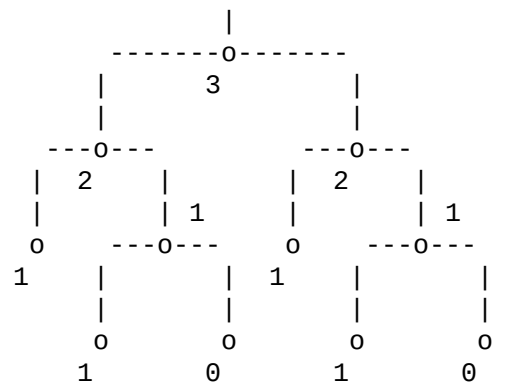


Figure 4-16. Register Requirements Tree.

Instruction	Type of Move	Vertex Pebble Placed On
L R1, c	1	c
L R2, x	1	x
M R1, R2	2	t1
L R3, b	1	b
A R1, R3	2	t2
M R1, R2	2	t3
L R2, a	4,1	a
A R1, R2	2	t4

Figure 4-14. Trace of Pebble Game for Expression of Figure 4-12.

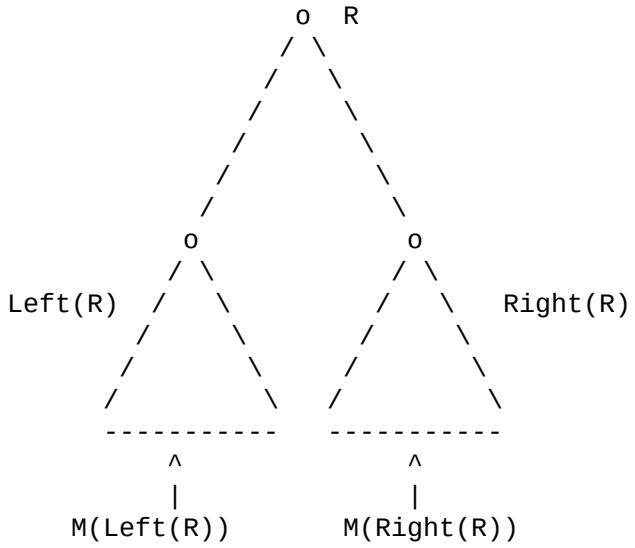


Figure 4-15. Recursive Structure of Register Requirements Algorithm.

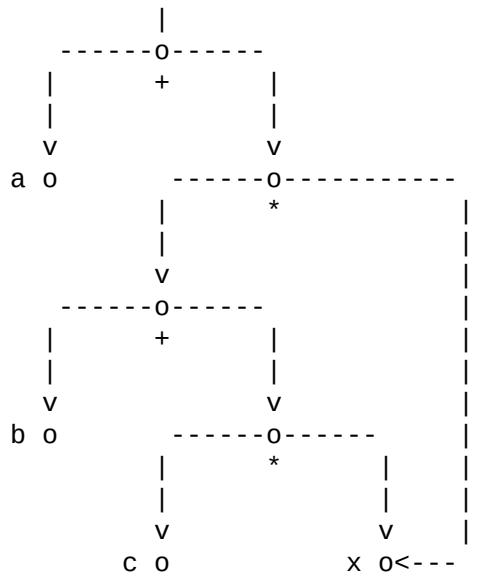


Figure 4-12. Acyclic Digraph Model for a + x * (b + c * x).

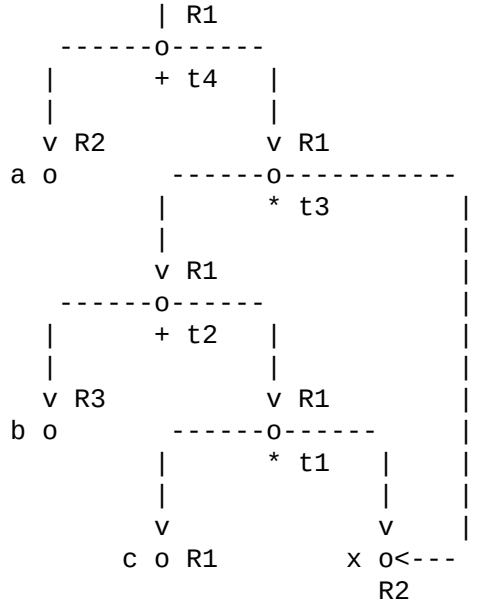
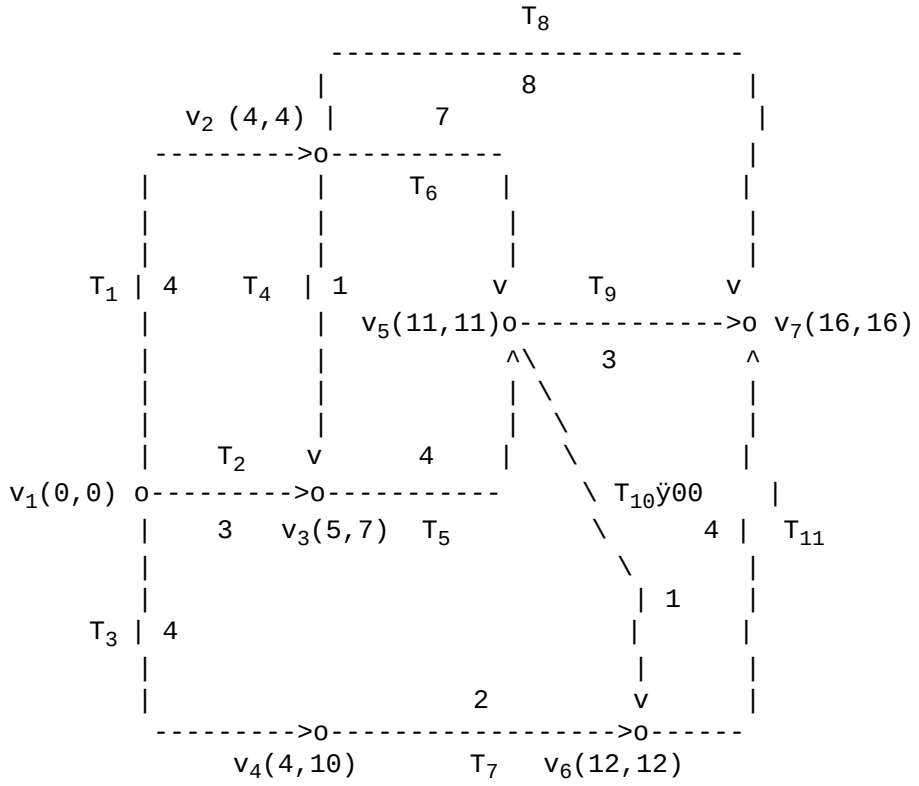


Figure 4-13. Digraph Model with Registers Indicated.



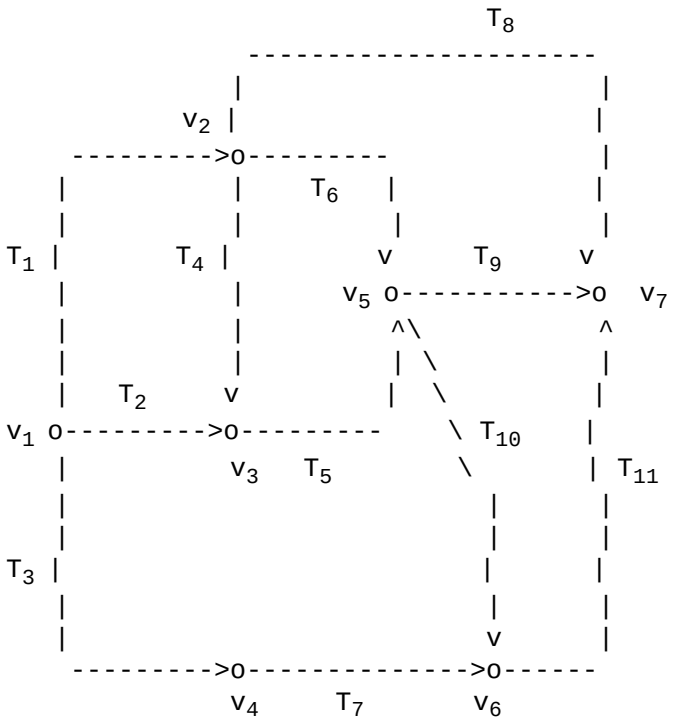
Critical Path: v_1 v_2 v_5 v_6 v_7

Notation:

label for Vertex v: (Early(v), Late(v))

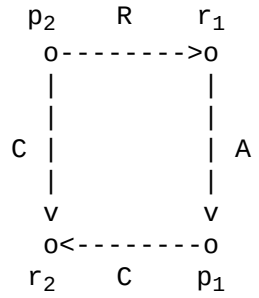
label for Edge (u,v): Task Length

Figure 4-11. Schedule Parameters for G.



Task	Precedents	Task Length
T_1	None	4
T_2	None	3
T_3	None	4
T_4	T_1	1
T_5	T_2 T_4	4
T_6	T_1	7
T_7	T_3	2
T_8	T_1	8
T_9	T_5 T_6	3
T_{10}	T_5 T_6	1
T_{11}	T_7 T_{10}	4

Figure 4-10. A Set of Tasks and Their Precedence Digraph.



Resources r_1, r_2

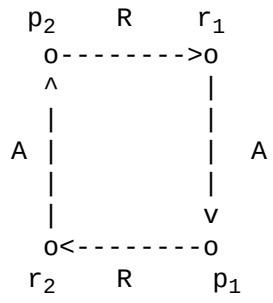
Processes p_1, p_2

Claims (p_1,r_2), (p_2,r_2)

Requests (p_2,r_1)

Assignments (r_1,p_1)

(a) State of System before p_2 Requests r_2: No cycle.



Resources r_1, r_2

Processes p_1, p_2

Claims (p_1,r_2)

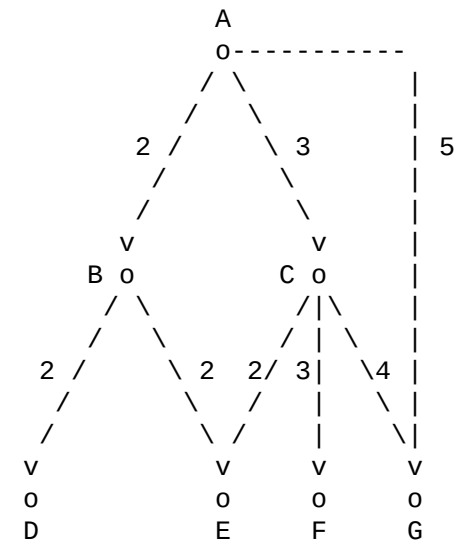
Requests (p_2,r_1)

Assignments (r_1,p_1), (r_2,p_2)

(b) State of System if p_2 Assigned r_2: Cycle

Figure 4-9. Resource Allocation Digraph for Deadlock Prevention.

SECTION 4-5-1.



(a) Acyclic Digraph for Parts Explosion.

	A	B	C	D	E	F	G
A	0	2	3	0	0	0	5
B	0	0	0	2	2	0	0
C	0	0	0	0	2	3	4
D	0	0	0	1	0	0	0
E	0	0	0	0	1	0	0
F	0	0	0	0	0	1	0
G	0	0	0	0	0	0	1

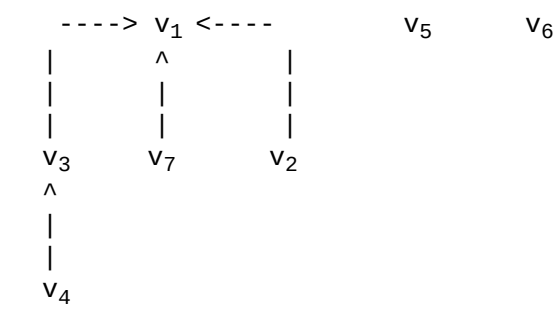
(b) Bill of Materials Matrix M.

	A	B	C	D	E	F	G
P:	1	0	0	0	0	0	0
PM:	0	2	3	0	0	0	5
(PM)M:	0	0	0	4	10	9	17

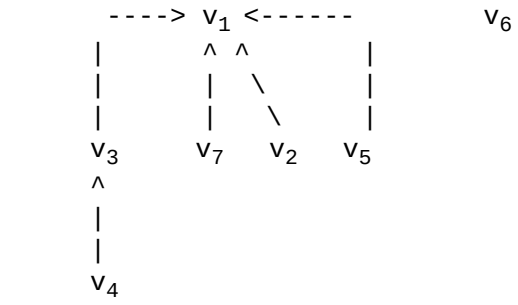
(c) Component Requirements for One A part: (PM)M.

