Space-Efficient Implementations of Graph Search Methods

ROBERT E. TARJAN
Bell Laboratories

Several space-efficient implementations of the two most common and useful kinds of graph search, namely, breadth-first search and depth-first search, are discussed. A straightforward implementation of each method requires n bits and $n + O(1)$ pointers of auxiliary storage, where n is the number of vertices in the graph. We devise methods that need only $2n + m$ bits, of which m are read-only, where m is the number of edges in the graph. We save space by folding the queue or stack required by the search into the graph representation; two of our methods for depth-first search are variants of the Deutsch-Schorr-Waite list-marking algorithm. Our algorithms are expressed in a version of Dijkstra's guarded command language.

Categories and Subject Descriptors: E.1 [**Data**]: Data Structures—*graphs*; E.2 [**Data**]: Data Storage Representations—*linked representations*, F.2.2 [**Analysis of Algorithms and Problem Complexity**]: Nonnumerical Algorithms and Problems—*computations of discrete functions*; G 2 2 [**Discrete Mathematics**]: Graph theory—*graph algorithms*

General Terms. Algorithms, Theory

Additional Key Words and Phrases: Depth-first search, breadth-first search, list marking

1. INTRODUCTION

In this paper we discuss several space-efficient implementations of the two most common and useful kinds of graph search, breadth-first search and depth-first search [1, 6]. We assume that the reader is familiar with these search methods. We restrict our attention to directed graphs, although our methods extend easily to undirected graphs. Our implementations are generic; we invite the reader to tailor them to his or her own applications.

If e is a directed edge, we denote its end vertices by $h(e)$, the *head* of e, and $t(e)$, the *tail* of e; e is directed from head to tail. We assume that the graph to be searched is represented by a list structure, as illustrated in Figure 1. The representation contains one node for each vertex and one for each edge; we generally do not distinguish between a node and the vertex or edge it represents. Each edge has a pointer to its tail. The edges out of a vertex v form a single list with header v and links stored in a field a (for "after"). The last edge on the list points to a special node **null**. This representation does not store heads with edges.

Author's address. Bell Laboratories, Murray Hill, NJ 07974.

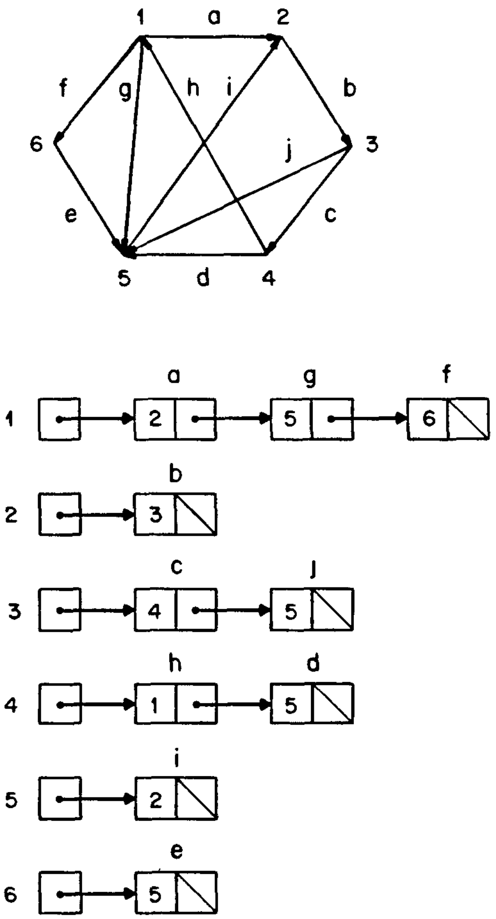

Fig. 1. Representation of a graph by single lists of outgoing edges.

Thus we require of any search implementation that the head of an edge e be accessible in $O(1)$ time (for use in processing e) when e is traversed during the search. When stating time and space bounds we use n and m to denote the number of vertices and edges in the graph, respectively.

For the purpose of labeling vertices as they are visited, we assume that all vertices are initially unlabeled and that the operation *label*(v) can be used during the search to label vertex v. For checking whether a vertex is labeled, we assume the existence of a predicate *labeled* defined by *labeled*(v) = "v is labeled." The easiest way to implement vertex labeling is to use a bit *labeled*(v), initially **false**, for each vertex v; to label v, we make its bit **true**:

```
procedure label(vertex v);
  labeled(v) := true
end label;
```

In many applications of graph search, the vertices are labeled with numbers as they are visited; in such cases the numbers can serve to identify labeled vertices, and an extra label bit is unnecessary.

