

Chapter 5

■ GRAPH TRAVERSAL ALGORITHMS

5.1 INTRODUCTION

Graphs or Networks are one of the most useful structures in science and mathematics. A graph $G = (N, A)$ is a set N of *nodes* and A is a set of ordered pairs of nodes called *arcs*. $|N| = n$ and $|A| = m$. A graph is *undirected* if for each arc $(u, v) \in A$ there is an arc $(v, u) \in A$. An undirected graph is *connected* if there is a *path* between every pair of arcs. A *weighted* graph is a graph with a weight or number associated with each arc. A graph with more than one arc between a pair of nodes is a *multi-graph*. A *spanning tree* of connected graph G is the sub-graph $T = (N, A')$ where $A' \subset A$ and T is a tree. A graph of n nodes can have m between 0 and n^2 . A graph with n nodes is called *dense* if it has $O(n^2)$ arcs, and *sparse* if it has $O(n)$ arcs.

Some Graph Operations. The following are some general operations :

- Traverse each arc.
- Visit each node.
- Find a path between two nodes.
- Find cycles.
- Find special nodes or arcs.
- Find special subgraphs, e.g., spanning trees.

The basic operation used to construct a graph is $CreateDArc(u, v, G)$. This creates a directed arc between nodes u and v , according to the representation used to store the graph. Applying this operation to each arc in G constructs the graph, as shown below

```
algorithm CreateGraph(A:stream of arcs)
while NOT EoS(A) do
    (u,v) := Get(A)
    CreateDArc(u,v,G)
endwhile
return(G)
endalg CreateGraph
```

5.2 REPRESENTATION OF GRAPHS

Graphs, like trees, have many different storage representations. Furthermore, because any tree is a special graph, any graph representation can be used to represent a tree.

Adjacency Matrix A . Let the $n \times n$ matrix A be defined as

$$a_{uv} = \begin{cases} 1, & \text{if there is a directed arc from } u \text{ to } v; \\ 0, & \text{otherwise.} \end{cases}$$

We can represent a *weighted graph* $G = (N, A, d)$ as follows :

$$a_{uv} = \begin{cases} d_{uv}, & \text{if there is a directed arc from } u \text{ to } v; \\ \infty, & \text{otherwise.} \end{cases}$$

If the graph is undirected then A is *symmetric*, i.e., $a_{uv} = a_{vu}$.

This representation requires $O(n^2)$ space and is wasteful if G is sparse. Most operations are inefficient – usually $O(n^2)$.

Node-Arc Incidence Matrix B . This is an $n \times m$ matrix such that

$$b_{ua} = \begin{cases} 1, & \text{if there is a directed arc } a \text{ into } u; \\ -1, & \text{if there is a directed arc } a \text{ out of } u; \\ 0, & \text{otherwise.} \end{cases}$$

This $O(mn)$ space representation is very wasteful of storage and graph operations on it are difficult. However, it arises naturally in *Linear Programming* formulations of network problems and, as such, has many interesting theoretical properties. For example, the flow conservation constraints in the LP formulation of the *Minimum Cost Flow Problem* form a node-arc incidence matrix.

List of Arcs. The graph is represented as a list of triples :

$$G = \{(u, v, d_{uv})\}, \quad \forall (u, v) \in A.$$

The graph below would be represented as follows :

$$G = ((A, B, 15), (A, C, 17), (B, D, 53), (C, D, 20), (C, E, 21), (C, F, 19), (D, F, 81))$$

We need a special triple for the unconnected node Z . Something like $(Z, \text{null}, \text{null})$ would do, but $(Z, Z, 0)$ is more consistent because it is not a special case.

Undirected graphs, strictly speaking, should be represented by each arc (u, v, d_{uv}) appearing twice : once in the forward direction and once in the reverse direction. This is not necessary if we include a ‘directed/undirected’ indicator variable at the start of the list.

This is the most space-efficient representation and requires $O(m)$ space. Most operations are very difficult. This representation is good for external storage and graph construction. Some algorithms, e.g., *Kruskal’s MST* algorithm, expect the graph to be given in this form.

Adjacency List. An array $G[1..n]$ stores pointers to lists of adjacent nodes, i.e., $G[i]$ points to the list of nodes adjacent to i . An element on $Node[i]$ contains the location of an adjacent node in array $D[1..n]$. Undirected arcs are represented as two oppositely-directed arcs. This is a very efficient and flexible and requires $O(n + 2m)$ space. Figure 5.1 shows this representation.

Sparse Adjacency Matrix Representations.