Figure 4-8. Bill of Materials Calculation.

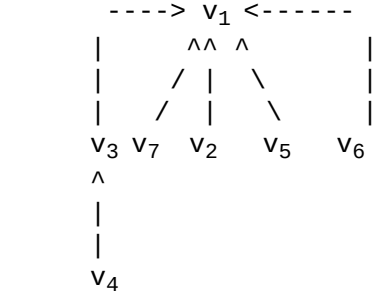
(b) After (v_2, v_7).



(c) After (v_3, v_7).



(d) After (v_4, v_5).



(e) After (v_1, v_6).

Figure 4-7. Disjoint Set Sequence for Example.

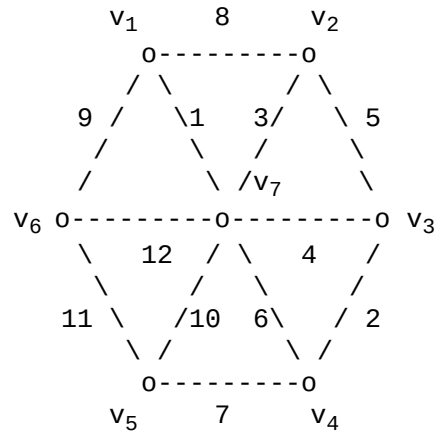


Figure 4-5. A Weighted Graph G.

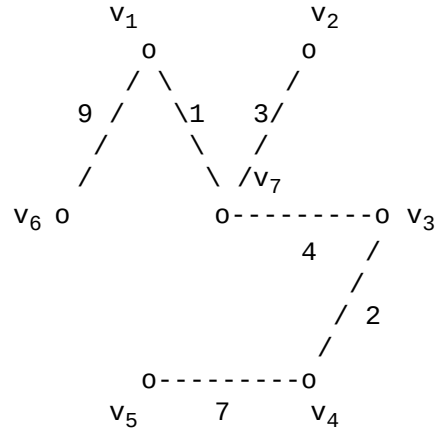
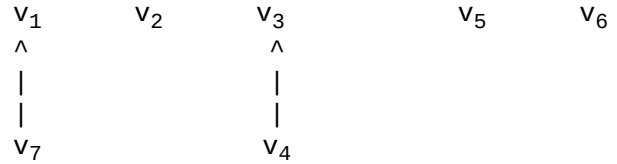
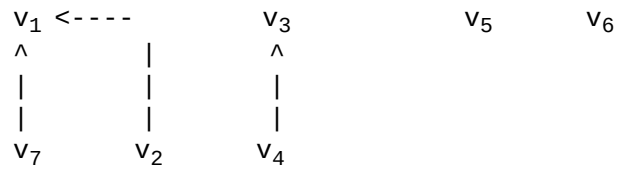
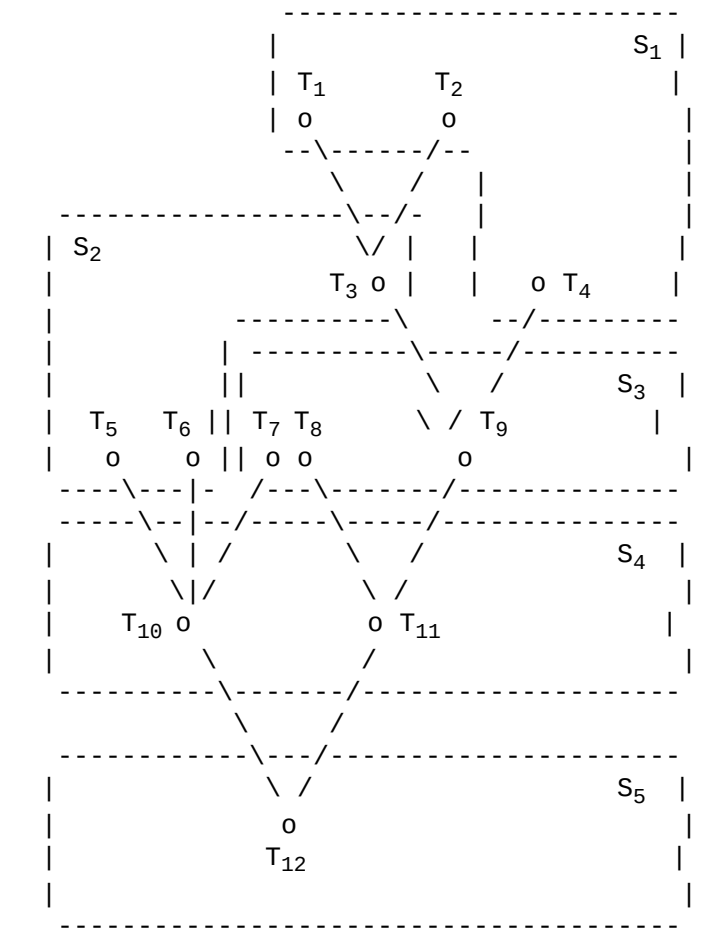


Figure 4-6. Minimum Spanning Tree on G.



(a) After (v_1, v_7) and (v_3, v_4)





S_i: The set of tasks started at time i - 1 and completing at time i.

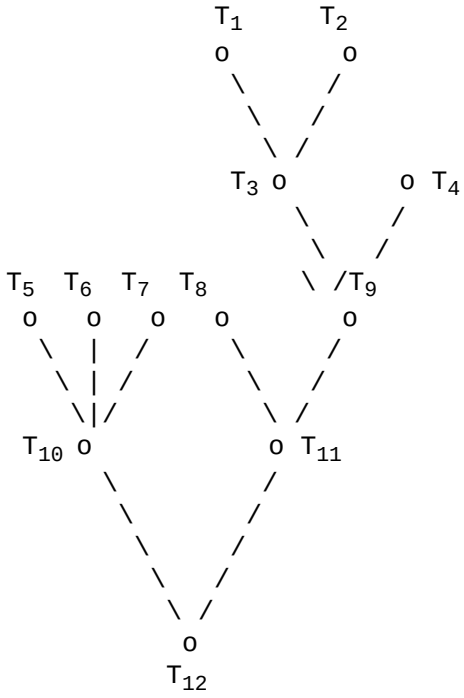
(b) Optimal Schedule.

Time:	0	1	2	3	4
P_1:	T_1	T_3	T_7	T_{10}	T_{12}
P_2:	T_2	T_5	T_8	T_{11}	
P_3:	T_4	T_6	T_9		

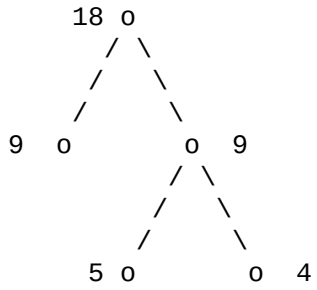
(c) Gannt Chart for Optimal Schedule.

Figure 4-4. Example of Hu's Optimal Scheduling Algorithm.

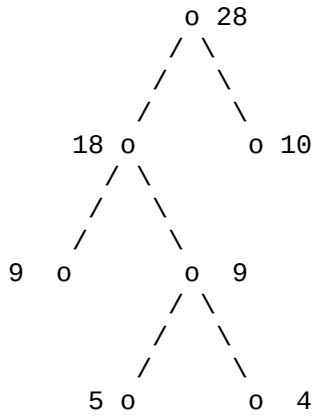
SECTION 4-2-4



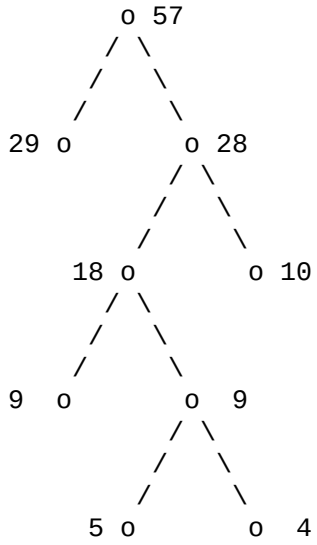
(a) Precedence Tree.



(b) Frequencies: 9, 9, 10, 29.

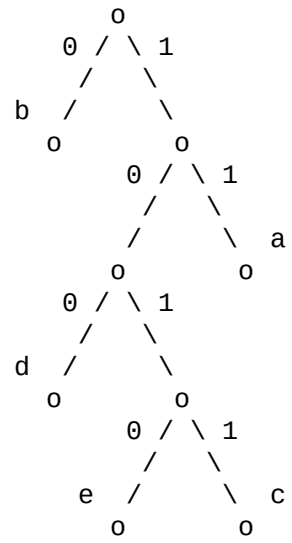


(c) Frequencies: 10, 18, 29.



(d) Frequencies: 28, 29.

Figure 4-3. Example of Huffman Algorithm.



(b) Binary Decoding Tree.

Figure 4-1. Huffman Representation.

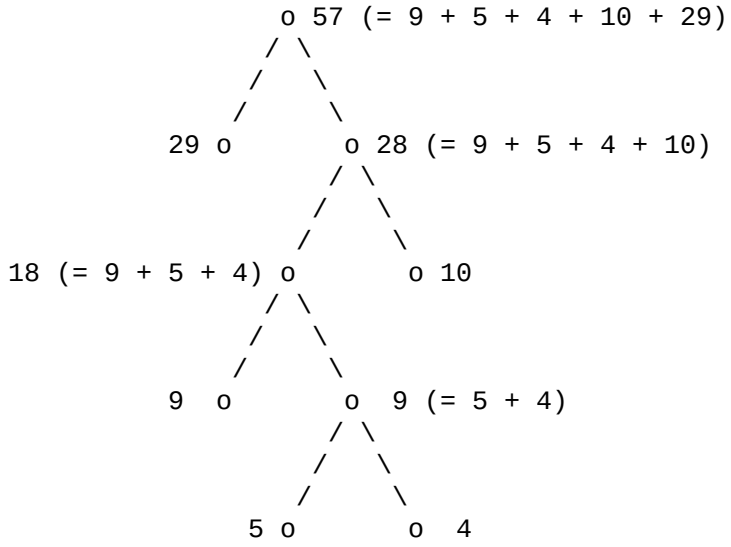
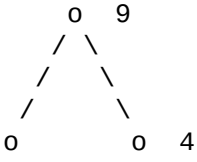


Figure 4-2. Binary Decoding Tree with Internal Labels Shown.



(a) Frequencies: 4, 5, 9, 10, 29.

(19) Compare the performance of implementations of the recursive and nonrecursive algorithms given for topological sort.

(20) Why does the longest path algorithm break down if the digraph is not acyclic?

(21) Show the minimum number of comparisons needed to insert a number into an ordered list of n numbers is O(log n).

(22) What is the maximum number of edges in an acyclic digraph?

(23) Let G(V,E) be a graph satisfying that every pair of its vertices have a unique neighbor in common. Characterize G (first show it has a spanning star).

(24) Generate a family of random weighted graphs (with edge weights chosen randomly, say, from {10,20,30}). Compare the weights of optimal spanning trees on these graphs to the weights of random spanning trees.

(25) Repeat the same experiment, this time for the shortest path problem, and compare the lengths of the shortest paths to the lengths of random paths between pairs of vertices.

SECTION 4-2-3

Character	Frequency	Representation
a	10	11
b	29	0
c	4	1011
d	9	100
e	5	1010

(a) Table of Character Frequencies and Their Huffman Codes.

i

where d_i's represent the degrees. Give an algorithm that counts the number of endpoints in a binary tree which is represented in a purely linked manner (like a binary search tree) so that its degree sequence is indeterminate. Make the algorithm also determine the height of the tree.

(6) Characterize all structured flowcharts algorithmically by showing how they can be derived from an initial primitive structured flowchart using suitable vertex and edge insertions.

(7) Graph-theoretically characterize the minimum number of counters that need to be maintained to monitor the frequency of execution of the statements in a structured program. Use the linear dependence of the values of the counters. Where do the optimal counters have to be placed?

(8) Try to find an efficient algorithm for determining whether reversing a given edge in an acyclic digraph introduces a cycle.