2. BREADTH-FIRST SEARCH

During a breadth-first search, in addition to being either unlabeled or labeled, a vertex is either *unscanned* or *scanned.* Each scanned vertex is also labeled; thus there are only three possible vertex states: unlabeled (and unscanned), labeled but unscanned, and scanned (and labeled). Initially the start vertex of the search is labeled but unscanned and every other vertex is unlabeled. The search consists of repeating the following step until no vertex is labeled but unscanned.

Search Step. Let v be the least-recently labeled vertex that is not yet scanned. Scan v by traversing each edge e out of v. When traversing an edge e, label $t(e)$ if it is unlabeled.

The edges e such that $t(e)$ was unlabeled when e was traversed define a tree rooted at the start vertex, spanning all initially unlabeled vertices reachable from the start vertex. (See Figure 2.) This is true of any kind of graph search, not just breadth-first.

The natural way to implement breadth-first search is to maintain a queue of the labeled but unscanned vertices. We call this the *queue version* of breadth-first search. Procedure *bfs* below is an implementation of breadth-first search written in a variant of Dijkstra's guarded command language [2], with a vertical line | substituted for Dijkstra's box $\square$. Input to the procedure is *start,* the start vertex of the search. The procedure uses three list-manipulating operations, which we denote as follows:

Access. If $q = [x_1, x_2, \ldots, x_n]$ is a list and i is an integer, $q(i) = x_i$ if $i \in [1..n]$, $q(i) =$ **null** otherwise.

Sublist. If $q = [x_1, x_2, \ldots, x_n]$ is a list and i and j are integers, $q[i..j]$ is the list $[x_i, x_{i+1}, \ldots, x_j]$. If i is missing or less than 1, it has an implied value of 1. Similarly, if j is missing or greater than n, it has an implied value of n. Thus, for example, $q[2..] = [x_2, x_3, \ldots, x_n]$. If $i > j$, $q[i..j]$ is the empty list, which we denote by [ ].

Concatenation. If $q = [x_1, x_2, \ldots, x_n]$ and $r = [y_1, y_2, \ldots, y_m]$ are two lists, their concatenation is

$$q \,\&\, r = [x_1, x_2, \ldots, x_n, y_1, y_2, \ldots, y_m].$$

To carry out a search using the breadth-first search procedure, we need only define *label, scan,* and *traverse* to perform whatever computations are desired. The procedure maintains the invariant that queue q is a list of the labeled but unscanned vertices in the order they were labeled.

```
   procedure bfs(vertex start);
      vertex v; edge e; queue q;
      label(start);
      q = [start];
      do q ≠ [ ] →
1.       v, q := q(1), q[2..];
         e := a(v);
         do e ≠ null →
            traverse(e);
            if not labeled(t(e)) →
2.                label(t(e)); q := q & [t(e)]
            fi;
```

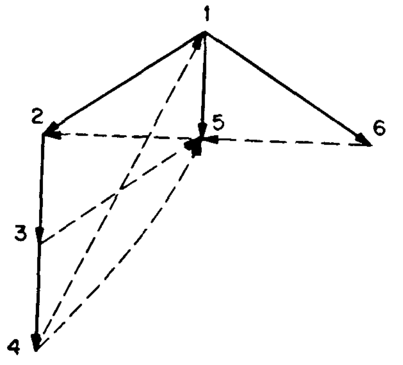

Fig. 2. Spanning tree generated by a breadth-first search of graph. Vertices are labeled in the order 1, 2, 5, 6, 3, 4. Solid edges are tree edges; dashed edges are nontree edges.

```
            $e := a(e)$
        od;
        $scan(v)$
    od
end $bfs$;
```

Remark 1. The statement in line 1 is a parallel assignment that simultaneously redefines v and q.

Remark 2. Procedure *bfs* applies *scan* to the vertices in the same order as it applies *label.*

The redefinitions of q on lines 1 and 2 destroy the old value of q. Thus there is no need for list copying, and each of the queue operations takes $O(1)$ time with a proper implementation. For this purpose we can represent q either by an array of (at least) $n - 1$ positions or by a singly linked list of vertices. In either case, the running time of the search is $O(n + m)$. The storage overhead for the search (not counting storage for the graph) is $n + O(1)$ pointers and n label bits.