Forward Star

Backward Star

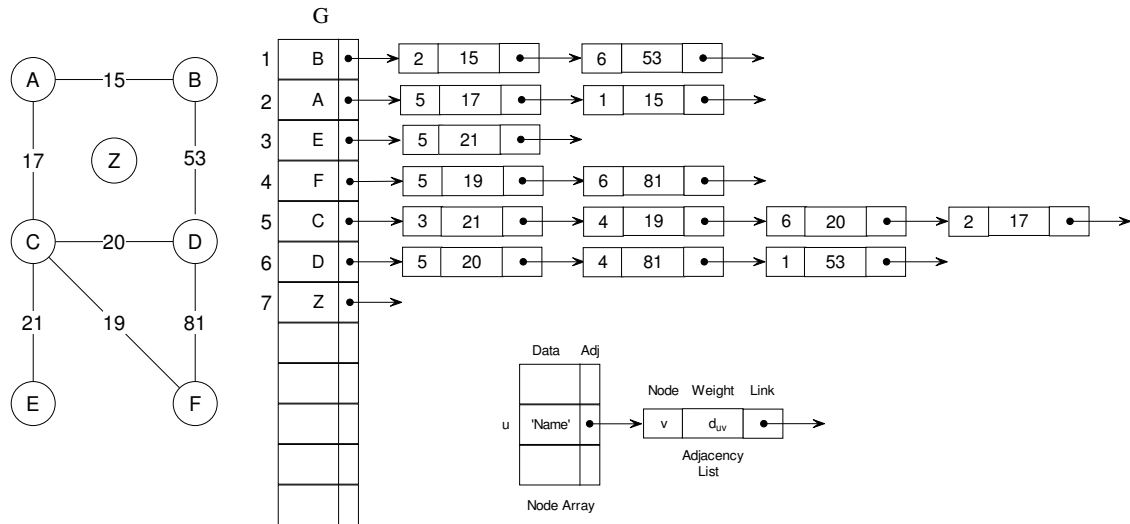


Figure 5.1. Adjacency List Graph Representation.

5.3 TRAVERSAL ALGORITHMS

In the algorithms that follow we will assume that the graph $G = (N, A, d)$ is represented by an array $G[1..n]$ of adjacency lists. We will denote the adjacency list of node u as $Adj(u)$. If the graph is weighted then the weight $d(u, v)$ is stored along with v on $Adj(u)$. See Figure 6.1 for details.

The solution of many graph problems is often a spanning tree of the graph G . For example, the cheapest way to connect all nodes of a graph is a minimum spanning tree. The solution to the shortest path problem is a *Shortest Path Spanning Tree*. It is usually sufficient to represent these trees by a single array of parent pointers $p[1..n]$. In addition we may need to store the depth or distance of node u from the root of the tree. The array $D[1..n]$ will contain these distances. We will augment the data structure below with these arrays when necessary. (see the *Implementation Notes* at the end of this chapter for details)

In this section we discuss general traversal algorithms. These algorithms start at some node in the graph and then ‘visit’ all those nodes that are reachable from the start node. As presented initially, these algorithms are skeletons which do nothing except traverse the graph : to be useful, additional statements must be added to them so that something useful is done at every node visited. This will be done when we apply these algorithms to specific problems.

5.3.1 Depth First Search.

This is an elegant recursive algorithm that has many applications. Indeed, it is the prototype graph algorithm, and with Breadth First Search, forms the basis for many other graph algorithms.

Depth first search for graphs is the same as that for trees except that, because of cycles, it must make sure not to visit a node twice. We assume that all nodes have been initially marked ‘unvisited’ before $DFS(u)$ is first called.

```

algorithm DFS(u:node)


---


  PreVisit(u)
  Mark u 'visited'
  FOR each node v ∈ Adj(u) DO
    IF NOT visited(v) THEN
      p[v] := u
      DFS(v)
    ENDIF
  ENDFOR
  PostVisit(u)
ENDALG DFS

```

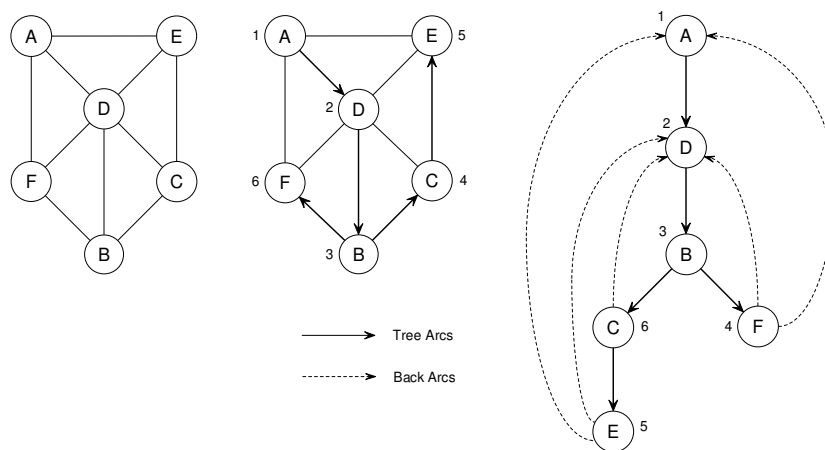


Figure 5.2. Depth First Search and its Tree.

The tree generated by DFS is not unique : it depends on the starting node and the order of nodes in the adjacency lists. DFS partitions the arcs of G into *Tree Arcs* and *Back Arcs* if G is undirected. Back arcs connect a node u to some ancestor of u , i.e., it connects nodes on the same path in the DFS tree. Figure 5.2 shows an example. If G is directed then DFS partitions the arcs into *Tree*, *Back*, *Forward*, and *Cross* arcs. Figure 5.3 shows these arcs.