(9) Write an algorithm that generates random permutations, and then use these to generate random binary search trees. Compare the average height of these trees with Devroye's estimate.

(10) Prove that the euclidean minimum spanning tree constructed using the Voronoi diagram has maximum degree equal to 5.

(11) Give an interpretation for the minimum spanning tree problem, where the edge weights are probabilities of edge failure and the objective function is the product of the edge weights of the spanning tree. Does the same algorithm work?

(12) Consider the following greedy algorithm for euclidean spanning trees in 2-space. First, sort the vertices on their x coordinates, then recursively construct a minimum spanning tree on each half of the vertices; then join the two subtrees by the shortest edge connecting them. Comment on correctness, error estimate, counterexample, and performance.

(13) Prove that if d_i, $i = 1, \ldots, n$ is a sequence of positive integers whose sum equals $2n - 2$, there exists a tree with this sequence as its degree sequence.

(14) Prove the complement of a tree is either connected or consists of an isolated vertex and a complete subgraph.

(15) What is the long term effect of an error in the transmission of a bit in a Huffman encoded message?

(16) Implement the minimum spanning tree algorithm using Voronoi diagrams.

(17) Let G(V,E) be a graph. If min(G) is at least k, then G contains every tree on k + 1 vertices as a subgraph.

(18) Write an efficient implementation of the algorithms for the PERT parameters described in the text.

the heap or fringe vertices. If a pass finishes with t' trees, there must be at least kt' such edges, which is at most 2|E'|. Thus, $t' \leq 2|E'|/k$. The heap size limit k' for the next pass then satisfies $k' = 2^{(2|E|/t')}$ which is therefore at least 2^k. Considering the heap size limit for the initial pass, and since by the definition of the algorithm the size limit equals |V| only on the final pass, it follows that the number of passes is bounded by 1 + $\min\{i|\ \log^{(i)}(|V|) \leq 2|E|/|V|\}$, as required. This completes the proof.

CHAPTER 4: REFERENCES AND FURTHER READING

Deo (1974) has a thorough discussion of the matrices and vector spaces associated with graphs. Kruskal (1956), Prim (1957), and Dijkstra (1959) developed minimum spanning tree algorithms. A history of the problem is given in Graham and Hell (1985). For a discussion of path compression for unary trees, see Tarjan (1983), as well as Tarjan (1979), Banachowski (1980), and Tarjan and van Leeuwen (1984). Flajolet and Odlyzko (1982) obtain results on the average heights of random binary trees using generating function methods. Devroye (1986) gives the average height of a binary search tree. See also Knuth Vol. 3 (1973). The parts explosion application is from Gotlieb and Gotlieb (1978). See Coffman and Denning (1973) for a discussion of the optimal task scheduling algorithm, and further references, and Mc Hugh (1984) for a short proof of the correctness of the algorithm. Sethi (1975) and Sethi and Ullman (1970) discuss the complete register allocation model. An interesting example of the use of fundamental cycles for determining the optimal placement of program probes is given in Knuth and Stevenson (1973). The minimum spanning tree algorithms based on Fibonacci heaps are from Fredman and Tarjan (1987). They improve earlier bounds, such as the O(|E| log(log(|V|))) bound of Yao (1975), for sufficiently sparse graphs.

CHAPTER 4: EXERCISES

(1) Write a program that accepts a graph G(V,E) and a spanning tree T of G as its input, and outputs the fundamental cycles of G with respect to T. Also, if a cycle C is additionally input, the program outputs the subset of fundamental cycles whose symmetric difference equals C.

(2) Prove every cut is the union of edge disjoint minimal cuts. This is not true of disconnecting sets in general. Give an example.

(3) Prove that in an eulerian graph a cutset must have an even number of edges.

(4) Prove that for a connected graph G(V,E)
(a) G is eulerian if and only if all its cuts are even, and
(b) G is eulerian if and only if all its fundamental cuts are even.

(5) Prove the number of endpoints in a nontrivial tree equals

$$1 + \left(\sum | d_i - 2 | \right)/2,$$

At this point, if another unscanned neighbor of v remains and the heap size is less than k, we repeat steps (a) through (c) for that neighbor; otherwise, if an unscanned neighbor remains and the heap size equals k, go to (5); otherwise go to (1).

(5) Empty the heap (if necessary) and reset the keys of all the vertices scanned during the current phase to +M.

This tree-building process (selecting an unmarked vertex and repeating steps (1) to (5)) is repeated until every vertex is included in some tree (that is, is marked). When no unmarked vertex remains, we have constructed a spanning forest of the graph. Then, we condense the graph by shrinking each tree to a super-vertex. We then repeat the whole process on this new graph of super-vertices, resulting in a new spanning forest for this condensed graph. The process terminates only when some phase produces a trivial (single vertex) graph.

Condensing a graph G(V,E) for which we have a current spanning forest can be done in time O(|E|). Assume that for each vertex v, we know the number num(v) of the tree containing v. We can then lexicographically sort the edges on the tree numbers of their endpoints using a two-pass radix sort. Thus, we construct an array of |V| queues, which are initially empty, and scan the second coordinates of the edge list, enqueueing each edge (u,v) on the queue for num(v). Then, we concatenate the |V| queues in the array of queues to form a new edge list. This takes time O(|V| + |E|). We then reinitialize the array of queues and scan the edge list on the first coordinate, inserting (u,v) on the queue for num(u). Repeating the concatenation process leaves the edges lexicographically sorted on the numbers of the trees of their endpoints. At this point, it is trivial to eliminate edges both of whose vertices are in the same tree, as well as to delete all but the least weight edge between a pair of distinct trees. The whole process takes O(|V| + |E|) time.

THEOREM (FREDMAN-TARJAN MST PERFORMANCE) Let G(V,E) be a weighted graph. Then, the Fredman-Tarjan minimum spanning tree algorithm takes $O(|E|(1+\min\{i \mid \log^{(i)}(|V|) \leq 2|E|/|V|\}))$ time.

The proof is as follows. We must first make a suitable choice for the heap size limit k. The choice involves a trade-off. The larger k is, the longer the passes will be, though there will be fewer of them. The smaller k is, the shorter the passes will be, but there will be more of them. If the current (condensed) graph contains t super-vertices, the algorithm will incur at most t deletemin operations and at most O(|E|) inserts and decrease-key operations. If the heap size is at most k, the total time for the pass will be $O(t \log(k) + |E|)$. If we take k equal to $2^{(2|E|/t)}$, the running time of the pass is O(|E|).

To bound the number of passes we argue as follows. Consider a pass where the number of trees equals t and the number of edges in the condensed graph is |E'|. Except for the final pass, the trees grown during a pass become blocked either because the heap size limit is reached or a pair of trees merge. In either case, the trees end up with at least k edges incident with

the operation of Dijkstra's algorithm. There is a distinguished tree (or subtree), corresponding in this case to the partial minimum spanning tree. There is an extended tree that includes not just the partial minimum spanning tree, but also all the vertices that have been reached by the search process. The vertices in this search tree which are not yet in the partial minimum spanning tree lie in the heap. Finally, there is an unknown set of vertices that have not yet been reached by the search process. The operations, such as decrease-key, are the same as in Dijkstra's algorithm. Thus the Fibonacci heap implementation is O(|V| log |V| + |E|).

FREDMAN-TARJAN FIBONACCI HEAP MST ALGORITHM

The Fredman-Tarjan MST algorithm is a variation on the basic Prim algorithm wherein we constrain the tree growing process by restricting the size of the vertex heap. Once the heap reaches a certain limit, say after it contains k vertices, we empty the heap and start constructing a new tree at a previously unexamined vertex. Each application of this process results in a partial spanning tree. The process is iterated until every vertex in the graph is included in some component of a spanning forest. Then, we condense each of the partial spanning trees in the spanning forest into a single (super) vertex, one per component of the forest. We then repeat the whole process on this new graph of supervertices. The algorithm terminates when some phase produces a single spanning tree.

We now describe this process in more detail. Each vertex is assigned a cost, which is initially +M, and a mark field, which is initially set to unmarked. We construct a tree by selecting an unmarked vertex v_0, set cost(v_0) to 0, and insert v_0 into the heap. We then grow the tree starting at v_0 by repeating the following steps until either the size of the heap exceeds k or the heap is emptied or the tree being grown is linked to a previously grown tree.

(1) Delete the smallest element v from the heap and set cost(v) to -M; otherwise go to (5) if the heap is empty.

(2) If v is not equal to v_0, add the associated edge (v,m(v)) to the tree.

(3) If v is marked, and so the tree being grown has just been linked by (v,m(v)) to a previous tree, go to (5).

(4) Otherwise, mark v and scan each neighbor w of v as follows.

(a) If cost(w) = -M, then w is already in the current tree and so, to prevent self-loops, we ignore w.

(b) If cost(w) = +M, then w has not been scanned before; so we set cost(w) to c(v,w), set m(w) to v, and insert w in the heap.

(c) If cost(w) <> +M or -M, then w is already in the fringe of vertices neighboring the current tree. If cost(w) > c(v,w), then we decrease the cost of w to c(v,w) and set m(w) to v.

procedures of algorithmic graph theory.

4-6 FIBONACCI HEAPS AND MINIMUM SPANNING TREES

We described how to use Fibonacci heaps in Chapter 3 to implement a fast version of Dijkstra's shortest path algorithm. Fibonacci heaps can also be used to implement a fast version of Prim's minimum spanning tree algorithm (see Section 4-4), leading to an algorithm of performance $O(|V| \log |V| + |E|)$ for a weighted graph $G(V,E)$. An even faster, but more complicated algorithm, due to Fredman and Tarjan and also based on Fibonacci heaps yields a minimum spanning tree algorithm of performance $O(|E|(1+\min\{i| \log^{(i)} (|V|) \leq 2|E|/|V|\}))$, where $\log^{(i)}$ is defined inductively by: $\log^{(0)} (n) = n$, and $\log^{(i+1)} (n) = \log(\log^{(i)} (n))$. We shall describe both algorithms.

FIBONACCI HEAP VERSION OF PRIM MST ALGORITHM

The structure of Prim's algorithm is exactly like that of Dijkstra's algorithm. The idea is to repeatedly extend a partial minimum spanning tree T one vertex at a time, always selecting as the next vertex to add to T that nontree vertex v whose connecting edge (v,t) to T has least weight. That is, (t,v) is the least weight edge between T and G - V(T). The extended tree includes both the vertex v and the connecting edge (t,v).

At any point in the operation of the algorithm, each nontree vertex w which is a neighbor of some vertex in T is assigned a cost c(w), equal to the weight of the least cost edge fom w to T. The endpoint of this edge in T is denoted by m(w). The set of vertices neighboring T are maintained in a heap, which is heap ordered on the costs of the vertices. The minimum element in the heap is then the nearest nontree vertex to the tree.

The detailed operation of the algorithm is as follows. The costs c of the vertices are initialized to +M (which plays the role of plus infinity). An arbitrary vertex v_0 is selected as the vertex from which the spanning tree will be grown, and is inserted in the heap with a cost of 0. We then iteratively delete the minimum vertex v from the heap, set its cost to -M, add it to the minimum spanning tree under construction, add its associated edge m(v) to the minimum spanning tree (except in the case of the startup vertex v_0 which has no associated edge), and then scan the neighbors of v as follows. The neighboring vertices fall into three categories: vertices not previously reached, vertices already reached and deleted from the heap, and vertices reached but not yet deleted from the heap. The vertices not yet reached are just those with cost equal to +M. If w is such a vertex, we set the cost of w to the weight of the edge (w,v) connecting w to the tree and insert w in the heap. It is now a recognized neighbor of the spanning tree. If the cost of w is -M, then w and its associated edge are already in the minimum spanning tree; so we ignore it. Otherwise, w has already been reached by the search but is not yet in the minimum sanning tree; w has some current associated edge (w,u), where u equals m(w). If c(w,v) < c(w,u), we change the associated edge of w to (w,v) and decrease the cost of w to c(w,v), using a heap decrease-key operation.

Observe the strict similarity between the operation of this algorithm and

(!) stores. We can compute M recursively by the following labelling algorithm.

if M(Left(R)) = M(Right(R))

then Set M(R) to M(Left(R)) + 1

else Set M(R) to max { M(Left(R)), M(Right(R)) }

We set M to 1 for a left endpoint and to 0 for a right endpoint. Refer to Figure 4-16 for an example.