We can eliminate the extra storage for the queue by using **null** pointers at the end of the edge lists to fold the queue into the graph representation. To do this, we maintain a queue consisting of a single list linked by a pointers containing every labeled vertex followed by its outgoing edges. (See Figure 3.) After labeling a vertex, we add it to the end of the queue; this automatically adds its outgoing edges to the queue. After scanning a vertex, we reset the a pointer of its last outgoing edge to **null.**

This method requires the ability to distinguish between a node representing a vertex and a node representing an edge. We assume the existence of a predicate *edge* for this purpose, defined by $edge(x)$ = "x represents an edge." We can implement this predicate using a read-only bit in each node. Alternatively, if nodes have integer addresses lying in separable intervals for vertices and edges, we can test that the address lies in the appropriate range, and we need no extra storage.

Our implementation of this method, as follows, uses three local variables; v, the vertex being scanned; e, the node just examined in the list; and f, the last node on the list. We call this method the *list version* of breadth-first search.

```
procedure $bfs$(vertex $start$);
    vertex $v$; node $e, f$;
```

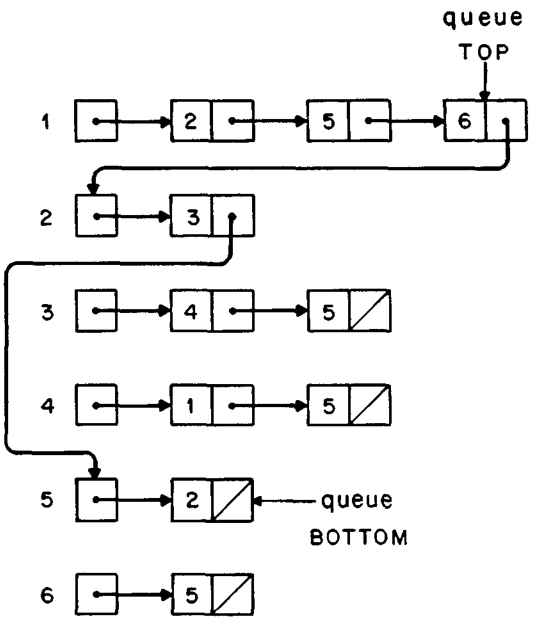

Fig. 3. Representation of graph after labeling 1, 2, and 5 during space-efficient breadth-first search.

```
f := v := start;
label(start);
do a(f) ≠ null → f := a(f) od;
do v ≠ null →
  e := v;
  do a(e) ≠ null and edge(a(e)) →
    e := a(e);
    traverse(e);
    if not labeled(t(e)) →
      f := a(f) := t(e);
      label(f);
      do a(f) ≠ null → f := a(f) od
    fi
  od;
  scan(v);
  v := a(e); a(e) := null
od
end bfs;
```

Remark. The statement "*f* := *v* := *start*" in this procedure is a sequential assignment of *s* to *v* and then to *f*.

This procedure scans each edge list twice, the first time merely to update *f*. We can avoid this initial scan if the edge lists are represented as single rings instead of single lists. In this representation each vertex points to the last edge on its list, which in turn points to the first edge on the list. (See Figure 4a.) Our third and last version of breadth-first search, which we call the *ring version*, uses this representation. After labeling a vertex *w*, we break the ring of edges out of *w* between its last and first node, add *w* and its outgoing edges to the end of the queue, and update *f*. After scanning a vertex, we repair its edge ring. (See Figure 4b). The following procedure implements this method. The procedure is complicated by the fact that empty rings require special handling.

