate with each edge e of G a nonnegative *cost* $c(e)$. The *cost of a path* is defined to be the sum of the costs of the edges in the path. The shortest-path problem is to find for each ordered pair of vertices (v, w) the lowest cost of any path from v to w.

It turns out that the ideas behind the best algorithms known for both the transitive closure and shortest-path problems are (easy) special cases of the problem of finding the (infinite) set of all paths between each pair of vertices. To discuss the problem in its generality, we introduce a special algebraic structure.

Definition. A *closed semiring* is a system $(S, +, \cdot, 0, 1)$, where S is a set of elements, and $+$ and $\cdot$ are binary operations on S, satisfying the follow-