Figures 4-15 and 4-16 here

THEOREM (CORRECTNESS OF LABELLING ALGORITHM) Let T be a rooted binary tree (with root R) representing an expression. Then, the labelling algorithm correctly calculates the minimum number of registers M(R) needed to evaluate the expression without using stores to memory.

The proof is as follows. Let T with root R be the smallest binary tree for which the theorem fails. That is, suppose T is the smallest tree that can be evaluated using fewer registers than M(R). Let us denote the minimum number of registers required by T by m, and the minimum number of registers required by Left(R) and Right(R) by m_1 and m_2, respectively. We distinguish three possibilities:

(1) $m_1 = m$, $m_2 < m$,
(2) $m_1 < m$, $m_2 = m$, and
(3) $m_1 = m - 1$, $m_2 = m - 1$.

(Any other combinations can be easily shown to force m to be less than m - 1, a contradiction.) To prove the procedure correctly calculates m in case (1), observe that since the subtrees Left(R) and Right(R) are smaller than T, then m_1 and m_2 must both equal the values M(Left(R)) and M(Right(R)) found by the algorithm. Furthermore, since the computation of T entails the computation of Left(R), then m must be at least as great as M(Left(R)). But, this is precisely the value assigned by the algorithm to M(R) in case (1). Therefore, m must equal M(R), as was to be shown. The proof is similar in case (2). To prove the result in case (3), we argue as follows. Suppose the first instruction of the computation that computes T loads (some register) with an operand from, say, Left(R). Then, there must be at least one register retained for the computation of Left(R) up to the point where the root R of T itself is evaluated. Therefore, during the remainder of the computation, there are at most m - 2 registers available for computing Right(R), contradicting the assumption that Right(R) requires at least m - 1 registers for its computation. This completes the proof of the theorem.

Applications such as those just described illustrate how graphs can be used to model, in an abstract and concise manner, the essential structural features of a problem which may not be obviously graphical in nature. This allows us to reformulate questions about the original problem as graph-theoretic questions about the graph model, which can then by attacked by the methods and

The pebble movements can be interpreted in terms of the machine instructions required for the evaluation of the expression.

(1) Load a register with a value.
(2) Take a binary function of the values in a pair of sibling registers and store the result in one of the registers, if the register is available at this point.
(3) Take a binary function of the values in a pair of sibling registers and store the result in a newly allocated register, if none of the previously allocated registers are available at this point.
(4) Make a register available for reuse.

The object of the pebble game is to compute the digraph by making a sequence of moves that, starting with an empty digraph, places a pebble on every vertex exactly once, and ends up with a vertex of in-degree zero pebbled. An example is shown in Figures 4-13 and 4-14.

Figure 4-13 here

Instruction	Type of Move	Vertex Pebble Placed On
L R1, c	1	c
L R2, x	1	x
M R1, R2	2	t1
L R3, b	1	b
A R1, R3	2	t2
M R1, R2	2	t3
L R2, a	4,1	a
A R1, R2	2	t4

Figure 4-14. Trace of Pebble Game for Expression of Figure 4-12.

It can be shown that the problem of determining the minimum number of pebbles to compute a digraph is NP-Complete. (Refer to Chapter 10 for this terminology.) Consequently, though the pebble game can be solved by enumeration for sufficiently small digraphs, both it and the register allocation problem it models are computationally intractable for large digraphs.

However, when the model digraph is a tree, there is a simple, efficient algorithm for the register allocation problem. Thus, following the notation of Figure 4-15, let M(R) denote the minimum number of registers required to compute an expression represented by a binary tree model with root R without

(1) Early (v) = Long (v_1, v)
(2) Late (v) = Long (v_1, v_n) - Long(v,v_n)

Tasks for which Early and Late are equal have no leeway in their scheduling. When such a task is delayed, the completion time of the whole system of tasks is increased. For this reason, these tasks are called *critical tasks*, and the paths along which they lie, which are precisely longest paths from v_1 to v_n, are called *critical paths*. Refer to Figure 4-11 for an example.

Figure 4-11 here

4-5-4 OPTIMAL REGISTER ALLOCATION (TREE LABELING)

A classical problem of compilation is the so-called *register allocation problem*: determine the minimum number of registers required to evaluate an algebraic expression under the assumption that no stores into memory of intermediate operands are allowed. We can analyze this problem using a digraph model of expressions.

Given an algebraic expression consisting of binary operators and operands, we can model the expression by a digraph where each operator and each distinct operand of the expression is represented by a distinct vertex of the digraph and there is an edge (x,y) between the vertices for an operator x and an operand y if x operates on y in the expression. Refer to Figure 4-12 for an example. Observe that the vertices of out-degree two correspond to binary operators; while vertices of out-degree zero correspond to operands. Operator vertices can be thought of as combining the values at their children. Operands such as x in Figure 4-12, which are the children of more than one vertex, are operated on by each of their parent operators. The induced subdigraph of vertices reachable from an arbitrary vertex v corresponds to the computation of a subexpression. We can think of the result of the subcomputation as available at v upon completion; so v behaves like an intermediate operand of the total computation. The model digraph may not be a tree, because of multiply occurring operands, but is acyclic.

Figure 4-12 here

A well-known graphical pebble game on the expression digraph models the register allocation problem. The rules of the game and its computational interpretation follow. We assume there is an unlimited supply of pebbles. The pebbles correspond to registers available for computing the expression represented by the digraph. Moving pebbles on the digraph corresponds to allocating and deallocating machine registers required by the computation. The pebble game moves are as follows.

(1) Place an available pebble on an endpoint.
(2) If there are pebbles on every child of v, then place one of the pebbles on v.
(3) If there are pebbles on every child of v, place a new pebble on v.
(4) Remove a pebble from a vertex v, making it available.

T_{im}} has been completed. We can model such a system by a digraph G(V,E) whose edges correspond to the tasks. The digraph is constructed so none of the tasks corresponding to edges emanating from a vertex v can be initiated until every task corresponding to an edge ending at v has been completed. The vertices of the digraph model themselves merely serve as loci where the edges that represent the precedence constraints can be brought together. If the precedence constraints are consistent, then the digraph model is acyclic. Refer to Figure 4-10 for an example. We are interested in the following kinds of questions:

Figure 4-10 here

(1) What is the shortest time in which the system of tasks can be completed?

(2) Is there any leeway in scheduling a task: What is the earliest time it can be started? What is the latest time it can be started without delaying the system?

The earliest time a task (represented by an edge (u,v)) can be started, consistent with the precedence constraints, equals the length of a longest path to u. While it is computationally infeasible to compute longest paths in general there is a simple and efficient procedure, based on topological order, available in the case of acyclic digraphs. For simplicity, we will assume that the vertices are already topologically ordered and that there is a vertex v_1 from which all vertices are reachable and a vertex v_n reachable from every vertex, where n equals the order of G. We compute the lengths Long(v,v_n) of the longest paths from each vertex v to v_n. The data definitions are suggestive only.

```
Procedure Longest (G)

(* Computes the lengths Long(i,n) of the longest paths from
   each vertex v_i in G to vertex v_n *)

var  G: Graph
     n: Integer constant
     k,j, Long(n,n): Integer
Set Long(n,n) to 0

for k = n - 1 to 1  do

  Set Long (k, n) to      max  { Len(k, j) + Long(j, n) }
                       (k,j) î E(G)
End_Procedure_Longest
```

The lengths of the longest paths from v_1 to v can be computed similarly. We can define a variety of scheduling parameters in terms of the parameters Long, such as Early(v), the earliest time at which a task starting at v can start, and Late (v), the latest time at which a task starting at v can start without increasing the earliest possible completion time of the overall system of tasks. In terms of Long,

starts executing simultaneously.

A method of allocating resources in such a way as to prevent deadlock using a *resource allocation digraph* G(V,E) is as follows. G(V,E) has two types of vertices, resource and process vertices, and so is bipartite. There is a resource vertex for each instance of a resource, and a process vertex for each process. There are three types of edges, *claim edges*, *request edges*, and *assignment edges*.

We add, delete, or change edges depending on the requests, allocations, and deallocations made according to the following rules:

(1) If a process p may require a resource r, we enter a claim edge (p,r) in G, when the system is initialized.

(2) If a process p requests a resource r and r is unavailable, we change the claim edge (p,r) to a request edge (p,r).

(3) If a process p requests a resource r and r is available, we remove the claim edge (p,r) and enter an assignment edge (r,p).

(4) If a process p releases a resource r, we remove the assignment edge (r,p) and enter a claim edge (p,r).

(5) If there is a process q blocked on resource r when r is released, we allocate r to q, enter the assignment edge (r,q), and remove the request edge (q,r).

The resource allocation is subject to the *deadlock avoidance rule*: allocate a resource r to a process p only if allocating r does not cause a cycle in G.

The required cycle testing can be done, for example, using depth first search (refer to Chapter 5), or using more efficient techniques based on the special structure of the problem (namely, the graph is bipartite and existing edges are merely reversed). An example is shown in Figure 4-9. Figure 4-9a gives the current state of the system. If p_2 requests r_2, then r_2 should not be allocated to p_2 even though r_2 is currently available, by virtue of the deadlock avoidance rule. Otherwise, the state in Figure 4-9b will occur. Although that state need not lead to a deadlock, we cannot guarantee at this point that a deadlock will not occur. For example, if p_1 subsequently requests r_2 before p_1 releases r_1, deadlock will occur because p_1 will be blocked waiting for r_2, while p_2 is blocked waiting for r_1.

Figure 4-9 here

4-5-3 PERT (LONGEST PATHS)

Let $\{T_1, ..., T_n\}$ be a system of tasks where each task T_i takes time t_i, and the tasks are constrained by precedence constraints of the following type: task T_i can be initiated only after some specified subset of tasks $\{T_{i1}, ...,$

```
for every v î G  do  Set Index(v) to 0

for every v î G  do  if Index(v) = 0
                    then Topological_Order (G,v)

End_Procedure_Topological_Order


Procedure Topological_Order (G,v)

(* Rank the unnumbered vertices in G that are reachable from v in
   topological order *)

var  G: Graph
     v, w: 1..|V|

for every w î ADJ(v) do if    Index(w) = 0
                        then  Topological_Order(G, w)

Set Index(v) to  Subtract_count

End_Procedure_Topological_Order
```

4-5-2 DEADLOCK AVOIDANCE (CYCLE TESTING)

Deadlock is a phenomenon of resource sharing arising when limited resources cannot be simultaneously shared. It occurs when the processes using a set of resources become embroiled in a vicious cycle of resource requests and allocations. As an example, consider the case where two processes A and B, each of which may require up to three tape drives, share access to four tape drives. If A and B are each initially allocated two drives and A requests a third drive, A must be suspended (blocked) until B releases one of the drives already allocated to it. If at this point B also happens to request another drive, B is also blocked. Now, both processes are deadlocked, since each is waiting for the release of a resource allocated to the other suspended process, which de facto cannot release the resource.

Both of the following conditions must be satisfied for a deadlock to be possible:

(1) *Mutual Exclusion*, that is, at least one of the resources must be nonshareable in the sense that only one process at a time can use the resource, and

(2) *No Preemption*, that is, the unshareable resource(s) in (1) cannot be preempted; that is, the resource can only be released by the process to which it has been allocated only after that process has completed using the resource.

We shall make several assumptions about the environment: there is only a single instance of each type of resource; each process in the system lists all the resources it may possibly require prior to executing; and every process

Figure 4-8 here

We have assumed the parts are indexed so that the bill of materials matrix is in upper-triangular form. This can be used to simplify both the matrix calculations and storage requirements. The matrix is automatically upper triangular if the vertices of the model digraph G(V,E) are indexed, from 1..|V|, in such a way that every vertex has a distinct index, and, if (u,v) is an edge, the index assigned to u is smaller than the index assigned to v. Such an ordering of the vertices is called a *topological ordering*. The vertices of an acyclic digraph can always be topologically ordered.

THEOREM (TOPOLOGICAL ORDERING) The vertices of a digraph G(V,E) can be ranked in topological order if and only if G is acyclic.