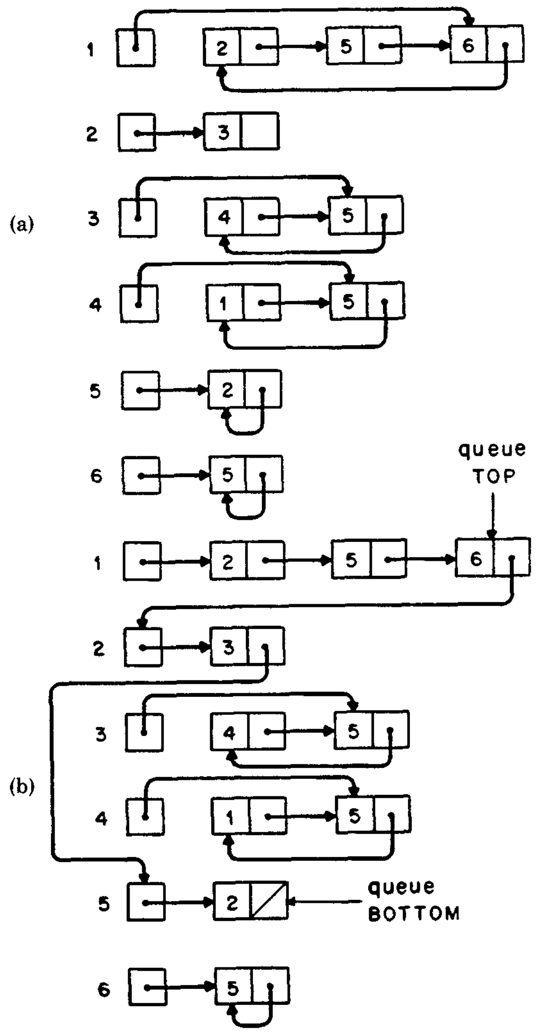

Fig. 4. (a) Representation of graph by rings of outgoing edges. (b) Representation after labeling 1, 2, and 5 during space-efficient breadth-first search.

```
procedure bfs(vertex start);
  node v, e, f;
  v := start;
  label(start);
  if a(start) = null → f := start
  | a(start) ≠ null → f := a(start); a(start) := a(f); a(f) := null
  fi;
  do v ≠ null →
    e := v;
    do edge(a(e)) →
      e := a(e);
      traverse(e):
      if not labeled(t(e)) →
        label(t(e));
        a(f) := t(e);
        if a(t(e)) = null → f := t(e)
        | a(t(e)) ≠ null → f := a(t(e)); a(t(e)) := a(f); a(f) := null
```

```
            fi
        fi
    od;
    scan(v);
    if v = e → v := a(e); a(e) := null
    | v ≠ e → v, a(v), a(e) := a(e), e, a(v)
    fi
    od
end bfs;
```

Whether this or the list version of *bfs* runs faster depends upon the initial representation of the graph and the lengths of the edge lists. Since converting from the single list to the ring representation requires a scan of each edge list, it does not pay to perform this conversion for a single breadth-first search. For general use we prefer the list version of *bfs*, since its code is more transparent and it will run at best faster and at worst slightly slower than the ring version. If space is not a problem, some implementation of the queue version is best.

3. DEPTH-FIRST SEARCH

We can define depth-first search recursively as follows: A depth-first search from a vertex v consists of labeling v and scanning v. To scan v, we traverse each edge out of v. To traverse an edge e, we test whether $t(e)$ is labeled. If $t(e)$ is not labeled, e is a tree edge: we advance along e, perform a depth-first search from $t(e)$, and retreat along e. If $t(e)$ is labeled, e is a *nontree* edge: we advance along e and immediately retreat along e. (See Figure 5.)

Procedure *dfs* below implements this definition. Input to the procedure is v, the start vertex of the search. To use *dfs*, we need only define the procedures that visit vertices and advance and retreat along edges so that they perform the desired computations. For edge processing that is independent of the edge type (tree or nontree), we can use *advance* and *retreat*; for edge processing that depends upon the edge type, we can use *advance tree* and *retreat tree, advance nontree* and *retreat nontree.* Normally we would define either *advance* or the pair *advance tree, advance nontree* to be empty procedures, and similarly either *retreat* or the pair *retreat tree, retreat nontree* to be empty.

```
procedure dfs(vertex v);
  edge e;
  label(v);
  e := a(v);
  do e ≠ null →
    advance(e);
    if not labeled(t(e)) →
        advance tree(e); dfs(t(e)); retreat tree(e)
     | labeled(t(e)) →
        advance nontree(e); retreat nontree(e)
    fi;
    retreat(e);
    e := a(e)
  od;
  scan(v)
end dfs;
```

Remark 1. We have included in *dfs* separate procedures for processing tree and nontree edges because in many if not most applications, the processing is

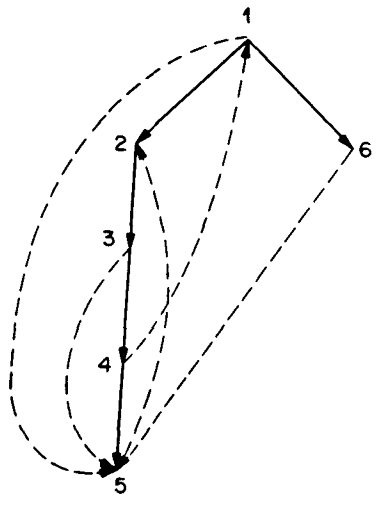

Fig. 5. Spanning tree generated by a depth-first search of graph

different for the two kinds of edges. It is easy to insert such procedures into *bfs* as well.

Remark 2. Procedure *dfs* applies *label* to the vertices in preorder, with respect to the spanning tree generated by the search, and applies *scan* to the vertices in postorder.