Depth first search will visit those nodes that are *reachable* from the initial node. If the graph is connected then $DFS(u)$ will visit all nodes in the graph. If the graph is not connected then we need to call $DFS(u)$ with the following loop.

```

algorithm DFS-All(G : graph)


---


  Mark all nodes in G 'unvisited'
  FOR each node u ∈ G DO
    IF NOT visited(u) THEN
      DFS(u)
    ENDIF
  ENDFOR
ENDALG DFS-All

```

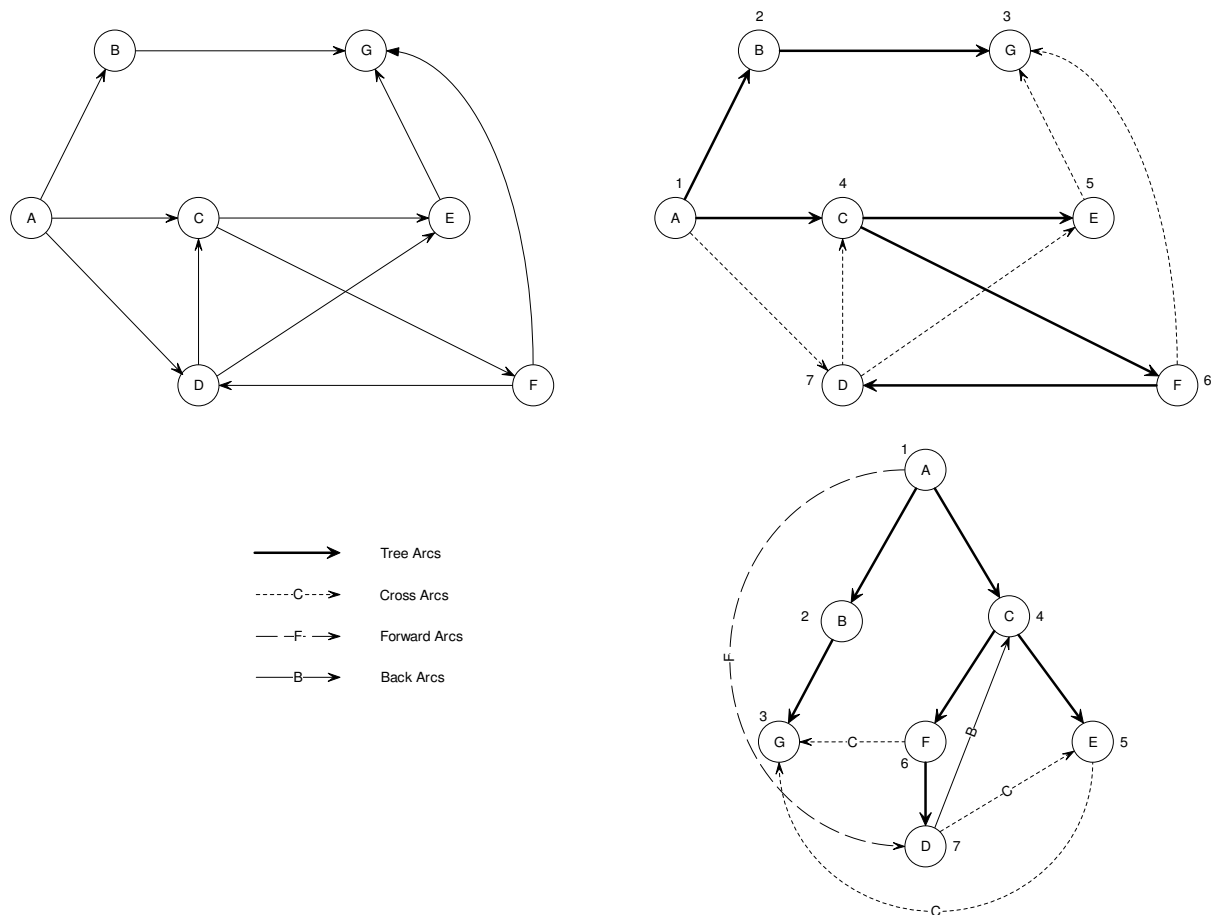


Figure 5.3. DFS on a Directed Graph.

Analysis of DFS. The FOR-loop of DFS-All takes $O(n)$ time, excluding the call to DFS. DFS is called exactly once for each $u \in N$ because it immediately marks u as ‘visited’ on entry to DFS. The FOR-loop in DFS is executed for each node adjacent to u , i.e., for each arc connected to u . Hence the total work for DFS is $\sum_u |Adj(u)| = O(m)$ and the total work for DFS-All is $O(m + n)$.