The proof is simple. The only if part is obvious. To prove sufficiency, observe that every digraph necessarily has at least one vertex of out-degree zero, such as one obtained by repeatedly extending any path until it becomes blocked. Similarly, there also exists some vertex v of in-degree zero. If we set Index(v) to 1, and repeat the same process on the acyclic subgraph G - v, each time incrementing the index assigned to the vertex of in-degree zero, eventually all of G will be indexed. By construction, whenever (u,v) is an edge of G, u must be indexed before v, so that Index(u) must be less than Index(v). This completes the proof.

A procedure for topologically numbering a digraph is as follows. We scan the adjacency list of the graph to determine the in-degrees of every vertex, in O(|V| + |E|) time, and then create a list of the vertices of in-degree zero, denoted the zero-list, in O(|V|) time. We then initialize a counter k to zero, select a vertex from the zero-list, label it with k (and increment k), and then decrement the in-degrees of every vertex on its adjacency list. Any vertices whose in-degrees are thereby reduced to zero, are also placed on the zero-list. We repeat this process until the zero-list is empty, which takes time O(|E|). If any vertices remain that were not put on the zero-list, G is not acyclic; otherwise G is acyclic and the labelling is a topological ordering of its vertices.

A recursive procedure Topological_Order for topologically indexing the vertices of an acyclic digraph is given below. From the viewpoint of Chapter 5, Topological_Order (G) can be considered as an enhanced depth first search. The procedure assumes the digraph is acyclic, and has O(|V| + |E|) performance, since each vertex and edge is visited only once. G(V,E) is represented by a linear array, wherein each vertex has a field, Index, where the topological rank of the vertex is stored. The procedure uses an auto-decrementing function Subtract_count which is initialized to |V| and decremented by 1 after each invocation.

```
Procedure Topological_Order (G)

(* Topologically order G *)

var  G: Graph
     v: 1..|V|
     Subtract_count: Integer function
```

weight edge of the form (v_1, u) or (t, u). By the definition of the Voronoi diagram, u must be a Voronoi neighbor of Vor(s,V) U Vor(t,V). Therefore, once again, whichever edge the algorithm adds to T is an edge of Dual(V). The general case follows the same idea. That is, the algorithm, as it proceeds, merely adds an edge of the dual of Vor(V) to T at each step. Since all the edges of the spanning tree constructed by the algorithm lie in the dual, the final spanning tree must be a subgraph of Dual(V), as was to be shown.

4-5 ACYCLIC DIGRAPHS

This section considers several applications of acyclic digraphs and the graphical properties they are based on: parts explosions and topological orders, deadlock avoidance and cycle detection, scheduling tasks under prerequisite constraints and longest paths, and register allocation for expression evaluation and digraph labelling.

4-5-1 BILL OF MATERIALS (TOPOLOGICAL SORTING)

A *parts explosion* is a method of describing the hierarchical structure of a composite product in terms of its components. An example is shown in Figure 4-8a where an acyclic digraph G(V,E) is used to model a parts explosion. Each vertex in the digraph corresponds to a part of the product or a subassembly of parts of the product and is labelled with the name of the part. There is an edge (x, y) if the part or subassembly y is a component of the assembly x. The edge is labelled with the number of copies of the part (subassembly) used in the parent part (assembly). A primitive part is by definition one containing no components. It is represented in the digraph model by a vertex of out-degree zero. The part represented by the parts explosion is called the root part. A *bill of materials matrix* M for the parts explosion is shown in Figure 4-8b. Component M(i,j) gives the multiplicity of j as a component of i.

We consider how to compute the number of primitive parts required to make a single composite part. The composite part has the nominal requirements list P = (p_1 ,..., p_n), where p_i gives the number of parts of type i required. Let k be the length of the longest path in the acyclic parts explosion digraph. The number of primitive parts required to realize P are determined by the following procedure. The final value returned in P is the list of primitive parts requirements. Refer to Figure 4-8c.

```
Procedure Parts_Requirement (M, P, k)

(* Returns, for the input parts requirements list P, the
   primitive parts requirements list, also in P *)

var  n: Integer constant
     M(n, n), P(n), i, k: Integer

for  i =  1 to k  do  Set P to  P M

End_Procedure_Parts_Requirement
```

three steps: (1) we define a planar (not merely euclidean) graph associated with the Voronoi diagram Vor(V), and denoted by Dual(V); (2) we find a minimum spanning tree of Dual(V), in O(|V| log |V|) time; and (3) we show a minimum spanning tree of Dual(V) is necessarily a minimum spanning tree of G(V,E).

The associated graph Dual(V) is defined as follows. The set of vertices of Dual(V), V(Dual(V)), equals V. Vertices v_i and v_j in V(Dual(V)) are adjacent if and only if the Voronoi polygons Vor(v_i,V) and Vor(v_j,V) of v_i and v_j share an edge of Vor(V). Dual(V) is a subgraph of G(V,E), and can be easily constructed from Vor(V) in O(|V|) time using its definition. Since Dual(V) is a planar graph, it has only O(|V|) edges; so the MST algorithm can find a minimum spanning tree of Dual(V) in O(|V| log |V|) time.

To prove that a minimum spanning tree of Dual(V) is automatically a minimum spanning tree of V, we use an alternative version of the MST algorithm devised by Prim. While the previous MST algorithm constructed a minimum spanning tree by greedily merging the component trees of a spanning forest; Prim's algorithm, in contrast, constructs a minimum spanning tree by expanding a (nonspanning) subtree in the greedy manner described by the following high level algorithm.

```
Procedure PRIM_MST(G,T)

(* Returns a minimum spanning tree for G in T *)

var  G,T: Graph
    v,v1: Vertex
       x: Edge

Set T to an arbitrary vertex v1 in V(G)

repeat   |V(G)| - 1   times

   Select the least weight x (= (u,v)) in E(G)
   such that u is in V(T) and v is not in V(T).

   Add x to T.
End_Procedure_PRIM_MST
```

This algorithm clearly constructs a spanning tree, which we leave it as an exercise to show is a minimum spanning tree. It then follows from the lemma below that a minimum spanning tree of Dual(V) is also a minimum spanning tree of G(V,E), as was to be shown.

Lemma. The tree constructed by Prim's algorithm lies in Dual(V).

The proof is as follows. The idea is to follow the algorithm in a step by step fashion, interpreting its operation in terms of the corresponding Voronoi diagram Vor(V). Thus, let (v_1,t) be the least weight edge of G which is incident with v_1. The vertex t belongs to the Voronoi polygon Vor(t,V) which borders Vor(v_1,V). Therefore, the edge (v_1,t) must be an edge of Dual(V). Consequently, the first edge added by the algorithm to T is an edge of the Dual(V). Similarly, at the next step, Prim's algorithm seeks the least

until |E(T)| = |V(G)| - 1

End_Procedure_MST

PATH COMPRESSION

Path Compression is a technique which can be used to improve the performance of a unary tree reresentation, and works as follows. After we find the root u of the tree containing a given node v, we make all the nodes along the unary tree path from v to u point directly to u. That is, for each node w on the path from v to u, we make u the parent of w. We shall also introduce a variation of the small-large rule where the so-called ranks of the roots of the unary trees are used for the small-large comparisons. (The term rank used here is not the same as the ranks to be defined in Section 4-6.) The ranks are initialized to 0 for each trivial root. Then, whenever a union is formed, the root of smaller rank is linked to the root of larger rank. The rank is incremented by 1 whenever a pair of trees of equal rank are joined; otherwise, the rank of the new root remains the same. If we combine both path compression and this version of the small-large rule, we improve the unary tree performance significantly. The performance bound is in terms of the inverse function of a very rapidly growing function called the Ackerman function A(i,j), defined for $i, j \geq 1$: $A(1,j) = 2^j$ for $j \geq 1$, $A(i,1) = A(i - 1, 2)$ for $i \geq 2$, and $A(i,j) = A(i - 1, A(i, j - 1))$ for $i, j \geq 2$. The function a(m,n), for $m \geq n \geq 1$, is then defined by $a(m, n) = \min \{i \geq 1 \mid A(i, G(m/n)) > \log n\}$, where G(m/n) denotes the greatest integer not larger than m/n. Practically speaking, a(m, n) is not larger than 4.

THEOREM (PERFORMANCE UNDER PATH COMPRESSION AND SMALL-LARGE RANK RULE) An intermixed sequence of m Root and Union operations on a unary tree on n elements takes time O(m a(m,n)). (We assume the tree is started in its trivial state on n nodes.)

The proof of this theorem is quite difficult and is discussed in Tarjan (1983) (other references at the end of the chapter). It follows from the theorem that if the edges of the graph are already sorted by edge weight, a minimum spanning tree can be found in O(|E|a(|V|,|E|)) steps. For an even faster, but more complicated, minimum spanning tree algorithm, see Section 4-6.

4-4 GEOMETRIC MINIMUM SPANNING TREES

The preceding section established an O(|E| log |V|) algorithm for finding a minimum spanning tree. We will now show how to use the Voronoi diagrams introduced in Chapter 2 to find a minimum spanning tree of a graph in the plane in O(|V| log |V|) steps, a substantial improvement over the previous algorithm. Assume V is a set of points in euclidean 2-space. Let G(V,E) be the induced complete euclidean graph on the points of V, where the edges of G(V,E) correspond to the euclidean line segments between the points of V. Recall that although G is embedded in the plane, it need not be a planar graph, since its edges can intersect at points other than the points (vertices) of V. We will show how to find a minimum spanning tree of G using Voronoi diagrams in

```
type  Heap = record
               H(|E|,2): 1..|V|
               Weight(|E|): Real
               Last: 0..|V|
             end
```

The edges of G, in heap order with respect to their weight, are stored in H. Their weights are stored in the same order in Weight. Last gives the index of the last element in the heap. Whenever we interchange heap elements in the process of maintaining the heap, we interchange both the entries in H and Weight.

The Unary Tree representation for disjoint sets is

```
type  Unary Tree  = record
                      Parent(|V|): 0..|V|
                      Size(|V|):   0..|V|
                    end
```

The pointers of the unary tree are stored in Parent. Initially, Parent(i) = i and Size(i) = 1, for i = 1..|V|. Generally, if vertex i is the owner (root) of a component, Size(i) equals the number of vertices in its component.

The enhanced MST algorithm follows. It uses the utility Create_Heap (G_Edge, G) to store the edges of G in the heap G_Edge. Create_Unary (T_Vert, G) initializes the disjoint set representation for the vertices of G in T_Vert. Initial_Graph(T,G) initializes the tree T using |V(G)|. Add(T, x) adds an edge x to T. The type Edge_Entry is as defined previously.

```
Procedure MST (G, T)

(* Returns a Minimum Spanning Tree for G in T *)

var  G, T: Graph
     G_Edge: Edge Heap
     T_Vert: Unary Tree
     x: Edge_Entry
     Rootx1, Rootx2: 1..|V|
     Root: Integer function

Create_Heap (G_Edge, G)
Create_Unary (T_Vert, G)
Initial_Graph (T, G)

repeat

  Delete (G_Edge, x)

  if    Root (T_Vert, Rootx1, x1) <> Root (T_Vert, Rootx2, x2)

  then  Union (T_Vert, Rootx1, Rootx2)
        Add (T, x)
```

The proof of the theorem is as follows. For convenience, we shall work with the equivalent relation,

If Height(T(S)) $\geq$ h then |T(S)| $\geq$ 2^h.

The proof is by induction on h. Assume that every unary tree of height h - 1, has size at least 2^{h-1}. We shall show that if T is a least cardinality tree of height h constructed by a sequence of operations of the type described in the theorem, T has at least 2^h vertices. It will then follow, by the definition of T, that every other tree of height h has at least that many vertices, completing the induction.

By supposition, T is the union of a pair of trees X and Y, whose sizes are less than |T|. Therefore, since T is the smallest tree of height h, X and Y must each have height less than h. Suppose without loss of generality that |Y| $\leq$ |X|. Therefore, we can assume that Y was appended to the root of X. Height(X) $\leq$ h - 1. If Height(Y) $\leq$ h - 2, the height of their union, namely, Height(T) $\leq$ h - 1, contrary to supposition. Therefore, Height(Y) must equal h - 1. It follows by induction that |Y| $\geq$ 2^{h-1}. Since |X| $\geq$ |Y|, then |X| $\geq$ 2^{h-1} also. Therefore, T must have at least 2^h vertices, as was to be shown. This completes the proof of the theorem.