A depth-first search takes $O(n + m)$ time and $O(n)$ space, not counting space for the graph. Our first step in improving the space efficiency of our implementation is to remove the recursion. Procedure *dfs* saves two variables per recursive call: v, the vertex being scanned, and e, the edge being traversed. However, there is no need to save v: if v is not the start vertex, the tree edge entering v was saved just before *dfs* was called on v, and v is available as the tail of this edge. Thus we can carry out the search using a single stack containing the edges on the tree path from the start vertex to the vertex being scanned.

More precisely, the search uses two variables, a stack of edges s and a current edge e, and maintains the following invariant: if $s = [e_1, e_2, \ldots, e_k]$, then $e_k, e_{k-1}, \ldots, e_1$ is the path of tree edges from the start vertex (*start*) to the vertex v being scanned; if $v =$ *start*, $s = [\,]$. The edges on the stack are exactly those along which an advance but not a retreat has taken place. The edge e is either the first edge out of v along which an advance has not yet occurred or **null** if the search has advanced along every edge out of v. A step of the search consists of processing the edge e. There are three cases, depending on whether e is a tree edge, a nontree edge, or **null**. The following procedure implements the search:

```
procedure dfs(vertex start);
  vertex v; edge e; stack s;
  s := [ ];
  label(start);
  v := start;
  e := a(start);
  do e ≠ null and not labeled(t(e))
     advance tree(e)
     s := [e] & s;
     label(t(e));
```

```
        v := t(e);
        e := a(t(e))
    | e ≠ null and labeled(t(e)) →
        advance nontree(e);
        retreat nontree(e);
        e := a(e)
    | e ≠ null and s ≠ [ ] →
        e, s := s(1), s[2 . . ];
        scan(t(e));
        v := if s = [ ] → start | s ≠ [ ] → t(s(1)) fi;
        retreat tree(e);
        e := a(e)
  od;
  scan(start)
end dfs;
```

Remark. Variable v, the vertex being scanned, is never used explicitly in the search. If v is not used by any of the advance or retreat procedures (v is the head of the edge along which the advance or retreat takes place), its computation can be dropped from the program.

There are three reasonable ways to implement the stack in this procedure. The two obvious methods are to use an array of (at least) $n - 1$ positions or to use a single list of edges. The latter representation needs one pointer per edge, for a total of m, of which at most $n - 1$ are actually used. We obtain a linked representation that needs only n pointers, one per vertex, by storing with each vertex the tree edge entering its parent. A search using this third representation has the beneficial side effect of constructing the spanning tree. The following version of *dfs* uses this representation. Local variable f is the tree edge entering the vertex being scanned; if this is the start vertex, f is **null**. For each vertex v, the procedure stores the tree edge entering the parent of v in the field $b(v)$.

```
procedure dfs (vertex start);
  vertex v; edge e, f;
  f := null
  label(start);
  v := start;
  e := a(start);
  do e ≠ null and not labeled(t(e)) →
        advance tree(e);
        label(t(e));
        b(t(e)), f, v, e := f, e, t(e), a(t(e))
    | e ≠ null and labeled(t(e)) →
        advance nontree(e);
        retreat nontree(e);
        e := a(e)
    | e = null and f ≠ null →
        e, f := f, b(t(f));
        scan(t(e));
        v := if f = null → start | f ≠ null → t(f) fi;
        retreat tree(e);
        e := a(e)
  od;
  scan(start)
end dfs;
```

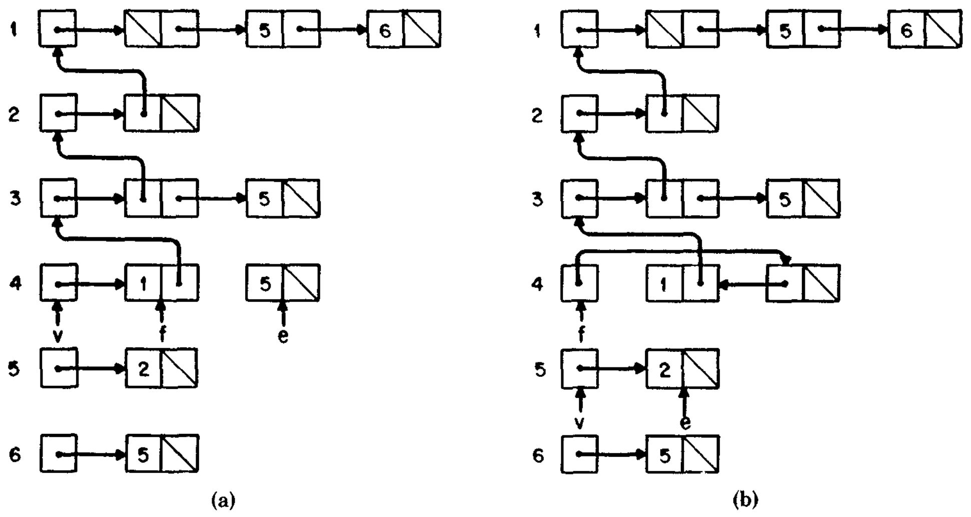

Fig. 6 (a) Graph representation during space-efficient depth-first search just before advancing along tree edge from vertex 4 to vertex 5. (b) Representation after advancing along tree edge from 4 to 5.