Applications of DFS. The first application we look at is **Graph Connectivity** : test a graph to see if it is connected or not. If it is not connected then give a list or number of its components.

```

algorithm DFS-Connect(G:graph)
    NoComps := 0
    Mark all nodes in G ‘unvisited’
    FOR each node u ∈ G DO
        IF NOT visited(u) THEN
            DFS(u)
            Inc(NoComps)
        ENDIF
    ENDFOR
    RETURN (NoComps)
endalg DFS-All
    
```

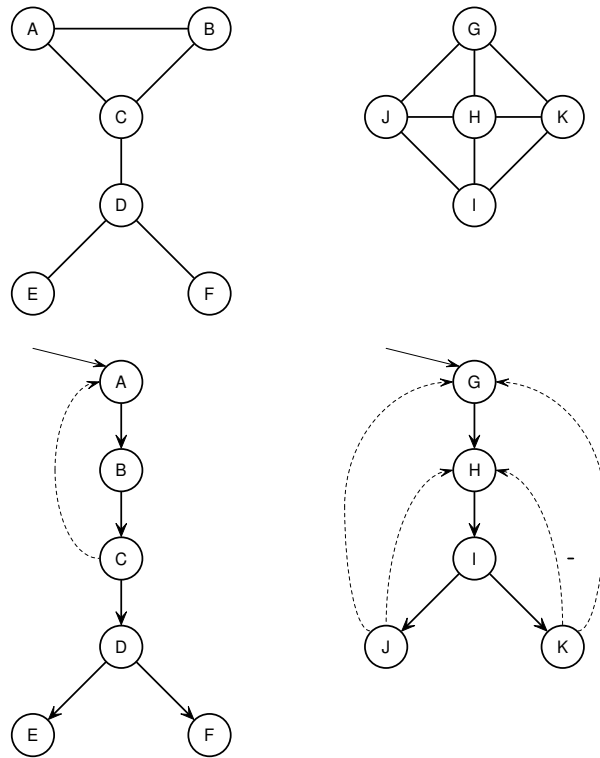


Figure 5.4. DFS-Connect on a Graph with 2 Components.

The results of this algorithm are shown in Figure 5.4

The second application is **Topological Labeling** of a directed acyclic graph (DAG). PERT networks are examples of such graphs where nodes represent tasks and a directed arc (u, v) represents the precedence relation ‘ u must be performed before v ’. The graph represents a project and we wish to find an ordering of the tasks so that no precedence relation is violated. A DAG with n nodes is *topologically labeled* if we can label each node with a number $i \in \{1 \dots n\}$ such that $i < j$ if there is a directed arc from node labeled i to node labeled j . Figure 5.5 shows such a labeling. We note that the labeling is not unique. An algorithm to topologically label a graph is a simple modification of Depth First Search. This is shown below.

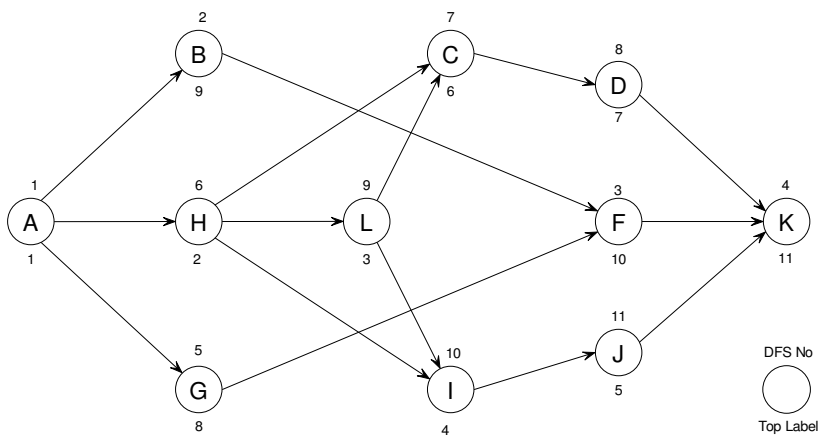


Figure 5.5. Topological Labeling of a DAG.

Exercise 5.3.1 : Acyclicity Testing. Modify DFS to test if a directed graph is acyclic. If it is not then identify the cycles.

algorithm *DFS-TopSort(DAG : graph)*

```

Mark all nodes in DAG 'unvisited'
DFSLabel := 1; TLabel := n
FOR each node  $u \in DAG$  DO
    IF NOT visited( $u$ ) THEN
        DFS-TS( $u$ )
    ENDIF
ENDFOR
endalg DFS-TopSort

```

DFS-TS is the same as *DFS* except that PostVisit labels the current node with the global variable *Label* and decrements *Label* by 1.

algorithm *DFS-TS(u :node)*

```

{ DFSLabel and TLabel are global variables }
{ They are initialized to 1 and  $n$  respectively }
DFSLabel[u] := DFSLabel
Inc(DFSLabel)
Mark  $u$  'visited'
FOR each node  $v \in Adj(u)$  DO
    IF NOT visited( $v$ ) THEN
         $p[v] := u$ 
        DFS-TS( $v$ )
    ENDIF
ENDFOR
TopLab[u] := TLabel
Dec(TLabel)
ENDALG DFS

```

We will come back to topologically labeled DAG's at the end of the chapter. There we will give a *Longest Path Algorithm* to find the *Critical Path* in a project network.