All the disjoint sets considered in the MST algorithm are initially singletons: corresponding to the isolated vertices of a spanning forest with no edges. Figure 4-7 illustrates the sequence of disjoint sets obtained for the MST algorithm, following the small-large rule, for the example of Figures 4-5 and 4-6.

Figure 4-7 here

The number M of acyclicity tests required by the MST algorithm depends on the graph G being processed. M acyclicity tests take time O(M log |V|). If M is O(|E|), this is comparable to the time for the worst case heap requirements. If M is O(|V|), the operations take time O(|V| log|V|). If we combine this with the corresponding O(|E| + M log |E|) time for the heap operations, we obtain an overall O(|E| + M log |E|) time for the algorithm. Even in the worst case, the enhanced algorithm only takes O(|E| log|V|) time, while the unenhanced version takes time O(|E| |V|). Thus, the new data structure will always improve the performance of the algorithm, though the improvement is a function of the problem dependent parameter M.

ENHANCED MINIMUM SPANNING TREE ALGORITHM

We will now give another implementation of the minimum spanning tree algorithm using the enhanced heap and disjoint set data structures we have introduced. The heap and unary tree data types are defined as follows.

EFFICIENT ACYCLIC TEST

We will now show how to test whether an edge forms a cycle in O(log |V|) time. As we have observed, the forest constructed by the MST algorithm consists of components. Each component is a tree comprising a set of disjoint vertices. We can represent each disjoint set by a unary tree. The *unary tree* representation of a set S is a tree with one node for each member of the set. Each node has a unique parent node in the unary tree which the node points to, except for the root of the unary tree which points to itself. We consider the root of the unary tree as the representative or owner of the set represented by the tree. We shall introduce two unary tree functions

(1) Root (UT, rootx, x), and
(2) Union (UT, rootx, rooty),

where UT is a (collection of) unary tree(s). Root (UT, rootx, x) returns the root of the tree representing the set containing x in both Root and rootx. Union(UT, rootx, rooty) forms the union of the pair of disjoint sets whose unary trees have roots rootx and rooty.

Root (UT, rootx, x) is trivial to implement. We merely follow the pointers to parent nodes, starting with x, all the way to the root of the tree containing x. We can implement Union(UT, rootx, rooty) by merely making rootx point to rooty. The performance of the Root operation depends on the heights of the trees; so it is important to keep the tree heights small. The following rule ensures this.

Small-large rule: When forming a union of two unary trees, always make the root of the tree of smaller cardinality point to the root of the tree of larger cardinality.

To apply the small-large rule we need a field (Size) in the root of each unary tree which contains the number of vertices in the tree. Then, whenever we form the union of two trees, we update the Size field for the root of the union.

The performance of Union (UT, rootx, rooty) is O(1) since it merely requires setting rootx to point to rooty, or vice versa. On the other hand, Root (UT, rootx, x) takes an amount of time proportional to the height of the unary tree that represents the set x belongs to. But, this in turn depends critically on how we implement Union. We can show that the heights of the unary trees that result if we follow the small-large rule are always logarithmic in their size, giving O(log |E|) performance for Root (for a collection of unary trees of total cardinality |E|). In the following theorem Height(UT) denotes the height of a unary tree and |UT| denotes its number of vertices.

THEOREM (LOGARITHMIC HEIGHT UNARY TREES) Let M be a set of disjoint singletons. Let S be a subset of M constructed by applying a sequence of Union operations to the members of X and following the small-large rule. If T(S) denotes the unary tree representing S, then |T(S)| and Height(T(S)) satisfy

$$\text{Height}(T(S)) \leq \lg |T(S)|.$$

performance bounds can be obtained from the recurrence relations determined by recursive procedures. We summarize the analysis in the following theorem.

THEOREM (LINEAR TIME HEAP CREATION) Let X be a set of N real numbers. Then, a heap on X can be created in O(N) steps.

The proof is as follows. Let W denote the amount of work required to create a heap on X using the recursive algorithm described. We can consider W as a function either of the cardinality N of X or the height of the heap. If we consider W as a function of N, we obtain the recurrence relation

$$W(N) = 2\ W(N/2) + c\ \log(N).$$

This reflects the recursive construction of the heap from two subheaps of size N/2, and the subsequent O(log N) steps required to sift the root of the heap into its proper position. On the other hand, if we consider W as a function of the height H of the heap created, the recurrence relation is

$$W(H) = 2\ W(H - 1) + c\ H.$$

We will work with the recurrence in this form, since it is slightly more convenient to handle. If we feed this recurrence back into itself repeatedly, we obtain

$$W(H) = 2^{H-1}W(1) + c\ (H + 2\ (H - 1) + \ldots + 2^{i}(H - i) + \ldots \\ \ldots + 2^{H-1}(1)),$$

where c is a constant. The leading term in this expression is O(N), since N is at most 2^H - 1. The remainder of the expression must be summed carefully. If we replace H - i by j and set r equal to 1/2, then the summation becomes

$$2^H \sum_{j=1}^{H} j\ r^j.$$

Since the factor 2^H is O(N) and the summation is bounded, this term is also O(N). This completes the proof of the theorem.

Summarizing, we can create an |E| element edge heap in O(|E|) time, and delete an element from the same heap in O(log |E|) time. Therefore, if the MST algorithm uses M heap operations, then we can perform them in O(|E| + M log |E|) time. In the case that M is O(|V|), this is O(|E| + |V| log |E|) steps; while if M is O(|E|), it is O(|E| log |E|) or equivalently O(|E| log |V|) steps.

```
Set  H(1)  to H(Last)
Set  Last  to Last - 1
Set  Delete to True

(* Sift root element into correct heap position *)

Set I to 1

while  Last ≥ 2I  do  Swap H(I) with its smaller child
                      Set I to position of child

End_Function_Delete
```

Insert adds an element to the heap. We implement it by adding the new element at H(Last + 1) and sifting it upwards as long as its value is smaller than the value of its parent. Insert also takes O(log |E|) time.

Create Algorithm and Its Performance

Creating a heap by iterating the Update procedure takes O(|E| log |E|) time, since there are |E| updates and each update takes O(log |E|) time. But, this is just the time it takes to sort |E| edges by weight, which we are attempting to avoid. Actually, we can Create a heap in O(|E|) time, as we shall now show.

A heap may be defined recursively as consisting of an element Root containing the smallest value in the heap, together with subtrees rooted at the left and right children of Root which are also heaps. This definition suggests a recursive procedure for creating a heap. We will assume for simplicity that the heap has |E| equal to $2^k - 1$ elements, for some positive integer k.

The Create procedure is as follows.

(1) Divide the set of elements to be represented in the heap into three sets of size 1, $2^{k-1} - 1$, and $2^{k-1} - 1$.

(2) Recursively construct subheaps on each of the two sets of size $2^{k-1} - 1$.

(3) Make the subheaps created in (2) the left and right subheaps of the remaining element.

(4) Sift the root of the structure in (3) into its correct heap position using the same technique as in the Delete algorithm.

Upon completion, the smallest element of the data structure lies at the root, and the left and right subtrees of the root are heaps. Therefore, the overall structure is a heap. The performance of this procedure can be analyzed using recurrence relations. The analysis is a nice illustration of how

EFFICIENT MINIMUM SELECTION

It takes time O(|E| log |E|), or equivalently O(|E| log |V|), to sort the edges by weight. Since we may examine as few as |V| - 1 edges, sorting all |E| edges seems excessive. But, how can we improve on this? If we use ordinary selection to get the next smallest edge in O(|E|) time and repeat this M times (in the case where the tree is constructed after examining M edges), the cost of this part of the algorithm is O(M |E|), which is inferior to fast complete sorting since M is at least |V| - 1. Moreover, the overall performance of the algorithm will improve only if faster edge selection is matched by faster cycle testing. With this in mind, we will now consider the advantages of a heap for minimum selection.

We will assume familiarity with the definition of a heap. Recall that a *heap* is a balanced binary tree, ordered on sort keys located at each of its nodes. A heap is usually represented by an array H(1..N), where H(i) acts as the parent of H(2i) and H(2i+1). The basic *heap operations* are

(H1) Create (a heap),
(H2) Find_min (find the minimum element in the heap),
(H3) Delete (the least element in the heap and restore the heap),
(H4) Insert (a new element into a heap),
(H5) Member (test if an element is in a heap), and
(H6) Change (the value and if necessary the position of an existing heap element).

We have already described the Member and Change operations under Dijkstra's algorithm. The MST algorithm uses only the Create and Delete operations. We will review the Delete, Create, and Insert operations. We use the following type definition, which will be slightly modified later when we deal with the edge heaps needed for the MST algorithm.

```
type  Heap =  record
                H(N): Integer
                Last: 0..|E|
              end
```

The Delete procedure follows. For an |E| edge heap, the procedure has performance O(log |E|).

```
Function Delete (A, Small)

(* Returns the least heap element in Small and deletes it, or
   fails *)

var  A: Heap
     I, Small: Integer
     Delete: Boolean function

if   Last = 0  then  Set Delete to False
                     return

Set  Small to H(1)
```

also in S. Since S is a spanning tree, x must be a chord of S, and so S U x must contain a cycle C. Since T is acyclic, there must exist some edge x' on C which lies in S but not in T. We will show the spanning tree S U x - x' weighs no more than S.

We will derive a contradiction from the assumption that x' weighs less than x. If x' weighs less than x, then x' must have been considered by the algorithm for inclusion in T before x was considered. But, by the definition of x, every edge in T added prior to x also lies in S. Since x' is not in T, the subgraph of T formed by the minimum spanning tree algorithm up to the point when x' was examined together with x', must have contained a cycle C'. But, all the edges on C', including x', lie in S, which is acyclic. Therefore, x' could not have caused a cycle in T. Therefore, the algorithm would have added x' to T, contrary to our assumption that x' is not in T. It follows that x' must weigh at least as much as x.

Therefore, the tree S U x - x' weighs no more than S and so is a minimum spanning tree. This tree also has one more edge in common with T than S does. Consequently, if we iterate this process, we will eventually arrive at a minimum spanning tree which is identical to T. This completes the proof.

PRELIMINARY PERFORMANCE ANALYSIS

We will give a rudimentary analysis of the performance of the algorithm in order to identify bottlenecks where special data structures might prove advantageous.

First, observe that although there are only |V| - 1 edges in the MST tree, we may have to examine as many as |E| edges in order to determine which edges belong in T. Therefore, the **repeat** loop in the algorithm may be executed as many as |E| times. At each execution of the loop body, we select the next least weight edge from a shrinking list of edges of cardinality as large as O(|E|). If the edges are sorted by weight, which takes (preprocessing) time O(|E| log |E|), each edge selection can be done in O(1) time.

Each execution of the loop also tests whether adding an edge introduces a cycle. Because the underlying graph T is always a forest, this test can be done easily. At any point, T is acyclic and so consists of a collection of components which are trees. Therefore, adding an edge to this forest can introduce a cycle only if both the endpoints of the edge lie in the same component of the forest. Each component of the forest comprises a set of vertices, and so we can represent the forest, at least for testing acyclicity, as a collection of disjoint sets. To determine whether adding an edge introduces a cycle, we need only test whether both the endpoints of the edge lie in the same disjoint set. Since these sets can be O(|V|) in size, this test can take O(|V|) time.

This preliminary analysis suggests the complexity of the algorithm is O(|V||E| + |E| log |E|), which is just O(|V||E|). Of course, we can drastically improve this performance by choosing suitable data structures for the two critical steps of the algorithm.

1..|E|. Weight(i) gives the length of the i^{th} edge, i = 1..|E|. For convenience, we also use a type Edge_Entry defined as

```
type  Edge_Entry = record
                     x1, x2: 1..|V|
                     xweight: Real
                   end
```

We assume that G is connected and has an order of at least 2. If H is a graph and (u,v) is an edge, H U (u,v) has the natural interpretation that if either vertex u or v is not in V(H), it is (they are) added to the set of vertices; then the edge is added. The statement of the algorithm is as follows.

```
Procedure MST(G,T)

(* Returns a minimum spanning tree for G in T *)

var  G, T: Graph
     x: Edge_Entry

Set |V(T)| to |V(G)|
Set |E(T)| to  0

repeat

   Select the next smallest edge x in E(G) - E(T)

   if  T U x  is acyclic  then  Set T to T U x

until   |E(T)| = V(G) -  1

End_Procedure_MST
```

Before proceeding to a refinement of the algorithm, we will establish its correctness.