Remark. As is the previous version of *dfs*, this version is written so that variable v is not needed for the search, and its computation can be dropped if none of the advance/retreat procedures use it.

Our next step in saving space is to fold the stack into the graph representation. We do this by extending the Deutsch-Schorr-Waite algorithm for marking a list structure [3, 4, 5]. As in the case of breadth-first search, we need the ability to distinguish between nodes representing vertices and nodes representing edges; we use the same predicate *edge*. The method uses three local variables: v, the vertex being scanned; e, the edge being traversed; and f, the *trailer*, defined as follows: if e is not the first edge out of v, then f is the edge before e out of v; otherwise, f is the parent of v in the spanning tree if v has a parent, **null** if v is the start vertex. Initially v, e, and f are *start*, $a(start)$, and **null**, respectively. The search proceeds in the same way as in the previous version of *dfs*, except that it modifies the graph representation as it goes. During the search, just before advancing along an edge e, the edges before e out of v are linked in reverse order via a pointers; f is the first edge on the reversed list, and the last edge (originally the first edge out of v) points to the parent of v. (See Figure 6a.) We say more about the representation below.

To carry out the search, we initialize v, e, and f, perform $label(v)$, repeat the following step until e = **null** and v = *start*, and then perform some postprocessing (described below), which includes scanning the start vertex:

General Step. Determine which one of the following three cases applies and perform the actions listed:

Case 1: *Advance along a tree edge.* If $e \neq$ **null** and $t(e)$ is unlabeled:
Advance along tree edge e. Redefine $a(v)$ to be e. Simultaneously redefine $t(e)$ to be f, f to be v, and v to be $t(e)$. Perform $label(v)$. Redefine e to be $a(v)$. Now

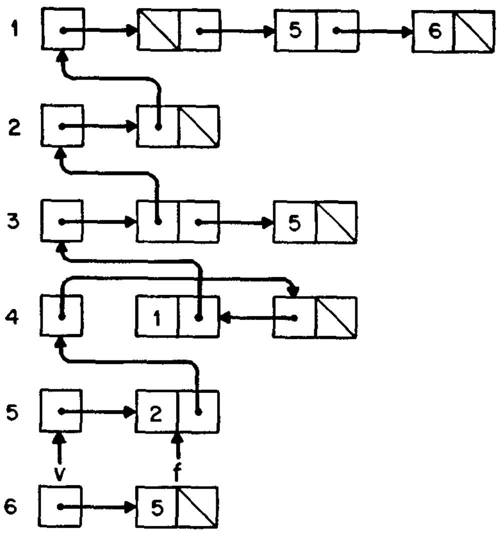

Fig. 7. Representation after traversing nontree edge from 5 to 2. Variable e = null.

v is the new vertex to be scanned, e is the first edge out of v, f is the parent of v, $a(f)$ is the tree edge entering v, and $t(a(f))$ is the edge before $a(f)$ out of f (or the parent of f if there is no such edge). (See Figure 6.)

Case 2: *Traverse a nontree edge.* If $e \neq$ **null** and $t(e)$ is labeled:
Advance and retreat along nontree edge e. Simultaneously redefine $a(e)$ to be f, f to be e, and e to be $a(e)$. Now e is the next edge out of v. (See Figures 6b and 7.)

Case 3: *Retreat along a tree edge.* If e = **null** and $v \neq$ *start*:
The list of edges out of v is now completely reversed. (See Figure 8a.) Restore it to its original form by repeating the following step until f is not an edge: simultaneously redefine $a(f)$ to be e, e to be f, and f to be $a(f)$. Now f is the parent of v and e is the first edge out of v (or **null**). Redefine $a(v)$ to be e. This completes the restoration of the list of edges out of v. (See Figure 8b.)

Continue the step by performing *scan*(v). Redefine e to be $a(f)$. Simultaneously redefine v to be f, f to be $t(e)$, and $t(e)$ to be v. This restores the representation to its state before advancing along e. (See Figure 8c.) Retreat along tree edge e. Simultaneously redefine $a(e)$ to be f, f to be e, and e to be $a(e)$. Now e is the next edge out of v. (See Figure 8d.)