Exercise 5.3.1 : Devise an algorithm to count the number of directed paths from the source to the sink of a directed acyclic graph.

5.3.2 Breadth First Search.

This is an algorithm that, like Depth First Search, has many applications. Unfortunately, there seems to be no elegant recursive version of it. It is a prototype for Prim's minimum spanning tree and Dijkstra's shortest path algorithm.

Breadth first search for graphs is the same as that for trees except that, because of cycles, it must make sure not to visit a node twice.

```

algorithm BFS(s : node, G : graph)
    Mark all nodes  $u \in G$  'unvisited'
    Create an empty queue Q
    EnQueue(s, Q)
    WHILE NOT Empty(Q) DO
        u := DeQueue(Q)
        Visit(u)
        FOR each  $v \in Adj(u)$  DO
            IF NOT visited(v) AND  $v \notin Q$  THEN
                p[v] := u
                EnQueue(v, Q)
        ENDFOR
    ENDWHILE
endalg BFS
    
```

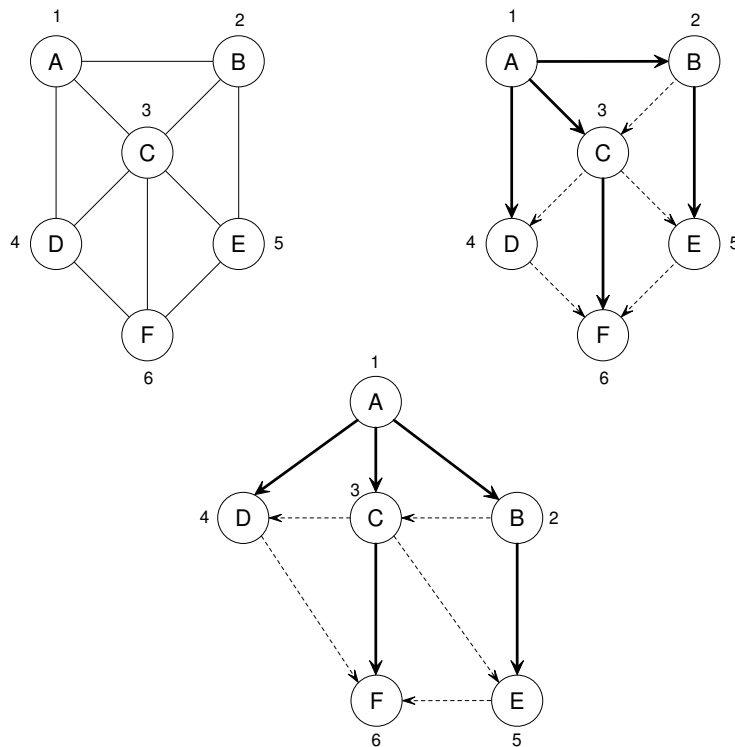


Figure 5.6. Breadth First Search and its Tree.

The tree generated by BFS is not unique : it depends on the starting node and the order of nodes in the adjacency lists. BFS partitions the arcs of G into *Tree Arcs* and *Cross Arcs* if G is undirected. Cross arcs connect a node u to some node v that is not on the same tree path as u .

Exercise 5.3.1 : BFS on Directed Graphs. Examine how BFS works on directed graphs and classify the arcs as in DFS.

Analysis of BFS. It is obvious that each node is enqueued and, therefore dequeued once and only once. Individual queue operations are $O(1)$ and so the total time spent on queue operations is $O(n)$. The

adjacency list for each node taken off the queue is scanned once. Hence the total time spent scanning adjacency lists is $O(m)$. Hence the total time spent by BFS is $O(m + n)$.

Applications of BFS. The first application we look at is **Shortest Arc-Paths** : find the path between nodes s and t that has the least number of arcs. A simple modification to BFS solves this problem : add the statement

$$D[v] := D[u] + 1$$

just before the `EnQueue(v, Q)` statement. We then call `BFS-SAP(s, t, G, D)` which starts at s and stops when t has been dequeued. The array $p[1..n]$ defines a tree rooted at s and the array $D[1..n]$ gives the minimum number of arcs from any node to the root.

The **Diameter** of a graph G is the largest of all the shortest arc paths in the graph.

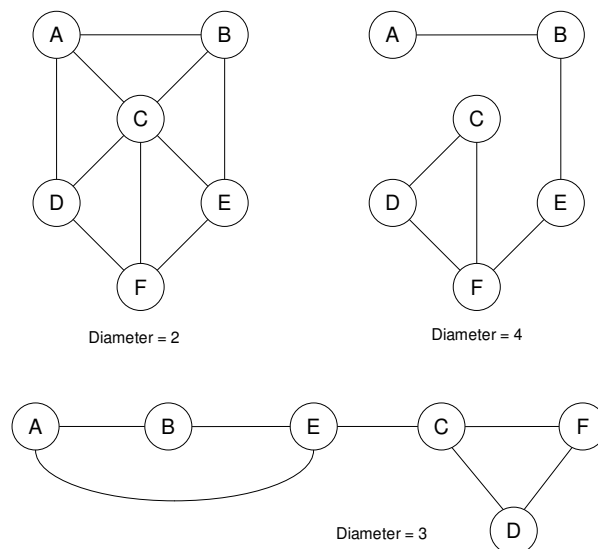


Figure 5.7. Graphs of 6 nodes with Different Diameters.