THEOREM (CORRECTNESS OF MINIMUM SPANNING TREE ALGORITHM) Let G(V,E) be a connected weighted graph, |V| > 1, and let T be the weighted spanning tree of G constructed by the MST algorithm. Then, T is a minimum spanning tree of G.

The proof is as follows. First, observe that the subgraph T constructed by the algorithm is certainly a spanning tree. Then, let S be an arbitrary minimum spanning tree in G. We will show S and T have the same total edge weight; so T must also be a minimum spanning tree of G. Our approach will be to iteratively add and remove edges from S, without increasing its total edge weight, but increasing the number of edges it has in common with T, until eventually it becomes identical to T.

If T and S are not identical, there is some edge in T which is not in S, and conversely. Let x be the first edge the algorithm adds to T which is not

The proof of the correctness of this straightforward algorithm is nontrivial.

4-3 MINIMUM SPANNING TREES

The minimum spanning tree problem is a classical situation where the greedy method yields an optimal solution. It also illustrates how the careful design of data structures can substantially improve the performance of an algorithm. We will use heaps and disjoint sets represented as trees to implement a fast version of the algorithm. The analysis of the performance of these data structures is also instructive. The analysis of the disjoint sets uses a classical tree-height versus tree-size argument; while the analysis of heap creation illustrates the application of recurrence relations in performance analysis.

We define the minimum spanning tree problem as follows. Given an undirected graph G with real-valued edge weights assigned to each of its edges, find a spanning tree T of G that has minimum total edge weight. The problem has a simple interpretation. Consider the complete graph K(n), where each vertex corresponds to a city and the weight of an edge corresponds to the cost of establishing a direct communication link between the pair of cities corresponding to the endpoints of the edge. A minimum spanning tree on K(n) corresponds to a least cost communication network connecting all the cities.

The optimal solution to the minimum spanning tree problem is surprisingly simple. We merely construct a tree, starting with an empty tree, iteratively adding edges to the tree in increasing order of edge weight, excluding an edge only if it forms a cycle with the previously added edges. We terminate the process when a spanning tree is formed. Figures 4-5 and 4-6 illustrate the idea. The algorithm is greedy because it optimizes the possible improvement in the objective function, here the weight of the spanning subgraph, at each successive step of the algorithm, subject only to the constraint that no cycles are formed.

Figures 4-5 and 4-6 here

MINIMUM SPANNING TREE ALGORITHM

We will initially describe the algorithm from a high-level point of view. Later, we will give a more detailed version using the heap and disjoint set data structures to be developed. The data type of the graph is

```
type  Graph = record
                |V|: Integer
                |E|: Integer
                Edges(|E|,2): 1..|V|
                Weight(|E|): Real
              end
```

The |E| x 2 array Edges contains the edge list for G. The i^{th} edge (u,v) is represented by the i^{th} row of the array Edges (that is, (Edges(i,1), Edges(i,2)), where Edges(i,1) equals u and Edges(i,2) equals v, for i =

minimizes W + (f_1 + f_2), the weight of the optimal tree on f_1 ,..., f_n, as was to be shown.

OPTIMAL FILE MERGING

The Huffman decoding tree has another application, to an apparently unrelated problem: How to merge a collection of sorted sequential files in such a way as to minimize the number of (file) records moved? For example, suppose we interpret the table in Figure 4-1a as referring to a list of files and interpret the frequencies in the table as referring to the number of records per file. Furthermore, rather than interpreting the tree in Figure 4-1b as a decoding tree, we think of it as defining the order in which to merge the files: Thus, first merge file-c and file-e; then merge the result of that merger with file-d; then merge that result with file-a; and finally merge that result with file-b. We define the total cost of the merger operations as the total number of records moved during the merging process. For example, in merging file-c and file-e, 9 (= 4 + 5) records are moved. In general, the total number of records moved depends both on the sizes of the files and the order in which they are merged. The cost is easily seen to equal the internal path length of the binary merging tree (where we consider the endpoint vertices as external vertices of the tree). Therefore, in order to minimize the cost, we merely construct a merge tree using the same procedure as for determining Huffman codes.

4-2-4 PRECEDENCE TREES FOR MULTIPROCESSOR SCHEDULING

The scheduling of tasks on a uniprocessor so as to optimize system performance parameters, like turn-around time, has been the subject of much investigation. It is far more complicated to properly schedule tasks in a multiprocessor environment, but there are some techniques available. A scheduling algorithm due to Hu (see Coffman and Denning (1973)) gives an optimal schedule under the following simplified circumstances. Suppose T_1 ,..., T_n are a set of tasks, each of which takes a unit time to execute. Suppose there are M homogeneous processors on which the tasks can be scheduled. Suppose also that the tasks are constrained by precedence relations defined by a rooted tree. Thus, if (T_i,T_j) is an edge of the precedence tree, task T_j cannot be started until task T_i has been completed. Figure 4-4a gives an example of a precedence tree. The following algorithm minimizes the completion time of such a set of tasks.

Figure 4-4 here

Hu's Scheduling Rule: Whenever a processor becomes free, start the next task all of whose precedent tasks have already completed *and* which has the highest level in the precedence tree of all tasks that have not yet been started.

Figure 4-4 illustrates the application of the algorithm to a set of 12 tasks, under the constraints of the precedence tree shown in Figure 4-4a and with M = 3 processors available. Figure 4-4b shows the successively scheduled sets of tasks. Figure 4-4c gives the *Gannt chart* for the schedule.

(2) Select two trees from S whose roots have the two smallest frequencies, call them f_1 and f_2, and form a new tree whose root has frequency $f_1 + f_2$, and has the roots of the selected trees as children.

(3) Remove the trees for f_1 and f_2 from S, and add the new tree of frequency $f_1 + f_2$ to S.

Refer to Figures 4-1 to 4-3 for an example of the procedure.

Figure 4-3 here

The correctness and performance of the procedure is established by the following theorem.

THEOREM (PERFORMANCE AND OPTIMALITY OF HUFFMAN ENCODING) Let L be a list of n distinct words (of some uniformly bounded length b, independent of n), and suppose the frequency f(w) of each word w in L is known with respect to a given text. Then, the Huffman encoding of the list can be determined in O(n log n) time, and the encoding minimizes the length of the compressed text.

The proof is as follows. Let L = $\{w_1, \ldots, w_n\}$ be the list of words, of frequencies f_i, i = 1 ,..., n, respectively. Consider first the performance of the algorithm. Observe that we can create a heap, ordered on word frequencies, in O(n) time. (For a proof of this and the subsequent heap properties used here, refer to the section on minimum spanning trees.) Using a heap, it takes the algorithm O(log n) steps to find and remove the two least frequent words in L, which we can denote without loss of generality by f_1 and f_2. It then takes an additional O(log n) steps to insert the new value $f_1 + f_2$ in the heap. At this point, there are n - 1 values in the heap. We can assume by induction that it takes O((n - 1)log(n - 1)) steps to create the Huffman tree on these frequencies. If we combine this with the preceding estimates, it follows that Huffman's algorithm can be implemented in O(n log n) time.

To establish the correctness of the procedure, we argue as follows. As we have observed, the cost of the encoding equals the sum of the weights of the internal vertices of the tree. Suppose f_1 and f_2 are the two smallest frequencies of the endpoints of the tree. We can assume that the endpoints for f_1 and f_2 are each at a maximum distance from the root of the tree; otherwise, we could reduce the weighted internal path length of the tree by exchanging, say, f_1 with a more distant endpoint (of necessarily greater frequency). We can also assume that f_1 and f_2 have the same parent; otherwise, we could exchange f_2 with the current sibling of f_1 again without increasing the weighted internal path length of the tree. Therefore, the parent vertex of f_1 and f_2, which must be of frequency $f_1 + f_2$, is an internal vertex of some optimal tree, which therefore has weight W + ($f_1 + f_2$), where W equals the sum of the labels of the other internal vertices of that tree. But, these are also precisely the internal vertices of a tree on the endpoint frequencies: $f_1 + f_2$, f_3 ,..., f_n, which we can assume by induction the algorithm finds correctly. Such a tree minimizes W, and so it also

$$\sum_{i=1}^{n} f_i d_i \ . \qquad (4\text{-}9)$$

We would like to determine a representation that minimizes the *compressed text length* (4-9).

The most obvious approach is to greedily represent the most frequent words in L with the shortest representations. For example, if L = {a, b, c} with frequencies 3, 2, and 1 respectively, we could greedily represent a by the bit string 0, b by 1, and c by 00. However, the problem with this representation is that the encoded text cannot be unambiguously decoded. For example, does the bit string 00 represent the text aa or c? We can guarantee unique decodability if we require the representation to satisfy the *prefix property*: the representation assigned to any word must not match the leading part (prefix) of the representation assigned to any other word.

Any binary tree decoding scheme, in which the endpoints of the tree correspond to the words represented, and a left (right) link in the path through the tree from the root to an endpoint corresponds to a 0 (1) bit in the representation of the word associated with the endpoint, defines a representation satisfying the prefix property. Refer, for example, to the binary tree in Figure 4-1b where an a is represented by the bit string 11, a b by the bit string 0, etc. Observe that the length d_i of the representation of a word w_i equals the distance of the endpoint corresponding to w_i from the root of the decoding tree. The words can be encoded initially using a lookup table whose entries give the word/representation correspondence.

Figure 4-1 here

There is a simple way to label a binary decoding tree that provides a useful way to interpret the cost (4-9) of a representation. Thus, suppose we label each endpoint of the tree with the frequency of its associated word; then label the parent of two endpoints with the sum of the labels of its children; and in general, for any internal vertex v (a vertex which is not an endpoint) label v with the sum of the labels of its children. Then, it is easy to see that the sum of the labels of the internal vertices equals the length (4-9) of the compressed message. Refer to Figure 4-2 for an illustration. Incidentally, (4-9) also equals the weighted internal path length of the decoding tree.

Figure 4-2 here

The following procedure constructs a binary decoding tree which minimizes the cost (4-9) for a set of words L of known frequency. The representation is called the *Huffman representation*.

(1) Sort the words in L in order of frequency. For each word w in L, create a rooted tree consisting of an isolated vertex labelled with the frequency of w. Denote the resulting set of trees by S.

Repeat steps (2) and (3) until S is exhausted.

in less than O(n log n) time.

Consider an arbitrary comparison sorting algorithm. The inputs to the algorithm are arrays of n elements whose values are the keys to be sorted. For convenience, we will assume that the keys are distinct. We may think of the function of the sorting algorithm as finding that unique permutation of any input array which sorts the array. If the elements are distinct, there are n! possible sorting permutations, any one of which may be the sorting permutation for a particular input.

THEOREM (COMPARISON SORT LOWER BOUND) Any comparison sort on n elements takes at least O(n log n) steps.

The proof is as follows. The sorting algorithm makes some sequence of data and algorithm dependent comparisons or decisions until the correct permutation has been identified (the array has been sorted). Though not explicit, the permutation is, nonetheless, always implicit in the operation of the algorithm. Thus, regardless of how the algorithm is implemented, we can interpret its operation as advancing through a binary decision tree. The internal vertices of the decision tree represent the binary decisions made by the algorithm. The endpoints of the tree represent the possible outcomes of an activation of the algorithm: one of the various possible sorting permutations. The size of the tree must be at least n!, since there are n! endpoints in the tree. Consequently, the height of the tree must be at least log (n!). By Stirling's Formula, a well-known asymptotic approximation to n!, log (n!) is asymptotic to O(n log n). This completes the proof of the theorem. 4-2-3 MERGE TREES

4-2-3 MERGE TREES

DATA COMPRESSION

An interesting application of trees is in the design of representations for compressing data. Suppose that $L = \{w_i, i = 1, ..., n\}$ is a list of words appearing in a text, and that word w_i has frequency f_i in the text. The words may be of fixed or varying length, or even single characters. If len(w) denotes the length in bits of the usual representation for w, the length of the text (from which the frequency statistics were derived) is

$$\sum_{i=1}^{n} f_i \, \text{len}(w_i). \qquad (4\text{-}8)$$

For characters, the representations are usually fixed-length binary codes; while for words, the representations are typically of variable length but are independent of the frequency of the words; so that it may be possible to compress the text by assigning frequency dependent representations. If we represent the word w_i, $i = 1, ..., n$, by a bit string of length d_i, the length of the text using this representation becomes:

while the probability of access for each external vertex is the same for each external vertex. Then, s(n) and u(n) are both O(log n).