When e = **null** and v = *start*, we complete the search by restoring the list of edges out of *start* and then scanning *start*. The following procedure implements this method:

```
procedure dfs(vertex start);
  vertex v;
  f := null;
  label(start);
  v := start;
  e := a(v);
  do e ≠ null and not labeled(t(e)) →
    advance tree(e);
```

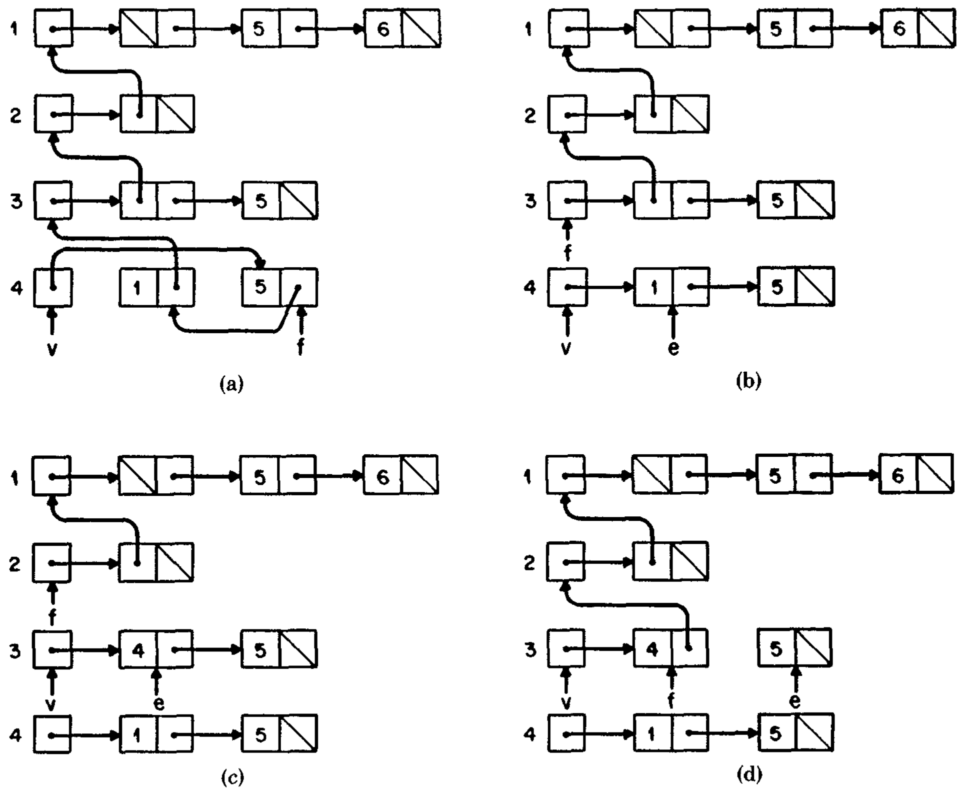

Fig. 8. (a) Representation before retreating along tree edge from 3 to 4. (Edge lists for 5 and 6 are deleted for clarity.) (b) Representation when scanning 4. (c) Representation when retreating. (d) Representation before advancing along next edge. Variable e = **null**.

```
        $a(v) := e$;
        $v, f, t(e) := t(e), v, f$;
        $label(v)$;
        $e := a(v)$
    | $e \neq$ **null and** $labeled(t(e)) \rightarrow$
        $advance\ nontree(e)$;
        $retreat\ nontree(e)$;
        $e, f, a(e) := a(e), e, f$
    | $e =$ **null and** $v \neq start \rightarrow$
        **do** $edge(f) \rightarrow e, f, a(f) := f, a(f), e$ **od**;
        $a(v) := e$;
        $scan(v)$;
        $e := a(f)$;
        $v, f, t(e) := f, t(e), v$;
        $retreat\ tree(e)$;
        $e, f, a(e) := a(e), e, f$
    **od**;
    **do** $f \neq$ **null** $\rightarrow e, f, a(f) := f, a(f), e$ **od**;
    $a\ (start) := e$;
    $scan(start)$
**end** $dfs$;
```

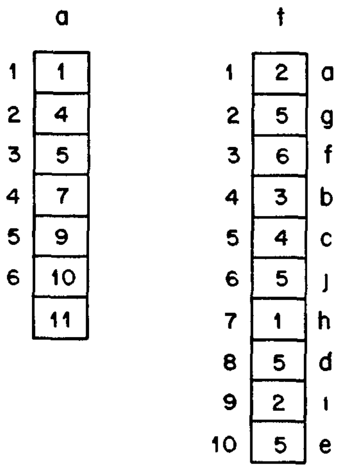

Fig. 9. Array representation of graph

Remark. In this version of *dfs*, vertex v is needed for the search.

This version of depth-first search requires the same space as the list or ring version of breadth-first search, namely, a label bit per vertex (in cases where no vertex numbering takes place) and a read-only bit per node (if vertex nodes and edge nodes can be distinguished in no other way). Like the list version of breadth-first search, it scans each edge list twice; in this case, the second scan is needed to restore the list to its original form. We can avoid the second scan by changing the graph representation, as we discuss below. However, the second scan does provide an opportunity to process the edge lists as the vertices are postvisited, which can be used to simplify some algorithms that use depth-first search.

Eliminating the second scan seems to require an array representation of the graph. (See Figure 9.) Each edge is a position in an array t; $t(e)$ is the tail of edge e. Edges with the same head are grouped together; the set of edges out of a vertex v is $\{e \mid a(v) \le e < a(v+1)\}$. We use $out(v) = a(v+1) - a(v)$ to denote the number of edges out of v.

To carry out the search, we use the method suggested by Gries [3, p. 231, note 5]. The auxiliary storage needed is an increment $i(v)$ for each vertex v; $i(v)$ is an integer between 0 and $out(v)$ (inclusive). A zero increment denotes an unlabeled vertex; a positive increment denotes a labeled vertex. We use three local variables: v, the vertex being scanned; e, the edge being traversed; and u, the parent of v. After advancing along a tree edge e, we store $e - a(v) + 1$ in $i(v)$ and simultaneously replace v, u, and $t(e)$ by $t(e)$, v, and u, respectively. When we finish the search from the new vertex v, we restore the situation by defining e to be $a(u) + i(u) - 1$ and simultaneously replacing v, u, and $t(e)$ by u, $t(e)$, and v, respectively. Then we retreat along e. The following procedure gives the details:

```
procedure dfs(vertex start);
    vertex u, v; edge e;
    u := null;
    label(start);
    (i(start) := 1;)
    v := start;
    e := a(v);
```

```
  **do** $e < a(v + 1)$ **and** $i(t(e)) = 0 \rightarrow$
      *advance tree*$(e)$;
      $i(v) := e - a(v) + 1$;
      $v, u, t(e) := t(e), v, u$;
      *label*$(v)$;
      $(i(v) := 1;)$
      $e := a(v)$
  | $e < a(v + 1)$ **and** $i(t(e)) \neq 0 \rightarrow$
      *advance nontree*$(e)$;
      *retreat nontree*$(e)$;
      $e := e + 1$
  | $e = a(v + 1)$ **and** $v \neq start \rightarrow$
      *scan*$(v)$;
      $e := a(u) + i(u) - 1$;
      $v, u, t(e) := u, t(e), f$;
      *retreat tree*$(e)$;
      $e := e + 1$
  **od**;
  *scan*$(start)$
**end** *dfs*;
```

Since vertex labeling is done by updating increments, the operation *label*(v) need only perform whatever calculations are needed by the application. The two statements "$i(v) := 1$" are necessary only if the graph contains loops; if not, the statement "$i(v) := e - a(v) + 1$" provides all the increment updating that is necessary.

We have presented a total of five versions of depth-first search. As with breadth-first search, the best choice depends on the intended application. We encourage the reader to experiment with these methods and adapt them to his or her own needs.

ACKNOWLEDGMENT

My thanks to David Gries for an exceptionally thorough refereeing job that greatly improved this paper.

REFERENCES

1. AHO, A.V., HOPCROFT, J.E., AND ULLMAN, J.D. *The Design and Analysis of Computer Algorithms*. Addison-Wesley, Reading, Mass., 1974.
2. DIJKSTRA, E.W. *A Discipline of Programming*. Prentice-Hall, Englewood Cliffs, N.J., 1976.
3. GRIES, D. The Schorr-Waite graph marking algorithm. *Acta Inf. 11* (1979), 223-232.
4. KNUTH, D.E. *The Art of Computer Programming*, vol. 1, *Fundamental Algorithms*, 2nd ed. Addison-Wesley, Reading, Mass., 1973.
5. SCHORR, H., AND WAITE, W.M. An efficient machine-independent procedure for garbage collection in various list structures. *Commun. ACM 10*, 8 (Aug. 1967), 501-506.
6. THORELLI, L.-E. Marking algorithms. *BIT 12* (1972), 555-568.

Received August 1982; accepted March 1983