We can determine the diameter of G by performing *BFS-SAP* on each node of G as follows :

```

algorithm BFS-Diameter( $G$ :graph)
  Diam := 0
  FOR each node  $u \in G$  DO
    FOR each node  $v \in G$  DO  $D[v] := 0$ 
    BFS-SAP( $u, v, G, D$ )
    Diam := Max(Diam, Max( $D[v]$ ,  $v=1..n$ ))
  ENDFOR
  RETURN (Diam)
endalg DFS-All

```

Exercise 5.3.1 : Determine the complexity of *BFS-Diameter*

Exercise 5.3.1 : Modify BFS to topologically label a directed acyclic graph.

A General Iterative Traversal Algorithm. The following is a general, iterative graph traversal algorithm. It is a slightly modified version of that given in Ruhe [1991].

```
algorithm SearchG(G:graph, r:node)
```

```
{ S is a selection set. from which nodes are selected }
Create(S);
FOR each node  $u \in N$  DO visited[u] := false
Insert(r,S)
visited[r] := true
WHILE NOT Empty(S) DO
  u := Select(S)
  FOR each node  $v \in Adj(u)$  DO
    IF NOT visited[v] THEN
      p[v] := u
      visited[v] := true
      Insert(v,S)
    ENDIF
  ENDFOR
ENDWHILE
endalg SearchG
```

5.4 MORE APPLICATIONS OF GRAPH TRAVERSAL ALGORITHMS

In this section we develop two fairly substantial applications of graph traversals

5.4.1 Bills Of Materials.

Materials Requirements Planning is a method of inventory control that tries to keep track of all the components required for a product that comprises many parts and sub-assemblies. Its latest name is *Just-In-Time Inventory Management*. A Bill of Materials shows the number and type of each part, sub-assembly etc. that make up the final product. For example, the bill of materials for a bicycle could be as follows :

Bicycle (end product)	
2 Wheels	Wheel (sub-assembly)
1 Frame	1 Rim
1 Saddle	30 Spokes
1 Handlebars	1 Tyre
1 Front Brake	1 Hub
1 Back Brake	1 Axle
1 Chain	
etc.	

Such a bill of materials can be conveniently represented by a directed acyclic graph, as shown below.

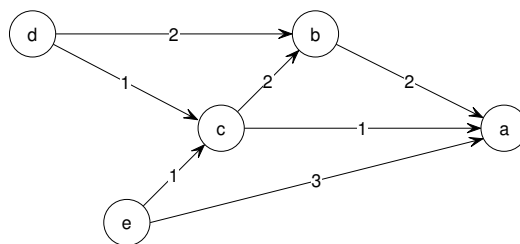


Figure 5.8. A Bill of Materials Graph.

The interpretation of this graph is as follows : the sink node *a* is the final product, the source nodes *d* and *e* are parts that have no sub-parts, and the remaining nodes are sub-assemblies. For example, node *c* is a sub-assembly made up of parts *d* and *e*, and is required in sub-assembly *b* and final product *a*. A directed arc $u \rightarrow v$ with weight w means that v requires w units of u . (See the Appendix at the end of this chapter)

Requirements.

- Design a data structure to represent a bill of materials.
- Write an algorithm to determine the number and type of each sub-assembly, part etc., required for a given demand of end product.
- Cater for a number of different end products, using some parts and sub-assemblies in common.
- The size of the graph may be large and sparse. The number of parts in a simple disposable lighter is at least 20. The number of parts in a *Boeing 747* is 10,000?, 100,000?, 1,000,000?, or greater? (Take a guess)

In essence, what we want is a specialized data base that allows us to answer many of the questions that arise in the manufacture of a complex product.

5.4.2 Critical Paths.

APPENDIX

Linear Graph Algorithms Have Big Payoffs

by Paul Davis

SIAM News, Vol. 26, No. 2, March 1993

Robert Frost and Robert Tarjan both advocate taking the road less traveled, although one is thinking of a yellow wood and the other of a graph, the mathematician's abstraction of a network. Evidence that the choice makes all the difference can be seen in the impact of Tarjan's 1972 paper *Depth-First Search and Linear Graph Algorithms*, which appeared in the first volume of the SIAM Journal on Computing (pages 146-160). The ideas in that paper have been used in such diverse settings as database software, feature extraction in computer-aided design, stereo correspondence in image recognition, circuit modeling, and the analysis of error spread in power networks.

Graphs, the object of Tarjan's work, are abstract representations of relationships. Mathematicians often visualize them as schematic maps, like the maps in airline magazines, with lines, called edges, connecting cities, called nodes. The nodes can be labeled, e.g., with the names of cities, and the edges can carry such information as the travel time between the cities they connect. The basic idea of depth-first search is remarkably simple. Begin a tour at a chosen node. From each successive node, traverse an edge that is as yet untraveled. If all available edges from a node have already been traveled, backtrack to the most recently visited node from which an untraveled edge emanates and continue the tour.

If the graph in question were a family tree, depth-first search from an ancient ancestor to a member of the present generation would provide a list of "begats" from antiquity to the present day. The method is quite easy to program because the labels of old nodes with unvisited edges can be stored in a simple list—the list of parents with children not yet on the begat list.

A New Generation of Database Software that Supports Rapid Recursive Queries. The power of these ideas lies in the ability of graphs to represent relations far more complex than the cities connected by an airline. Tarjan's contribution lies in his utilization of depth-first search to develop elegant algorithms that identify important structures within those graphs.

H.V. Jagadish of AT&T Bell Laboratories suggests, for example, that the *bill of materials* (the parts list) for an aircraft engine could be stored in a database represented as a graph, a kind of family tree in which descendants are subassemblies on successively finer scales down to individual parts, such as screws.

The user of such a database might ask, "How many half-inch screws do I need to build one of these engines?" Answering that question relies ultimately on visiting every node of the database graph to determine which nodes reach the half-inch screw node. Each edge connecting a part to a half-inch screw might be labeled with the number of half-inch screws required for that part, much as the edges on the airline map are labeled with the travel time between the cities connected by the edge. Totaling the number of screws requires visiting every node (to determine whether it reaches a half-inch screw node) and summing the edge labels (numbers of screws) on every edge reaching a node labeled "half-inch screw."

Regardless of how it is conducted, such a search requires computational time proportional to the number of edges. In the relational databases commonly used in industry, that search must be accomplished with repeated queries to the database. When data like the number of half-inch screws are needed repeatedly, the results of those time-consuming queries are best stored in special tables separate from the database itself.

Alternatively, Rakesh Agrawal of IBM's Almaden Research Center explains, "If the database itself could accommodate recursive queries, one could avoid constructing outside tables and rapidly handle a great many important problems—network reliability, bill of materials inquiries [like the half-inch screw problem], airline scheduling, and so on."

Roger Miller, a database specialist at IBM's Santa Teresa Laboratory, adds: "Efficiently providing a bill of materials is a key to just-in-time manufacturing. Even high-volume products like automobiles are effectively built to order.

"A typical automobile might offer forty or fifty options, many of them, like color options, more complex than simple yes-no choices. Individual dealers may tend to generate similar orders—high-priced stereos from a dealer in an affluent suburb and economy engines from a rural dealership—but the daily flow of orders from across the nation for, say, ten thousand automobiles may include six to eight thousand essentially unique vehicles.

"The manufacturing manager has to answer two questions rapidly and accurately: What parts do I send to the plant? What cars can I build today? The old saying in the manufacturing business is, 'Without a bill of materials, I don't know what to build, where to get the pieces, or where to ship it.'

"The competitive pressures of cost-saving strategies like just-in-time manufacturing demand a new generation of database software that can accommodate the recursive queries essential to efficient construction of a bill of materials (in addition to meeting other needs of materials planning and inventory management). Beyond providing quick answers to questions of manufacturing process control, databases that support rapid recursive queries would also open the door to complex optimal scheduling algorithms that could minimize costly changes in tooling, for example, while still meeting rigid delivery schedules with little or no manufactured inventory on hand.

Interest in recursive query processing on the part of the database community dates back to a proposal by Henschen and Naqvi in 1981. Recursive queries, in fact, were possible in the PROLOG programming language, although only over small amounts of data resident in the main memory (PROLOG also uses a depth-first search strategy). Among the approximately 10 projects undertaken since 1981 to investigate recursive query processing over large amounts of data are the LDL project at MCC, ECRC's Deductive Database Project, Ramakrishnan's CORAL project at the University of Wisconsin, Stanford University's GLUE/NAIL, and Melbourne University's Deductive Database Project.

Starburst, a research prototype relational database program now being developed by Agrawal and colleagues, has recursive capability because of a two-stage search algorithm. The first stage eliminates what might be called multiple references to identical parts, which result in circular paths in the graph of the database. The second takes advantage of the cycle-free structure of the graph to provide a recursive query capability through the construction of what is known as the transitive closure of the database graph. The closure can be thought of as a compressed set of relationships between distant ancestors and today's children.

Biconnected components—parts of the graph in which nodes are connected by two or more disjoint paths—could arise if a subassembly of the aircraft engine, say, contained two different sheet metal panels, both to be secured with half-inch screws. Two paths would extend from the node for that subassembly to the node for half-inch screws, one passing through each of the nodes for the sheet metal panels.

Tarjan's 1972 paper includes an algorithm for identifying these biconnected components; that algorithm is the key to Starburst's first-stage reduction. Tarjan has explained that the biconnected component algorithm is one of several algorithms presented in the paper that illustrate how the theoretical properties of depth-first search can be used to devise efficient graph algorithms. Efficiency, of course, is a key issue in any software that will be used frequently. (How long are you willing to wait for an automatic teller machine to respond?!) The work required by Tarjan's algorithms is linear in the size of the graph—that is, it is directly proportional to the number of edges (or relations between parts) and to the number of nodes (or parts) in the bill of materials for the engine. Hence, doubling the size of the database only doubles the processing time for this phase of the search.

In contrast, the time required for matrix-based algorithms is proportional to the cube of the number of nodes; the size of the node-by-node matrix that represents the database relations grows like the square of the problem size, and repeated processing (as in Gaussian elimination) adds a third factor that is proportional to the problem size. Doubling the size of the database would increase processing time by a factor of eight.

Agrawal and Jagadish have developed hybrid algorithms that combine the lineardependence of Tarjan's graph algorithms and their relatives with characteristics of matrix algorithms that are particularly well suited to the blocked character of disk resident data.

One by-product of Tarjan's biconnected component algorithm is a topological sort, a renumbering of the nodes, of the database graph. A modification of that idea permits a compression of the closure graph in which nodes are assigned a range of numbers (representing a range of parts, for example). Relations can then be tested simply by testing inclusion in the range assigned to a node. The influence of the ideas presented in Tarjan's paper extends beyond databases to many other applications, and even to settings in which a graph is not obviously present.

Grouping Variables with Common Effects in a Variety of Settings. Complex optimization problems in chemical engineering (such as distillation and heat exchanger design) and in many other engineering settings mix continuous variables (like the smooth variation in a car radio's volume control setting) with discrete variables (like the distinct steps in a car radio's digital tuner or the need to conduct cash transactions in units no finer than a penny). Efficient solution algorithms—even the problem of finding a true minimum—depend on decomposition of the enormous original problems into more tractable, coordinated subproblems.

Similar large problems arise in models that couple differential equations expressing rates of change of variables with algebraic equations expressing static relations. The former might represent chemical reactions in the atmosphere and the latter, relations between electricity demand and the quantities of certain pollutants. These large differential-algebraic models are also frequently solved by decomposition techniques.

In both settings, effective decomposition strategies can rely on Tarjan's algorithms to group variables with common effects, similar in spirit if not in detail to the grouping of parts that use half-inch screws. To automate the recovery of the image of a three-dimensional object from two-dimensional views taken from slightly different angles, as when a satellite scans a mountain on a planet passing beneath it, corresponding features from the two views must be matched. However, the match will not be perfect—a continuous ridge line in one view may be broken in the other because a peak has swung into the field of view.

Each possible pairing of features from the two views can be thought of as a node in a graph, and two pairings that are consistent with one another (e.g., appear to represent neighboring segments of the same ridge) can be connected by an edge. The most plausible three-dimensional view is represented by the largest subgraph in which every node is connected to every other.

A somewhat similar problem that arises in computer-aided design and manufacturing is the location of holes or protrusions in a representation of a solid object. The topography of the three-dimensional object can be represented in an appropriate graph.

In these settings, the ideas of Tarjan's paper are again applied to locate the connected components of the graphs

that correspond to the relations being sought among the data. The relative simplicity of the algorithms and their computational economy are both keys to their practical application in settings that involve large amounts of data. Connectivity is an issue in a completely different setting as well. Because it is impractical for an electric utility to measure the power flow in every line in its network, the utility's control center computes the flows from measurements made at selected points of the network. The control center can take corrective action when flows fall outside specified limits. Unfortunately, faulty transducers or telemetry can corrupt the utility's data and the corresponding computed power flows, masking problems that may require immediate attention. An analysis of the relation between such data errors and the computed flows shows that the influence of the data errors is confined to biconnected components of the network; the nodes in a biconnected component of a graph are connected by at least two distinct paths. Commercial implementations of this error analysis tool by a number of utilities rely precisely on Tarjan's algorithm for finding biconnected components.

Beyond these specific applications, the real measure of the power of "Depth-First Search and Linear Graph Algorithms," and closely related work done by Tarjan and John Hopcroft of Cornell University, may be its very ubiquity. The elegance and deceptive simplicity of these ideas have placed them in the hands of a community much larger than the circle of theoretical computer scientists to which they were originally addressed.

Like the metallurgy that produces a surgeon's scalpel, an artist's palette knife, and an electrician's stripping blade, the simplicity of the final result is so powerful that it is adopted and used with hardly a second thought. Tarjan's work is the starting point for an entire family of ideas that are part of the fabric of thought in diverse areas of science and engineering.

Paul Davis is a professor of mathematics at Worcester Polytechnic Institute.

Reprinted from SIAM NEWS Volume 26-2, March 1993 (C) 1993 by Society for Industrial and Applied Mathematics All rights reserved.

5.5 References

- [1] G. Ruhe: *Algorithmic Aspects of Flows in Networks*, Mathematics and its Applications, Volume 69, Kluwer Academic Publishers, Dordrecht/Boston/London, 1991
- [2] Tarjan, R.E. : “Depth-First Search and Linear Graph Algorithms”, *SIAM Journal on Computing*, Vol., 1 No. 2, June 1972.