The proof is as follows. By assumption, any of the n + 1 possible types of failed searches are equally likely. Therefore, since the key we are accessing is (in our random model) equally likely to have been the first key, the second key, the third key, etc., inserted in the tree, and since the number of comparisons that are needed to find a key is one more than the number of comparisons needed when the key was first inserted, it follows that s(n) satisfies:

$$s(n) = 1 + (u(0) + u(1) + \ldots + u(n - 1))/n . \quad (4\text{-}2)$$

There is a simple relation between u(n) and s(n) that will allow us to obtain a recurrence relation involving u (or s) alone. First, observe that by definition, s and u satisfy

$$s(n) = 1 + I(n)/n, \quad (4\text{-}3)$$

$$u(n) = E(n)/(n + 1), \quad (4\text{-}4)$$

from which it follows from (4-1) that

$$s(n) = (n + 1)u(n)/n - 1. \quad (4\text{-}5)$$

Combining (4-2) and (4-5), we obtain

$$(n + 1)u(n) = 2n + u(0) +\ldots+ u(n - 1). \quad (4\text{-}6)$$

Subtracting (4-6) for u(n) from (4-6) for u(n + 1), we obtain

$$u(n + 1) = u(n) + 2/(n + 2), \quad (4\text{-}7)$$

from which it easily follows that u(n) is O(log n). By (4-5), s(n) is also O(log n). This completes the proof of the theorem.

A more precise result is given in Devroye (1986) who shows that the average height of a binary search tree on n vertices is asymptotic to c ln (n), where c = 4.31107... . Flajolet and Odlyzko (1982) show that for rooted binary trees, where every vertex other than the root has either two children or is an endpoint, and where the tree is of "size" n, where n is the number of vertices in the tree with two children, the average height of the tree is 2(ãn)(1/2).

4-2-2 ABSTRACT MODELS OF COMPUTATION

Any sorting technique that sorts elements on the basis of a binary comparison of the (complete) sort keys is called a *comparison sort*, in contrast to a radix sort which makes sorting decisions on the basis of key fragments. Heapsort is a well-known comparison sort which sorts n elements in O(n log n) key comparisons and data movements. Using a theoretical model of comparison sorting which models the essential elements of any possible comparison sort, it is possible to show that no comparison can sort n elements

T, the search terminates at a null pointer. For convenience, we will introduce special vertices, called *external vertices*, at each null pointer. The *internal path length* of T is then defined as the sum of the lengths of the paths from the root of T to its internal vertices; while the *external path length* of T is defined as the sum of the lengths of the paths from the root of T to its external vertices. The *average internal path length* of T equals the internal path length divided by the number of internal vertices; while the *average external path length* of T equals the external path length divided by the number of external vertices.

If each key (vertex) is weighted according to the frequency with which it is accessed, the sum of the lengths of all search paths from the root of T to all its internal vertices, weighted according to the frequency with which they are accessed, is called the *weighted internal path length* of T. If estimates are available for the frequencies with which each external vertex is accessed (corresponding to the frequency of a failed search for a given range of nonkeys), we can define the *weighted external path length* of T analogously. The optimal Huffman encoding technique described in a subsequent section, and the optimal search tree organization considered in Chapter 2, optimize the weighted path lengths of different tree models.

The performance of a binary search algorithm can be characterized in terms of the average internal and external path length of the tree. However, in order to model average behavior meaningfully, we need an appropriate model of randomness for trees. Since the trees we are interested in are generated by the standard binary tree insertion algorithm, the appropriate random model is to consider the trees produced by randomly permuting the possible keys (1..n) and inserting them in permuted order, starting, of course, with an empty tree. Each permutation is equally likely and different permutations may lead to the same tree, so the likelihood of a given tree will be proportional to the number of permutations that give rise to the tree. Let us denote by s(n) the average number of comparisons needed by a successful key search in a given random tree, and by u(n) the average number of comparisons needed by an unsuccessful key search. Denote the internal path length for a given random tree with n keys by I(n), and its external path lengths by E(n).

There is a simple relationship between I(n) and E(n)

$$E(n) = I(n) + 2n. \qquad (4\text{-}1)$$

To prove this, we first observe that (4-1) is trivially true for a tree with one vertex. In general, when a new vertex v is added to an existing binary tree at distance x from the root of the tree, the internal path length is increased by x, while the external path length is decreased by x (because of the removal of the original external vertex) and then increased by 2x + 2 (because of the addition of two new external vertices), for a net increase of x + 2. Thus, the difference between E and I is increased by 2 every time a vertex is added, from which (4-1) follows.

The following theorem establishes the average performance of binary search.

THEOREM (LOGARITHMIC SEARCH TIMES) Let T be a random binary search tree of order n. Suppose the probability of access of an internal vertex (which contains a distinct integer key on 1..n) is the same for each internal vertex;

separate the vertices of T (and G) into disconnected parts X and X^c, thus defining a cut. Furthermore, this is a minimal cut because if we leave any of its edges in G, the resulting graph is still connected (adding w would reconnect T; adding a chord would reconnect the parts of T disconnected by removing w). In analogy with fundamental cycles, we call such a minimal cut a *fundamental minimal cut* of G. The analog for fundamental minimal cuts of the representation theorem for fundamental cycles is given in the following theorem.

THEOREM (FUNDAMENTAL MINIMAL CUTS REPRESENTATION) Let G(V,E) be a connected graph, and let T be a spanning tree of G. Then, every minimal cut m in G is the symmetric difference of the fundamental minimal cuts determined by the edges of m lying in T.

Cycles are symmetric differences of fundamental cycles, and minimal cuts are symmetric differences of fundamental minimal cuts; circuits are unions of edge disjoint cycles, and cuts are unions of edge disjoint minimal cuts. That is, we have the following.

THEOREM (CIRCUIT AND CUT REPRESENTATION) Let G(V,E) be a connected graph. Then, every circuit c in G is the union of edge disjoint cycles, and every cut m in G is the union of edge disjoint minimal cuts.

TREES AS MODELS

Trees and their invariants are used pervasively in the analysis of algorithms. Tree invariants such as height and path length determine the performance of search tree algorithms, while attempts to enhance the performance of search algorithms are often based on procedures that affect graph-theoretic invariants. For example, balancing algorithms minimize the height of binary search trees in order to enhance their performance. N-ary search trees (like B-trees) reduce tree height in exchange for increased vertex degree. Tree algorithms arise naturally in the compilation of algebraic expressions. Trees also occur as theoretical models of computation, such as the sort trees used to obtain lower bounds on the speed with which sorting can be done. Well-known techniques for optimal merging and data compression also use tree based algorithms. We shall consider some of these applications in the following.

4-2-1 SEARCH TREE PERFORMANCE

The performance of binary search trees can be analyzed in terms of the graph-theoretic properties of trees. For example, the height of a tree gives a worst case bound on the performance of a binary search, but we can estimate the average length of a binary search using the concept of the path length of a tree.

Let T be a binary search tree. When T is accessed for a key lying in T, the search terminates with a match at a vertex of T, which we will call an *internal vertex*. On the other hand, when T is accessed for a key not lying in

THEOREM (COMPLETE BINARY TREES) A complete balanced binary tree of height h has $2^h - 1$ vertices. Equivalently, a complete balanced binary tree of order |V| (where |V| + 1 is a power of 2) has height lg(|V| + 1).

An N-ary tree is a generalization of binary trees where we allow each vertex to have as many as N ordered children. In a complete balanced N-ary tree, every endpoint has the same level. We have

THEOREM (COMPLETE N-ARY TREES) A complete balanced N-ary tree of height h has

$$(N^h - 1)/(N - 1)$$ vertices.

Equivalently, a complete balanced N-ary tree of order |V| (where (N - 1)|V| + 1 is a power of N) has height

$$\log_N ((N - 1)|V| + 1).$$

SPANNING TREES, FUNDAMENTAL CYCLES, AND MINIMAL CUTS

There are simple, but important and well-known relations between the spanning trees, cycles, and edge disconnecting sets of a graph. To describe these relations, we will introduce some terminology. Let G(V,E) be a connected graph and let T be a spanning tree of G. An edge of G not lying in T is called a *chord* of T. Each chord of T determines a cycle in G, called a *fundamental cycle*; namely, the cycle caused by adding the chord to T. Any cycle in G can be represented in terms of these fundamental cycles. Let us define the *symmetric difference* of two paths P_1 and P_2 (including closed paths) as the graph that results by removing any edges that P_1 and P_2 have in common as well as any isolated vertices that result after the removal of these edges.

THEOREM (FUNDAMENTAL CYCLE REPRESENTATION) Let G(V,E) be a connected graph and let T be a spanning tree of G. Then, every cycle c in G can be represented as the symmetric difference of the fundamental cycles determined by the edges of c which are chords of T.

There is an analogous theorem for certain sets of edge that disconnect G. Let G(V,E) be a connected graph. An *edge disconnecting set* is a set of edges whose removal disconnects G. A *minimal edge disconnecting set* is an edge disconnecting set not properly containing another edge disconnecting set. A minimal edge disconnecting set is also called a *cutset*. Equivalently, a cutset is an edge disconnecting set whose removal disconnects G into exactly two components. If X is a nonempty set of vertices of G, the set of edges in E(G) of the form $\{(x, x') \mid x \text{ in } X, x' \text{ in } X^c\}$ is called a *cut* and is denoted by (X, X^c). A cut which does not properly contain another cut is called a *minimal cut*. Equivalently, a minimal cut is a cut whose removal disconnects G into exactly two components. Minimal cuts are cutsets, and conversely.

Just as a unique (fundamental) cycle is created when we add a chord to a spanning tree, similarly a unique minimal cut is created when we add a spanning tree edge w to the set of chords (of the spanning tree). That is, removing all the chords of T leaves G connected, since the spanning tree T still remains. Then, if we further remove the spanning tree edge w, we

TREES AND ACYCLIC DIGRAPHS

4-1 BASIC CONCEPTS

A *tree* is an acyclic connected graph. The following theorem summarizes the basic properties of trees:

THEOREM (CHARACTERIZATIONS OF TREES) Let G(V,E) be a graph. Then G is a Tree if and only if one of the following properties hold:

(1) G is connected and |E| = |V| - 1,
(2) G is acyclic and |E| = |V| - 1,
(3) There exists a unique path between every pair of vertices in G.

An *endpoint* of a tree is a tree vertex of degree one. A nontrivial tree has from 2 to |V| - 1 endpoints. A *center* of a tree is a tree vertex of minimum eccentricity. A tree has exactly 1 or 2 centers. A *star* is a tree of diameter 1 or 2. An arbitrary acyclic graph is called a *forest*. A forest with k components has |V| - k edges.

A *rooted tree* is a tree in which we identify a distinguished vertex v which we call the root. We then define the *level* of a vertex: vertices at distance i from the root lie at Level i + 1; v itself lies at Level 1. The *height* of the rooted tree is defined as its maximum level.

DIRECTED TREES

We call a digraph a *tree* if its underlying undirected graph is a tree in the undirected sense. We call a vertex v a *root* of a digraph G if there are directed paths from v to every other vertex in G. We call a digraph a *directed tree* or *arborescence* if it is a tree and contains a root. We call a vertex of out-degree zero in a directed tree an *endpoint*.

We use genealogical terminology to describe the relations between the vertices in a directed tree. Thus, if (u,v) is a (directed) edge, then u is called the *parent* or *father* of v, and v is called the *child* or *son* of u. If there is also an edge (u,w), then v and w are called *siblings*. If there is a path from a vertex u to a vertex v, then u is called an *ancestor* of v, and v is called a *descendant* of u. The subgraph of a directed tree T induced by a vertex u and all its descendants is called the *subtree rooted at u* and is denoted by T(u).

ORDERED TREES

An *ordered tree* is a directed tree in which the set of children of each vertex is ordered. A *binary tree* is an ordered tree in which no vertex has more than two children. One of the children is called the *left child*, while the other is called a *right child*. The subtree rooted at the left child of v is called the *left subtree* of v, and the one at the right is the *right subtree* of v. In a *complete binary tree*, every vertex has either two children or none. In a *balanced complete binary tree*, every endpoint has the same level. We have: