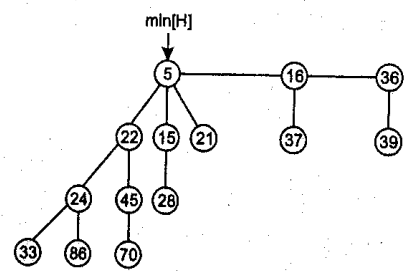
Exercise

1. Prove that, for a node x in a Fibonacci Heap, if c_i is the ith youngest child of x, then rank of c_i is at least $i-2$.
2. Prove that the rank of any node in a Fibonacci heap is $O(\lg n)$.
3. Suppose that a root x in a Fibonacci heap is marked. Explain how x came to be a market root. Argue that it doesn't matter to analysis that x is marked, even though it is not a root that was first linked to another node and then lost one child.
4. Show that if only the mergeable-heap operations are supported, the maximum degree $D(n)$ in an n-node Fibonacci heap is at most $\lfloor \lg n \rfloor$.
5. Suppose we generalize the cascading-cut rule to cut a node x from its parent as soon as it loses its kth child, for some integer constant k. For what values of k is $D(n) = O(\lg n)$?
6. Create a Fibonacci-heap for the following list $\langle 20,10,5,30,35,55,25,45,36,32 \rangle$.
   After creation, change the value of 45 to 23.
7. Decrease the key 45 to 34 in a Fibonacci heap shown in the Fig. Also find the resultant fibonacci heap after decreasing the key.



CHAPTER 20

Data Structures for Disjoint Sets

20.1 Introduction

An important application of the tree is the representation of sets, where "n" distinct elements are needed to be grouped into a number of disjoint sets.

A ***disjoint-set data structure*** maintains a collection $S = \{S_1, S_2, \ldots, S_k\}$ of disjoint dynamic sets. Each set is identified by a *representative* which is some member of the set. If we have two sets S_x and S_y, $x \neq y$ such that $S_x = \langle 3,4,5,6,7 \rangle$ and $S_y = \langle 1,2 \rangle$ then these sets are called disjoint sets as there is no element which is common in both sets.

20.2 Disjoint-Set Operations

In the dynamic-set implementations, an object represents each element of a set. Letting x denote an object, we wish to support the following operations.

- **MAKE-SET(x)** creates a new set whose only member (and thus representative) is pointed to by x. Since the sets are disjoint, we require that x not already be in a set.

- UNION(x, y) unites the dynamic sets that contain x and y, say S_x and S_y into a new set that is the union of these two sets. The two sets are assumed to be disjoint prior to the operation. The representative of the resulting set is some member of $S_x \cup S_y$, although many implementations of UNION choose the representative of either S_x or S_y, as the new representative. Since we require the sets in the collection to be disjoint, we "destroy" sets S_x and S_y removing them from the collection S.
- FIND-SET(x) returns a pointer to the representative of the (unique) set containing x.

The running times of disjoint-set data structures in terms of two parameters : n, the number of MAKE-SET operations, and m, and the total number of MAKE-SET, UNION, and FIND-SET operations. Since the sets are disjoint, each UNION operation reduces the number of sets by one. After $n-1$ UNION operation, therefore, only one set remains. The number of UNION operations is thus at most $n-1$. Note also that since the MAKE-SET operations are included in the total number of operations m, we have $m \geq n$.

20.3 UNION-FIND Algorithm

A **union-find algorithm** is an algorithm that performs two useful operations on such a data structure :

- **Find.** Determine which set a particular element is in. Also useful for determining if two elements are in the same set.
- **Union.** Combine or merge two sets into a single set.

Because it supports these two operations, a disjoint-set data structure is sometimes called a ***merge-find set.***

20.4 Applications of Disjoint-set Data Structures

One of the many applications of disjoint-set data structures arises in determining the connected components of an undirected graph.

The procedure CONNECTED-COMPONENTS that follows uses the disjoint-set operations to compute the connected components of a graph. Once CONNECTED-COMPONENTS has been run as a preprocessing step, the procedure SAME-COMPONENT answers queries about whether two vertices are in the same connected component. The set of vertices of a graph G is denoted by $V[G]$, and the set of edges is denoted by $E[G]$.

```
CONNECTED-COMPONENTS(G)
for each vertex v ∈ V[G]
    do MAKE-SET(v)
for each edge (u,v) ∈ E[G]
    do if FIND-SET(u) ≠ FIND-SET(v)
        then UNION(u,v)
```



```
SAME-COMPONENT(u,v)
if FIND-SET(u) = FIND SET(v)
    then return TRUE
    else return FALSE
```

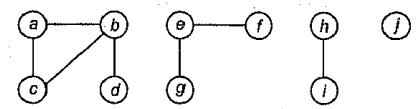


Figure 20.1 A graph with four connected components : $\{a,b,c,d\}$, $\{e,f,g\}$, $\{h,i\}$, and $\{j\}$. The collection of disjoint sets after each edge is processed.

Edge processed	*Collection of disjoint sets*									
initial sets	$\{a\}$	$\{b\}$	$\{c\}$	$\{d\}$	$\{e\}$	$\{f\}$	$\{g\}$	$\{h\}$	$\{i\}$	$\{j\}$
(b,d)	$\{a\}$	$\{b,d\}$	$\{c\}$		$\{e\}$	$\{f\}$	$\{g\}$	$\{h\}$	$\{i\}$	$\{j\}$
(e,g)	$\{a\}$	$\{b,d\}$	$\{c\}$		$\{e,g\}$	$\{f\}$		$\{h\}$	$\{i\}$	$\{j\}$
(a,c)	$\{a,c\}$	$\{b,d\}$			$\{e,g\}$	$\{f\}$		$\{h\}$	$\{i\}$	$\{j\}$
(h,i)	$\{a,c\}$	$\{b,d\}$			$\{e,g\}$	$\{f\}$		$\{h,i\}$		$\{j\}$
(a,b)	$\{a,b,c,d\}$				$\{e,g\}$	$\{f\}$		$\{h,i\}$		$\{j\}$
(e,f)	$\{a,b,c,d\}$				$\{e,f,g\}$			$\{h,i\}$		$\{j\}$
(b,c)	$\{a,b,c,d\}$				$\{e,f,g\}$			$\{h,i\}$		$\{j\}$

The procedure CONNECTED-COMPONENTS initially places each vertex v in its own set. Then, for each edge (u,v), it unites the sets containing u and v. After all the edges are processed, two vertices are in the same connected component if and only if the corresponding objects are in the same set. Thus, CONNECTED-COMPONENTS computes sets in such a way that the procedure SAME-COMPONENT can determine whether two vertices are in the same connected component.

20.5 Linked-list Representation of Disjoint Sets

A simple way to implement a disjoint-set data structure is to represent each set by a linked list. The first object in each linked list serves as its set's representative. Each object in the linked list

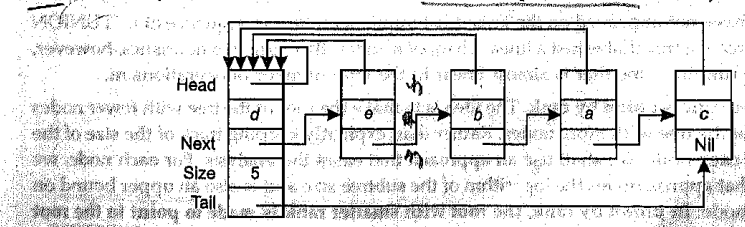


Figure 20.2

contains a set member, a pointer to the object containing the next set member, and a pointer back to the representative. Within each linked list, the objects may appear in any order (subject to our assumption that the first object in each list is the representative).

With this linked-list representation, both MAKE-SET and FIND-SET are easy, requiring $O(1)$ time. To carry out MAKE-SET(x), we create a new linked list whose only object is x. For FIND-SET (x), we just return the pointer from x back to the representative.

20.6 Disjoint-Set Forests

In a faster implementation of disjoint sets, we represent sets by rooted trees, with each node containing one member and each tree representing one set. In a ***disjoint-set forest***, each member points only to its parent. The root of each tree contains the representative and is its own parent.

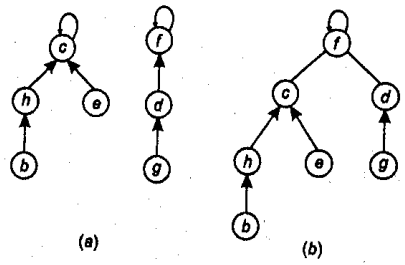


Figure 20.3 A disjoint-set forest. (*a*) Two trees representing the two sets of above Fig. The tree on the left represents the set $\{b,c,e,h\}$, with c as the representative, and the tree on the right represents the set $\{d,f,g\}$, with f as the representative. (*b*) The result of UNION(e,g).

We perform the three disjoint-set operations as follows. A MAKE-SET operation simply creates a tree with just one node. We perform a FIND-SET operation by chasing parent pointers until we find the root of the tree. The nodes visited on this path toward the root constitute the ***find path***. A UNION operation causes the root of one tree to point to the root of the other.

Heuristics to improve the running time

So far, we have not improved on the linked-list implementation. A sequence of $n-1$ UNION operations may create a tree that is just a linear chain of n nodes. By using two heuristics, however, we can achieve a running time that is almost linear in the total number of operations m.

The first heuristic is **union by rank**. The idea is to make the root of the tree with fewer nodes point to the root of the tree with more nodes. Rather than explicitly keeping track of the size of the subtree rooted at each node, we shall use an approach that eases the analysis. For each node, we maintain a ***rank*** that approximates the logarithm of the subtree size and is also an upper bound on the height of the node. **In union by rank, the root with smaller rank is made to point to the root with larger rank during a UNION operation.**

(log n)

The second heuristic, ***path compression***, is also quite simple and very effective. We use it during FIND-SET operations to make each node on the find path point directly to the root. Path compression does not change any ranks.

To implement a disjoint-set forest with the union-by-rank heuristic, we must keep track of ranks. **With each node x, we maintain the integer value *rank* [x], which is an upper bound on the height of *x* (the number of edges in the longest path between *x* and a descendant leaf).** When a singleton set is created by MAKE-SET, the initial rank of the single node in the corresponding tree is 0. Each FIND-SET operation leaves all ranks unchanged. When applying UNION to two trees, we make the root of higher rank the parent of the root of lower rank. In case of a tie, we arbitrarily choose one of the roots as the parent and increment its rank. We assign the parent of node x by $p[x]$. The LINK procedure, a subroutine called by UNION, takes pointers to two roots as inputs.

```
MAKE-SET(x)
  1.  p[x] ← x
  2.  rank[x] ← 0

UNION(x, y)
  1.  LINK(FIND-SET(x), FIND-SET(y))

LINK(x, y)
  1.  if rank[x] > rank[y]
  2.      then p[y] ← x
  3.      else p[x] ← y
  4.          if rank[x] = rank[y]
  5.              then rank[y] ← rank[y] + 1
```

The FIND-SET procedure with path compression is as follows :

```
FIND-SET(x)
  1.  if x ≠ p[x]
  2.      then p[x] ← FIND-SET(p[x])
  3.  return p[x]
```

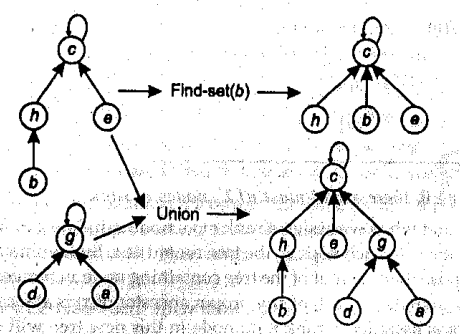


Figure 20.4

20.7 Properties of Ranks

Lemma 1

For all nodes x, we have $\text{rank}[x] \le \text{rank}[p[x]]$, if $x \ne p[x]$. The value of $\text{rank}[x]$ is initially 0 and increases through time until $x \ne p[x]$; from then on, $\text{rank}[x]$ does not change. The value of $\text{rank}[p[x]]$ is a monotonically increasing function of time.

We define $\text{size}(x)$ to be the number of nodes in the tree rooted at node x, including node x itself.

Lemma 2

For all tree roots x, $\text{size}(x) \ge 2^{\text{rank}[x]}$.

Proof. The proof is by induction on the number of LINK operations. Note that FIND-SET operations change neither the rank of a tree root nor the size of its tree.

Basis : The lemma is true before the first LINK, since ranks are initially 0 and each tree contains at least one node.

Inductive step : Assume that the lemma holds before performing the operation LINK(x, y). Let *rank* denote the rank just before the LINK, and let rank′ denote the rank just after the LINK. Define size and size′ similarly.

If $rank[x] \ne rank[y]$, assume without loss of generality that $rank[x] < rank[y]$. Node y is the root of the tree formed by the LINK operation, and

$$\begin{aligned}\text{size}'(y) &= \text{size}(x) + \text{size}(y)\\ &\ge 2^{\text{rank}[x]} + 2^{\text{rank}[y]}\\ &\ge 2^{\text{rank}[y]}\\ &= 2^{\text{rank}'[y]}\end{aligned}$$

No ranks or sizes change for any nodes other than y.

If $rank[x] = rank[y]$, node y is again the root of the new tree, and

$$\begin{aligned}\text{size}'(y) &= \text{size}(x) + \text{size}(y)\\ &\ge 2^{rank[x]} + 2^{rank[y]}\\ &= 2^{rank[y]+1}\\ &= 2^{rank'[y]}\end{aligned}$$

Lemma 3

For any integer $r \ge 0$, there are at most $n/2^r$ nodes of rank r.

Proof. Suppose that when we assign a rank r to a node x (in line 2 of MAKE-SET or in line 5 of LINK), we attach a label x to each node in the tree rooted at x. By Lemma 2, at least 2^r nodes are labeled each time. Suppose that the root of the tree containing node x changes. Lemma 1 assures us that the rank of the new root (or, in fact, of any proper ancestor of x) is at least $r+1$. Since we assign labels only when a root is assigned a rank r, no node in this new tree will ever again be labeled.

Thus, each node is labeled at most once, when its root is first assigned rank r. Since there are n nodes, there are at most n labeled nodes, with at least 2^r labels assigned for each node of rank r. If there were more than $n/2^r$ nodes of rank r, then more than $2^r \cdot (n/2^r) = n$ nodes would be labeled by a node of rank r, which is a contradiction. Therefore, at most $n/2^r$ nodes are ever assigned rank r.

Theorem

Every node has rank at most $\lfloor \lg n \rfloor$.

Proof. If we let $r > \lg n$, then there are at most $n/2^r < 1$ nodes of rank r. Since ranks are natural numbers, thus every node has rank at most $\lfloor \lg n \rfloor$ follows.

Example. *Give a non-recursive implementation of FIND-SET with path compression.*

Solution.

```
FIND-SET(x)
  y ← p[x]
  p[x] ← x
  while x ≠ y do
      z ← p[y]
      p[y] ← x
      x ← y
      y ← z
  end while
  root ← x
  y ← p[x]
  p[x] ← x
  while x ≠ y do
      z ← p[y]
      p[y] ← root
      x ← y
      y ← z
  end while
  return root
```

The first while loop traverses the path from x to the root of the tree while reversing all parent pointers along the way. The second while loop returns along this path and sets all parent pointers along the way to point to the root.

Example. *Give a sequence of m MAKE-SET, UNION and FIND-SET operations, n of which are MAKE-SET operations, that takes $\Omega(m \lg n)$ time when we use union by rank only.*

Solution. First perform the n (assume n is a power of 2) MAKE-SET operations on each of the elements $\{x_1, x_2, x_3, \ldots x_n\}$. Then perform a union on each pair (x_1, x_2), (x_3, x_4) and so on, yielding $n/2$ new sets. Continue the process until there is only a single set left. This uses a total of $n-1$ operations and produces a tree with depth $\Omega(\lg n)$.

Perform additional $k \ge n$ FIND-SET operations on the node of greatest depth.

Setting $m = n + n - 1 + k$ gives the desired running time of $\Omega(n + m \lg n) = \Omega(m \lg n)$.

Example. *Show that the running time of the disjoint-set forest with union by rank only is $O(m \lg n)$.*

Solution. We know that the rank of a node is at most the height of the subtree rooted at that node and every node has rank at most $\lfloor \lg n \rfloor$ and therefore the height h of any subtree is at most $\lfloor \lg n \rfloor$. Clearly, each call to FIND-SET (and thus UNION) takes at most $O(h) = O(\lg n)$ time.

So the resulting implementation runs in time $O(m \lg n)$.

Exercise

1. Show that after all edges are processed by CONNECTED-COMPONENTS, two vertices are in the same connected components if and only if they are in the same set.
2. How many times FIND-SET and UNION called during the execution of CONNECTED-COMPONENTS on an undirected graph $G = (V, E)$ with k connected components.
3. For each node x, how many bits are necessary to store size(x) ? How about rank[x] ?
4. Prove that the linked-list representation of disjoint sets and the weight-union heuristic, a sequence of m MAKE-SET, UNION, and FIND-SET operations, n of which are MAKE-SET operations, take $O(m + n \lg n)$ time.

CHAPTER

21

Dynamic Programming

21.1 Introduction

Dynamic programming is a problem solving technique that, like Divide and Conquer, solves problems by dividing them into subproblems. Dynamic programming is used when the subproblems are not independent, *e.g.* when they share the same subproblems. In this case, divide and conquer may do more work than necessary, because it solves the same subproblem multiple times.

Dynamic Programming solves each subproblem once and stores the result in a table so that it can be rapidly retrieved if needed again. It is often used in Optimization Problems : A problem with many possible solutions for which we want to find an optimal (the best) solution. (There may be more than 1 optimal solution). Dynamic Programming is an approach developed to solve sequential, or multi-stage, decision problems ; hence, the name "dynamic" programming. But, as we shall see, this approach is equally applicable for decision problems where sequential property is induced solely for computational convenience.

Dynamic programming can be thought of as being the reverse of recursion. Recursion is a top-down mechanism – we take a problem, split it up, and solve the smaller problems that are created. Dynamic programming is a **bottom-up mechanism** – we solve all possible small problems and then combine them to obtain solutions for bigger problems.

Dynamic programming works when a problem has the following characteristics :

- **Optimal Substructure :** If an optimal solution contains optimal sub solutions, then a problem exhibits *optimal substructure.*
- **Overlapping subproblems :** When a recursive algorithm would visit the same subproblems repeatedly, then a problem has *overlapping subproblems.*

If a problem has *optimal substructure,* then we can recursively define an optimal solution. If a problem has *overlapping subproblems,* then we can improve on a recursive implementation by computing each subproblem only once.

If a problem doesn't have *optimal substructure,* there is no basis for defining a recursive algorithm to find the optimal solutions. If a problem doesn't have *overlapping subproblems,* we really don't have anything to gain by using dynamic programming.

If the space of subproblems is small enough (*i.e.* polynomial in the size of the input), dynamic programming can be much more efficient than recursion.

Applications

- Knapsack problem
- Mathematical optimization problems
- Shortest path problems
- Matrix chain multiplication
- Longest common sequence (LCS)
- Control (Cruise control, Robotics, Thermostates)
- Flight control (Balance factors that oppose one another, *e.g.,* maximize accuracy, minimize time).
- Time Sharing : Schedule user and jobs to maximize CPU usage
- Other types of scheduling.

Development of Dynamic Programming Algorithm : It can be broken into *four* steps :

1. Characterize the structure of an optimal solution
2. Recursively define the value of the optimal solution. Like divide and conquer, divide the problem into 2 or more optimal parts recursively. This helps to define what the solution will look like.
3. Compute the value of the optimal solution from the bottom up (starting with the smallest subproblem).
4. Construct the optimal solution for the entire problem from the computed values of smaller subproblems.

21.2 Common Characteristics

There are a number of characteristics that are to all dynamic programming problems. These are :

1. The problem can be divided into stages with a decision required at each stage. *For example,* in the shortest path problem, they were defined by the structure of the graph. The decision was where to go next.
2. Each stage has a number of states associated with it. The states for the shortest path problem was the node reached.
3. The decision at one stage transforms one state into a state in the next stage. The decision of where to go next defined where we arrived in the next stage.
4. Given the current state, the optimal decision for each of the remaining states does not depend on the previous states or decisions. In the shortest path problem, it was not necessary to know how we got to a node, only that we did.
5. There exists a recursive relationship that identifies the optimal decision for stage j, given that stage $j+1$ has already been solved.
6. The final stage must be solvable by itself.

The last *two* properties are tied up in the recursive relationships given above.

The big skill in dynamic programming, and the art involved, is to take a problem and determine stages and states so that all of the above hold. If we can, then the recursive relationship makes finding the values relatively easy. Because of the difficulty in identifying stages and states, we will do a fair number of examples.

21.3 Matrix Multiplication

To multiply two matrices, multiply the rows in the first matrix by the columns of the second matrix. In general :

If $A = \lfloor a_{ij} \rfloor$ is a $p \times q$ matrix, $B = \lfloor b_{ij} \rfloor$ is a $q \times r$ matrix and $C = \lfloor c_{ij} \rfloor$ is a $p \times r$ matrix

then $$AB = C \quad \text{if} \quad c_{ij} = \sum_{k=1}^{q} a_{ik} b_{kj}$$

Given following matrices $\{A_1, A_2, A_3, \ldots, A_n\}$ and we have to perform the matrix multiplication, which can be accomplished by a series of matrix multiplications

$$A_1 \times A_2 \times A_3 \times \ldots \times A_n$$

Matrix multiplication operation is **associative** in nature rather commutative. By this, we mean that we have to follow the above matrix order for multiplication but we are free to **parenthesize** the above multiplication depending upon our need.

In general, for $1 \le i \le p$ and $1 \le j \le r$

$$C[i,j] = \sum_{k=1}^{q} A[i,k]\, B[k,j]$$

It can be observed that the total entries in matrix 'C' is 'pr' as the matrix is of dimension $p \times r$ Also each entry takes $O(q)$ times to compute, thus the total time to compute all possible entries for the matrix 'C' which is the multiplication of 'A' and 'B' is proportional to the product of the dimensions $p\ q\ r$.

It is also to be noticed that we can save the number of operations by reordering the parenthesis.

Example : Consider three matrices of

$$A_{5\times4},\quad B_{4\times3},\quad C_{3\times5}$$

then the possible multiplication

$$[(AB)C] = (5\times4\times3)+(5\times3\times5) = 135$$

$$[A(BC)] = (5\times4\times5)+(4\times3\times5) = 160$$

So, we have seen that by changing the evaluation sequence cost of operations also changes.

Number of ways for parenthesizing the matrices

If we have two or three matrices to which we have to compute the multiplication then the number of possible ways for parenthesizing these matrices is not more than 3-4. In general if we have 'n' matrices then there exist $(n-1)$ places where we can break the sequence of matrices with the outermost pair of parentheses namely just after the 1st matrix, just after the 2nd matrix etc. and just after the $(n-1)$th matrix. It can be observed that after splitting the kth matrix, we are left with two parenthesized sequence of matrices : one consists 'k' matrices and other consists '$n-k$' matrices. It can be seen that for parenthesizing the choicing are independent, thus if there are X ways for parenthesizing the left sequence and Y ways for parenthesizing the right sequence. The total will be $X \cdot Y$.

$$p(n) = \begin{cases} 1 & n=1 \\ \sum_{k=1}^{n-1} p(k)\, p(n-k) & n=2 \end{cases}$$

also $p(n) = c(n-1)$ where $c(n)$ is the nth **Catalon number**

$$c(n) = \frac{1}{n+1}\binom{2n}{n}$$

on applying Stirling's formula we have

$$c(n) \in \Omega\left(\frac{4^n}{n^{1.5}}\right),$$

which shows that 4^n grows faster, as it is an exponential function, then $n^{1.5}$.

Dynamic programming approach

Let $A_{i..j}$ indicates the multiplication of matrices through i to j. It can be seen that the dimension of $A_{i..j}$ is $P_{i-1} \times P_j$. Our task is to break the problem into several simpler and smaller subproblems of similar structure. We consider the highest level of parenthesizing where two matrix multiplications are considered. If we consider 'k' where $1 \le k \le n-1$, then

$$A_{1..n} = A_{1..k} \times A_{k+1..n}$$

Thus we are left with *two* questions :

- How to split the sequence of matrices ?
- How to parenthesize the subsequence $A_{1..k}$ and $A_{k+1..n}$?

One possible answer to the first question for finding the best value of 'k' is to check all possible choices of 'k' and consider the best among them. But it can be observed that checking all possibilities will lead to an exponential number of total possibilities. It can also be noticed that there exists only $O(n^2)$ different sequence of matrices, in this way we do not reach to the exponential growth.

The structure of an optimal parenthesization

Let us adopt the notation $A_{i..j}$ for the matrix that results from evaluating the product $A_i\, A_{i+1} \ldots A_j$. An optimal parenthesization of the product $A_1\, A_2 \ldots A_n$. Splits the product between A_k and A_{k+1} for some integer k in the range $1 \le k \le n$ *i.e.* for some value of k, we first compute the matrices $A_{1..k}$ and $A_{k+1..n}$ and then multiply them together to produce the final product $A_{1..n}$. The cost of this optimal parenthesization is thus the cost of computing the matrix $A_{1..k}$ + the cost of computing $A_{k+1..n}$ + cost of multiplying them together.

Let $m[i,j]$ be the minimum number of scalar multiplications needed to compute the matrix $A_{i..j}$, the cost of a cheapest way to compute $A_{1..n}$ would thus be $m[1..n]$.

We can define $m[i,j]$ recursively as follows :

If $i=j$ the chain consists of just one matrix $A_{i..i} = A_i$ so no scalar multiplications are necessary to compute the product . Thus $m[i,j]=0$ for $i=1,2,3,\ldots n$.

To compute $m[i,j]$, when $i<j$. Let us assume that the optimal parenthesization splits the product $A_i\, A_{i+1} \ldots A_j$ between A_k and A_{k+1} where $i \le k \le j$. Then $m[i,j]$ is equal to the minimum cost for computing the subproducts $A_{i..k}$ and $A_{k+1..j}$ + cost of multiplying them together. Since computing the matrix product $A_{i..k}$ and $A_{k+1..j}$ takes $p_{i-1}\, p_k\, p_j$ scalar multiplications, we obtain

$$m[i,j] = m[i,k] + m[k+1,j] + p_{i-1} p_k p_j$$

There are only $(j-1)$ possible values for 'k' namely $k = i, i+1, \ldots j-1$. Since the optimal parenthesization must use one of these values for 'k', we need only check them all to find the best . So the minimum cost of parenthesizing the product $A_i\, A_{i+1} \ldots A_j$ becomes

$$m[i,j] = \begin{cases} 0 & \text{if } i=j \\ \min_{i \le k < j} \{m[i,k] + m[k+1,j] + p_{i-1} p_k p_j\} & \text{if } i<j \end{cases}$$

To construct an optimal solution, let us define $s[i,j]$ to be the value of 'k' at which we can split the product $A_i\, A_{i+1} \ldots A_j$ to obtain an optimal parenthesization *i.e.* $s[i,j]=k$ such that

$$m[i,j]=m[i,k]+m[k+1,\ j]+p_{i-1}\ p_k\ p_j$$

Computing optimal costs

The following pseudocode assumes that matrix A_i has dimensions $p_{i-1} \times p_i$ for $i=1,2,3,\ldots n$. The input is a sequence $\langle p_0\ p_1 \ldots p_n \rangle$ where length$[p]=n+1$. The procedure uses an auxiliary table $m[1..n,\ 1..n]$, for storing the $m[i,j]$ costs and an auxiliary table $s[1..n,\ 1..n]$ that records which index of k achieved the optimal cost in computing $m[i,j]$.

The algorithm first computes $m[i,j] \leftarrow 0$ for $i=1,2,3..n$, the minimum costs for chains of length 1.

```
MATRIX-CHAIN-ORDER(p)
n ← length[p] − 1
for i ← 1 to n
    do m[i,i] ← 0
for l ← 2 to n          ▷ l is the chain length
    do for i ← 1 to n − l + 1
        do j ← i + l − 1
            m[i, j] ← ∞
            for k ← i to j − 1
                do q ← m[i,k] + m[k + 1, j] + p_{i−1} p_k p_j
                    if q < m[i, j]
                        then m[i, j] ← q
                            s[i, j] ← k
return m and s
```

It then uses recurrence to compute $m[i,i+1]$ for $i=1,2,\ldots n-1$ (the minimum cost of length 2) during the first execution of the loop and in the second time, it computes $m[i,i+2]\ i=1,2,\ldots n-2$ (the minimum cost of length 3) and so on.

Since we have defined $m[i,j]$ only for $i<j$, only the portion of the table m strictly above the main diagonal is used. MATRIX-CHAIN-ORDER computes the rows from bottom to top and from left to right within each row. An entry $m[i,j]$ is computed using the products $p_{i-1}\ p_k\ p_j$ for $k=i,\ i+1,\ldots j-1$.

A simple inspection of the nested loop structure of MATRIX-CHAIN-ORDER yields a running time of $O(n^3)$ for the algorithm.

Now we want to know the parenthesization that gives this minimum. To find it, we use the array, $s.s[i,j]$ records k for the optimal split in each sub array : $A_{i..j}$ becomes $(A_{i..k})$ and $(A_{k+1}..j)$. To split $A_{1..n}$, use $s[1,n]$ value for k : $A_{1..n} = A_{1..s[1,n]}\ A_{s[1,n]+1..n}$. We recursively split each of the sub sequences using the array s to find the optimal k at which to split each one.

The following recursive procedure computes the matrix-chain product $A_{i..j}$ given the matrices $A=\langle A_1\ A_2 \ldots A_n \rangle$, the s table computed by MATRIX-CHAIN-ORDER, and the indices i and j. The initial call is MATRIX-CHAIN-MULTIPLY $(A,s,1,n)$.

```
MATRIX-CHAIN-MULTIPLY (A, s, i, j)
if j > i
then X ← MATRIX-CHAIN-MULTIPLY (A, s, i, s[i, j])
     Y ← MATRIX-CHAIN-MULTIPLY (A, s, s[i, j] + 1, j)
     return MATRIX-MULTIPLY (X, Y)
else return A_i
```

Example of Finding the Multiplication Sequence

Consider $n=6$. Assume that the array $s[1..6,1..6]$ has been computed. The multiplication sequence is recovered as follows.

$s[1,6]=3 \quad (A_1\ A_2\ A_3)(A_4\ A_5\ A_6)$

$s[1,3]=1 \quad (A_1(A_2\ A_3))$

$s[4,6]=5 \quad ((A_4\ A_5)A_6)$

Hence the final multiplication sequence is

$$(A_1\,(A_2 A_3))((A_4 A_5)\,A_6)$$

Example. *We are given the sequence {4, 10, 3, 12, 20, 7}. The matrices have sizes 4×10, 10×3, 3×12, 12×20, 20×7. We need to compute $M[i,j]$, $0 \le i, j \le 5$. We know $M[i,i]=0$ for all i.*

1	2	3	4	5	
0					1
	0				2
		0			3
			0		4
				0	5

We proceed, working away from the diagonal. We compute the optimal solutions for products of 2 matrices.

1	2	3	4	5	
0	120				1
	0	360			2
		0	720		3
			0	1680	4
				0	5

{4, 10, 3, 12, 20, 7}

Now products of 3 matrices

$$M[1,3]=\min\begin{cases} M[1,2]+M[3,3]+p_0p_2p_3=120+0+4\cdot3\cdot12=264 \\ M[1,1]+M[2,3]+p_0p_1p_3=0+360+4\cdot10\cdot12=840 \end{cases} = 264$$

$$M[2,4]=\min\begin{cases}M[2,3]+M[4,4]+p_1p_3p_4=360+0+10\cdot12\cdot20=2760\\ M[2,2]+M[3,4]+p_1p_2p_4=0+720+10\cdot3\cdot20=1320\end{cases}=\mathbf{1320}$$

$$M[3,5]=\min\begin{cases}M[3,4]+M[5,5]+p_2p_4p_5=720+0+3\cdot20\cdot7=1140\\ M[3,3]+M[4,5]+p_2p_3p_5=0+1680+3\cdot12\cdot7=1932\end{cases}=\mathbf{1140}$$

k=3
k=4

1	2	3	4	5	
0	120				1
	0	360			2
		0	720		3
			0	1680	4
				0	5

⇒

1	2	3	4	5	
0	120	264			1
	0	360	1320		2
		0	720	1140	3
			0	1680	4
				0	5

Now products of 4 matrices

$$M[1,4]=\min\begin{cases}M[1,3]+M[4,4]+p_0p_3p_4=264+0+4\cdot12\cdot20=1224\\ M[1,2]+M[3,4]+p_0p_2p_4=120+720+4\cdot3\cdot20=1080\\ M[1,1]+M[2,4]+p_0p_1p_4=0+1320+4\cdot10\cdot20=2120\end{cases}=\mathbf{1080}$$

k=1
k=2
k=3

$$M[2,5]=\min\begin{cases}M[2,4]+M[5,5]+p_1p_4p_5=1320+0+10\cdot20\cdot7=2720\\ M[2,3]+M[4,5]+p_1p_3p_5=360+1680+10\cdot12\cdot7=2880\\ M[2,2]+M[3,5]+p_1p_2p_5=0+1140+10\cdot3\cdot7=1350\end{cases}=\mathbf{1350}$$

1	2	3	4	5	
0	120	264			1
	0	360	1320		2
		0	720	1140	3
			0	1680	4
				0	5

⇒

1	2	3	4	5	
0	120	264	1080		1
	0	360	1320	1350	2
		0	720	1140	3
			0	1680	4
				0	5

Now products of 5 matrices

$$M[1,5]=\min\begin{cases}M[1,4]+M[5,5]+p_0p_4p_5=1080+0+4\cdot20\cdot7=1544\\ M[1,3]+M[4,5]+p_0p_3p_5=264+1680+4\cdot12\cdot7=2016\\ M[1,2]+M[3,5]+p_0p_2p_5=120+1140+4\cdot3\cdot7=1344\\ M[1,1]+M[2,5]+p_0p_1p_5=0+1350+4\cdot10\cdot7=1630\end{cases}=\mathbf{1344}$$

k=1
k=2
k=3
k=4

1	2	3	4	5	
0	120	264	1080		1
	0	360	1320	1350	2
		0	720	1140	3
			0	1680	4
				0	5

⇒

1	2	3	4	5	
0	120	264	1080	1344	1
	0	360	1320	1350	2
		0	720	1140	3
			0	1680	4
				0	5

To print the optimal parenthesization, we use the PRINT-OPTIMAL-PARENS procedure.

PRINT-OPTIMAL-PARENS (s, i, j)

```
if i = j
    then print "A"
    else print "("
        PRINT-OPTIMAL-PARENS (s, i, s[i, j])
        PRINT-OPTIMAL-PARENS (s, s[i, j] + 1, j)
        print ")"
```

Now for optimal parenthesization, Each time we find the optimal value for M[i,j] we also store the value of k that we used. If we did this for the example, we would get

1	2	3	4	5	
0	120/1	264/2	1080/2	1344/2	1
	0	360/2	1320/2	1350/2	2
		0	720/3	1140/4	3
			0	1680/4	4
				0	5

The k value for the solution is 2, so we have $((A_1\ A_2)(A_3\ A_4\ A_5))$. The first half is done. The optimal solution for the second half comes from entry $M[3,5]$.The value of k here is 4, so now we have $((A_1\ A_2)((A_3\ A_4)\ A_5))$.Thus the optimal solution is to parenthesize $((A_1\ A_2)((A_3\ A_4)\ A_5))$.

21.4 Memoization

So far we have talked about implementing dynamic programming in a bottom-up fashion.

Dynamic programming can also be implemented using **memoization**. With memoization, we implement the algorithm recursively, but we keep track of all of the sub solutions. If we encounter a subproblem that we have seen, we look up the solution. If we encounter a subproblem that we have not seen, we compute it, and add it to the list of sub solutions we have seen. Each subsequent time that the subproblem is encountered, the value stored in the table is simply looked up and returned.

MEMOIZED-MATRIX-CHAIN (p)

```
n ← length[p] − 1
for i ← 1 to n do
    for j ← 1 to n do
        m[i, j] ← infinity (∞)
return LOOKUP-CHAIN(p, 1, n)
```

LOOKUP-CHAIN (p, i, j)

```
if m[i, j] < infinity then
    return m[i, j]
if i = j then
    m[i, j] ← 0
else for k ← i to j − 1 do
    q ← LOOKUP-CHAIN(p, i, k) + LOOKUP-CHAIN (p, k + 1, j) + p_{i−1}p_k p_j
    if q < m[i, j] then
        m[i, j] ← q
return m[i, j]
```

Memoization offers the efficiency of dynamic programming. It maintains the **top-down recursive strategy.**

Comparison with dynamic programming

Dynamic programming algorithm usually outperforms a top-down memoized algorithm by constant factor, because there is no over-head for recursion and fewer overheads for maintaining the table. In situations where not every subproblem is computed, memoization only solves those that are needed but dynamic programming solves all the subproblems.

In summary, the matrix-chain multiplication problem can be solved in $O(n^3)$ time by either a top-down, memorized algorithm or a bottom-up dynamic-programming algorithm.

21.5 Longest common subsequence (LCS)

A subsequence of a given sequence is just the given sequence with some elements left out. Given two sequences X and Y, we say that a sequence Z is a common sequence of X and Y if Z is a subsequence of both X and Y.

In the longest common subsequence problem, we are given two sequences $X = \langle x_1\, x_2 \ldots x_m \rangle$ and $Y = \langle y_1\, y_2 \ldots y_n \rangle$ and wish to find a maximum length common subsequence of X and Y. LCS problem can be solved using dynamic programming.

Characterizing a longest common subsequence

A brute-force approach to solving the LCS problem is to specify all subsequences of X and check each subsequence to see if it is also a subsequence of Y, keeping track of the longest subsequence found. Each subsequence of X corresponds to a subset of the indices $\{1,2,\ldots m\}$ of X, there are 2^m subsequences of X, so this approach requires exponential time.

The LCS problem has an optimal-substructure property. Given a sequence $X = \langle x_1\, x_2 \ldots x_m \rangle$, we define the ith prefix of X, for $i = 0,1,2\ldots m$ as $X_i = \langle x_1\, x_2 \ldots x_i \rangle$. For example, if $X = \langle A, B, C, B, C, A, B, C \rangle$ then $X_4 = \langle A, B, C, B \rangle$.

Theorem (Optimal substructure of an LCS)

Let $X = \langle x_1, x_2, \ldots, x_m \rangle$ and $Y = \langle y_1, y_2, \ldots, y_n \rangle$ be the sequences, and let $Z = \langle z_1, z_2, \ldots, z_k \rangle$ be any LCS of X and Y.

1. If $x_m = y_n$, then $z_k = x_m = y_n$ and Z_{k-1} is an LCS of X_{m-1} and Y_{n-1}
2. If $x_m \neq y_n$, then $z_k \neq x_m$ implies that Z is an LCS of X_{m-1} and Y.
3. If $x_m \neq y_n$, then $z_k \neq y_n$ implies that Z is an LCS of X and Y_{n-1}

The above theorem implies that there are either one or two subproblems to examine when finding an LCS of $X = \langle x_1, x_2, \ldots, x_m \rangle$ and $Y = \langle y_1, y_2, \ldots, y_n \rangle$. If $x_m = y_n$ we must find an LCS of X_{m-1} and Y_{n-1}. If $x_m \neq y_n$, then we must solve two subproblems finding as LCS of X_{m-1} and Y and finding an LCS of X and Y_{n-1}. Whenever of these LCS's is longer is an LCS of X and Y. But each of these subproblems has the subproblem of finding the LCS of X_{m-1} and Y_{n-1}.

Let us define $c[i, j]$ to be the length of an LCS of the sequence X_i and Y_j. If either $i = 0$ or $j = 0$, one of the sequences has length 0, so the LCS has length 0. The optimal substructure of the LCS problem gives the recurrence formula

$$c[i,j] = \begin{cases} 0 & \text{if } i=0 \text{ or } j=0 \\ c[i-1,j-1]+1 & \text{if } i,j>0 \text{ and } x_i = y_j \\ \max(c[i,j-1], c[i-1,j]) & \text{if } i,j>0 \text{ and } x_i \neq y_j \end{cases}$$

Now, we will develop an exponential recursive algorithm **LCS-LENGTH** to compute an LCS of two strings.

Procedure LCS-LENGTH takes two sequences X and Y as inputs. It stores the $c[i, j]$ values in a table $c[0\ldots m, 0\ldots n]$ whose entries are computed in row-major order, that is first row of c is filled from left to right, then the second row and so on. It also maintains the table $b[1..m, 1..n]$ to simplify construction of an optimal solution. The procedure returns the b and c tables. $c[m, n]$ contains the length of LCS of X and Y.

```
LCS-LENGTH (X, Y)
m ← lenght[X]
n ← length[Y]
for i ← 1 to m
    do c[i, 0] ← 0
for j ← 0 to m
    do c[0, j] ← 0
for i ← 1 to m
    do for j ← 1 to n
        do if x_i = y_j
            then c[i, j] ← c[i − 1, j − 1] + 1
                b[i, j] ← " ↖"
            else if c[i − 1, j] ≥ c[i, j − 1]
                then c[i, j] ← c[i − 1, j]
                    b[i, j] ← "↑"
                else c[i, j] ← c[i, j − 1]
                    b[i, j] ← "←"
return c and b.
```

Example. *Given two sequences $X[1..m]$ and $Y[1..n]$. Find the longest subsequence common to both. Note : not substring, subsequence.*

So if $x : A\ B\ C\ B\ D\ A\ B$

$y : B\ D\ C\ A\ B\ A$

the longest subsequence turns out to be $B\ C\ B\ A$. There could be other subsequences of the longest length, the algorithm will get one of them.

Here $X = \langle A, B, C, B, D, A, B \rangle$ and $Y = \langle B, D, C, A, B, A \rangle$

$m = \text{length}[X]$ and $n = \text{length}[Y]$

$m = 7$ and $n = 6$

Now, filling in the $m \times n$ table with the values of $c[i, j]$ and the appropriate arrow for the value of $b[i, j]$. Initialize top row and left column to 0 which takes $\theta(m+n)$ time.

Work across the rows starting at the top. Any time $x_i = y_j$ fill in the diagonal neighbour + 1 and mark the box with the wingding '↖' otherwise fill in the box with the max of the box above and the box to the left. That is, the entry of $c[i, j]$ depends only on whether $x_i = y_j$ and the values in entries $c[i-1, j]$, $c[i, j-1]$ and $c[i-1, j-1]$ which are computed before $c[i, j]$. The max length is the lower right hand corner. In $c[i-1, j]$ and $c[i, j-1]$ entries if $c[i-1, j] \geq c[i, j-1]$ then $b[i, j]$ entry is '↑' otherwise "←".

Then, to reconstruct the elements of an LCS, follow the $b[i, j]$ arrows from the lower right-hand corner. Each '↖' on the path corresponds to an entry for which $x_i = y_j$ is a member of LCS.

Here

$x_1 = x[1] = A$ $\quad y_1 = y[1] = B$

$x_2 = B$ $\quad y_2 = D$

$x_3 = C$ $\quad y_3 = C$

$x_4 = B$ $\quad y_4 = A$

$x_5 = D$ $\quad y_5 = B$

$x_6 = A$ $\quad y_6 = A$

$x_7 = B$

Now, fill the values of $c[i, j]$ in $m \times n$ table.

Initially, for $i = 1$ to 7 $c[i, 0] = 0$

for $j = 0$ to 6 $c[0, j] = 0$

That is

i	j	0	1	2	3	4	5	6
		y_j	B	D	C	A	B	A
0	x_i	0	0	0	0	0	0	0
1	A	0						
2	B	0						
3	C	0						
4	B	0						
5	D	0						
6	A	0						
7	B	0						

Now, for $i = 1$ and $j = 1$

x_1 and y_1 we get $x_1 \neq y_1$ *i.e.*, $A \neq B$

and $c[i-1, j] = c[0, 1] = 0$

$c[i, j-1] = c[1, 0] = 0$

That is, $c[i-1, j] = c[i, j-1]$ so $c[1,1] = 0$ and $b[1,1] = '\uparrow'$

Now, $i = 1$ and $j = 2$

x_1 and y_2 we get $x_1 \neq y_2$ *i.e.*, $A \neq D$.

$c[i-1, j] = c[0, 2] = 0$

$c[i, j-1] = c[1, 1] = 0$

That is, $c[i-1, j] = c[i, j-1]$ and $c[1,2] = 0$ $b[1,2] = '\uparrow'$

Now $i = 1$ and $j = 3$

x_1 and y_3 we get $A \neq C$ *i.e.*, $x_1 \neq y_3$.

$c[i-1, j] = c[0, 3] = 0$

$c[i, j-1] = c[1, 2] = 0$

So, $c[1,3] = 0$ $b[1,3] =$ "↑"

Now $i = 1$ and $j = 4$

x_1 and y_4 we get $x_1 = y_4$ *i.e.*, $A = A$

$c[1, 4] = c[1-1, 4-1] + 1$

$= c[0, 3] + 1$

$= 0 + 1 = 1$

$c[1, 4] = 1$

$b[1, 4] = '\nwarrow'$

Now $i = 1$ and $j = 5$

x_1 and y_5 we get $x_1 \neq y_5$

$c[i-1, j] = c[0, 5] = 0$

$c[i, j-1] = c[1, 4] = 1$

Thus, $c[i, j-1] > c[i-1, j]$ *i.e.*, $c[1,5] = c[i, j-1] = 1$. So, $b[1,5] = '\leftarrow'$

Now for $i = 1$ and $j = 6$

x_1 and y_6 we get $x_1 = y_6$

$c[1, 6] = c[1-1, 6-1] + 1$

$= c[0, 5] + 1 = 0 + 1 = 1$

$c[1, 6] = 1$

$b[1, 6] = '\nwarrow'$

i	j	0	1	2	3	4	5	6
		y_j	B	D	C	A	B	A
0	x_i	0	0	0	0	0	0	0
1	A	0	↑ 0	↑ 0	↑ 0	↖ 1	← 1	↖ 1
2	B	0						
3	C	0						
4	B	0						
5	D	0						
6	A	0						
7	B	0						

Now $i=2$ and $j=1$

we get x_2 and y_1 $B=B$ *i.e.*, $x_2=y_1$

$$c[2,1]=c[2-1,1-1]+1$$
$$=c[1,0]+1$$
$$=0+1=1$$

$c[2,1]=1$ and $b[2,1]=$ '↖'

Similarly, we fill the all values of $c[i,j]$ and we get

i	j	0	1	2	3	4	5	6
		y_j	(B)	D	(C)	A	(B)	(A)
0	x_i	0	0	0	0	0	0	0
1	A	0	↑ 0	↑ 0	↑ 0	↖ 1	←1	↖ 1
2	(B)	0	↖ (1)	←1	←1	↑ 1	↖ 2	←2
3	(C)	0	↑ 1	↑ 1	↖ (2)	←2	↑ 2	↑ 2
4	(B)	0	↖ 1	↑ 1	↑ 2	↑ 2	↖ (3)	←3
5	D	0	↑ 1	↖ 2	↑ 2	↑ 2	↑ 3	↑ 3
6	(A)	0	↑ 1	↑ 2	↑ 2	↖ 3	↑ 3	↖ (4)
7	B	0	↖ 1	↑ 2	↑ 2	↑ 3	↖ 4	↑ 4

Figure 21.1

The entry 4 in $c[7,6]$ is the length of the LCS $\langle B,C,B,A\rangle$ for X and Y.

To find, the elements of an LCS, follow the $b[i,j]$ arrows from the lower right-hand corner. In Fig. 21.1 the path is shaded.

Each '↖' on the path corresponds to an entry for which $x_i=y_j$ is a member of an LCS.

Constructing an LCS

The b table returned by LCS-LENGTH can be used to quickly construct an LCS of $X=\langle x_1\, x_2 \ldots x_m\rangle$ and $Y=\langle y_1\, y_2 \ldots y_n\rangle$. The following recursive procedure prints out an LCS of X and Y in proper, forward order.

```
PRINT-LCS (b, X, i, j)
if i = 0 or j = 0
    then return
if b[i, j] = "↖"
    then PRINT-LCS (b, X, i − 1, j − 1)
         print x_i
elseif b[i, j] = "↑"
    then PRINT-LCS (b, X, i − 1, j)
else PRINT-LCS (b, X, i, j − 1)
```

For the b table in the Fig 21.1, this procedure prints "$BCBA$". The procedure takes time $O(m+n)$, since atleast one of i and j is decremented in each stage of the recursion.

Example. *What are the basic four steps of dynamic programming ?*

Solution. The development of a DP algorithm consists of the following *four* steps :

1. Characterize the structure of an optimal solution
2. Define the value of an optimal solution recursively
3. Compute the value of an optimal solution in a bottom-up fashion
4. Construct an optimal solution from the computed information

Example. *Determine the LCS of* $\langle 1,0,0,1,0,1,0,1\rangle$ *and* $\langle 0,1,0,1,1,0,1,1,0\rangle$.

Solution. Let $X=\langle 1,0,0,1,0,1,0,1\rangle$ and $Y=\langle 0,1,0,1,1,0,1,1,0\rangle$. We know

$$c[i,j]=\begin{cases} 0 & \text{if } i=0 \text{ or } j=0 \\ c[i-1,j-1]+1 & \text{if } i,j>0 \text{ and } x_i=y_j \\ \max(c[i,j-1],c[i-1,j]) & \text{if } i,j>0 \text{ and } x_i\neq y_j \end{cases}$$

We are looking for $c[8,9]$. The following table is built.

$x = \langle 1,0,0,1,0,1,0,1 \rangle \qquad y = \langle 0,1,0,1,1,0,1,1,0 \rangle$

		0	1	2	3	4	5	6	7	8	9
		y_j	0	1	0	1	1	0	1	1	0
0	x_i	0	0	0	0	0	0	0	0	0	0
1	1	0	↑ 0	↖ 1	← 1	↖ 1	↖ 1	← 1	↖ 0	↖ 1	← 1
2	0	0	↖ 1	↑ 1	↖ 2	← 2	← 2	↖ 2	← 2	← 2	↖ 2
3	0	0	↖ 1	↑ 1	↖ 2	↑ 2	↑ 2	↖ 3	← 3	← 3	↖ 3
4	1	0	↑ 1	↖ 2	↑ 2	↖ 3	↖ 3	↑ 3	↖ 4	↖ 4	← 4
5	0	0	↖ 1	↑ 2	↖ 3	↑ 3	↑ 3	↖ 4	↑ 4	↑ 4	↖ 5
6	1	0	↑ 1	↖ 2	↑ 3	↖ 4	↖ 4	↑ 4	↖ 5	↖ 5	↑ 5
7	0	0	↖ 1	↑ 2	↖ 3	↑ 4	↑ 4	↖ 5	↑ 5	↑ 5	↖ 6
8	1	0	↑ 1	↖ 2	↑ 3	↖ 4	↖ 5	↑ 5	↖ 6	↖ 6	↑ 6

From the table, we can deduce that LCS = 6. There are several such sequences, for instance $\langle 1,0,0,1,1,0 \rangle$, $\langle 0,1,0,1,0,1 \rangle$ and $\langle 0,0,1,1,0,1 \rangle$.

Example. *Show how to reconstruct an LCS from the completed c table and the original sequence* $X = \langle x_1 x_2 \ldots x_m \rangle$ *and* $Y = \langle y_1 y_2 \ldots y_n \rangle$ *in* $O(m+n)$ *time, without using the b table.*

Solution. It is easy to see that it takes $O(m+n)$ time if the b table is used. We now eliminate the b table. Each $c[i,j]$ entry depends on only three other entries : $[i-1,j]$, $c[i,j-1]$ and $c[i-1,j-1]$. Given the value of $c[i,j]$, we can determine in $O(1)$ time which of these three values was used to compute $c[i;j]$, without inspecting table b. Thus, we can reconstruct LCS in $O(m+n)$ time using the c table, X, and Y.

```
PRINT-LCS(C, X, Y, i, j)
    if i = 0 or j = 0
        then return
    if x_i = y_j and c[i, j] = 1 + c[i − 1, j − 1]        ▸ ↖
        then PRINT-LCS(c, X, Y, i − 1, j − 1)
        print x_i
    else if x_i ≠ y_j and c[i, j] = c[i, j − 1]           ▸ ↑
        then PRINT-LCS(c, X, Y, i, j − 1)
    else if x_i ≠ y_j and c[i, j] = c[i − 1, j]           ▸ ←
        then PRINT-LCS(c, X, Y, i − 1, j)
    else
        error
```

Initial call is PRINT-LCS(c, X, Y, m, n)

Example. *Give an* $O(n^2)$ *time algorithms to find a longest decreasing subsequence of a sequence of n numbers.*

Solution. Let $a_1, a_2, a_3, \ldots, a_n$ be the n numbers. We follow the four steps of devising a dynamic programming algorithm in the following.

Step 1. *Characterize the structure of an optimal solution*

Let $X_i = a_1, a_2, \ldots, a_i$ the ith prefix of the sequence and $l(i)$ be the longest decreasing subsequence of X_i $1 \le i \le n$. Then, the problem is to find $\max\limits_{1 \le i \le n} \{l(i)\}$.

Step 2. *Define the value of an optimal solution recursively*

$$l(1) = 1$$

$$l(i) = \max_{1 \le j \le n} \{l(j) + 1 : a_j \ge a_i\}$$

Step 3. Compute l (index) $= \max\limits_{1 \le i \le n} \{l(i)\}$

Algorithm Longest-decreasing-sequence (a, n)

```
l[1] ← 1 ; S[1] ← 0
for i ← 2 to n do
    l_max ← 0, S[i] ← 0
    for j ← 1 to i − 1 do
        if (a[j] > a[i]) and (l[j] + 1 > l_max) then
            l_max ← l[j] + 1
            S[i] ← j
        l[i] ← l_max
        l_max ← 0 ; index ← 0
    for i ← 1 to n do
        if (l[i] > l_max) then
        l_max ← l[i]
        index ← i
```

Step 4. *Construct an optimal solution from the computed information.*

Algorithm Find-Sequence (S, index)

```
if S[index] = 0 then exist ;
call Find-Sequence (S, S[index]) ;
print a_index ;
```

The running time complexity of algorithm longest decreasing sequence is

$$\sum_{i=1}^{n} \sum_{j=1}^{i-1} c \le cn^2$$

Thus, its running time is $O(n^2)$, where c is a constant.

Example. *"Give an $O(n^2)$ time algorithm to find the longest monotonically increasing subsequence of a sequence of n numbers."*

Solution. Here is one suggestion :

1. Let $X = \langle x_1, \ldots, x_n \rangle$ be the sequence.
2. $Y \leftarrow$ MERGE-SORT (X)
3. $\langle c, b \rangle \leftarrow$ LCS-LENGTH (X, Y)
4. PRINT-LCS (b, X, n, n)

Example. *"Give a memoized version of LCS-LENGTH that runs in $O(mn)$ time."*

Solution.

```
MEMO-LCS-LENGTH (X,Y,i, j)
    for k ← 0 to length (X)
        for l ← 0 to length [Y]
            m[k,l] = ∞
    return MEMO-LCS-LENGTH' (X,Y,i, j)

MEMO-LCS-LENGTH' (X,Y,i, j)
    if m[i, j] ≠ ∞
        then return m[i, j]
    if i = 0 or j = 0
        then m[i, j] = 0
            return m[i, j]
    if x_i = y_j
        then m[i, j] = MEMO-LCS-LENGTH' [i − 1, j − 1] +1
            return m[i, j]
    m[i, j] = max (MEMO-LCS-LENGTH' (i, j − 1), MEMO-LCS-LENGTH' (i − 1, j))
    return m[i, j]
```

Exercise

1. Find an optimal parenthesization of a matrix-chain product whose sequence of dimensions is $\langle 5,10,3,12,5,50,6 \rangle$
2. Show that a full parenthesization of an n-element expression has exactly $n-1$ pairs of parentheses.
3. Determine an LCS of $\langle A,B,C,D,B,A,C,D,F \rangle$ and $\langle C,B,A,F \rangle$.
4. Show how to compute the length of an LCS using only 2 $\min(m,n)$ entries in the c table plus $O(1)$ additional space. Then, show how to do this using $\min(m,n)$ entries plus $O(1)$ additional space.
5. Find the longest common subsequence from the given two sequence of characters.
   (a) $P = \langle A,B,C,D,B,C,D,C,D,D \rangle$; $Q = \langle B,C,D,C,D \rangle$
   (b) $P = \langle 1,0,0,1,0,1,1,0,1,1,0,1 \rangle$; $Q = \langle 0,1,1,0 \rangle$
6. Compute the optimal sequence for the multiplication operation for the following matrices.
   (a) $P_{5\times3}\, Q_{3\times7}\, R_{7\times3}\, T_{3\times9}$ (b) $A_{5\times3}\, B_{3\times7}\, C_{7\times2}\, D_{2\times5}$

CHAPTER

22

Greedy Algorithms

22.1 Introduction

Greedy algorithms solve problems by making the choice that seems best at the particular moment. Many optimization problems can be solved using a greedy algorithm. Some problems have no efficient solution, but a greedy algorithm may provide a solution that is close to optimal. A greedy algorithm works if a problem exhibits the following *two* properties :

1. *Greedy choice property.* A globally optimal solution can be arrived at by making a locally optimal solution. In other words, an optimal solution can be obtained by making "greedy" choices.
2. *Optimal substructure.* Optimal solutions contain optimal sub solutions. In other words, solutions to sub problems of an optimal solution are optimal.

22.2 An Activity-Selection Problem

Our first example is the problem of scheduling a resource among several competing activities. We shall find that a greedy algorithm provides a well-designed and simple method for selecting a maximum-size set of mutually compatible activities.

Suppose $S = \{1, 2, \ldots, n\}$ is the set of n proposed activities. The activities share a resource, which can be used by only one activity at a time *e.g.*, a Tennis Court, a Lecture Hall etc . Each activity i has a **start time** s_i and a **finish time** f_i, where $s_i \le f_i$. If selected, activity i takes place during the half-open time interval $[s_i, f_i)$. Activities i and j are **compatible** if the intervals $[s_i, f_i)$ and $[s_j, f_j)$ do not overlap (*i.e.*, i and j are compatible if $s_i \ge f_j$ or $s_j \ge f_i$). The **activity-selection problem** selects the maximum-size set of mutually compatible activities.

In this strategy we first select the activity with minimum duration $(f_i - s_i)$ and schedule it. Then, we skip all activities that are not compatible to this one, which means we have to select compatible activity having minimum duration and then we have to schedule it. This process is repeated until all the activities are considered. It can be observed that the process of selecting the activity becomes faster if we assume that the input activities are in order by **increasing finishing time** : $f_1 \le f_2 \le f_3 \le \ldots \le f_n$

The running time of algorithm GREEDY-ACTIVITY-SELECTOR is $\theta(n \log n)$, as sorting can be done in $O(n \log n)$. There are $O(1)$ operations per activity, thus total time is

$$O(n \log n) + n \cdot O(1) = O(n \log n).$$

The pseudo code of GREEDY-ACTIVITY-SELECTOR is as follows :

```
GREEDY-ACTIVITY-SELECTOR (s, f)
n ← length[s]
A ← {1}
j ← 1
for i ← 2 to n
        do if s_i ≥ f_j
                then A ← A ∪ {i}
                     j ← i
return A
```

Theorem

The GREEDY-ACTIVITY-SELECTOR algorithm gives an optimal solution to the activity selection problem.

Proof. Let $S = \{1, 2, \ldots, n\}$ be the set of activities to schedule. Since we are assuming that the activities are in order by finish time, activity 1 has the earliest finish time. We wish to show that, there is an optimal solution that begins with a greedy choice (with activity 1, which as the earliest finish time).

Suppose that $A \subseteq S$ is an optimal solution to the given instance of the activity-selection problem, and let us order the activities in A by finish time. Suppose further that the first activity in A is activity k.

- If $k = 1$, then schedule A begins with a greedy choice.
- If $k \ne 1$, we want to show that there is another optimal solution B to S that begins with the greedy choice, activity 1.

Let $B = A - \{k\} \cup \{1\}$. Because $f_1 \le f_k$, the activities in B are disjoint, and since B has the same number of activities as A, it is also optimal. Thus, B is an optimal solution for S that contains the greedy choice of activity 1.

Once the greedy choice of activity 1 is made, the problem reduces to finding an optimal solution for the activity-selection problem over those activities in S that are compatible with activity 1. That is, if A is an optimal solution to the original problem S, then $A' = A - \{1\}$ is an optimal solution to the activity-selection problem $S' = \{i \in S : S_i \le f_1\}$.

Why ? If we could find a solution B' to S' with more activities than A', *adding activity* 1 *to* B' would yield a solution B to S with more activities than A, thereby contradicting the optimality of A. Therefore, after each greedy choice is made, we are left with an optimization problem of the same form as the original problem. By induction on the number of choices made, making the greedy choice at every step produces an optimal solution.

Example. *Given 10 activities along with their start and finish time as*

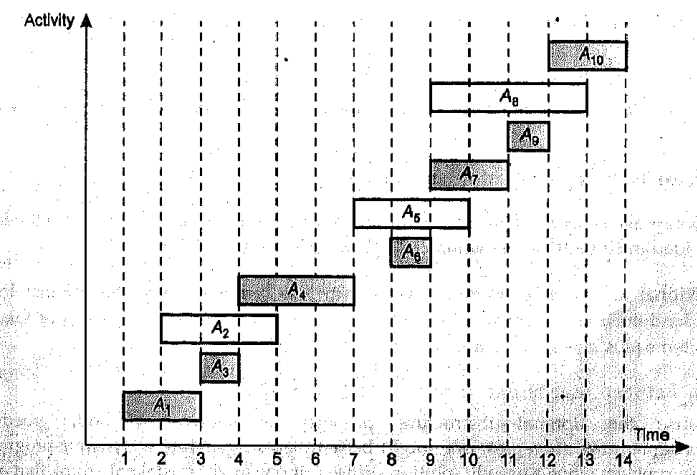
$$S = \langle A_1, A_2, A_3, A_4, A_5, A_6, A_7, A_8, A_9, A_{10} \rangle$$
$$S_i = \langle 1,2,3,4,7,8,9,9,11,12 \rangle$$
$$f_i = \langle 3,5,4,7,10,9,11,13,12,14 \rangle$$

Compute a schedule where the largest number of activities takes place.

Solution. The solution for the above activity scheduling problem using greedy strategy is illustrated below.

Arranging the activities in increasing order of finish time.

Activity	A_1	A_3	A_2	A_4	A_6	A_5	A_7	A_9	A_8	A_{10}
Start	1	3	2	4	8	7	9	11	9	12
Finish	3	4	5	7	9	10	11	12	13	14



Now, schedule A_1

Next, schedule A_3, as A_1 and A_3 are non-interfering

Next, Skip A_2 as it is interfering.

Next, schedule A_4 as A_1, A_3 and A_4 are non-interfering, then next, schedule A_6 as A_1, A_3, A_4 and A_6 are non-interfering.

Skip A_5 as it is interfering.

Next, schedule A_7 as A_1, A_3, A_4, A_6 and A_7 are non-interfering.

Next, schedule A_9 as A_1, A_3, A_4, A_6, A_7 and A_9 are non-interfering.

Skip A_8, as it is interfering.

Next, schedule A_{10} as $A_1, A_3, A_4, A_6, A_7, A_9$ and A_{10} are non-interfering.

Thus, the final activity schedule is

$$\langle A_1, A_3, A_4, A_6, A_7, A_9, A_{10} \rangle$$

22.3 Knapsack Problems

We want to pack n items in your luggage

- The ith item is worth v_i dollars and weighs w_i pounds
- Take as valuable a load as possible, but cannot exceed W pounds
- v_i, w_i, W are integers

0-1 Knapsack Problem

- each item is taken or not taken.
- cannot take a fractional amount of an item or take an item more than once

Fractional Knapsack Problem

- fractions of items can be taken rather than having to make a binary (0-1) choice for each item

Both exhibit the optimal-substructure property

0-1 knapsack problem. Consider a optimal solution. If item j is removed from the load, the remaining load must be the most valuable load weighing at most $W - w_j$.

Fractional knapsack problem. If w of item j is removed from the optimal load, the remaining load must be the most valuable load weighing at most $W - w$ that can be taken from other $n-1$ items plus $w_j - w$ of item j.

Difference Between Greedy and Dynamic Programming

Because the optimal-substructure property is shown by both greedy and dynamic-programming strategies, one might be tempted to generate a dynamic-programming solution to a problem when a greedy solution suffices, or one might mistakenly think that a greedy

solution works when in fact a dynamic-programming solution is required. The most important difference between greedy algorithms and dynamic programming is that we don't solve every optimal sub-problem with greedy algorithms. In some cases, greedy algorithms can be used to produce sub-optimal solutions. That is, solutions which aren't necessarily optimal, but are perhaps very close.

In dynamic programming, we make a choice at each step, but the choice may depend on the solutions to sub-problems. In a greedy algorithm, we make whatever choice seems best at the moment and then solve the sub-problems arising after the choice is made. The choice made by a greedy algorithm may depend on choices so far, but it cannot depend on any future choices or on the solutions to sub-problems. Thus, unlike dynamic programming, which solves the sub-problems bottom up, a greedy strategy usually progresses in a top-down fashion, making one greedy choice after another, interactively reducing each given problem instance to a smaller one.

Fractional knapsack problem can be solvable by the greedy strategy whereas the 0-1 problem is not. To solve the fractional problem

- Compute the value per pound v_i / w_i for each item
- Obeying a greedy strategy, we take as much as possible of the item with the greatest value per pound.
- If the supply of that item is exhausted and we can still carry more, we take as much as possible of the item with the next value per pound, and so forth until we cannot carry any more.
- sorting the items by value per pound, the greedy algorithm runs in $O(n \lg n)$ time .

Fractional Knapsack (Array v, Array w, int W)

```
for i = 1 to Size(v)
    do p[i] = v[i] / w[i]
Sort-Descending(p)
i ← 1
while (W > 0)
    do amount = min(W, w[i])
    solution[i] = amount
    W = W-amount
    i ← i + 1
return solution
```

0-1 knapsack problem cannot be solved by the greedy strategy because it is unable to fill the knapsack to capacity, and the empty space lowers the effective value per pound of the load and we must compare the solution to the sub-problem in which the item is included with the solution to the sub-problem in which the item is excluded before we can make the choice.

Example. *Consider 5 items along their respective weights and values*

$$I = \langle I_1, I_2, I_3, I_4, I_5 \rangle$$
$$w = \langle 5, 10, 20, 30, 40 \rangle$$
$$v = \langle 30, 20, 100, 90, 160 \rangle$$

The capacity of knapsack $W = 60$. *Find the solution to the fractional knapsack problem.*

Solution. Initially,

Item	w_i	v_i
I_1	5	30
I_2	10	20
I_3	20	100
I_4	30	90
I_5	40	160

Taking value per weight ratio *i.e.*, $p_i = v_i / w_i$

Item	w_i	v_i	$p_i = v_i / w_i$
I_1	5	30	6.0
I_2	10	20	2.0
I_3	20	100	5.0
I_4	30	90	3.0
I_5	40	160	4.0

Now, arrange the value of p_i in decreasing order.

Item	w_i	v_i	$p_i = v_i / w_i$
I_1	5	30	6.0
I_3	20	100	5.0
I_5	40	160	4.0
I_4	30	90	3.0
I_2	10	20	2.0

Now, fill the knapsack according to the decreasing value of p_i.

First we choose item I_1 whose weight is 5, then choose item I_3 whose weight is 20. Now the total weight in knapsack is $5+20=25$.

Now, the next item is I_5 and its weight is 40, but we want only 35. So we choose fractional part of if *i.e.*,

35
20
5

$\rightarrow 60$

The value of fractional part of I_5 is

$$\frac{160}{40} \times 35 = 140$$

Thus the maximum value is

$$= 30 + 100 + 140 = 270$$

22.4 Huffman Codes

Data can be encoded efficiently using Huffman codes. It is a widely used and very effective technique for compressing data ; savings of 20% to 90% are typical, depending on the characteristics of the file being compressed. Huffman's greedy algorithm uses a table of the frequencies of occurrence of each character to build up an optimal way of representing each character as a binary string.

Suppose we have 10^5 characters in a data file. Normal storage: 8 bits per character (ASCII)- 8×10^5 bits in file. But, we want to compress the file and store it compactly. Suppose only 6 characters appear in the file :

	a	*b*	*c*	*d*	*e*	*f*	Total
Frequency	45	13	12	16	9	5	100

How can we represent the data in a compact way ?

(i) **Fixed Length code.** Each letter represented by an equal number of bits. With a fixed length code, at least 3 bits per character :

For example :

a	000
b	001
c	010
d	011
e	100
f	101

For a file with 10^5 characters, we need 3×10^5 bits.

(ii) A **variable-length code** can do considerably better than a fixed-length code, by giving frequent characters short code words and infrequent characters long code words.

For example,

a	0
b	101
c	100
d	111
e	1101
f	1100

Number of bits $= (45*1+13*3+12*3+16*3+9*4+5*4)*1000$

$= 2.24 \times 10^5$ bits.

Thus, 224,000 bits to represent the file, a saving of approximately 25%. In fact, this is an optimal character code for this file.

Let us denote the characters by $C_1, C_2, \ldots, C_n$ and denote their frequencies by $f_1, f_2, \ldots, f_n$. Suppose there is an encoding E in which a bit string S_i of length s_i represents C_i, the length of the file compressed by using encoding E is

$$L(E, F) = \sum_{i=1}^{n} s_i \cdot f_i$$

22.5 Prefix Codes

The prefixes of an encoding of one character must not be equal to a complete encoding of another character *e.g.* 1100 and 11001 are not valid codes because 1100 is a prefix of 11001. This constraint is called the **prefix constraint**. Codes in which no codeword is also a prefix of some other code word are called **prefix codes**.

Shortening the encoding of one character may lengthen the encoding of others. To find an encoding E that satisfies the prefix constraint and minimizes $L(E, F)$.

Prefix codes are desirable because they simplify encoding (compression) and decoding. Encoding is always simple for any binary character code; we just concatenate the code words representing each character of the file. Decoding is also quite simple with a prefix code. Since no codeword is a prefix of any other, the codeword that begins an encoded file is unambiguous. We can simply identify the initial codeword, translate it back to the original character, remove it from the encoded file, and repeat the decoding process on the remainder of the encoded file.

The decoding process needs a convenient representation for the prefix code so that the initial codeword can be easily picked off. A binary tree whose leaves are the given characters provides one such representation. We interpret the binary codeword for a character as the path from the root to that character, where 0 means "go to the left child" and 1 means "go to the right child." Note that these are not binary search trees, since the leaves need not appear in sorted order and internal nodes do not contain character keys.

An optimal code for a file is always represented by a full binary tree, in which every non-leaf node has two children. The fixed-length code in our example is not optimal since its tree, because it is not a full binary tree : there are code words beginning 10 . . . , but none beginning 11 . . . . Since we can now restrict our attention to full binary trees, we can say that if C is the alphabet from which the characters are drawn, then the tree for an optimal prefix code has exactly $|C|$ *leaves, one for each letter of the alphabet, and exactly* $|C|-1$ internal nodes.

Given a tree T corresponding to a prefix code, it is a simple matter to compute the number of bits required to encode a file. For each character c in the alphabet C, let $f(c)$ denote the frequency of c in the file and let $d_T(c)$ denote the depth of c's leaf in the tree. Note that $d_T(c)$ is also the length of the codeword for character c. The number of bits required to encode a file is thus

$$B(T) = \sum_{c \in C} f(c) d_T(c)$$

which we define as the *cost* of the tree T.

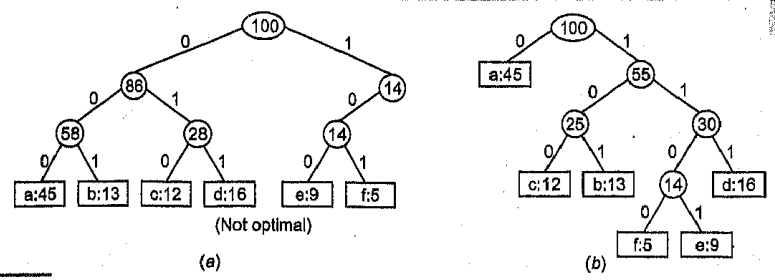
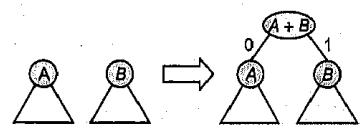


Figure 22.1

Fig. 22.1, Trees corresponding to the coding schemes. Each leaf is labeled with a character and its frequency of occurrence. Each internal node is labeled with the sum of the weights of the leaves in its subtree. (*a*) The tree corresponding to the fixed-length code $a = 000, \ldots, f = 100$ (*b*) The tree corresponding to the optimal prefix code $a = 0$, $b = 101$, ..., $f = 1100$.

22.6 Greedy Algorithm for Constructing a Huffman Code

Huffman invented a greedy algorithm that constructs an optimal prefix code called a ***Huffman code.***



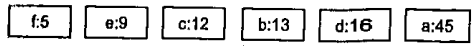
The algorithm builds the tree T corresponding to the optimal code in a bottom-up manner. It begins with a set of $|C|$ leaves and performs a sequence of $|C|-1$ "merging" operations to create the final tree.

In the pseudo code HUFFMAN(C), we assume that C is a set of n characters and that each character $c \in C$ is an object with a defined frequency $f[c]$. A priority queue Q, keyed on f, is used to identify the two least-frequent objects to merge together. The result of the merger of two objects is a new object whose frequency is the sum of the frequencies of the two objects that were merged.

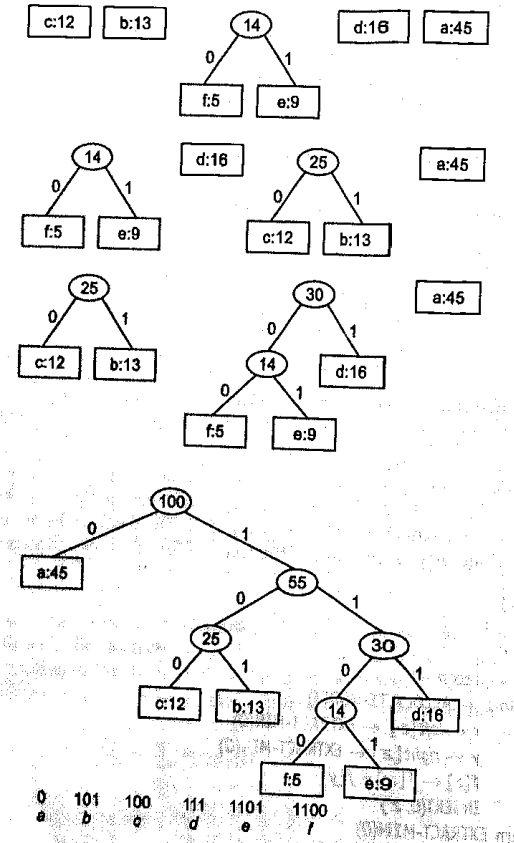
```
HUFFMAN(C)
n ← |C|
Q ← C
for i ← 1 to n − 1
    do z ← ALLOCATE-NODE()
        x ← left[z] ← EXTRACT-MIN(Q)
        y ← right[z] ← EXTRACT-MIN(Q)
        f[z] ← f[x] + f[y]
        INSERT(Q, z)
return EXTRACT-MIN(Q)
```

The analysis of the running time of Huffman's algorithm assumes that Q is implemented as a binary heap. For a set C of n characters, the initialization of Q in line 2 can be performed in $O(n)$ time using the BUILD-HEAP procedure. The for loop in lines 3-8 is executed exactly $|n|-1$ times, and since each heap operation requires time $O(n \lg n)$, the loop contributes $O(n \lg n)$ to the running time. Thus, the total running time of HUFFMAN on a set of n characters is $O(n \lg n)$.

Example :



The algorithm is based on a reduction of a problem with n characters to a problem with $n-1$ characters. A new character replaces two existing ones.



Using Huffman codes

- Each message has a different tree. The tree must be saved with the message.
- Huffman codes are effective for long files where the savings in the message can offset the cost for storing the tree.
- Decode files by starting at root and proceeding down the tree according to the bits in the message (0 = left, 1 = right). When a leaf is encountered, output the character at that leaf and restart at the root.
- Huffman codes are also effective when the tree can be pre-computed and used for a large number of messages (*e.g.*, a tree based on the frequency of occurrence of characters in the English language).
- Huffman codes are not very good for random files (each character about the same frequency).

22.7 Activity or Task Scheduling Problem

This is the problem of optimally scheduling unit-time tasks on a single processor, where each task has a dead line and a penalty that must be paid if the dead line is missed.

A **unit time task** is a job such as a program to be run on a computer that requires exactly one unit of time to complete. Given a finite set S of unit-time tasks, a schedule for S is a permutation of S specifying the order in which these tasks are to be performed. The first task in the schedule begins at time 0 and finishes at time 1, the second task begins at time 1 and finishes at time 2 and so on.

The problem of scheduling unit time tasks with dead lines and penalties for a single processor has the following inputs :

- a set $S = \{1, 2, 3, \ldots n\}$ of n-unit-time tasks.
- a set of n integer dead lines $d_1, d_2, d_3, \ldots d_n$ such that each d_i satisfies $1 \le d_i \le n$ and task i is supposed to finish by time d_i and
- a set of n non-negative weights or penalties $w_1, w_2, \ldots w_n$ such that a penalty w_i is incurred if task i is not finished by time d_i and no penalty is incurred if a task finishes by its dead line.

Here we find a schedule for S that minimizes the total penalty incurred for missed dead lines.

A task is **late** in this schedule if it finished after its dead line. Otherwise, the task is **early** in the schedule. An arbitrary schedule can always be put into **early-first form**, in which the early tasks precede the late tasks, *i.e.*, if some early task x follows some late task y, then we can switch the positions of x and y without affecting x being early or y being late.

An arbitrary schedule can always be put into **canonical form**, in which the early tasks precede the late tasks and the early tasks are scheduled in order of non-decreasing dead lines.

The search for an optimal schedule reduces to finding a set A of tasks that are to be early in the optimal schedule. Once A is determined, we can create the actual schedule by listing the elements of A in order of non-decreasing dead line, then listing the late tasks (*i.e.*, $S - A$) in any order, producing a canonical ordering of the optimal schedule.

A set A of tasks is **independent** if there exists a schedule for these tasks such that no tasks are late. So, the set of early tasks for a schedule forms an independent set of tasks 'I' denote the set of all independent sets of tasks.

For any set of tasks A, A is independent if for $t = 0,1,2,\ldots n$ we have $N_t(A) \le t$ where $N_t(A)$ denote the number of tasks in A whose dead line is t or earlier, *i.e.*, if the tasks in A are scheduled in order of monotonically increasing dead lines, then no task is late.

Example. *Find the optimal schedule for the following task with given weights (penalties) and dead lines.*

	1	2	3	4	5	6	7
d_i	4	2	4	3	1	4	6
w_i	70	60	50	40	30	20	10

Solution. According to Greedy algorithm we sort the jobs in decreasing order of their penalties so that minimum of penalties will be charged.

In this problem, we can see that maximum time for which uniprocessor machine will run is 6 units because it is the maximum dead line.

Let T_i represents the tasks where $i = 1$ to 7

T_2	T_3	T_4	T_1	T_7	T_5	T_6

0 1 2 3 4 5 6 7

T_5 and T_6 cannot be accepted after T_7 so penalty is

$$w_5 + w_6 = 30 + 20 = 50. \qquad \langle 2\ 3\ 4\ 1\ 7\ 5\ 6 \rangle$$

other schedule is

T_2	T_4	T_1	T_3	T_7	T_5	T_6

0 1 2 3 4 5 6 7 $\langle 2\ 4\ 1\ 3\ 7\ 5\ 6 \rangle$

There can be many other schedules but $\langle 2\ 4\ 1\ 3\ 7\ 5\ 6 \rangle$ is optimal.

Note *In this smaller dead line come first and complete the task earlier than larger dead line tasks.*

Example. *Let $n = 4$ $(P_1, P_2, P_3, P_4) = (100, 10, 15, 27)$ and $(d_1, d_2, d_3, d_4) = (2, 1, 2, 1)$ where P_i are profits on processes or job and d_i are dead line of completion. Find the optimal schedule.*

Solution. Max. dead line is 2 so max. number of processes that are scheduled is 2.

Feasible Solution	Processing Sequence	Value
(1, 2)	(2, 1)	10 + 100 = 110
(1, 3)	(1, 3) or (3, 1)	100 + 15 = 115
(1, 4)	(4, 1)	27 + 100 = 127
(2, 3)	(2, 3)	25
(3, 4)	(4, 3)	42
(1)	1	100
(2)	2	10
(3)	3	15
(4)	4	27

Thus, the optimal schedule is (4, 1) and profit 127.

22.8 Travelling Sales Person Problem

In this a salesman needs to visit 'n' cities in such a manner that all cities must be visited at once and in the end he returns to the city from where he started with minimum cost.

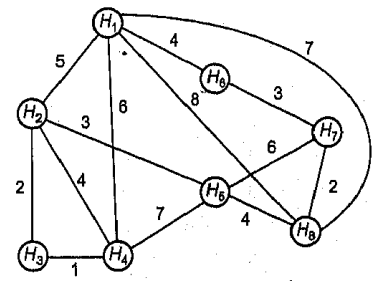
Suppose the cities are $x_1, x_2, \ldots x_n$ where cost c_{ij} denotes the cost of travelling from city x_i to x_j. The travelling salesman problem is to find a route starting and ending at x_1 that will take in all the cities with the minimum cost.

Greedy Strategy for Travelling Salesman Problem

Start with any arbitrary city called x_1 then choose the minimum cost city from x_1. The method is represented until all the cities are visited and at every step we have to select the city with the minimum weight.

Example. *A newspaper agent daily drops the newspaper to the area assigned in such a manner that he has to cover all the houses in the respective area with minimum travel cost. Compute the minimum travel cost.*

The area assigned to the agent where he has to drop the newspaper is shown in the Fig.



Solution. The cost-adjacency matrix of graph G is as follows :

$\text{cost}_{ij} =$

	H_1	H_2	H_3	H_4	H_5	H_6	H_7	H_8
H_1	0	5	0	6	0	4	0	7
H_2	5	0	2	4	3	0	0	0
H_3	0	2	0	1	0	0	0	0
H_4	6	4	1	0	7	0	0	0
H_5	0	3	0	7	0	0	6	4
H_6	4	0	0	0	0	0	3	0
H_7	0	0	0	0	6	3	0	2
H_8	7	0	0	0	4	0	2	0

The tour starts from area H_1 and then select the minimum cost area reachable from H_1.

	H_1	H_2	H_3	H_4	H_5	H_6	H_7	H_8
H_1	0	5	0	6	0	4	0	7
H_2	5	0	2	4	3	0	0	0
H_3	0	2	0	1	0	0	0	0
H_4	6	4	1	0	7	0	0	0
H_5	0	3	0	7	0	0	6	4
H_6	4	0	0	0	0	0	3	0
H_7	0	0	0	0	6	3	0	2
H_8	7	0	0	0	4	0	2	0

Mark area H_6 because it is the minimum cost area reachable from H_1 then select minimum cost area reachable from H_6.

	H_1	H_2	H_3	H_4	H_5	H_6	H_7	H_8
H_1	0	5	0	6	0	4	0	7
H_2	5	0	2	4	3	0	0	0
H_3	0	2	0	1	0	0	0	0
H_4	6	4	1	0	7	0	0	0
H_5	0	3	0	7	0	0	6	4
H_6	4	0	0	0	0	0	3	0
H_7	0	0	0	0	6	3	0	2
H_8	7	0	0	0	4	0	2	0

Mark area H_7 because it is the minimum cost area reachable from area H_6 and then select minimum cost area reachable from H_7.

	H_1	H_2	H_3	H_4	H_5	H_6	H_7	H_8
H_1	0	5	0	6	0	4	0	7
H_2	5	0	2	4	3	0	0	0
H_3	0	2	0	1	0	0	0	0
H_4	6	4	1	0	7	0	0	0
H_5	0	3	0	7	0	0	6	4
H_6	4	0	0	0	0	0	3	0
H_7	0	0	0	0	6	3	0	2
H_8	7	0	0	0	4	0	2	0

Mark area H_8 and select minimum cost area reachable from H_8.

	H_1	H_2	H_3	H_4	H_5	H_6	H_7	H_8
H_1	0	5	0	6	0	4	0	7
H_2	5	0	2	4	3	0	0	0
H_3	0	2	0	1	0	0	0	0
H_4	6	4	1	0	7	0	0	0
H_5	0	3	0	7	0	0	6	4
H_6	4	0	0	0	0	0	3	0
H_7	0	0	0	0	6	3	0	2
H_8	7	0	0	0	4	0	2	0

Mark area H_5 and select minimum cost area reachable from H_5.

	H_1	H_2	H_3	H_4	H_5	H_6	H_7	H_8
H_1	0	5	0	6	0	4	0	7
H_2	5	0	2	4	3	0	0	0
H_3	0	2	0	1	0	0	0	0
H_4	6	4	1	0	7	0	0	0
H_5	0	3	0	7	0	0	6	4
H_6	4	0	0	0	0	0	3	0
H_7	0	0	0	0	6	3	0	2
H_8	7	0	0	0	4	0	2	0

Mark area H_2 and select minimum cost area reachable from H_2.

	H_1	H_2	H_3	H_4	H_5	H_6	H_7	H_8
H_1	0	5	0	6	0	4	0	7
H_2	5	0	2	4	3	0	0	0
H_3	0	2	0	1	0	0	0	0
H_4	6	4	1	0	7	0	0	0
H_5	0	3	0	7	0	0	6	4
H_6	4	0	0	0	0	0	3	0
H_7	0	0	0	0	6	3	0	2
H_8	7	0	0	0	4	0	2	0

Mark area H_3 and select minimum cost area reachable from H_3.

	H_1	H_2	H_3	H_4	H_5	H_6	H_7	H_8
H_1	0	5	0	6	0	4	0	7
H_2	5	0	2	4	3	0	0	0
H_3	0	2	0	1	0	0	0	0
H_4	6	4	1	0	7	0	0	0
H_5	0	3	0	7	0	0	6	4
H_6	4	0	0	0	0	0	3	0
H_7	0	0	0	0	6	3	0	2
H_8	7	0	0	0	4	0	2	0

Mark area H_4 and select minimum cost area reachable from H_4 it is H_1.
So, using the greedy strategy we get the following

$$H_1 \xrightarrow{4} H_6 \xrightarrow{3} H_7 \xrightarrow{2} H_8 \xrightarrow{4} H_5 \xrightarrow{3} H_2 \xrightarrow{2} H_3 \xrightarrow{1} H_4 \xrightarrow{6} H_1$$

Thus, the minimum travel cost

$$= 4+3+2+4+3+2+1+6 = 25$$

22.9 Matroids

A **matroid** is an ordered pair $M(S, I)$ satisfying the following conditions.

1. S is a finite nonempty set.
2. I is a nonempty family of subsets of S, called the **independent** subsets of S, such that if $B \in I$ and $A \subseteq B$, then $A \in I$. We say that I is **hereditary** if it satisfies this property. Note that the empty set ϕ is necessarily a member of I.
3. If $A \in I$, $B \in I$ and $|A| < |B|$, then there is some element $x \in B - A$ such that $A \cup \{x\} \in I$. We say that M satisfies the **exchange property**.

We say that a matroid $M(S, I)$ is ***weighted*** if there is an associated weight function w that assigns a strictly positive weight $w(x)$ to each element $x \in S$. The weight function w extends to subsets of S by summation :

$$w(A) = \sum_{x \in A} w(x)$$

For any $A \subseteq S$.

For example, if we let $w(e)$ denote the length of an edge e in a graphic matroid M_G, then $w(A)$ is the total length of the edges in edge set A.

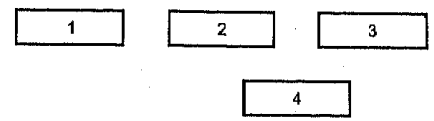
22.10 Minimum Spanning Tree

This is yet another example of greedy technique where we compute the minimum spanning tree in a given graph. This problem can be comprehensively studied by including Kruskal's and Prim's minimum spanning tree algorithm which are discussed in detail in the coming chapters.

Example. *"Not just any greedy approach to the activity-selection problem produces a maximum-size set of mutually compatible activities. Give an example to show that the approach of selecting the activity of least duration from those that are compatible with previously selected activities does not work. Do the same for the approaches of always selecting the compatible activity that overlaps the fewest other remaining activities and always selecting the compatible remaining activity with the earliest start time".*

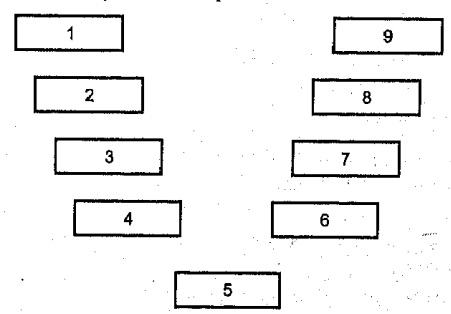
Solution.

◄ Choose the next activity with the shortest duration :



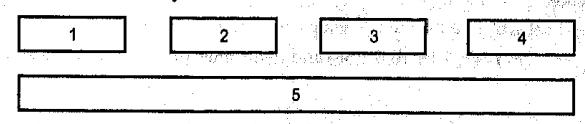
This gives us $A = \{1, 4\}$. The original algorithm returns $A = \{1, 2, 3\}$

◄ Choose the next activity that overlaps the last with the remaining activities.



This gives us $A = \{1, 5, 9\}$ (since they only overlap two others). The original algorithm returns $A = \{1, 4, 6, 9\}$

◄ Choose the next activity with the earliest start time.



If we go by earliest start time, we have to pick 5 first, thus excluding all other ones.
The original greedy algorithm returns $A = \{1, 2, 3, 4\}$

Example. *Consider the problem of making change for N cents using the least number of coins. Describe a greedy algorithm to make change consisting of quarters, dimes, nickels and pennies.*

(*Hint. a quarter* $=25c$, *a dime* $=10c$, *a nickel* $=5c$ *and a penny* $=1c$)

Solution.

Step 1. Let $f(N)$ be the minimum number of change coins needed for n cents.

Step 2. Define the recurrence. Then we have

$$f(N)=\begin{cases} 1 \text{ if } N=1 \\ 1 \text{ if } N=5 \\ 1 \text{ if } N=10 \\ 1 \text{ if } N=25 \\ \min\{f(N-25)+1, f(N-10)+1, f(N-5)+1, f(N-1)+1\} \text{ otherwise} \end{cases}$$

Step 3. Compute $f(N)$

Step 4. If we need to find all different coins, another array is needed to keep the intermediate computation.

Exercise

1. Given 10 activities along with their start and finish time as

   $S=\langle A_1, A_2, A_3, A_4, A_5, A_6, A_7, A_8, A_9, A_{10}\rangle$
   $S_i=\langle 1,2,3,4,5,6,7,8,9,10\rangle$
   $f_i=\langle 5,3,4,6,7,8,11,10,12,13\rangle$

   Compute a schedule where the largest number of activities take place.
2. Find an optimal solution for the knapsack Instances

   $n=7,\ M=15\ (P_1, P_2, \ldots, P_7)=(10,5,15,7,6,18,3)$ and $(w_1, w_2, \ldots w_7)=(2,3,5,7,1,4,1)$
3. Consider the knapsack instance $n=3(w_1, w_2, w_3)=(2,3,4)$ and $(P_1, P_2, P_3)=(2,3,5)\ \ M=5$. Find the optimal solution.
4. Show how to solve fractional knapsack problem in $\theta(n)$ time.
5. Generalize Huffman's algorithm for ternary code words and prove that it yields optimal ternary codes.
6. Let $C=\{0,1,\ldots,n-1\}$ be a set of characters. Show that any optimal prefix code on C can be represented by a sequence of $2n-1+n\lceil \lg n\rceil$.
7. Find the optimal schedule for the following jobs with $n=7$ profits

   $(P_1, P_2, \ldots P_7)=(3,5,18,20,6,1,38)$ and dead lines $(d_1, d_2, d_3, \ldots d_7)=(1,3,3,4,1,2,1)$?

CHAPTER

23

Backtracking

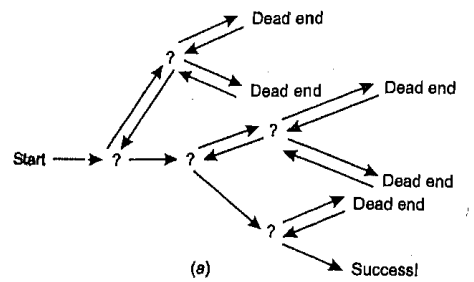
23.1 Introduction

Have you ever seen poor blind people walking in roads? If they find any obstacles in their way, they would just move backward. Then they will proceed in other direction. How a blind person could move backward when he finds obstacles ? Simple answer... by intelligence! Similarly, if an algorithm backtracks with intelligence, it is called *Backtracking algorithm*.

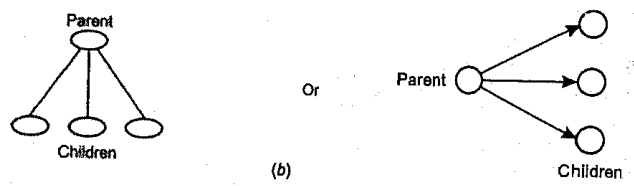
Suppose we have to make a series of *decisions*, among various *choices*, where we don't have enough information to know what to choose and each decision leads to a new set of choices. Some sequence of choices (possibly more than one) may be a solution to our problem. **Backtracking** is a methodical way of trying out various sequences of decisions, until we find one that "works".

In the following Figure,

- Each non-leaf node in a tree is a parent of one or more other nodes (its children)
- Each node in the tree, other than the root, has exactly one parent



Usually, however, we draw our trees *downward*, with the root at the top.



A tree is composed of nodes

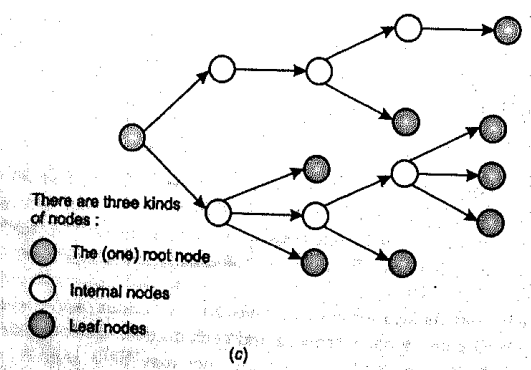


Figure 23.1

Backtracking can be thought of as searching a tree for a particular "goal" leaf node.

There is a type of data structure called a tree, but we are **not** using it here. Actually, if we diagram the sequence of choices we make, the diagram looks like a tree. Our backtracking algorithm "sweeps out a tree" in "problem space". Backtracking is really quite simple–we "explore" each node, as follows :

To "explore" node N :

1. If N is a goal node, return "success"
2. If N is a leaf node, return "failure"
3. For each child C of N,

   Explore C

   If C was successful, return "success"
4. Return "failure"

The basic idea of backtracking is to build up a vector one component at a time and to test whether the vector being formed has any chance of success. The major advantage of back tracking algorithm is that if it is realized that the partial vector generated does not lead to an optimal solution then that vector may be ignored.

Backtracking algorithm determines the solution by systematically searching the solution space for the given problem. Backtracking is a **depth first search** with some bounding function. All solutions using backtracking are required to satisfy a complex set of constraints. The constraints may be explicit or implicit.

Explicit constraints are rules, which restrict each vector element to be chosen from the given set. **Implicit constraints** are rules, which determine which of the tuples in the solution space, actually satisfy the criterion function.

23.2 Recursive Maze Algorithm

Recursive maze algorithm is one of the good examples for backtracking algorithms. In fact Recursive maze algorithm is one of the most available solutions for solving maze.

Maze

Maze is an area surrounded by walls; in between we have a path from starting position to ending position. We have to start from the starting point and travel towards the ending point.

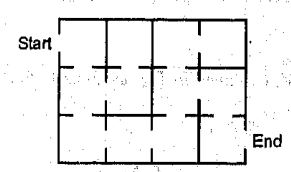


Figure 23.2

Principle of Maze

As explained above, in maze we have to travel from the starting point to ending point. The problem is to choose the path. If we find any dead-end before ending point, we have to backtrack and change the direction. The direction for traversing is North, East, West and South. We have to continue "move and backtrack" until we reach the ending point.

Assume that we are having a two-dimensional maze cell [WIDTH][HEIGHT]. Here cell[x][y]=1 denotes wall and cell[x][y]=0 denotes free cell in the particular location x, y in the maze. The directions we can move in the array are North, East, West and South. The first step is to make the boundary of the two-dimensional array as 1 so that we won't go out of the maze, and always reside inside the maze at any time.

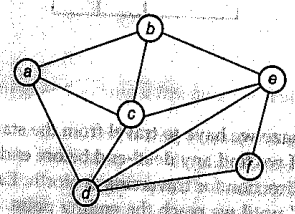
Example Maze						
1	1	1	1	1	1	1
1	0	0	0	1	1	1
1	1	1	0	1	1	1
1	1	1	0	0	0	1
1	1	1	1	1	0	1
1	1	1	1	1	1	1

Now start moving from the starting position (since the boundary is filled by 1) and find the next free cell, then move to the next free cell and so on. If we reach a dead-end, we have to backtrack and make the cells in the path as 1(wall). Continue the same process till the ending point is reached.

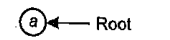
23.3 Hamiltonian Circuit Problem

Given a graph $G=(V, E)$, we have to find the ***Hamiltonian Circuit*** using Backtracking approach. We start our search from any arbitrary vertex, say '*a*'. This vertex '*a*' becomes the root of our implicit tree. The first element of our partial solution is the first intermediate vertex of the Hamiltonian Cycle that is to be constructed. The next adjacent vertex is selected on the basis of alphabetical (or numerical) order. If at any stage any arbitrary vertex makes a cycle with any vertex other than vertex '*a*' then we say that **dead end** is reached. In this case we backtrack one step, and again the search begins by selecting another vertex and backtrack the element from the partial solution must be removed. The search using backtracking is successful if a Hamiltonian cycle is obtained.

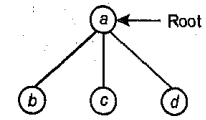
Example. *Consider a graph $G=(V, E)$ shown in Fig. we have to find a Hamiltonian circuit using Back tracking method.*



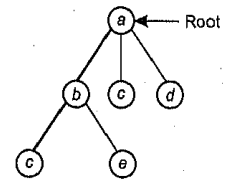
Solution. Firstly, we start our search with vertex '*a*', this vertex '*a*' becomes the root of our implicit tree.



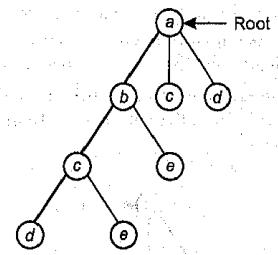
Next, we choose vertex '*b*' adjacent to '*a*' as it comes first in lexicographical order (b, c, d).



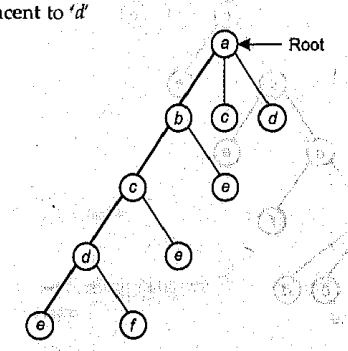
Next, we select '*c*' adjacent to '*b*'



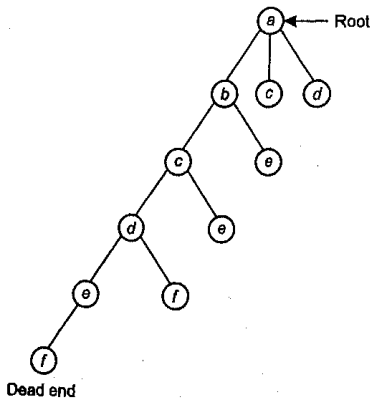
Next, we select '*d*' adjacent to '*c*'



Next, we select '*e*' adjacent to '*d*'

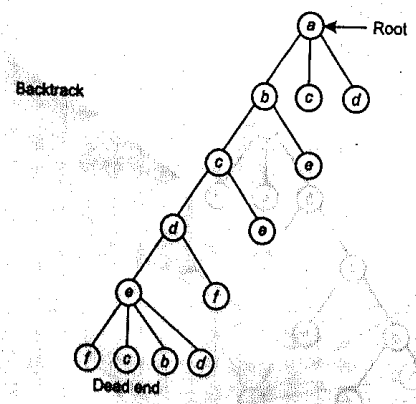


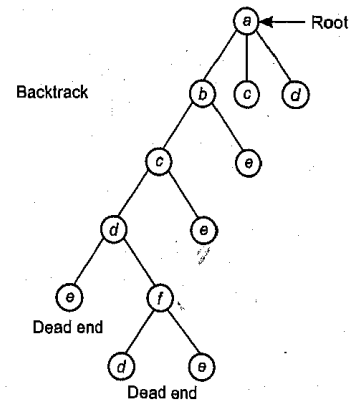
Next, we select vertex 'f' adjacent to 'e'. The vertex adjacent to 'f' are d and e but they have already visited. Thus, we get the dead end and we back track one step and remove the vertex 'f' from partial solution.



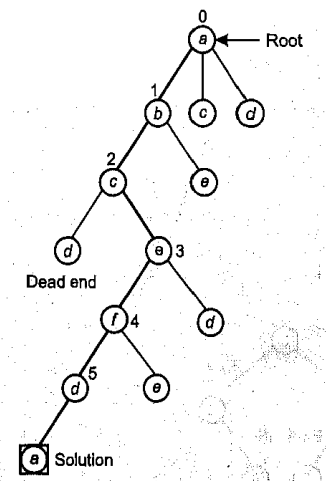
From back tracking, the vertex adjacent to 'e' are b, c, d, f from which vertex 'f' has already been checked and b, c, d have already visited. So, again we back track one step. Now, the vertex adjacent to d are e, f from which e has already been checked and adjacent of 'f' are d and e. If 'e' vertex visited then again we get dead state. So again we backtrack one step.

Now, adjacent to c is 'e' and adjacent to 'e' is 'f' and adjacent to 'f' is 'd' and adjacent to 'd' is 'a.' Here, we get the Hamiltonian cycle as all the vertex other than the start vertex 'a' is visited only once. ($a-b-c-e-f-d-a$).





Again back track



Here we have generated one Hamiltonian circuit but other Hamiltonian circuit can also be obtained by considering other vertex.

23.4 Subset-sum Problem

In the subset-sum problem we have to find a subset s' of the given set $S = \langle S_1, S_2, S_3, \ldots S_n \rangle$ where the elements of the set S are n positive integers in such a manner that $s' \in S$ and sum of the elements of subset 's' ' is equal to some positive integer 'X'.

The subset-sum problem can be solved by using the backtracking approach. In this implicit tree is a binary tree. The root of the tree is selected in such a way that it represents that no decision is yet taken on any input. We assume that the elements of the given set are arranged in an increasing order :

$$S_1 \le S_2 \le S_3 \ldots \le S_n$$

The left child of the root node indicates that we have to include 'S_1' from the set 'S' and the right child of the root node indicates that we have to exclude 'S_1' . Each node stores the sum of the partial solution elements. If at any stage the number equals to 'X' then the search is successful and terminates.

The dead end in the tree occurs only when either of the two inequalities exist :

- The sum of s' is too large *i.e.*

$$s' + S_i + 1 > X$$

- The sum of s' is too small *i.e.*

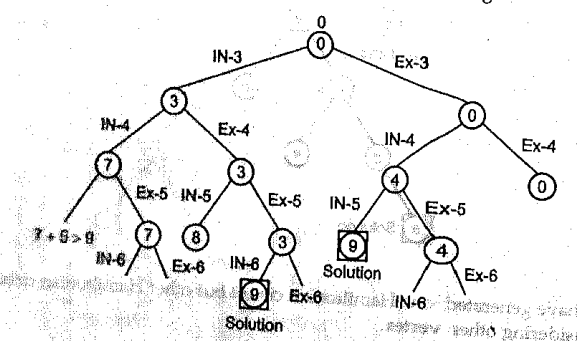
$$s' + \sum_{j=i+1}^{n} S_j < X$$

Example. *Given a set $S = \langle 3, 4, 5, 6 \rangle$ and $X = 9$. Obtain the subset sum using Back tracking approach.*

Solution. Initially $S = \langle 3, 4, 5, 6 \rangle$ and $X = 9$

$$S' = \langle \phi \rangle$$

The implicit binary tree for subset sum problem is shown in Fig.



The number inside a node is the sum of the partial solution elements at a particular level.

Thus, if our partial solution elements sum is equal to the positive integer 'X' then at that time search will terminates, or it continues if all the possible solutions need to be obtained.

23.5 *N*-Queens Problem

N-queens problem is to place n-queens in such a manner on an $n \times n$ chessboard that no two queens attack each other by being in the same row, column or diagonal.

It can be seen that for $n = 1$, the problem has a trivial solution, and no solution exists for $n = 2$ and $n = 3$. So first we will consider the 4-queens problem and then generalise it to n-queens problem.

Given, a 4×4 chessboard and number the rows and column of the chessboard 1 through 4.

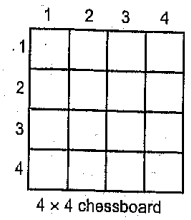


Figure 23.3

Since we have to place 4 queens such as q_1, q_2, q_3 and q_4 on a chessboard, such that no two queens attack each other. In such a condition each queen must be placed on a different row, *i.e.*, we place queen "i" on row "i".

Now, we place queen q_1 in the very first acceptable position (1, 1). Next, we place queen q_2 so that both these queens do not attack each other. We find that if we place q_2 in column 1 and 2 then the dead end is encountered. Thus the first acceptable position for q_2 is column 3 *i.e.*, (2, 3) but then no position is left for placing queen 'q_3' safely. So we backtrack one step and place the queen 'q_2' in (2, 4), the next best possible solution. Then we obtain the position for placing 'q_3' which is (3, 2). But later this position also leads to dead end and no place is found where 'q_4' can be placed safely. Then we have to backtrack till 'q_1' and place it to (1, 2) and then all the other queens are placed safely by moving q_2 to (2, 4), q_3 to (3, 1) and q_4 to (4, 3). That is, we get the solution $\langle 2, 4, 1, 3 \rangle$. This is one possible solution for 4-queens problem. For other possible solution the whole method is repeated for all partial solutions. The other solution for 4-queens problem is $\langle 3, 1, 4, 2 \rangle$ *i.e.*,

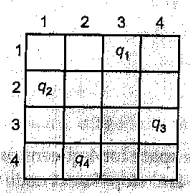


Figure 23.4

The implicit tree for 4-queen problem for solution $\langle 2,4,1,3 \rangle$ is as follows :

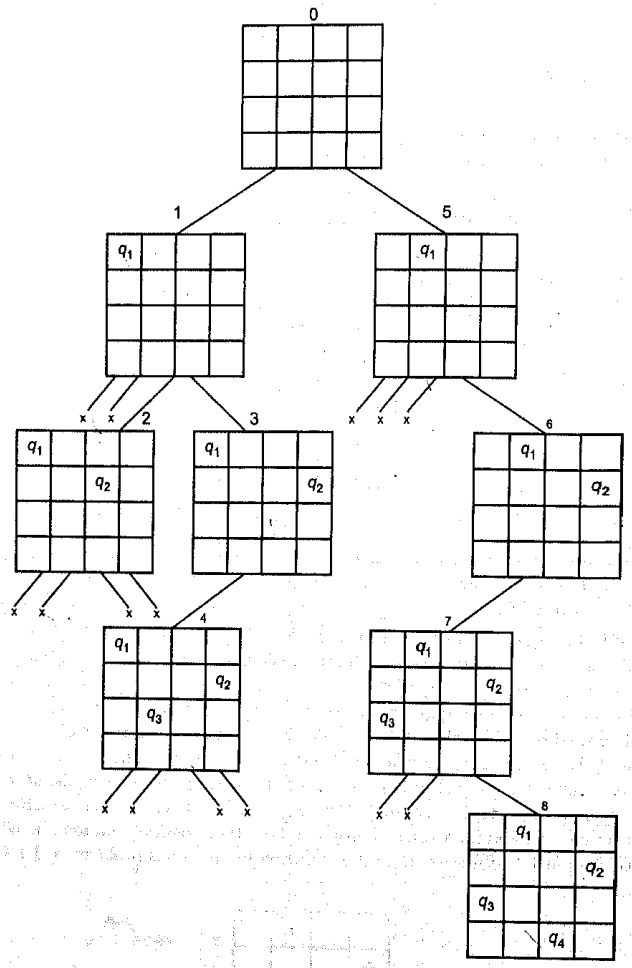


Figure 23.5

Fig.23.6 shows the complete state space for 4-queens problem. But we can use backtracking method to generate the necessary node and stop if next node violates the rule *i.e.*, if two queens are attacking.

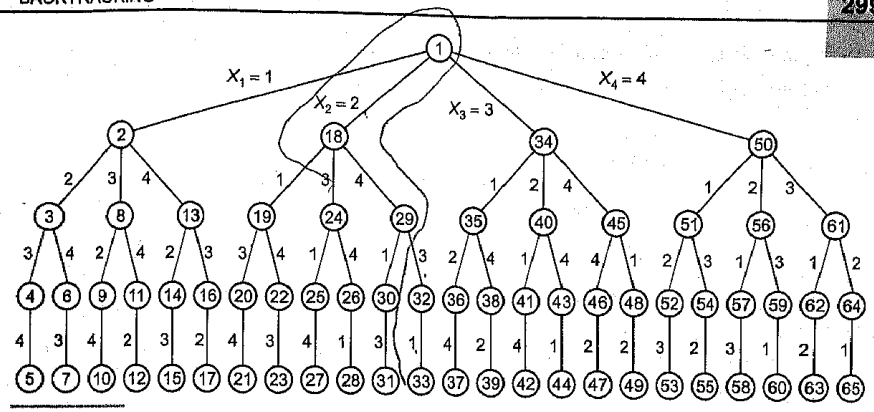


Figure 23.6 **4-queens solution space with nodes numbered in DFS**

It can be seen that all the solutions to the 4-queens problem can be represented as 4-tuples (x_1, x_2, x_3, x_4) where x_i represents the column on which queen "q_i" is placed.

One possible solution for 8 queens problem is shown in the Fig. 23.7

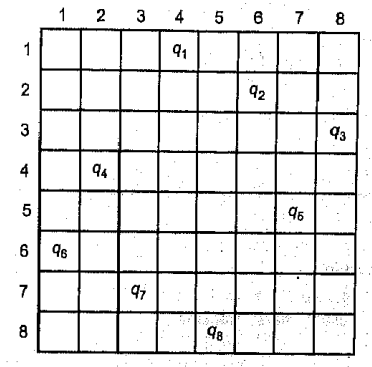


Figure 23.7

Thus, solution for 8-queen problem for $\langle 4,6,8,2,7,1,3,5 \rangle$.

If two queens are placed at positions (i, j) and (k, l).

Then, they are on the same diagonal only if $(i-j)=k-l$ or $i+j=k+l$.

Then *first* equation implies that $j-l=i-k$

The *second* equation implies that $j-l=k-i$

Therefore, two queens lie on the same diagonal if and only if $|j-l|=|i-k|$

Place (k, i) returns a boolean value that is true if the kth queen can be placed in column i. It tests both whether i is distinct from all previous values $x_1, x_2 \ldots x_{k-1}$ and whether there is no other queen on the same diagonal.

Using place, we give a precise solution to the n-queens problem.

```
Place (k,i)
    {
        For j ← 1 to k − 1
            do if ( x[j] = i)
                or (Abs (x[j]) − i) = (Abs (j − k))
            then return false ;
        return true ;
    }
```

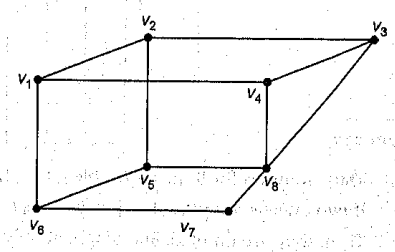
Place (k, i) returns true if a queen can be placed in the kth row and ith column otherwise returns false.

$x[\,]$ is a global array whose first $k-1$ values have been set. $Abs(r)$ returns the absolute value of r.

```
N-Queens (k,n)
 {
     for i ← 1 to n
         do if Place (k,i) then
             { x[k] ← i ;
                 if (k = n) then
                     write (x[1..n)) ;
                 else
                     N-Queens (k + 1, n) ;
             }
 }
```

Exercise

1. For a graph shown, find the Hamiltanian circuit using back tracking technique.



2. Given a set $S = \{1,3,4,5\}$ and $X = 8$, we have to find subset-sum using back tracking approach.
3. Find all the possible solutions for the
   (a) 4×4 chessboard, 4 queens problem
   (b) 8×8 chessboard, 8 queens problem.
4. Run the n-queens program for $n = 8, 9$ and 10. Find the number of solutions your program finds for each value of n.
5. Design a back tracking algorithm for the Hamiltanian cycle problem.
6. Design an algorithm for the n-colouring problem, considering back tracking technique.
7. For which values of n does the n queens problem have no solutions ? Prove your answer.

CHAPTER

24

Branch and Bound

24.1 Introduction

Branch and bound is a systematic method for solving optimization problems. Branch and bound is a rather general optimization technique that applies where the greedy method and dynamic programming fail. However, it is much slower. Indeed, it often leads to exponential time complexities in the worst case. On the other hand, if applied carefully, it can lead to algorithms that run reasonably fast on average. This method was first proposed by **A. H. Land** and **A. G. Doig** in 1960 for linear programming.

Branch-and-bound procedure requires two tools. The first one is a way of covering the feasible region by several smaller feasible sub-regions (ideally, splitting into sub-regions). This is called **branching**; since the procedure may be repeated recursively to each of the sub regions and all produced sub-regions naturally form a tree structure, called search tree or branch-and-bound-tree or something similar. Its nodes are the constructed sub-regions. Another tool is **bounding**, which is a fast way of finding upper and lower bounds for the optimal solution within a feasible sub-region.

The Branch and bound technique like **Backtracking** explores the implicit graph and deals with the optimal solution to a given problem. In this technique at each stage we calculate the bound for a particular node and check whether this bound will be able to give the solution or not .If we find that at any node the solution so obtained is appropriate but the remaining solution is not leading to a best case then we leave this node without exploring. Depth-first search or breadth-first

search is used for calculating the bound. At each stage we have for each node a bound – lower bound for minimization problem and upper bound for maximization problem. For each node, bound is calculating by means of partial solution. The calculated bound is checked with previous best result and if found that new partial solution results leads to worse case, than bound with the best solution so far, is selected and we leave this part without exploring it further. Otherwise the checking is done with the previous best result obtained so far, every partial solution while exploring.

The backtracking algorithm is effective for decision problem, but it is not designed for optimization problems, where we also have a cost function $f(x)$ that we wish to minimize or maximize. The branch and bound design has all the elements of backtracking, except that rather than simply stopping the entire search process any time a solution is found, we continue processing until the best solution is found. In addition, the algorithm has a scoring mechanism to always choose the most promising configuration to explore in each iteration. Because of this approach, branch and bound is sometimes called a **best-first search** strategy. Starting by considering the root problem (the original problem with the complete feasible region), the lower-bounding and upper-bounding procedures are applied to the root problem. If the bounds match, then an optimal solution has been found and the procedure terminates. Otherwise, the feasible region is divided into two or more regions ; these sub-problems partition the feasible region. The algorithm is applied recursively to the sub-problems. If an optimal solution is found to a sub-problem, it is a feasible solution to the full problem, but not necessarily globally optimal. If the lower bound for a node exceeds the best-known feasible solution, no globally optimal solution can exist in the subspace of the feasible region represented by the node. Therefore, the node can be removed from consideration. The search proceeds until all nodes have been solved or pruned, or until some specified threshold is met between the best solution found and the lower bounds on all unsolved sub-problems.

The core of this approach is a simple observation that (for a minimization task) if the lower bound for a sub-region A from the search tree is greater than the upper bound for any other (previously examined) sub region B, then A may be safely discarded from the search. This step is called **pruning**. It is usually implemented by maintaining a global variable 'm' that records the minimum upper bound seen among all sub-regions examined so far ; any node whose lower bound is greater than 'm' can be discarded.

It may happen that the upper bound for a node matches its lower bound ; that value is then the minimum of the function within the corresponding sub-region. Sometimes there is a direct way of finding such a minimum. In both these cases it is said that the node is solved. Note that this node may still be pruned as the algorithm progresses.

Ideally the procedure stops when all nodes of the search tree are either pruned or solved. At that point, all non-pruned sub-regions will have their upper and lower bounds equal to the global minimum of the function. In practice the procedure is often terminated after a given time ; at that point, the minimum lower bound and the maximum upper bound, among all non-pruned sections, define a range of values that contains the global minimum. Alternatively, within an overriding time constraint, the algorithm may be terminated when some error criterion such as (max – min)/(min + max) falls below a specified value.

The efficiency of the method depends critically on the effectiveness of the branching and bounding algorithms used; bad choices could lead to repeated branching, without any pruning,

until the sub-regions become very small. In that case the method would be reduced to an exhaustive enumeration of the domain, which is often impractically large. There is no universal bounding algorithm that works for all problems, and there is little hope that one will ever be found ; therefore the general paradigm needs to be implemented separately for each application, with branching and bounding algorithms that are specially designed for it.

Branch and bound methods may be classified according to the bounding methods and according to the ways of creating/inspecting the search tree nodes. This approach is used for a number of NP-hard problems, such as

- Knapsack problem
- Assignment problem
- Integer programming
- Nonlinear programming
- Travelling salesman problem (TSP) etc.

Backtracking algorithms determine problem solutions by systematically searching the solution space for the given problem instance. This search is facilitated by using a tree **organization** for the **solution** space. For a given solution space many tree organizations may be possible. Figure 23.6 shows a possible tree organization. A tree such as this is called a **permutation tree**.

At this point it is useful to develop some terminology regarding tree organizations of solution spaces. Each node in this tree defines a **problem state**. All paths from the root to other nodes define the **state space** of the problem. **Solution states** are those problem states 'S' for which the path from the root to 's' defines a tuple in the solution space. In Figure 23.6 only leaf nodes are solution states.

Answer states are those solution states 's' for which the path from the root to 'S' defines a tuple that is a member of the set of solutions. The tree organization of the solution space is referred to as the **state space tree**.

24.2 Live Node, Dead Node and Bounding Functions

Once a state space tree has been conceived of for any problem, the problem can be solved by systematically generating the problem states, determining which of these are solution states, and finally determining which solution states are answer states.

There are two fundamentally different ways to generate the problem states. Both of these begin with the root node and generate other nodes.

A node which has been generated and all of whose children have not yet been generated is called a **live node**. The live node whose children are currently being generated is called the E-node (node being expanded). A ***dead node*** is a generated node which is not to be expanded further or all of whose children have been generated. In both methods of generating problem states, we have a list of live nodes. In the first of these two methods as soon as a new child C of the current E-node R is generated, this child will become the new E-node. Then R will become the E-node again when the subtree C has been fully explored. This corresponds to a depth first generation of the problem states. In the second state generation method, the E-node remains the E-node until it is dead. In both methods, ***bounding functions*** are used to kill live nodes without generating all their children.

This is done carefully enough that at the conclusion of the process at least one answer node is always generated or all answer nodes are generated if the problem requires us to find all solutions. Depth first node generation with bounding functions is called *backtracking*. State generation methods in which the *E*-node remains the *E*-node until it is dead lead to **branch-and-bound** methods.

The term branch and bound refers to all state space search methods in which all children of the E-node are generated before any other live node can become the E-node. In branch-and-bound terminology, a BFS-like state space search will be called FIFO (First In First Out) search as the list of live nodes is a first-in-first-out list (or queue). A D-search like state space search will be called LIFO (Last In First Out) search as the list of live nodes is a last-in-first-out list (or stack). As in the case of backtracking, bounding functions are used to help avoid the generation of subtrees that do not contain an answer node.

24.3 FIFO Branch-and-Bound Algorithm

Let us see how a FIFO branch-and-bound algorithm would search the state space tree for the 4-queens problem.

- Initially, there is only one live node, node 1. This is the case when no queen has been placed on the chessboard.
- This node becomes the *E*-node.
- Expand and generate all its children, nodes 2, 18, 34 and 50 are generated.
- These nodes represent a chessboard with queen 1 in row 1 and columns 1, 2, 3 and 4 respectively.
- The only live nodes now are nodes 2, 18, 34 and 50. If the nodes are generated in this order, then the next *E*-node is node 2.
- It is expanded and nodes 3, 8 and 13 are generated.
- Node 3 is immediately killed using the bounding function. As a bounding function, we use the obvious criteria that if $(x_1, x_2, \ldots\ldots x_i)$ is the path to the current *E*-node, then all children nodes with parent-child labelings x_{i+1} are such that $(x_1, x_2, \ldots\ldots x_{i+1})$ represents a chessboard configuration in which no two queens are attacking.
- Nodes 8 and 13 are added to the queue of live nodes and node 18 becomes the next *E*-node.
- Nodes 19, 24 and 29 are generated. Nodes 19 and 24 are killed as a result of the bounding functions.
- Node 29 is added to the queue of live odes.
- The *E*-node is node 34.
- Figure 24.1 shows the portion of the tree of Figure 23.6 that is generated by a FIFO branch and bound search.
- In Figure, nodes that are killed as a result of the bounding function have a "*B*" under them and the numbers outside the nodes give the order in which the nodes are generated by FIFO branch and bound.
- Compare with backtracking algorithm indicates that backtracking is a superior method for this search problem.

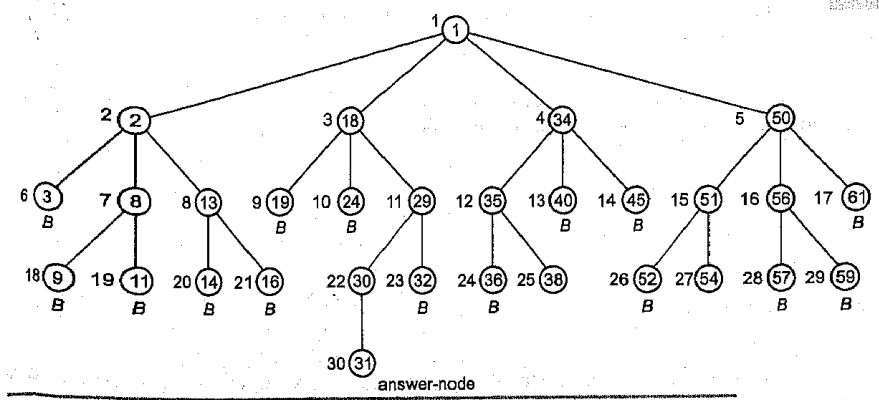


Figure 24.1 Portion of 4-queens state space tree generated by FIFO branch and bound.

24.4 Least Cost (LC) Search

In both LIFO and FIFO branch-and-bound the selection rule does not give preference to nodes that will lead to answer quickly but just queues those behind the current live nodes. In 4-queen problem, if three queens have been placed on the board, it is obvious that the answer may be reached in one more move. The rigid selection rule requires that other live nodes be expanded and then, the current node be tested.

The search for an answer node can often be speeded by using an "intelligent" ranking function $\hat{C}(.)$ for live nodes. The next *E*-node is selected on the basis of this ranking function.

The ideal way to assign ranks would be on the basis of the additional computational effort (or cost) needed to reach an answer node from the life node. For any node 'x', this cost could be :

1. The number of nodes in the subtree x that need to be generated before an answer node is generated.
2. The number of levels the nearest answer node is from 'x'.

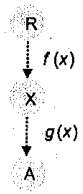
Using cost measure 2, the cost of the root of the tree of figure 24.1 to 4 (node 31 is four levels from node 1) and the costs of nodes 18 and 34, 29 and 35 and 30 and 38 are respectively 3, 2, and 1. The costs of all remaining nodes on levels 2, 3 and 4 are respectively greater than 3, 2 and 1. Using these costs as a basis to select the next *E*-node, the *E*-nodes are nodes 1, 18, 29 and 30 (in that order). The only other nodes to get generated are nodes 2, 34, 50, 19, 24, 32 and 31. If cost measure 1 is used, then the search would always generate the minimum number of nodes every branch-and-bound type algorithm must generate. If cost measure 2 is used then the only nodes to become E-nodes are the nodes on the path from the root to the nearest answer node.

The difficulty with using either of these ideal cost functions is that computing the cost of a node usually involves a search of the subtree x for an answer node.

Let x be a node in the state space tree and the cost function of x is

$$C(x) = f(x) + g(x)$$

where $f(x)$ is the real cost of traversing from the root to x and is a non-decreasing function and $g(x)$ is an estimated cost from node x to an answer node.



Let $\hat{g}(x)$ be an estimate of the additional effort needed to reach an answer node from x. Node x is assigned a rank using a function $\hat{c}(.)$ such that

$$\hat{c}(x) = f(h(x)) + \hat{g}(x)$$

where $h(x)$ is the cost of reaching x from the root and $f(.)$ is any non-decreasing function.

A search strategy that uses a cost function $\hat{c}(x) = f(h(x)) + \hat{g}(x)$ to select the next E-node would always choose for its next E-node a live node with least $\hat{c}(.)$. Hence, such a search strategy is called an LC-search (Least Cost search). It is interesting to note that BFS and D-search are special cases of LC-search. If we use $\hat{g}(x) \equiv 0$ and $f(h(x)) =$ level of node x, then a LC-search generates nodes by levels. This is essentially the same as a BFS. If $f(h(x)) \equiv 0$ and $\hat{g}(x) \geq \hat{g}(y)$ whenever y is a child of x, then the search is essentially a D-search. An LC-search coupled with **bounding functions** is called an LC branch-and-bound search.

24.5 The 15-Puzzle : An Example

The 15-puzzle consists of 15 numbered tiles on a square frame with a capacity of 16 tiles. We are given an initial arrangement of the tiles and the objective is to transform this arrangement into the goal arrangement through a series of legal moves. The only legal moves are ones in which a tile adjacent to the empty spot (ES) is moved to ES.

1	3	4	15
2		5	12
7	6	11	14
8	9	10	13

(*a*) An arrangement

1	2	3	4
5	6	7	8
9	10	11	12
13	14	15	

(*b*) Goal arrangement

From the initial arrangement four moves are possible. We can move any one of the tiles numbered 2, 3, 5 or 6 to the empty spot. Following this move, other moves can be made.

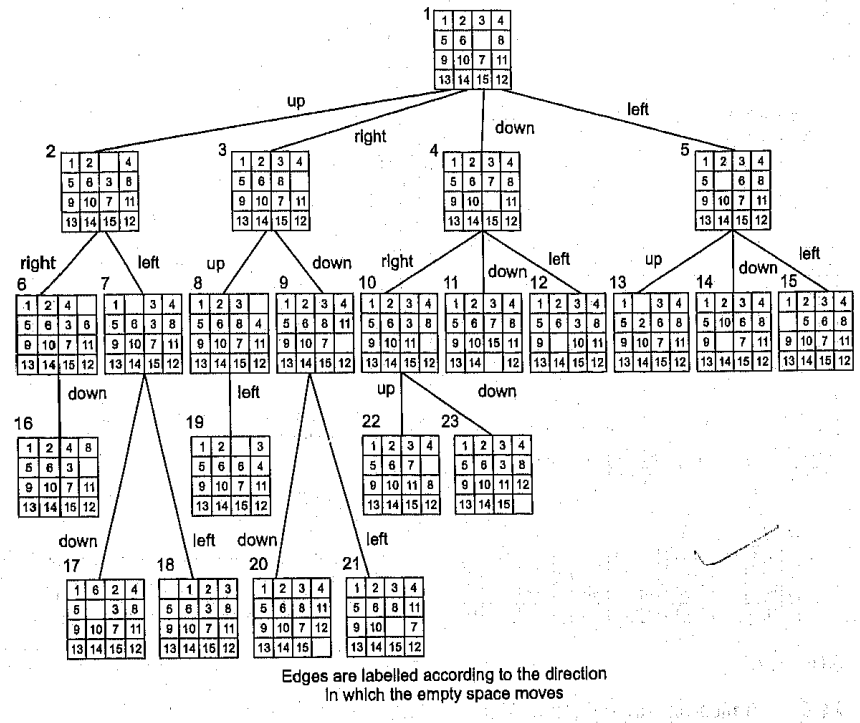


Figure 24.2 Part of the state space tree for the 15-puzzle.

Branch-and-Bound Method

- $c(x) = f(x) + g(x)$, where $f(x)$ is the length of the path from the root to node x and $g(x)$ is an estimate of the length of a shortest path from x downward to a goal node.
- $g(x) =$ number of non-blank tiles not in their goal position.
- Initially, the bound is set to infinity $B \leftarrow \infty$.
- Then whenever an answer is reached, the bound is compared.
- If the cost of the new answer is less than the bound, then the bound is replaced by the new smaller cost.
- If the estimated cost $c(x) < B$, then x is bounded. This is because the cost traversing from the root via x to goal node must be greater than or equal to B. Thus, x cannot lead to better solution than B.

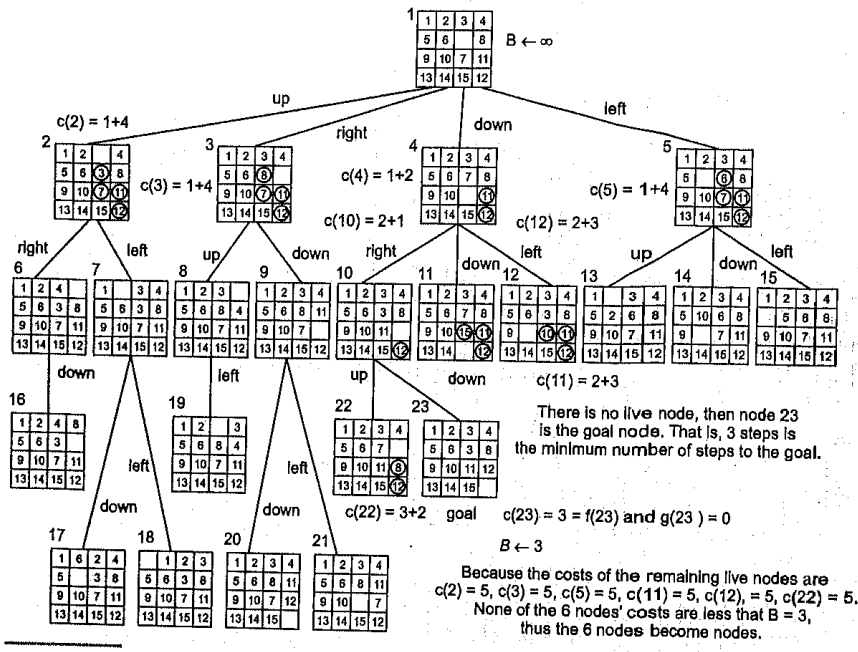


Figure 24.3

24.6 Branch-and-Bound Algorithm for TSP

Now, let us consider how branch and bound can be applied to solve the optimization version of the travelling salesman (TSP) problem. In the optimization version of this problem, we are given a graph G with a cost function $c(e)$ defined for each edge e in G, and we want to find the smallest cost that visits every vertex in G and returning back to its starting vertex.

We can design an algorithm for TSP by computing for each edge $e=(u,v)$, the minimum-cost path that begins at u and ends at v while visiting all other vertices in G along the path. To find such a path, we apply branch and bound technique. We generate the path from u to v in $G-\{e\}$ by augmenting a current path by one vertex in each loop of the branch-and-bound algorithm.

- After we have build a partial path P starting with vertex u, we only consider augmenting P with vertices not in P.
- We can classify a partial path P with "dead end" if the vertices not in P are disconnected in $G-\{e\}$.
- To define lower bound *i.e.*, lb, we can use total cost of the edges in $P+c(e)$.

This will certainly be a lower bound for any tour that will be built from e and p.

After we have run the algorithm for one edge e in G, we can use the best path found so far over all tested edges than restarting the current best solution.

24.7 Knapsack Problem

In knapsack problem, we have to fill the knapsack of capacity W, with a given set of items $I_1, I_2, \ldots I_n$ having weight $w_1, w_2, w_3, \ldots w_n$ in such a manner that the total weight of the items should not exceed the knapsack capacity and the maximum possible value can be obtained.

By using Branch and Bound technique, we have a bound that none of the items can have total sum more than the capacity of knapsack and must give the maximum possible value. The implicit tree for this problem is constructed as a binary tree, where left branch signifies the inclusion of the item and the right branch signifies exclusion. The node structure is as follows it contains three parts, the first part indicates the total weight of the item, the second part indicates the value of the current item and the third part indicates the upper bound for the node *i.e.*,

wt.	value
ub	
node	

The upper bound ub of the node can be computed as

$$ub = v + (W - w)(v_{i+1} / w_{i+1})$$

where v is the value of the current node.

W is the total weight *i.e.*, knapsack capacity

w is the wt. of current node.

Example. *Consider three items along with their respective weights and values as*

$$I = \langle I_1, I_2, I_3 \rangle; \quad w = \langle 5,4,3 \rangle; \quad v = \langle 6,5,4 \rangle$$

the knapsack has the maximum capacity W =7, we have to pack this knapsack using the branch and bound technique so as to give the maximum possible value while considering all constraints.

Solution.

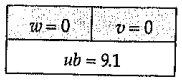
Items	w_i	v_i	v_i / w_i
I_1	5	6	1.2
I_2	4	5	1.25
I_3	3	4	1.3

i.e., calculate value per weight ratio. Now, we arrange the table according to its descending values.

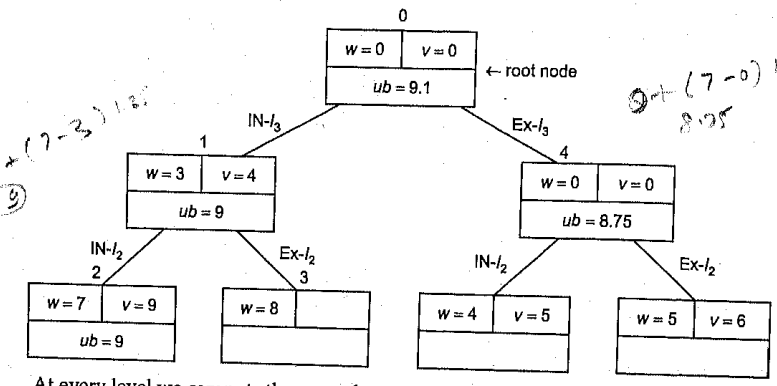
Items	w_i	v_i	v_i / w_i
I_3	3	4	1.3
I_2	4	5	1.25
I_1	5	6	1.2

Now, we start with root node, the upper bound for the root node can be computed as

$$ub = 0 + (7-0) * 1.3 \qquad (v=0,\ w=0,\ v_1/w_1 = 1.3)$$
$$= 9.1$$



Next, we include item I_3, which is indicated by the left branch and exclude item I_3 which is indicated by right branch.



At every level we compute the upper bound and explore the node while selecting the item. Finally the node with maximum upper bound is selected as an optimum solution.

In this example node with I_3 and I_2 gives the optimum solution *i.e.*, maximum value is 9.

24.8 Assignment Problem

In assignment problem 'n' people are assigned 'n' tasks. We have to find the minimum total cost of assignment where each person has exactly one task to perform. For solving this problem we use n by n cost matrix.

Given cost matrix C, where C_{ij} refers to the minimum cost for person 'P_i' to perform task 't_j', $1 \le i \le n$ and $1 \le j \le n$, then the problem is to assign person to tasks so as to minimize the total cost of performing the 'n' tasks *i.e.*, choose the element from the matrix in such a manner that no two elements are in the same column and total cost should be minimum.

Example. *Given a cost matrix for three persons P_1, P_2, P_3 which are assigned following tasks.*

$$C = \begin{array}{c|ccc} & T_1 & T_2 & T_3 \\ \hline P_1 & 6 & 9 & 5 \\ P_2 & 4 & 8 & 3 \\ P_3 & 5 & 11 & 6 \end{array}$$

Obtain the minimum total cost of assignment using Branch and Bound technique where each person has exactly one task to perform.

Solution. It is noticeable that the cost of any solutions, including the optimal one, cannot be smaller than the sum of the minimum elements present in each row of the cost matrix.

In this example the sum of minimum elements present in each row is $5+3+5=13$. This sum becomes the lower bound for our example.

In branch and bound, the node currently having the best possible solution is selected and explored. We select the node for exploring by comparing the lower bounds. The node with the minimum lower bound is selected and explored.

We start from the root which signifies that no assignment is yet done. The node structure contains two parts. In first part the information about the assignments is stored and in the second part lower bound stores.

For instance,

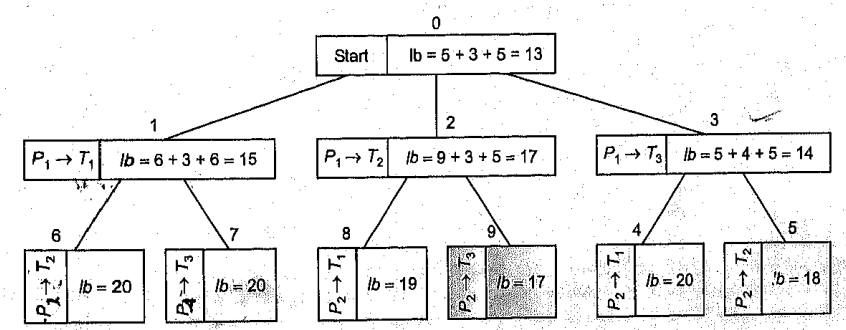
$P_1 \to T_3$	$lb = 14$

$$lb = 5+4+5 = 14$$

In the above node, P_1 is assigned task T_3 and possible lower bound is 14 *i.e.*, $5+4+5$.

The first level of the tree corresponds to 3 tasks. Let us assume that these tasks are assigned to person P_1. Then, we calculate the possible lower bound and select the node having minimum lower bound.

Thus, we select the node where task T_2 is assigned to P_1



Implicit graph for the instance of assignment problem.

Now, we explore this node and can see that P_2 can be assigned T_1 or T_2. It can be seen that lower bounds of these nodes exceed the lower bounds of other nodes. So we explore it and calculate the optimal lower bound. After examining each node, we obtain node labeled '9' having the optimal solution *i.e.*,

P_1 is assigned T_2, P_2 is assigned T_3 and P_3 is assigned T_1.

Thus the total cost $= 9+3+5 = 17$.

Exercise

1. Use Branch and Bound to solve the assignment problems with the following cost matrices :

	1	2	3	4
a	94	1	54	68
b	74	10	88	82
c	62	88	8	76
d	11	74	81	21

2. Devise algorithms for the following using Branch and Bound techniques :
   (*a*) Assignment Problem
   (*b*) Knapsack Problem
   (*c*) Travelling Sales person problem.
3. Solve travelling sales person problem by using Branch and Bound technique.
4. Solve the following instance of the knapsack problem by using Branch and Bound technique.

Items	*w*	*v*
I_1	9	15
I_2	6	6
I_3	7	5
I_4	2	1

CHAPTER

25

Amortized Analysis

25.1 Introduction

An ***amortized analysis*** is any strategy for analyzing a sequence of operations to show that the average cost per operation is small, even though a single operation within the sequence might be expensive. It refers to finding the average running time per operation over a worst-case sequence of operations

This is not an algorithmic technique. It's just a more *accurate* technique for analyzing algorithms *under some conditions.*

Amortized analysis differs from average-case performance in that probability is not involved ; amortized analysis guarantees the time per operation over worst-case performance. The basic idea is that a worst case operation can alter the state in such a way that the worst case cannot occur again for a long time, thus "amortizing" its cost.

Notice that average-case analysis and probabilistic analysis are not the same thing as amortized analysis. In average-case analysis, we are averaging over all possible inputs ; in probabilistic analysis, we are averaging over all possible random choices ; in amortized analysis, we are averaging over a sequence of operations. Amortized analysis assumes worst-case input and typically does not allow random choices.

There are several techniques used in amortized analysis :

- **Aggregate analysis** determines the upper bound $T(n)$ on the total cost of a sequence of n operations, then calculates the average cost to be $T(n)/n$.
- **Accounting method** determines the individual cost of each operation, combining its immediate execution time and its influence on the running time of future operations. Usually, many short-running operations accumulate a "debt" of unfavorable state in small increments, while rare long-running operations decrease it drastically.
- **Potential method** is like the accounting method, but overcharges operations early to compensate for undercharges later.

25.2 Aggregate Analysis

The aggregate method, though simple, lacks the exactness of the other two methods. In particular, the accounting and potential methods allow a specific ***amortized cost*** to be allocated to each operation.

Example : *Stack operations with MULTIPOP*

The push operation inserts an element to the top of the stack and POP operation deletes an element from the top of the stack. Here, we are augmenting the stack data structure with a new operation "MULTIPOP"

```
MULTIPOP (S, k)
    While not Stack empty (S) and k ≠ 0
        do pop (S)
        k ← k − 1
```

Here a stack S and number k, let's add the operation MULTIPOP which removes the top k objects on the stack. Multipop just calls Pop either k times or until the stack is empty.

Example. *What is the running time of Multipop(S, k) on a stack of x objects ?*

Solution. The cost is $\min(s, k)$ pop operations.

If there are n stack operations, in the worst case, a single Multipop can take $O(n)$ time. Let's analyze a sequence of n push, pop, and multipop operations on an initially empty stack. The worst case cost of a multipop operation is $O(n)$ since the stack size is at most n, so the worst case time for any operation is $O(n)$. Hence a sequence of n operations costs $O(n^2)$.

This analysis is technically correct, but overly pessimistic while some of the multipop operations can take $O(n)$ time, not all of them. We need some way to average over the entire sequence of n operations. In fact, the total cost of n operations on an initially empty stack is $O(n)$. Why ? Because each object can be popped at most once for each time that it is pushed. Hence the number of times POP (including calls within multipop) can be called on a nonempty stack is at most the number of push operations which is $O(n)$.

In aggregate analysis, for any value of n, any sequence of n Push, Pop and Multipop operations on an initially empty stack takes $O(n)$ time. The average cost of an operation is thus $O(n)/n = O(1)$. Thus all stack operations have an amortized cost of $O(1)$.

25.3 Accounting Method or Taxation Method

In this we assign different charge to different operations. The operations may be charged more or less than their actual cost. The charging amount of the operation is referred as its *amortized cost*. When amortized cost > actual cost the difference is saved in specific objects as '*Credits*'. These credits are used when the operation charge is less than the actual cost. The total credit store in the data structure is not allowed to be negative.

The sum of the amortized costs for any sequence of operations must be upper bound for the actual total cost of these operations.

If we denote the actual cost of the ith operation by c_i and its amortized cost by $\hat{c}_i$

$$\sum_{i=1}^{n} c_i \le \sum_{i=1}^{n} \hat{c}_i$$

for all sequence of n operation.

Example : *Stack Operations*

The actual costs of the operations are :

Push : 1
Pop : 1
Multipop : $\min(k, S)$ where k is the argument supplied to multipop and 'S' is the stack size when it is called.

Let us assign the following amortized costs :

Push \$2
Pop \$0
Multipop \$0

Note that the amortized cost of Multipop is a constant (0), whereas the actual cost is variable.

Then each time we do a push, we spend one dollar of the tax to pay for the push and then save the other dollar of the tax to pay for the inevitable pop or multipop of that item. Note that if we do n operations, the total amount of taxes we collect is then $2n$.

When we do a push, we use 1 dollar of the tax to pay for the push and then store the extra dollar with the item that was just pushed on the stack. Then all items on the stack will have one dollar stored with them. Whenever we do a pop, we can use the dollar stored with the item popped to pay for the cost of that pop. Morever, whenever we do a multipop, for each item that we pop off in the multipop, we can use the dollar stored with that item to pay for the cost of popping that item.

We've shown that we can use the 2 tax on each item pushed to pay for the cost of all pops, pushes and multipops. Morever we know that this taxation scheme collects at most $2n$ dollars in taxes over n stack operation. Hence we've shown that the amortized cost per operation is $O(1)$.

Thus, for any sequence of n operations of push, pop and multipop the total amortized cost is an upper bound on the total actual cost. Since total amortized cost is $O(n)$, so it is the total actual cost.

25.4 Potential Method

Instead of representing prepaid work, as credit stored with specific objects in the data structure, the potential method of amortized analysis represent the prepaid work as *'potential energy'* or *'potential'*, that can be released to pay for future operations. The stored potential is associated with the entire data structure rather than specific objects within the data structure.

Notations

- D_0 is the initial data structure on which n operations are performed
- D_i is the data structure after the ith operation.
- c_i is the actual cost of the ith operation.
- Φ is a potential function maps each D_i value to its potential value $\Phi(D_i)$ such that $\Phi(D_0)=0$ and $\Phi(D_i)\geq 0$ for all i.

The **amortized cost** of the ith operation with respect to Φ is defined as

$$\hat{c}_i = c_i + \Phi(D_i) - \Phi(D_{i-1})$$

$$\hat{c}_i = c_i + \underbrace{\Phi(D_i) - \Phi(D_{i-1})}_{\textit{potential difference } \Delta\Phi_i}$$

The total amortized cost for n operation is

$$\sum_{i=1}^{n} \hat{c}_i = \sum_{i=1}^{n} (c_i + \Phi(D_i) - \Phi(D_{i-1}))$$

Summing both sides

$$\sum_{i=1}^{n} \hat{c}_i = \sum_{i=1}^{n} (c_i + \Phi(D_i) - \Phi(D_{i-1}))$$

$$= \sum_{i=1}^{n} c_i + \Phi(D_n) - \Phi(D_0)$$

$$\geq \sum_{i=1}^{n} c_i \text{ since } \Phi(D_n)\geq 0 \text{ and } \Phi(D_0)=0$$

Example. *"If the set of stack operations included a MULTIPUSH operation, which pushes k items onto the stack, would the O(1) bound on the amortized cost of stack operations continue to hold ?"*

Solution. No, because one could perform n numbers of MULTIPUSH-k element-operations, costing $O(k)$ per operation, which totals $O(nk)$.

Example. *A sequence of n operations is performed on a data structure. The ith operation costs i if i is an exact power of 2 and 1 otherwise, use aggregate analysis to determine the amortized cost per operation.*

Solution. The ith operation costs i if $i=2^k$, otherwise 1.

Operation	*Real Cost*
1	1
2	2
3	1
4	4
5	1
6	1
7	1
8	8
9	1
⋮	⋮

$$T(n) = 1+2+1+4+1+1+1+8+1+1+\ldots$$

$$= n + \sum_{i=0}^{\lfloor \lg n \rfloor} 2^i - \lfloor \lg n \rfloor$$

$$= n + 2^{\lfloor \lg n \rfloor + 1} - 1 - \lfloor \lg n \rfloor$$

$$< n + 2n \ldots$$

$$< 3n.$$

$$\text{The amortized cost per operation} = \frac{\text{total cost}}{\text{number of operations}}$$

$$< \frac{3n}{n} = 3$$

We can therefore claim that each operation is $O(1)$.

Exercise

1. A sequence of stack operations is performed on stack whose size never exceeds k. After every k operations, a copy of the entire stack is made for back up purpose. Show the cost of n stack operations, including copying the stack is $\theta(n)$ by assigning suitable amortized cost to various stack operations.
2. Explain the amortized cost analysis in splaying ?
3. Find the amortized cost of 8-bit Binary counter using potential method.
4. Obtain the amortized cost using accounting method and potential method for
   (a) Depth first search.
   (b) Breadth first search.
   (c) Minimum spanning tree.

CHAPTER

26

Elementary Graphs Algorithms

26.1 Introduction

Many problems are naturally formulated in terms of objects and the connections between them. *For example,*

- *Airline Routes.* What is the fastest (or cheapest) way to get from one city to another ?
- *Electrical circuits.* Circuit elements are wired together. How does the current flow ?
- *Job Scheduling.* The objects are tasks that need to be performed. The connections indicate which jobs should be done before which other jobs.

A **graph** is a mathematical object that describes such situations. A **graph** is a collection of vertices, V, and edges, E. An edge connects *two* vertices.

A **simple path** is a path with no vertex repeated. A **simple cycle** is a simple path except the first and last vertex is repeated and, for an undirected graph, number of vertices >=3. A **tree** is a graph with no cycles. A **complete graph** is a graph in which all edges are present. A **sparse graph** is a graph with relatively few edges. A **dense graph** is a graph with many edges. An **undirected**

graph has no specific direction between the vertices. A **directed graph** has edges that are "one way". We can go from one vertex to another, but not vice versa. A **weighted graph** has weights associated with each edge. The weights can represent distances, costs, etc.

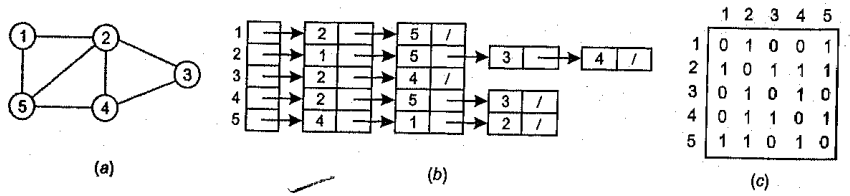
26.2 How is a Graph Represented ?

26.2.1 Representing Graphs as an Adjacency List

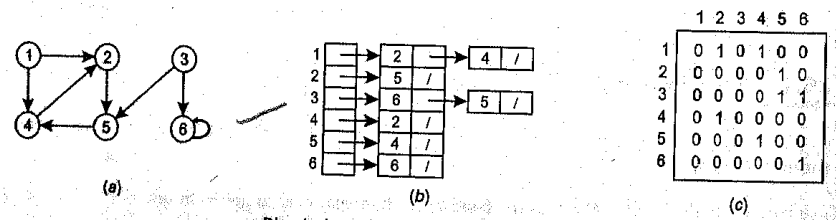
An **adjacency list** is an array, which contains a list for each vertex. The list for a given vertex contains all the other vertices that are connected to the first vertex by a single edge.

26.2.2 Representing Graphs as an Adjacency Matrix

An **adjacency matrix** is a matrix with a row and column for each vertex. The matrix entry is 1 if there is an edge between the row vertex and the column vertex. The diagonal may be zero or one. Each edge is represented twice.



Undirected graph : Matrix is symmetric



Directed graph : Matrix is asymmetric

Figure 26.1

26.2.3 Lists vs. Matrices

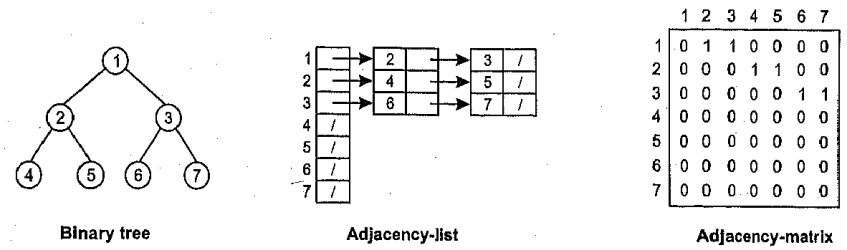
Speed. An adjacency matrix provides the fastest way to determine whether or not a particular edge is present in the graph. For an adjacency list, there is no faster way than to search the entire list for the presence of the edge.

Memory. Normally, an adjacency matrix requires more space (n^2 entries). However, if n is small and the graph is unweighted, an adjacency matrix only needs 1 bit per entry. The adjacency list requires at least 1 word per entry (for the address of the next node of the list). This may offset the advantage of fewer entries.

Example. *Give an adjacency-list representation for a complete binary tree on 7 vertices. Give an equivalent adjacency-matrix representation. Assume that vertices are numbered from 1 to 7 as in a binary heap.*

Solution.

Adjancency Matrix



Example. *Given an adjacency-list representation of a directed graph, how long does it take to compute the out-degree of every vertex ? How long does it take to compute the in-degree ?*

Solution. Given the adjacency-list representation of a graph the out and in-degree of every node can easily be computed in $O(E+V)$ time.

Example. *The transpose of a directed graph $G^T=(V,E^T)$, where $E^T=\{(v,u)\in V\,X\,V : (u,v)\in E\}$. Thus, G^T is G with all its edges reversed. Describe efficient algorithm for computing G^T from G, for both the adjacency-list and adjacency-matrix representations of G. Analyze the running times of your algorithms.*

Solution. The transpose of a directed graph G^T can be computed as follows : If the graph is represented by an adjacency-matrix A simply compute the transpose of the matrix A^T in time $O(V^2)$. If the graph is represented by an adjacency-list a single scan through this list is sufficient to construct the transpose. The time used is the $O(E+V)$.

Example. *The square of a directed graph $G=(V,E)$ is the graph $G^2=(V,E^2)$ such that $(u,w)\in E^2$ if and only if for some $v\in V$, both $(u,v)\in E$ and $(v,w)\in E$. That is, G^2 contains an edge between u and w whenever G contains a path with exactly two edges between u and w. Describe efficient algorithms for computing G^2 from G for both the adjacency-list and adjacency-matrix representations of G. Analyze the running times of your algorithms.*

Solution. The square of a graph can be computed as follows: If the graph is represented as an adjacency matrix A simply compute the product A^2 where multiplication and addition is exchanged by and 'ing and or 'ing. Using the trivial algorithm yields a running time of $O(n^3)$. Using Strassen algorithm improves the bound to $O(n^{\lg 7})=O(n^{2.81})$. If we are using the adjacency-list representation we can simply append lists eliminating duplicates. Assuming the lists are sorted we can proceed as follows : For each node v in some list replace the v with $Adj[v]$ and merge this into the list. Each list can be at most V long and therefore each merge operation takes at most $O(V)$ time. Thus the total running time is $O((E+V)V)=O(E+V^2)$.

```
for i = 1 to V
    for j = 1 to V
    {
        G²[i][j] = 0
        for k = 1 to V
            if (g[i][k] == 1 && g[k][j] == 1)
                {
                    G²[i][j] == 1 ;
                    break ;
                }
    }
```

For many applications, one must systematically search through a graph to determine the structure of the graph.

Two common elementary algorithms for tree searching are :

- **Breadth-first search (BFS)**, and
- **Depth-first search (DFS).**

Both of these algorithms work on directed or undirected graphs. Many advanced graph algorithms are based on the ideas of BFS or DFS. Each of these algorithms traverses edges in the graph, discovering new vertices as it proceeds. The difference is in the order in which each algorithm discovers the edge

26.3 Breadth First Search

Breadth first search trees have a nice property: Every edge of G can be classified into one of three groups. Some edges are in T themselves. Some connect two vertices at the same level of T. And the remaining ones connect two vertices on two adjacent levels. It is not possible for an edge to skip a level.

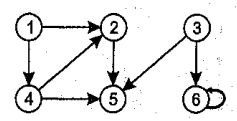
Therefore, the breadth first search tree really is a shortest path tree starting from its root. Every vertex has a path to the root, with path length equal to its level (just follow the tree itself), and no path can skip a level so this really is a shortest path

Analysis

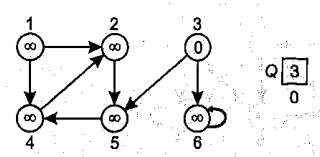
We use aggregate analysis. After initialization, no vertex is ever whitened, and thus the test in line 13 ensures that each vertex is enqueued at most once, and hence dequeued at most once. The operations of enqueuing and dequeuing take $O(1)$ time, so the total time devoted to queue operations is $O(V)$. Because the adjacency list of each vertex is scanned only when the vertex is dequeued, each adjacency list is scanned at most once. Since the sum of the lengths of all the adjacency list is $\Theta(E)$, the total time spent in scanning adjacency list is $O(E)$. The overhead for initialization is $O(V)$, and thus total running time of BFS is $O(V+E)$. Thus, BFS runs in linear time in the size of the adjacency-list representation of G.


```
BFS (G, s)
1.  for each vertex u ∈ V[G] − {s}
2.      do color[u] ← WHITE
3.          d[u] ← ∞
4.          π[u] ← NIL
5.  color[s] ← GRAY
6.  d[s] ← 0
7.  π[s] ← NIL
8.  Q ← ϕ
9.  ENQUEUE(Q, s)
10. while Q ≠ ϕ
11.     do u ← DEQUEUE(Q)
12.         for each v ∈ Adj[u]
13.             do if color[v] = WHITE
14.                 then color[v] ← GRAY
15.                     d[v] ← d[u] + 1
16.                     π[v] ← u
17.                     ENQUEUE(Q, v)
18.             color[u] ← BLACK
```

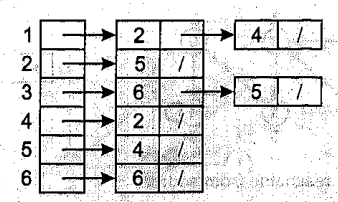
Example. *Consider the graph G in Fig. Describe the whole process of breadth first search. Using vertex 3 as the source.*



Solution.



First, we create adjacency-list representation.



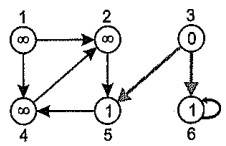
So, adj[4] = {6, 5} so color [6] = GRAY

color [5] = GRAY

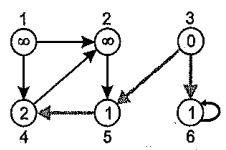
and $d[6] = 1$ $d[5] = 1$

$\pi[6] \leftarrow 3$ $\pi[5] \leftarrow 3$

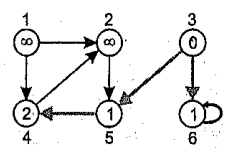
Now



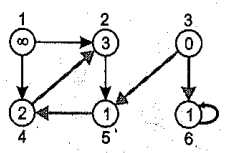
Q | 5 | 6
1 1



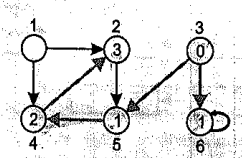
Q | 6 | 4
1 2



Q | 4 |
2



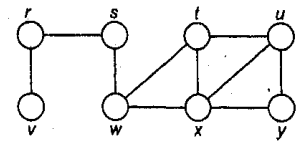
Q | 2 |
3



Q | ϕ |

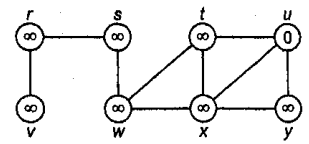
Thus vertex 1 cannot be reachable from source.

Example. *Consider the graph G shown in the Fig. Describe the whole process of BFS.*

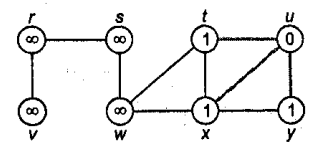


Using vertex u as the source.

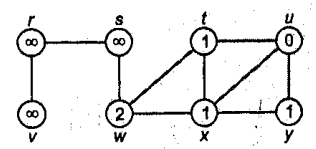
Solution.



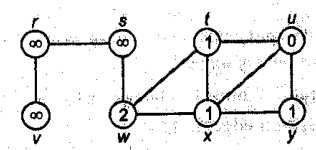
Q | u |
0



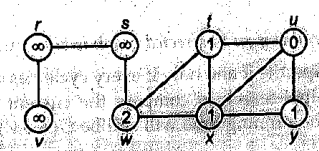
Q | t | x | y |
1 1 1



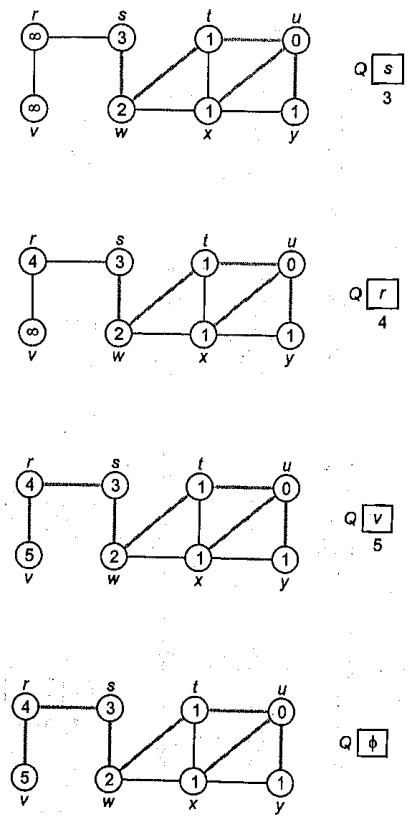
Q | x | y | w |
1 1 2



Q | y | w |
1 2



Q | w |
2



Example. *What is the running time of BFS if its input graph is represented by an adjacency matrix and the algorithm is modified to handle this form of input ?*

Solution. The line "for each $v \in$ Adj[u] ... " must be changed to

"for each $v \in V[G]$ do if edge[u, v]..."

and so the time used scanning for neighbor can increase to $O(V^2)$, yielding a total running time of $O(V^2 + V)$.

Example. *Give an efficient algorithm to determine if an undirected graph is bipartite.*

Solution. we know that a graph is bipartite if and only if every cycle has even length. Using this fact we can simply modify breadth-first-search to compare the current distance with the distance of an encountered gray vertex. The running time will still be $O(E+V)$.

26.4 Depth First Search

In depth-first search, edges are explored out of the most recently discovered vertex v that still has unexplored edges leaving it. When all of v's edges have been explored, the search backtracks to explore edges leaving the vertex from which v was discovered. This process continues until we have discovered all the vertices that are reachable from the original source vertex. If any undiscovered vertices remain, then one of them is selected as a new source and the search is repeated from that source. This entire process is repeated until all vertices are discovered.

As in the BFS, whenever a vertex v is discovered during a scan of the adjacency list of an already discovered vertex u, DFS records this by setting v's predecessor field $\pi[v]$ to u. In DFS predecessor subgraph is composed of several trees, because the search may be repeated from multiple sources.

In DFS, each vertex v has two timestamps: the first timestamp $d[v]$ *i.e.* discovery time records when v is first discovered *i.e.* grayed, and the second timestamp $f[v]$ *i.e.* finishing time records when the search finishes examining v's adjacency list *i.e.* blacked. For every vertex $d[u] < f[u]$.

```
DFS(G)
1.  for each vertex u ∈ V[G]
2.      do color[u] ← WHITE
3.          π[u] ← NIL
4.  time ← 0
5.  for each vertex u ∈ V[G]
6.      do if color[u] = WHITE
7.          then DFS-VISIT(u)
```

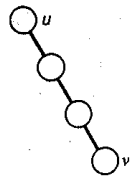
```
DFS-VISIT(u)
1.  color[u] ← GRAY                ▷ White vertex u has just been discovered.
2.  time ← time + 1
3.  d[u] ← time
4.  for each v ∈ Adj[u]            ▷ Explore edge (u,v)
5.      do if color[v] = WHITE
6.          then π[v] ← u
7.              DFS-VISIT (v)
8.  color[u] ← BLACK               ▷ Blacken u, it is finished.
9.  f[u] ← time ← time + 1
```

Analysis

In this we use aggregate analysis. The loops on lines 1-3 and lines 5-7 of DFS take time $\theta(V)$, exclusive the calls to DFS-VISIT. The DFS-Visit calls exactly once for each vertex $v \in V$. During an execution of DFS-VISIT, the loop on lines 4-7 is executed $|\text{Adj}[v]|$ times. The total cost of executing lines 4-7 of DFS-VISIT is $\theta(E)$. The running time of DFS is therefore $\theta(V+E)$.

An important property is that :

The **white path property** states that v is a descendant of u if and only if, at the time of discovery of u, there is a white path from u to v.



If v is a descendant of u, then at time $d[u]$, there must be at least one white path from u to v for the DFS algorithm to follow. Conversely, if there is a white path from u to v at time $d[u]$, then the last node on that path is a neighbor of v which has not yet been processed.

One useful aspect of the DFS algorithm is that it traverses connected components one at a time, and thus it can be used to identify the connected components in a given graph. As an example, if a handwriting sample is scanned into a pixelized image and DFS is performed on that image, all connected components below a certain threshold size (which represent unwanted dirt in the sample) can be removed.

Parenthesis Theorem

In any DFS of a graph $G=(V, E)$, for any *two* vertices u and v exactly one of the following holds :

- The intervals $[d[u], f[u]]$ and $[d[v], f[v]]$ are entirely disjoint, and u nor v is a descendant of the other in the depth-first forest,
- The interval $[d[u], f[u]]$ is contained entirely within the interval $[d[v], f[v]]$, and u is a descendant of v in a depth-first tree, or
- The interval $[d[v], f[v]]$ is contained entirely within the interval $[d[u], f[u]]$, and v is a descendant of u in a depth-first tree.

Classification of Edges

DFS can be used to classify the edges of the input graph $G=(V, E)$. We can define *four* edge types in terms of the depth-first forest.

- **Tree edges.** edges in the depth-first forest.
- **Back edges.** (u,v) which v is an ancestor of u in a depth-first forest. Self-loop are also considered as back edges.
- **Forward edges.** nontree edges (u,v) which v is an descendant of u in depth-first forest.
- **Cross edges.** all other edges. They can go between vertices in the same depth-first tree or in different depth-first trees.

Edge (u,v) can be classified by the color of v

WHITE, tree edge
GRAY, back edge
BLACK, forward or cross edge

26.5 BFS and DFS in Directed Graphs

Although we have discussed them for undirected graphs, the same search routines work essentially unmodified for directed graphs. The only difference is that when exploring a vertex v, we only want to look at edges (v,w) going out of v ; we ignore the other edges coming into v.

For directed graphs, too, we can prove nice properties of the BFS and DFS tree that help to classify the edges of the graph. For BFS in directed graphs, each edge of the graph connects two vertices at the same level, goes down exactly one level, or goes up any number of levels. For DFS, each edge either connects an ancestor to a descendant, a descendant to an ancestor, or one node to a node in a previously visited sub tree. It is not possible to get "forward edges" connecting a node to a sub tree visited later than that node. We'll use this property to test if a directed graph is strongly connected (every vertex can reach every other one).

Example. *Make a 3-by-3 chart with row and column labels WHITE, GRAY, and BLACK. In each cell (i, j), indicate whether, at any point during a depth-first search of a directed graph, there can be an edge from a vertex of color i to a vertex of color j. For each possible edge, indicate what edge it can be. Make a second such chart for depth first search of an undirected graph.*

Solution. In the following table we indicate if there can be an edge (i, j) with the specified colors during a depth-first search. If the entry in the table is present we also indicate which type the edge might be: Tree, Forward or Back edge. For the directed case :

(i,j)	WHITE	GRAY	BLACK
WHITE	TBFC	BC	C
GRAY	TF	TFB	TFC
BLACK		B	TFBC

T : Tree edge ; F : Forward edge ; B : Black edge ; C ; Cross edge.

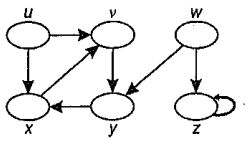
For undirected graph

(i,j)	WHITE	GRAY	BLACK
WHITE	TB	TB	
GRAY	TB	TB	TB
BLACK		TB	TB

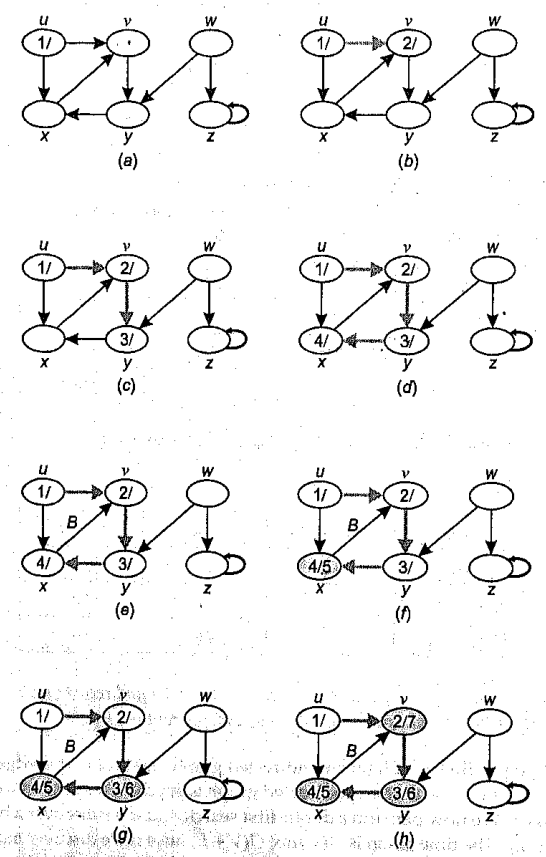
Example. *Give an algorithm that determines whether or not a given undirected graph $G=(V, E)$ contains a cycle, Algorithm should run in $O(V)$ time, independent of $|E|$.*

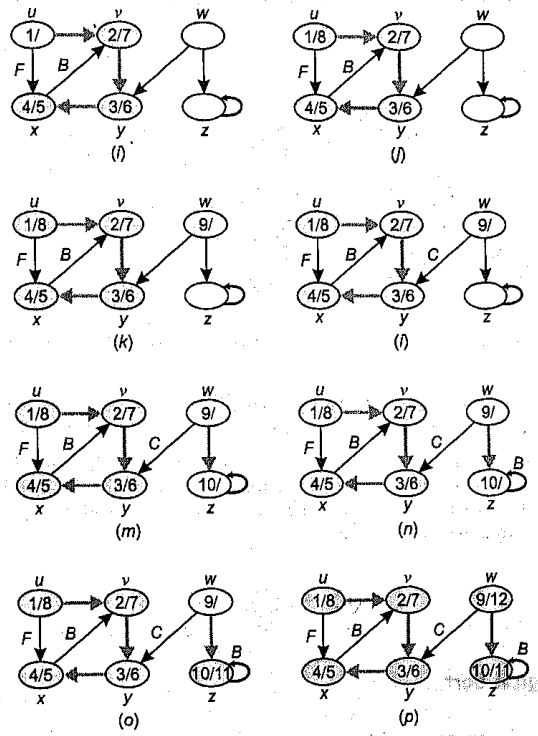
Solution. A depth-first search on an undirected graph yields only tree edges and back edges as shown in the above table. So that an undirected graph is acyclic if and only if a depth-first search yields no back edges. We now perform a depth-first search and if we discover a back edge then the graph must be acyclic. The time taken is $O(V)$ not $O(V+E)$ since if we discover more than V distinct edges the graph must be acyclic and we will have seen a back edge.

Example. *Show the progress of the depth-first-search (DFS) algorithm on a directed graph.*

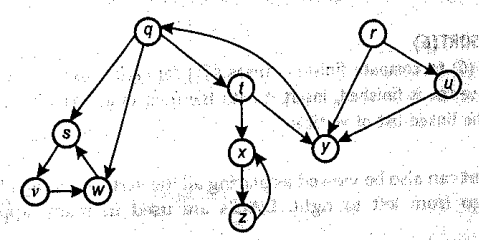


Solution.



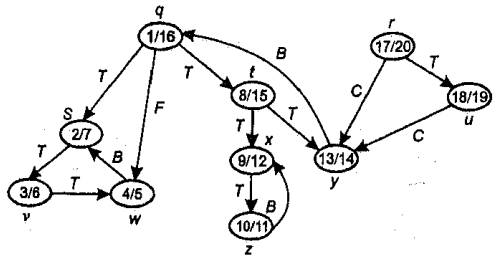


Example. *Show how DFS works on the graph.*



Assume that the for loop of lines 5-7 of the DFS procedure considers the vertices in alphabetical order, and assume that each adjacency list is ordered alphabetically. Show the discovery and finishing times for each vertex, and show the classification of each edge.

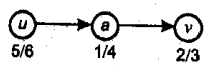
Solution. Shown below are the assignments of $d[\,]$ and $f[\,]$ fields and the classification of edges.



T : Tree edges *B* : Back edges *F* : Forward edges *C* : Cross edges

Example. *Give a counter example to the conjecture that if there is a path from u to v in a directed graph G, and if d [u] < d [v] in a DFS of G, then v is a descendant of u in the depth-first forest produced.*

Solution. The statement is not true since one may get different results depending on which order DFS-VISIT visits the vertices. A counter example is shown in the fig. If DFS- VISIT chooses to visit 'u' first, 'v' will be visited in the next step. Let 's say that DFS-VISIT (u) is called next. Then $d[v] < d[u]$ but u is not a descendant of v.



26.6 Topological Sort

Directed acyclic graphs or DAG's are used for topological sorts. A topological sort of a directed acyclic graph $G=(V, E)$ is a linear ordering of $v \in V$ such that if $(u, v) \in E$ then u appears before v in this ordering. If G is cyclic, no such ordering exists .

TOPOLOGICAL-SORT(G)

1. call DFS (G) to compute finishing times $f[v]$ for each vertex v
2. as each vertex is finished, insert it onto the front of a linked list
3. return the linked list of vertices

Topological sort can also be viewed as placing all the vertices along a horizontal line so that all directed edges go from left to right. DAG's are used in many applications to indicate precedence.

We can perform a topological sort in time $\theta(V + E)$, since DFS takes $\theta(V + E)$ time and it takes $O(1)$ time to insert each of the $|V|$ vertices onto the front of the linked list.

Example. *Another way to perform topological sorting on a directed acyclic graph $G=(V, E)$ is to repeatedly find a vertex of in-degree 0 , output it , and remove it and all of its outgoing edges from the graph. Explain how to implement this idea so that it runs in time $O(V + E)$. What happens to this algorithm if G has cycles ?*

Solution.

```
TOPOLOGICAL-SORT (G)
for each vertex u ∈ V
    do in-degree[u] ← 0
for each vertex u ∈ V
    do for each v ∈ Adj[u]
          do in-degree[v] ← in-degree[v] +1
Q ← ϕ
for each vertex u ∈ V
    do if in-degree[u] = 0
          then ENQUEUE(Q, u)
while Q ≠ ϕ
    do u ← DEQUEUE(Q)
          output u
          for each v ∈ Adj[u]
               do in-degree[v] ← in-degree[v] − 1
                    if in-degree[v] = 0
                    then ENQUEUE(Q, v)
    do if in-degree[u] ≠ 0
          then report that there is a cycle
```

Another way to check the cycles would be to count the vertices that are output and report a cycle if that number is $<|V|$.

To find vertices with in-degree 0 we store these in a table. This takes $\theta(V + E)$ time. ($|V|$) visited adjacency lists, a total of $|E|$ visited edges and $\theta(1)$ time per edge.) We store vertices with in-degree 0 on a stack or in a queue so that they can be retrieved in $\theta(1)$ time.

To initialize the queue takes $\theta(V)$ time. When we have printed a vertex we remove its out-going edges from the graph by determining the in-degree on all its neighbors. Those vertices whose in-degree dropped to 0 are inserted into the queue. The total running time is $O(V + E)$. If the graph contains no cycles, all the vertices will be printed.

26.7 Strongly Connected Components

A directed graph is called **strongly connected** if for every pair of vertices u and v there is a path from u to v and a path from v to u. The **strongly connected components** (SCC) of a directed graph are its maximal strongly connected subgraphs. These form a partition of the graph.

Kosaraju's algorithm efficiently computes the strongly connected components of a directed graph.

Kosaraju's algorithm

Strongly-connected Components (G)

1. call DFS(G) to compute finishing times $f[u]$ for each vertex u.
2. compute G^T
3. call DFS(G^T), but in the main loop of DFS, consider the vertices in order of decreasing $f[u]$
4. produce as output the vertices of each tree in the DFS forest formed in line 3 as a separate SCC.

Analysis

A linear-time algorithm, if the graph is represented as an adjacency list $\theta(V+E)$, as a matrix it is an $O(V^2)$ algorithm, computes the strongly connected components of a directed graph $G=(V, E)$ using two Depth-first searches (DFSs), one on G, and one on G^T, the transpose graph. Equivalently, Breadth-first search (BFS) can be used instead of DFS.

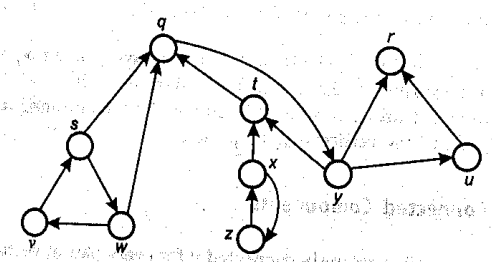
A transpose graph is a graph with all of the edges reversed. It is named because its adjacency matrix is the transpose of the adjacency matrix of the original graph.

Example. *Find the strongly connected components in the Fig. shown in the previous question.*

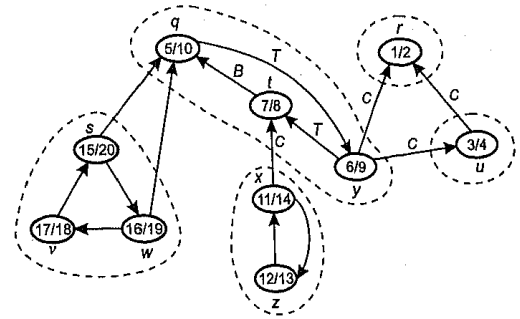
Solution. The first DFS gives the list

r, u, q, t, y, x, z, s, v, w (reverse order of turning black)

Now, G^T as follows.



Call DFS (G^T)



Strongly connected components are $\{s, w, v\}$, $\{q, y, t\}$, $\{x, z\}$, $\{r\}$, $\{u\}$.

Example. *How can the number of strongly connected components of a graph change if a new edge is added ?*

Solution. The number of strongly connected components can be reduced, when the new added edge makes two strongly connected components combined into one. Otherwise, the number will be the same.

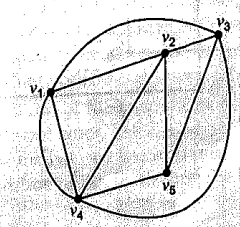
Example. *Prove that for any directed graph G, we have* $((G^T)^{SCC})^T = G^{SSC}$. *That is, the transpose of the component graph of* G^T *is the same as the component graph of G.*

Solution. Since the strongly connected relationship is an equivalence relation, G and G^T will always have the same strongly connected components. If two vertices X and Y are not strongly connected, then there is a unique path from X to Y in G iff there is a unique path from Y to X in G^T.

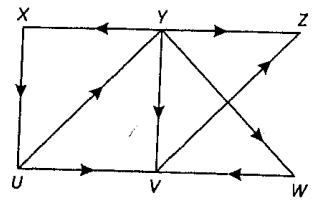
A similar property will also hold for the component graph and transpose of the component graph, *i.e.*, if two vertices X and Y are not strongly connected in G, then here is a unique path from SCC(G,X) to SCC(G,Y) in $(G)^{SCC}$ iff there is a unique path from SCC(G^T,Y) to SCC(G^T, X) in $(G^T)^{SCC}$.

Exercise

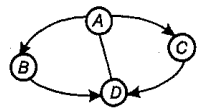
1. For the graph shown below find the following :
   (*a*) adjacency list representation. (*b*) adjacency matrix representation.



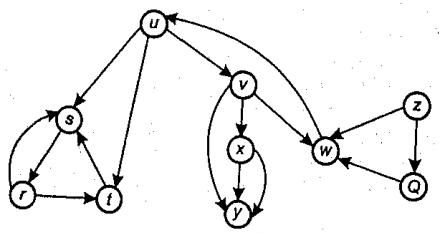
2. Let G be an undirected graph G with n vertices and m edges. Describe an algorithm running in $O(n+m)$ time traverses each edge of G exactly once in each direction.
3. Show that algorithm to find a cycle using BFS or DFS must be $O(n)$.
4. Find the adjacency matrix for the following graph in Fig. Find in-degree and out-degree of all the nodes.



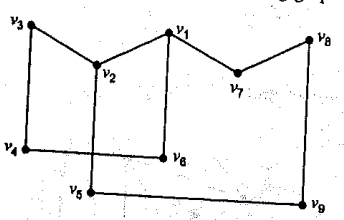
5. Consider the following graph in Fig. State, if this graph is strongly connected. List all the simple paths.



6. Show the parenthesis structure of the following graph.



7. Starting from vertex v_4 apply BFS and DFS in the following graph.



CHAPTER

27

Minimum Spanning Tree

27.1 Spanning Tree

A **spanning tree** of a graph is just a sub graph that contains all the vertices and is a tree. A graph may have many spanning trees. We can also assign a *weight* to each edge, which is a number representing how unfavorable it is, and use this to assign a weight to a spanning tree by computing the sum of the weights of the edges in that spanning tree. A **minimum spanning tree** or **minimum weight spanning tree** is then a spanning tree with weight less than or equal to the weight of every other spanning tree. More generally, any undirected graph has a **minimum spanning forest**.

To explain further upon the Minimum Spanning Tree (MST), and what it applies to, let's consider a couple of real-world examples :

1. One practical application of a MST would be in the design of a network. For instance, a group of individuals, who are separated by varying distances, wish to be connected together in a telephone network. Although MST cannot do anything about the distance from one connection to another, it can be used to determine the least costly paths with no cycles in this network, thereby connecting everyone at a minimum cost.
2. Another useful application of MST would be finding airline routes. The vertices of the graph would represent cities, and the edges would represent routes between the cities. Obviously, the further one has to travel, the more it will cost, so MST can be applied to optimize airline routes by finding the least costly paths with no cycles.

To explain how to find a Minimum Spanning Tree, we will look at two algorithms: the Kruskal algorithm and the Prim algorithm. Both algorithms differ in their methodology, but both eventually end up with the MST. ***Kruskal's algorithm uses edges, and Prim's algorithm uses vertex connections in determining the MST.*** Both algorithms are greedy algorithms that run in polynomial time. At each step of an algorithm, one of several possible choices must be made.

27.2 Kruskal's Algorithm

Kruskal's algorithm is an algorithm that finds a minimum spanning tree for a connected weighted graph. It finds a safe edge to add to the growing forest by finding, of all the edges that connect any two trees in the forest, an edge (u,v) of least weight. This means it finds a subset of the edges that forms a tree that includes every vertex, where the total weight of all the edges in the tree is minimized. **If the graph is not connected, then it finds a minimum spanning forest (a minimum spanning tree for each connected component).**

```
MST-KRUSKAL (G, w)
A ← ϕ
for each vertex v ∈ V[G]
    do MAKE-SET(v)
sort the edges of E into nondecreasing order by weight w
for each edge (u,v) ∈ E, taken in nondecreasing order by weight
    do if FIND-SET(u) ≠ FIND-SET(v)
        then A ← A ∪ {(u,v)}
            UNION (u,v)
return A
```

→ u & v belong to diff ~~forests~~ trees
no creation of cycles.

Analysis

Where E is the number of edges in the graph and V is the number of vertices, Kruskal's algorithm can be shown to run in $O(E \log E)$ time, or equivalently, $O(E \log V)$ time, all with simple data structures. These running times are equivalent because :

- E is at most V^2 and $\log V^2 = 2 \times \log V$ is $O(\log V)$.
- If we ignore isolated vertices, which will each be their own component of the minimum spanning tree anyway, $V \le 2E$, so $\log V$ is $O(\log E)$.

We can achieve this bound as follows: first sort the edges by weight using a comparison sort in $O(E \log E)$ time ; Next, we use a disjoint-set data structure to keep track of which vertices are in which components. We need to perform $O(E)$ operations, two find operations and possibly one union for each edge. Even a simple disjoint-set data structure such as disjoint-set forests with union by rank can perform $O(E)$ operations in $O(E \log V)$ time. Thus the total time is

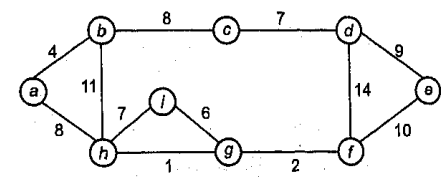
$$O(E \log E) = O(E \log V).$$

Example. *Let (u,v) be a minimum weight edge in a graph G. Show that (u,v) belongs to some minimum spanning tree of G.*

Solution. Proof by contradiction : assume that (u,v) doesn't belong to some MST of G. If we would add (u,v) to the MST we would get a cycle because MST is a tree. But if we then removed another edge in the cycle we would end up with a MST that is at least as cheap as the original.

Hence, (u,v) does belong to some minimum spanning tree of G.

Example. *Find the minimum spanning tree of the following graph using kruskals algorithm.*

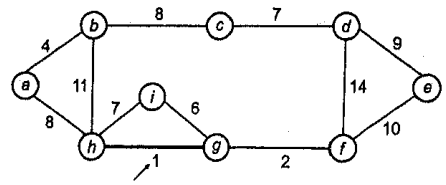


Solution. First we initialize the set A to the empty set and create $|v|$ trees, one containing each vertex with MAKE-SET procedure. Then sort the edges in E into order by non-decreasing weight, *i.e.*,

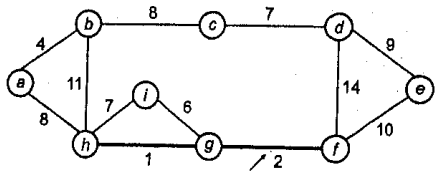
(h,g)	1
(g,f)	2
(a,b)	4
(c,i)	6
(h,i)	7
(c,d)	7
(b,c)	8
(a,h)	8
(d,e)	9
(b,i)	11
(f,e)	10
(d,f)	14

Now, check for each edge (u,v), whether the end points u and v belong to the same tree. If they do, then the edge (u,v) cannot be added. Otherwise, the two vertices belong to different trees and the edge (u,v) is added to A and the vertices in the two trees are merged in by UNION procedure.

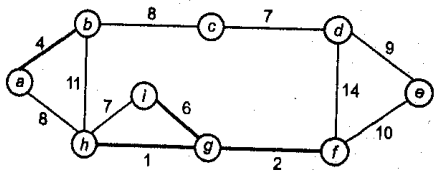
So, first take (h,g) edge



the (g, f) edge.

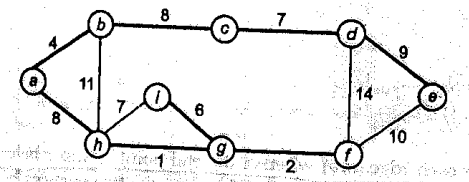


then (a,b) and (i,g) edges are considered and forest becomes.



Now, edge (h,i). Both h and i vertices are in same set, thus it creates a cycle. So this edge is discarded.

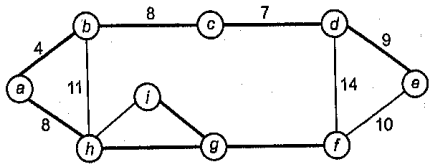
Then edge (c,d), (b,c), (a,h), (d,e), (e,f) are considered and forest becomes.



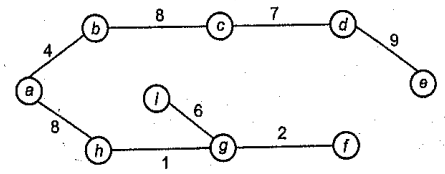
In (e, f) edge both end points e and f exist in same tree so discarded this edge.

Then (b,h) edge, it also creates a cycle.

After that edge (d, f) and the final spanning tree is shown as in dark lines.



or



Minimum cost MST

27.3 Prim's Algorithm

The main idea of Prim's algorithm is similar to that of Dijkstra's algorithm (discussed later) for finding shortest path in a given graph. Prim's algorithm has the property that the edges in the set A always form a single tree. We begin with some vertex v in a given graph $G=(V, E)$, defining the initial set of vertices A. Then, in each iteration, we choose a minimum-weight edge (u,v) connecting a vertex v in the set A to the vertex u outside of set A. Then vertex u is brought in to A. This process is repeated until a spanning tree is formed. Like Kruskal's algorithm, here too, the important fact about MSTs is we always choose the smallest-weight edge joining a vertex inside set A to the one outside the set A. The implication of this fact is that it adds only edges that are safe for A; therefore when the algorithm terminates, the edges in set A form a MST.

```
MST-PRIM(G, w, r)
for each u ∈ V[G]
    do key[u] ← ∞
       π[u] ← NIL
key[r] ← 0
Q ← V[G]
while Q ≠ ϕ
    do u ← EXTRACT-MIN(Q)
       for each v ∈ Adj[u]
           do if v ∈ Q and w(u,v) < key[v]
                 then π[v] ← u
                      key[v] ← w(u,v)
```

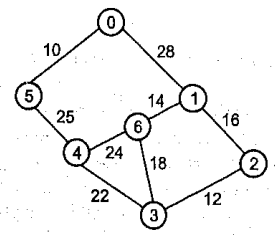
Example. *Suppose that all edge weights in a graph are integers in the range from 1 to $|V|$. How fast can you make Prim's algorithm run ? What if the edge weights are integers in the range from 1 to W for some constant W ?*

Solution. The running time of Prims algorithm is composed :

$O(V)$ initialization.
$O(V)$-time for EXTRACT-MIN.
$O(E)$-time for DECREASE-KEY.

If the edges are in the range 1.....$|V|$ the priority queue can speed up EXTRACT_MIN and DECREASE-KEY to $O(\lg\lg V)$ thus yielding a total running time of $O(V \lg\lg V + E \lg\lg V) = O(E \lg\lg V)$. If the edges are in the range from 1 to W we can implement the queue as an array $[1 \ldots W+1]$ where the ith slot holds a doubly linked list of the edges with weight i. The $W+1$st slot contains ∞. EXTRACT_MIN now runs in $O(W) = O(1)$ time since we can simply scan for the first nonempty slot and return the first element of that list. DECREASE-KEY runs in $O(1)$ time as well since it can be implemented by moving an element from one slot to another.

Example. *Generate minimum cost spanning tree for the following graph using Prim's algorithm.*



Solution. In Prim's algorithm, first we initialize the priority queue Q to contain all the vertices and the key of each vertex to ∞ except for the root, whose key is set to 0. Suppose 0 vertex is the root, *i.e.*, r. By EXTRACT-MIN(Q) produre, now $u = r$ and $\text{Adj}[u] = \{5,1\}$.

Removing u from the set Q and adds it to the set $V - Q$ of vertices in the tree. Now, update the key and π fields of every vertex v adjacent to u but not in the tree

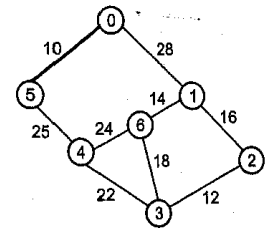
$key\,[5] = \infty$

$w[0,5) = 10$ *i.e.,* $w(u,v) < key\,[5]$

so, $\pi[5] = 0$ and $key\,[5] = 10$.

and $key\,[1] = \infty$

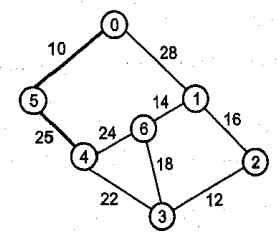
$w(0,1) = 28$ *i.e.,* $w(u,v) < key\,[5]$ so $\pi[1] = 0$ and $key\,[1] = 28$.



Now, by EXTRACT_MIN(Q) Remove 5 because key[5] = 10 which is minimum so $u = 5$.

$Adj\,[5] = \{4\}$

$w(u,v) < key\,[v]$ then $key\,[4] = 25$



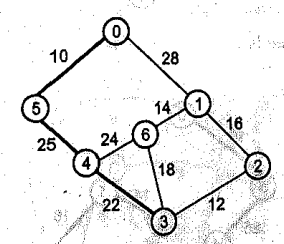
Now remove 4 because $key\,[4] = 25$ which is minimum so $u = 4$.

$Adj\,[4] = \{6, 3\}$

$w(u,v) < key\,[v]$ then $key\,[6] = 24$

$key\,[3] = 22$

Now remove 3 because $key\,[3] = 22$ is minimum so $u = 3$.



$Adj\,[3] = \{4, 6, 2\}$

$4 \neq Q$ $key\,[6] = 24$ now becomes $key\,[6] = 18$.

$key\,[2] = 12$

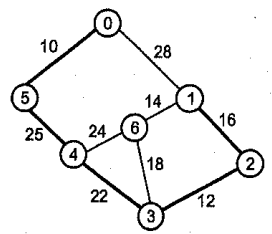
Now, in Q. *key* [2] = 12, *key* [6] = 18, key [1] = 28

By EXTRACT_MIN(Q) Remove 2, because *key* [2] = 12 is minimum.

Adj [2] = {3, 1}

$3 \neq Q$.

Now *key* [1] = 16.

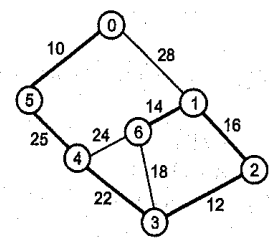


By EXTRACT_MIN(Q) Remove 1 because *key* [1] = 16 is minimum.

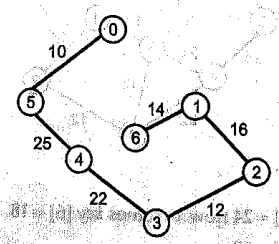
Adj [1] = {0, 6, 2}

$\{0, 2\} \notin Q$. *key*[6] = 14.

Now becomes Q contains only one vertex 6. By EXTRACT_MIN remove it



Thus, the final spanning tree is

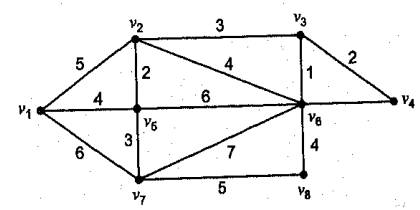


Example. *There are two well-known algorithms for finding minimum spanning trees. Point out the difference between Prim's algorithm and Kruskal's algorithm in terms of the construction of the MST tree.*

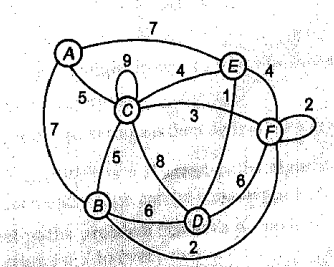
Solution. The construction of an MST tree by Prim's algorithm is started from a single node, and the tree is expanded until it covers all the nodes in the graph. In other words, there is only one partial tree during the construction of the MST tree from the very beginning to the end. However, in the construction of an MST tree using Kruskal's algorithm, the algorithm starts from a *n*-tree forest. Each time it merges two trees in the forest into one, the algorithm continues until there is only one tree left in the forest.

Exercise

1. Show that if all the weights in a connected weighted graph G are distinct, then there is exactly one minimum spanning tree for G.
2. For the graph shown below obtain the following :
   (*a*) MST by Kruskal's Method
   (*b*) MST by Prim's Method.

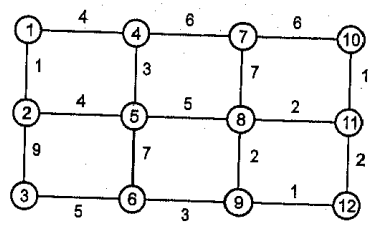


3. Discuss Kruskal and Prim algorithms and write their pseudo codes.
4. Find out the MST of the following graph.



5. Thre is a network given in the Fig. below as a highway map and the number recorded next to each arc as the maximum elevation enountered in traversing the arc.

A traveller plans to drive from node 1 to 12 on this highway. The traveller dislikes high altitudes and so would like to find a path connecting node 1 to 12 that minimizes the maximum altitude. Find the best path for the traveller using a minimum spanning tree.



CHAPTER 28

Single-Source Shortest Paths

28.1 Introduction

In a **shortest-paths problem**, we are given a weighted, directed graph $G=(V, E)$, with weight function $w : E \to \mathbf{R}$ mapping edges to real-valued weights. The weight of path $p=\langle v_0, v_1, \ldots, v_k \rangle$ is the sum of the weights of its constituent edges :

$$w(P)=\sum_{i=1}^{k} w(v_{i-1}, v_i)$$

We define the **shortest-path weight** from u to v by $\delta(u,v)=\min(w(p) : u \rightsquigarrow v)$, if there is a path from u to v, and $\delta(u,v)=\infty$, otherwise.

A **shortest path** from vertex s to vertex t is then defined as any path p with weight $w(p)=\delta(s,t)$.

The *breadth-first-search algorithm* is a shortest-paths algorithm that works on unweighted graphs, that is, graphs in which each edge can be considered to have unit weight.

Single-source shortest-paths problem : Given a graph $G=(V, E)$, we want to find a shortest path from a given source vertex $s \in V$ to every vertex $v \in V$.

Variants

But, the algorithm for the single-source problem, including the following variants, can solve many other problems.

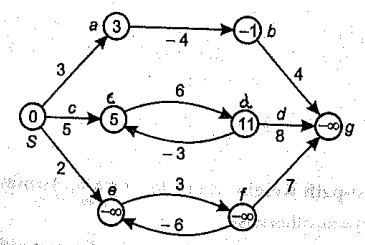
- **Single-destination shortest-paths problem.** Find a shortest path to a given ***destination*** vertex t from every vertex v. By reversing the direction of each edge in the graph, we can reduce this problem to a single-source problem.
- **Single-pair shortest-path problem.** Find a shortest path from u to v for given vertices u and v. If we solve the single-source problem with source vertex u, we solve this problem also. Moreover, no algorithms for this problem are known that run asymptotically faster than the best single-source algorithms in the worst case.
- **All-pairs shortest-paths problem.** Find a shortest path from u to v for every pair of vertices u and v. Running a single-source algorithm once from each vertex can solve this problem ; but it can usually be solved faster, and its structure is of interest in its own right.

28.2 Shortest Path : Existence

If some path from s to v contains a negative cost cycle then, there does not exist a shortest path. Otherwise, there exists a shortest $s-v$ that is simple.

Cost of $C < 0$

If there is a negative-weight cycle on some path from s to v, we define $\delta(s,v) = -\infty$.



There are infinitely many paths from s to c: $\langle s,c\rangle$, $\langle s,c,d,c\rangle$, $\langle s,c,d,c,d,c\rangle$, and so on. Because the cycle $\langle c,d,c\rangle$ has weight $6+(-3)=3>0$, the shortest path from s to c is $\langle s,c\rangle$, with weight $\delta(s,c)=5$. Similarly, there are infinitely many paths from s to e: $\langle s,e\rangle$, $\langle s,e,f,e\rangle$, $\langle s,e,f,e,f,e\rangle$, and so on. Since the cycle $\langle e,f,e\rangle$ has weight $3+(-6)=-3<0$, however, there is no shortest path from s to e. By traversing the negative-weight cycle $\langle e,f,e\rangle$ arbitrarily many times, we can find paths from s to e with arbitrarily large negative weights, and so $\delta(s,e)=-\infty$.

Some shortest-paths algorithms, such as **Dijkstra's algorithm**, assume that all edge weights in the input graph are nonnegative, as in the road-map example. Others, such as the **Bellman-Ford algorithm**, allow negative-weight edges in the input graph and produce a correct answer as long as no negative-weight cycles are reachable from the source. Typically, if there is such a negative-weight cycle, the algorithm can detect and report its existence.

28.3 Representing Shortest Paths

Given a graph $G=(V,E)$, we maintain for each vertex $v \in V$ a ***predecessor*** $\pi[v]$ that is either another vertex or NIL. During the execution of a shortest-paths algorithm, however, the π values need not indicate shortest paths. As in breadth-first search, we shall be interested in the ***predecessor subgraph*** $G_\pi = (V_\pi, E_\pi)$ induced by the values π. Here again, we define the vertex set V_π, to be the set of vertices of G with non-NIL predecessors, plus the source s:

$$\mathbf{V}_\pi = \{v \in V : \pi[v] \neq \text{NIL}\} \cup \{s\}$$

The directed edge set E_π is the set of edges induced by the π values for vertices in $\mathbf{V}_\pi$:

$$\mathbf{E}_\pi = \{(\pi[v], v) \in E : v \in \mathbf{V}_\pi - \{s\}\}$$

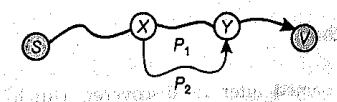
A ***shortest-paths tree*** rooted at s is a directed sub graph $G' = (V', E')$, where $V' \subseteq V$ and $E' \subseteq E$, such that

1. V' is the set of vertices reachable from s in G,
2. G' forms a rooted tree with root s, and
3. for all $v \in V'$, the unique simple path from s to v in G' is a shortest path from s to v in G.

Shortest paths are not necessarily unique, and neither are shortest-paths trees.

28.4 Shortest Path : Properties

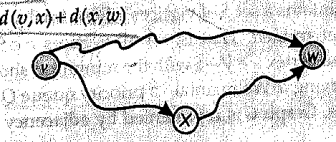
- **Optimal substructure property.** All sub-paths of shortest paths are shortest paths.



Let P_1 be $x-y$ sub-path of shortest $s-v$ path P. Let P_2 be any $x-y$ path. Then cost of $P_1 \leq$ cost of P_2, otherwise P not shortest $s-v$ path.

- **Triangle inequality.** Let $d(v,w)$ be the length of the shortest path from v to w. Then,

$$d(v,w) \leq d(v,x) + d(x,w)$$



Relaxation

The single-source shortest-paths algorithms are based on a technique known as **relaxation**, a method that repeatedly decreases an upper bound on the actual shortest-path weight of each vertex until the upper bound equals the shortest-path weight. For each vertex $v \in V$, we maintain an attribute $d[v]$, which is an upper bound on the weight of a shortest path from source s to v. We call $d[v]$ a **shortest-path estimate**. We initialize the shortest-path estimates and predecessors by the following procedure.

```
INITIALIZE-SINGLE-SOURCE (G, s)
for each vertex v ∈ V[G]
    do d[v] ← ∞
       π[v] ← NIL
d[s] ← 0
```

After initialization, $\pi[v] = \text{NIL}$ for all $v \in V$, $d[v] = 0$ for $v = s$, and $d[v] = \infty$ for $v \in V - \{s\}$.

The **process of relaxing** an edge (u, v) consists of testing whether we can improve the shortest path to v found so far by going through u and, if so, updating $d[v]$ and $\pi[v]$. A relaxation step may decrease the value of the shortest-path estimate $d[v]$ and update v's predecessor field $\pi[v]$. The following code performs a relaxation step on edge (u, v).

```
RELAX (u, v, w)
if d[v] > d[u] + w(u, v)
    then d[v] ← d[u] + w(u, v)
         π[v] ← u
```

In **Dijkstra's algorithm** and the shortest-paths algorithm for directed acyclic graphs, each edge is relaxed exactly once. In the **Bellman-Ford algorithm**, each edge is relaxed several times.

28.5 Dijkstra's Algorithm

Dijkstra's algorithm, named after its discoverer, Dutch computer scientist **Edsger Dijkstra**, is a greedy algorithm that solves the single-source shortest path problem for a directed graph $G = (V, E)$ with nonnegative edge weights *i.e.* we assume that $w(u, v) \geq 0$ for each edge $(u, v) \in E$.

Dijkstra's algorithm maintains a set S of vertices whose final shortest-path weights from the source s have already been determined. That is, for all vertices $v \in S$, we have $d[v] = \delta(s, v)$. The algorithm repeatedly selects the vertex $u \in V - S$ with the minimum shortest-path estimate, inserts u into S, and relaxes all edges leaving u. We maintain a priority queue Q that contains all the vertices in $V - S$, keyed by their d values. Graph G is represented by adjacency lists.


```
DIJKSTRA(G, w, s)
INITIALIZE-SINGLE-SOURCE (G, s)
S ← φ
Q ← V[G]
while Q ≠ φ
    do u ← EXTRACT-MIN(Q)
       S ← S ∪ {u}
       for each vertex v ∈ Adj[u]
           do RELAX (u, v, w)
```

Because Dijkstra's algorithm always chooses the "lightest" or "closest" vertex in $V - S$ to insert into set S, we say that it uses a **greedy strategy**.

Dijkstra's algorithm bears some similarity to both **breadth-first search** and **Prim's algorithm** for computing minimum spanning trees. It is like breadth-first search in that set S corresponds to the set of black vertices in a breadth-first search; just as vertices in S have their final shortest-path weights, so black vertices in a breadth-first search have their correct breadth-first distances. Dijkstra's algorithm is like Prim's algorithm in that both algorithms use a priority queue to find the "lightest" vertex outside a given set (the set S in Dijkstra's algorithm and the tree being grown in Prim's algorithm), insert this vertex into the set, and adjust the weights of the remaining vertices outside the set accordingly.

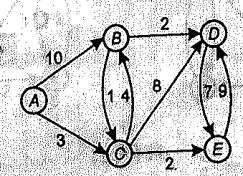
Analysis

The running time of Dijkstra's algorithm on a graph with edges E and vertices V can be expressed as a function of $|E|$ and $|V|$ using the Big-O notation. The simplest implementation of the Dijkstra's algorithm stores vertices of set Q in an ordinary linked list or array, and operation Extract-Min (Q) is simply a linear search through all vertices in Q. In this case, the running time is $O(|V|^2 + |E| = O(V^2)$.

For sparse graphs, that is, graphs with many fewer than $|V|^2$ edges, Dijkstra's algorithm can be implemented more efficiently by storing the graph in the form of adjacency lists and using a binary heap or Fibonacci heap as a priority queue to implement the Extract-Min function. With a binary heap, the algorithm requires $O((|E| + |V|)\log|V|)$ time (which is dominated by $O(|E|\log|V|)$ assuming every vertex is connected), and the Fibonacci heap improves this to $O(|E| + |V|\log|V|)$.

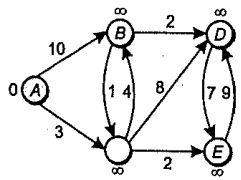
A C E B D

Example :



A	B	C	D	E
0	∞	∞	∞	∞
	10	3	∞	∞
	7	3	11	5
	7		3	11	5
	7	3		5
	7	3		5

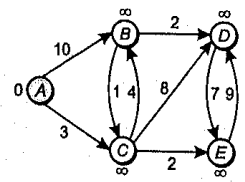
Initialize :



Q:	A	B	C	D	E
	0	∞	∞	∞	∞

$S : \{\}$

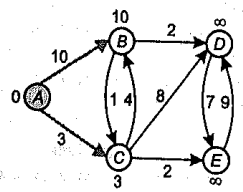
"A" ← EXTRACT-MIN(Q) :



Q:	A	B	C	D	E
	0	∞	∞	∞	∞

$S : \{A\}$

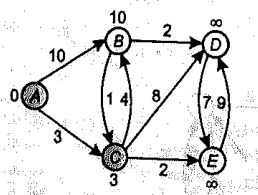
Relax all edges leaving A:



Q:	A	B	C	D	E
	0	∞	∞	∞	∞
		10	3	–	–

$S : \{A\}$

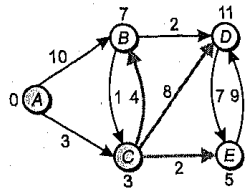
"C" ← EXTRACT-MIN(Q) :



Q:	A	B	C	D	E
	0	∞	∞	∞	∞
		10	3	–	–

$S : \{A, C\}$

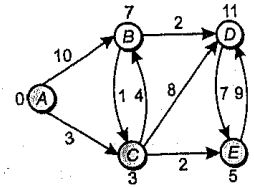
Relax all edges leaving C:



Q:	A	B	C	D	E
	0	∞	∞	∞	∞
		10	3	–	–
		7		11	5

$S : \{A, C\}$

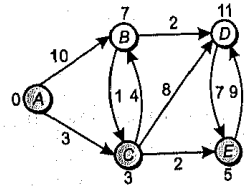
"E" ← EXTRACT-MIN(Q) :



Q:	A	B	C	D	E
	0	∞	∞	∞	∞
		10	3	–	–
		7		11	5

$S : \{A, C, E\}$

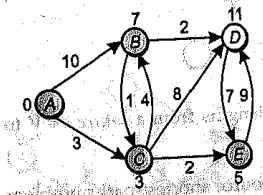
Relax all edges leaving E:



Q:	A	B	C	D	E
	0	∞	∞	∞	∞
		10	3	∞	∞
		7		11	5
		7		11	

$S : \{A, C, E\}$

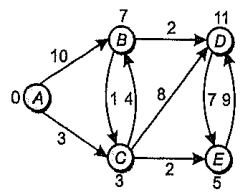
"B" ← EXTRACT-MIN(Q) :



Q:	A	B	C	D	E
	0	∞	∞	∞	∞
		10	3	∞	∞
		7		11	5
		7		11	

$S : \{A, C, E, B\}$

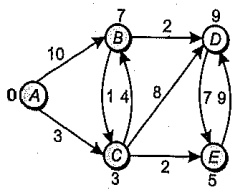
Relax all edges leaving B :



Q :	A	B	C	D	E
	0	∞	∞	∞	∞
		10	3	∞	∞
		7		11	5
		7		11	
				9	

$S : \{A, C, E, B\}$

"D" ← EXTRACT-MIN(Q) :



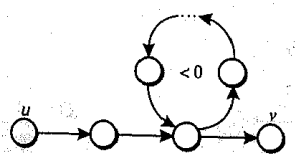
Q :	A	B	C	D	E
	0	∞	∞	∞	∞
		10	3	∞	∞
		7		11	5
		7		11	
				9	

$S : \{A, C, E, B, D\}$

28.6 The Bellman-Ford Algorithm

If a graph $G=(V, E)$ contains a negative weight cycle, then some shortest paths may not exist.

Example :



Bellman-Ford algorithm finds all shortest-path lengths from a *source* $s \in V$ to all $v \in V$ or determines that a negative-weight cycle exists.

Thus, ***Bellman-Ford algorithm*** solves the single-source shortest-paths problem in the more general case in which edge weights can be **negative**. Given a weighted, directed graph $G=(V, E)$ with source s and weight function $w : E \to R$, the Bellman-Ford algorithm returns a **boolean value**

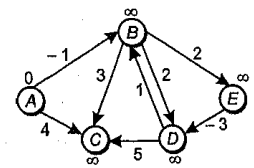
indicating whether or not there is a negative-weight cycle that is reachable from the source. If there is such a cycle, the algorithm indicates that **no solution** exists. If there is no such cycle, the algorithm produces the shortest paths and their weights. The algorithm returns TRUE if and only if the graph contains no negative-weight cycles that are reachable from the source.

BELLMAN-FORD(G, w, s)

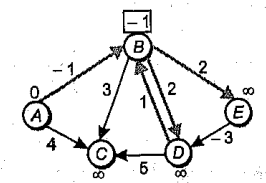
```
INITIALIZE-SINGLE-SOURCE(G, s)
for i ← 1 to |V[G]| − 1
    do for each edge (u,v) ∈ E[G]
        do RELAX(u,v,w)
for each edge (u,v) ∈ E[G]
    do if d[v] > d[u] + w(u,v)
        then return FALSE
return TRUE
```

Example :

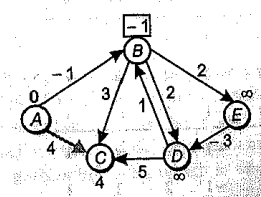
Order of edges : $(B, E), (D, B), (B, D), (A, B), (A, C), (D, C), (B, C), (E, D)$



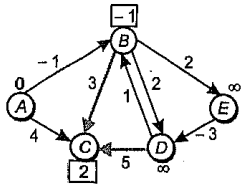
A	B	C	D	E
0	∞	∞	∞	∞



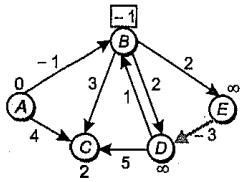
A	B	C	D	E
0	∞	∞	∞	∞
0	−1	∞	∞	∞



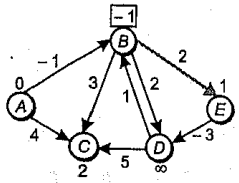
A	B	C	D	E
0	∞	∞	∞	∞
0	−1	∞	∞	∞
0	−1	4	∞	∞



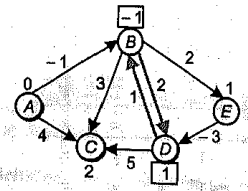
A	B	C	D	E
0	∞	∞	∞	∞
0	−1	∞	∞	∞
0	−1	4	∞	∞
0	−1	2	∞	∞



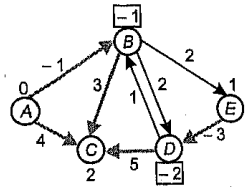
A	B	C	D	E
0	∞	∞	∞	∞
0	−1	∞	∞	∞
0	−1	4	∞	∞
0	−1	2	∞	∞



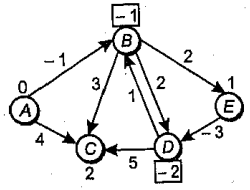
A	B	C	D	E
0	∞	∞	∞	∞
0	−1	∞	∞	∞
0	−1	4	∞	∞
0	−1	2	∞	∞
0	−1	2	∞	1



A	B	C	D	E
0	∞	∞	∞	∞
0	−1	∞	∞	∞
0	−1	4	∞	∞
0	−1	2	∞	∞
0	−1	2	∞	1
0	−1	2	1	1



A	B	C	D	E
0	∞	∞	∞	∞
0	−1	∞	∞	∞
0	−1	4	∞	∞
0	−1	2	∞	∞
0	−1	2	∞	1
0	−1	2	1	1
0	−1	2	−2	1



A	B	C	D	E
0	∞	∞	∞	∞
0	−1	∞	∞	∞
0	−1	4	∞	∞
0	−1	2	∞	∞
0	−1	2	∞	1
0	−1	2	1	1
0	−1	2	−2	1

Note *Values decrease monotonically.*

28.7 Single-source shortest paths in directed acyclic graphs

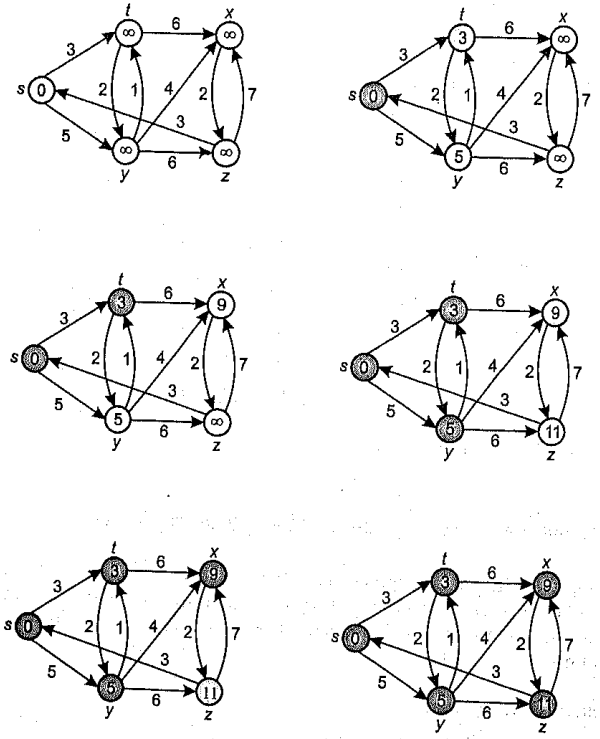
By relaxing the edges of a weighted dag (directed acyclic graph) $G=(V, E)$ according to a topological sort of its vertices, we can compute shortest paths from a single source in $\theta(V+E)$ time. Shortest paths are always well defined in a dag, since even if there are negative-weight edges, no negative-weight cycles can exist.

```
DAG-SHORTEST-PATHS (G, w, s)
1.  Topologically sort the vertices of G
2.  INITIALIZE-SINGLE-SOURCE (G, s)
3.  for each vertex u taken in topologically sorted order
4.      do for each vertex v ∈ Adj[u]
5.          do RELAX(u, v, w)
```

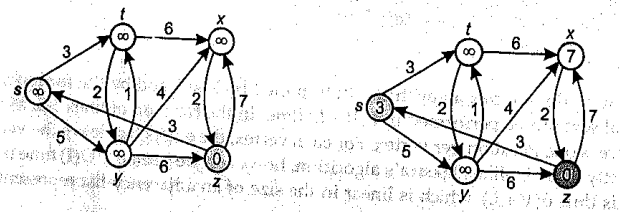
The running time of this algorithm is determined by line 1 and by the **for** loop of lines 3-5. The topological sort can be performed in $\theta(V+E)$ time. In the **for** loop of lines 3-5, as in Dijkstra's algorithm, there is one iteration per vertex. For each vertex, the edges that leave the vertex are each examined exactly once. Unlike Dijkstra's algorithm, however, we use only $O(1)$ time per edge. The running time is thus $\theta(V+E)$, which is linear in the size of an adjacency-list representation of the graph.

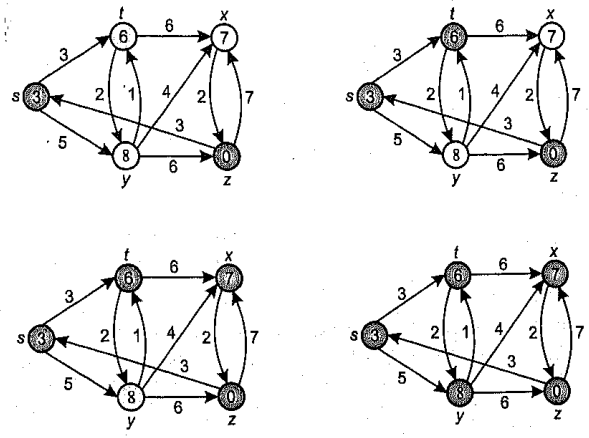
Example. *Run Dijkstra's algorithm on the directed graph of Fig. , First using vertex s as the source and then using vertex z as the source.*

Solution. First using vertex s as the source.



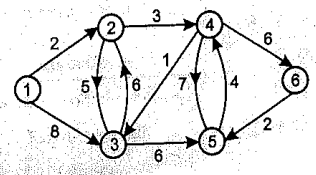
Then using vertex z as the source



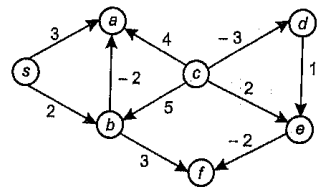


Exercise

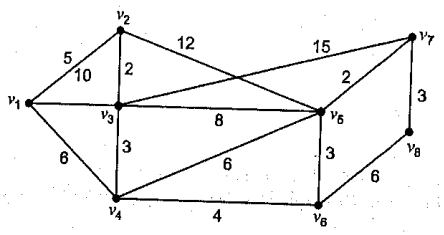
1. We would like to solve, as efficiently as possible, the single source shortest path problems in each of the following graphs. For each graph, state which algorithm would be best to use, and give its running time :
   - (*i*) A weighted directed acyclic graph.
   - (*ii*) A weighted directed graph where all edges weights are non-negative, the graph contains a directed cycle.
   - (*iii*) A weighted directed graph in which some, but not all of the edges have negative weights, the graph contains a directed cycle.
2. Solve the shortest path problems using Dijkstra's algorithm. Count the number of distance updates.



3. Find shortest path using BELLMANFORD algorithm from S to F of the following Fig.



4. Draw a simple, connected, weighted graph with 8 vertices and 16 edges, each with unique edge weights. Identify one vertex as a "start" vertex and illustrate a running of Dijkstra's algorithm on this graph.
5. Show how to modify Dijkstra's algorithm for the case when the graph is directed and we want to compute shortest directed paths from the source vertex to all other vertices.
6. Draw a simple (directed) weighted graph G with 10 vertices and 18 edges, such that G contains a minimum-weight cycle with at least 4 edges. Show that the Bellman-Ford algorithm will find this cyle.
7. Find shortest path from V_1 to V_8.



CHAPTER

29

All-Pairs Shortest Paths

29.1 Introduction

The all-pairs shortest path problem can be considered the mother of all routing problems. It aims to compute the shortest path from each vertex v to every other u. Using standard single-source algorithms, we can expect to get a naive implementation of $O(n^3)$ if we use Dijkstra for example, *i.e.* running a $O(n^2)$ process n times. Likewise, if we use the Bellman-Ford-Moore algorithm on a *dense graph*, it'll take about $O(n^4)$ but handle negative arc-lengths too.

Storing all the paths explicitly can be very memory expensive indeed, as we need one spanning tree for each vertex. This is often impractical in terms of memory consumption, so these are usually considered as all-pairs shortest distance problems, which aim to find just the distance from each to each node to another. We typically want the output in tabular form : the entry in u's row and v's column should be the weight of a shortest path from u to v.

Three approaches for improvement :

algorithm	*cost*
matrix multiplication	$O(V^3 \lg V)$
Floyd-Warshall	$O(V^3)$
Johnson	$O(V^2 \lg V + VE)$

Unlike the single-source algorithms, which assume an adjacency-list representation of the graph, most of the algorithms use an adjacency-matrix representation. (Johnson's algorithm for sparse graphs uses adjacency lists.) The input is an $n \times n$ matrix W representing the edge weights of an n-vertex directed graph $G=(V, E)$. That is, $W=(w_{ij})$, where

$$w_{ij} = \begin{cases} 0 & \text{if } i=j, \\ w(i,j) & \text{if } i \neq j \text{ and } (i,j) \in E, \\ \infty & \text{if } i \neq j \text{ and } (i,j) \notin E. \end{cases}$$

Negative-weight edges are allowed, but we assume for the time being that the input graph contains no negative-weight cycles.

The tabular output of the all-pairs shortest-paths algorithms is an $n \times n$ matrix $D=(d_{ij})$, where entry d_{ij} contains the weight of a shortest path from vertex i to vertex j. That is, if we let $\delta(i,j)$ denote the shortest-path weight from vertex i to vertex j, then $d_{ij} = \delta(i,j)$ at termination.

To solve the all-pairs shortest-paths problem on an input adjacency matrix, we need to compute not only the shortest-path weights but also a ***predecessor matrix*** $\Pi=(\pi_{ij})$, where π_{ij} is NIL if either $i=j$ or there is no path from i to j, and otherwise π_{ij} is some predecessor of j on a shortest path from i. For each vertex $i \in V$, we define the ***predecessor sub graph*** of G for i as $G_{\pi,i} = (V_{\pi,i}, E_{\pi,i})$, where

$$V_{\pi,i} = \{j \in V : \pi_{ij} \neq \text{NIL}\} \cup \{i\} \quad \text{and} \quad E_{\pi,i} = \{(\pi_{ij}, j) : j \in V_{\pi,i} \text{ and } \pi_{ij} \neq \text{NIL}\}$$

29.2 Matrix Multiplication

Let us define the $V \times V$ matrix

$$D^{(m)} = (d_{ij}^{(m)})$$

$d_{ij}^{(m)}$ = the length of the shortest path from i to j with $\leq m$ edges.

When $m=0$, there is a shortest path from i to j with no edges if and only if $i=j$.

$$d_{ij}^{(0)} = \begin{cases} 0 & \text{if } i=j \\ w_{ij} & \text{if } i \neq j, (i,j) \in E, \\ \infty & \text{otherwise} \end{cases}$$

29.3 The Floyd-Warshall Algorithm

Floyd–Warshall algorithm (sometimes known as the **Roy–Floyd algorithm**, since Bernard Roy described this algorithm in 1959) is a graph analysis algorithm for finding shortest paths in a weighted, directed graph. A single execution of the algorithm will find the shortest path between all pairs of vertices. It does so in $\theta(V^3)$ time, where V is the number of vertices in the graph. Negative-weight edges may be present, but we shall assume that there are no negative-weight cycles.

The algorithm considers the *"intermediate"* vertices of a shortest path, where an ***intermediate*** vertex of a simple path $p=\langle v_1, v_2, \ldots, v_m \rangle$ is any vertex of p other than v_1 or v_m, that is, any vertex in the set $\{v_2, v_3, \ldots, v_{m-1}\}$.

The Floyd-Warshall algorithm is based on the following observation. Let the vertices of G be $V=\{1,2,\ldots,n\}$, and consider a subset $\{1,2,\ldots,k\}$ of vertices for some k. For any pair of vertices $i, j \in V$, consider all paths from i to j whose intermediate vertices are all drawn from $\{1,2,\ldots,k\}$, and let p be a minimum-weight path from among them. (Path p is simple, since we assume that G contains no negative-weight cycles.) The Floyd-Warshall algorithm exploits a relationship between path p and shortest paths from i to j with all intermediate vertices in the set $\{1,2,\ldots k-1\}$. The relationship depends on whether or not k is an intermediate vertex of path p.

If k is not an intermediate vertex of path p, then all intermediate vertices of path p are in the set $\{1,2,\ldots k-1\}$. Thus, a shortest path from vertex i to vertex j with all intermediate vertices in the set $\{1,2,\ldots k-1\}$ is also a shortest path from i to j with all intermediate vertices in the set $\{1,2,\ldots,k\}$.

If k is an intermediate vertex of path p, then we break p down into $i \xrightarrow{p_1} k \xrightarrow{p_2} j$.

Let $d_{ij}^{(k)}$ be the weight of a shortest path from vertex i to vertex j with all intermediate vertices in the set $\{1, 2, \ldots, k\}$.

A recursive definition is given by

$$d_{ij}^{(k)} = \begin{cases} w_{ij} & \text{if } k=0 \\ \min(d_{ij}^{(k-1)}, d_{ik}^{(k-1)} + d_{kj}^{(k-1)}) & \text{if } k \geq 1) \end{cases}$$

```
FLOYD-WARSHALL (W)
1.  n ← rows[W]
2.  D^(0) ← W
3.  for k ← 1 to n
4.      do for i ← 1 to n
5.          do for j ← 1 to n
6.              do d_ij^(k) ← min(d_ij^(k-1), d_ik^(k-1) + d_kj^(k-1))
7.  return D^(n)
```

The strategy adopted by the Floyd-Warshall algorithm is **dynamic programming**.

The running time of the Floyd-Warshall algorithm is determined by the triply nested for loops of lines 3-6. Each execution of line 6 takes $O(1)$ time. The algorithm thus runs in time $\theta(n^3)$.

29.3.1 Constructing a Shortest Path

There are a variety of different methods for constructing shortest paths in the Floyd-Warshall algorithm. One way is to compute the matrix D of shortest-path weights and then construct the predecessor matrix Π from the D matrix. This method can be implemented to run in $O(n^3)$ time. Given predecessor matrix Π, the PRINT-ALL-PAIRS-SHORTEST-PATH procedure can be used to print the vertices on a given shortest path.

We can compute the predecessor matrix Π just as the Floyd-Warshall algorithm computes the matrices $D(k)$. Specifically, we compute a sequence of matrices $\Pi^{(0)}, \Pi^{(1)}, \ldots, \Pi^{(n)}$, where $\Pi = \Pi^{(n)}$ and is defined to be the predecessor of vertex j on a shortest path from vertex i with all intermediate vertices in the set $\{1, 2, \ldots, k\}$.

The recursive formulation of $\pi_{ij}^{(k)}$. When $k=0$, a shortest path from i to j has no intermediate vertices at all. Thus,

$$\pi_{ij}^{(0)} = \begin{cases} \text{NIL} & \text{if } i=j \text{ or } w_{ij}=\infty \\ i & \text{if } i=j \text{ and } w_{ij}<\infty \end{cases}$$

For $k \geq 1$, if we take the path $i \rightsquigarrow k \rightsquigarrow j$, where $k \neq j$ then the predecessor of j we choose is the same as the predecessor of j we chose on a shortest path from k with all intermediate vertices in the set $\{1, 2, \ldots, k-1\}$. Otherwise, we choose the same predecessor of j that we chose on a shortest path from i with all intermediate vertices in the set $\{1, 2, \ldots, k-1\}$. Formally, for $k \geq 1$,

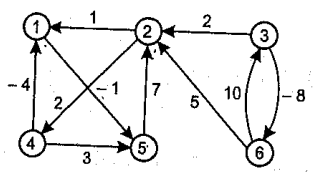
$$\pi_{ij}^{(k)} = \begin{cases} \pi_{ij}^{(k-1)} & \text{if } d_{ij}^{(k-1)} \leq d_{ik}^{(k-1)} + d_{kj}^{(k-1)} \\ \pi_{kj}^{(k-1)} & \text{if } d_{ij}^{(k-1)} > d_{ik}^{(k-1)} + d_{kj}^{(k-1)} \end{cases}$$

Example. *How can the output of the Floyd-Warshall algorithm be used to detect the presence of a negative weight cycle ?*

Solution. Check for every edge (i, j) whether

$$w_{ij} + d_{ji} < 0$$

Example. *Apply Floyd-Warshall algorithm for constructing shortest path. Show the matrix $D^{(k)}$ that results each iteration.*



Solution.

$$D^{(0)} = \begin{bmatrix} 0 & \infty & \infty & \infty & -1 & \infty \\ 1 & 0 & \infty & 2 & \infty & \infty \\ \infty & 2 & 0 & \infty & \infty & -8 \\ -4 & \infty & \infty & 0 & 3 & \infty \\ \infty & 7 & \infty & \infty & 0 & \infty \\ \infty & 5 & 10 & \infty & \infty & 0 \end{bmatrix}$$

We know

$$d_{ij}^{(k)} = \min[d_{ij}^{(k-1)}, d_{ik}^{(k-1)} + d_{kj}^{(k-1)}]$$

k	i	j	$d_{ij}^{(k-1)}$	$d_{ik}^{(k-1)}$	$d_{kj}^{(k-1)}$	d_{ij}^{k}
1	1	1	0	0	0	0
1	1	2	∞	0	∞	∞
1	1	3	∞	0	∞	∞
1	1	4	∞	0	∞	∞
1	1	5	−1	0	−1	−1
1	1	6	∞	0	∞	∞
1	2	1	1	1	0	1
1	2	2	0	1	∞	0
1	2	3	∞	1	∞	∞
1	2	4	2	1	∞	2
1	2	5	∞	1	−1	0
1	2	6	∞	1	∞	∞
1	3	1	∞	∞	0	∞
1	3	2	2	∞	∞	2
1	3	3	0	∞	∞	0
1	3	4	∞	∞	∞	∞
1	3	5	∞	∞	−1	∞
1	3	6	−8	∞	∞	−8
1	4	1	−4	−4	0	−4
1	4	2	∞	−4	∞	∞
1	4	3	∞	−4	∞	∞
1	4	4	0	−4	∞	0
1	4	5	3	−4	−1	−5
1	4	6	∞	−4	∞	∞
1	5	1	∞	∞	0	∞
1	5	2	7	∞	∞	7
1	5	3	∞	∞	∞	∞
1	5	4	∞	∞	∞	∞
1	5	5	0	∞	−1	0
1	5	6	∞	∞	∞	∞
1	6	1	∞	∞	0	∞
1	6	2	5	∞	∞	5
1	6	3	10	∞	∞	10
1	6	4	∞	∞	∞	∞
1	6	5	∞	∞	−1	∞
1	6	6	0	∞	∞	0

$D^{(1)}$	1	2	3	4	5	6
1	0	∞	∞	∞	−1	∞
2	1	0	∞	2	0	∞
3	∞	2	0	∞	∞	−8
4	−4	∞	∞	0	−5	∞
5	∞	7	∞	∞	0	∞
6	∞	5	10	∞	∞	0

Similarly, we get

$D^{(2)}$	1	2	3	4	5	6
1	0	∞	∞	∞	−1	∞
2	1	0	∞	2	0	∞
3	3	2	0	4	2	−8
4	−4	∞	∞	0	−5	∞
5	8	7	∞	9	0	∞
6	6	5	10	7	5	0

$D^{(3)}$	1	2	3	4	5	6
1	0	∞	∞	∞	−1	∞
2	1	0	∞	2	0	∞
3	3	2	0	4	2	−8
4	−4	∞	∞	0	−5	∞
5	8	7	∞	9	0	∞
6	6	5	10	7	5	0

$D^{(4)}$	1	2	3	4	5	6
1	0	∞	∞	∞	−1	∞
2	−2	0	∞	2	−3	∞
3	0	2	0	4	−1	−8
4	−4	∞	∞	0	−5	∞
5	5	7	∞	9	0	∞
6	3	5	10	7	2	0

$D^{(5)}$	1	2	3	4	5	6
1	0	6	∞	8	−1	∞
2	−2	0	∞	2	−3	∞
3	0	2	0	4	−1	−8
4	−4	2	∞	0	−5	∞
5	5	7	∞	9	0	∞
6	3	5	10	7	2	0

and

$D^{(6)}$	1	2	3	4	5	6
1	0	6	∞	8	−1	∞
2	−2	0	∞	2	−3	∞
3	−5	−3	0	−1	−6	−8
4	−4	2	∞	0	−5	∞
5	5	7	∞	9	0	∞
6	3	5	10	7	2	0

The shortest path from 3 to 1 is

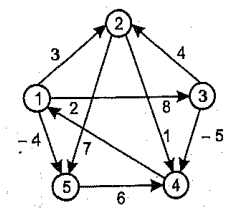
$3 \to 6 \to 2 \to 4 \to 1$ and is -5 units.

Example. *"As it appears above, the Floyd-Warshall algorithm requires $\theta(n^3)$ space, since we compute $d_{ij}^{(k)}$ for $i, j, k = 1,2,3,\ldots,n$. Show that the following procedure, which simply drops all superscripts, is correct and thus only $\theta(n^2)$ space is required".*

```
FLOYD-WARSHALL'(W)
n ← rows(W)
D ← W
for k ← 1 to n
    do for i ← 1 to n
        do for j ← 1 to n
            do d_ij ← min(d_ij, d_ik + d_kj)
return D.
```

Solution. Since $d_{ik}^{(k)} = d_{ik}^{(k-1)}$ and $d_{kj}^{(k-1)} = d_{kj}^{(k)}$, no entry with either subscript changes during iteration. Therefore we can perform the computation with only copy of D.

Example. *Apply Floyd-Warshall algorithm for constructing shortest path. Show the matrices $D^{(k)}$ and $\pi^{(k)}$ computed by the Floyd-Warshall algorithm for the graph.*



Solution. $d_{ij}^{(k)} = \min[d_{ij}^{(k-1)}, d_{ik}^{(k-1)} + d_{kj}^{(k-1)}]$

$$\pi_{ij}^{(k)} = \begin{cases} \pi_{ij}^{(k-1)} & \text{if } d_{ij}^{(k-1)} \le d_{ik}^{(k-1)} + d_{kj}^{(k-1)} \\ \pi_{kj}^{(k-1)} & \text{if } d_{ij}^{(k-1)} > d_{ik}^{(k-1)} + d_{kj}^{(k-1)} \end{cases}$$

$$D^{(0)} = \begin{pmatrix} 0 & 3 & 8 & \infty & -4 \\ \infty & 0 & \infty & 1 & 7 \\ \infty & 4 & 0 & \infty & \infty \\ 2 & \infty & -5 & 0 & \infty \\ \infty & \infty & \infty & 6 & 0 \end{pmatrix} \quad \pi^{(0)} = \begin{bmatrix} \text{NIL} & 1 & 1 & \text{NIL} & 1 \\ \text{NIL} & \text{NIL} & \text{NIL} & 2 & 2 \\ \text{NIL} & 3 & \text{NIL} & \text{NIL} & \text{NIL} \\ 4 & \text{NIL} & 4 & \text{NIL} & \text{NIL} \\ \text{NIL} & \text{NIL} & \text{NIL} & 5 & \text{NIL} \end{bmatrix}$$

$$D^{(1)} = \begin{pmatrix} 0 & 3 & 8 & \infty & -4 \\ \infty & 0 & \infty & 1 & 7 \\ \infty & 4 & 0 & \infty & \infty \\ 2 & 5 & -5 & 0 & -2 \\ \infty & \infty & \infty & 6 & 0 \end{pmatrix} \quad \pi^{(1)} = \begin{bmatrix} \text{NIL} & 1 & 1 & \text{NIL} & 1 \\ \text{NIL} & \text{NIL} & \text{NIL} & 2 & 2 \\ \text{NIL} & 3 & \text{NIL} & \text{NIL} & \text{NIL} \\ 4 & 1 & 4 & \text{NIL} & 1 \\ \text{NIL} & \text{NIL} & \text{NIL} & 5 & \text{NIL} \end{bmatrix}$$

$$D^{(2)} = \begin{pmatrix} 0 & 3 & 8 & 4 & -4 \\ \infty & 0 & \infty & 1 & 7 \\ \infty & 4 & 0 & 5 & 11 \\ 2 & 5 & -5 & 0 & -2 \\ \infty & \infty & \infty & 6 & 0 \end{pmatrix} \qquad \pi^{(2)} = \begin{bmatrix} \text{NIL} & 1 & 1 & 2 & 1 \\ \text{NIL} & \text{NIL} & \text{NIL} & 2 & 2 \\ \text{NIL} & 3 & \text{NIL} & 2 & 2 \\ 4 & 1 & 4 & \text{NIL} & 1 \\ \text{NIL} & \text{NIL} & \text{NIL} & 5 & \text{NIL} \end{bmatrix}$$

$$D^{(3)} = \begin{pmatrix} 0 & 3 & 8 & 4 & -4 \\ \infty & 0 & \infty & 1 & 7 \\ \infty & 4 & 0 & 5 & 11 \\ 2 & -1 & -5 & 0 & -2 \\ \infty & \infty & \infty & 6 & 0 \end{pmatrix} \qquad \pi^{(3)} = \begin{bmatrix} \text{NIL} & 1 & 1 & 2 & 1 \\ \text{NIL} & \text{NIL} & \text{NIL} & 2 & 2 \\ \text{NIL} & 3 & \text{NIL} & 2 & 2 \\ 4 & 3 & 4 & \text{NIL} & 1 \\ \text{NIL} & \text{NIL} & \text{NIL} & 5 & \text{NIL} \end{bmatrix}$$

$$D^{(4)} = \begin{pmatrix} 0 & 3 & -1 & 4 & -4 \\ 3 & 0 & -4 & 1 & -1 \\ 7 & 4 & 0 & 5 & 3 \\ 2 & -1 & -5 & 0 & -2 \\ 8 & 5 & 1 & 6 & 0 \end{pmatrix} \qquad \pi^{(4)} = \begin{bmatrix} \text{NIL} & 1 & 4 & 2 & 1 \\ 4 & \text{NIL} & 4 & 2 & 1 \\ 4 & 3 & \text{NIL} & 2 & 1 \\ 4 & 3 & 4 & \text{NIL} & 1 \\ 4 & 3 & 4 & 5 & \text{NIL} \end{bmatrix}$$

$$D^{(5)} = \begin{pmatrix} 0 & 1 & -3 & 2 & -4 \\ 3 & 0 & -4 & 1 & -1 \\ 7 & 4 & 0 & 5 & 3 \\ 2 & -1 & -5 & 0 & -2 \\ 8 & 5 & 1 & 6 & 0 \end{pmatrix} \qquad \pi^{(5)} = \begin{bmatrix} \text{NIL} & 3 & 4 & 5 & 1 \\ 4 & \text{NIL} & 4 & 2 & 1 \\ 4 & 3 & \text{NIL} & 2 & 1 \\ 4 & 3 & 4 & \text{NIL} & 1 \\ 4 & 3 & 4 & 5 & \text{NIL} \end{bmatrix}$$

29.4 Transitive Closure

The transitive closure of a graph G is defined as $G^* = (V, E^*)$ where

$$E^* = \{(i,j) \text{ there is a path from vertex } i \text{ to vertex } j \text{ in } G\}$$

For $i, j, k = 1, 2, \ldots n$ we define $t_{ij}^{(k)}$ to be 1 if there exists a path in graph G from vertex i to vertex j with all intermediate vertices in the set $\{1, 2, \ldots, k\}$ and 0 otherwise we construct the transitive closure $G^* = (V, E^*)$ by putting edge (i,j) into E^* if and only if $t_{ij}^{(n)} = 1$.

The recursive definition of $t_{ij}^{(k)}$ is

$$t_{ij}^{(0)} = \begin{cases} 0 & \text{if } i \neq j \text{ and } (i,j) \notin E \\ 1 & \text{if } i = j \text{ or } (i,j) \in E \end{cases}$$

and for $k \geq 1$

$$t_{ij}^{(k)} = t_{ij}^{(k-1)} \vee (t_{ik}^{(k-1)} \wedge t_{kj}^{(k-1)})$$

TRANSITIVE-CLOSURE (G)

```
n ← |V[G]|
for i ← 1 to n
    do for j ← 1 to n
        do if i = j or (i, j) ∈ E[G]
            then t_ij^(0) ← 1
            else t_ij^(0) ← 0
for k ← 1 to n
    do for i ← 1 to n
        do for j ← 1 to n
            do t_ij^(k) ← t_ij^(k-1) ∨ (t_ik^(k-1) ∧ t_kj^(k-1))
```


Example. *Give an $O(VE)$-time algorithm for computing the transitive closure of a directed graph $G = (V, E)$.*

Solution.

1. Construct a new graph $G^* = (V, E^*)$
2. E^* is initially empty.
3. For each vertex v traverse the graph G adding edges for every node encountered in E^*.

   This takes $O(VE)$-time.

29.5 Johnson's Algorithm

Johnson's algorithm finds shortest paths between all pairs in $O(V^2 \lg V + VE)$ time. The algorithm returns a matrix of shortest path weight for all pairs or reports that the input graph contains a negative-weight cycle.

Johnson's algorithm use the technique of "**reweighting**". If all edge weights w in a graph $G = (V, E)$ are non-negative, we can find shortest paths between all pairs of vertices by running Dijkstra's algorithm once from each vertex. If G has negative-weight edges, we simply compute a new-set of non-negative edge weights that allows us to use the same method. The new set of edges weights $\hat{w}$ must satisfy *two* important properties :

1. For all pairs of vertices $u, v \in V$, a shortest path from u to v using weight function w is also a shortest path from u to v using weight function $\hat{w}$.
2. For all edges (u, v), the new weight $\hat{w}(u, v)$ is non-negative.

Given a weighted, directed graph $G = (V, E)$ with weight function $w : E \to R$ and let $h : v \to R$ be any function mapping vertices to real number.

For each edge $(u, v) \in E$ define

$$\hat{w}(u, v) = w(u, v) + h(u) - h(v)$$

where $h(u)$ = label of u

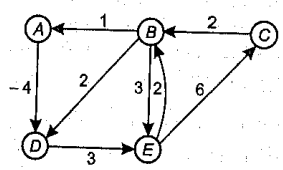
$h(v)$ = label of v

JOHNSON(G)

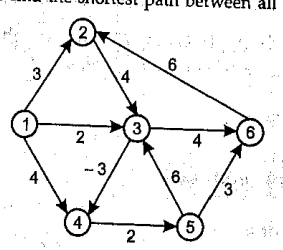
1. Compute G' where $V[G'] = V[G] \cup \{S\}$ and
   $E[G'] = E[G] \cup \{(s,v) : v \in V[G]\}$
2. If BELLMAN-FORD (G', w, s) = FALSE
   then "input graph contains a negative wt. cycle"
   else
   for each vertex $v \in V[G']$
   do $h(v) \leftarrow \delta(s,v)$
   computed by Bellman-Ford algorithm.
   for each edge $(u,v) \in E[G']$
   do $\hat{w}(u,v) \leftarrow w(u,v) + h(u) - h(v)$
   for each vertex $u \in V[G]$
   do run DIJKSTRA $(G, \hat{w}, u)$ to compute
   $\hat{\delta}(u,v)$ for all $v \in V[G]$
   for each vertex $v \in V[G]$
   do $d_{uv} \leftarrow \hat{\delta}(u,v) + h(v) - h(u)$
   return D.

Exercise

1. Write any algorithm to find all-pair shortest path. Drive its complexity.
2. For the graph (directed, weighted)
   Apply Floyd-Warshall algorithm for constructing shortest path. Show the matrix $D^{(k)}$ that results each iteration.



3. Use Johnson's algorithm to find the shortest path between all pairs of vertices in the graph.



4. Modify the Floyd-Warshall algorithm to find the negative weight cycle.

Maximum Flow

Some real-life problems, like those involving the flow of liquids through pipes, current through wires, and delivery of goods, can be modeled using flow networks. Other problems which seem unrelated to networks can also be modeled using flow networks. We discuss the basics of flow networks, several network flow problems, and develop a few algorithms to solve the maximum network flow problem.

30.1 Flow Networks and Flows

A **flow network** is a directed graph $G = (V, E)$ such that

1. For edge $(u,v) \in E$, we associate a nonnegative **capacity** $c(u,v) \geq 0$. If $(u,v) \notin E$, we assume that $c(u,v) = 0$.
2. There are two distinguished points, the **source** s, and the **sink** t;
3. For every vertex $v \in V$, there is a path from s to t containing v.

Let $G = (V, E)$ be a flow network. Let s be the source of the network, and let t be the sink. A ***flow*** in G is a real-valued function $f : V \times V \rightarrow \mathbf{R}$ such that the following properties hold :

- **Capacity constraint.** For all $u, v \in V$, we require $f(u,v) \leq c(u,v)$.
- **Skew symmetry.** For all $u, v \in V$, we require $f(u,v) = -f(v,u)$.

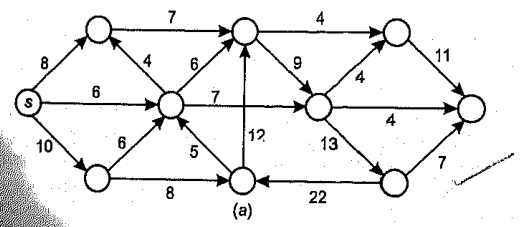
◂ **Flow conservation.** For all $u \in V-\{s,t\}$, we require

$$\sum_{v \in V} f(u,v) = \sum_{u \in V} f(u,v) = 0$$

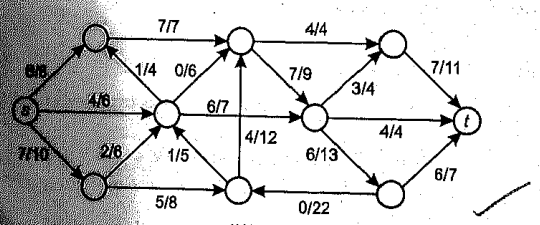
The quantity $f(u,v)$, which can be positive or negative, is called the ***net flow*** from vertex u to vertex v. In the ***maximum-flow problem***, we are given a flow network G with source s and sink t, and we wish to find a flow of maximum value from s to t.

The *three* properties can be described as follows :

1. **Capacity constraint** makes sure that the flow through each edge is not greater than the capacity.
2. **Skew symmetry** simply means that the flow from u to v is the negative of the flow from v to u.
3. The **flow-conservation property** says that the total net flow out of a vertex other than the source or sink is 0. In other words, the amount of flow into a v is the same as the amount of flow out of v for every vertex $v \in V-\{s,t\}$



(a)



(b)

rk $G=(V,E)$ and Fig. 30.1(*b*) a flow f in G. Only the positive flows are shown. beled with *flow/capacity*. For example, 3/4 means the flow ity is 4.

The **value** of the flow is the net flow from the source,

$$|f| = \sum_{v \in V} f(s,v)$$

The **positive net flow** entering a vertex v is defined by

$$\sum_{\{u \in V:\, f(u,v)>0\}} f(u,v)$$

The positive net flow leaving a vertex is defined symmetrically. One interpretation of the flow-conservation property is that the positive net flow entering a vertex other than the source or sink must equal the positive net flow leaving the vertex.

A flow f is said to be **integer-valued** if $f(u,v)$ is an integer for all $(u,v) \in E$. Clearly the value of the flow is an integer in an integer-valued flow.

Implicit summation notation

For convenience, we will omit the set braces when it simplifies notation. For instance, we will write $V-s-t$ instead of $V-\{s,t\}$ The following notation will greatly simplify the writing of proofs involving functions like the net flow f.

If X and Y are sets, and g is a 2-variable function, we define

$$g(X,Y) = \sum_{x \in X} \sum_{y \in Y} g(x,y)$$

For example, the flow-conservation constraint can be expressed as the condition that $f(u,V)=0$ for all $u \in V-s-t$.

Lemma

If $X \subseteq V-s-t$, then $f(V,X) = f(X,V) = 0$.

Proof. Since $X \subseteq V-s-t$,,

$$f(V,X) = \sum_{x \in X} f(V,x) = \sum_{x \in X} 0 = 0$$

The proof is similar for $f(X,V)=0$.

The following identities will also be useful. Let $G=(V,E)$ be a flow network, and let f be a flow in G. Then, for $X,Y,Z \subseteq V$ and $Y \cap Z = \phi$,

◂ $f(X,X)=0$.

◂ $f(X,Y) = -f(Y,X)$.

◂ $f(X \cup Y, Z) = f(X,Z) + f(Y,Z)$ and

◂ $f(Z, X \cup Y) = f(Z,X) + f(Z,Y)$

We can prove that the value of a flow is the total net flow into the sink ; that is,

$$|f| = f(V,t)$$

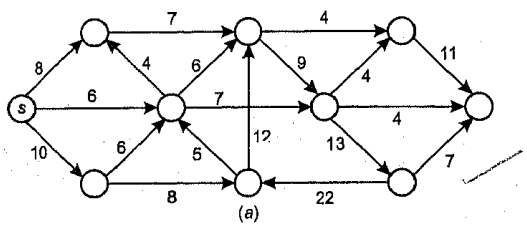
◄| **Flow conservation.** For all $u \in V-\{s,t\}$, we require

$$\sum_{v \in V} f(u,v) = \sum_{u \in V} f(u,v) = 0$$

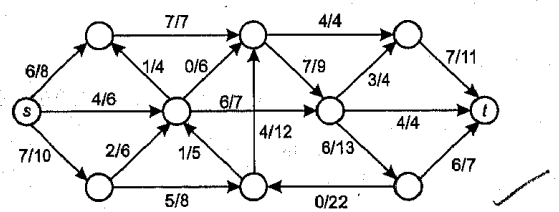
The quantity $f(u,v)$, which can be positive or negative, is called the ***net flow*** from vertex u to vertex v. In the ***maximum-flow problem***, we are given a flow network G with source s and sink t, and we wish to find a flow of maximum value from s to t.

The *three* properties can be described as follows :

1. **Capacity constraint** makes sure that the flow through each edge is not greater than the capacity.
2. **Skew symmetry** simply means that the flow from u to v is the negative of the flow from v to u.
3. The **flow-conservation property** says that the total net flow out of a vertex other than the source or sink is 0. In other words, the amount of flow into a v is the same as the amount of flow out of v for every vertex $v \in V-\{s,t\}$



(a)



(b)

Figure 30.1 (*a*) A flow network $G=(V,E)$ and Fig. 30.1(*b*) a flow f in G. Only the positive flows are shown. Each edge is labeled with *flow/capacity*. For example, 3/4 means the flow is 3 and the capacity is 4.

The **value** of the flow is the net flow from the source,

$$|f| = \sum_{v \in V} f(s,v)$$

The **positive net flow** entering a vertex v is defined by

$$\sum_{\{u \in V:\, f(u,v)>0\}} f(u,v)$$

The positive net flow leaving a vertex is defined symmetrically. One interpretation of the flow-conservation property is that the positive net flow entering a vertex other than the source or sink must equal the positive net flow leaving the vertex.

A flow f is said to be **integer-valued** if $f(u,v)$ is an integer for all $(u,v) \in E$. Clearly the value of the flow is an integer in an integer-valued flow.

Implicit summation notation

For convenience, we will omit the set braces when it simplifies notation. For instance, we will write $V-s-t$ instead of $V-\{s,t\}$ The following notation will greatly simplify the writing of proofs involving functions like the net flow f.

If X and Y are sets, and g is a 2-variable function, we define

$$g(X,Y) = \sum_{x \in X} \sum_{y \in Y} g(x,y)$$

For example, the flow-conservation constraint can be expressed as the condition that $f(u,V)=0$ for all $u \in V-s-t$.

Lemma

If $X \subseteq V-s-t$, then $f(V,X) = f(X,V) = 0$.

Proof. Since $X \subseteq V-s-t$,,

$$f(V,X) = \sum_{x \in X} f(V,x) = \sum_{x \in X} 0 = 0$$

The proof is similar for $f(X,V)=0$.

The following identities will also be useful. Let $G=(V,E)$ be a flow network, and let f be a flow in G. Then, for $X,Y,Z \subseteq V$ and $Y \cap Z = \phi$,

◄| $f(X,X)=0$.

◄| $f(X,Y) = -f(Y,X)$.

◄| $f(X \cup Y, Z) = f(X,Z) + f(Y,Z)$ and

◄| $f(Z, X \cup Y) = f(Z,X) + f(Z,Y)$

We can prove that the value of a flow is the total net flow into the sink ; that is,

$$|f| = f(V,t)$$

This is intuitively true, since all vertices other than the source and sink have a net flow of 0 by flow conservation, and thus the sink is the only other vertex that can have a nonzero net flow to match the source's nonzero net flow. Our formal proof goes as follows :

$$
\begin{aligned}
|f| &= f(s,V) && \textit{(by definition)}\\
&= f(V,V) - f(V-s,V) && \textit{(by Lemma)}\\
&= f(V,V-s) && \textit{(by Lemma)}\\
&= f(V,t) + f(V,V-s-t) && \textit{(by Lemma)}\\
&= f(V,t) && \textit{(by flow conservation)}
\end{aligned}
$$

30.2 Network Flow Problems

The most obvious flow network problem is the following.

Problem 1. *Given a flow network $G=(V,E)$, the maximum flow problem is to find a flow with maximum value.*

Problem 2. *The multiple source and sink maximum flow problem is similar to the maximum flow problem, except there is a set $\{s_1, s_2, s_3 \ldots s_n\}$ of sources and a set $\{t_1, t_2, t_3 \ldots t_n\}$ of sinks.*

Fortunately, this problem is no harder than ordinary maximum flow. Given a multiple source and sink flow network G, we define a new flow network G′ by adding

- A super source s,
- A super sink t,
- For each s_i, add edge (s, s_i) with capacity ∞, and
- For each t_i, add edge (t_i, t) with capacity ∞

Figure shows a multiple source and sinks flow network and an equivalent single source and sink flow network.

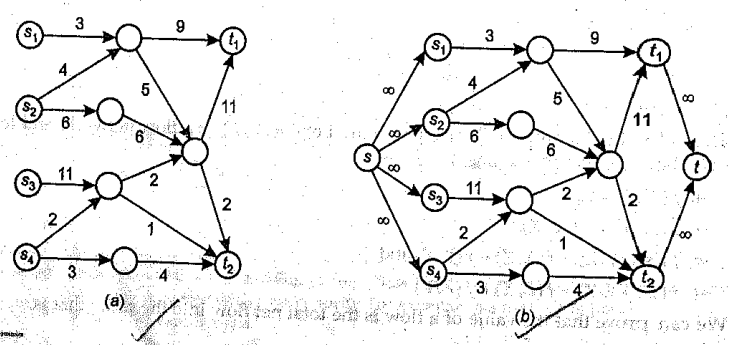


Figure 30.2

30.3 Residual Networks, Augmenting Paths, and Cuts

The residual network consists of edges that can admit more net flow. More formally, suppose that we have a flow network $G=(V,E)$ with source s and sink t. Let f be a flow in G, and consider a pair of vertices $u, v \in V$. The amount of **additional net flow** we can push from u to v before exceeding the capacity $c(u,v)$ is the **residual capacity** of (u,v), given by

$$c_f(u,v) = c(u,v) - f(u,v).$$

When the net flow $f(u,v)$ is negative, the residual capacity $c_f(u,v)$ is greater than the capacity $c(u,v)$. For example, if $c(u,v)=16$ and $f(u,v)=-4$, then the residual capacity $c_f(u,v)$ is 20.

Given a flow network $G=(V,E)$ and a flow f, the ***residual network*** of G induced by f is $G_f = (V, E_f)$, where

$$E_f = \{(u,v) \in V \times V : c_f(u,v) \geq 0\}$$

That is, each edge of the residual network, or ***residual edge***, can admit a strictly positive net flow.

Lemma

Let $G=(V,E)$ be a flow network with flow f. G_f be the residual network of G induced by f, and let f_r be a flow in G_f. Let $f' = f + f_r$ that is for each pair $u, v \in V$, $f'(u,v) = f(u,v) + f_r(u,v)$. Then f' is a flow in G with value $|f'| = |f| + |f_r|$.

Proof. We need to verify that the 3 properties of a flow are satisfied

1. By definition

$$f_r(u,v) \leq c_f(u,v) = c(u,v) - f(u,v) \text{ for all } u, v \in V.$$

Given this, we can demonstrate capacity constraint as follows

$$
\begin{aligned}
f'(u,v) &= f(u,v) + f_r(u,v)\\
&\leq f(u,v) + c(u,v) - f(u,v)\\
&= c(u,v)
\end{aligned}
$$

2. To see that f' has skew symmetry, notice that $u, v \in V$.

$$
\begin{aligned}
f'(u,v) &= f(u,v) + f_r(u,v)\\
&= -f(v,u) - f_r(v,u)\\
&= -(f(v,u) + f_r(v,u))\\
&= -f'(v,u)
\end{aligned}
$$

3. f' has flow conservation, since for all $u \in V - s - t$

$$\sum_{v \in V} f'(u,v) = \sum_{v \in V} (f(u,v) + f_r(u,v))$$

$$= \sum_{v \in V} f(u,v) + \sum_{v \in V} f_r(u,v)$$

$$= 0 + 0 = 0$$

Finally we can easily see that

$$|f'| = \sum_{v \in V} f'(s,v)$$

$$= \sum_{v \in V} (f(s,v) + f_r(s,v))$$

$$= \sum_{v \in V} f(s,v) + \sum_{v \in V} f_r(s,v)$$

$$= |f| + |f_r|.$$

Given a flow network $G = (V, E)$ and a flow f, an **augmenting path** p is a simple path from s to t in the residual network G_f. By the definition of the residual network, each edge (u,v) on an augmenting path admits some additional positive net flow from u to v without violating the capacity constraint on the edge.

Let $G = (V, E)$ be a flow network with flow f. The **residual capacity** of an augmenting path p is

$$c_f(p) = \min\{c_f(u,v) : (u,v) \text{ is on } p\}$$

The residual capacity is the maximal amount of flow that can be pushed through the augmenting path. Notice that if there is an augmenting path, then each edge on it has positive capacity. We will use this fact to compute a maximum flow in a flow network.

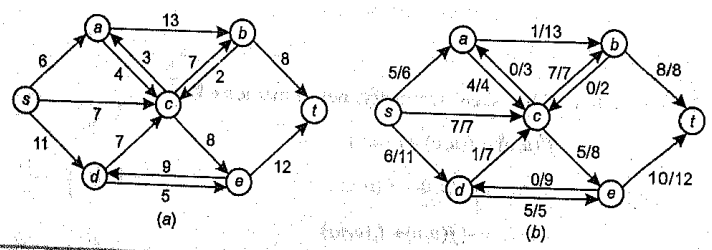


Figure 30.3 (a) A flow network $G = (V,E)$ (b) A flow in f in G

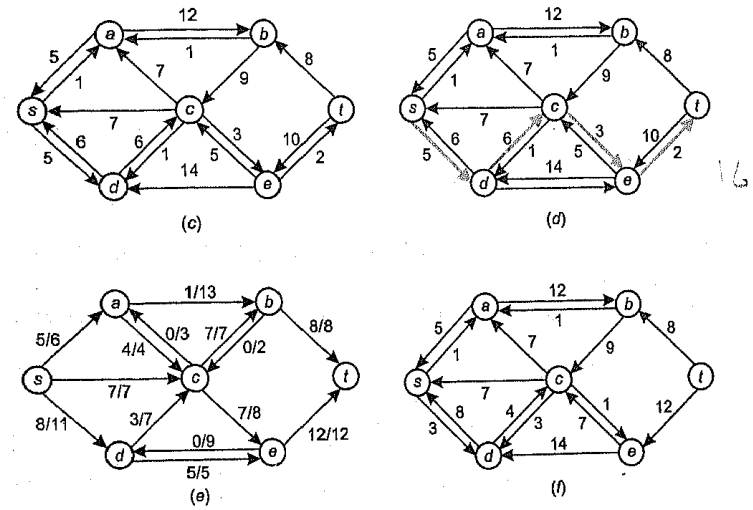


Figure 30.3 (c) The residual network G_f. (d) The grey edges form an augmenting path with capacity 2. (e) A new flow $f' = f + f_p$. (f) The residual network G_f.

Corollary. Let $G = (V, E)$ be a flow network, f a flow in G, and p an augmenting path in G_f. Then $f' = f + f_p$ is a flow in G with value $|f'| = |f| + |f_p| > |f|$.

A **cut** (S, T) of flow network $G = (V, E)$ is a partition of V into S and $T = V - S$ such that $s \in S$ and $t \in T$. If f is a flow, then the **net flow** across the cut (S, T) is defined to be $f(S,T)$. The **capacity** of the cut (S,T) is $c(S,T)$.

For example, consider the flow network $G = (V, E)$ with flow f from Fig 30.3(b). Let $(\{s, a, b, c, d\}, \{e, t\})$ be a cut in G. Then the flow across the cut is

$$f(b,t) + f(c,e) + f(d,e) = 8 + 5 + 5 + 0 = 18$$

and the capacity is

$$c(b,t) + c(c,e) + c(d,e) = 8 + 8 + 5 = 21$$

Observe that the net flow across a cut can include negative net flows between vertices, but that the capacity of a cut is composed entirely of non-negative values.

Lemma

Let f be a flow in a flow network G with source s and sink t, and let (S, T) be a cut of G. Then, the net flow across (S, T) is $f(S,T) = |f|$.

Proof. It is straight forward to see that

$$f(S,T) = f(S,V) - f(S,S)$$
$$= f(S,V)$$
$$= f(s,V) + f(S-s,V)$$
$$= f(s,V)$$
$$= |f|.$$

Corollary. *Let $G=(V,E)$ be a flow network. Then the value of any flow in is bounded above by the capacity of any cut.*

Proof. Let (S, T) be any cut of G and let f be any flow . Using above lemma and the capacity constraints

$$|f| = f(S,T)$$
$$= \sum_{u \in S} \sum_{v \in T} f(u,v)$$
$$\le \sum_{u \in S} \sum_{v \in T} c(u,v)$$
$$= c(S,T).$$

30.4 Max-flow min-cut Theorem

If f is a flow in a flow network $G=(V,E)$ with source s and sink t, then the following conditions are equivalent :

1. f is a maximum flow in G.
2. The residual network G_f contains no augmenting paths.
3. $|f| = c(S,T)$ for some cut (S, T) of G.

Proof. **(1) ⇒ (2) :** Suppose f is a maximum flow in G but that G_f has an augmenting path p. Then, by Corollary, the flow sum $f' = f + f_p$, where f_p is a flow in G with value strictly greater than $|f|$, contradicting the assumption that f is a maximum flow.

(2) ⇒ (3) : Suppose that G_f has no augmenting path, that is, that G_f contains no path from s to t. Define

$$S = \{v \in V : \text{There exists a path from s to } v \text{ in } G_f\}$$

and $T = V - S$. The partition (S, T) is a cut: we have $s \in S$ and $t \notin S$ because there is no path from s to T in G_f. For each pair of vertices u and v such that $u \in S$ and $v \in T$, we have $f(u,v) = c(u,v)$, since otherwise $(u,v) \in E_f$ and v is in set S. By Lemma, therefore, $|f| = f(S,T) = c(S,T)$.

(3) ⇒ (1) : By Corollary, $|f| \le c(S,T)$ for all cuts (S, T). The condition $|f| = c(S,T)$ thus implies that f is a maximum flow.

30.5 Ford-Fulkerson Algorithm

The Ford-Fulkerson method is iterative. We start with $f(u,v)=0$ for all $u,v \in V$, giving an initial flow of value 0. At each iteration, we increase the flow value by finding an "augmenting path," which we can think of simply as a path from the source s to the sink t along which we can push more flow, and then augmenting the flow along this path. We repeat this process until no augmenting path can be found. The max-flow min-cut theorem will show that upon termination, this process yields a maximum flow.

```
FORD-FULKERSON-METHOD (G, s, t)

1.  initialize flow f to 0
2.  while there exists an augmenting path p
3.      do augment flow f along p
4.  return f
```

The FORD-FULKERSON algorithm simply expands on the FORD-FULKER-SON-METHOD pseudo code given above. Lines 1-3 initialize the flow f to 0. The **while** loop of lines 4-8 repeatedly finds an augmenting path p in G_f and augments flow f along p by the residual capacity $c_f(p)$. When no augmenting paths exist, the flow f is a maximum flow.

```
FORD-FULKERSON (G, s, t)

1.  for each edge (u,v) ∈ E[G]
2.  do f[u,v] ← 0
3.  f[v,u] ← 0
4.  while there exists a path p from s to t in the residual network G_f
5.  do c_f(p) ← min {c_f(u,v) : (u,v) is in p}
6.  for each edge (u,v) in p
7.      do f[u,v] ← f[u,v] + c_f(p)
8.      f[v,u] ← −f[u,v]
```

Analysis of Ford-Fulkerson

The running time of FORD-FULKERSON depends on how the augmenting path p in line 4 is determined. If it is chosen poorly, the algorithm might not even terminate: the value of the flow will increase with successive augmentations, but it need not even converge to the maximum flow value. If the augmenting path is chosen by using a breadth-first search, the algorithm runs in polynomial time.

Example. *Using the definition of a flow, prove that if $(u,v) \notin E$ and $(v,u) \notin E$ then*

$$f(u,v) = f(v,u) = 0$$

Solution. Assume $(u,v) \notin E$ and $(v,u) \notin E$ then by capacity constraint $f(u,v) \le 0$ and $f(v,u) \le 0$. By skew symmetry $f(u,v) = f(v,u) = 0$.

Example. *Given a flow network $G=(V,E)$. Let f_1 and f_2 be functions from $V\times V$ to R. The flow sum f_1+f_2 is the function from $V\times V$ to R defined by*

$$(f_1+f_2)(u,v)=f_1(u,v)+f_2(u,v) \qquad \forall \quad u,v\in V$$

If f_1 and f_2 are flows in G, which of the three flow properties must the flow sum f_1+f_2 satisfy, and which might it violate ?

Solution. **Capacity constraint.** May clearly be violated

Skew symmetry. We have

$$(f_1+f_2)(u,v)=f_1(u,v)+f_2(u,v)$$

$$=-f_1(v,u)-f_2(v,u)$$

$$=-(f_1(v,u)+f_2(v,u))$$

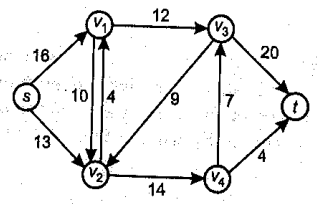
$$=-(f_1+f_2)(v,u)$$

Flow conservation. Let $u\in V-s$ be given. Then

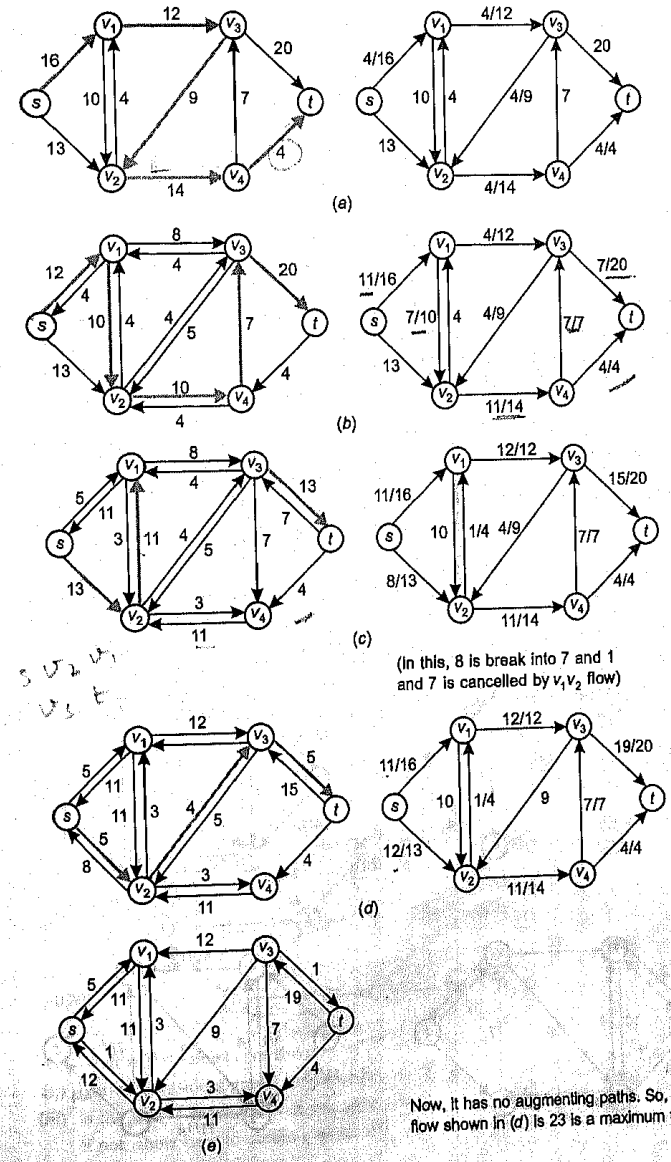
$$\sum_{v\in V}(f_1+f_2)(u,v)=\sum_{v\in V}(f_1(u,v)+f_2(u,v))$$

$$=\sum_{v\in V}f_1(u,v)+\sum_{v\in V}f_2(u,v)$$

Example. *In the flow network illustrated below, each directed edge is labelled with its capacity. Use the Ford-Fulkerson algorithm to find the maximum flow.*

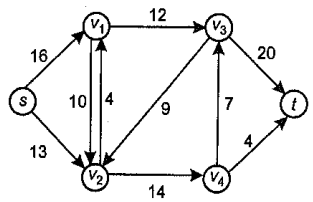


Solution. The left side of each part shows the residual network G_f with a shaded augmenting path p and the right side of each part shows the net flow f.

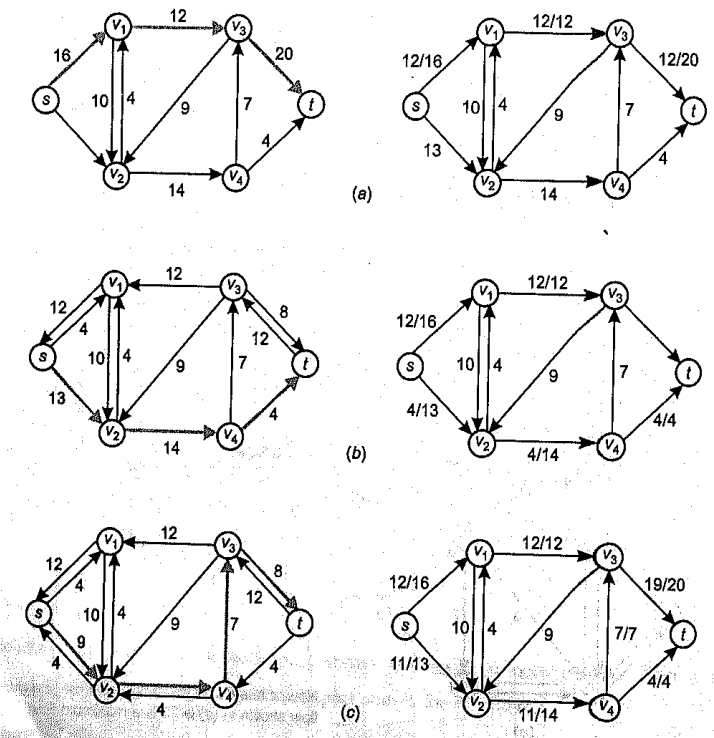


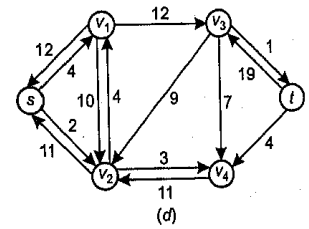
Now, it has no augmenting paths. So, the maximum flow shown in (d) is 23 is a maximum flow.

Example. *Show the execution of the Edmonds-Karp algorithm on the flow network shown in the figure.*



Solution. The bound on FORD-FULKERSON can be improved if we implement the computation of the augmenting path p with a breadth-first search, that is, if the augmenting path is a *shortest path* from s to t in the residual network. We call the Ford-Fulkerson method so implement the Edmonds-Karp algorithm. This algorithm runs in $O(VE^2)$ time.





It has no augmenting paths. So the flow f shown in (c) is therefore a maximum flow =23

Example. *What is the worst-case running time of the Ford-Fulkerson algorithm if all edge capacities are bounded by a constant ?*

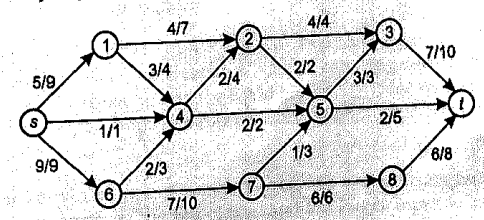
Solution. If every edge capacity is bounded by a constant m, then the maximum flow must be bounded by $O(m)$ and the Ford-Fulkerson algorithm would run in $O(m^2)$ in worst-case.

Example. *Let N be a flow network with n vertices and m edges. Show how to compute an augmenting path with the largest residual capacity in $O((n+m)\log n)$ time.*

Solution. To compute an augmenting path with the largest residual capacity, we use maximum spanning tree algorithm, which is just like a minimum spanning tree algorithm with all the weights multiplied by (−1).

Exercise

1. We are given a flow f in the following flow network. On each edge, the label f/c represents the flow f and the capacity c on the edge :

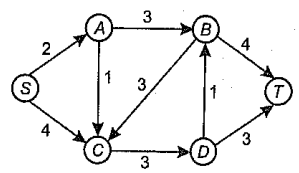


(*i*) What is the value $|f|$ of the flow ?

(*ii*) Draw the residual graph corresponding to f.

(*iii*) Is the flow f maximum ? If yes, state why. If not, show how to obtain a large flow.

2. In the flow network illustrated below, each directed edge is labelled with its capacity. Use the Ford-Fulkerson algorithm to find the maximum flow. The first augmenting path is S-A-C-D-T and the second augmenting path S-A-B-C-D-T.



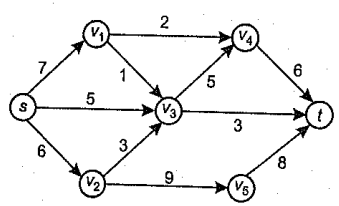
   (*i*) Draw the residual network after we have updated the flow using these two augmenting paths.

   (*ii*) What is the numerical value of the maximum flow. Draw a dotted line through the original graph to represent the minimum cut.

3. Draw a flow network with 9 vertices and 12 edges. Illustrate an execution of the Ford-Fulkerson algorithm on it.

4. Show that, given a maximum flow in a network with m edges, a minimum cut of N can be computed in $O(m)$ time.

5. Illustrate the execution of the Ford-Fulkerson algorithm in the flow network.



CHAPTER 31

Sorting Networks

31.1 Comparison Networks

A comparison network is made of wires and comparators. A comparator, is a device with two inputs, x and y, and two outputs, x' and y', where

$$x' = \min(x, y)$$
$$y' = \max(x, y)$$

Comparison networks differ from RAM's in two important respects. *First*, they can only perform comparisons. Thus, an algorithm such as counting sort cannot be implemented on a comparison network. *Second*, unlike the RAM model, in which operations occur serially - that is, one after another-operations in a comparison network may occur at the same time, or "in parallel."

In comparison networks inputs appear on the left and outputs on the right, with the smaller input value appearing on the top output and the larger input value appearing on the bottom output. We can thus think of a comparator as sorting its two inputs. We shall assume that each comparator operates in $O(1)$ time. In other words, we assume that the time between the appearance of the input values x and y and the production of the output values x' and y' is a constant.

A wire transmits a value from place to place. A comparison network contains n input wires $a_1, a_2, \ldots, a_n$, through which the values to be sorted enter the network, and n output wires $b_1, b_2, \ldots, b_n$ which produce the results computed by the network.

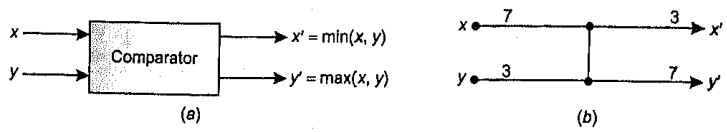


Figure 31.1

Comparison network is a set of comparators interconnected by wires. Under the assumption that each comparator takes unit time, we can define the "running time" of a comparison network. Running time of comparator can define in terms of **depth**. More formally, we define the **depth** of a wire as follows. An input wire of a comparison network has depth 0. Now, if a comparator has two input wires with depths d_x and d_y, then its output wires have depth $\max(d_x, d_y)+1$.

A *sorting network* is a comparison network for which the output sequence is monotonically increasing (that is, $b_1 \le b_2 \le \ldots \le b_n$) for every input sequence. Thus not every comparison network is a sorting network.

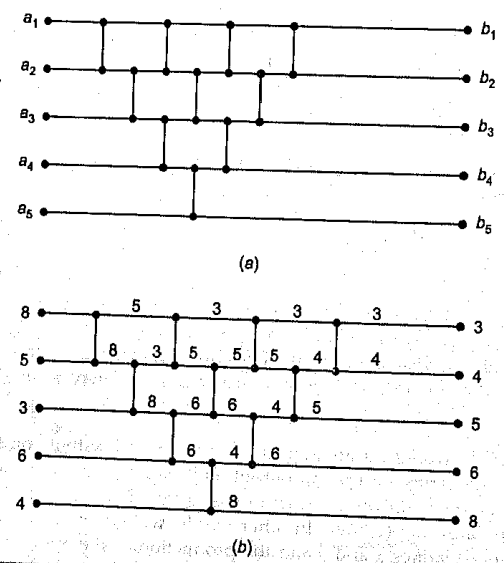


Figure 31.2 A sorting network based on insertion sort.

A comparison network is like a procedure in that it specifies how comparisons are to occur, but it is unlike a procedure in that its physical size depends on the number of inputs and outputs. Therefore, we shall actually be describing "families" of comparison networks. Any sorting network on n inputs has depth at least lg n the number of comparators in any sorting network is at least $\Omega(n \lg n)$.

The zero-one Principle

The *zero-one principle* says that if a sorting network works correctly when each input is drawn from the set {0,1}, then it works correctly on arbitrary input numbers. (The numbers can be integers, reals, or, in general, any set of values from any linearly ordered set.) The proof of the zero-one principle relies on the notion of a monotonically increasing function.

Lemma

If a comparison network transforms the input sequence $a = \langle a1, a2, \ldots, an \rangle$ into the output sequence $b = \langle b_1, b_2, \ldots, b_n \rangle$, then for any monotonically increasing function f, the network transforms the input sequence $f(a) = \langle f(a_1), f(a_2), \ldots, f(a_n) \rangle$ into the output sequence $f(b) = \langle f(b_1), f(b_2), \ldots, f(b_n) \rangle$.

Proof. If f is a monotonically increasing function, then a single comparator with inputs $f(x)$ and $f(y)$ produces outputs $f(\min(x,y))$ and $f(\max(x,y))$. Let us consider a comparator whose input values are x and y. The upper output of the comparator is $\min(x,y)$ and the lower output is

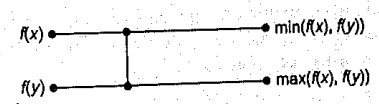


Figure 31.3

$\max(x,y)$. Suppose we now apply $f(x)$ and $f(y)$ to the inputs of the comparator, as is shown in Fig. 31.3. The operation of the comparator yields the value $\min(f(x), f(y))$ on the upper output and the value $\max(f(x), f(y))$ on the lower output. Since f is monotonically increasing, $x \le y$ implies $f(x) \le f(y)$. Thus, we have the identities

$$\min(f(x), f(y)) = f(\min(x,y))$$
$$\max(f(x), f(y)) = f(\max(x,y))$$

Thus, the comparator produces the values $f(\min(x,y))$ and $f(\max(x,y))$ when $f(x)$ and $f(y)$ are its inputs, which completes the proof.

(Zero-one Principle)

k with n inputs sorts all 2^n possible sequences of 0's and 1's correctly, and then rary numbers correctly.

he contradiction i.e. the network sorts all zero-one sequences, but there ry numbers that the network does not correctly sort. That is, there exists $\ldots, a_n\rangle$ containing elements a_i and a_j such that $a_i < a_j$, but the network utput sequence. We define a monotonically increasing function f as

$$f(x) = \begin{cases} 0 & \text{if } x \le a_i \\ 1 & \text{if } x > a_i \end{cases}$$

ny monotonically increasing function f, the network transforms the input $a_2), \ldots, f(a_n)\rangle$ into the output sequence $f(b) = \langle f(b_1), f(b_2), \ldots, f(b_n)\rangle$. By work places $f(a_j)$ before $f(a_i)$ in the output sequence when input. But since $f(a_j) = 1$ and $f(a_i) = 0$, we obtain the contradiction that he zero-one sequence $\langle f(a_1), f(a_2), \ldots, f(a_n)\rangle$ correctly. Hence theorem is

ng Network

ther monotonically increases and then monotonically decreases, or else nd then monotonically increases is called bitonic sequence. For example,) and $\langle 8,7,5,2,4,6\rangle$ are both bitonic. The **bitonic sorter** is a comparison sequences of 0's and 1's.

comprised of several stages, each of which is called a **half-cleaner**. Each on network of depth 1 in which input line i is compared with line $1 + n/2$ ume that n is even.)

quence of 0's and 1's is applied as input to a half-cleaner, the half-cleaner uence in which smaller values are in the top half, larger values are in the halves are bitonic. In fact, at least one of the halves is clean-consisting of and it is from this property that we derive the name "**half-cleaner.**" (Note es are bitonic.)

combining half-cleaners, we can build a bitonic sorter, which is a network uences. The first stage of BITONIC-SORTER[n] consists of HALF-CLEANER[n], bitonic sequences of half the size such that every element in the top half is at ry element in the bottom half. Thus, we can complete the sort by using two SORTER [$n/2$] to sort the two halves recursively.

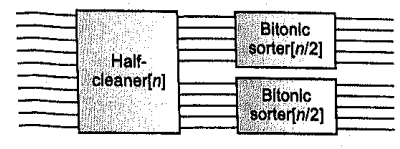


Figure 31.4

The depth D(n) of BITONIC-SORTER[n] is given by the recurrence whose solution is $D(n) = \lg n$.

31.3 Merging Network

We modify BITONIC-SORTER[n] to create the merging network MERGER[n]. Merging networks can merge two sorted input sequences into one sorted output sequence. The merging network is based on the following intuition. Given two sorted sequences, if we reverse the order of the second sequence and then concatenate the two sequences, the resulting sequence is bitonic. For example, given the sorted zero-one sequences $X = 00000111$ and $Y = 00001111$, we reverse Y to get $Y^R = 11110000$. Concatenating X and Y^R yields 0000011111110000, which is bitonic.

The sorting network SORTER[n] uses the merging network to implement a parallel version of merge sort. The first stage of SORTER[n] consists of $n/2$ copies of MERGER[2] that work in parallel to merge pairs of 1-element sequences to produce sorted sequences of length 2. The second stage consists of $n/4$ copies of MERGER [4] that merge pairs of these 2-element sorted sequences to produce sorted sequences of length 4. In general, for $k = 1, 2, \ldots, \lg n$, stage k consists of $n/2^k$ copies of MERGER[2^k] that merge pairs of the $2^k - 1$-element sorted sequences to produce sorted sequences of length 2^k. At the final stage, one sorted sequence consisting of all the input values is produced. This sorting network can be shown by induction to sort zero-one sequences, and consequently, by the zero-one principle, it can sort arbitrary values.

The depth of SORTER[n] is given by the recurrence

$$D(n) = \begin{cases} 0 & \text{if } n = 1 \\ D(n/2) + \lg n & \text{if } n = 2^k \text{ and } k \ge 1 \end{cases}$$

whose solution is $D(n) = \theta(\lg^2 n)$. Thus, we can sort n numbers in parallel in $O(\lg^2 n)$ time.

Exercise

1. Prove that any sorting network, on n input has a depth alleast $\lg n$
2. Prove that the number of comparators in any sorting network is $\Omega(n \lg n)$.

3. Let n be an exact power of 2. Show how to construct an n-input, n-output comparison network of depth $\lg n$ in which the top output wire always carries the minimum input value and the bottom output wire always carries the maximum input value.
4. State and prove an analog of the zero-one principle for a decision-tree model.
5. How many zero-one bitonic sequences of length n are there ?
6. How many different zero-one input sequences must be applied to the input comparison network to verify that it is merging network ?
7. Prove that any merging network, regardless of the order of inputs, requires $\Omega(n \lg n)$ comparators.
8. Show that any network that can merge 1 item with $n-1$ sorted items to produce a sorted sequence of length n must have depth atleast $\lg n$.

CHAPTER

32

Algorithms for Parallel Computers

32.1 Parallel Computing

Traditionally, software has been written for serial computation :

- To be run on a single computer having a single Central Processing Unit (CPU) ;
- A problem is broken into a discrete series of instructions.
- Instructions are executed one after another.
- Only one instruction may execute at any moment in time.

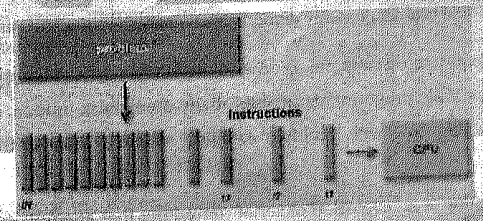


Figure 32.1

In the simplest sense, ***parallel computing*** is the simultaneous use of multiple compute resources to solve a computational problem.

- To be run using multiple CPUs
- A problem is broken into discrete parts that can be solved concurrently
- Each part is further broken down to a series of instructions
- Instructions from each part execute simultaneously on different CPUs

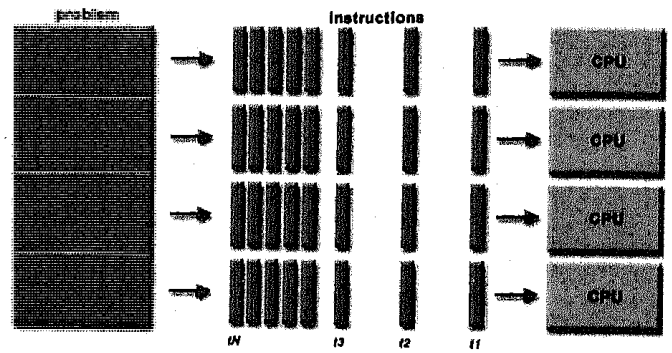


Figure 32.2

The compute resources can include :

- A single computer with multiple processors ;
- An arbitrary number of computers connected by a network ;
- A combination of both.

The computational problem usually demonstrates characteristics such as the ability to be :

- Broken apart into discrete pieces of work that can be solved simultaneously ;
- Execute multiple program instructions at any moment in time ;
- Solved in less time with multiple compute resources than with a single compute resource.

Parallel computing is an evolution of serial computing that attempts to emulate what has always been the state of affairs in the natural world : many complex, interrelated events happening at the same time, yet within a sequence.

Some examples :

- Planetary and galactic orbits
- Weather and ocean patterns
- Automobile assembly line
- Daily operations within a business
- Building a shopping mall

32.2 Why Use Parallel Computing ?

The primary reasons for using parallel computing :

- Save time
- Solve larger problems
- Provide concurrency (do multiple things at the same time)

Other reasons might include :

- *Taking advantage of non-local resources.* using available compute resources on a wide area network, or even the Internet when local compute resources are scarce.
- *Cost savings.* using multiple "cheap" computing resources instead of paying for time on a supercomputer.
- *Overcoming memory constraints.* single computers have very finite memory resources. For large problems, using the memories of multiple computers may overcome this obstacle.

Limits to serial computing both physical and practical reasons pose significant constraints to simply building ever faster serial computers :

- *Transmission speeds.* the speed of a serial computer is directly dependent upon how fast data can move through hardware. Absolute limits are the speed of light (30 cm/nanosecond) and the transmission limit of copper wire (9 cm/nanosecond). Increasing speeds necessitate increasing proximity of processing elements.
- *Limits to miniaturization.* processor technology is allowing an increasing number of transistors to be placed on a chip. However, even with molecular or atomic-level components, a limit will be reached on how small components can be.
- *Economic limitations.* it is increasingly expensive to make a single processor faster. Using a larger number of moderately fast commodity processors to achieve the same (or better) performance is less expensive.

32.3 von Neumann Architecture

For over 40 years, virtually all computers have followed a common machine model known as the von Neumann computer. Named after the Hungarian mathematician **John von Neumann.** A von Neumann computer uses the stored-program concept. The CPU executes a stored program that specifies a sequence of read and write operations on the memory.

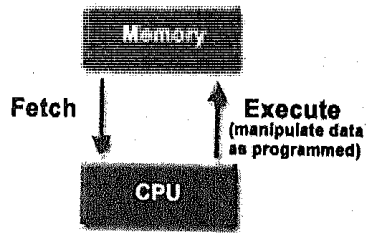


Figure 32.3

Basic design :

- Memory is used to store both program and data instructions
- Program instructions are coded data which tell the computer to do something
- Data is simply information to be used by the program
- A central processing unit (CPU) gets instructions and/or data from memory, decodes the instructions and then *sequentially* performs them.

32.4 Flynn's Classical Taxonomy

- There are different ways to classify parallel computers. One of the more widely used classifications, in use since 1966, is called Flynn's Taxonomy.
- Flynn's taxonomy distinguishes multi-processor computer architectures according to how they can be classified along the two independent dimensions of **Instruction** and **Data.** Each of these dimensions can have only one of two possible states : **Single** or **Multiple.**
- The matrix below defines the 4 possible classifications according to Flynn.

SISD Single Instruction, Single Data	SIMD Single Instruction, Multiple Data
MISD Multiple Instruction, Single Data	MIMD Multiple Instruction, Multiple Data

Single Instruction, Single Data (SISD)

- A serial (non-parallel) computer
- *Single instruction.* only one instruction stream is being acted on by the CPU during any one clock cycle
- *Single data.* only one data stream is being used as input during any one clock cycle
- Deterministic execution
- This is the oldest and until recently, the most prevalent form of computer Examples : mainframes most PCs, single CPU workstations

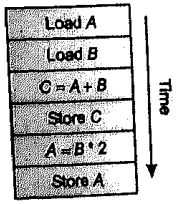


Figure 32.4

Single Instruction, Multiple Data (SIMD)

- A type of parallel computer
- *Single instruction.* All processing units execute the same instruction at any given clock cycle
- *Multiple data.* Each processing unit can operate on a different data element
- This type of machine typically has an instruction dispatcher, a very high-bandwidth internal network, and a very large array of very small-capacity instruction units.
- Best suited for specialized problems characterized by a high degree of regularity,such as **image processing.**
- Synchronous (lockstep) and deterministic execution
- *Two varieties.* Processor Arrays and Vector Pipelines
- *Examples.*
  Processor Arrays. Connection Machine CM-2, Maspar MP-1, MP-2
  Vector Pipelines. IBM 9000, Cray C90, Fujitsu VP, NEC SX-2, Hitachi S820

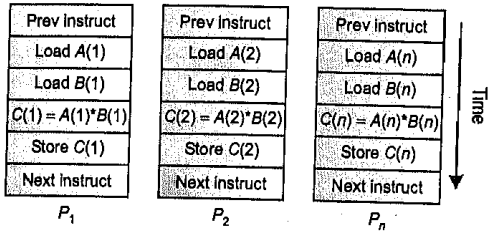


Figure 32.5

Multiple Instruction, Single Data (MISD)

- A single data stream is fed into multiple processing units.
- Each processing unit operates on the data independently via independent instruction streams.
- Few actual examples of this class of parallel computer have ever existed. One is the experimental Carnegie-Mellon C.mmp computer (1971).
- Some conceivable uses might be :

  multiple frequency filters operating on a single signal stream

  multiple cryptography algorithms attempting to crack a single coded message

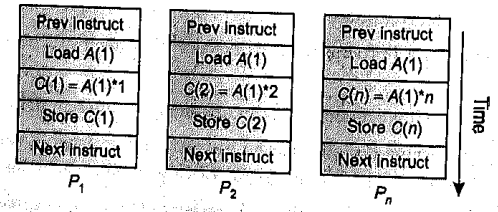


Figure 32.6

Multiple Instruction, Multiple Data (MIMD)

- Currently, the most common type of parallel computer. Most modern computers fall into this category.
- *Multiple Instruction*, every processor may be executing a different instruction stream.
- *Multiple Data*, every processor may be working with a different data stream.

- Execution can be synchronous or asynchronous, deterministic or non-deterministic.
- *Examples*, most current supercomputers, networked parallel computer "grids" and multi-processor SMP computers - including some types of PCs.

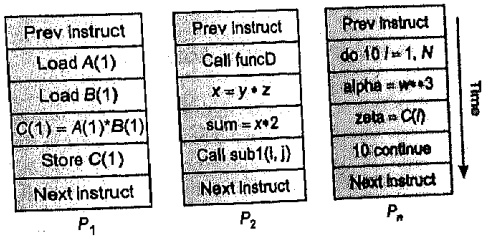


Figure 32.7

32.5 Parallel Algorithms

Parallel algorithms are the algorithms that perform more than one operation at a time. The study of parallel algorithms we must choose an appropriate model for parallel computing. The random-access machine, or RAM is serial rather than parallel. The parallel models are sorting **networks and circuits or PRAM** (pronounced "PEE-ram"). Many parallel algorithms for arrays, lists, trees, and graphs can be easily described in the PRAM model. The basic architecture of the parallel random-access machine (PRAM) as follows :

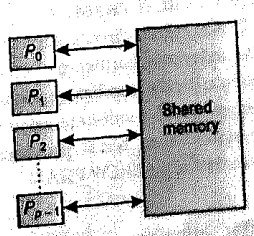


Figure 32.8

There are p ordinary (serial) processors $P_0, P_1, \ldots, P_{p-1}$ that have as storage a shared, global memory. All processors can read from or write to the global memory "in parallel" (at the same time). The processors can also perform various arithmetic and logical operations in parallel.

The key assumption in the PRAM model is that running time can be measured as the number of parallel memory accesses an algorithm performs. The running time of a parallel

algorithm depends on the number of processors executing the algorithm as well as the size of the problem input. Therefore, we take both time and processor count when analyzing PRAM algorithms; this contrasts with serial algorithms, in whose analysis we have focused mainly on time.

32.6 Concurrent Versus Exclusive Memory Accesses

A **concurrent-read** algorithm is a PRAM algorithm during whose execution multiple processors can read from the same location of shared memory at the same time. An **exclusive-read** algorithm is a PRAM algorithm in which no two processors ever read the same memory location at the same time. Similar we can divide PRAM algorithms into **concurrent-write** and **exclusive-write** algorithms.

Commonly used abbreviations for the types of algorithms are

EREW exclusive read and exclusive write,
CREW concurrent read and exclusive write,
ERCW exclusive read and concurrent write, and
CRCW concurrent read and concurrent write.

(These abbreviations are usually pronounced not as words but rather as strings of letters.)

A PRAM that supports only EREW algorithms is called an **EREW PRAM**, and one that supports CRCW algorithms is called a **CRCW PRAM**. A CRCW PRAM can, of course, execute EREW algorithms, but an EREW PRAM cannot directly support the concurrent memory accesses required in CRCW algorithms. The underlying hardware of an EREW PRAM is relatively simple, and therefore fast, because it needn't handle conflicting memory reads and writes. A CRCW PRAM requires more hardware support if the unit-time assumption is to provide a reasonably accurate measure of algorithmic performance, but it provides a programming model that is arguably more straightforward than that of an EREW PRAM.

32.7 List Ranking by Pointer Jumping

We can store a list in a PRAM much as we store lists in an ordinary RAM. To operate on list objects in parallel, however, it is convenient to assign a "responsible" processor to each object. We shall assume that there are as many processors as list objects, and that the ith processor is responsible for the ith object. For example, shows a linked list consisting of the sequence of objects $\langle 2,4,6,1,8,5\rangle$. Since there is one processor per list object, every object in the list can be operated on by its responsible processor in $O(1)$ time.

Suppose that we are given a singly linked list L with n objects and wish to compute, for each object in L, its distance from the end of the list. More formally, if next is the pointer field, we wish to compute a value $d[i]$ for each object i in the list such that

$$d[i]=\begin{cases} 0 & \text{if next}[i]=\text{NIL} \\ d[\text{next}[i]]+1 & \text{if next}[i]\neq\text{NIL}\end{cases}$$

We call the problem of computing the d values the **list-ranking problem.**

One solution to the list-ranking problem is simply to propagate distances back from the end of the list. This method takes $\Theta(n)$ time, since the kth object from the end must wait for the $k-1$ objects following it to determine their distances from the end before it can determine its own. This solution is essentially a serial algorithm.

An efficient parallel solution, requiring only $O(\lg n)$ time, is given by the following pseudo code.

```
LIST-RANK (L)
1.  for each processor i, in parallel
2.      do if next[i] = NIL
3.          then d [i] ← 0
4.          else d [i] ← 1
5.  while there exists an object i such that next[i] ≠ NIL
6.      do for each processor i, in parallel
7.          do if next[i] ≠ NIL
8.              then d [i] ← d [i] + d[next[i]]
9.                  next[i] ← next[next[i]]
```

The idea implemented by line 9, in which we set next[i] ← next[next[i]] for all non-nil pointers next[i], is called **pointer jumping**. Note that the pointer fields are changed by pointer jumping, thus destroying the structure of the list. If the list structure must be preserved, then we make copies of the next pointers and use the copies to compute the distances.

Parallel prefix on a list

A **prefix computation** is defined in terms of a binary, associative operator $\otimes$. The computation takes as input a sequence $\langle x_1, x_2, \ldots, x_n\rangle$ and produces as output a sequence $\langle y_1, y_2, \ldots, y_n\rangle$ such that $y_1 = x_1$ and

$$\begin{aligned} y_k &= y_{k-1}\otimes x_k \\ &= x_1\otimes x_2\otimes \ldots \otimes x_k \end{aligned}$$

For $k=2,3,\ldots,n$. In other words, each y_k is obtained by "multiplying" together the first k elements of the sequence of x_k-hence, the term "prefix."

```
LIST-PREFIX(L)
for each processor i, in parallel
    do y[i] ← x[i]
while there exists an object i such that next[i] ≠ NIL
    do for each processor i, in parallel
        do if next[i] ≠ NIL
            then y[next[i]] ← y[i] ⊗ y[next[i]]
                next[i] ← next[next[i]]
```

In List-Rank, processor i updates $d[i]$ - its own d value - whereas in LIST-PREFIX, processor i updates $y[\text{next}[i]]$ - another processor's y value.

32.8 The Euler-Tour Technique

It is used for finding the depth of a node in a binary tree. To store binary trees in a PRAM, we use a simple binary-tree representation. Each node i has fields parent[i], left[i], and right[i], which point to node i's parent, left child, and right child, respectively. Let us assume that each node is identified by a non-negative integer.

An **Euler tour** of a graph is a cycle that traverses each edge exactly once, although it may visit a vertex more than once. A connected, directed graph has an Euler tour if and only if for all vertices v, the in-degree of v equals the out-degree of v. Since each undirected edge (u, v) in an undirected graph maps to two directed edges (u, v) and (v, u) in the directed version, the directed version of any connected, undirected graph – and therefore of any undirected tree – has an Euler tour.

To compute the depths of nodes in a binary tree T, we first form an Euler tour of the directed version of T (viewed as an undirected graph). The tour corresponds to a walk of the tree and is represented by a linked list running through the nodes of the tree. Rules for connecting various processors of a node to a another node is as follows :

- "A" processor of a node will point to "A" processor of its left node, if it exists, and otherwise points to a "B" processor of the same node.
- "B" processor of a node will point to "A" processor of its right child, if it exists, and otherwise points to a "C" processor of the same node.
- "C" processor of a node will point to the "B" processor of its parent if it is a left child and to the "C" processor of its parent if it is a right child. The root's C processor points to NIL.

Thus, the head of the linked list formed by the Euler tour is the root's A processor, and the tail is the root's C processor. Given the pointers composing the original tree, an Euler tour can be constructed in $O(1)$ time and is represented in the Fig. 32.9.

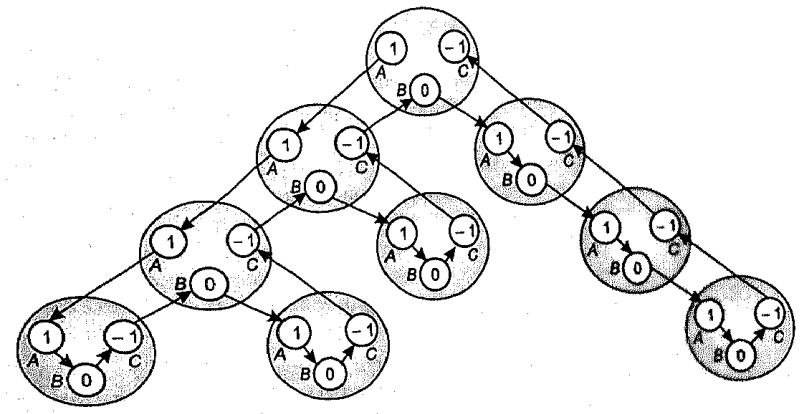


Figure 32.9 Using the Euler-Tour technique to compute the depth of each node in a binary tree.

Once we have the linked list representing the Euler tour of T, we place a 1 in each A processor, a 0 in each B processor, and –1 in each C processor. We then perform a parallel prefix computation using ordinary addition as the associative operation, which is represented in Fig. 32.10.

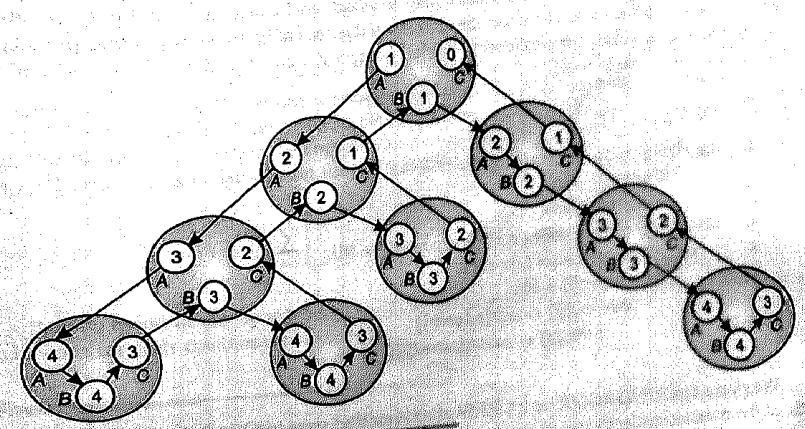


Figure 32.10 Result of the parallel prefix computation.

We claim that after performing the parallel prefix computation, the depth of each node resides in the node's C processor. The numbers are placed into the A, B, and C processors in such a way that the net effect of visiting a subtree is to add 0 to the running sum. The A processor of each node i contributes 1 to the running sum in i's left subtree, reflecting the depth of i's left child being one greater than the depth of i. The B processor contributes 0 because the depth of node i's left child equals the depth of node i's right child. The C processor contributes -1, so that from the perspective of node i's parent, the entire visit to the subtree rooted at node i has no effect on the running sum.

The list representing the Euler tour can be computed in $O(1)$ time. It has $3n$ objects, and thus the the parallel prefix computation takes only $O(\lg n)$ time. Thus, the total amount of time to compute all node depths is $O(\lg n)$. Because no concurrent memory accesses are needed, the algorithm is an EREW algorithm.

32.9 CRCW Algorithms Versus EREW Algorithms

The debate about whether or not concurrent memory accesses should be provided by the hardware of a parallel computer is a messy one. Some argue that hardware mechanisms to support CRCW algorithms are too expensive and used too infrequenly to be justified. Others complain that EREW PRAM's provide too restrictive a programming model. The answer to this debate probably lies somewhere in the middle, and various compromise models have been proposed.

We shall show that there are problems on which a CRCW algorithm outperforms the best possible EREW algorithm. For the problem of finding the identities of the roots of trees in a forest, concurrent reads allow for a faster algorithm. For the problem of finding the maximum element in an array, concurrent writes permit a faster algorithm.

Suppose we are given a forest of binary trees in which each node i has a pointer parent[i] to its parent, and we wish each node to find the identity of the root of its tree. Associating processor i with each node i in a forest F, the following pointer-jumping algorithm stores the identity of the root of each node i's tree in root[i].

```
FIND-ROOTS (F)
for each processor i, in parallel
    do if parent[i] = NIL
        then root[i] ← i
while there exists a node i such that parent[i] ≠ NIL
    do for each processor i, in parallel
        do if parent[i] ≠ NIL
            then root[i] ← root[parent[i]]
                parent[i] ← parent[parent[i]]
```

We claim that FIND-ROOTS is a CREW algorithm that runs in $O(\lg d)$ time, where d is the depth of the maximum-depth tree in the forest.

We know that CRCW algorithms can solve some problems more quickly than can EREW algorithms. Moreover, any EREW algorithm can be executed on a CRCW PRAM. Thus, the CRCW model is strictly more powerful than the EREW model.

32.10 Brent's Theorem

Brent's theorem shows how we can efficiently simulate a combinational circuit by a PRAM. A ***combinational circuit*** is an acyclic network of ***combinational elements***. Each combinational element has one or more inputs, and assumes that each element has exactly one output. The number of inputs is the ***fan-in*** of the element, and the number of places to which its output feeds is its ***fan-out***. The ***size*** of a combinational circuit is the number of combinational elements that it contains. The number of combinational elements on a longest path from an input of the circuit to an output of a combinational element is the element's ***depth***. The ***depth*** of the entire circuit is the maximum depth of any of its elements.

Theorem

Any depth-d, size-n combinational circuit with bounded fan-in can be stimulated by a p-processor CREW algorithm in $O(n/p+d)$ time.

Proof. An element cannot be simulated until the outputs from any elements that feed it have been computed. Concurrent reads are employed whenever several combinational elements being simulated in parallel require the same value.

Since all elements at depth 1 depend only on circuit inputs, they are the only ones that can be simulated initially. Once they have been simulated, all elements at depth 2 can be simulated, and so forth, until we finish with all elements at depth d. The key idea is that if all elements from depths 1 to i have been simulated, we can simulate any subset of elements at depth $i+1$ in parallel, since their computations are independent of one another.

For $i=1,2,\ldots,d$, let n_i be the number of elements at depth i in the circuit. Thus,

Consider the n_i combinational elements at depth i. By grouping them into $\lceil n_i/p \rceil$ groups, where the first $\lfloor n_i/p \rfloor$ groups have p elements each and the leftover elements, if any, are in the last group, the PRAM can simulate the computations performed by these combinational elements in $O(\lceil n_i/p \rceil)$ time. The total simulation time is therefore on the order of

$$\sum_{i=1}^{d} \left\lceil \frac{n_i}{p} \right\rceil \le \sum_{i=1}^{d} \left(\frac{n_i}{p} + 1 \right) = \frac{n}{p} + d$$

Brent's theorem can be extended to EREW simulations when a combinational circuit has $O(1)$ fan-out for each combinational element.

Theorem

Any depth-d, size-n combinational circuit with bounded fan-in and fan-out can be simulated on a p-processor EREW PRAM in $O(n/p+d)$ time.

Proof. We perform a simulation similar to that in the proof of Brent's theorem. The only difference is in the simulation of wires, which is where requires concurrent reading. For the EREW simulation, after the output of a combinational element is computed, it is not directly read by processors requiring its value. Instead, the output value is copied by the processor simulating the element to the $O(1)$ inputs that require it. The processors that need the value can then read it without interfering with each other.

Exercise

1. Discuss Brent's Theorem in detail.
2. Give an EREW algorithm to perform prefix computation on an array $A[1..n]$. Do not use pointers but use index property.
3. Merging of two sorted sequences each of length m can be computed in $O(\lg m)$ time using m CREW PRAM processors.
4. Given an $O(\lg n)$ EREW alogorithm that determines for each object in an n-object list whether it is the middle ($\lfloor n/2 \rfloor$) object.

CHAPTER

33

Matrix Operations

33.1 Introduction

A **matrix** is a rectangular array of numbers.

For example, $A = \begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{bmatrix}$

is a 3×3 matrix $A = (a_{ij})$ where for $i = 1,2,3$ and $j = 1,2,3$ the element of the matrix in row i and column j is a_{ij}. We use uppercase letters to denote matrices and corresponding subscripted lowercase letters to denote their elements. The set of all $m \times n$ matrices with real-valued entries is denoted $R^{m \times n}$. In general, the set of $m \times n$ matrices with entries drawn from a set S is denoted $S^{m \times n}$.

The **transpose** of a matrix A is the matrix A^T obtained by exchanging the rows and columns of A.

$$(A^T)_{ij} = A_{ji}$$

$$\begin{bmatrix} 0 & 4 \\ 7 & 0 \\ 3 & 1 \end{bmatrix}^T = \begin{bmatrix} 0 & 7 & 3 \\ 4 & 0 & 1 \end{bmatrix}$$

A **vector** is a one-dimensional array of numbers. The standard form of a vector to be as a **column vector** equivalent to an $n \times 1$ matrix; the corresponding **row vector** is obtained by taking the transpose

The **unit vector** e_i is the vector whose ith element is 1 and all of whose other elements are 0. A **zero matrix** is a matrix that's every entry is 0. A **diagonal matrix** has $a_{ij} = 0$ whenever $i \neq j$. Because all of the off-diagonal elements are zero, listing the elements along the diagonal can specify the matrix. The $n \times n$ **identity matrix** I_n is a diagonal matrix with 1's along the diagonal :

$$I_n = \text{diag}(1, 1, \ldots, 1)$$

A **tridiagonal matrix** T is one for which $t_{ij} = 0$ if $|i - j| > 1$. Non zero entries appear only on the main diagonal, immediately above the main diagonal ($t_{i,i+1}$ for $i = 1, 2, \ldots, n-1$), or immediately below the main diagonal ($t_{i+1,i}$ for $i = 1, 2, \ldots, n-1$) :

$$T = \begin{bmatrix} t_{11} & t_{12} & 0 & 0 & \cdots & 0 & 0 & 0 \\ t_{21} & t_{22} & t_{23} & 0 & \cdots & 0 & 0 & 0 \\ 0 & t_{32} & t_{33} & t_{34} & \cdots & 0 & 0 & 0 \\ \vdots & \vdots & \vdots & \vdots & \ddots & \vdots & \vdots & \vdots \\ 0 & 0 & 0 & 0 & \cdots & t_{n-2,n-2} & t_{n-2,n-1} & 0 \\ 0 & 0 & 0 & 0 & \cdots & t_{n-1,n-2} & t_{n-1,n-1} & t_{n-1,n} \\ 0 & 0 & 0 & 0 & \cdots & 0 & t_{n,n-1} & t_{n,n} \end{bmatrix}$$

An **upper-triangular matrix** U is one for which $u_{ij} = 0$ if $i > j$. All entries below the diagonal are zero :

$$U = \begin{bmatrix} u_{11} & u_{12} & \cdots & u_{1n} \\ 0 & u_{22} & \cdots & u_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & u_{nn} \end{bmatrix}$$

An upper-triangular matrix is **unit upper-triangular** if it has all 1's along the diagonal. A **lower-triangular matrix** L is one for which $l_{ij} = 0$ if $i < j$. All entries above the diagonal are zero :

$$L = \begin{bmatrix} l_{11} & 0 & \cdots & 0 \\ l_{21} & l_{22} & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ l_{n1} & l_{n2} & \cdots & l_{nn} \end{bmatrix}$$

A lower-triangular matrix is **unit lower-triangular** if it has all 1's along the diagonal. A **permutation matrix** P has exactly one 1 in each row or column, and 0's elsewhere. An example of a permutation matrix is

$$P = \begin{bmatrix} 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 \\ 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 & 0 \end{bmatrix}$$

Such a matrix is called a permutation matrix because multiplying a vector x by a permutation matrix has the effect of permuting (rearranging) the elements of x.

A **symmetric matrix** A satisfies the condition $A = A^T$

33.2 Operations on Matrices

1. Matrix addition & subtraction

We define **matrix addition** as follows. If $A = (a_{ij})$ and $B = (b_{ij})$ are $m \times n$ matrices, then their **matrix sum** $C = (c_{ij}) = A + B$ is the $m \times n$ matrix defined by

$c_{ij} = a_{ij} + b_{ij}$ for $i = 1, 2, \ldots, m$ and $j = 1, 2, \ldots, n$. That is, matrix addition is performed component wise.

$$\begin{bmatrix} 0 & 4 \\ 7 & 0 \\ 3 & 1 \end{bmatrix} + \begin{bmatrix} 1 & 2 \\ 2 & 3 \\ 0 & 4 \end{bmatrix} = \begin{bmatrix} 1 & 6 \\ 9 & 3 \\ 3 & 5 \end{bmatrix}$$

A zero matrix is the identity for matrix addition :

$$A + 0 = A = 0 + A$$

Properties of matrix addition

1. **commutative.** $A + B = B + A$
2. **associative.** $(A + B) + C = A + (B + C)$, so we can write as $A + B + C$
3. $A + 0 = 0 + A = A$; $A - A = 0$
4. $(A + B)^T = A^T + B^T$

We can multiply a number (scalar) by a matrix by multiplying every entry of the matrix by the scalar.

$$(-2)\begin{bmatrix} 1 & 6 \\ 9 & 3 \\ 6 & 0 \end{bmatrix} = \begin{bmatrix} -2 & -12 \\ -18 & -6 \\ -12 & 0 \end{bmatrix}$$

$$(\alpha + \beta) A = \alpha A + \beta A \,;\; (\alpha\beta) A = (\alpha)(\beta A)$$

$$\alpha (A + B) = \alpha A + \alpha B$$

$$0 \cdot A = 0 \,;\; 1 \cdot A = 0$$

As a special case, we define the negative of a matrix $A=(a_{ij})$ to be $-1A=-A$, so that the ijth entry of $-A$ is $-a_{ij}$. Thus,

$$A+(-A)=0$$
$$=(-A)+A$$

Given this definition, we can define **matrix subtraction** as the addition of the negative of a matrix :

$$A-B=A+(-B)$$

2. Multiplication of Matrices

Consider two matrices **A** and **B** with the following characteristics : the number of columns in **A** equals the number of rows in **B**. These are conformable with respect to one another, and they can be multiplied together to form a new matrix **Z**.

The expression

$$z_{ij}=a_{i1}*b_{1j}+a_{i2}*b_{2j}+a_{i3}*b_{3j}+\ldots a_{im}*b_{nj}$$

means "add the products obtained by multiplying elements in each i row of matrix **A** by elements in each j column of matrix **B**". Fig illustrates what we mean by this statement.

$$A=\begin{bmatrix}4 & 1 & 9\\ 6 & 2 & 8\\ 7 & 3 & 5\\ 11 & 10 & 12\end{bmatrix} \qquad B=\begin{bmatrix}2 & 9\\ 5 & 12\\ 8 & 10\end{bmatrix} \qquad Z=A*B=\begin{bmatrix}85 & 138\\ 86 & 158\\ 69 & 149\\ 168 & 339\end{bmatrix}$$

$$z_{ij}=a_{i1}*b_{1j}+a_{i2}*b_{2j}+a_{i3}*b_{3j}+\ldots a_{im}*b_{nj}$$

$z_{11}=4*2+1*5+9*8=85$ $\qquad$ $z_{12}=4*9+1*12+9*10=138$

$z_{21}=6*2+2*5+8*8=86$ $\qquad$ $z_{22}=6*9+2*12+8*10=158$

$z_{31}=7*2+3*5+5*8=69$ $\qquad$ $z_{32}=7*9+3*12+5*10=149$

$z_{41}=11*2+10*5+12*8=168$ $\qquad$ $z_{42}=11*9+10*12+12*10=339$

The order in which we multiply terms does matter. The reason for this is that we need to multiply row elements by column elements and one by one. Therefore $A*B$ and $B*A$ can produce different results. We say "can produce" because there exist special cases in which the operation is commutative (order does not matter). An example of this is when we deal with diagonal matrices.

3. Multiplication and Division of Matrices by a Scalar

The rules for multiplication and division of a matrix by a scalar are similar. Since multiplying a number x by $1/c$ is the same as dividing x by c, lets consider these operations at once.

If all elements of matrix A are multiplied by a scalar c to construct matrix Z, hence $z_{ij}=c*a_{ij}$. Similarly dividing matrix A by c gives $z_{ij}=(1/c)*a_{ij}$. The expression

$$z_{ij}=c*a_{ij}$$

means "multiply each element in row i column j times c", and the expression

$$z_{ij}=1/c*a_{ij}=a_{ij}/c$$

means "divide each element in row i column j by c". These two operations are shown in below, where $c=2$.

$$A=\begin{bmatrix}3 & 6\\ 5 & 8\\ -2 & 9\end{bmatrix} \quad Z=cA=\begin{bmatrix}6 & 12\\ 10 & 16\\ -4 & 18\end{bmatrix} \quad Z=\frac{A}{c}=\begin{bmatrix}1.5 & 3\\ 2.5 & 4\\ -1 & 4.5\end{bmatrix}$$

$z_{ij}=c*a_{ij}$ $\qquad$ $z_{ij}=\dfrac{a_{ij}}{c}$

If $c=2$ $\qquad$ if $c=2$

$z_{11}=2*3=6$ $\qquad$ $z_{12}=2*6=12$ $\qquad$ $z_{11}=3/2=1.5$ $\qquad$ $z_{12}=6/2=3$

$z_{21}=2*5=10$ $\qquad$ $z_{22}=2*8=16$ $\qquad$ $z_{21}=5/2=2.5$ $\qquad$ $z_{22}=8/2=4$

$z_{31}=2*(-2)=-4$ $\qquad$ $z_{32}=2*9=18$ $\qquad$ $z_{31}=-2/2=-1$ $\qquad$ $z_{32}=9/2=4.5$

Example shows that a scalar matrix is obtained when an identity matrix is multiplied by a scalar.

$$I=\begin{bmatrix}1 & 0 & 0\\ 0 & 1 & 0\\ 0 & 0 & 1\end{bmatrix} \quad Z=c*I=\begin{bmatrix}c & 0 & 0\\ 0 & c & 0\\ 0 & 0 & c\end{bmatrix}$$

Identity matrix $\qquad$ Scalar matrix

Multiplying by a zero matrix gives a zero matrix :

$$A0=0$$

Matrix multiplication is associative:

$$A(BC)=(AB)C$$

For compatible matrices A, B, and C. Matrix multiplication distributes over addition:

$$A(B+C)=AB+AC$$
$$(B+C)D=BD+CD$$

4. Matrix-Vector Product

Very important special case of matrix multiplication : $y = Ax$

- A is an $m \times n$ matrix
- x is an n-vector
- y is an m-vector

$$y_i = A_{i1}x_1 + A_{i2}x_2 + \ldots + A_{in}x_n \qquad \text{for } i = 1, \ldots m$$

5. Inner Product

If v is a row n-vector and w is a column n-vector, then vw makes sense, and has size 1×1, *i.e.*, is a scalar :

$$vw = v_1 w_1 + \ldots + v_n w_n$$

if x and y are n-vectors, $x^T y$ is a scalar called inner product or dot product of x, y, and denoted $\langle x, y \rangle$ or $x \cdot y$:

$$\langle x, y \rangle = x^T y = x_1 y_1 + \ldots + x_n y_n$$

6. Matrix inverses, ranks, and determinants

We define the inverse of an $n \times n$ matrix A to be the $n \times n$ matrix, denoted A^{-1} (if it exists), such that $AA^{-1} = I_n = A^{-1}A$. *For example,*

$$\begin{bmatrix} 1 & -1 \\ 1 & 2 \end{bmatrix}^{-1} = \frac{1}{3}\begin{bmatrix} 2 & 1 \\ -1 & 1 \end{bmatrix}$$

Many nonzero $n \times n$ matrices do not have inverses. A matrix without an inverse is is called **noninvertible**, or **singular**. An example of a nonzero singular matrix is

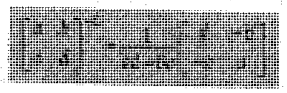
$$\begin{bmatrix} 1 & -1 \\ -2 & 2 \end{bmatrix}$$

$$\begin{bmatrix} a & b \\ c & d \end{bmatrix}\begin{bmatrix} 1 & -1 \\ -2 & 2 \end{bmatrix} = \begin{bmatrix} a-2b & -a+2b \\ c-2d & -c+2d \end{bmatrix} = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}$$

...but we can't have $a - 2b = 1$ and $-a + 2b = 0$.

Inverse of 2 x 2 matrix

It's useful to know the general formula for the inverse of a 2 x 2 matrix:



provided $ad - bc \neq 0$ (if $ad - bc = 0$, the matrix is singular)

If a matrix has an inverse, it is called **invertible**, or **nonsingular**. If A and B are nonsingular $n \times n$ matrices, then

$$(BA)^{-1} = A^{-1}B^{-1}$$

The inverse operation commutes with the transpose operation:

$$(A^{-1})^T = (A^T)^{-1}$$

The vectors $x_1, x_2, \ldots, x_n$ are **linearly dependent** if there exist coefficients $c_1, c_2, \ldots, c_n$ not all of which are zero, such that $c_1x_1 + c_2x_2 + \ldots + c_nx_n = 0$. For example, the vectors $x_1 = (1\ 2\ 3)^T$, $x_2 = (2\ 6\ 4)^T$, and $x_3 = (4\ 11\ 9)^T$ are linearly dependent, since $2x_1 + 3x_2 - 2x_3 = 0$. If vectors are not linearly dependent, they are **linearly independent**. For example, the columns of an identity matrix are linearly independent.

The **column rank** of a nonzero $m \times n$ matrix A is the size of the largest set of linearly independent columns of A. Similarly, the **row rank** of A is the size of the largest set of linearly independent rows of A. A fundamental property of any matrix A is that its row rank always equals its column rank, so that we can simply refer to the **rank of** A. The rank of an $m \times n$ matrix is an integer between 0 and $\min(m, n)$, inclusive. (The rank of a zero matrix is 0, and the rank of an $n \times n$ identity matrix is n.) An alternate, but equivalent and often more useful, definition is that the rank of a nonzero $m \times n$ matrix A is the smallest number r such that there exist matrices B and C of respective sizes $m \times r$ and $r \times n$ such that

$$A = BC.$$

A square $n \times n$ matrix has **full rank** if its rank is n. A null vector for a matrix A is a nonzero vector x such that $Ax = 0$.

For a 2×2 matrix, the **determinant** is $ad - bc$.

$$\begin{bmatrix} a & b \\ c & d \end{bmatrix}$$

For a 3×3 matrix, the determinant is $aei + bfg + dhc - ceg - ibd - ahf$. We should be able to see a pattern.

$$\begin{bmatrix} a & b & c \\ d & e & f \\ g & h & i \end{bmatrix}$$

Really though, for anything bigger than a 2 x 2 matrix we should use the det

Some important properties of determinants

The determinant of a square matrix A has the following properties :

1. If any row or any column of A is zero, then $\det(A) = 0$.
2. The determinant of A is multiplied by λ if the entries of any one row (or any one column) of A are all multiplied by λ.
3. The determinant of A is unchanged if the entries in one row (respectively, column) are added to those in another row (respectively, column).
4. The determinant of A equals the determinant of A^T.
5. The determinant of A is multiplied by -1 if any two rows (respectively, columns) are exchanged.
6. Also, for any square matrices A and B, we have $\det(AB) = \det(A)\det(B)$.
7. An $n \times n$ matrix A is singular if and only if $\det(A) = 0$.

33.3 Strassen's Algorithm for Matrix Multiplication

It is an application of a familiar design technique 'divide and conquer'. Suppose we wish to compute the product $C = AB$ where each A, B and C are $n \times n$ matrices. Assuming that n is an exact power of 2.

We divide each of A, B and C into four $n/2 \times n/2$ matrices.

Rewriting the eqution $C = AB$ as

$$\begin{pmatrix} r & s \\ t & u \end{pmatrix} = \begin{pmatrix} a & b \\ c & d \end{pmatrix} \begin{pmatrix} e & g \\ f & h \end{pmatrix} \quad ...(1)$$

For convenience, the submatrices of A are labelled alphabetical from left to right, whereas those of B are labelled from top to bottom. So that matrix multiplication is performed.

Equation (1) corresponds to the 4 equations

$$r = ae + bf \quad ...(2)$$

$$s = ag + bh \quad ...(3)$$

$$t = ce + df \quad ...(4)$$

$$u = cg + dh \quad ...(5)$$

Each of these 4 equations specifies two multiplications of $n/2 \times n/2$ matrices and the addition of their $n/2 \times n/2$ products. Using these equations to define a stratight-forward divide and conquer strategy. We derive the following recurrence for the time $T(n)$ to multiply two $n \times n$ matrices :

$$T(n) = 8T(n/2) + \theta(n^2)$$

Unfortunately this recurrence has the solution $T(n) = \theta(n^3)$ and thus, this method is no faster than the ordinary one.

Volker Strassen has discovered a different recursive approach that required only 7 recursive multiplication of $n/2 \times n/2$ matrices and $\theta(n^2)$ scalar additions and subtractions yielding the recurrence.

$$T(n) = 7T(n/2) + \theta(n^2)$$

$$= \theta(n^{\log 7}) \approx O(n^{2.81})$$

Strassen's Method has 4 steps :

1. Divide the input matrices A and B into $n/2 \times n/2$ submatrices.
2. Using $\theta(n^2)$ scalar additions and subtractions compute 14 $n/2 \times n/2$ matrices $A_1, B_1, A_2, B_2, \ldots A_7, B_7$.

3. Recursively compute the seven matrix products
   $P_i = A_i B_i$ for $i = 1, 2, \ldots 7$
4. Compute the desired submatrices r, s, t, u of the result matrix C by adding and/or subtracting various combinations of the P_i matrices using only $\theta(n^2)$ scalar additions and subtractions.

Suppose, $A = \begin{bmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{bmatrix}$ $B = \begin{bmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{bmatrix}$ and $C = \begin{bmatrix} C_{11} & C_{12} \\ C_{21} & C_{22} \end{bmatrix}$

$$\begin{bmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{bmatrix} \begin{bmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{bmatrix} = \begin{bmatrix} C_{11} & C_{12} \\ C_{21} & C_{22} \end{bmatrix}$$

In Strassen method first compute the 7 $n/2 \times n/2$ matrices.

P, Q, R, S, T, U, V as follows :

$$P = (A_{11} + A_{22})(B_{11} + B_{22})$$

$$Q = (A_{21} + A_{22}) B_{11}$$

$$R = A_{11} (B_{12} - B_{22})$$

$$S = A_{22} (B_{21} - B_{11})$$

$$T = (A_{11} + A_{12}) B_{22}$$

$$U = (A_{21} - A_{11})(B_{11} + B_{12})$$

$$V = (A_{12} - A_{22})(B_{21} + B_{22})$$

Then the c_{ij}'s are computed using the formulas

$$c_{11} = P + S - T + V$$
$$c_{12} = R + T$$
$$c_{21} = Q + S$$
$$c_{22} = P + R - Q + U$$

As can be seen P, Q, R, S, T, U and V can be computed using 7 matrix multiplications and 10 matrix additions or substrations and c_{ij}'s require an additional 8 addition or substrations.

The resulting recurrence relation for $T(n)$ is

$$T(n) = \begin{cases} b & n \le 2 \\ 7T(n/2) + an^2 & n > 2 \end{cases}$$

where a and b are constants.

Working with this formula we get

$$T(n) = an^2\left[1+\frac{7}{4}+\left(\frac{7}{4}\right)^2+\ldots+\left(\frac{7}{4}\right)^{k-1}\right]+7^k\,T(1)$$

$$\le cn^2\left(\frac{7}{4}\right)^{\lg n}+7^{\log_2 n}$$

$$= cn^{\log_2 4+\log_2 7+(-\log_2 4)}+n^{\log_2 7}$$

$$= O(n^{\log_2 7}) \approx O(n^{2.81})$$

33.4 Solving Systems of Linear Equations

A linear system can be expressed as a matrix equation in which each matrix or vector element belong to a field. We know a ring is a field if it satisfied two additional properties :

1. The operator is commutative.
2. Every non-zero element in S has a multiplicative inverse.

Let us consider a set of linear equations in n unkwons $x_1, x_2, \ldots, x_n$ are

$$a_{11}x_1+a_{12}x_2+\ldots+a_{1n}x_n=b_1$$
$$a_{21}x_1+a_{22}x_2+\ldots+a_{2n}x_n=b_2$$
$$\text{----------}$$
$$a_{n1}x_1+a_{n2}x_2+\ldots+a_{nn}x_n=b_n$$

Rewrite these equations as the matrix-vector equation

$$\begin{bmatrix} a_{11} & a_{12} & \cdots & a_{1n} \\ a_{21} & a_{22} & \cdots & a_{2n} \\ \vdots & & & \vdots \\ a_{n1} & a_{n2} & \cdots & a_{nn} \end{bmatrix}\begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{bmatrix}=\begin{bmatrix} b_1 \\ b_2 \\ \vdots \\ b_n \end{bmatrix}$$

Letting $A=(a_{ij})$, $x=(x_j)$ and $b=(b_i)$ as $Ax=b$.

In this case, we only treat exactly n equations in n unknown. So, the rank of A is equal to the number n of unknowns. If the number of equations is less than the number n of unknowns or more generally, if the rank of A is less than n then the system is **under determined**. An under determined system typically has infinitely many solutions although it may have no solutions at all if the equations are inconsistent. If the number of equations exceeds the number n of unknowns, the system is **over determined** and there may not exist any solutions.

To solving the system $Ax=b$ of n equations in n unknowns. One approach is to compute A^{-1} and then multiply both sides yielding $A^{-1}Ax=A^{-1}b$ or $x=A^{-1}b$.

This approach has round-off errors tend to accumulate unduly when floating-point number representations are used instead of ideal real numbers.

There are another approach – *LUP* **decomposition** that is numerically stable and has the further advantage of being about a factor of 3 faster.

The idea behind *LUP* decomposition is to find three $n\times n$ matrices L, U and P such that

$$PA=LU$$

where L is a unit lower-triangular matrix.
U is an upper triangular matrix and
P is a permutation matrix.

Permutation matrix P has exactly one 1 in each row or column and 0's elsewhere.

If A decomposed in such a way *i.e.*, $PA=LU$, then it is called an *LUP* decomposition of the matrix A. Having found an *LUP* decomposition for A, we can solve the equation $Ax=b$ by solving only triangular linear systems.

Multiplying both sides of $Ax=b$ by P, yielding then

$$PAx = Pb$$

i.e.,

$$LUx = Pb$$

Suppose $y=Ux$, where x is the desired solution. First, we solve the lower-triangular system. $Ly=Pb$ for a unknown vector y by a method called **"Forward Substitution"** then solve the upper-triangular system. $Ux=y$ by the **'backward substitution'**. The vector x is our solution to $Ax=b$.

33.5 Computing an *LU* decomposition

When matrix A is an $n\times n$ non-singular matrix and P is absent. In this case, we must find a factorization

$$A=LU$$

We call the two matrices L and U an *LU* decomposition of A and the process by which we perform *LU* decomposition is called **'Gaussian elimination'**. If $n=1$ then we can choose $L=I_1$ and $U=A$.

For $n>1$, we break A into *four* parts :

$$A=\left(\begin{array}{c|ccc} a_{11} & a_{12} & \cdots & a_{1n} \\ \hline a_{21} & a_{22} & \cdots & a_{2n} \\ \vdots & & & \vdots \\ a_{n1} & a_{n2} & \cdots & a_{nn} \end{array}\right)=\begin{pmatrix} a_{11} & w^T \\ v & A' \end{pmatrix}$$

where v is a size $(n-1)$ column vector.
w^T is a size $(n-1)$ row vector.
A' is an $(n-1)\times(n-1)$ matrix.

$$A = \begin{pmatrix} a_{11} & w^T \\ v & A' \end{pmatrix}$$

$$= \begin{pmatrix} 1 & 0 \\ V/a_{11} & I_{n-1} \end{pmatrix} \begin{pmatrix} a_{11} & w^T \\ 0 & A' - vw^T / a_{11} \end{pmatrix}$$

The $(n-1)\times(n-1)$ matrix $A' - vw^T / a_{11}$ is called the **'Schur Complement'** of A with respect to a_{11}.

We now recursively find the LU decomposition of the schur complement.

Now, to decompose into LU.

Step 1 Divide all the elements of first column by a_{11} except a_{11}.

Step 2 Now at place of $a_{22} \leftarrow a_{22} - a_{21} \times a_{12}$ and $a_{23} \leftarrow a_{23} - a_{21} \times a_{13}$ and $a_{24} \leftarrow a_{24} - a_{21} \times a_{14}$ etc.

Step 3 Similarly $a_{32} \leftarrow a_{32} - a_{31} \times a_{12}$

$a_{33} \leftarrow a_{33} - a_{31} \times a_{13}$ etc.

Step 4 Similarly $a_{42} \leftarrow a_{42} - a_{41} \times a_{12}$ etc.

For example, $\begin{bmatrix} 2 & 3 & 1 & 5 \\ 6 & 13 & 5 & 19 \\ 2 & 19 & 10 & 23 \\ 4 & 10 & 11 & 31 \end{bmatrix}$ here $a_{11} = 2$

$$\begin{bmatrix} 2 & 3 & 1 & 5 \\ 3 & 4 & 2 & 4 \\ 1 & 16 & 9 & 18 \\ 2 & 4 & 9 & 21 \end{bmatrix} \cdot \begin{bmatrix} 2 & 3 & 1 & 5 \\ 3 & 4 & 2 & 4 \\ 1 & 4 & 1 & 2 \\ 2 & 1 & 7 & 3 \end{bmatrix}$$

$$\underset{A}{\begin{bmatrix} 2 & 3 & 1 & 5 \\ 3 & 4 & 2 & 4 \\ 1 & 4 & 1 & 2 \\ 2 & 1 & 7 & 3 \end{bmatrix}} = \underset{L}{\begin{bmatrix} 1 & 0 & 0 & 0 \\ 3 & 1 & 0 & 0 \\ 1 & 4 & 1 & 0 \\ 2 & 1 & 7 & 1 \end{bmatrix}} \times \underset{U}{\begin{bmatrix} 2 & 3 & 1 & 5 \\ 0 & 4 & 2 & 4 \\ 0 & 0 & 1 & 2 \\ 0 & 0 & 0 & 3 \end{bmatrix}}$$

i.e., $A = LU$

LU-Decomposition (A)

```
n ← rows[A]
for k ← 1 to n
    do u_kk ← a_kk
        for i ← k+1 to n
            do l_ik ← a_ik / u_kk          ▷ l_ik holds v_i
            u_ki ← a_ki                    ▷ u_ki holds w_i^T
        for i ← k+1 to n
            do for j ← k+1 to n
                do a_ij ← a_ij − l_ik u_kj
return L and U
```


33.6 Computing an LUP decomposition

LUP decomposition is similar to that of LU decomposition. We are given an $n \times n$ non-singular matrix A and we wish to find a permutation matrix P, a unit lower-triangular matrix L, and an upper-triangular matrix U such that $PA = LU$. Before we partition the matrix A, as we did for LU decomposition, we move a non-zero element say a_{k1}, from the first column to the (1, 1) position of the matrix that is largest element in the first column.

LUP-Decomposition (A)

```
n ← rows[A]
for i ← 1 to n
    do π[i] ← i
for k ← 1 to n−1
    do p ← 0
        for i ← k to n
            do if |a_ik| > p
                then p ← |a_ik|
                    k' ← i
        if p = 0
            then error "singular matrix"
        exchange π[k] ↔ π[k']
        for i ← 1 to n
            do exchange a_ki ↔ a_k'i
        for i ← k+1 to n
            do a_ik ← a_ik / a_kk
            for j ← k+1 to n
                do a_ij ← a_ij − a_ik a_kj
```

Example : LUP decomposition is as follows

$$A = \begin{bmatrix} 2 & 0 & 2 & 0.6 \\ 3 & 3 & 4 & -2 \\ 5 & 5 & 4 & 2 \\ -1 & -2 & 3.4 & -1 \end{bmatrix}$$

The identity permutation of the rows are on the left of A

$$\begin{matrix} 1 \\ 2 \\ 3 \\ 4 \end{matrix} \quad \begin{matrix} 2 & 0 & 2 & 0.6 \\ 3 & 3 & 4 & -2 \\ 5 & 5 & 4 & 2 \\ -1 & -2 & 3.4 & -1 \end{matrix}$$

The maximum element in column first is 5 which is in the third row. So interchange third row and first row and update the permutation.

3	5	5	4	2
2	3	3	4	−2
1	2	0	2	0.6
4	−1	−2	3.4	−1

Divide by 5 in first column and update *i.e.*, $a_{22} \leftarrow a_{22} - a_{21} \times a_{12}$. $a_{32} \leftarrow a_{32} - a_{31} \times a_{12}$ etc.

3	5	5	4	2
2	0.6	0	1.6	−3.2
1	0.4	−2	0.4	−0.2
4	−0.2	−1	4.2	−0.6

Swap Row second and Row third.

3	5	5	4	2
1	0.4	−2	0.4	−0.2
2	0.6	0	1.6	−3.2
4	−0.2	−1	4.2	−0.6

3	5	5	4	2
1	0.4	−2	0.4	−0.2
2	0.6	0	1.6	−3.2
4	−0.2	0.5	4	−0.5

Swap Row forth and Row third.

3	5	5	4	2
1	0.4	−2	0.4	−0.2
4	−0.2	0.5	4	−0.5
2	0.6	0	1.6	−3.2

3	5	5	4	2
1	0.4	−2	0.4	−0.2
4	−0.2	0.5	4	−0.5
2	−0.6	0	0.4	−3

Thus, $P = \begin{pmatrix} 0 & 0 & 1 & 0 \\ 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 1 & 0 & 0 \end{pmatrix}$ $L = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0.4 & 1 & 0 & 0 \\ -0.2 & 0.5 & 1 & 0 \\ 0.6 & 0 & 0.4 & 1 \end{pmatrix}$ $U = \begin{pmatrix} 5 & 5 & 4 & 2 \\ 0 & -2 & 0.4 & -0.2 \\ 0 & 0 & 4 & -0.5 \\ 0 & 0 & 0 & -3 \end{pmatrix}$

and $PA = LU$

where $A = \begin{bmatrix} 2 & 0 & 2 & 0.6 \\ 3 & 3 & 4 & -2 \\ 5 & 5 & 4 & 2 \\ -1 & -2 & 3.4 & -1 \end{bmatrix}$

Example. *Solve the equation*

$$\begin{pmatrix} 1 & 5 & 4 \\ 2 & 0 & 3 \\ 5 & 8 & 2 \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \\ x_3 \end{pmatrix} = \begin{pmatrix} 12 \\ 9 \\ 5 \end{pmatrix}$$ *by using an LUP decomposition.*

Solution. Here $A = \begin{pmatrix} 1 & 5 & 4 \\ 2 & 0 & 3 \\ 5 & 8 & 2 \end{pmatrix}$ $x = \begin{pmatrix} x_1 \\ x_2 \\ x_3 \end{pmatrix}$ $b = \begin{pmatrix} 12 \\ 9 \\ 5 \end{pmatrix}$ *i.e.*, $Ax = b$.

The LUP decomposition of A is as follows :

1	1	5	4
2	2	0	3
3	5	8	2

3	5	8	2
2	2	0	3
1	1	5	4

3	5	8	2
2	0.4	−3.2	2.2
1	0.2	3.4	3.6

3	5	8	2
2	0.4	−3.2	2.2
1	0.2	3.4	3.6

3	5	8	2
1	0.2	3.4	3.6
2	0.2	−3.2	2.2

3	5	8	2
1	0.2	3.4	3.6
2	0.4	−0.9	5.44

Thus $P = \begin{pmatrix} 0 & 0 & 1 \\ 1 & 0 & 0 \\ 0 & 1 & 0 \end{pmatrix}$ $L = \begin{pmatrix} 1 & 0 & 0 \\ 0.2 & 1 & 0 \\ 0.4 & -0.9 & 1 \end{pmatrix}$ $U = \begin{pmatrix} 5 & 8 & 2 \\ 0 & 3.4 & 3.6 \\ 0 & 0 & 5.44 \end{pmatrix}$

$PA = LU$

Using forward substitution, we solve $Ly = Pb$ for y.

$$\begin{pmatrix} 1 & 0 & 0 \\ 0.2 & 1 & 0 \\ 0.4 & -0.9 & 1 \end{pmatrix} \begin{pmatrix} y_1 \\ y_2 \\ y_3 \end{pmatrix} = \begin{pmatrix} 0 & 0 & 1 \\ 1 & 0 & 0 \\ 0 & 1 & 0 \end{pmatrix} \begin{pmatrix} 12 \\ 9 \\ 5 \end{pmatrix}$$

$$\begin{pmatrix} 1 & 0 & 0 \\ 0.2 & 1 & 0 \\ 0.4 & -0.9 & 1 \end{pmatrix} \begin{pmatrix} y_1 \\ y_2 \\ y_3 \end{pmatrix} = \begin{pmatrix} 5 \\ 12 \\ 9 \end{pmatrix}$$

Obtaining

$$y_1 = 5$$
$$0.2y_1 + y_2 = 12$$
$$0.4y_1 - 0.9y_2 + y_3 = 9$$

Solving these, we get $y = \begin{pmatrix} 5 \\ 11 \\ 16.9 \end{pmatrix}$ by computing first y_1 then y_2 and finally y_3.

Using back substitution solve $Ux = y$ for x

$$\begin{pmatrix} 5 & 8 & 2 \\ 0 & 3.4 & 3.6 \\ 0 & 0 & 5.44 \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \\ x_3 \end{pmatrix} = \begin{pmatrix} 5 \\ 11 \\ 16.9 \end{pmatrix}$$

$$5x_1 + 8x_2 + 2x_3 = 5$$
$$3.4x_2 + 3.6x_3 = 11$$
$$5.44x_3 = 16.9$$

Computing first x_3, then x_2 and finally x_1.

$$x = \begin{pmatrix} x_1 \\ x_2 \\ x_3 \end{pmatrix} = \begin{pmatrix} -0.17 \\ -0.04 \\ 3.10 \end{pmatrix}$$

Exercise

1. How would you modify strassen's algorithm to multiply $n \times n$ matrices in which n is not an exact power of 2 ? Show that the resulting algorithm runs in time $\theta(n^{\log 7})$.
2. Use strassen's algorithm to compute the matrix product $\begin{pmatrix} 1 & 5 \\ 3 & 8 \end{pmatrix}\begin{pmatrix} 7 & 4 \\ 6 & 2 \end{pmatrix}$. Show your work.
3. Show how to multiply the complex numbers $a + bi$ and $c + di$ using only three real multiplications. The algorithm should take a, b, c and d as input and produce the real component $ac - bd$ and the imaginary component $ad + bc$ seperately.
4. Solve the equation $\begin{pmatrix} 1 & 0 & 0 \\ 4 & 1 & 0 \\ -7 & 6 & 1 \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \\ x_3 \end{pmatrix} = \begin{pmatrix} 3 \\ 14 \\ -7 \end{pmatrix}$ by using forward substitution.
5. Solve the equation $\begin{pmatrix} 1 & 5 & 4 \\ 2 & 0 & 3 \\ 5 & 8 & 2 \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \\ x_3 \end{pmatrix} = \begin{pmatrix} 12 \\ 9 \\ 5 \end{pmatrix}$ by using an LUP decomposition.
6. Describe the LUP decomposition of a diagonal matrix.
7. Discuss the Strassen's algorithm for matrix multiplication. Show that two $n \bullet n$ matrices with elements from an arbitrary ring can be multiplied in $c(n^{\log 7})$ arithmetic operations.

CHAPTER

34

Number-Theoretic Algorithms

34.1 Some Facts From Elementary Number Theory

We know, set $Z = \{\ldots, -2, -1, 0, 1, 2, \ldots\}$ of integers and the set $N = \{0, 1, 2, \ldots\}$ of natural numbers. Given positive numbers a and d, the notation $d \mid a$ (read "d divides a") means that $a = kd$ for some integer k. Every integer divides 0. If $a > 0$ and $d \mid a$, then $|d| \le |a|$. If $d \mid a$, then we also say that a is a multiple of d.

If $d \mid a$ and $d \ge 0$, we say that d is a divisor of a. Note that $d \mid a$ if and only if $-d \mid a$ so that no generality is lost by defining the divisors to be nonnegative, with the understanding that the negative of any divisor of a also divides a. A divisor of an integer a is at least 1 but not greater than $|a|$. For example, the divisors of 24 are 1, 2, 3, 4, 6, 8, 12, and 24.

Let a, b and c are arbitrary integers then

- If $a \mid b$ and $b \mid c$, then $a \mid c$
- If $a \mid b$ and $b \mid c$, then $a \mid (ib + jc)$, for all integers i and j.
- If $a \mid b$ and $b \mid a$, then $a = b$ or $a = -b$.

Every integer a is divisible by the *trivial divisors* 1 and a. Nontrivial divisors of a are also called factors of a. For example, the factors of 20 are 2, 4, 5, and 10.

Prime and Composite Numbers

An integer $a > 1$ whose only divisors are the trivial divisors 1 and a is said to be a **prime number** (or, more simply, a prime). An integer $a > 1$ that is not prime is said to be a **composite number** (or, more simply, a composite). So, for example, 5, 11, 101 are prime and 25, 39 is composite. The integer 1 is said to be a **unit** and is neither prime nor composite. Similarly, the integer 0 and all negative integers are neither prime nor composite.

The Division Theorem, Remainders, and Modular Equivalence

Given an integer n, the integers can be partitioned into those that are multiples of n and those that are not multiples of n.

Division Theorem

For any integer a and any positive integer n, there are unique integers q and r such that $0 \le r < n$ and $a = qn + r$.

The value $q = \lfloor a/n \rfloor$ is the quotient of the division. The value $r = a \bmod n$ is the remainder (or residue) of the division. We have that $n \mid a$ if and only if $a \bmod n = 0$. It follows that

$$a = \lfloor a/n \rfloor n + (a \bmod n) \text{ or } a \bmod n = a - \lfloor a/n \rfloor n.$$

The integers can be divided into n equivalence classes according to their remainders modulo n. The **equivalence class modulo** n containing an integer a is

$$[a]_n = \{a + kn : k \in Z\}$$

For example, $[3]_7 = \{\ldots, -11, -4, 3, 10, 17, \ldots\}$; other denotations for this set are $[-4]_7$ and $[10]_7$. Writing $a \in [b]_n$ is the same as writing $a \equiv b \pmod n$. The set of all such equivalence classes is

$$Z_n = \{[a]_n : 0 \le a \le n-1\}$$

Or

$$Z_n = \{0, 1, \ldots, n-1\}$$

where 0 represents $[0]_n$, 1 represents $[1]_n$, and so on ; each class is represented by its least nonnegative element.

It is sometimes convenient to talk about **congruence modulo** n. If $a \bmod n = b \bmod n$, we say that a is congruent to b modulo n, which we call the modulus, and we write

$$a \equiv b \pmod n$$

Therefore, if $a \equiv b \pmod n$, then $a - b = kn$ for some integer k.

Common Divisors and Greatest Common Divisors

If d is a divisor of a and also a divisor of b, then d is a **common divisor** of a and b. For example, the divisors of 30 are 1, 2, 3, 5, 6, 10, 15, and 30, and so the common divisors of 24 and 30 are 1, 2, 3, and 6. Note that 1 is a common divisor of any two integers.

An important property of common divisors is that

$$d \mid a \text{ and } d \mid b \text{ implies } d \mid (a+b) \text{ and } d \mid (a-b).$$

More generally, we have that

$$d \mid a \text{ and } d \mid b \text{ implies } d \mid (ax + by) \text{ for any integers } x \text{ and } y.$$

The greatest common divisor of two integers a and b, not both zero, is the largest of the common divisors of a and b; it is denoted $\gcd(a, b)$. Alternatively, we could say that $\gcd(a, b)$ is the number c, such that if $d \mid a$ and $d \mid b$, then $d \mid c$. If $\gcd(a, b) = 1$, we say that a and b are **relatively prime**. For example, $\gcd(24, 30) = 6$, $\gcd(5, 7) = 1$, and $\gcd(0, 9) = 9$. If a and b are not both 0, then $\gcd(a, b)$ is an integer between 1 and $\min(|a|, |b|)$. We define $\gcd(0, 0)$ to be 0 ; The following are elementary properties of the gcd function :

- $\gcd(a, b) = \gcd(b, a)$,
- $\gcd(a, b) = \gcd(-a, b)$,
- $\gcd(a, b) = \gcd(|a|, |b|)$
- $\gcd(a, 0) = \gcd(0, a) = |a|$
- $\gcd(a, ka) = |a|$ for any $k \in Z$.

34.2 Euclid's GCD Algorithm

To compute the greatest common divisor of two numbers, we can use one of the oldest algorithms known, Euclid's algorithm. This algorithm is based on the following property of $\gcd(a, b)$

Theorem

For any nonnegative integer a and any positive integer b, $\gcd(a, b) = \gcd(b, a \bmod b)$.

Proof. We shall show that $\gcd(a, b)$ and $\gcd(b, a \bmod b)$ divide each other, so that they must be equal (since they are both nonnegative).

We first show that $\gcd(a, b) \mid \gcd(b, a \bmod b)$. If we let $d = \gcd(a, b)$, then $d \mid a$ and $d \mid b$. $(a \bmod b) = a - qb$, where $q = \lfloor a/b \rfloor$. Since $(a \bmod b)$ is thus a linear combination of a and b, implies that $d \mid (a \bmod b)$. Therefore, since $d \mid b$ and $d \mid (a \bmod b)$, implies that $d \mid \gcd(a, b)$ or, equivalently, that

$$\gcd(a, b) \mid \gcd(b, a \bmod b) \quad \text{...(1)}$$

Showing that $\gcd(b, a \bmod b) \mid \gcd(a, b)$ is almost the same. If we now let $d = \gcd(b, a \bmod b)$, then $d \mid b$ and $d \mid (a \bmod b)$. Since $a = qb + (a \bmod b)$, where $q = \lfloor a/b \rfloor$, we have that a is a linear combination of b and $(a \bmod b)$. Then, we conclude that $d \mid a$. Since $d \mid b$ and $d \mid a$, we have that $d \mid \gcd(a, b)$ or, equivalently, that

$$\gcd(b, a \bmod b) \mid \gcd(a, b) \quad \text{...(2)}$$

We know $a \mid b = b \mid a$ implies $a = \pm b$

Using this to combine equations (1) and (2) completes the proof.

This theorem leads us easily to an ancient algorithm, known as Euclid's algorithm

Euclid's algorithm

The following gcd algorithm is described in the *Elements* of Euclid (circa 300 B.C.), although it may be of even earlier origin. It is written as a recursive program based directly on above theorem. The inputs a and b are arbitrary nonnegative integers.

```
EUCLID (a, b)
1.  if b = 0
2.      then return a
3.      else return EUCLID (b, a mod b)
```

As an example of the running of EUCLID, consider the computation of gcd (30,21) :

$$\begin{aligned} \text{EUCLID}(30, 21) &= \text{EUCLID}(21, 9) \\ &= \text{EUCLID}(9, 3) \\ &= \text{EUCLID}(3, 0) \\ &= 3 . \end{aligned}$$

The running time of Euclid's algorithm

Let a and b be two positive integers. Euclid's algorithm computes $\gcd(a,b)$ by executing $O(\log \max(a,b))$ arithmetic operations.

The extended form of Euclid's algorithm

We now rewrite Euclid's algorithm to compute additional useful information. Specifically, we extend the algorithm to compute the integer coefficients x and y such that

$$d = \gcd(a,b) = ax + by$$

Note that x and y may be zero or negative. The procedure EXTENDED-EUCLID takes as input an arbitrary pair of integers and returns a triple of the form (d, x, y).

```
EXTENDED-EUCLID (a, b)
1.  if b = 0
2.      then return (a, 1, 0)
3.  (d', x', y') ← EXTENDED-EUCLID(b, a mod b)
4.  (d, x, y) ← (d', y', x' − ⌊a / b⌋ y')
5.  return (d, x, y)
```

Example. *Compute gcd(99,78) with EXTENDED-EUCLID .*

Solution. Let $a = 99$ and $b = 78$

a	b	$\lfloor a/b \rfloor$	d	x	y
99	78	1	3	−11	14
78	21	3	3	3	−11
21	15	1	3	−2	3
15	6	2	3	1	−2
6	3	2	3	0	1
3	0	—	3	1	0

The call EXTENDED-EUCLID(99, 78) returns (3, − 11, 14), so gcd(99, 78) = 3 and $\gcd(99, 78) = 3 = 99(-11) + 78 \cdot 14$.

Since the number of recursive calls made in EUCLID is equal to the number of recursive calls made in EXTENDED-EUCLID, the running times of EUCLID and EXTENDED-EUCLID are the same, to within a constant factor. That is, for $a > b > 0$, the number of recursive calls is $O(\lg b)$.

34.3 The Chinese Remainder Theorem

Around A.D. 100, the Chinese mathematician sun-tsi solved the problem of finding those integers x that leave remainder 2 when divided by 3, remainder 3 when divided by 5, and remainder 2 when divided by 7. One such solution is $x = 23$; all solutions are of the form $23 + 105k$ for arbitrary integers k.

The "Chinese remainder theorem" provides a correspondence between a system of equations modulo a set of pair wise relatively prime moduli (for example, 3, 5, and 7) and an equation modulo their product (for example, 105).

Theorem

Let $n = n_1 n_2 \ldots n_k$, where the n_i are pairwise relatively prime. Consider the correspondence

$$a \leftrightarrow (a_1, a_2, \ldots, a_k)$$

where $a \in Z_n$, $a_i \in Z_{n_i}$, and $a_i = a \bmod n_i$

for $i = 1, 2, \ldots, k$. Then, mapping of the above equation is a one-to-one correspondence (bijection) between Z_n and the Cartesian product $Z_{n1} \times Z_{n2} \times \ldots \times Z_{nk}$. Operations performed on the elements of Z_n can be equivalently performed on the corresponding k-tuples by performing the operations independently in each coordinate position in the appropriate system. That is, if

$$a \leftrightarrow (a_1, a_2, \ldots, a_k),$$
$$b \leftrightarrow (b_1, b_2, \ldots, b_k),$$

then

$$(a+b) \bmod n \leftrightarrow ((a_1 + b_1) \bmod n_1, \ldots, (a_k + b_k) \bmod n_k),$$
$$(a-b) \bmod n \leftrightarrow ((a_1 - b_1) \bmod n_1, \ldots, (a_k - b_k) \bmod n_k),$$
$$(ab) \bmod n \leftrightarrow ((a_1 b_1) \bmod n_1, \ldots, (a_k b_k) \bmod n_k)$$

34.4 The RSA Public-key Cryptosystem

A public-key cryptosystem can be used to encrypt messages sent between two communicating parties so that an eavesdropper who overhears the encrypted messages will not be able to decode them. A public-key cryptosystem also enables a party to append an unforgeable "**digital signature**" to the end of an electronic message. Such a signature is the electronic version of a handwritten signature on a paper document. It can be easily checked by anyone, forged by no one, yet loses its validity if any bit of the message is altered. It therefore provides authentication of both the identity of the signer and the contents of the signed message. It is the perfect tool for electronically signed business contracts, electronic checks, electronic purchase orders, and other electronic communications that must be authenticated.

The RSA public-key cryptosystem is based on the dramatic difference between the ease of finding large prime numbers and the difficulty of factoring the product of two large prime numbers.

Public-key Cryptosystems

In a public-key cryptosystem, each participant has both a *public key* and a *secret key*. Each key is a piece of information. For example, in the RSA cryptosystem, each key consists of a pair of integers. Suppose the participants are "A" and "B". We denote their public and secret keys as P_A, S_A for A and P_B, S_B for B.

Each participant creates his own public and secret keys. Each keeps his secret key secret, but he can expose his public key to anyone or even publish it. In fact, it is often convenient to assume that everyone's public key is available in a public directory, so that any participant can easily obtain the public key of any other participant.

The function corresponding to A's public key P_A is denoted $P_A()$, and the function corresponding to his secret key S_A is denoted $S_A()$. The public and secret keys for any participant are a "matched pair" in that they specify functions that are inverses of each other. That is,

$$M = S_A(P_A(M))$$

$$M = P_A(S_A(M))$$

for any message M. Transforming M with the two keys P_A and S_A successively, in either order, yields the message M back.

In a public-key cryptosystem, it is essential that no one but A be able to compute the function $S_A()$ in any practical amount of time. The assumption that only A can compute $S_A()$ must hold even though everyone knows P_A and can compute $P_A()$, the inverse function to $S_A()$, efficiently. The major difficulty in designing a workable public-key cryptosystem is in figuring out how to create a system in which we can reveal a transformation $P_A()$ without thereby revealing how to compute the corresponding inverse transformation $S_A()$.

In a public-key cryptosystem, encryption works as follows. Suppose B wishes to send a message M to A and encrypted it, so that it will look like meaningless garbage to an eavesdropper.

The scenario for sending the message goes as follows.

- B obtains A's public key P_A (from a public directory or directly from A).
- B computes the ciphertext $C = P_A(M)$ corresponding to the message M and sends C to A.
- When A receives the ciphertext C, he applies his secret key S_A to retrieve the original message : $M = S_A(C)$.

Because $S_A()$ and $P_A()$ are inverse functions, A can compute M from C. Because only A is able to compute $S_A()$, only A can compute M from C.

Digital signatures are similarly easy to implement in a public-key cryptosystem. Suppose that A wishes to send B a digitally signed response M'. The digital-signature scenario proceeds as follows.

A computes his *digital signature* σ for the message M' using his secret key S_A and the equation $\sigma = S_A(M')$

A sends the message/signature pair (M', σ) to B.

When B receives (M', σ), he can verify that it originated from A using A's public key by verifying the equation $M' = P_A(\sigma)$. If the equation holds, then B concludes that the message M' was actually signed by A. If the equation doesn't hold, B concludes either that the message M' or the digital signature σ was corrupted by transmission errors or that the pair (M', σ) is an attempted forgery.

An important property of a digital signature is that it is verifiable by anyone who has access to the signer's public key. A signed message can be verified by one party and then passed on to other parties who can also verify the signature. After B verifies A's signature on the check, he can give the check to his bank, who can then also verify the signature and effect the appropriate funds transfer.

The RSA Cryptosystem

In the RSA public-key cryptosystem, a participant creates his public and secret keys with the following procedure.

1. Select at random two large prime numbers p and q.
2. Compute n by the equation $n = pq$.
3. Select a small odd integer e that is relatively prime to $\phi(n)$, which equals to $(p-1)(q-1)$.
4. Compute d as the multiplicative inverse of e, modulo $\phi(n)$.
5. Publish the pair $P = (e, n)$ as his ***RSA public key***.
6. Keep secret the pair $S = (d, n)$ as his ***RSA secret key***.

The transformation of a message M associated with a public key $P = (e, n)$ is

$$P(M) = M^e \pmod{n}$$

The transformation of a ciphertext C associated with a secret key $S = (d, n)$ is

$$S(C) = C^d \pmod{n}$$

These equations apply to both encryption and signatures. To create a signature, the signer applies his secret key to the message to be signed, rather than to a ciphertext. To verify a signature, the public key of the signer is applied to it, rather than to a message to be encrypted.

Exercise

1. Write the gcd binary algorithm for computing gcd of binary numbers.
2. Show that GCD operator is associative.
3. Write an algorithm for chinese remainder theorem using Ex Euclid.
4. Find all solutions to the equations

$$x \equiv 4 (\bmod 5) \text{ and } x \equiv 5 (\bmod 11)$$

5. Find all integer x that leave remainders 1, 2, 3 when divided by 9, 8, 7 respectively.
6. Consider an RSA key set with $p = 11$, $q = 29$, $n = 319$ and $e = 3$. What value of d should be used in the secret key ? What is the encryption of the message $M = 100$?
7. Discuss RSA cryptosystem.

CHAPTER

35

Polynomials and the FFT

35.1 Polynomial

A Polynomial in the variable x over an algebraic field F is representation of a function $A(x)$ as a formal sum.

$$A(x) = \sum_{j=0}^{n-1} a_j x^j$$

$a_0, a_1, a_2, \ldots, a_{n-1}$ are called the coefficients of the polynomial. A polynomial $A(x)$ is said to have degree k if its highest non-zero coefficient is a_k. Any integer strictly greater than the degree of a polynomial is a **degree-bound** of that polynomial. Therefore, the degree of a polynomial of degree-bound n may be any integer between 0 and $n-1$.

35.2 Representation of Polynomials

Here, we introduce the *two* representations

1. Coefficient Representation
2. Point-value Representation

Coefficient Representation

A coefficient representation of a polynomial $A(x)=\sum_{j=0}^{n-1} a_j x^j$ of degree bound n is a vector of coefficients $a=(a_0, a_1, a_2, \ldots, a_{n-1})$.

Point-value Representation

A point-value representation of a polynomial $A(x)$ of degree-bound n is a set of n point-value pairs

$$\{(x_0, y_0), (x_1, y_1), \ldots, (x_{n-1}, y_{n-1})\}$$

such that all of the x_k are distinct and $y_k = A(x_k)$ for $k=0,1,2,\ldots,n-1$.

A polynomial has many different point-value representations since any set of n distinct points $x_0, x_1, \ldots, x_{n-1}$ can be used as a basis for the representation.

Computing a point-value representation for a polynomial given in coefficient form is in principle straight forward. We select n distinct points $x_0, x_1, \ldots, x_{n-1}$ and then evaluate $A(x_k)$ for $k=0,1,2,\ldots,n-1$.

With Horner's Method, this n-point evaluation takes time $\theta(n^2)$.

The inverse of evaluation that is determining the coefficient form of a polynomial from a point-value representation is called **interpolation**.

35.3 Evaluating Polynomial Functions

Consider the polynomial $P(x)=a_n x^n + a_{n-1}x^{n-1} + \ldots + a_1 x + a_0$ with real coefficients and $n \geq 1$ then to evaluate $P(x)$ the algorithm is as follows :

Poly (A, n, x)

```
1.  P ← A[0], Power ← 1
2.  for i ← 1 to n
3.      do power ← power * x
4.          P ← A[i]* power + p
5.  return P
```

This algorithm does $2n$ multiplications and n additions.

Horner's Method

We have a better way of computing $ab+ac$ with fewer multiplications as $a(b+c)$. So we can write Horner's method for evaluating $P(x)$ using factorization.

$$P(x)=(\ldots((a_n x + a_{n-1})x + a_{n-2})x + \ldots + a_1)x + a_0$$

Horner Polynomial (A, n, x)

```
1.  P ← A[n]
2.  for j ← n − 1 down to 0 steps −1
3.      P ← P * x + A[j]
```

Thus simply by factoring P we have cut the number of multiplications in half without increasing the additions.

35.4 Complex Roots of Unity

A complex number ω is a complex nth root of unity, for $n \geq 2$, if it satisfies the following properties :

1. $\omega^n = 1$ that is w is an nth root of 1.
2. The numbers $1, \omega, \omega^2, \omega^3, \ldots, \omega^{n-1}$ are distinct.

The notion of complex nth root of unity has several important instances. One important one is complex number.

$$e^{2\pi i/n} = \cos(2\pi/n) + i\sin(2\pi/n)$$

which is complex nth root of unity, when we take our arithmetic over the complex numbers, where $i=\sqrt{-1}$.

The value $\omega_n = e^{2\pi i/n}$ is called the **principal nth root of unity**, and all of the other complex nth roots of unity are powers of ω_n.

Thus, the n complex nth root of unity are :

$\omega_n^0, \omega_n^1, \omega_n^2, \ldots, \omega_n^{n-1}$, which form a group under multiplication. This group has the same structure as the additive group $(Z_n, +)$ modulo n, since $\omega_n^n = \omega_n^0 = 1$ implies that $\omega_n^j \omega_n^k = \omega_n^{j+k} = \omega_n^{(j+k) \bmod n}$.

Complex nth root of unity have a number of important properties such as

1. ***For any integer $n \geq 0$ $k \geq 0$ and $d > 0$***

$$\omega_{dn}^{dk} = \omega_n^k$$

Proof. We know $\omega_n = e^{2\pi i/n}$.

$$\omega_{dn}^{dk} = (e^{2\pi i/dn})^{dk}$$

$$= (e^{2\pi i/n})^k = \omega_n^k$$

2. ***For any integer $n \geq 1$ and non-negative integer k not divisible by n.***

$$\sum_{j=0}^{n-1} (\omega_n^k)^j = 0$$

Proof. $$\sum_{j=0}^{n-1} (\omega_n^k)^j = \frac{(\omega_n^k)^n - 1}{\omega_n^k - 1} = \frac{(\omega_n^n)^k - 1}{\omega_n^k - 1}$$

$$= \frac{1^k - 1}{\omega_n^k - 1} = \frac{1-1}{\omega_n^k - 1} = 0$$

Since k is not divisible by n ensures that ω_n^k is not equal to 1 because $\omega_n^k = 1$ only when k is divisible by n. Thus denominator is not 0.

3. ***For any even integer $n > 0$***

$$\omega_n^{n/2} = -1 = \omega_2$$

Proof. $\omega_n^k = e^{2\pi i k/n}$

$$\omega_n^{n/2} = \omega^{2\pi i n/2n} = e^{\pi i} = \omega_2$$

$$= \cos(\pi) + i\sin(\pi)$$

$$= -1$$

$$\therefore \quad \omega_n^{n/2} = \omega_2 = -1$$

4. ***Halving Lemma***

If $n > 0$ is even, then the squares of the n complex nth roots of unity are the $n/2$ complex $(n/2)$th roots of unity.

Proof. We know $\omega_n^{n/2} = -1$

$$\omega_n^{n/2} \cdot \omega_n^k = -1 \cdot w_n^k$$

$$\omega_n^{n/2+k} = -\omega_n^k$$

Thus $(\omega_n^{n/2+k})^2 = (\omega_n^k)^2$

Thus, if we square all of the complex nth roots of unity then each $(n/2)$th root of unity is obtained exactly twice.

$$(\omega_n^k)^2 = (\omega_n^{2k})$$

$$= \omega_{n/2}^k \qquad [\because \omega_{dn}^{dk} = \omega_n^k]$$

35.5 Discrete Fourier Transform

We wish to evaluate a polynomial

$$A(x) = \sum_{j=0}^{n-1} a_j x^j \text{ of degree-bound } n \text{ at } \omega_n^k \text{ where } k = 0,1,2,\ldots,n-1.$$

i.e., $\omega_n^0, \omega_n^1, \omega_n^2, \ldots, \omega_n^{n-1}$

We assume that A is given in coefficient form $a = (a_0, a_1, \ldots, a_{n-1})$.

Thus DFT is to evaluate $A(x)$ at the nth roots of unity $\omega_n^0, \omega_n^1, \omega_n^2, \ldots, \omega_n^{n-1}$.

Let us define the results y_k, for $k = 0,1,2,\ldots,n-1$ by

$$y_k = A(\omega_n^k) \qquad i.e., \; x = \omega_n^k$$

$$= \sum_{j=0}^{n-1} a_j \omega_n^{kj}$$

The vector $y = (y_0, y_1, \ldots, y_{n-1})$ is the Discrete Fourier Transform (DFT) of the coefficient vector $(a_0, a_1, \ldots, a_{n-1})$.

We also write $y = \text{DFT}_n(a)$

35.6 Fast Fourier Transform (FFT)

The Fast Fourier Transform (FFT) algorithm computes a Discrete Fourier Transform (DFT) of an n-length vector in $O(n \lg n)$ time. In the FFT algorithm, we apply the divide-and-conquer approach to polynomial evaluation by observing that if n is even, we can divide a degree-$(n-1)$ polynomial

$$A(x) = a_0 + a_1 x + a_2 x^2 + \ldots + a_{n-1} x^{n-1}$$

into two degree $\left(\frac{n}{2} - 1\right)$ polynomials.

$$A^{[0]}(x) = a_0 + a_2 x + a_4 x^2 + \ldots + a_{n-2} x^{n/2-1}$$

$$A^{[1]}(x) = a_1 + a_3 x + a_5 x^2 + \ldots + a_{n-1} x^{n/2-1}$$

Note that $A^{[0]}$ contains all the even index coefficients of A and $A^{[1]}$ contains all the odd index coefficients and noting that we can combine these two polynomials into A, using the equation.

$$A(x) = A^{[0]}(x^2) + xA^{[1]}(x^2) \qquad \ldots(1)$$

So that the problem of evaluating $A(x)$ at w_n^k where $k = 0,1,2,\ldots,n-1$ reduces to

1. evaluating the degree $\left(\frac{n}{2}-1\right)$ polynomial $A^{[0]}(x)$ and $A^{[1]}(x)$ at the point $(\omega_n^k)^2$ *i.e.*,

   $$(\omega_n^0)^2, (\omega_n^1)^2, \ldots, (\omega_n^{n-1})^2$$

   because we know that if ω_n^k is a complex nth root of unity then $(\omega_n^k)^2$ is a complex $n/2$th root of unity. Thus, we can evaluate each $A^{[0]}(x)$ and $A^{[1]}(x)$ at $(\omega_n^k)^2$ values.

2. Combining the results according to the equation (1). This observation is the basis for the following procedure which computes the DFT of an n-element vector $a = (a_0, a_1, \ldots, a_{n-1})$ where for sake of simplicity, we assume that n is a power of 2.

```
FFT (a,w)
1.  n ← length[a]                    ▷ n is a power of 2.
2.  if n = 1
3.      then return a
4.  ω_n ← e^(2πi/n)
5.  x ← ω^0                          ▷ x will store powers of ω, initially x = 1.
6.  a^[0] ← (a_0, a_2, ... a_(n-2))
7.  a^[1] ← (a_1, a_3, ... a_(n-1))
8.  y^[0] ← FFT(a^[0], ω^2)          ▷ Recursive calls with ω^2 as (n / 2)th root of unity.
9.  y^[1] ← FFT(a^[0], ω^2)
10. for k ← 0 to n / 2 - 1
11.     do y_k ← y_k^[0] + x y_k^[1]
12.        y_(k+(n/2)) ← y_k^[0] - x y_k^[1]
13.        x ← x ω_n
14. return y
```


Line 2-3 represents the basis of recursion, the DFT of one element is the element if self. Since in this case

$$y_0 = a_0 \omega_1^0 = a_0 1 = a_0$$

Line 6-7 define the recursive the coefficient vectors for the polynomials $A^{[0]}$ and $A^{[1]}$. $\omega = \omega_n^k$

Line 8-9 perform the recursive DFT$_{n/2}$ computations setting for $k = 0,1,2,\ldots,\frac{n}{2}-1$ *i.e.*,

$$y_k^{[0]} = A^{[0]}(\omega_n^{2k})$$

$$y_k^{[1]} = A^{[1]}(\omega_n^{2k})$$

Lines 11-12 combine the results of the recursive DFT$_{n/2}$ calculations.

For $y_0, y_1, y_2, \ldots y_{(n/2)-1}$ line 11 yields.

$$\begin{aligned} y_k &= y_k^{[0]} + \omega_n^k y_k^{[1]} \\ &= A^{[0]}(\omega_n^{2k}) + \omega_n^k A^{[1]}(\omega_n^{2k}) \\ &= A(\omega_n^k) && \triangleright \text{ follows equation (1)} \end{aligned}$$

For $y_{n/2}, y_{(n/2)+1} \ldots y_{n-1}$, line 12 yields.

$$\begin{aligned} y_{k+(n/2)} &= y_k^{[0]} - \omega_n^k y_k^{[1]} \\ &= y_k^{[0]} + \omega_n^{k+(n/2)} y_k^{[1]} && [\because \omega_n^{k+(n/2)} = -\omega_n^k] \\ &= A^{[0]}(\omega_n^{2k}) + \omega_n^{k+(n/2)} A^{[1]}(\omega_n^{2k}) \\ &= A^{[0]}(\omega_n^{2k}\ \omega_n^n) + \omega_n^{k+(n/2)} A^{[1]}(\omega_n^{2k}\ \omega_n^n) && [\because \omega_n^n = 1] \\ &= A^{[0]}(\omega_n^{2k+n}) + \omega_n^{k+(n/2)} A^{[1]}(\omega_n^{2k+n}) \\ &= A(\omega_n^{k+(n/2)}) && \triangleright \text{ follows equation (1)} \end{aligned}$$

for each $k = 0,1,2,\ldots,(n/2)-1$.

Thus, the vector y returned by the FFT algorithm will store the values of $A(x)$ at each of the nth roots of unity.

Analyzing the FFT Algorithm

The FFT algorithm follows the divide-and-conquer paradigm, dividing the original problem of size n into two subproblem of size $n/2$, which are solved recursively. We assume that each

arithmetic operation performed by algorithm takes $O(1)$ time. The divide step as well as the combine step for merging the recursive solution, each take $O(n)$ time. Thus

$$T(n) = 2T(n/2) + bn \quad \text{for } b > 0$$

By solving, master theorem, we get

$$T(n) = O(n \lg n)$$

Exercise

1. Multiply the polynomial $A(x) = 7x^3 - x^2 + x - 10$ and $B(x) = 8x^3 - 16x + 3$.
2. Compute the DFT of the vector (0, 1, 2, 3,)
3. Describe the generalization of the FFT procedure to the case in which n is a power of 3. Give a recurrence for the running time and solve the recurrence.

CHAPTER

36

String Matching

36.1 Introduction

String matching consists of finding one or more generally, all of the occurrences of a pattern in a text. Finding a certain pattern in a text is a problem arises in text-editing programs and web "surfing". Given a text array, $T[1..n]$, of n character and a pattern array, $P[1..m]$ of m characters. The problem is to find an integer s, called **valid shift** where $0 \le s < n-m$ and $T[s+1...s+m] = P[1..m]$. In other words, to find whether P in T i.e., where P is a **substring of** T. The elements of P and T are character drawn from some finite alphabet such as {0, 1} or {A, B,...Z, a, b,....z}.

Given a string $T[1..n]$, the **substrings** is define as $T[i..j]$ for some $0 \le i \le j \le n-1$, that is, the string formed by the characters in T from index i to index j, inclusive. This means that a string is a substring of itself (simply take $i = 0$ and $j = m$).

The **proper substring** of string $T[1..n]$ is $T[1..j]$ for some $0 < i \le j \le n-1$. That is, we must have either $i > 0$ or $j < m-1$.

Using these definition, we can say given any string $T[1..n]$, the substrings are

$$T[i..j] = T[i]\ T[i+1]\ T[i+2]...T[j] \text{ for some } 0 \le i \le j \le n-1.$$

And proper substrings are

$$T[i..j] = T[i]T[i+1]T[i+2]...T[j] \text{ for some } 0 < i \le j \le n-1$$

Note that if $i > j$, then $T[i..j]$ is equal to the empty string or null, which has length zero.

We say that a string is a prefix of a string x, denoted $w \sqsubset x$, if $x = wy$ for some string $y \in \Sigma^*$. Note that if $w \sqsubset x$, then $|w| \le |x|$. Similarly, we say that a string w is a suffix of a string x, denoted $w \sqsupset x$, if $x = yw$ for some $y \in \Sigma^*$. It follows from $w \sqsupset x$ that $|w| \le |x|$. The empty string ε is both a suffix and a prefix of every string.

36.2 The Naive String-matching Algorithm

The naive approach simply test all the possible placement of Pattern $P[1..m]$ relative to text $T[1..n]$. Specifically, we try shift $s = 0,1,\ldots,n-m$, successively and for each shift, s. Compare $T[s+1..s+m]$ to $P[1..m]$.

The naive algorithm finds all valid shifts using a loop that checks the condition $P[1..m] = T[s+1..s+m]$ for each of the $n-m+1$ possible values of s.

NAIVE-STRING-MATCHER (T, P)

```
1.  n ← length[T]
2.  m ← length[P]
3.  for s ← 0 to n − m
4.      do if P[1..m] = T[s + 1..s + m]
5.      then print "Pattern occurs with shift" s
```

The for loop beginning on line 3 considers each possible shift explicitly. The test on line 4 determines whether the current shift is valid or not ; this test involves an implicit loop to check corresponding character positions until all positions match successfully or a mismatch is found. Line 5 prints out each valid shift s.

The naive string-matching procedure can be interpreted graphically as a sliding a pattern $P[1..m]$ over the text $T[1..n]$ and noting for which shift all of the characters in the pattern match the corresponding characters in the text.

In other to analysis the time of naive matching, we would like to implement above algorithm to understand the test involves in line 4.

Note that in this implementation, we use notation P[1 . . j] to denote the substring of P from index i to index j. That is, $P[1..j] = P[i]P[i+1]\ldots P[j]$.

NAIVE-STRING-MATCHER (T, P)

```
1.  n ← length [T]
2.  m ← length [P]
3.  for s ← 0 to n − m do
4.      j ← 1
5.      while j ≤ m and T[s + j] = p[j] do
6.          j ← j + 1
7.      If  j > m then
8.      return valid shift s
9.  return no valid shift exist        // i.e., there is no substring of T matching P.
```


Referring to implementation of naive matcher, we see that the for-loop in line 3 is executed at most $n-m+1$ times, and the while-loop in line 5 is executed at most m times. Therefore, the running time of the algorithm is $O((n-m+1)m)$, which is clearly $O(nm)$. Hence, in the worst case, when the length of the pattern, m are roughly equal, this algorithm runs in the quadratic time.

36.3 The Rabin-Karp Algorithm

The **Rabin-Karp algorithm** is a string searching algorithm created by **Michael O. Rabin** and **Richard M. Karp** that seeks a pattern, *i.e.* a substring, within a text by using **hashing**. This string searching algorithm calculates a hash value for the pattern, and for each M-character subsequence of text to be compared. If the hash values are unequal, the algorithm will calculate the hash value for next M-character sequence. If the hash values are equal, the algorithm will compare the pattern and the M-character sequence. In this way, there is only one comparison per text subsequence, and character matching is only needed when hash values match.

Thus, Rabin-Karp exploits the fact that if two strings are equal, their hash values are also equal. Thus, it would seem all we have to do is compute the hash value of the substring we're searching for, and then look for a substring with the same hash value.

However, there are two problems with this. First, because there are so many different strings, to keep the hash values small we have to assign some strings the same number. This means that if the hash values match, the strings might not match; we have to verify that they do, which can take a long time for long substrings. Luckily, a good hash function promises us that on most reasonable inputs, this won't happen too often, which keeps the average search time good.

This algorithm makes use of elementary number-theoretic notions such as the equivalence of two numbers modulo a third number. Given a pattern $P[1..m]$, we let p denote its corresponding decimal value. In a similar manner, given a text $T[1..n]$, we let t_s denote the decimal value of the length-m substring $T[s+1..s+m]$, for $s = 0,1,\ldots,n-m$. Certainly, $t_s = p$ if and only if $T[s+1..s+m] = P[1..m]$; thus, s is a valid shift if and only if $t_s = p$. If we could compute p in time $O(m)$ and all of the t_i values in a total of $O(n)$ time, then we could determine all valid shifts s in time $O(n)$ by comparing p with each of the t_s's. We can compute p in time $O(m)$ using **Horner's rule.**

$$p = P[m] + 10(P[m-1] + 10(P[m-2] + \ldots + 10(P[2] + 10P[1]))\ldots))$$

The value t_0 can be similarly computed from $T[1..m]$ in time $O(m)$.

To compute the remaining values $t_1, t_2, \ldots, t_{n-m}$ in time $O(n-m)$, it suffices to observe that t_{s+1} can be computed from t_s in constant time, since

$$t_{s+1} = 10(t_s - 10^{m-1}T[s+1]) + T[s+m+1]$$

Subtracting $10^{m-1}T[s+1]$ removes the high-order digit from t_s, multiplying the result by 10 shifts the number left one position, and adding $T[s+m+1]$ brings in the appropriate low-order digit.

In general, with a d-ary alphabet $\{0, 1, \ldots, d-1\}$, we choose q so that $d\,q$ fits within a computer word and adjust the above recurrence equation to work modulo q, so that it becomes

$$t_{s+1} = (d(t_s - T[s+1]h) + T[s+m+1]) \bmod q,$$

where $h \equiv d^{m-1} \pmod q$ is the value of the digit "1" in the high-order position of an m-digit text window. Since $t_s \equiv p \pmod q$ does not imply that $t_s = p$. On the other hand, if $t_s \neq p \pmod q$, then we definitely have that $t_s \neq p$, so that shift s is invalid. We can thus use the test $t_s \equiv p \pmod q$ as a fast heuristic test to rule out invalid shifts s. Any shift s for which $t_s \equiv p \pmod q$ must be tested further to see if s is really valid or we just have a spurious hit. This testing can be done by explicitly checking the condition $P[1..m] = T[s+1..s+m]$. If q is large enough, then we can hope that spurious hits occur infrequently enough that the cost of the extra checking is low.

The following procedure makes these ideas precise. The inputs to the procedure are the text T, the pattern P, the radix d and the prime q to use.

RABIN-KARP-MATCHER (T, P, d, q)

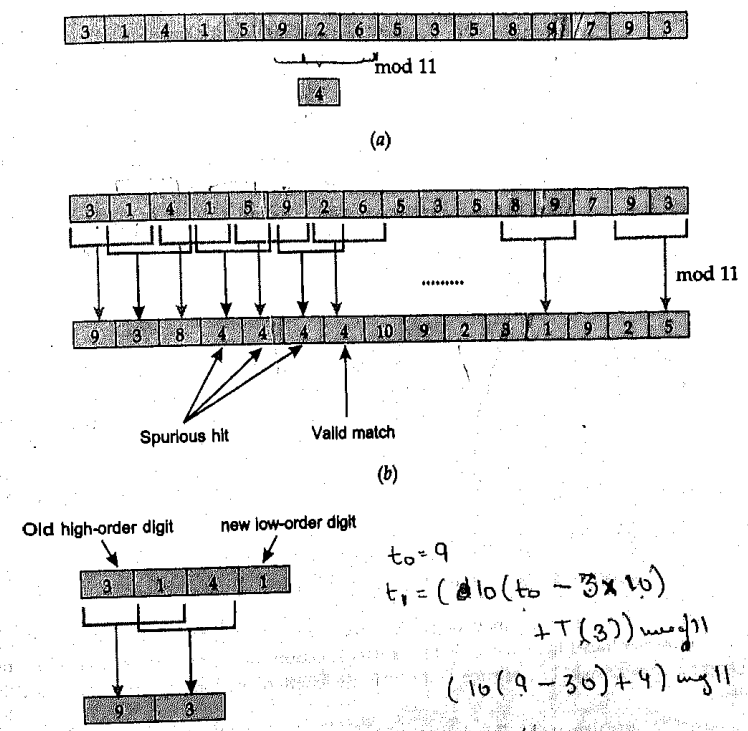
```
1.  n ← length[T]
2.  m ← length[P]
3.  h ← d^(m-1) mod q
4.  p ← 0
5.  t0 ← 0
6.  for i ← 1 to m
7.      do p ← (dp + P[i]) mod q
8.      t0 ← (dt0 + T[i]) mod q
9.  for s ← 0 to n − m
10.     do if p = ts
11.           then if P[1..m] = T[s+1..s+m]
12.           then "Pattern occurs with shift" s
13.     if s < n − m
14.  then ts+1 ← (d(ts − T[s+1]h) + T[s+m+1]) mod q
```

The procedure RABIN-KARP-MATCHER works as follows. Line 3 initializes h to the value of the high-order digit position of an m-digit window. Lines 4-8 compute p as the value of $P[1..m] \bmod q$ and t_0 as the value of $T[1..m] \bmod q$. The for loop beginning on line 9 iterates through all possible shifts s. The loop has the following invariant: whenever line 10 is executed, $t_s = T[s+1..s+m] \bmod q$. If $p = t_s$ in line 10 (a "hit"), then we check to see if $P[1..m] = T[s+1..s+m]$ in line 11 to rule out the possibility of a spurious hit. Any valid shifts found are printed out on line 12. If $s < n-m$ (checked in line 13), then the for loop is to be executed at least one more time, and so line 14 is first executed to ensure that the loop invariant holds when line 10 is again reached. Line 14 computes the value of $t_{s+1} \bmod q$ from the value of $t_s \bmod q$ in constant time.

The running time of RABIN-KARP-MATCHER is $\Theta((n-m+1)m)$ in the worst case, since (like the naive string-matching algorithm) the Rabin-Karp algorithm explicitly verifies every valid shift. If $P = a^m$ and $T = a^n$, then the verifications take time $\Theta((n-m+1)m)$, since each of the $n-m+1$ possible shifts is valid.

Example. *For string matching working module $q = 11$, how many spurious hits does the Rabin-Karp matcher encounter in the text $T = 3141592653589793$, when looking for the pattern $P = 26$?*

Solution.



$t_0 = 9$

$t_1 = (10(t_0 - 3 \times 10) + T(3)) \bmod 11$

$(10(9 - 30) + 4) \bmod 11$

Figure 36.1

Here $m = 2$ so

$$\begin{aligned} t_{s+1} &= 10(31 - 10^{2-1} \cdot 3) + 4 \\ &= 10(31 - 30) + 4 \\ &= 14 \end{aligned}$$

Fig. 36.1(a) shows a text string and a window of length 2. The numerical value of the window is computed modulo 11 yielding the value 4.

The same text string with values computed modulo 11 for each possible position of a length two window.

Four such window are found. The first three windows are spurious hit because in these modulo is same but the pattern is not same.

Thus, the total number of spurious hits are 3.

36.4 String Matching with Finite Automata

String-matching automata are very efficient because they examine each text character exactly once, taking constant time per text character. The time used-after the automaton is built - is therefore $\theta(n)$. The time to build the automaton, however, can be large if Σ is large. A finite automaton M is a 5-tuple $(Q, q_0, A, \Sigma, \delta)$, where

- Q is a finite set of states,
- $q_0 \in Q$ is the start state,
- $A \subseteq Q$ is a distinguished set of accepting states,
- Σ is a finite input alphabet,
- δ is a function from $Q \times \Sigma$ into Q called the transition function of M.

The finite automaton begins in state q_0 and reads the characters of its input string one at a time. If the automaton is in state q and reads input character a, it moves from state q to state $\delta(q, a)$. Whenever its current state q is a member of A, the machine M is said to have accepted the string read so far. An input that is not accepted is said to be rejected. A finite automaton M induces a function ϕ called the **final-state function**, from Σ^* to Q such that $\phi(w)$ is the state M ends up in after scanning the string w. Thus, M accepts a string w if and only if $\phi(w) \in A$. The function f is defined as

$$\phi(\varepsilon) = q_0$$

$$\phi(wa) = \delta(\phi(w), a) \text{ for } w \in \Sigma^*,\ a \in \Sigma$$

In order to specify the string-matching automaton corresponding to a given pattern $P[1..m]$, we first define an auxiliary function σ, called the **suffix function** corresponding to P. The function σ is a mapping from Σ^* to $\{0,1,\ldots,m\}$ such that $\sigma(x)$ is the length of the longest prefix of P that is a suffix of x:

$$\sigma(x) = \max\{k : P_k \sqsupset x\}$$

We define the string-matching automaton corresponding to a given pattern $P[1..m]$ as follows.

The state set Q is $\{0,1,\ldots,m\}$. The start state q_0 is state 0, and state m is the only accepting state.

The transition function δ is defined by the following equation, for any state q and character a:

$$\delta(q, a) = \sigma(P_q a)$$

As for any string-matching automaton for a pattern of length m, the state set Q is $\{0,1,\ldots,m\}$, the start state is 0, and the only accepting state is state m.

FINITE-AUTOMATON-MATCHER (T, δ, m)

```
n ← length[T]
q ← 0
for i ← 1 to n
    do q ← δ(q, T[i])
        if q = m
            then s ← i − m
                print "Pattern occurs with shift" s
```

The simple loop structure of FINITE-AUTOMATON-MATCHER implies that its running time on a text string of length n is $O(n)$.

Computing the Transition Function

The following procedure computes the transition function δ from a given pattern $P[1..m]$

COMPUTE-TRANSITION-FUNCTION (P, Σ)

```
m ← length[P]
for q ← 0 to m
    do for each character a ∈ Σ*
    do k ← min(m + 1, q + 2)
repeat k ← k − 1
until
    δ(q, a) ← k
return δ
```

36.5 The Knuth-Morris-Pratt (KMP) Algorithm

Knuth, Morris and Pratt discovered first linear time string-matching algorithm by following a tight analysis of the naive algorithm. Knuth-Morris-Pratt algorithm keeps the information that naive approach wasted gathered during the scan of the text. By avoiding this waste of information, it achieves a running time of $O(n+m)$, which is achieved by using the auxiliary function π. That is, in the worst-case Knuth-Morris-Pratt algorithm we have to examine all the characters in the text and pattern at least once.

The function π computes the shifting of pattern matches within itself. It can be observed that if we know that how the pattern matches shifts against itself, then we can slide the pattern more characters towards right than just one character.

The basic idea is to slide the pattern towards the right along the string so that the longest prefix of P that we have matched matches the longest suffix of T that we have already matched. If the longest prefix of P that matches a suffix of T is nothing, then we slide the whole pattern towards right. The algorithm computing prefix function is as follows.

```
COMPUTE-PREFIX-FUNCTION (P)
m ← length [P]
π[1] ← 0
k ← 0
for q ← 2 to m
    do while k > 0 and P[k+1] ≠ P[q]
        do k ← π[k]
    if P[k+1] = P[q]
        then k ← k+1
    π[q] ← k
return π
```

The Knuth-Morris-Pratt matching algorithm is given in pseudocode below as the procedure KMP-MATCHER. KMP-MATCHER calls the auxiliary procedure COMPUTE-PREFIX-FUNCTION to compute π

```
KMP-MATCHER (T,P)
n ← length [T]
m ← length [P]
π ← COMPUTE-PREFIX-FUNCTION (P)
q ← 0
for i ← 1 to n
    do while q > 0 and P[q+1] ≠ T[i]
        do q ← π[q]
    if P[q+1] = T[i]
        then q ← q+1
if q = m
then print "Pattern occurs with shift" i − m
q ← π[q]
```

Example. *Compute the prefix function π for the pattern ababbabbabbababbabb when the alphabet is $\Sigma = \{a, b\}$.*

Solution. Here pattern $P = ababbabbabbababbabb$

length $[P] = 19$ so $m = 19$

i	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19
$P[i]$	a	b	a	b	b	a	b	b	a	b	b	a	b	a	b	b	a	b	b
$\pi(i)$	0	0	1	2	0	1	2	0	1	2	0	1	2	3	4	5	6	7	8

Initially $\pi(1) = 0$ and $k = 0$ and $q = 2$ and $P[k+1] \neq P[q]$. So $\pi[2] = 0$

Now $P[k+1] = P[q]$ for $q = 3$

then $k = k+1$ *i.e.*, $k = 1$ and $\pi[3] = 1$

then $k > 0$ and $P[k+1] = P[q]$ for $q = 4$

then $k = k+1$ *i.e.*, $k = 2$

and $\pi[4] = 2$

Now $q = 5$ $k = 2$

i.e., $k > 0$ and $P[k+1] \neq P[q]$

$a \neq b$

$k = \pi[2]$

$k = 0$

Now, $P[1] \neq P[q]$ so $\pi[5] = 0$

Similarly, we compute other prefix functions.

36.6 The Boyer-Moore Algorithm

The Boyer-Moore algorithm is consider the most efficient string-matching algorithm in usual applications, for example, in text editors and commands substitutions. The reason is that it woks the fastest when the alphabet is moderately sized and the pattern is relatively long. It was developed by **Bob Boyer** and **J Strother Moore** in 1977

The algorithm scans the characters of the pattern from right to left beginning with the rightmost character, which is also referred to as "**looking glass heuristic**". If the text symbol that is compared with the rightmost pattern symbol does not occur in the pattern at all, then the pattern can be shifted by m positions behind this text symbol.

Example

0	1	2	3	4	5	6	7	8	9	...
a	b	b	a	d	a	b	a	c	b	a
b	a	b	a	c						
					b	a	b	a	c	

The first comparison $d - c$ at position 4 produces a mismatch. The text symbol d does not occur in the pattern. Therefore, the pattern cannot match at any of the positions 0, ..., 4, since all corresponding windows contain ad. The pattern can be shifted to position 5. The best case for the Boyer-Moore algorithm is attained if at each attempt the first compared text symbol does not occur in the pattern. Then the algorithm requires only $O(n/m)$ comparisons.

For determining the smallest possible shifts, Boyer-Moore works with two different preprocessing strategies. Each time when a mismatch occurs the algorithm computes both and chooses the largest possible shift, thus making use of the most efficient strategy for each individual case.

Bad character Heuristics

This method is called bad character heuristics. It can also be applied if **the bad character**, i.e. the text symbol that causes a mismatch, occurs somewhere else in the pattern. Then the pattern can be shifted so that it is aligned to this text symbol. The next example illustrates this situation.

Example

Comparison $b-c$ causes a mismatch. Text symbol b occurs in the pattern at positions 0 and 2. The pattern can be shifted so that the rightmost b in the pattern is aligned to text symbol b.

0	1	2	3	4	5	6	7	8	9	...
A	*b*	*b*	*a*	*b*	*a*	*b*	*a*	*c*	*b*	*a*
B	*a*	*b*	*a*	*c*						
		b	*a*	*b*	*a*	*c*				

Good Suffix Heuristics

Sometimes the bad character heuristics fails. In the following situation the comparison $a-b$ causes a mismatch. An alignment of the rightmost occurence of the pattern symbol a with the text symbol a would produce a **negative shift**. Instead, a shift by 1 would be possible. However, in this case it is better to derive the maximum possible shift distance from the structure of the pattern. This method is called **good suffix heuristics**.

Example

0	1	2	3	4	5	6	7	8	9	...
A	*b*	*a*	*a*	*b*	*a*	*b*	*a*	*c*	*b*	*a*
c	*a*	*b*	*a*	*b*						
		c	*a*	*b*	*a*	*b*				

The suffix ab has matched. The pattern can be shifted until the next occurence of ab in the pattern is aligned to the text symbols ab, *i.e.* to position 2.

In the following situation the suffix ab has matched. There is no other occurence of ab in the pattern.Therefore, the pattern can be shifted behind ab, *i.e.* to position 5.

BOYER-MOORE-MATCHER (T, P, Σ)

```
1.  n ← length[T]
2.  m ← length[P]
3.  λ ← COMPUTE-LAST-OCCURRENCE-FUNCTION (P, m, Σ)
4.  γ ← COMPUTE-GOOD-SUFFIX-FUNCTION (P, m)
5.  s ← 0
6.  while s ≤ n − m
7.      do j ← m
8.          while j > 0 and P[j] = T[s + j]
9.              do j ← j − 1
```



```
10.         if j = 0
11.             then print "Pattern occurs at shift" s
12.                 s ← s + γ[0]
13.             else s ← s + max(γ[j], j − λ[T[s + j]])
```

COMPUTE-LAST-OCCURRENCE-FUNCTION (P, m, Σ)

```
1.  for each character a ∈ Σ
2.      do λ[a] = 0
3.  for j ← 1 to m
4.      do λ[P[j]] ← j
5.  return λ
```

COMPUTE-GOOD-SUFFIX-FUNCTION (P, m)

```
1   π ← COMPUTE-PREFIX-FUNCTION (P)
2   P' ← reverse(P)
3   π' ← COMPUTE-PREFIX-FUNCTION (P')
4   for j ← 0 to m
5       do γ[j] ← m − π[m]
6   for l ← 1 to m
7       do j ← m − π'[l]
8           if γ[j] > l − π'[l]
9               then γ[j] ← 1 − π'[l]
10  return γ
```

The worst-case running time of the Boyer-Moore algorithm is clearly $O((n-m+1)m+|\Sigma|)$, since COMPUTE-LAST-OCCURRENCE-FUNCTION takes time $O(m+|\Sigma|)$, COMPUTE-GOOD-SUFFIX-FUNCTION takes time $O(m)$, and the Boyer-Moore algorithm (like the Rabin-Karp algorithm) spends $O(m)$ time validating each valid shift s. In practice, however, the Boyer-Moore algorithm is often the algorithm of choice.

Example. *Suppose that all characters in the pattern P are different. Show how to accelerate NAIVE-STRING-MATCHER to run in time O(n) on an n-character text T.*

Solution. Assume all the characters of P are different. A mismatch with T a position i of P in line 4 of NAIVE-STRING-MATCHER then implies that we can continue our search from position $s+i$ in T. Thus a linear search of T is a sufficient.

Example. *Suppose we allow the pattern P to contain occurrences of a gap character ◊ that can match an arbitrary string of characters. (even one of zero length).*

For example, the pattern ab◊ba◊c occurs in the text cabccbacbacab as

$$c\ \underbrace{ab}_{ab}\ \underbrace{cc}_{\Diamond}\ \underbrace{ba}_{ba}\ \underbrace{cba}_{\Diamond}\ \underbrace{c}_{c}\ ab$$

Note that the gap character may occur an arbitrary number of times in the pattern but is assumed not to occur at all in the text. Give a polynomial time algorithm to determine if such a pattern P occurs in a given text T, and analyze the running time of your algorithm.

Solution. To determine if a pattern P with gap characters exists in T. We partition P into substrings $P_1, P_2, P_3, \ldots, P_k$ determine by the gap characters.

Search for P_1 and if found continue searching for P_2 and so on. This clearly find a apttern if one exists.

Example. *Given a pattern P containing gap characters. Show how to build a finite automaton that can find an occurrence of P in a text T in O(n) time, where* $n = |T|$.

Solution. We can construct the finite automaton corresponding to a pattern. P using the same idea as in above question. Partition P into substrings $P_1, P_2, \ldots P_k$ determined by the gap characters. Construct finite automatons for each P_i and combine sequentially *i.e.,* the accepting state of P_i, $1 \geq i < k$ is no longer accepting but has a single transition to P_{i+1}.

Example. *Give a linear-time algorithm to determine if a text T is a cycle rotation of another string T'. For example arc and car are cyclic rotations of each other.*

Solution. We can determine if T is a cycle rotation of T' matching T' against TT.

Exercise

1. Show that the worst-case time for the naive string matcher to find the first occurrence of a pattern in a text is $\theta((n-m+1)(m-1))$.
2. Show the comparisons the naive string matcher makes for the pattern $P = 10001$ in the text $T =$ 0000100010010
3. How would you extend the Rabin-karp method to the problem of searching a text string for an occurrence of any once of a given set of k patterns ?
4. Draw a state-transition diagram for a string-matching automaton for the pattern *ababbabbababbababbabb* over the alphabet $\sum = \{a,b\}$.
5. Compute the prefix function π for the pattern *ababbabbababbababbabb* when the alphabet is $\sum = \{a,b\}$.
6. Explain how to determine the occurrences of pattern P in the text T by examining the π function for the string PT. (the string of length $m+n$ that is the concatenation of P and T).
7. Compute the λ and γ functions for the pattern $P = 0101101201$ and the alphabet $\sum = \{0,1,2\}$.

CHAPTER 37

Computational Geometry

37.1 Introduction

Computational geometry is a term claimed by a number of different groups. The term was coined perhaps first by **Marvin Minsky** in his book **"Perceptrons"**, which was about pattern recognition, and it has also been used often to describe algorithms for manipulating curves and surfaces in solid modeling.

In computer science, computational geometry is the study of algorithms to solve problems stated in terms of geometry. Some purely geometrical problems arise out of the study of computational geometry. In modern engineering and mathematics, computational geometry has applications in, among other fields, computer graphics, robotics, VLSI design, computer-aided design, and statistics. The input to a computational-geometry problem is typically a description of a set of geometric objects, such as a set of points, a set of line segments, or the vertices of a polygon in counterclockwise order. The output is often a response to a query about the objects, such as whether any of the lines intersect, or perhaps a new geometric object, such as the **convex hull** (smallest enclosing convex polygon) of the set of points.

There are many fields of computer science that deal with solving problems of a geometric nature. These include computer graphics, computer vision and image processing, robotics, computer-aided design and manufacturing, computational fluid dynamics, and geographic information systems, to name a few. One of the goals of computational geometry is to provide the basic geometric tools needed from which application areas can then build their programs. There has been significant progress made towards this goal, but it is still far from being realized.

37.2 Strengths and Limitations of Computational Geometry

Strengths of Computational Geometry

1. Development of Geometric Tools
2. Emphasis on Provable Efficiency
3. Emphasis on Correctness/Robustness
4. Linkage to Discrete Combinatorial Geometry

Limitations of Computational Geometry

1. Emphasis on discrete geometry
2. Emphasis on flat objects
3. Emphasis on low-dimensional spaces

37.3 Polygons

A polygon is just a collection of line segments, forming a cycle, and not crossing each other. We can represent it as a sequence of points, each of which is just a pair of coordinates. For instance the points (0, 0), (0, 1), (1, 1), (1, 0) form a square. The line segments of the polygon connect adjacent points in the list, together with one additional segment connecting the first and last point.

Not all sequences of points form a polygon ; for instance the points (0, −1), (0,1), (1,0), (0, −1) would result in two segments that cross each other. Sometimes we use the phrase simple polygon to emphasize the requirement that no two segments cross.

Testing if a point in a polygon

It is a famous theorem (the **Jordan curve theorem**) that any polygon cuts the plane into exactly two connected pieces : the *inside* and the *outside*. (The inside always has some finite size, while the outside contains points arbitrarily far from the polygon.) Actually, the Jordan curve theorem is more generally true of certain curves in the plane, not just shapes formed by straight line segments ; the more general fact is often proved by approximating these curves by polygons.

For uncomplicated enough polygons, it's easy to see by eyes, which parts of the place are inside and which are outside. But this is not always easy. For instance is the marked point inside the following polygon ?

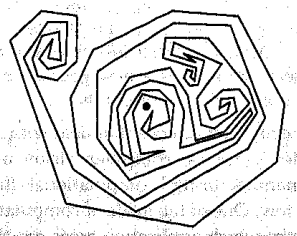


Figure 37.1

Seemingly even more difficult, we'd like to answer questions like this on a computer which doesn't have build into it the powerful visual processing system that we have, and can only deal with this sort of problem by translating it to a collection of numbers. The general problem we'd like to solve is, given a point (x, y) (represented by those two numbers) and a polygon P (represented by its sequence of vertices), is (x, y) in P, on the boundary, or outside ?

This is a commonly occurring problem. For instance if we want to display a polygon on a computer screen, we need to be able to test whether each pixel of the screen corresponds to a point that's inside or outside the polygon, so we can tell what color to draw it.

Fortunately the problem has a simple and elegant answer. Just draw a ray (portion of a line extending infinitely in one direction) down (or in any other direction) from (x, y) :

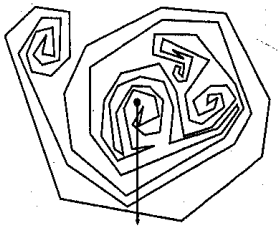


Figure 37.2

Every time the polygon crosses this ray, it separates a part of the ray that's inside the polygon from a part that's outside. Eventually the ray will get far from the polygon, at which point we know that part is outside the polygon. We can then work backwards from there, declaring different parts of the ray to be inside or outside alternately at each crossing. Actually, we don't even need to look at what order these crossings occur in ; all we really need to know is whether there's an even or odd number of them.

In the example shown above, there are eight crossings, so the point is outside the polygon. We can now write a rough outline of pseudo-code for this problem :

```
int crossings = 0
for (each line segment of the polygon)
    if (ray down from (x, y) crosses segment)
        crossings++ ;
if (crossings is odd)
    return (inside) ;
else return (outside) ;
```

odd inside

even outside

37.4 Convex Polygon

A figure is convex if every line segments drawn between any two points inside the figure lies entirely inside the figure. A figure that is not convex is called a concave figure.

Example :

The following figures are convex.

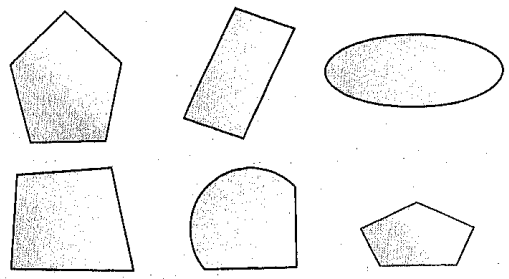


Figure 37.3

The following figures are concave.

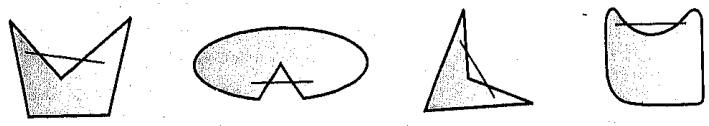


Figure 37.4

A **convex polygon** can also be defined formally as having the property that any two points inside the polygon can be connected by a line segment that doesn't cross the polygon. A polygon vertex is *convex* if its interior angle $\leq \pi (180°)$. It is *reflex* if its interior angle $> \pi$. In a convex polygon, all the vertices are convex. In other words, any polygon with a reflex vertex is not convex.

37.5 Convex Hulls

A convex hull is an important structure in geometry that can be used in the construction of many other geometric structures. The convex hull of a set S of points in the plane is defined to be the smallest convex polygon containing all the points of S. The vertices of the convex hull of a set S of points form a (not necessarily proper) subset of S.

There are two variants of the convex hull problem :

1. Obtain the vertices of the convex hull (these vertices are also called extreme points) and
2. Obtain the vertices of the convex hull in same order (clockwise or anticlockwise).

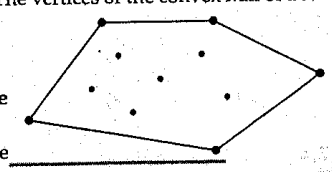


Figure 37.4A Convex hull

Here is a simple algorithm for obtaining the extreme points of a given set S of points in the plane. To check whether a particular point $p \in S$ is extreme, look at each possible triplet of points and see whether p lies in the triangle formed by these three points. If p lies in any such triangle, it is

not extreme, otherwise it is. Testing whether p lies in a given triangle can be done in $\theta(1)$ time. Since there are $\theta(n^3)$ possible triangles, it takes $\theta(n^3)$ time to determine whether a given input is an extreme point or not. Since there are 'n' points, this algorithm runs in a total of $\theta(n^4)$ time.

Using divide and conquer, we can solve both versions of the convex hull problem in $\theta(n \log n)$ time. Given a set $S = \{p_1, \ldots\ldots p_n\}$ ..... .

Given a set $S = \{p_1, p_2, \ldots, p_n\}$ of points in the plane, the convex **hull** of a set S of points is the smallest convex polygon P for which each point in S is either on the boundary of P or in its interior. We denote the convex hull of S by $CH(S)$. The intersection of convex sets is convex, so the convex hull is a convex set. It is also a polytope, a bounded intersection of half spaces, which in two dimensions means that it is a polygon. Another useful characterization of the convex hull is as a convex closure : take all possible convex combinations of points in P, and the result is the convex hull.

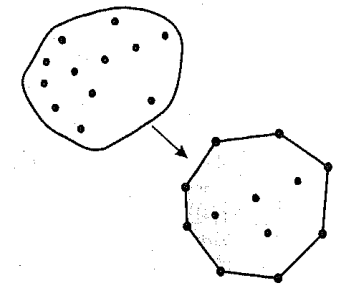


Figure 37.5

Here, we shall present two algorithms that compute the convex hull of a set of n points. Both algorithms output the vertices of the convex hull in counterclockwise order. The first, known as **Graham's scan**, runs in $O(n \lg n)$ time. The second, called **Jarvis's march**, runs in $O(nh)$ time, where h is the number of vertices of the convex hull.

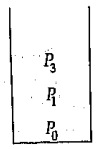
37.6 Graham's Scan Algorithm

In 1972, **R. L. Graham** proposed an efficient algorithm for planar convex hull. Historically, this is the first publication that showed convex hull computation in $O(n \log n)$ time in the worst case. In 1981, **A. C. Yao** proved that it is optimal in the worst-case sense. The problem with Graham's algorithm is that it has no obvious extension to three dimensions. The reason is that the Graham's scan depends on angular sorting, which has no direct counterpart in three dimensions. In this, first we select a base point p_0, normally this is the point with minimum y-coordinate. We select leftmost point in the set in case of the tie. Next, sort the points of Q lexicographically by polar angle, measured in radians. Interior points on the ray cannot be a convex hull points and remove these points during sort. Sort remaining points in counterclockwise order with respect to p_0. Push each point of set Q onto the stack once and the points that are not of $CH(Q)$ are eventually popped from the stack.

Actually, the algorithm works in *three* phases :

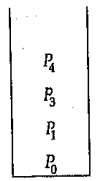
1. Find an extreme point. This point will be the pivot, is guaranteed to be on the hull, and is chosen to be the point with smallest y-coordinate.
2. Sort the points in order of increasing angle about the pivot. We end up with a star-shaped polygon (one in which one special point, in this case the pivot, can "see" the whole polygon).
3. Build the hull, by marching around the star-shaped poly, adding edges when we make a left turn, and backtracking when we make a right turn.

Now $\angle P_0, P_1, P_3$ is left turn, then push P_3 in the stack. Now the elements in the stack are



Stack S

$\angle P_1, P_3, P_4$ is counter-clock wise *i.e.*, left turn. So push p_4 in the stack and the elements in the stack are



Stack S

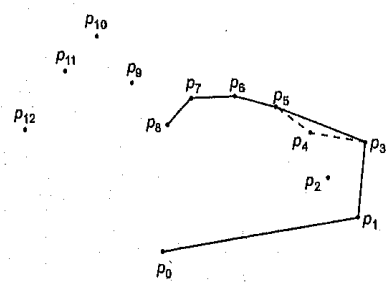
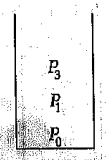


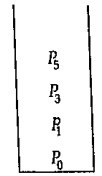
Figure 37.6 (c)

$\angle P_3\ P_4\ P_5$ clockwise turn *i.e.*, non left turn. So pop the stack. Now the elements in the stack are



Stack S

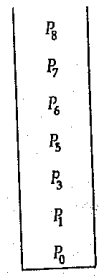
$\angle P_1\ P_3\ P_5$ is left turn. So push P_5 in the stack, now stack S contains



Stack S

$\angle P_3\ P_5\ P_6$ is also left turn so push P_6 in the stack.

$\angle P_5\ P_6\ P_7$ is also left turn push P_7 in the stack and $\angle P_6\ P_7\ P_8$ is also left turn push P_8 in the stack and now, the stack S contains



Stack S

$\angle P_7\ P_8\ P_9$ is clockwise turn *i.e.*, non left turn. So pop the stack then $\angle P_6\ P_7\ P_9$ is also non left turn, so again pop the stack. Now, the stack S contains.

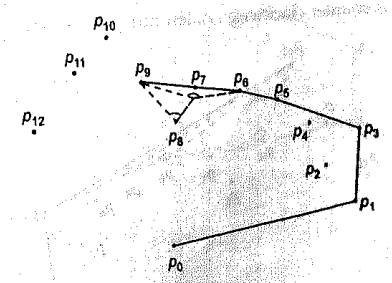
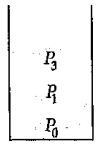


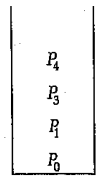
Figure 37.6 (d)

Now $\angle P_0, P_1, P_3$ is left turn, then push P_3 in the stack. Now the elements in the stack are



Stack S

$\angle P_1, P_3, P_4$ is counter-clock wise *i.e.*, left turn. So push p_4 in the stack and the elements in the stack are



Stack S

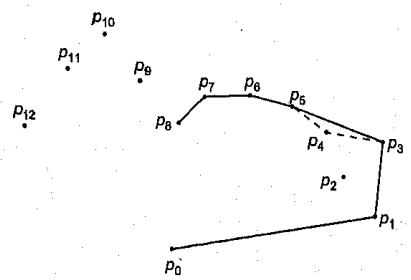
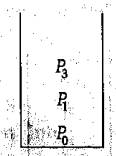


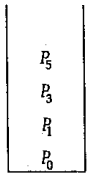
Figure 37.6 (c)

$\angle P_3\ P_4\ P_5$ clockwise turn *i.e.*, non left turn. So pop the stack. Now the elements in the stack are



Stack S

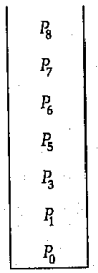
$\angle P_1\ P_3\ P_5$ is left turn. So push P_5 in the stack, now stack S. contains



Stack S

$\angle P_3\ P_5\ P_6$ is also left turn so push P_6 in the stack.

$\angle P_5\ P_6\ P_7$ is also left turn push P_7 in the stack and $\angle P_6\ P_7\ P_8$ is also left turn push P_8 in the stack and now, the stack S contains



Stack S

$\angle P_7\ P_8\ P_9$ is clockwise turn *i.e.*, non left turn. So pop the stack then $\angle P_6\ P_7\ P_9$ is also non left turn, so again pop the stack. Now, the stack S contains.

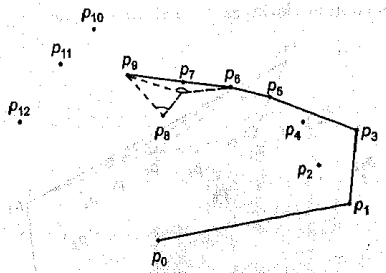


Figure 37.6 (d)

P_6
P_5
P_3
P_1
P_0

Stack S

$\angle P_5\ P_6\ P_9$ is left turn so push P_9 in the stack and now the elements of stack S are

P_9
P_6
P_5
P_3
P_1
P_0

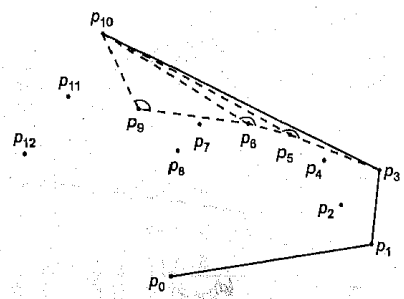
Stack S

Now $\angle P_6\ P_9\ P_{10}$ is non left turn so pop P_9 and $\angle P_5\ P_6\ P_{10}$ is also non left turn and $\angle P_3\ P_5\ P_{10}$ is non left turn also, thus pop P_6 and P_5 respectively and the stack S contains.

P_3
P_1
P_0

Stack S

Now, $\angle P_1, P_3, P_{10}$ is counter clockwise *i.e.*, left turn. So push P_{10} into the stack



(e)

Figure 37.6

Now the stack contains.

P_{10}
P_3
P_1
P_0

Stack S

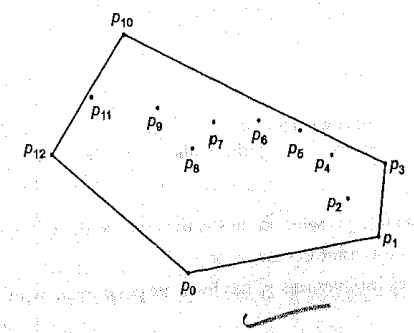
$\angle P_3\ P_{10}\ P_{11}$ is counter clock wise that is left turn so push P_{11} into the stack and stack contains.

P_{11}
P_{10}
P_3
P_1
P_0

Stack S

$\angle P_{10}\ P_{11}\ P_{12}$ is clock wise that is not left turn so pop the element P_{11} and $\angle P_3\ P_{10}\ P_{12}$ is anticlockwise.

Hence push this into stack S and the final convex hull returned by the produce is show in the figure.



(f)

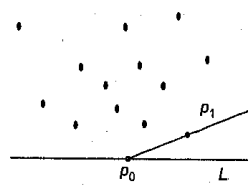
Figure 37.6

37.7 Jarvis's March Algorithm

Jarvis's march computes the convex hull of a set Q of points by a technique known as package wrapping (or gift wrapping). The algorithm runs in time $O(nh)$, where h is the number of vertices of $CH(Q)$. The algorithm operates by considering any one point that is on the hull, say, the lowest point. We then find the "next" edge on the hull in counterclockwise order. Assuming that $p(k)$ and $p(k-1)$ were the last two points added to the hull, compute the point q that maximizes the angle $[p(k-1)\,p(k)\,q]$. Thus, we can find the point q in $O(n)$ time. After repeating this h times, we will return back to the starting point and we are done. Thus, the overall running time is $O(nh)$. Note that if h is $o(\log n)$ (asymptotically smaller than $\log n$) then this is a better method than Graham's algorithm Jarvis's march is asymptotically faster than Graham's scan.

The basic idea is as follows :

- Locating the point p_0 with minimum y-coordinate (or Start at some extreme point, which is guaranteed to be on the hull).
- Let L be the horizontal line through p_0, L is clearly tangent to $CH(S)$ at p_0. We orient L from left to right. Then we perform "wrapping" step to locate point p_1 that forms the smallest counterclockwise angle with L Point p_1 must also be on $CH(S)$ (See Fig).
- Repeat the wrapping step at p_1 with line pp_1 until we return to the point p_0



Jarvis' march

Figure 37.7

Because this process marches around the hull in counter-clockwise order, like a ribbon wrapping itself around the points, this algorithm also called the "gift-wrapping" algorithm.

Algorithm Jarvis March

- First, a base point p_0 is selected, this is the point with the minimum y-coordinate.
  - Select leftmost point in case of tie.
- The next convex hull vertices p_1 has the least polar angle w.r.t. the positive horizontal ray from p_0.
  - Measure in counterclockwise direction.
  - If tie, choose the farthest such point.
- Vertices $p_2, p_3, \ldots, p_k$ are picked similarly until $y_k = y_{max}$

- p_{i+1} has least polar angle w.r.t. positive ray from p_0.
- If tie, choose the farthest such point.

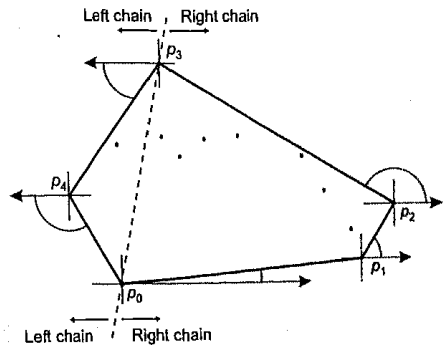


Figure 37.9

- The sequence $p_0, p_1, \ldots, p_k$ is right chain of $CH(Q)$
- To choose the left chain of $CH(Q)$ start with p_k.
  - Choose p_{k+1} as the point with least polar angle w.r.t. the negative ray from p_k.
  - Again measure counterclockwise direction.
  - If tie occurs, choose the farthest such point.
  - Continue picking $p_{k+1}, p_{k+2}, \ldots, p_t$ in same fashion until obtain $P_t = p_0$.

Complexity of Jarvis March

For each vertex p belongs to $CH(Q)$, it takes

- $O(1)$ time to compare polar angles.
- $O(n)$ time to find minimum polar angel.
- $O(n)$ total time.

If $CH(Q)$ has h vertices, then running time $O(nh)$. If $h = o(\lg n)$, then this algorithm is asymptotically faster than the Graham's scan. If points in set Q are generated by random generator, then we expect $h = c \lg n$ for $c \approx 1$.

In practice, Jarvis march is normally faster than Graham's scan on most application. Worst case occurs when $O(n)$ points lie on the convex hull *i.e.*, all points lie on the circle.

Exercise

1. Show how to determine in $O(n^2 \lg n)$ time whether any three points in a set of n points are collinear.
2. Show how to compute the area of an n-vertex simple but not necessarily convex, polygon in $\theta(n)$ time.
3. Write an algorithm to find whether two polygons intersect.
4. Give an algorithm for finding Convex Hull of a star shaped polygon.
5. Give a divide-and conquer method for finding Convex Hull ?
6. Describe the method for finding Convex Hull using Graham's scan and using Jarvis's march.
7. Explain Graham Scan's method ?

CHAPTER

38

NP-Completeness

The field of complexity theory deals with how fast can one solve a certain type of problem. Or more generally how much resource does it take : time, memory-space, number of processors etc. The most common resource is time : number of steps. This is generally computed in terms of n, the length of the input string. We will use an informal model of a computer and an algorithm. All the definitions can be made precise by using a model of a computer such as a **Turing machine**.

While we are interested in the difficulty of a computation, we will focus our hardness results on the difficulty of yes/no questions. These results immediately generalize to questions about general computations. It is also possible to state definitions in terms of languages, where a language is defined as a set of strings : the language associated with a question is the set of all strings representing questions for which the answer is Yes.

38.1 Classes of Problems

We can categorize the problems into the following broad classes :

1. Problems which cannot even be defined formally.
2. Problems which can be formally defined but cannot be solved by computational means.
3. Problems which, though theoretically can be solved by computational means, yet are infeasible, *i.e.*, these problems require so large amount of computational resources that practically is not feasible to solve these problems by computational means. These problems are called **intractable** or **infeasible** problems.

4. Problems that are called feasible or theoretically not difficult to solve by computational means. The distinguishing feature of the problems is that for each instance of any of these problems, there exists a **Deterministic Turing Machine** that solves the problem having time-complexity as a polynomial function of the size of the problem. The class of problem is denoted by *P*.
5. Last, but probably most interesting class includes large number of problems for each of which it is not known whether it is in *P* or not in *P*.

These problems fall somewhere between class (3) and class (4) given above. However, for each of the problems in the class, it is known that it is in *NP*. *i.e.*, each can be solved by at least one-Non-Deterministic Turing Machine, the time complexity of which is a polynomial function of the size of the problem. Now, we can go still further and categorize the problems as :

Decision Problems

Decision problems are the computational problems for which the intended output is either "yes" or "no". In other words, a decision problem is a problem with yes/no answers. Hence in a decision problem, we can equivalently talk of the language associated with the decision problem, namely, the set of inputs for which the answer is yes.

Typically, we assume that the input is coded in binary, so the set of all possible inputs is $\{0,1\}^*$ and the language associated with a decision problem Q is

$$L(Q) = \{x \in \{0,1\}^* \mid \text{the answer is yes for problem } Q \text{ on input } x\}$$

38.1.1 Optimization Problems vs. Decision Problems

Decision Problems have Yes/No answers. *Optimization Problems* require answers that are optimal configurations. Decision problems are "easier" than optimization problems; if we can show that a decision problem is hard that will imply that its corresponding optimization problem is also hard.

38.1.2 The Classes *P* and *NP*

We define *P* (Polynomial) as the class of decision problems that are solvable by algorithms that run in time polynomial in the length of the input. That is, a decision question is in *P* if there exists an exponent k and an algorithm for the question that runs in time $O(n^k)$ where n is the length of the input.

P roughly captures the class of practically solvable problems. Or at least that is the conventional wisdom. Something that runs in time 2^n requires double the time if one adds one character to the input. Something that runs in polynomial time does not endure from this problem.

Class *P* : The set of all polynomially solvable problems.

- *P* is *closed* under addition, multiplication and composition.
- *P* is *independent* of particular formal models of computation or implementations.
- Any problem not in *P* is *hard*.
- A problem in *P* does not necessarily have an efficient algorithm.

Examples :

(*a*) **True Boolean Formulas.** A Boolean formula consists of variables and negated variables (known collectively as literals), and the operations "and" and "or". We use $\vee$ for "or", $\wedge$ for "and", and $\bar{x}$ for "not x". For example : $x \wedge (x \vee y \vee \bar{z}) \wedge (\bar{x} \vee \bar{y})$

An assignment is a setting of each of the variables to either true or false. For example if x and y are true and z is false, then the above boolean formula is false.

```
TRUEBF
Input : ϕ a boolean formula, asignment Ψ
Question : Does ψ make ϕ TRUE ?
```

Given a boolean function and an assignment of the variables, we can determine whether the formula is true or not in linear time. (A formula can be evaluated like a normal arithmetic expression using a single stack.) So it follows that the TRUEBF problem is in *P*.

(*b*) **Paths.** Recall that a path is a sequence of edges leading from one node to another.

```
PATH
INPUT : graph G, nodes a and b
Question : Is there a path from a to b in G ?
```

This problem is in *P*. To see if there is a path from node a to node *b*, one might determine all the nodes reachable from a by doing for instance a breadth-first search or Dijkstra's algorithm. Note that the actual running time depends on the representation of the graph.

We will next define the class NP, or Nondeterministic Polynomial Time. Before we can define this class, we need some definitions.

38.1.3 Verification Algorithm.

A verification algorithm is an algorithm *A*, that takes two inputs : an ordinary input x (coded in binary), and a **certificate** y, and outputs a 1 on certain combinations of x and y.

Verification algorithm A verifies an input string x if there exists a certificate y such that

$$A(x,y) = 1$$

The language verified by verification algorithm *A* is

$$L = \{\text{input strings } x \mid \text{there exists certificate string } y \text{ such that } A(x,y) = 1\}$$

Note that for an x that is not in *L*, for every certificate y, $A(x,y) \neq 1$

38.1.4 Polynomial-time Verification Algorithm

A verification algorithm *A* for a language *L* is a **polynomial-time verification algorithm** for *L* if

- For each $x \in L$, there is a certificate y of size polynomial in the size of x such that $A(x,y) = 1$, and $A(x,y)$ returns 1 in time polynomial in x.
- Since *A* is a verification algorithm for *L*, for every x not in *L* there is no certificate y for which $A(x,y) = 1$

38.2 The Class NP (Non Deterministic Polynomial)

The set of all problems that can be solved if we always guess correctly what computation path we should follow. Roughly speaking, it includes problems with exponential algorithms but

has not proved that they cannot have polynomial time algorithms. A language $L \in NP$ if and only if there exists a polynomial-time verification algorithm A for L.

Note that $P \subseteq NP$. At this time we do not know if $P = NP$ or $P \subset NP$.

Examples of Languages in NP

Let us consider the following examples of decision problems

- HAM-CYCLE = $\{\langle G \rangle \mid G$ is a Hamiltonian graph$\}$
- CIRCUIT-SAT = $\{\langle C \rangle \mid C$ is a satisfiable boolean circuit$\}$
- SAT = $\{\langle \phi \rangle \mid \phi$ is satisfiable bollean formula$\}$
- CNF – SAT = $\{\langle \phi \rangle\ \phi$ is a satisfiable bollean formula in CNF$\}$
- 3-CNF-SAT = $\{\langle \phi \rangle \mid \phi$ is a satisfiable boolean formula in CNF$\}$
- CLIQUE = $\{\langle G, k \rangle \mid G$ is an undirected graph with a clique of size $k\}$
- IS = $\{\langle G, k \rangle \mid G$ is an undirected graph with an independent set of size $k\}$
- VERTEX-COVER = $\{\langle G, k \rangle \mid$ undirected graph G has a vertex cover of size $k\}$
- TSP = $\{\langle G, c, k \rangle \mid G = (V, E)$ is a complete graph, $c : V \times V \to Z$ is a cost function, $k \in Z$ and G has a traveling salesman tour with cost at most $k\}$
- SUBSET-SUM = $\{\langle S, t \rangle \mid$ there is a subset $S' \subseteq S$ such that $t = \sum_{s \in S'} s\}$

It is quite straightforward to come up with a polynomial-time verification algorithm for each of these problems, *i.e.*, it is quite easy to show that each of these problems is in NP.

38.3 *NP*-Completeness

38.3.1 What is Reduction ?

Let L_1 and L_2 be two decision problems. Suppose algorithm A_2 solves L_2. That is, if y is an input for L_2 then algorithm A_2 will answer Yes or No depending upon whether $y \in L_2$ or not.

The idea is to find a transformation f from L_1 to L_2 so that the algorithm A_2 can be part of an algorithm A_1 to solve L_1.

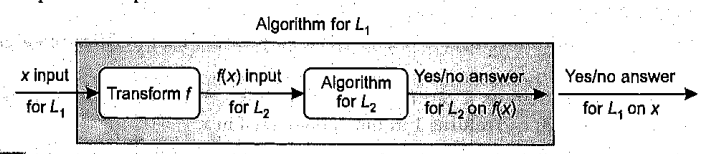


Figure 38.1

38.3.2 Polynomial-time Reductions

Let L_1 and L_2 be languages that are subsets of $\{0,1\}^*$.

We say that L_1 is polynomial-time reducible to L_2 if there exists a function f

$$f : \{0,1\}^* \to \{0,1\}^*$$

with the following properties :

- f transforms an input x for L_1 into an input $f(x)$ for L_2 such that $f(x)$ is a yes-input for L_2 if and only if x is a yes input for L_1. We require a yes-input of L_1 maps to a yes-input of L_2, and a no-input of L_1 maps to a no-input of L_2.

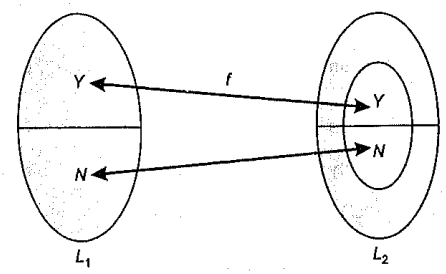


Figure 38.2

- $f(x)$ is computable in polynomial time (in size(x)). If such an f exists, we say that L_1 is **polynomial-time reducible** to L_2, and write $L_1 \le_p L_2$.

Example. *What can we do with a polynomial time reduction $f : L_1 \to L_2$.*

Solution. Given an algorithm A_2 for the decision problem L_2, we can develop an algorithm A_1 to solve L_1.

If A_2 is a polynomial time algorithm for L_2 and $L_1 \le_p L_2$ then we can construct a polynomial time algorithm for L_1.

Theorem

If $L_1 \le_p L_2$ and $L_2 \in P$, then $L_1 \in P$.

Proof. $L_2 \in P$ means that we have a polynomial time algorithm A_2 for L_2. Since $L_1 \le_p L_2$, we have a polynomial-time transformation f mapping input x for L_1 to an input for L_2. Combining these, we get the following polynomial-time algorithm for solving L_1 :

1. take input x for L_1 and compute $f(x)$;
2. run A_2 on input $f(x)$ and return the answer found (for L_2 on $f(x)$) as the answer for L_1 on x.

Each of Steps (1) and (2) takes polynomial time. So the combined algorithm takes polynomial time. Hence $L_1 \in P$.

Note *This does not imply that if $L_1 \le_p L_2$ and $L_1 \in P$, then $L_2 \in P$. This statement is not true.*

38.3.3 NP-Completeness

Polynomial-time reductions provide a formal means for showing that one problem is at least as hard as another, to within polynomial-time factor. That is, if $L_1 \le_p L_2$, then L_1 is not more than a polynomial factor harder than L_2, which is why the "less than or equal to " notation for reduction is mnemonic. We can now define the set of NP-Complete languages, which are the hardest problems in NP.

A language $L \subseteq \{0,1\}^*$ is **NP-complete** if it satisfies the following *two* properties :

1. $L \in NP$; and
2. For every $L' \in NP$, $L' \le_p L$

If a language L satisfies property 2, but not necessarily property 1, we say that L is **NP-hard.**

We use the notation $L \in NPC$ to denote that L is NP-complete.

Theorem

If any NP-complete problem is polynomial time solvable, then P = NP. If any problem in NP is not polynomial time solvable, then all NP complete problems are not polynomial time solvable.

Proof. Suppose that $L \in P$ and also that $L \in NPC$. For any $L' \in NP$, we have $L' \le_p L$ by property 2 of the definition of NP-completeness. We know if $L' \le_p L$ then $L \in P$ implies $L' \in P$, which proves the first statement.

To prove the second statement, suppose that there exists an $L \in NP$ such that $L \notin P$. Let $L' \in NPC$ be any NP-complete language, and for the purpose of contradiction, assume that $L' \in P$. But then we have $L \le_p L'$ and thus $L \in P$.

38.3.4 $P = NP$?

One of the most important problems in computer science is whether $P = NP$ or $P \ne NP$? Observe that $P \subseteq NP$. Given a problem $A \in P$, and a certificate, to verify the validity of a yes-input (an instance of A), we can simply solve A in polynomial time (since $A \in P$). It implies $A \in NP$.

Intuitively, $NP \subseteq P$ is doubtful. After all, just able to verify a certificate (corresponds to a yes-input) in polynomial time does not necessary mean we can able to tell whether an input is an yes-input of no-input in polynomial time.

However, 30 years after the $P = NP$? problem was first proposed, we are still no closer to solving it and do not know the answer. The search for a solution, though, has provided us with deep insights into what distinguishes an "easy" problem from a "hard" one.

38.4 The Class co-NP

Note that if $L \in NP$, there is no guarantee that $\overline{L} \in NP$ (since having certificates for yes-inputs, does not mean that we have certificates for the no-inputs).

The class of decision problems L such that $\overline{L} \in NP$ is called co-NP (observe it is does not require $L \in NP$).

Example. COMPOSITE $\in NP$ and so PRIMES = $\overline{\text{COMPOSITE}} \in$ co-NP :

Remark. in contrast, to the fact that $L \in NP$ does not necessarily imply $\overline{L} \in NP$ we do have that

$$L \in P, \text{ if and only if } \overline{L} \in P$$

This is because a polynomial time algorithm for L is also a polynomial time algorithm for (the NO-answers for L become Yes-answers for $\overline{L}$ and vice-versa).

38.5 Satisfiability (SAT)

A given Boolean formula is **satisfiable** if there is a way to assign truth values (0 or 1) to the variables such that the final result is 1.

Example. $f(x,y,z) = (x \wedge (y \vee \overline{z})) \vee (\overline{y} \wedge z \wedge \overline{x})$

x	y	z	$(x \wedge (y \vee \overline{z}))$	$(\overline{y} \wedge z \wedge \overline{x})$	$f(x,y,z)$
0	0	0	0	0	0
0	0	1	0	1	1
0	1	0	0	0	0
0	1	1	0	0	0
1	0	0	1	0	1
1	0	1	0	0	0
1	1	0	1	0	1
1	1	1	1	0	1

The assignment $x = 1$, $y = 1$, $z = 0$ makes $f(x,y,z)$ true, and hence it is satisfiable.

SAT problem. Determine whether an input Boolean formula is satisfiable. If a Boolean formula is satisfiable, it is a yes-input ; otherwise, it is a no-input.

Lemma

SAT $\in NP$.

Proof. The certificate consists of a particular 0 or 1 assignment to the variables. Given this assignment, we can evaluate the formula of length n (counting variables, operations, and parentheses), it requires at most n evaluations, each taking constant time. Hence, to check a certificate takes time $O(n)$. So we have SAT $\in$ NP.

38.6 Circuit Satisfiability

A combinational circuit is called **circuit satisfiable** (CIRCUIT-SAT) if for set of inputs applied, output of this circuit should always be one.

The circuit-satisfiability problem is, **"Given a Boolean combinational circuit composed of AND, OR, and NOT gates, is it satisfiable ?"** We can give a formal language for circuit satisfiability as

$$\text{CIRCUIT-SAT} = \{\langle C \rangle : C \text{ is a satisfiable Boolean combinational circuit}\}$$

Given a circuit C, we might attempt to determine whether it is satisfiable by simple checking all possible assignments to the inputs. If there are k inputs, there are 2^k possible assignments. When the size of C is polynomial in k, each one leads to a super polynomial time algorithm. In fact there is strong indication that no polynomial-time algorithm exists that solves the *circuit-satisfiability problem is NP-Complete (Cook-Levin Theorem)*. We break the proof of this into two parts. In the first part we prove this problem belongs to NP class and in second part we show that this problem is NP-hard.

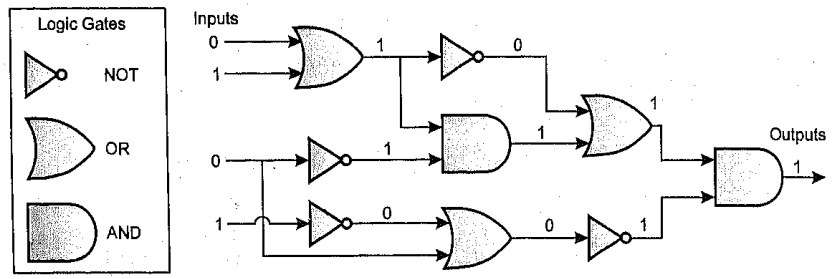


Figure 38.3 An example Boolean circuit.

Lemma

CIRCUIT-SAT is in NP.

Proof. We first use the choose method to "guess" the values of the input nodes as well as the output value of each logic gate. Then, we simply visit each logic gate g in C, *i.e.* each vertex with at least one incoming edge. We then check that the "guessed" value for the output of g is in fact the correct vale for g's Boolean function based on the given values for the inputs of g. This evaluation process can easily be performed in polynomial time. If any check for a gate fails, or if the "guessed" value for the output is 0, then we output "no". If, on the other, the check for every gate succeeds and the output is "1", the algorithm outputs "yes". Thus, if there is indeed a satisfying assignment of input values for C, then there is a possible collection of outcomes to the choose statements so that the algorithm will output "yes" in polynomial time. Likewise, if there is a collection of outcomes to the choose statements so that the algorithm outputs "yes" in polynomial time algorithm, there must be a satisfying assignment of input values for C. Therefore CIRCUIT-SAT is in NP.

Lemma

CIRCUIT-SAT is NP-hard.

Proof. We must show that every language in NP is polynomial-time reducible to CIRCUIT-SAT. Let L be any language in NP. We shall describe a polynomial time algorithm D computing a reduction function f that maps every binary string x to a circuit $C = f(x)$ such that $x \in L$ if and only if $C \in$ CIRCUIT-SAT.

Recall that any deterministic algorithm, such as D, can be implemented on a simple computational model (called RAM) that consists of a CPU and a bank M of addressable memory cells. In our case, the memory M contains the input x, the certificate y, the working storage W, that D needs to perform its computations, and the code for algorithm D itself. The working storage W for D includes all the registers used for temporary calculations and the stack frames for the procedures that D calls during its execution. The topmost such stack frame in W contains the program counter (PC) that identifies where D currently is in its program execution. Thus, there are no memory cells in the CPU itself. In performing each step of D, the CPU reads the next instruction i and performs the calculation indicated by i, and then updates the PC to point to the next instruction to be performed. Thus, the current state of D is completely characterized by the contents of its memory cells. Since D accepts an x in L in a polynomial $p(n)$ number of steps, where n is the size of x, then the entire effective collection of its memory cells can be assumed to consist of just $p(n)$ bits. For in $p(n)$ steps, D can access at most $p(n)$ memory cells. We refer to the $p(n)$ sized collection M of memory cells for an execution of D as the configuration of the algorithm D.

The heart of the reduction of L to CIRCUIT-SAT depends on our constructing a Boolean circuit that simulates the workings of the CPU in our computational model. CPU can be designed as a Boolean circuit consisting of AND, OR, and NOT gates. This circuit can be designed so as to take a configuration of D as input and provide as output the configuration resulting from processing the next computational step. In addition, this simulation circuit, which we will call S, can be constructed so as to consist of at most $cp(n)^2$ AND,OR, and NOT gates, for some constant $c > 0$.

To then simulate the entire $p(n)$ steps of D, we make $p(n)$ copies of S, with the output from one copy serving as input for the next. Part of the input for the first copy of S consists of "hard wired" values for the program for D, the value of x, the initial stack frame and the remaining working storage (initialized to all 0's). The only unspecified true inputs to the first copy of S are cells of D's configuration for the certificate y. These are the true inputs to our circuit. Similarly, we ignore all the outputs from the final copy of S, except the single output that indicates the answer from D, with "1" for "yes" and "0" for "no". The total size of the circuit C is $O(p(n)3)$, which is still polynomial in the size of x.

Consider an input x that D accepts for some certificate y after $p(n)$ steps. Then there is an assignment of values to the input to C corresponding to y, such that, by having C simulate D on this input and the hard-wired values for x, we will ultimately have C output a "1". Thus, C is satisfiable in this case. Conversely, consider a case when C is satisfiable. Then there is a set of inputs, which correspond to the certificate y, such that C outputs a "1". But, since C exactly simulates the algorithm D, this implies that there is an assignment of values to the certificate y, such that D outputs "yes". Thus, D will verify x in this case. Therefore, D accepts x with certificate y if and only if C is satisfiable.

38.7 Proving NP-Completeness

To prove that a problem P is NP-complete, we have following methods:

Method 1 (direct proof) :

(*a*) P is in NP.

(*b*) All problems in **NP-Complete** can be reduced to P.

Example : Conjunctive Normal Form (CNF) Satisfiability problem

Method 2 (equally general but potentially easier) :

(*a*) P is in NP.

(*b*) Find a problem P' that has already been proven to be in NP-Complete.

(*c*) Show that $P' \leq_p P$

Example. The k-clique problem.

(*a*) K-clique is in NP.

(*b*) CNF- Satisfiability is in NP-Complete.

(*c*) CNF- Satisfiability $\leq_p$ k-clique.

Method 3 (restriction; simple but not always available to all problems)

(*a*) P is in NP.

(*b*) Find a special case of P is in NP-Complete.

Example. Sub graph isomorphism : Given two graphs $G(V_1, E_1)$ and $H(V_2, E_2)$, does G contains a sub graph isomorphic to H? That is, can we find a subset V of V_1 and E of E_1 such that $|V| = |V_2|$ and $|E| = |E_2|$ and there exists a one to one function $f : V_2 \to V$ such that $\{u, v\}$ in E_2 if and only if $\{f(u), f(v)\}$ is in E?

(*a*) Sub graph isomorphism is in NP.

(*b*) k-clique is a special case of sub graph isomorphism when H is a clique with k nodes.

Thus, a formal method for proving that a language L is NP-Complete is :

1. Prove $L \in$ NP.
2. Select a known NP-Complete language L'.
3. Describe an algorithm that computes a function f mapping every instance of L' to an instance of L.
4. Prove that the function f satisfies $x \in L'$ if and only if $f(x) \in L$.
5. Prove that the algorithm computing f runs in polynomial time.

38.8 Techniques for NP-complete Problem

Techniques for dealing with NP-complete problems in practice are :

1. Backtracking 2. Branch and Bound 3. Approximation Algorithms

Examples of **NP-complete** problems :

1. Circuit satisfiability
2. Formula satisfiability
3. 3-CNF satisfiability
4. Clique
5. Vertex cover
6. Subset-sum
7. Hamiltonian cycle
8. Traveling salesman
9. Sub graph isomorphism
10. Integer programming
11. Set-partition
12. Graph coloring

A problem is NP-complete if it is both NP-hard and an element of NP. NP-complete problems are the hardest problems in NP. If anyone finds a polynomial-time algorithm for even one NP-complete problem, then that would imply a polynomial-time algorithm for every NP-complete problem. Literally thousands of problems have been shown to be NP-complete, so a polynomial-time algorithm for one (*i.e.*, all) of them seems incredibly unlikely.

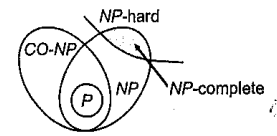


Figure 38.4

To prove that a problem is NP-hard, we use a reduction argument, exactly like we're trying to prove a lower bound.

To prove that problem A is NP-hard, reduce a known NP-hard problem to A.

38.9 (Formula) Satisfiability

An instance of the formula satisfiability problem (SAT) is a Boolean formula ϕ composed of :

1. Boolean variables variables : $x_1, x_2, \ldots$
2. Boolean connectives connectives : any Boolean function with one or two inputs and one : output (such as $\wedge$ (AND), $\vee$ (OR), $\neg$ (NOT), $\to$ (IMPLICATION), $\leftrightarrow$ (if and only if)).
3. Parenthesis

The satisfiability problem asks whether a given Boolean formula is satisfiable.

SAT = {(ϕ) : ϕ is a satisfiable boolean formula}

SAT is NP-complete

SAT $\in$ NP

SAT is **NP-Hard**

SAT belongs to NP, since a certificate consisting of a satisfying assignment for an input formula ϕ can be easily verified in polynomial time. Replace each variable in the formula with its value, and then evaluate the expression.

To prove that SAT is NP-Hard, we show that CIRCUIT-SAT $\leq_p$ SAT

Circuit satisfiability can be reduced in polynomial time to formula satisfiability :

The formula produced by the reduction algorithm has one variable for each wire in the circuit.

Polynomial reduction : for each gate, generate a formula expressing the proper operation of the gate in terms of a formula involving its incident wires ...

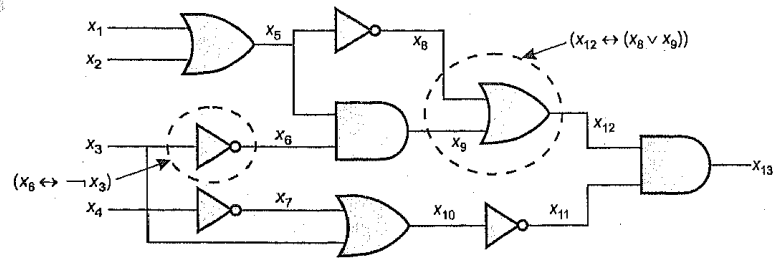


Figure 38.5

The formula produced by the algorithm is the AND of the circuit output with the conjuction of clauses describing the operation of each gate.

By construction, the Boolean combinational circuit is satisfiable if and only if formula ϕ is satisfiable.

$$\phi = x_{13} \wedge (x_5 \leftrightarrow (x_1 \vee x_2)) \wedge (x_6 \leftrightarrow \neg x_3) \wedge (x_7 \leftrightarrow \neg x_4) \wedge (x_8 \leftrightarrow \neg x_5) \wedge (x_9 \leftrightarrow (x_6 \wedge x_5))$$
$$\wedge (x_{10} \leftrightarrow (x_7 \vee x_3)) \wedge (x_{12} \leftrightarrow (x_8 \vee x_9)) \wedge (x_{11} \leftrightarrow \neg x_{10}) \wedge (x_{13} \leftrightarrow (x_{12} \wedge x_{11}))$$

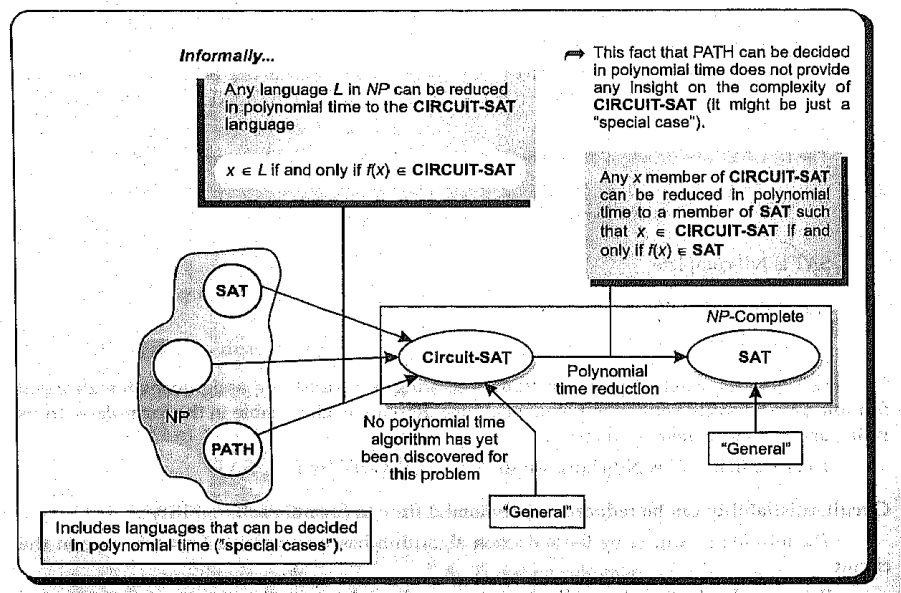


Figure 38.6

38.10 3-CNF Satisfiability

A literal in a Boolean formula is an occurrence of a variable or its negation. A Boolean formula is in **conjunctive normal form** (or CNF) if it is expressed as an AND of clauses, each of which is the OR of one or more literals.

A Boolean formula is in 3-conjunctive normal form (or 3-CNF) if each clause has ecaxtly three distinct literals

The 3-CNF-SAT problem asks whether a given Boolean formula ϕ in 3-CNF is satisfiable.

$$\text{3-CNF-SAT} = \{\langle\phi\rangle : \phi \text{ is a satisfiable 3-CNF Boolean formula}\}$$

3-CNF-SAT is NP-complete :

- 3-CNF-SAT $\in$ NP.
- 3-CNF-SAT is **NP-Hard** that is $L' \leq_p$ 3-CNF-SAT for every $L' \in$ NP

3-CNF-SAT belongs to NP, since a certificate consisting of a satisfying assignment for an input formula ϕ can be easily verified in polynomial time. Replace each variable in the formula with its value, and then evaluate the expression.

To prove that 3-CNF-SAT is NP-Hard, we show that SAT $\leq_p$ 3-CNF-SAT

Formula satisfiability can be reduced in polynomial time to 3-CNF satisfiability :

- Construct a binary parse tree for the input formula, with literals as leaves and connectives as internal nodes (expression tree) (Associatively can be used to parenthesize the expression fully so that every internal node in the resulting tree has 1 or 2 children).

$$\phi = ((x_1 \rightarrow x_2) \vee \neg((\neg x_1 \leftrightarrow x_3) \vee x_4)) \wedge \neg x_2$$

- Introduce a variable y_i variable for the output of each internal node (temporary)

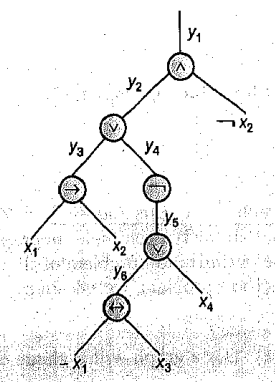


Figure 38.7

We must show that this transformation of ϕ into G is a reduction. First, suppose that ϕ has a satisfying assignment. Then, each clause C_r contains at least one literal l_i^r that is assigned 1. Picking one such literal from each clause yields a set of V' of k vertices. We claim that V' is a clique. For any two vertices $v_i^r, v_j^s \in V'$ where $r \neq s$, both corresponding literals l_i^r and l_j^s are mapped to 1 by the given satisfying assignment and thus the literals cannot be complements. Thus, by the construction of G, the edge (v_i^r, v_j^s) belongs to E.

Conversely, suppose that G has a clique V' of size k. No edges in G connect vertices in the same triple and so v' contains exactly one vertex per triple.

We can assign 1 to each literal l_i^r such that $v_i^r \in V'$ without fear of assigning 1 to both a literal and its complement, since G contains no edges between inconsistent literals. Now, each clause is satisfied and so ϕ is satisfied.

38.12 The Vertex-cover Problem

A vertex-cover of an undirected graph $G=(V, E)$ is a subset $V' \subseteq V$ such that if $(u, v) \in E$ then $u \in V'$ or $v \in V'$ (or both). Vertex-cover for G is a set of vertices that covers all the edges in E. The size of a vertex-cover is the number of vertices in it. In vertex-cover problem we find a vertex-cover of minimum size in a given graph. As a decision problem, we wish to determine whether a graph has a vertex cover of a given size k.

$$\text{VERTEX-COVER} = \{\langle G, k\rangle : \text{graph } G \text{ has vertex cover of size } k\}$$

To show that VERTEX-COVER is NP-complete, we first show that VERTEX-COVER $\in$ NP and then we prove that the vertex-cover problem is NP-hard by showing that CLIQUE $\leq_p$ VERTEX-COVER.

38.13 The Subset-Sum Problem

In the subset-sum problem, we are given a finite set $S \subset N$ and a target $t \in N$. We ask whether there is a subset $S' \subseteq S$ whose elements sum to t.

We define the problem as a language

$$\text{SUBSET-SUM} = \{\langle S, t\rangle : \text{there exists a subset } S' \subseteq S \text{ such that } t = \sum_{s \in S'} s\}$$

38.14 The Hamiltonian-Cycle Problem

A hamiltonian cycle of an undirected graph $G=(V, E)$ is a simple [illegible] ontai[illegible] vertex in V. A graph that contains a hamiltonian cycle is said to be ham[illegible] herw[illegible] non-hamiltonian. We can define the hamiltonian-cycle problem as "D[illegible] h G[illegible] hamiltonian cycle ?" in formal language.

$$\text{HAM-CYCLE} = \{\langle G\rangle : G \text{ is a hamiltonian graph}\}$$

38.15 The traveling-Salesman Problem

In the traveling salesman problem, a salesman must visit n cities. We can say that salesman wishes to make a tour or hamiltonian cycle, visiting each city exactly once and finishing at the city he starts from. There is an integer cost $C(i, j)$ to travel from city i to city j and the salesman wishes to make the tour whose total cost is minimum, where the total cost is the sum of individual costs along the edges of the tour.

In formal language,

$$\text{TSP} = \{\langle G, C, k\rangle : G=(V, E) \text{ is a complete graph } C \text{ is the function from } V \times V \to Z \\ k \in Z \text{ and } G \text{ has a traveling salesman tour with cost at most } k\}$$

To show that TSP is NP-complete, we first show that TSP $\in$ NP.

Given an instance of the problem, we use as a certificate the sequence of n vertices in the tour. The verification algorithm checks that this sequence contains each vertex exactly once, sums up the edge costs and checks whether the sum is at most k. This process can certainly be done in polynomial time.

To show NP-hard, we show that HAM-CYCLE $\leq_p$ TSP. Let $G=(V, E)$ be an instance of HAM-CYCLE. We construct an instance of TSP as follows. We form the complete graph $G' = (V, E')$ where $E' = \{(i, j) : i, j \in V\}$ and we define the cost function c by

$$c(i, j) = \begin{cases} 0 & \text{if } (i, j) \in E \\ 1 & \text{if } (i, j) \notin E \end{cases}$$

The instance of TSP is then $(G', C, 0)$ which is easily formed in polynomial time.

We now show that graph G has a hamiltonian cycle if and only if graph G' has a tour of cost at most 0. Suppose that graph G has a hamiltonian cycle h. Each edge in h belongs to E and thus has cost 0 in G'.

Thus h is a tour in G' with cost 0. Conversely, suppose that graph G' has a tour h' of cost at most 0. Since the costs of the edges in E' are 0 and 1 the cost of tour h' is exactly 0. Therefore, h' contains only edges in E. We conclude that h is a hamiltonian cycle in graph G.

Exercise

1. Give a formal definition for the problem of finding the longest simple cycle in an undirected graph. Give a related decision problem. Give the language corresponding to the decision problem.
2. Prove that if G is an undirected bipartite graph with an odd number of vertices, then G is non-hamiltonian.
3. Prove that the class NP of languages is closed under UNION, INTERSECTION, CONCATENATION, and KLEENE STAR. Discuss the closure of NP under complement.
4. Prove that if NP $\neq$ Co-NP, then $P \neq$ NP.

We must show that this transformation of ϕ into G is a reduction. First, suppose that ϕ has a satisfying assignment. Then, each clause C_r contains at least one literal l_i^r that is assigned 1. Picking one such literal from each clause yields a set of V' of k vertices. We claim that V' is a clique. For any two vertices $v_i^r, v_j^s \in V'$ where $r \neq s$, both corresponding literals l_i^r and l_j^s are mapped to 1 by the given satisfying assignment and thus the literals cannot be complements. Thus, by the construction of G, the edge (v_i^r, v_j^s) belongs to E.

Conversely, suppose that G has a clique V' of size k. No edges in G connect vertices in the same triple and so v' contains exactly one vertex per triple.

We can assign 1 to each literal l_i^r such that $v_i^r \in V'$ without fear of assigning 1 to both a literal and its complement, since G contains no edges between inconsistent literals. Now, each clause is satisfied and so ϕ is satisfied.

38.12 The Vertex-cover Problem

A vertex-cover of an undirected graph $G=(V, E)$ is a subset $V' \subseteq V$ such that if $(u, v) \in E$ then $u \in V'$ or $v \in V'$ (or both). Vertex-cover for G is a set of vertices that covers all the edges in E. The size of a vertex-cover is the number of vertices in it. In vertex-cover problem we find a vertex-cover of minimum size in a given graph. As a decision problem, we wish to determine whether a graph has a vertex cover of a given size k.

$$\text{VERTEX-COVER} = \{\langle G, k\rangle : \text{graph } G \text{ has vertex cover of size } k\}$$

To show that VERTEX-COVER is NP-complete, we first show that VERTEX-COVER $\in$ NP and then we prove that the vertex-cover problem is NP-hard by showing that CLIQUE $\leq_p$ VERTEX-COVER.

38.13 The Subset-Sum Problem

In the subset-sum problem, we are given a finite set $S \subset N$ and a target $t \in N$. We ask whether there is a subset $S' \subseteq S$ whose elements sum to t.

We define the problem as a language

$$\text{SUBSET-SUM} = \{\langle S, t\rangle : \text{there exists a subset } S' \subseteq S \text{ such that } t = \sum_{s \in S'} s\}$$

38.14 The Hamiltonian-Cycle Problem

A hamiltonian cycle of an undirected graph $G=(V, E)$ is a simple cycle that contains each vertex in V. A graph that contains a hamiltonian cycle is said to be hamiltonian otherwise, it is non-hamiltonian. We can define the hamiltonian-cycle problem as "Does a graph G have a hamiltonian cycle ?" in formal language.

$$\text{HAM-CYCLE} = \{\langle G\rangle : G \text{ is a hamiltonian graph}\}$$

38.15 The traveling-Salesman Problem

In the traveling salesman problem, a salesman must visit n cities. We can say that salesman wishes to make a tour or hamiltonian cycle, visiting each city exactly once and finishing at the city he starts from. There is an integer cost $C(i, j)$ to travel from city i to city j and the salesman wishes to make the tour whose total cost is minimum, where the total cost is the sum of individual costs along the edges of the tour.

In formal language,

$$\text{TSP} = \{\langle G, C, k\rangle : G=(V, E) \text{ is a complete graph } C \text{ is the function from } V \times V \to Z,\ k \in Z \text{ and } G \text{ has a traveling salesman tour with cost at most } k\}$$

To show that TSP is NP-complete, we first show that TSP $\in$ NP.

Given an instance of the problem, we use as a certificate the sequence of n vertices in the tour. The verification algorithm checks that this sequence contains each vertex exactly once, sums up the edge costs and checks whether the sum is at most k. This process can certainly be done in polynomial time.

To show NP-hard, we show that HAM-CYCLE $\leq_p$ TSP. Let $G=(V, E)$ be an instance of HAM-CYCLE. We construct an instance of TSP as follows. We form the complete graph $G' = (V, E')$ where $E' = \{(i, j) : i, j \in V\}$ and we define the cost function c by

$$c(i, j) = \begin{cases} 0 & \text{if } (i, j) \in E \\ 1 & \text{if } (i, j) \notin E \end{cases}$$

The instance of TSP is then $(G', C, 0)$ which is easily formed in polynomial time.

We now show that graph G has a hamiltonian cycle if and only if graph G' has a tour of cost at most 0. Suppose that graph G has a hamiltonian cycle h. Each edge in h belongs to E and thus has cost 0 in G'.

Thus h is a tour in G' with cost 0. Conversely, suppose that graph G' has a tour h' of cost at most 0. Since the costs of the edges in E' are 0 and 1 the cost of tour h' is exactly 0. Therefore, h' contains only edges in E. We conclude that h is a hamiltonian cycle in graph G.

Exercise

1. Give a formal definition for the problem of finding the longest simple cycle in an undirected graph. Give a related decision problem. Give the language corresponding to the decision problem.
2. Prove that if G is an undirected bipartite graph with an odd number of vertices, then G is non-hamiltonian.
3. Prove that the class NP of languages is closed under UNION, INTERSECTION, CONCATENATION, and KLEENE STAR. Discuss the closure of NP under complement.
4. Prove that if NP $\neq$ Co-NP, then $P \neq$ NP.

5. Prove that $P \subseteq$ Co-NP.
6. Show that the hamiltonian-path problem can be solved in polynomial time on directed acyclic graphs. Give an efficient algorithm for the problem.
7. Show that ciruit satisfiability problem is NP-complete.
8. Show that the $\leq_p$ relation is a transitive relation on languages. That is, show that if $L_1 \leq_p L_2$ and $L_2 \leq_p L_3$ then $L_1 \leq_p L_3$.
9. Show that the hamiltonian-path problem is NP-complete.
10. Write short notes on the following :
    (*i*) NP-hard problem
    (*ii*) NP-complete problem.
11. Prove that satisfiability of boolean formula in 3-conjunctive normal form (3-CNF) is NP-complete.
12. What do you mean by formula. Satisfiability. Check this for the formula $\phi = ((x_1 \rightarrow x_2)v)$

CHAPTER

39

Approximate Algorithms

39.1 Introduction

Many important problems are NP-Complete (NPC), which are likely to be quite hard to solve exactly. We cannot just forget those problems, as they are so important. There are several things we can do :

- Try *exponential time algorithm*. An optimal solution is found. Not feasible if problem size is large.
- Try *general optimization methods*, *e.g.*, branch and-bound, genetic algorithms, neural nets. Some is hard to show how good they are compared with the optimal solution.
- Try *approximate algorithms*. Generally fast, but may not get an optimal solution. But they can be proved to be close to the optimal solutions.

So, among many approaches, one approach to solving NP-Complete optimization problems is the use of fast (*i.e.* polynomial bounded) algorithms that are not guaranteed to give best solution but will give one that is close to the optimal. Such algorithms are called *approximation algorithms* or *heuristic algorithms*.

The use of heuristics in an existing algorithm may enable it to quickly solve a large instance of a problem provided the heuristic works on that instance. A heuristic, however, does not work

equally effectively on all problem instances. Exponential time algorithms, even coupled with heuristics, still show exponential behaviour on some set of inputs.

In approximation algorithm, we remove the requirement that the algorithm that solves the optimization problem P must always generate an optimal solution. This requirement is replaced by the requirement that the algorithm for P must always generate a feasible solution with value close to the value of an optimal solution. A feasible solution with value close to the value of an optimal solution is called an **approximate solution**. An approximation algorithm for P is an algorithm that generates approximate solutions for P.

In many applications an approximate solution is good enough, especially when the time required finding an optimal solution is considered. In the case of *NP*-hard problems, approximate solutions have added importance as exact solutions (*i.e.*, optimal solutions) may not be obtainable in a feasible amount of computing time. An approximate solution may be all one can get using a reasonable amount of computing time.

The goal of an approximation is to come as close to the optimum value as possible in reasonable amount of time *i.e.*

- For the traveling salesperson problem, the optimization problem is to find the shortest cycle, and the approximation problem is to find the short cycle.
- For the vertex cover problem, the optimization problem is to find the vertex cover with fewest vertices and the approximation problem is to find the vertex cover with few vertices.

39.2 Performance Ratios

Suppose we work on an optimization problem where each solution carries a cost. An approximate algorithm returns a legal solution, but the cost of that legal solution may not be optimal.

For example, suppose we are looking for a **minimum size vertex-cover** (VC). An approximate algorithm returns a VC for us, but the size (cost) may not be minimum.

Another example is we are looking for a **maximum size independent set** (IS). An approximate algorithm returns an IS for us, but the size (cost) may not be Maximum. Let C be the cost of the solution returned by an approximate algorithm, and C^* is the cost of the optimal solution.

We say the approximate algorithm has an *approximation ratio* $P(n)$ for an input size n, where

$$\max\left(\frac{C}{C^*}, \frac{C^*}{C}\right) \le P(n)$$

Intuitively, the *approximation ratio* measures how bad the approximate solution is compared with the optimal solution. A large (small) *approximation ratio* means the solution is much worse than (more or less the same as) an optimal solution.

Observe that $P(n)$ is always ≥ 1; if the ratio does not depend on n, we may just write P. Therefore, a 1-approximation algorithm gives an optimal solution. Some problems have polynomial-time approximation algorithms with small constant approximate ratios, while others have best-known polynomial-time approximation algorithms whose approximate ratios grow with n.

39.3 Examples of Approximate Algorithms

Some examples of approximate algorithms are as follows :

39.3.1 Vertex-cover

Vertex Cover. A vertex cover of a graph G is a set of vertices such that every edge in G is incident to at least one of these vertices.

The decision vertex-cover problem was proven NPC. Now, we want to solve the optimal version of vertex cover problem, *i.e.*, we want to find a minimum size vertex cover of a given graph. We call such vertex cover an optimal vertex cover C^*.

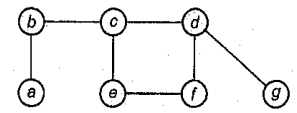


Figure 39.1

An approximate algorithm for Vertex Cover :

```
Approx-Vertex-Cover (G = (V, E) )
 {
     C = empty-set ;
     E' = E ;
     while E' is not empty do
     {
        let (u,v) be any edge in E' ; (*)
        add u and v to C ;
        remove from E' all edges incident to
        u or v ;
     }
     return C ;
 }
```

The idea is to take an edge (u, v) one by one, put both vertices to C, and remove all the edges incident to u or v. We carry on until all edges have been removed. Obviously, C is a *VC*. But how good is C?

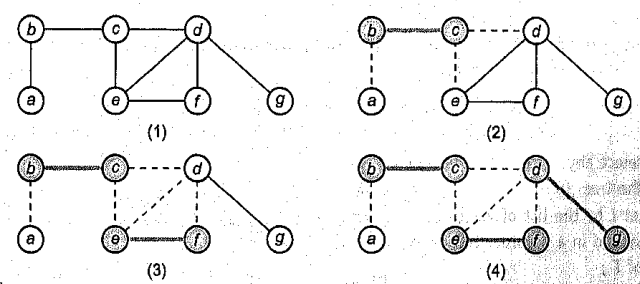


Figure 39.2

$VC = \{b, c, d, e, f, g\}$

Theorem

Approx-Vertex-Cover is a 2-approximation algorithm, i.e.,

$$\frac{|C|}{|C^*|} \le 2$$

It means the number of vertices in C returned by Approx-Vertex-Cover guarantees to be at most twice of the optimal value.

Proof.

1. Let A be the edge set selected by line (*). Observe that $|C| = 2|A|$.
2. Observe that the edge in A does not have any common vertex between them. It means for $e = (x, y) \in A$, either x or y must be selected to the optimal vertex cover C^*. It follows $|C^*| \ge |A|$.
3. Now we have

$$\frac{|C|}{2} = |A| \le |C^*| \quad \Rightarrow \quad \frac{|C|}{|C^*|} \le 2.$$

39.3.2 Traveling-salesman problem

Imagine we are a salesman, and we need to visit n cities. We want to start a *tour* at a city and visit every city *exactly one time*, and finish the tour at the city from where we start. There is a non-negative cost $c(i, j)$ to travel from city i to city j. The goal is to find a tour (which is a Hamiltonian cycle) of minimum cost. We assume every two cities are connected. Such problem is called *Traveling-salesman problem* (TSP).

We can model the cities as a complete graph of n vertices, where each vertex represents a city.

It can be shown that TSP is NPC.

If we assume the cost function c satisfies the *triangle inequality*, then we can use the following approximate algorithm.

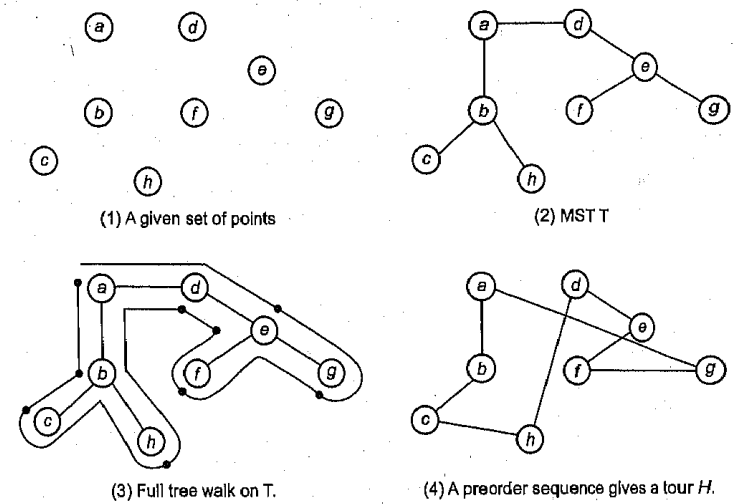
Triangle inequality. Let u, v, w be any three vertices, we have

$$c(u, w) \le c(u, v) + c(v, w)$$

One important observation to develop an approximate solution is if we remove an edge from H^*, the tour becomes a *spanning tree*.

```
Approx-TSP (G = (V, E)) {
1. compute a MST T of G ;
2. select any vertex r be the root of
   the tree ;
3. let L be the list of vertices
   visited in a preorder tree walk
   of T ;
4. return the hamiltonian cycle H that
   visits the vertices in the order L ;
}
```


Traveling-salesman Problem



(1) A given set of points

(2) MST T

(3) Full tree walk on T.

(4) A preorder sequence gives a tour H.

Figure 39.3

Intuitively, Approx-TSP first makes a full walk of MST T, which visits every edge exactly two times. To create a Hamiltonian cycle from the full walk, it bypasses some vertices (which corresponds to making a shortcut).

Theorem

Approx-TSP is a 2-approximation algorithm i.e.,

$$\frac{c(H)}{c(H^*)} \le 2$$

Proof.

1. Observe that if we remove any edge from H^*, then it becomes to a spanning tree, hence we have $c(T) \le c(H^*)$:
2. The cost of a full walk on T is $2c(T)$; since H makes a short-cut on the full walk, by *triangle inequality*, we have $c(H) \le 2c(T)$:
3. Combining, we have

$$\frac{c(H)}{2} \le c(T) \le c(H^*) \quad \Rightarrow \quad \frac{c(H)}{c(H^*)} \le 2$$

Exercise

1. Suppose that a complete undirected graph $G = (V, E)$ with at least 3 vertices has a cost function c that satisfies the triangle inequality. Prove that $c(u,v) \geq 0$ for all $u,v \in V$.
2. Show that the decision version of the set-covering problem is NP-complete by reduction from the vertex-cover problem.
3. Write the short note on approximation algorithm.
4. Explain approximation algorithm with an appropriate example.

CHAPTER

40

Randomized Algorithm

40.1 Introduction

Historically, the study of randomized algorithms was spurred by the discovery by **Miller** and **Robin** in 1976 that the problem of determining the primality of a number can be solved efficiently by a randomized algorithm. At that time, no practical deterministic algorithm for primality was known. Given an integer 'n', the problem of deciding whether 'n' is a prime is known as primality testing.

A randomized algorithm is defined as an algorithm that is allowed to access a source of independent, unbiased bits, and it is then allowed to use these random bits to influence its computation. An algorithm is randomized if its output is determined by the input as well as the values produced by a random-number generator.

A randomized algorithm is one that makes use of a randomizer such as a random number generator. Some of the decisions made in the algorithm depend on the output of the randomizer. Since the output of any randomizer might differ in an unpredictable way from run to run, the output of a randomized algorithm could also differ from run to run for the same input. So, the execution time of a randomized algorithm could also vary from run to run for the same input.

In common practice, this means that the machine implementing the algorithm has access to a pseudo-random number generator. The algorithm typically uses the random bits as an auxiliary input to guide its behaviour, in the hope of achieving good performance in the "average case". Formally, the algorithm's performance will be a random variable determined by the random bits, with (hopefully) good expected value ; this expected value is called the *expected runtime*. The "worst case" is typically so unlikely to occur that it can be ignored.

Randomized algorithms are particularly useful when faced with a malicious "adversary" or attacker who deliberately tries to feed a bad input to the algorithm. It is for this reason that randomness is ubiquitous in cryptography. In cryptographic applications, pseudo-random numbers cannot be used, since the adversary can predict them, making the algorithm effectively deterministic. Therefore either a source of truly random numbers or a cryptographically secure pseudo-rando number generator is required. Another area in which randomness is inherent is quantum computing.

Randomized algorithms can be categorized into *two* classes :

1. Las Vegas algorithms
2. Monte Carlo algorithms

Las Vegas algorithm always produces the same (correct) output for the same input. The execution time of a Las Vegas algorithm depends on the output of the randomizer. If we are lucky, the algorithm might terminate fast, and if not, it might run for a longer period of time. In general, its runtime for each input is a random variable whose expectation is bounded.

The second is algorithms whose output might differ from run to run for the same input. These are called **Monte Carlo algorithms.** Consider any problem for which there are only two possible answers, say yes and no. If a Monte Carlo algorithm is used to solve such a problem, then the algorithm might give incorrect answers depending on the output of the randomizer. We require that the probability of an incorrect answer from a Monte Carlo algorithm be low.

Typically, for a fixed input, a Monte Carlo algorithm does not display much variation in execution time between runs, whereas in the case of a Las Vegas algorithm this variation is significant.

40.2 Applications

40.2.1 Randomized Quick Sort Algorithm

- In randomized quick sort, we will pick randomly an element as the pivot for partitioning.
- The expected runtime of any input is $O(n \log n)$.

Analysis of Randomized Quick Sort

- Let $s(i)$ be the ith smallest in the input list S.
- X_{ij} is a random variable such that $X_{ij} = 1$ if $s(i)$ is compared with $s(j)$; $X_{ij} = 0$ otherwise.

- Expected runtime t of randomized QS is :

$$t = E\left[\sum_{i=1}^{n} \sum_{j \geq 1} X_{ij}\right] = \sum_{i=1}^{n} \sum_{j \geq 1} E[X_{ij}]$$

- $E[X_{ij}]$ is the expected value of X_{ij} over the set of all random choices of the pivots, which is equal to the probability p_{ij} that $s(i)$ will be compared with $s(j)$.

The randomized quick sort algorithm uses $i \leftarrow$ Random (p, r) to be the new pivot element.

Randomized-Partition (A, p, r)

1. $i \leftarrow$ Random (p, r)
2. exchange $A[r] \leftrightarrow A[i]$
3. return Partition (A, p, r)

Randomized-Quicksort (A, p, r)

1. If $p < r$
2. then $q \leftarrow$ Randomized-Partition (A, p, r)
3. Randomized-Quicksort $(A, p, q-1)$
4. Randomized-Quicksort $(A, q+1, r)$

40.2.2 Randomized Minimum-cut Algorithm

A more complex example, representative of the use of randomized algorithms to solve graph theoretic problems, is the following **randomized minimum cut algorithm** :

```
Find-min-cut(undirected graph G)
{
  while there are more than 2 nodes in G do
  {
      pick an edge (u,v) at random in G
      contract the edge, while preserving multi-edges
      remove all loops
  }
  output the remaining edges
}
```

Here, contracting an edge (u, v) means adding a new vertex w, replacing any edge (u, x) or (v, x) with (w, x), and then deleting u and v from G.

- Let k be the min-cut of the given $G(E, V)$ where $|V| = n$.
- Then $|E| \geq kn/2$.

- The probability q_1 of picking one of those k edges in the first merging step $\leq 2/n$.
- The probability p_1 of not picking any of those k edges in the first merging step $\geq (1-(2/n))$
- Repeat the same argument for the first $n-2$ merging steps.
- Probability p of not picking any of those k edges in all the merging steps $\geq (1-2/n)(1-2/(n-1)(1-2/(n-2))\ldots(1-2/3)$.
- Therefore, the probability of finding the min-cut :

$$p \geq \left(1-\frac{2}{n}\right)\left(1-\frac{2}{n-1}\right)\ldots\left(1-\frac{2}{3}\right)$$

$$= \frac{(n-2)(n-3)\ldots(1)}{n(n-1)\ldots 3}$$

$$= \frac{2}{n(n-1)}$$

- If we repeat the whole procedure $n^2/2$ times, the probability of not finding the min-cut is at most

$$(1-2/n^2)^{n^2/2} \cong 1/e$$

Randomized Min-cut is a **Monte Carlo Algorithm.**

40.2.3 There are some interesting complexity classes involving randomized algorithms

- Randomized Polynomial time (*RP*)
- Zero-error Probabilistic Polynomial time (*ZPP*)
- Probabilistic Polynomial time (*PP*)
- Bounded-error Probabilistic Polynomial time (*BPP*)

1. *RP Definition.* The class *RP* consists of all languages L that have a randomized algorithm A running in worst-case polynomial time such that for any input x in Σ^*.

$$x \in L \quad \Rightarrow \quad \Pr[A(x) \text{ accepts}] \geq \frac{1}{2} \qquad \text{Pr means probability.}$$

$$x \notin L \quad \Rightarrow \quad \Pr[A(x) \text{ accepts}] = 0$$

Independent repetitions of the algorithms can be used to reduce the probability of error to exponentially small.

Notice that the success probability can be changed to an inverse polynomial function of the input size without affecting the definition of *RP*.

2. *ZPP Definition.* The class ZPP is the class of languages, which have Las Vegas algorithms running in expected polynomial time.

$$\text{ZPP} = \text{RP} \cap \text{co-RP}$$

(Note that a language L is in co-X where X is a comlexity class if and only if it's complement $\Sigma^* - L$ is in X.)

3. PP Definition. The class *PP* consists of all languages L that have a randomized algorithm A running in worst-case polynomial time such that for any input x in Σ^*.

$$x \in L \quad \Rightarrow \quad \Pr[A(x) \text{ accepts}] \geq \frac{1}{2}$$

$$x \notin L \quad \Rightarrow \quad \Pr[A(x) \text{ accepts}] < \frac{1}{2}$$

To reduce the error probability, we can repeat the algorithm several times on the same input and produce the output which occurs in the majority of those trials.

However, the definition of *PP* is quite weak since we have no bound on how far from 1/2 the probabilities are. It may not be possible to use a small number (*e.g.*, polynomial number) of repetitions to obtain a significantly small error proability.

4. *BPP Definition.* The class BPP consists of all languages L that have a randomized algorithm A running in worst-case polynomial time such that for any input x in Σ^*.

$$x \in L \quad \Rightarrow \quad \Pr[A(x) \text{ accepts}] \geq \frac{3}{4}$$

$$x \notin L \quad \Rightarrow \quad \Pr[A(x) \text{ accepts}] \leq \frac{1}{4}$$

For this class of algorithms, the error probability can be reduced to $1/2n$ with only a polynomial number of iterations.

In fact, the probability bounds 3/4 and 1/4 can be changed to $\frac{1}{2}+\frac{1}{p(n)}$ and $\frac{1}{2}-\frac{1}{p(n)}$ respectively where $p(n)$ is a polynomial function of the input size n without affecting the definition of BPP.

40.3 Advantages and Disadvantages

Two of the most important advantages of using randomized algorithms are their **simplicity** and **efficiency**. A majority of the randomized algorithms found in the literature are simpler than the best deterministic algorithms for the same problems. Randomized algorithms have also been shown to yield better complexity bounds. Not only do randomized algorithms yield superior asymptotic run-time bounds, but they also have demonstrated to be competitive in practice.

The randomized algorithm performs badly only if the random-number generator produces an unlucky permutation to be sorted.

A randomized strategy is typically useful when there are many ways in which an algorithm can proceed but it is difficult to determine a way that guaranteed to be good. If many of the alternatives are good, simply choosing one randomly can yield a good strategy.

Often, an algorithm must make many choices during its execution. If the benefits of good choice outweigh the costs of bad choices, a random selection of good and bad choices can yield an efficient algorithm.

Main Disadvantages of Randomized Algorithm

1. Computer architecture tries to predict the future behaviour of algorithms to optimize its execution.
2. Randomization is useful only for very hard problems.

Randomized algorithms can be used to solve a wide variety of real-world problems like approximation algorithms, they can be used to more quickly solve tough *NP*-complete problem. An advantage over the approximation algorithms, however, is that a randomized algorithm will eventually yield an exact answer if executed enough times.

BIBLIOGRAPHY

Cormen, Thomas H. *"Introduction to algorithms"*, Cambridge, Mass : MIT Press ; New York : McGraw-Hill, c1990

Aho, Alfred V., *"Data structures and algorithms"*, Reading, Mass : Addison-Wesley, c1983.

Aho, Alfred V., *"The design and analysis of computer algorithm"*. edited by Attallah, Mikhail J. *"Algorithms and theory of computation handbook"*, CRC Press, 1998 (QA76.9.A43 1999).

Baase, Sara., *"Computer algorithms : introduction to design and analysis"*, Reading, Mass : Addison-Wesley Pub. Co., c1978.

Even, Shimon., *"Graph Algorithms"*,Computer Science Press, 1979.

Garey, Michael R.,*"Computers and intractability : a guide to the theory of NP"*, San Francisco : W. H. Freeman, c1979.

Gibbons, Alan M., *"Efficient Parallel Algorithms"*, Cambridge University Press, 1988.

Greene, Daniel H., *"Mathematics for the analysis of algorithms"*, Boston : Birkhauser, c1981.

Goodrich T. Michael, *"Algorithm Design - Foundations, Analysis & Internet Examples "*

Gonnet, Gaston H., *"Handbook of algorithms and data structures : in Pascal and C"*, Wokingham, England ; Reading, Mass : Addison-Wesley Pub. Co.,

Gonnet, Gaston H. and Baeza-Yates, Ricardo editors, *"Handbook of Algorithms and Data Structures in Pascal and C"*, Addison-Wesley, 1991.

Harel, David., *"Algorithmics : the spirit of computing"*, second edition Addison-Wesley, 1992.

Horowitz, Ellis., *"Fundamentals of computer algorithms"*, Potomac, Md : Computer Science Press, c1978.

Smith, Jeffrey D., *"Design and Analysis of Algorithms"*, PWS-Kent, 1989.

Kingston, Jeffrey Howard., *"Algorithms and data structures : design, correctness, analysis"*, Sydney : Addison-Wesley, 1990.

Koren, Israel., *"Computer Arithmetic Algorithms"*, Prentice-Hall, 1993.

Kozen, Dexter C. *"The design and analysis of algorithms"*, 1992.(QA76.9.A43 K69 1992).

Kreher, Donald L. and Stinson, Douglas Combinatorial Algorithms : Generation, Enumeration and Search, (CRC Press, 1998)

Kruse, Robert L., *"Data Structures and Program Design"*, second edition Prentice-Hall, 1996.