Navlakhi®

A Concise Book on Analysis of Algorithms

(ver. 2017v2)

By

Abhishek Navlakhi

This is a private release for students of Navlakhi's. More educational material can be found at **navlakhi.com and navlakhi.mobi**

Tel: 9820246760 / 9769479368 / 9820009639
Email: abhishek@navlakhi.com

String Matching Algorithms

- The naïve string matching Algorithms
- The Rabin Karp algorithm
- String matching with finite automata
- The knuth-Morris-Pratt algorithm
- Longest common subsequence **algorithm**

PATTERN MATCHING ALGORITHM

We are given a text string T of length n & a pattern string P of length m, and we want to find whether P is a substring of T.

- **Brute Force**

Algorithm BruteForceMatch(T, P):
Input: Strings T (text) with n characters and P (pattern) with m characters
Output: Starting index of the first substring of T matching P, or an indication that P is not a substring of T

```
for i ← 0 to n − m {for each candidate index in T} do
  j ← 0
  while (j < m and T[i + j] = P[j]) do
    j ← j + 1
  if j = m then
    return i
return "There is no substring of T matching P."
```

The outer loop runs for all the characters of text and the inner loop runs through the pattern array incrementing by 1 each time a match is obtained.

Performance

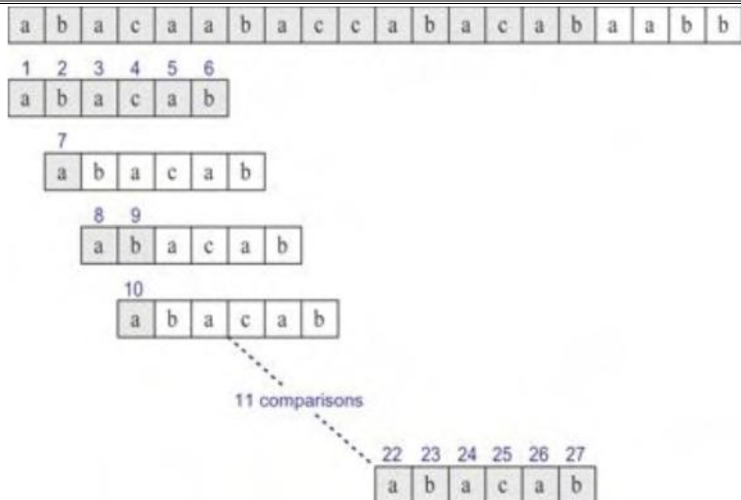
The outer **for** loop is executed at most n − m+ 1 times, and the inner loop is executed at most m times. Thus, the running time of the brute-force method is O((n − m+ 1)m), which is simplified as O(nm). Note that when m = n/2, this algorithm has quadratic running time $O(n^2)$.

Example

`T = "abacaabaccabacabaabb"`

and the pattern string

`P= "abacab".`



1.1 Rabin-Karp algorithm

Rabin-Karp string searching algorithm calculates a numerical (hash) value for the pattern p, and for each m-character substring of text t. Then it compares the numerical values instead of comparing the actual symbols. If any match is found, it compares the pattern with the substring by naive approach. Otherwise it shifts to next substring of t to compare with p.

We can compute the numerical (hash) values using Horner's rule.

Lets assume, $h_0 = k$

$$h_1 = d\left(k - p\left[1\right].d^{m-1}\right) + \ p\left[m+1\right]$$

Suppose, we have given a text $t = [3, 1, 4, 1, 5, 2]$ and m = 5, q = 13;

$t_0 = 31415$

So t_1 = 10(31415 - 10^{5-1}.t[1]) + t[5+1]

$= 10(31415 - 10^4.3) + 2$

$= 10(1415) + 2 = 14152$

Here p and substring t_i may be too large to work with conveniently. The simple solution is, we can compute p and the t_i modulo a suitable modulus q.

So for each i,

$h_{i+1} = \left(d\left(h_i - t\left[i+1\right].d^{m-1}\right) + \ t\left[m+i+1\right]\right)$ mod q

The modulus q is typically chosen as a prime such that d.q fits within one computer word.

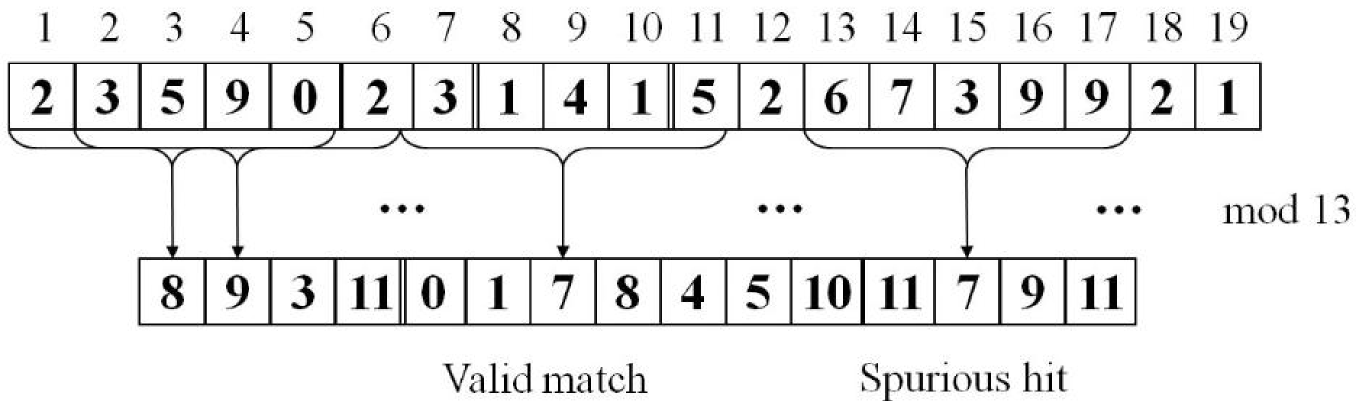
Algorithm

Compute h_p (for pattern p)
Compute h_t (for the first substring of t with m length)
For $i = 1$ to $n - m$
  If $h_p = h_t$
    Match $t[i \ldots . i+m]$ with p, if matched return 1
  Else
    $h_t = \left(d\left(h_t - t\left[i+1\right].d^{m-1}\right) + \ t\left[m+i+1\right]\right)$ mod q
End

Suppose, t= 2359023141526739921 and $p = 31415$,

Now, $h_p = 7$ (31415 = 7 (mod 13))

substring beginning at position 7 = valid match



This algorithm has a significant improvement in average-case running time over naive approach.

The Rabin-Karp algorithm has the complexity of O(nm) where **n**, of course, is the length of the text, while **m** is the length of the pattern

3 Reasons Why Rabin-Karp is Cool

1. Good for plagiarism, because it can deal with multiple pattern matching!
2. Not faster than brute force matching in theory, but in practice its complexity is O(n+m)!
3. With a good hashing function it can be quite effective and it's easy to implement!

2 Reasons Why Rabin-Karp is Not Cool

1. There are lots of string matching algorithms that are faster than O(n+m)
2. It's practically as slow as brute force matching and it requires additional space

- **Knuth – Morris – Pratt (KMP) Algorithm**

The failure function $f(j)$ is defined as the length of the longest prefix of P that is a suffix of $P[1..j]$

Consider the pattern string P = "abacab". The Knuth-Morris-Pratt (KMP) failure function f(j) for the string P is as shown in the following table:

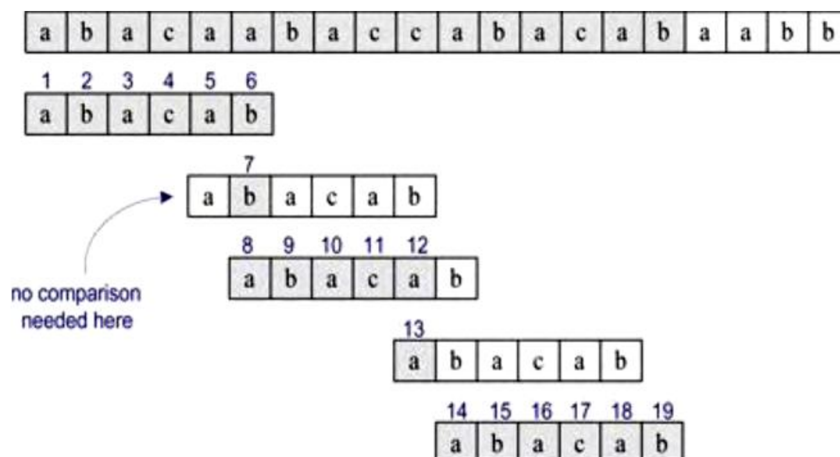
j	0	1	2	3	4	5
$P[j]$	a	b	a	c	a	b
$f(j)$	0	0	1	0	1	2

Algorithm KMPMatch(T, P):

Input: Strings T (text) with n characters and P (pattern) with m characters
Output: Starting index of the first substring of T matching P, or an indication that P is not a substring of T

```
f ← KMPFailureFunction(P)          {construct the failure function f for P}
i ← 0
j ← 0
while i < n do
   if P[j] = T[i] then
      if j = m − 1 then
         return i − m + 1          {a match!}
      i ← i + 1
      j ← j + 1
   else if j > 0 {no match, but we have advanced in P} then
      j ← f(j − 1)          {j indexes just after prefix of P that must match}
   else
      i ← i + 1
return "There is no substring of T matching P."
```

The KMP pattern matching algorithm, shown above, incrementally processes the text string T comparing it to the pattern string P. Each time there is a match, we increment the current indices. On the other hand, if there is a mismatch and we have previously made progress in P, then we consult the failure function to determine the new index in P where we need to continue checking P against T. Otherwise (there was a mismatch and we are at the beginning of P), we simply increment the index for T (and keep the index variable for P at its beginning). We repeat this process until we find a match of P in T or the index for T reaches n, the length of T (indicating that we did not find the pattern Pin$T$).



Navlakhi®

Algorithm KMPFailureFunction(P):

Input: String P (pattern) with m characters

Output: The failure function f for P, which maps j to the length of the longest prefix of P that is a suffix of $P[1..j]$

```
i ← 1
j ← 0
f(0) ← 0
while i < m do
    if P[j] = P[i] then
        {we have matched j + 1 characters}
        f(i) ← j + 1
        i ← i + 1
        j ← j + 1
    else if j > 0 then
        {j indexes just after a prefix of P that must match}
        j ← f(j − 1)
    else
        {we have no match here}
        f(i) ← 0
        i ← i + 1
```

Performance

For the sake of the analysis, let us define $k = i - j$. Intuitively, k is the total amount by which the pattern P has been shifted with respect to the text T. Note that throughout the execution of the algorithm, we have k n. One of the following three cases occurs at each iteration of the loop.

- If $T[i] = P[j]$, then i increases by 1, and k does not change, since j also increases by 1.
- If $T[i]$ $P[j]$ and $j > 0$, then i does not change and k increases by at least 1, since in this case k changes from $i - j$ to $i - f(j - 1)$, which is an addition of $j - f(j - 1)$, which is positive because $f(j - 1) < j$.
- If $T[i]$ $P[j]$ and $j = 0$, then i increases by 1 and k increases by 1, since j does not change.

Thus, at each iteration of the loop, either i or k increases by at least 1 (possibly both); hence, the total number of iterations of the **while** loop in the KMP pattern matching algorithm is at most $2n$.

Algorithm `KMPFailureFunction` runs in $O(m)$ time. Its analysis is analogous to that of algorithm `KMPMatch`. Thus, we have:

The Knuth-Morris-Pratt algorithm performs pattern matching on a text string of length n and a pattern string of length m in O(n + m) time.

TEXT SIMILARITY TESTING

A common text processing problem, which arises in genetics and software engineering, is to test the similarity between two text strings. In a genetics application, the two strings could correspond to two strands of DNA, which could, for example, come from two individuals, who we will consider genetically related if they have a long subsequence common to their respective DNA sequences.

The specific text similarity problem we address here is the longest common subsequence (LCS) problem. In this problem, we are given two character strings,

$X = X_0X_1X_2 \ldots X_{n-1}$ and $Y = Y_0Y_1Y_2 \ldots Y_{m-1}$,

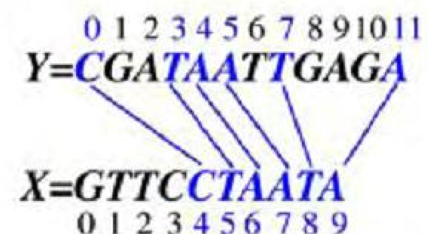
over some alphabet (such as the alphabet {A,C, G, T} common in computational genetics) and are asked to find a longest string S that is a subsequence of both X and Y.

Longest Common Subsequence Problem

- **Brute Force**
  One way to solve the longest common subsequence problem is to enumerate all subsequences of X and take the largest one that is also a subsequence of Y. Since each character of X is either in or not in a subsequence, there are potentially 2^n different subsequences of X, each of which requires O(m) time to determine whether it is a subsequence of Y. Thus, this brute-force approach yields an exponential-time algorithm that runs in $O(2^n m)$ time, which is very inefficient.

- **Dynamics Programming**

L	-1	0	1	2	3	4	5	6	7	8	9	10	11
-1	0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	1	1	1	1	1	1	1	1	1	1	1
1	0	0	1	1	2	2	2	2	2	2	2	2	2
2	0	0	1	1	2	2	2	3	3	3	3	3	3
3	0	1	1	1	2	2	2	3	3	3	3	3	3
4	0	1	1	1	2	2	2	3	3	3	3	3	3
5	0	1	1	1	2	2	2	3	4	4	4	4	4
6	0	1	1	2	2	3	3	3	4	4	5	5	5
7	0	1	1	2	2	3	4	4	4	4	5	5	6
8	0	1	1	2	3	3	4	5	5	5	5	5	6
9	0	1	1	2	3	4	4	5	5	5	6	6	6



Navlakhi®

If X[i] = Y [j] then

$$L[i,j]=L[i-1,j-1]+1 \text{ if } x_i = y_j$$

If X[i] ≠ Y[j] then

$$L[i,j] = \max\{L[i-1,j], L[i,j-1]\}$$

Performance: Given a string X of n characters and a string Y of m characters, we can find the longest common subsequence of X and Y in O(nm) time.

Justification: Algorithm LCS computes L[n − 1,m − 1], the length of a longest common subsequence, in O(nm) time. Given the table of L[i, j] values, constructing a longest common subsequence is straightforward. One method is to start from L[n, m] and work back through the table, reconstructing a longest common subsequence from back to front. At any position L[i, j], we can determine whether X[i] = Y[j] . If this is true, then we can take X[i] as the next character of the subsequence (noting that X[i] is before the previous character we found, if any), moving next to L[i − 1, j − 1]. If X[i] ≠ Y[j] , then we can move to the larger of L[i, j − 1] and L[i −1,j]. We stop when we reach a boundary cell (with i = − 1 or j = −1). This method constructs a longest common subsequence in O(n + m) additional time.

Algorithm LCS(X,Y):
  Input: Strings X and Y with n and m elements, respectively
  Output: For $i = 0,\ldots,n-1$, $j = 0,\ldots,m-1$, the length $L[i,j]$ of a longest string that is a subsequence of both the string $X[0..i] = x_0x_1x_2\cdots x_i$ and the string $Y[0..j] = y_0y_1y_2\cdots y_j$

```
for i ← −1 to n − 1 do
    L[i, −1] ← 0
for j ← 0 to m − 1 do
    L[−1, j] ← 0
for i ← 0 to n − 1 do
    for j ← 0 to m − 1 do
        if x_i = y_j then
            L[i, j] ← L[i − 1, j − 1] + 1
        else
            L[i, j] ← max{L[i − 1, j], L[i, j − 1]}
return array L
```

Greedy Method

- General Method
- Knapsack problem
- Job sequencing with deadlines
- Minimum cost spanning trees-Kruskal and prim's algorithm
- Optimal storage on tapes
- Single source shortest path

GENERAL METHOD

```
Algorithm Greedy(a, n)
// a[1 : n] contains the n inputs.
{
    solution := ∅; // Initialize the solution.
    for i := 1 to n do
    {
        x := Select(a);
        if Feasible(solution, x) then
            solution := Union(solution, x);
    }
    return solution;
}
```

The greedy algorithm works in stages, considering one input at a time. At each stage, a decision is made regarding whether a particular input is in an optimal solution. If the inclusion of the next input into the partially constructed optimal solution will result in an infeasible solution, then this into is not added to the partial solution. Otherwise, it is added.

e.g. consider that Rs.33 are to be paid by a customer to a shopkeep & he hands over a Rs.100. Hence the shopkeeper needs to return back Rs.67. In the greedy approach we are trying to minimize the number of currency notes (or coins) given. Note the full amount Rs.67 has to be paid back, then only it's a feasible solution. But naturally the shopkeeper returns Rs.67 by first paying back Rs.50, then Rs.10, then Rs.5 & then two Rs.2 coins. Greedy approach is often a very natural approach to solving a problem.

KNAPSACK PROBLEM

There are n objects and a knapsack (or bag) available. Each object has an associated weight(w_i) & profit(p_i). Each knapsack has a maximum capacity it can carry. Whatever algorithm we may use we have to try to

$$\text{maximize} \sum_{1 \le i \le n} p_i x_i$$

$$\text{subject to} \sum_{1 \le i \le n} w_i x_i \le m$$

$$\text{and } 0 \le x_i \le 1, \quad 1 \le i \le n$$

The profits and weights are positive numbers.

Consider the following problem n=3, m=20, (p1,p2,p3)=(25,24,15), and (w1,w2,w3)=(18,15,10). The problem can be solved in 4 ways, all feasible but only one is optimal

Option 1: Ordered by profit

p1=25 w1=18
p2=24 w2=15
p3=15 w3=10

We select item 1 first, as its expected to grow the profit the maximum. Thus the knapsack capacity was 20, but now after inserting item1 its 20 – 18 =2. These 2 units will be taken from item2 resulting in a profit of 2/15*24 = 3.2. Thus the total profit is 25+3.2 = 28.2

Option 2: Ordered by descending on weight

In order to accommodate most items we can choose by decrasing weights

w3=10 p3=15
w2=15 p2=24
w1=18 p1=25

We select item 3 first, the knapsack capacity is 20 & after adding item3 its 20 – 10=10. Now adding item2 the profit is totally 15 + (10/15) 24 = 31. The answer turned out to be more than option1, hence option1 wasn’t optimal.

Option 3: Ordering by profit by weigth ratio

p2/w2=1.6 w2=15 p2=24
p3/w3=1.5 w3=10 p3=15
p1/w1=1.39 w1=18 p1=25

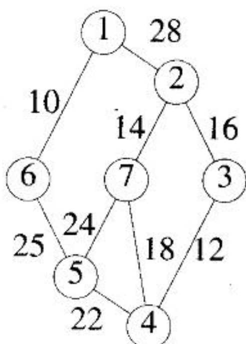
We choose item2 in our knapsack. The knapsack capacity from 20 is now reduced to 20 – 15 = 5. Now we choose item3. The total profit is 24 + 5*1.5 = 31.5. Hence Option 2 was feasible but not optimal.

Irrespective of the ordering (assuming that the elements are ordered in the option of choice), greedy algoritm can be implemented as follows:

```
Algorithm GreedyKnapsack(m, n)
{
    for i := 1 to n do x[i] := 0.0; // Initialize x.
    U := m;
    for i := 1 to n do
    {
        if (w[i] > U) then break;
        x[i] := 1.0; U := U - w[i];
    }
    if (i ≤ n) then x[i] := U/w[i];
}
```

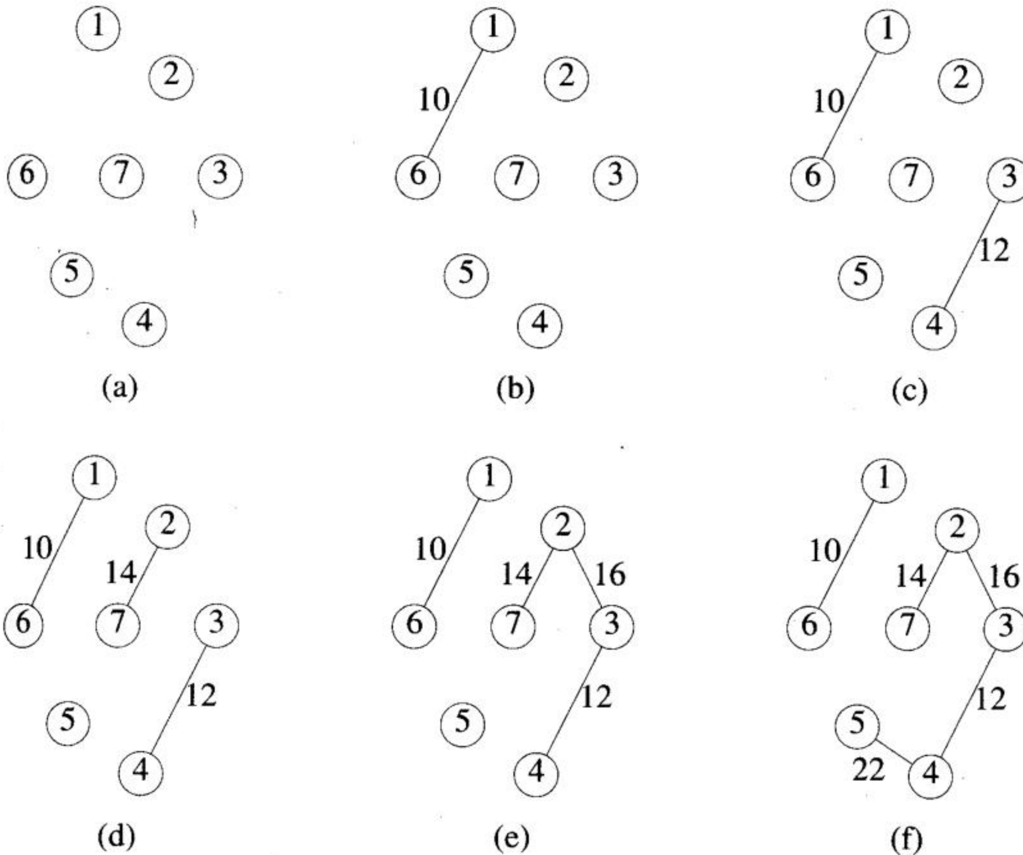
KRUSKAL'S ALGORITHM

The objective is to construct a minimum cost tress from a given graph. The graph is entered in a form of a n X n matrix with the weights entered for corresponding row & column representing the corresponding nodes of the graph. Consider the following graph



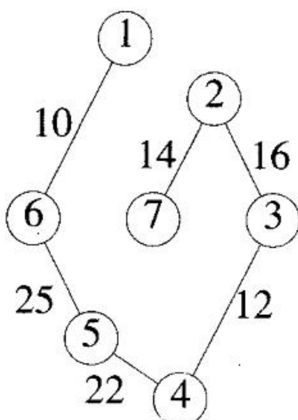
Navlakhi®

We start choosing the cheapest edge first i.e. 1 – 6. Then the next cheapest & so on… At each step no loops should be formed. If a loop is being formed at any step, we discard that selection & look for the next cheapest edge. This technique is demonstrated below



igure 4.9 Stages in Kruskal's algorithm

The final answer being



Navlakhi®

The basic algorithm being

```
t := ∅;
while ((t has less than n − 1 edges) and (E ≠ ∅)) do
{
    Choose an edge (v, w) from E of lowest cost;
    Delete (v, w) from E;
    if (v, w) does not create a cycle in t then add (v, w) to t;
    else discard (v, w);
}
```

A more detailed implementation can consider making a min heap for selecting the minimum & then to reheapify it for getting the next minimum & so on. The algorithm is shown below:

```
Algorithm Kruskal(E, cost, n, t)
{
    Construct a heap out of the edge costs using Heapify;
    for i := 1 to n do parent[i] := −1;
    // Each vertex is in a different set.
    i := 0; mincost := 0.0;
    while ((i < n − 1) and (heap not empty)) do
    {
        Delete a minimum cost edge (u, v) from the heap
        and reheapify using Adjust;
        j := Find(u); k := Find(v);
        if (j ≠ k) then
        {
            i := i + 1;
            t[i, 1] := u; t[i, 2] := v;
            mincost := mincost + cost[u, v];
            Union(j, k);
        }
    }
    if (i ≠ n − 1) then write ("No spanning tree");
    else return mincost;
}
```

Find(u) and Find(v) check to see it the two already exists & helps in deciding to know if inserting the new edge would create any loop. j and k represent the sets to which the vertices u & v belong. If the sets are the same (i.e. j=k) then the current edge will result in a loop, hence discarded. But, if j≠k, implies that u & v belong to different sets, hence the current edge can be inserted into the forest. Our final objective is to generate a tree if minimum cost, but the intermediate stages may not be a tree.

Performance

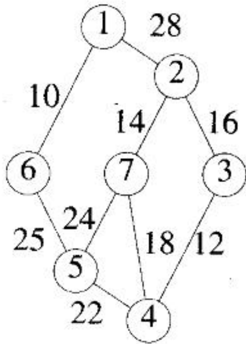
Maintaining the structure as a heap allows us to find the next edge in O(log |E|) time. The heap construction will take O(|E|) time. Hence the worst case time is O(|E|log|E|)

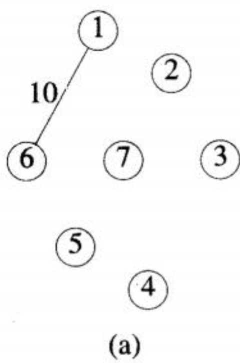
Navlakhi®

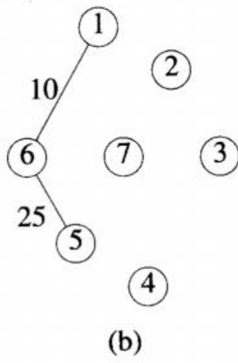
PRIM'S ALGORITHM

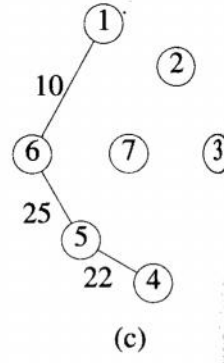
This is another method for creating a MST (minimum spanning tree). Here we start off with the least cost edge and then we proceed from any visited node (i.e. node in the tree) to any unvisited node (not in the tree) which has the next cheapest cost. Since we are going from an existing to a new node, there are no chances of loop formation, hence the need to detect loops no longer exists in Prim's algorithm.

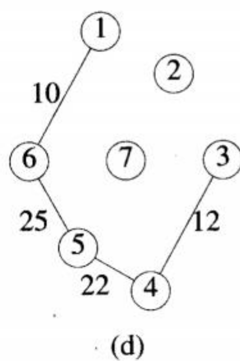
Considering the same example we show how to solve a problem involving Prim's algorihm

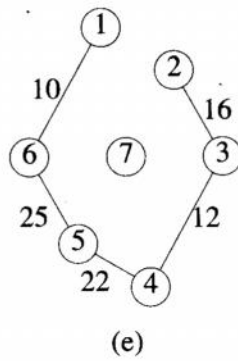












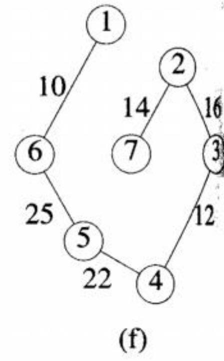


Figure 4.8 Stages in Prim's algorithm

Navlakhi®

```
Algorithm Prim(E, cost, n, t)
// E is the set of edges in G. cost[1 : n, 1 : n] is the cost
// adjacency matrix of an n vertex graph such that cost[i, j] is
// either a positive real number or ∞ if no edge (i, j) exists.
// A minimum spanning tree is computed and stored as a set of
// edges in the array t[1 : n − 1, 1 : 2]. (t[i, 1], t[i, 2]) is an edge in
// the minimum-cost spanning tree. The final cost is returned.
{
    Let (k, l) be an edge of minimum cost in E;
    mincost := cost[k, l];
    t[1, 1] := k; t[1, 2] := l;
    for i := 1 to n do // Initialize near.
        if (cost[i, l] < cost[i, k]) then near[i] := l;
        else near[i] := k;
    near[k] := near[l] := 0;
    for i := 2 to n − 1 do
    { // Find n − 2 additional edges for t.
        Let j be an index such that near[j] ≠ 0 and
        cost[j, near[j]] is minimum;
        t[i, 1] := j; t[i, 2] := near[j];
        mincost := mincost + cost[j, near[j]];
        near[j] := 0;
        for k := 1 to n do // Update near[ ].
            if ((near[k] ≠ 0) and (cost[k, near[k]] > cost[k, j]))
                then near[k] := j;
    }
    return mincost;
}
```

ithm 4.9 Prim's minimum-cost spanning tree algorithm

As can be seen we start from the edge with minimum cost (k,l). Array t is a 2 column array with column 1 representing the source & column 2 the destination. Thus the source of the 1st edge of the tree t[1,1]=k and the destination of this 1st edge is t[1,2]. Each vertex – vertex pair has a cost associated with it (finite positive if the edge exists & +∞ if the edge does not exists between the pair of vertices). Once the first edge is selected the neighbor of each vertex is set to either k or l depending on who is more cheaper (in terms of cost). [Note Prims algorithm chooses edges from visited to non-visited node, hence each unvisited node neighbor has to be one of the visited nodes]. For the remaining n -2 edges (to connect n nodes we need n – 1 edges) we repeat a similar procedure of finding minimum cost between some j & near[j] (note near[j] will always be visited node) for near[j]≠0 (all unvisited nodes have near[j] as non-zero, hence j is the unvisited node). We choose the j, update the array t to add another edge & update

Navlakhi®

near[j] to 0 & also update the near array for all unvisited vertices if they are nearer to this newly chosen node instead of the previously chosen node (Note near[unvisited]=closest visited node).

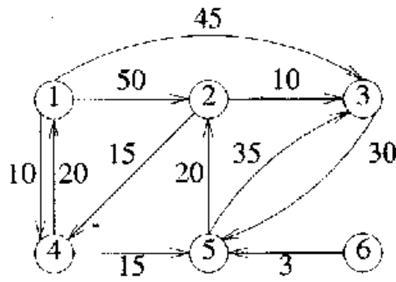
Performance

Prims, in the above form, take time $O(n^2)$. The for loop of i going from 2 to n – 1 internally has 2 independent (not nested) loops first of time O(n) & the other time O(|E|) (one to find j index & the other to update 'near'). This the total time take is $O(n^2)$.

If the unvisited (non – tree) nodes are stored in a more efficient structure like a red – black tree then the searching time is O(log n) & the updating of near has to examine only nodes adjacent to j hence the time there is O(|E|) and O(log n) for updating. Hence the total time is O((n+|E|) log n).

SINGLE SOURCE SHORTEST PATH (Dijkstra)

We are given a directed graph G=(V,E), a weighting function cost of edges of G, and a source vertex v_0. The problem is to determine the shortest paths from v_0 to all the remaining vertices of G. It is assumed that all the weights are positive.



(a) Graph

	Path	Length
1)	1, 4	10
2)	1, 4, 5	25
3)	1, 4, 5, 2	45
4)	1, 3	45

(b) Shortest paths from 1

Graph and shortest paths from vertex 1 to all destinations

```
Algorithm ShortestPaths(v, cost, dist, n)
// dist[j], 1 ≤ j ≤ n, is set to the length of the shortest
// path from vertex v to vertex j in a digraph G with n
// vertices. dist[v] is set to zero. G is represented by its
// cost adjacency matrix cost[1 : n, 1 : n].
{
    for i := 1 to n do
    { // Initialize S.
        S[i] := false; dist[i] := cost[v, i];
    }
    S[v] := true; dist[v] := 0.0; // Put v in S.
    for num := 2 to n do
    {
        // Determine n - 1 paths from v.
        Choose u from among those vertices not
        in S such that dist[u] is minimum;
        S[u] := true; // Put u in S.
        for (each w adjacent to u with S[w] = false) do
            // Update distances.
            if (dist[w] > dist[u] + cost[u, w])) then
                    dist[w] := dist[u] + cost[u, w];
    }
}
```


Navlakhi®

Performance

Time taken by the algorithm on a graph with n vertices is $O(n^2)$. The second for loop is executed n – 2 times & each execution requires O(n) time to find the minimum distance & update the distances. Thus the asymptotic time would be $O(n^2)$.

If a more efficient storage structure like a adjacency list is used then the update of distance would take O(|E|) time, since *dist* can change only for the vertices adjacent from u. If V – S is maintained in a red – black tree, then, finding the minimum would take O(log n) time. But this has to be done approx. n times, thus O(n log n). Updating an edge would also take O(log n) time & since there are |E| edges to update, the time for update is O(|E|log n). Thus the total time would be
O(n log n + |E| log n) = O((n + |E|)log n)

JOB SCHEDULING WITH DEADLINES

Theorem 4.6 Function JS is a correct implementation of the greedy-based method described above.

Proof: Since $d[i] \geq 1$, the job with the largest p_i will always be in the greedy solution. As the jobs are in nonincreasing order of the p_i's, line

```
1    Algorithm JS(d, j, n)
2    // d[i] ≥ 1, 1 ≤ i ≤ n are the deadlines, n ≥ 1. The jobs
3    // are ordered such that p[1] ≥ p[2] ≥ ··· ≥ p[n]. J[i]
4    // is the ith job in the optimal solution, 1 ≤ i ≤ k.
5    // Also, at termination d[J[i]] ≤ d[J[i + 1]], 1 ≤ i < k.
6    {
7        d[0] := J[0] := 0; // Initialize.
8        J[1] := 1; // Include job 1.
9        k := 1;
10       for i := 2 to n do
11       {
12           // Consider jobs in nonincreasing order of p[i]. Find
13           // position for i and check feasibility of insertion.
14           r := k;
15           while ((d[J[r]] > d[i]) and (d[J[r]] ≠ r)) do r := r − 1;
16           if ((d[J[r]] ≤ d[i]) and (d[i] > r)) then
17           {
18               // Insert i into J[ ].
19               for q := k to (r + 1) step −1 do J[q + 1] := J[q];
20               J[r + 1] := i; k := k + 1;
21           }
22       }
23       return k;
24   }
```

Algorithm 4.7 Greedy algorithm for sequencing unit time jobs with deadlines and profits

8 in Algorithm 4.7 includes the job with largest p_i. The **for** loop of line 10 considers the remaining jobs in the order required by the greedy method described earlier. At all times, the set of jobs already included in the solution is maintained in J. If $J[i]$, $1 \leq i \leq k$, is the set already included, then J is such that $d[J[i]] \leq d[J[i+1]]$, $1 \leq i < k$. This allows for easy application of the feasibility test of Theorem 4.4. When job i is being considered, the **while** loop of line 15 determines where in J this job has to be inserted. The use of a fictitious job 0 (line 7) allows easy insertion into position 1. Let w be such that $d[J[w]] \leq d[i]$ and $d[J[q]] > d[i]$, $w < q \leq k$. If job i is included into J, then jobs $J[q]$, $w < q \leq k$, have to be moved one position up in J (line 19). From Theorem 4.4, it follows that such a move retains feasibility of J iff $d[J[q]] \neq q$, $w < q \leq k$. This condition is verified in line 15. In

Navlakhi®

addition, i can be inserted at position $w+1$ iff $d[i] > w$. This is verified in line 16 (note $r = w$ on exit from the **while** loop if $d[J[q]] \neq q$, $w < q \leq k$). The correctness of JS follows from these observations. □

This rule simply delays the processing of job i as much as possible. Consequently, when J is being built up job by job, jobs already in J do not have to be moved from their assigned slots to accommodate the new job. If for the new job being considered there is no α as defined , then it cannot be included in J.

Example 4.6 Let $n = 5, (p_1, \ldots, p_5) = (20, 15, 10, 5, 1)$ and $(d_1, \ldots, d_5) = (2, 2, 1, 3, 3)$. Using the above feasibility rule, we have

J	assigned slots	job considered	action	profit
$\emptyset$	none	1	assign to [1, 2]	0
{1}	[1, 2]	2	assign to [0, 1]	20
{1, 2}	[0, 1], [1, 2]	3	cannot fit; reject	35
{1, 2}	[0, 1], [1, 2]	4	assign to [2, 3]	35
{1, 2, 4}	[0, 1], [1, 2], [2, 3]	5	reject	40

The optimal solution is $J = \{1, 2, 4\}$ with a profit of 40. □

Navlakhi®

4.7 OPTIMAL STORAGE ON TAPES

There are n programs that are to be stored on a computer tape of length l. Associated with each program i is a length $l_i, 1 \leq i \leq n$. Clearly, all programs can be stored on the tape if and only if the sum of the lengths of the programs is at most l. We assume that whenever a program is to be retrieved from this tape, the tape is initially positioned at the front. Hence,

Example 4.11 Let $n = 3$ and $(l_1, l_2, l_3) = (5, 10, 3)$. There are $n! = 6$ possible orderings. These orderings and their respective d values are:

ordering I	$d(I)$		
1, 2, 3	5 + 5 + 10 + 5 + 10 + 3	=	38
1, 3, 2	5 + 5 + 3 + 5 + 3 + 10	=	31
2, 1, 3	10 + 10 + 5 + 10 + 5 + 3	=	43
2, 3, 1	10 + 10 + 3 + 10 + 3 + 5	=	41
3, 1, 2	3 + 3 + 5 + 3 + 5 + 10	=	29
3, 2, 1	3 + 3 + 10 + 3 + 10 + 5	=	34

The optimal ordering is 3, 1, 2. □

```
1   Algorithm Store(n, m)
2   // n is the number of programs and m the number of tapes.
3   {
4       j := 0; // Next tape to store on
5       for i := 1 to n do
6       {
7           write ("append program", i,
8               "to permutation for tape", j);
9           j := (j + 1) mod m;
10      }
11  }
```

Algorithm 4.13 Assigning programs to tapes

Backtracking

- The general method
- 8 queens problem
- Sum of subsets
- Graph colouring

N-QUEENS PROBLEM

The objective is to place N Queens on a N X N chessboard so that no queen kills any other queen.

```
1   Algorithm Place(k, i)
2   // Returns true if a queen can be placed in kth row and
3   // ith column. Otherwise it returns false. x[ ] is a
4   // global array whose first (k − 1) values have been set.
5   // Abs(r) returns the absolute value of r.
6   {
7       for j := 1 to k − 1 do
8           if ((x[j] = i) // Two in the same column
9                or (Abs(x[j] − i) = Abs(j − k)))
10                   // or in the same diagonal
11              then return false;
12      return true;
13  }
```

Algorithm 7.4 Can a new queen be placed?

```
1   Algorithm NQueens(k, n)
2   // Using backtracking, this procedure prints all
3   // possible placements of n queens on an n × n
4   // chessboard so that they are nonattacking.
5   {
6       for i := 1 to n do
7       {
8           if Place(k, i) then
9           {
10              x[k] := i;
11              if (k = n) then write (x[1 : n]);
12              else NQueens(k + 1, n);
13          }
14      }
15  }
```

Algorithm 7.5 All solutions to the n-queens problem

Navlakhi®

BACKTRACKING – SUM OF SUBSETS

```
1   Algorithm SumOfSub(s, k, r)
2   // Find all subsets of w[1 : n] that sum to m. The values of x[j],
3   // 1 ≤ j < k, have already been determined. s = Σ_{j=1}^{k-1} w[j] * x[j]
4   // and r = Σ_{j=k}^{n} w[j]. The w[j]'s are in nondecreasing order.
5   // It is assumed that w[1] ≤ m and Σ_{i=1}^{n} w[i] ≥ m.
6   {
7       // Generate left child. Note: s + w[k] ≤ m since B_{k-1} is true.
8       x[k] := 1;
9       if (s + w[k] = m) then write (x[1 : k]); // Subset found
10          // There is no recursive call here as w[j] > 0, 1 ≤ j ≤ n.
11      else if (s + w[k] + w[k + 1] ≤ m)
12          then SumOfSub(s + w[k], k + 1, r − w[k]);
13      // Generate right child and evaluate B_k.
14      if ((s + r − w[k] ≥ m) and (s + w[k + 1] ≤ m)) then
15      {
16          x[k] := 0;
17          SumOfSub(s, k + 1, r − w[k]);
18      }
19  }
```

Algorithm 7.6 Recursive backtracking algorithm for sum of subsets problem

The problem is to figure out, given a set of subsets each with weights w[i], can there be a combination that can add up to m. The weights are already sorted in ascending order. We start with the 1st item, i.e. k=1. For any k, we select the item by setting x[k]=1. The condition to stop recursion (when we get a feasible solution) is when the sum so far (i.e. s) plus this new item's weight w[k] equals m, we stop & display the list. If we haven't reached a solution & the weight so far + w[k]+ weight of next item w[k+1] is lesser than the capacity m , then we can select k+1 item also. Hence we call the function again where sum so far is set to s+w[k], item to choose is k+1, and the residue weight so far is r – w[k] . If no solution is being reached & the addition of (k+1)th item, for some k, causes the weights total to exceed m, then we have to drop one of the previous item & hence for an alternate route to solving the problem. Note before dropping say the kth item we first need to check that a solution is possible without this item (i.e. sum without the kth item i.e. 's'+ sum of residues assuming kth item not taken i.e. r – w[k] is ≥m, means solution yet possible without kth item) and if with (k+1)th item [next just heavier item… NOTE the list is already arranged in terms of weight] added to the current sum 's' results is a sum ≤m, then we drop kth item (i.e. x[k]=0) and choose the (k+1)th item by calling the function again with the parameters of old sum, k as k+1 and residue as r – w[k] (as kth item unavailable).

GRAPH COLOURING

```
1   Algorithm mColoring(k)
2   // This algorithm was formed using the recursive backtracking
3   // schema. The graph is represented by its boolean adjacency
4   // matrix G[1 : n, 1 : n]. All assignments of 1, 2, ..., m to the
5   // vertices of the graph such that adjacent vertices are
6   // assigned distinct integers are printed. k is the index
7   // of the next vertex to color.
8   {
9       repeat
10      {// Generate all legal assignments for x[k].
11          NextValue(k); // Assign to x[k] a legal color.
12          if (x[k] = 0) then return; // No new color possible
13          if (k = n) then     // At most m colors have been
14                              // used to color the n vertices.
15              write (x[1 : n]);
16          else mColoring(k + 1);
17      } until (false);
18  }
```

Algorithm 7.7 Finding all m-colorings of a graph

The objective is to colour all 'n' node with 'm' colours with no two adjacent nodes having the same colour. The function NextValue checks for node k if we can find a valid colour. x is an array & it represents the node colour. If x[k] is zero means no valid colour could be found. If k reaches n then all nodes are over. If neither of these 2 is possible then we need to proceed to the next node.

```
1   Algorithm NextValue(k)
2   // x[1],...,x[k-1] have been assigned integer values in
3   // the range [1,m] such that adjacent vertices have distinct
4   // integers. A value for x[k] is determined in the range
5   // [0,m]. x[k] is assigned the next highest numbered color
6   // while maintaining distinctness from the adjacent vertices
7   // of vertex k. If no such color exists, then x[k] is 0.
8   {
9       repeat
10      {
11          x[k] := (x[k] + 1) mod (m + 1); // Next highest color.
12          if (x[k] = 0) then return; // All colors have been used.
13          for j := 1 to n do
14          {   // Check if this color is
15              // distinct from adjacent colors.
16              if ((G[k,j] ≠ 0) and (x[k] = x[j]))
17              // If (k,j) is and edge and if adj.
18              // vertices have the same color.
19                  then break;
20          }
21          if (j = n + 1) then return; // New color found
22      } until (false); // Otherwise try to find another color.
23  }
```

Algorithm 7.8 Generating a next color

1. Choose the next colour for node k.
2. If no colours left (i.e. x[k]=0) then return control back
3. Loop through all the columns of the adjacency matrix for row k
    a. If edge found G[k,j]≠0 & the adjacent node has same colour (i.e. x[k]=x[j]) then go back to step 1
4. Return control back

Dynamic Programming

- The general method
- Multistage Graphs
- All pair shortest path
- Single source shortest path
- Optimal BST
- 0/1 knapsack
- TSP
- Flow shop scheduling

DYNAMIC PROGRAMMING – MULTISTAGE GRAPH

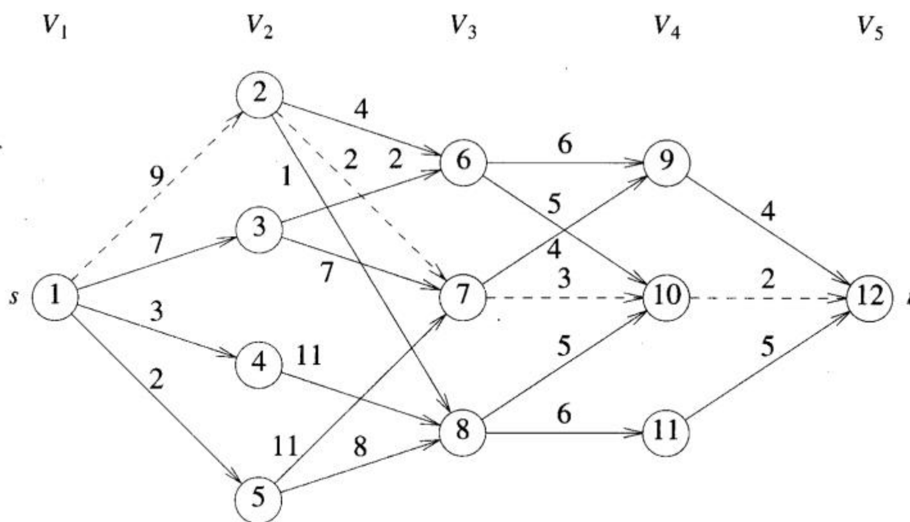


Figure 5.2 Five-stage graph

Navlakhi®

$$
\begin{aligned}
cost(3,6) &= \min\ \{6 + cost(4,9), 5 + cost(4,10)\} \\
&= 7 \\
cost(3,7) &= \min\ \{4 + cost(4,9), 3 + cost(4,10)\} \\
&= 5 \\
cost(3,8) &= 7 \\
cost(2,2) &= \min\ \{4 + cost(3,6), 2 + cost(3,7), 1 + cost(3,8)\} \\
&= 7 \\
cost(2,3) &= 9 \\
cost(2,4) &= 18
\end{aligned}
$$

$$
\begin{aligned}
cost(2,5) &= 15 \\
cost(1,1) &= \min\ \{9 + cost(2,2), 7 + cost(2,3), 3 + cost(2,4), \\
&\qquad 2 + cost(2,5)\} \\
&= 16
\end{aligned}
$$

The complexity analysis of the function FGraph is fairly straightforward. If G is represented by its adjacency lists, then r in line 9 of Algorithm 5.1 can be found in time proportional to the degree of vertex j. Hence, if G has $|E|$ edges, then the time for the **for** loop of line 7 is $\Theta(|V| + |E|)$. The time for the **for** loop of line 16 is $\Theta(k)$. Hence, the total time is $\Theta(|V| + |E|)$. In addition to the space needed for the input, space is needed for $cost[\ ]$, $d[\ ]$, and $p[\ ]$.

```
1   Algorithm FGraph(G, k, n, p)
2   // The input is a k-stage graph G = (V, E) with n vertices
3   // indexed in order of stages. E is a set of edges and c[i, j]
4   // is the cost of ⟨i, j⟩. p[1 : k] is a minimum-cost path.
5   {
6       cost[n] := 0.0;
7       for j := n − 1 to 1 step −1 do
8       { // Compute cost[j].
9           Let r be a vertex such that ⟨j, r⟩ is an edge
10          of G and c[j, r] + cost[r] is minimum;
11          cost[j] := c[j, r] + cost[r];
12          d[j] := r;
13      }
14      // Find a minimum-cost path.
15      p[1] := 1; p[k] := n;
16      for j := 2 to k − 1 do p[j] := d[p[j − 1]];
17  }
```

Algorithm 5.1 Multistage graph pseudocode corresponding to the forward approach

Navlakhi®

```
1   Algorithm BGraph(G, k, n, p)
2   // Same function as FGraph
3   {
4       bcost[1] := 0.0;
5       for j := 2 to n do
6       { // Compute bcost[j].
7           Let r be such that ⟨r, j⟩ is an edge of
8           G and bcost[r] + c[r, j] is minimum;
9           bcost[j] := bcost[r] + c[r, j];
10          d[j] := r;
11      }
12      // Find a minimum-cost path.
13      p[1] := 1; p[k] := n;
14      for j := k - 1 to 2 do p[j] := d[p[j + 1]];
15  }
```

Algorithm 5.2 Multistage graph pseudocode corresponding to backward approach

```
Algorithm AllPaths(cost, A, n)
// cost[1 : n, 1 : n] is the cost adjacency matrix of a graph with
// n vertices; A[i, j] is the cost of a shortest path from vertex
// i to vertex j. cost[i, i] = 0.0, for 1 ≤ i ≤ n.
{
    for i := 1 to n do
        for j := 1 to n do
            A[i, j] := cost[i, j]; // Copy cost into A.
    for k := 1 to n do
        for i := 1 to n do
            for j := 1 to n do
                A[i, j] := min(A[i, j], A[i, k] + A[k, j]);
}
```

If for some k, the cost of i to j is costlier than i to k and k to j, then we update the cost of i to j to cost of i,k + cost of k,j. Thus we need a tripple for loop & the efficiency is $O(n^3)$.

```
Algorithm BellmanFord(v, cost, dist, n)
// Single-source/all-destinations shortest
// paths with negative edge costs
{
    for i := 1 to n do // Initialize dist.
        dist[i] := cost[v, i];
    for k := 2 to n − 1 do
        for each u such that u ≠ v and u has
                at least one incoming edge do
            for each ⟨i, u⟩ in the graph do
                if dist[u] > dist[i] + cost[i, u] then
                    dist[u] := dist[i] + cost[i, u];
}
```

Here we have to find the shortest path of all nodes from v. Again we start by initializing the distance of all nodes from v in array dist (if there is no link then too it will be initialized to ∞). For every node u & its adjacent node i incident on u, we see if distance of u (from v) is more than the distance of i from v + cost of link i,v then we update the distance of u to distance of i + cost of link i,v.

Since there are 3 nested for loop each a function of n, hence the efficiency is $O(n^3)$.

DYNAMIC PROGRAMMING – TSP

The objective of the travelling salesperson problem is to start from a node, travel through all the nodes only once & reach back to the start node.

Example 5.26 Consider the directed graph of Figure 5.21(a). The edge lengths are given by matrix c of Figure 5.21(b).

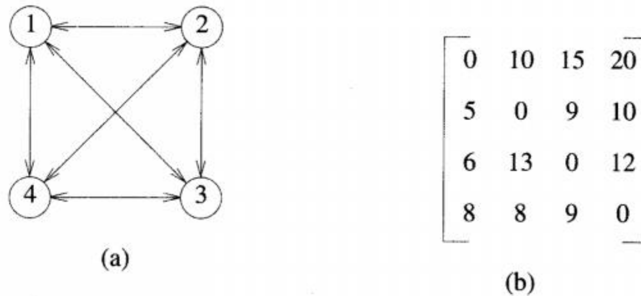


Figure 5.21 Directed graph and edge length matrix c

Thus $g(2,\phi) = c_{21} = 5, g(3,\phi) = c_{31} = 6$, and $g(4,\phi) = c_{41} = 8$. Using (5.21), we obtain

$$
\begin{array}{lllll}
g(2,\{3\}) & = & c_{23} + g(3,\phi) & = & 15 \\
g(3,\{2\}) & = & 18 & & \\
g(4,\{2\}) & = & 13 & &
\end{array}
\qquad
\begin{array}{lll}
g(2,\{4\}) & = & 18 \\
g(3,\{4\}) & = & 20 \\
g(4,\{3\}) & = & 15
\end{array}
$$

Next, we compute $g(i,S)$ with $|S| = 2$, $i \neq 1$, $1 \notin S$ and $i \notin S$.

$$
\begin{array}{lllll}
g(2,\{3,4\}) & = & \min\ \{c_{23} + g(3,\{4\}), c_{24} + g(4,\{3\})\} & = & 25 \\
g(3,\{2,4\}) & = & \min\ \{c_{32} + g(2,\{4\}), c_{34} + g(4,\{2\})\} & = & 25 \\
g(4,\{2,3\}) & = & \min\ \{c_{42} + g(2,\{3\}), c_{43} + g(3,\{2\})\} & = & 23
\end{array}
$$

Finally, from (5.20) we obtain

$$
\begin{array}{lll}
g(1,\{2,3,4\}) & = & \min\{c_{12} + g(2,\{3,4\}), c_{13} + g(3,\{2,4\}), c_{14} + g(4,\{2,3\})\} \\
& = & \min\{35, 40, 43\} \\
& = & 35
\end{array}
$$

An optimal tour of the graph of Figure 5.21(a) has length 35. A tour of this length can be constructed if we retain with each $g(i,S)$ the value of j that minimizes the right-hand side of (5.21). Let $J(i,S)$ be this value. Then, $J(1,\{2,3,4\}) = 2$. Thus the tour starts from 1 and goes to 2. The remaining tour can be obtained from $g(2, \{3, 4\})$. So $J(2, \{3, 4\}) = 4$. Thus the next edge is $\langle 2,4 \rangle$. The remaining tour is for $g(4, \{3\})$. So $J(4, \{3\}) = 3$. The optimal tour is 1, 2, 4, 3, 1. □

Navlakhi®

OBST

0/1 KNAPSACK

Let me start with an example

5.10 FLOW SHOP SCHEDULING

Often the processing of a job requires the performance of several distinct tasks. Computer programs run in a multiprogramming environment are input and then executed. Following the execution, the job is queued for output and the output eventually printed. In a general flow shop we may have n jobs each requiring m tasks $T_{1i}, T_{2i}, \ldots, T_{mi}$, $1 \leq i \leq n$, to be performed. Task T_{ji} is to be performed on processor P_j, $1 \leq j \leq m$. The time required to complete task T_{ji} is t_{ji}. A schedule for the n jobs is an assignment of tasks to time intervals on the processors. Task T_{ji} must be assigned to processor P_j. No processor may have more than one task assigned to it in any time interval. Additionally, for any job i the processing of task T_{ji}, $j > 1$, cannot be started until task $T_{j-1,i}$ has been completed.

Example 5.27 Two jobs have to be scheduled on three processors. The task times are given by the matrix $\mathcal{J}$

$$\mathcal{J} = \begin{bmatrix} 2 & 0 \\ 3 & 3 \\ 5 & 2 \end{bmatrix}$$

Two possible schedules for the jobs are shown in Figure 5.22. □

A *nonpreemptive* schedule is a schedule in which the processing of a task on any processor is not terminated until the task is complete. A schedule for which this need not be true is called *preemptive*. The schedule of Figure 5.22(a) is a preemptive schedule. Figure 5.22(b) shows a nonpreemptive schedule. The *finish time* $f_i(S)$ of job i is the time at which all tasks of job i have been completed in schedule S. In Figure 5.22(a), $f_1(S) = 10$ and $f_2(S) = 12$. In Figure 5.22(b), $f_1(S) = 11$ and $f_2(S) = 5$. The finish time $F(S)$ of a schedule S is given by

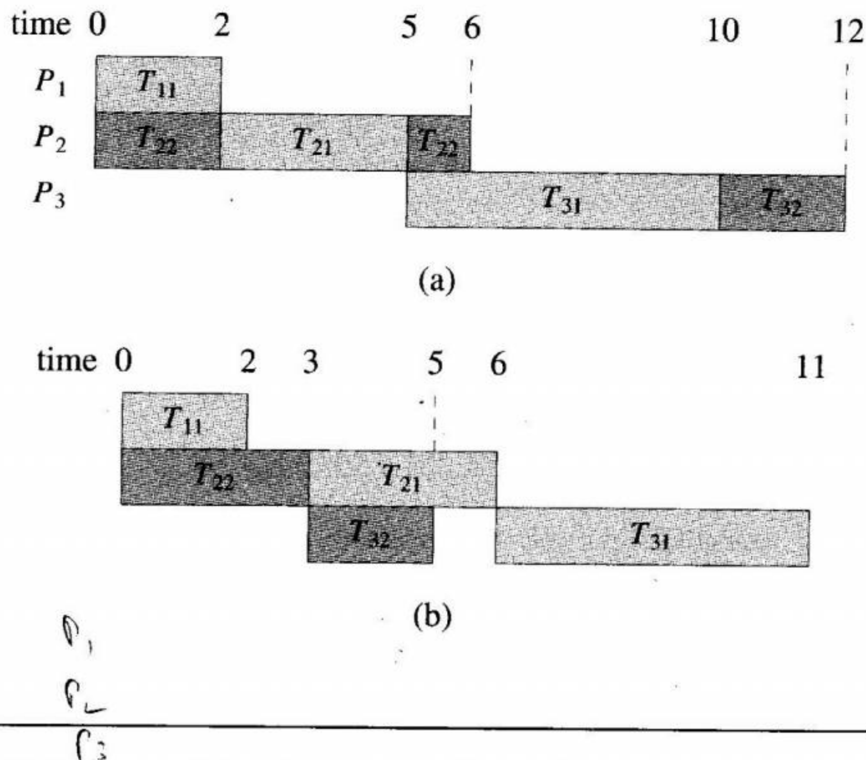


Figure 5.22 Two possible schedules for Example 5.27

$$F(S) = \max_{1 \le i \le n} \{f_i(S)\} \tag{5.22}$$

The *mean flow time* MFT(S) is defined to be

$$\text{MFT}(S) = \frac{1}{n} \sum_{1 \le i \le n} f_i(S) \tag{5.23}$$

An optimal finish time (OFT) schedule for a given set of jobs is a nonpreemptive schedule S for which $F(S)$ is minimum over all nonpreemptive schedules S. A preemptive optimal finish time (POFT) schedule, optimal mean finish time schedule (OMFT), and preemptive optimal mean finish (POMFT) schedule are defined in the obvious way.

Although the general problem of obtaining OFT and POFT schedules for $m > 2$ and of obtaining OMFT schedules is computationally difficult (see Chapter 11), dynamic programming leads to an efficient algorithm to obtain OFT schedules for the case $m = 2$. In this section we consider this special case.

Navlakhi®

Hence, it suffices to generate any schedule for which (5.29) holds for every pair of adjacent jobs. We can obtain a schedule with this property by making the following observations from (5.29). If $\min\{a_1, a_2, \ldots, a_n, b_1, b_2, \ldots, b_n\}$ is a_i, then job i should be the first job in an optimal schedule. If $\min\{a_1, a_2, \ldots, a_n, b_1, b_2, \ldots, b_n\}$ is b_j, then job j should be the last job in an optimal schedule. This enables us to make a decision as to the positioning of one of the n jobs. Equation 5.29 can now be used on the remaining $n - 1$ jobs to correctly position another job, and so on. The scheduling rule resulting from (5.29) is therefore:

1. Sort all the a_i's and b_j's into nondecreasing order.
2. Consider this sequence in this order. If the next number in the sequence is a_j and job j hasn't yet been scheduled, schedule job j at the leftmost available spot. If the next number is b_j and job j hasn't yet been scheduled, schedule job j at the rightmost available spot. If j has already been scheduled, go to the next number in the sequence.

Note that the above rule also correctly positions jobs with $a_i = 0$. Hence, these jobs need not be considered separately.

Example 5.28 Let $n = 4$, $(a_1, a_2, a_3, a_4) = (3, 4, 8, 10)$, and $(b_1, b_2, b_3, b_4) = (6, 2, 9, 15)$. The sorted sequence of a's and b's is $(b_2, a_1, a_2, b_1, a_3, b_3, a_4, b_4) = (2, 3, 4, 6, 8, 9, 10, 15)$. Let $\sigma_1, \sigma_2, \sigma_3$, and σ_4 be the optimal schedule. Since the smallest number is b_2, we set $\sigma_4 = 2$. The next number is a_1 and we set $\sigma_1 = a_1$. The next smallest number is a_2. Job 2 has already been scheduled. The next number is b_1. Job 1 has already been scheduled. The next is a_3 and we set σ_3. This leaves σ_3 free and job 4 unscheduled. Thus, $\sigma_3 = 4$. □

The scheduling rule above can be implemented to run in time $O(n \log n)$ (see exercises). Solving (5.24) and (5.25) directly for $g(1, 2, \ldots, n, 0)$ for the optimal schedule will take $\Omega(2^n)$ time as there are these many different S's for which $g(S, t)$ will be computed.

Branch & Bound

- The General method
- 15 puzzle: An example
- Travelling salesman problem

BRANCH BOUND – LC SEARCH – LEAST COST SEARCH

Terminology

- **Definition 1**

Live node is a node that has been generated but whose children have not yet been generated.

- **Definition 2**

E-node is a live node whose children are currently being explored. In other words, an E-node is a node currently being expanded.

- **Definition 3**

Dead node is a generated node that is not to be expanded or explored any further. All children of a dead node have already been expanded.

- **Definition 4**

Branch-and-bound refers to all state space search methods in which all children of an E-node are generated before any other live node can become the E-node.

```
listnode = record {
    listnode *next, *parent; float cost;
}

Algorithm LCSearch(t)
// Search t for an answer node.
{
    if *t is an answer node then output *t and return;
    E := t; // E-node.
    Initialize the list of live nodes to be empty;
    repeat
    {
        for each child x of E do
        {
            if x is an answer node then output the path
                from x to t and return;
            Add(x); // x is a new live node.
            (x → parent) := E; // Pointer for path to root.
        }
        if there are no more live nodes then
        {
            write ("No answer node"); return;
        }
        E := Least();
    } until (false);
}
```

Let t be the E-node to be expanded. Expanding the E-node results in number of live nodes. For each such child check if it is the answer. If so then stop & output the solution. If not then store it for further reference. If any live nodes left to be expanded, then choose the least one (LC based technique) & make it the E-node & repeat the cycle. If no more live nodes left & we haven't yet reached a solution then return NO ANSWER NODE POSSIBLE.

NOTE: The choice of the next E-Node if it is based on some minimum cost then its LC (Least Cost) method. If choices are put in a queue & then the selection is made then it is called FIFO method & if a stack is used instead of a queue then its called LIFO method. (Least is the pop/dequeue function)

Navlakhi®

15 PUZZLE

- 15 numbered tiles on a square frame with a capacity for 16 tiles
- Given an initial arrangement, transform it to the goal arrangement through a series of legal moves

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

Initial Arrangement

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

Goal Arrangement

- Legal move involves moving a tile adjacent to the empty spot E to E
- Four possible moves in the initial state above: tiles 2, 3, 5, 6
- Each move creates a new arrangement of tiles, called *state* of the puzzle
- Initial and goal states
- A state is *reachable* from the initial state iff there is a sequence of legal moves from initial state to this state
- The state space of an initial state is all the states that can be reached from initial state
- Search the state space for the goal state and use the path from initial state to goal state as the answer
- Number of possible arrangments for tiles: $16! \approx 20.9 \times 10^{12}$
  * Only about half of them are reachable from any given initial state
- Check whether the goal state is reachable from initial state
  * Number the frame positions from 1 to 16
  * p_i is the frame position containing tile numbered i
  * p_{16} is the position of empty spot
  * For any state, let l_i be the number of tiles j such that $j < i$ and $p_j > p_i$
  * For the initial arrangement above, $l_1 = 0$, $l_4 = 1$, and $l_{12} = 6$
  * Let $x = 1$ if in the initial state, the empty spot is in one of the following positions: $2, 4, 5, 7, 10, 12, 13, 15$; otherwise $x = 0$

  Theorem 1 *The goal state is reachable from the initial state iff* $\sum_{i=1}^{16} l_i + x$ *is even.*

NOTE:
l1=number of number after 1 which are less than 1 & after 1 i.e. 0 in this case
l2= number of number after 2 which are less than 2 & after 2 i.e. 0 in this case (take blank as 16)
l3= number of number after 3 which are less than 3 & after 3 i.e. 1 in this case (take blank as 16)
l4= number of number after 4 which are less than 4 & after 4 i.e. 1 in this case (take blank as 16)
l5= number of number after 5 which are less than 5 & after 5 i.e. 0 in this case (take blank as 16)
l6= number of number after 6 which are less than 6 & after 6 i.e. 0 in this case (take blank as 16)
l7= number of number after 7 which are less than 7 & after 7 i.e. 1 in this case (take blank as 16)
l8= number of number after 8 which are less than 8 & after 8 i.e. 0 in this case (take blank as 16)
l9= number of number after 9 which are less than 9 & after 9 i.e. 0 in this case (take blank as 16)
l10= number of number after 10 which are less than 10 & after 10 i.e. 0 in this case (take blank as 16)
l11= number of number after 11 which are less than 11 & after 11 i.e. 3 in this case (take blank as 16)
l12= number of number after 12 which are less than 12 & after 12 i.e. 6 in this case (take blank as 16)
& so on till l16

Now since blank is in slot 6 of the board, we take x=0

Navlakhi®

If sum of all l's above & x is odd, then solution not possible from this position.

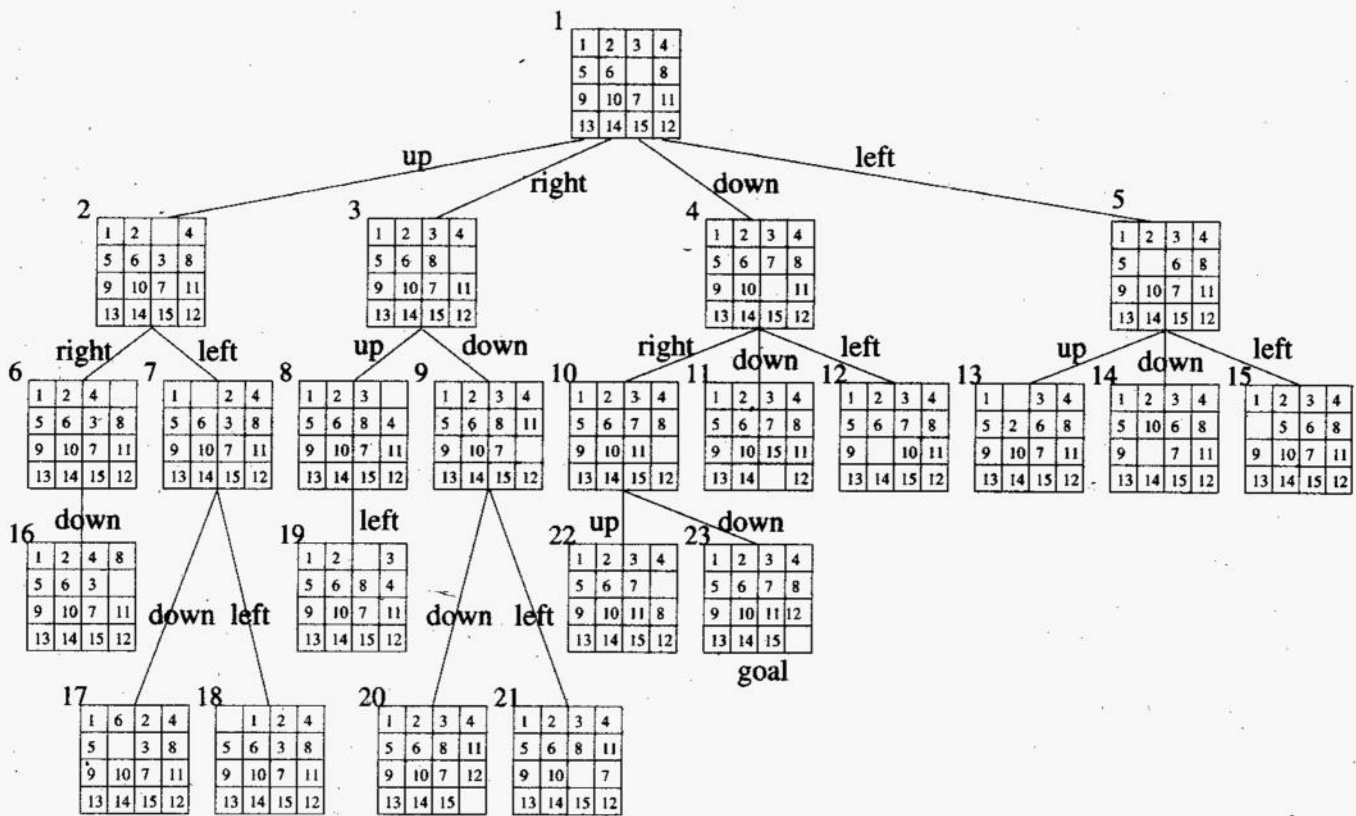
If it was even then we now test for (utmost) 4 possibilities out of this board & find the ∑l + x for each & if any is odd then that part of the tree will not be expanded further (bounded).

[SEE EXAMPLE SOLVED IN CLASS]

NOTE:
The above in LIFO would mean that we need to stack the (utmost) 4 option we get from the current board & then pop one & work on it & then push in the other options resulting from it, & so on (obviously bounding the branches where $\sum_{i=1}^{16} l_i + x$ is odd).

Using a queue instead of a stack will result in the FIFO method.

For LC Search method we need to device a cost function which can be calculated at each stage & the choice of the next E-Node can be made based on the cheapest cost. Such a cost function can be called c(x) which is the length of path from the root node to a nearest goal node (if any) in the sub tree rooted at x.



Edges are labeled according to the direction in which the empty space moves

Thus, c(1)=c(4)=c(10)=c(23)=3. If we had the costs then a very efficient search can be made from the root node to the goal state along the path where c(x) is same. Hence the only E-nodes are the nodes along the path from the root to the goal state. Unfortunately, this is an impractical strategy as we cannot find c(x) looking at a node x &/or its predecessors only.

Thus we try to

Compute an estimate $\hat{c}(x)$ of $c(x)$

$\hat{c}(x) = f(x) + \hat{g}(x)$ where $f(x)$ is the length of the path from root to x and $\hat{g}(x)$ is an estimate of the length of a shortest path from x to a goal node in the subtree with root x

One possible choice for $\hat{g}(x)$ is the number of nonblank tiles not in their goal position

There are at least $\hat{g}(x)$ moves to transform state x to a goal state

- $\hat{c}(x)$ is a lower bound on $c(x)$

At each stage among all the live – nodes we choose that E-Node which has the least approximation $\hat{c}(x)$ & then make it the E-Node & generate its live nodes, & then from the pool of all live nodes we choose the next least approximation & so on. LC, with a good choice of $\hat{c}(x)$, leads to a quicker solution than FIFO & LIFO.

[TRY TO RE-DO THE CLASSWORK SUM USING LC METHOD]

TRAVELLING SALESMAN PROBLEM

8.3 TRAVELING SALESPERSON (*)

An $O(n^2 2^n)$ dynamic programming algorithm for the traveling salesperson problem was arrived at in Section 5.9. We now investigate branch-and-bound algorithms for this problem. Although the worst-case complexity of these algorithms will not be any better than $O(n^2 2^n)$, the use of good bounding functions will enable these branch-and-bound algorithms to solve some problem instances in much less time than required by the dynamic programming algorithm.

Let $G = (V, E)$ be a directed graph defining an instance of the traveling salesperson problem. Let c_{ij} equal the cost of edge $\langle i, j \rangle$, $c_{ij} = \infty$ if $\langle i, j \rangle \notin E$, and let $|V| = n$. Without loss of generality, we can assume that every tour starts and ends at vertex 1. So, the solution space S is given by $S = \{1, \pi, 1 | \pi$ is a permutation of $(2, 3, \ldots, n)\}$. Then $|S| = (n-1)!$. The size of S can be reduced by restricting S so that $(1, i_1, i_2, \ldots, i_{n-1}, 1) \in S$ iff $\langle i_j, i_{j+1} \rangle \in E$, $0 \leq j \leq n-1$, and $i_0 = i_n = 1$. S can be organized into a state space tree similar to that for the n-queens problem (see Figure 7.2). Figure 8.10 shows the tree organization for the case of a complete graph with $|V| = 4$. Each leaf node L is a solution node and represents the tour defined by the path from the root to L. Node 14 represents the tour $i_0 = 1, i_1 = 3, i_2 = 4, i_3 = 2$, and $i_4 = 1$.

To use LCBB to search the traveling salesperson state space tree, we need to define a cost function $c(\cdot)$ and two other functions $\hat{c}(\cdot)$ and $u(\cdot)$ such that $\hat{c}(r) \leq c(r) \leq u(r)$ for all nodes r. The cost $c(\cdot)$ is such that the solution node with least $c(\cdot)$ corresponds to a shortest tour in G. One choice for $c(\cdot)$ is

$$c(A) = \begin{cases} \text{length of tour defined by the path from the root to } A, & \text{if } A \text{ is a leaf} \\ \text{cost of a minimum-cost leaf in the subtree } A, & \text{if } A \text{ is not a leaf} \end{cases}$$

A simple $\hat{c}(\cdot)$ such that $\hat{c}(A) \leq c(A)$ for all A is obtained by defining $\hat{c}(A)$ to be the length of the path defined at node A. For example, the path defined at node 6 of Figure 8.10 is $i_0, i_1, i_2 = 1, 2, 4$. It consists of the edges $\langle 1, 2 \rangle$ and $\langle 2, 4 \rangle$. A better $\hat{c}(\cdot)$ can be obtained by using the reduced cost matrix corresponding to G. A row (column) is said to be *reduced* iff it contains at

Navlakhi®

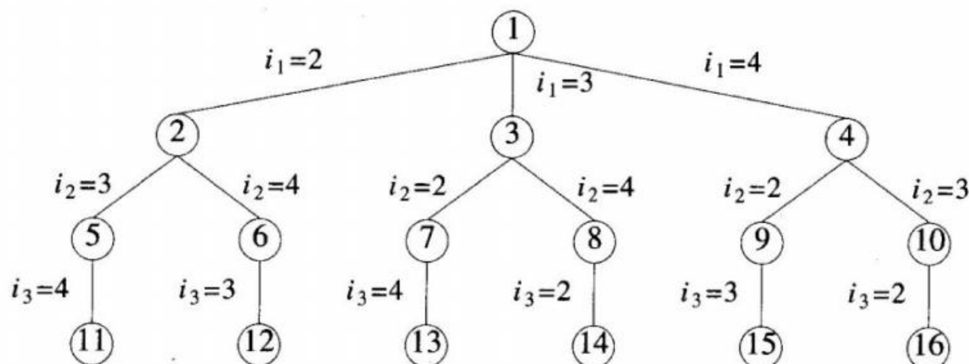


Figure 8.10 State space tree for the traveling salesperson problem with $n = 4$ and $i_0 = i_4 = 1$

least one zero and all remaining entries are non-negative. A matrix is *reduced* iff every row and column is reduced. As an example of how to reduce the cost matrix of a given graph G, consider the matrix of Figure 8.11(a). This corresponds to a graph with five vertices. Since every tour on this graph includes exactly one edge $\langle i,j\rangle$ with $i = k$, $1 \le k \le 5$, and exactly one edge $\langle i,j\rangle$ with $j = k$, $1 \le k \le 5$, subtracting a constant t from every entry in one column or one row of the cost matrix reduces the length of every tour by exactly t. A minimum-cost tour remains a minimum-cost tour following this subtraction operation. If t is chosen to be the minimum entry in row i (column j), then subtracting it from all entries in row i (column j) introduces a zero into row i (column j). Repeating this as often as needed, the cost matrix can be reduced. The total amount subtracted from the columns and rows is a lower bound on the length of a minimum-cost tour and can be used as the $\hat{c}$ value for the root of the state space tree. Subtracting 10, 2, 2, 3, 4, 1, and 3 from rows 1, 2, 3, 4, and 5 and columns 1 and 3 respectively of the matrix of Figure 8.11(a) yields the reduced matrix of Figure 8.11(b). The total amount subtracted is 25. Hence, all tours in the original graph have a length at least 25.

We can associate a reduced cost matrix with every node in the traveling salesperson state space tree. Let A be the reduced cost matrix for node R. Let S be a child of R such that the tree edge (R, S) corresponds to including edge $\langle i,j\rangle$ in the tour. If S is not a leaf, then the reduced cost matrix for S may be obtained as follows: (1) Change all entries in row i and column j of A to ∞. This prevents the use of any more edges leaving vertex i or entering vertex j. (2) Set $A(j, 1)$ to ∞. This prevents the use of edge $\langle j, 1\rangle$. (3) Reduce all rows and columns in the resulting matrix except for rows and

Navlakhi®

columns containing only ∞. Let the resulting matrix be B. Steps (1) and (2) are valid as no tour in the subtree s can contain edges of the type $\langle i, k\rangle$ or $\langle k, j\rangle$ or $\langle j, 1\rangle$ (except for edge $\langle i, j\rangle$). If r is the total amount subtracted in step (3) then $\hat{c}(S) = \hat{c}(R) + A(i,j) + r$. For leaf nodes, $\hat{c}(\cdot) = c()$ is easily computed as each leaf defines a unique tour. For the upper bound function u, we can use $u(R) = \infty$ for all nodes R.

$$\begin{bmatrix} \infty & 20 & 30 & 10 & 11 \\ 15 & \infty & 16 & 4 & 2 \\ 3 & 5 & \infty & 2 & 4 \\ 19 & 6 & 18 & \infty & 3 \\ 16 & 4 & 7 & 16 & \infty \end{bmatrix}$$

(a) Cost matrix

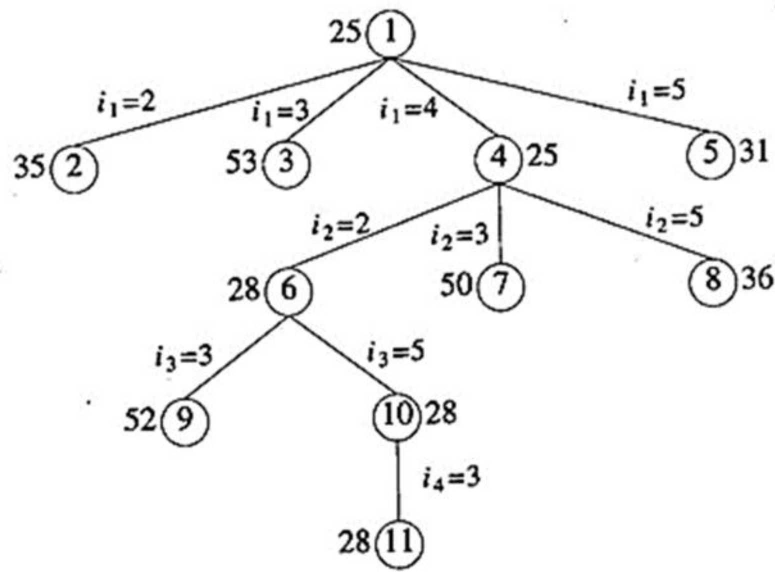
$$\begin{bmatrix} \infty & 10 & 17 & 0 & 1 \\ 12 & \infty & 11 & 2 & 0 \\ 0 & 3 & \infty & 0 & 2 \\ 15 & 3 & 12 & \infty & 0 \\ 11 & 0 & 0 & 12 & \infty \end{bmatrix}$$

(b) Reduced cost matrix
L = 25

Figure 8.11 An example

Let us now trace the progress of the LCBB algorithm on the problem instance of Figure 8.11(a). We use $\hat{c}$ and u as above. The initial reduced matrix is that of Figure 8.11(b) and *upper* $= \infty$. The portion of the state space tree that gets generated is shown in Figure 8.12. Starting with the root node as the E-node, nodes 2, 3, 4, and 5 are generated (in that order). The reduced matrices corresponding to these nodes are shown in Figure 8.13. The matrix of Figure 8.13(b) is obtained from that of 8.11(b) by (1) setting all entries in row 1 and column 3 to ∞, (2) setting the element at position (3, 1) to ∞, and (3) reducing column 1 by subtracting by 11. The $\hat{c}$ for node 3 is therefore 25 + 17 (the cost of edge $\langle 1,3\rangle$ in the reduced matrix) + 11 = 53. The matrices and $\hat{c}$ value for nodes 2, 4, and 5 are obtained similarly. The value of *upper* is unchanged and node 4 becomes the next E-node. Its children 6, 7, and 8 are generated. The live nodes at this time are nodes 2, 3, 5, 6, 7, and 8. Node 6 has least $\hat{c}$ value and becomes the next E-node. Nodes 9 and 10 are generated. Node 10 is the next E-node. The solution node, node 11, is generated. The tour length for this node is $\hat{c}(11) = 28$ and *upper* is updated to 28. For the next E-node, node 5, $\hat{c}(5) = 31 >$ *upper*. Hence, LCBB terminates with 1, 4, 2, 5, 3, 1 as the shortest length tour.

Navlakhi®



Numbers outside the node are $\hat{c}$ values

Figure 8.12 State space tree generated by procedure LCBB

$$\begin{bmatrix} \infty & \infty & \infty & \infty & \infty \\ \infty & \infty & 11 & 2 & 0 \\ 0 & \infty & \infty & 0 & 2 \\ 15 & \infty & 12 & \infty & 0 \\ 11 & \infty & 0 & 12 & \infty \end{bmatrix}$$

(a) Path 1,2; node 2

$$\begin{bmatrix} \infty & \infty & \infty & \infty & \infty \\ 1 & \infty & \infty & 2 & 0 \\ \infty & 3 & \infty & 0 & 2 \\ 4 & 3 & \infty & \infty & 0 \\ 0 & 0 & \infty & 12 & \infty \end{bmatrix}$$

(b) Path 1,3; node 3

$$\begin{bmatrix} \infty & \infty & \infty & \infty & \infty \\ 12 & \infty & 11 & \infty & 0 \\ 0 & 3 & \infty & \infty & 2 \\ \infty & 3 & 12 & \infty & 0 \\ 11 & 0 & 0 & \infty & \infty \end{bmatrix}$$

(c) Path 1,4; node 4

$$\begin{bmatrix} \infty & \infty & \infty & \infty & \infty \\ 10 & \infty & 9 & 0 & \infty \\ 0 & 3 & \infty & 0 & \infty \\ 12 & 0 & 9 & \infty & \infty \\ \infty & 0 & 0 & 12 & \infty \end{bmatrix}$$

(d) Path 1,5; node 5

$$\begin{bmatrix} \infty & \infty & \infty & \infty & \infty \\ \infty & \infty & 11 & \infty & 0 \\ 0 & \infty & \infty & \infty & 2 \\ \infty & \infty & \infty & \infty & \infty \\ 11 & \infty & 0 & \infty & \infty \end{bmatrix}$$

(e) Path 1,4,2; node 6

$$\begin{bmatrix} \infty & \infty & \infty & \infty & \infty \\ 1 & \infty & \infty & \infty & 0 \\ \infty & 1 & \infty & \infty & 0 \\ \infty & \infty & \infty & \infty & \infty \\ 0 & 0 & \infty & \infty & \infty \end{bmatrix}$$

(f) Path 1,4,3; node 7

$$\begin{bmatrix} \infty & \infty & \infty & \infty & \infty \\ 1 & \infty & 0 & \infty & \infty \\ 0 & 3 & \infty & \infty & \infty \\ \infty & \infty & \infty & \infty & \infty \\ \infty & 0 & 0 & \infty & \infty \end{bmatrix}$$

(g) Path 1,4,5; node 8

$$\begin{bmatrix} \infty & \infty & \infty & \infty & \infty \\ \infty & \infty & \infty & \infty & \infty \\ \infty & \infty & \infty & \infty & 0 \\ \infty & \infty & \infty & \infty & \infty \\ 0 & \infty & \infty & \infty & \infty \end{bmatrix}$$

(h) Path 1,4,2,3; node 9

$$\begin{bmatrix} \infty & \infty & \infty & \infty & \infty \\ \infty & \infty & \infty & \infty & \infty \\ 0 & \infty & \infty & \infty & \infty \\ \infty & \infty & \infty & \infty & \infty \\ \infty & \infty & 0 & \infty & \infty \end{bmatrix}$$

(i) Path 1,4,2,5; node 10

Figure 8.13 Reduced cost matrices corresponding to nodes in Figure 8.12

Navlakhi®

DIVIDE & CONQUER

- General method
- Binary search
- Finding minimum and maximum
- Merge sort analysis
- Quick sort analysis
- Strassen's matrix multiplication
- The problem of multiplying long integers
- - constructing Tennis tournament

BINARY SEARCH

```
1   Algorithm BinSrch(a, i, l, x)
2   // Given an array a[i : l] of elements in nondecreasing
3   // order, 1 ≤ i ≤ l, determine whether x is present, and
4   // if so, return j such that x = a[j]; else return 0.
5   {
6       if (l = i) then  // If Small(P)
7       {
8           if (x = a[i]) then return i;
9           else return 0;
10      }
11      else
12      { // Reduce P into a smaller subproblem.
13          mid := ⌊(i + l)/2⌋;
14          if (x = a[mid]) then return mid;
15          else if (x < a[mid]) then
16                  return BinSrch(a, i, mid − 1, x);
17              else return BinSrch(a, mid + 1, l, x);
18      }
19  }
```

Algorithm 3.3 Recursive binary search

Navlakhi®

```
1   Algorithm BinSearch(a, n, x)
2   // Given an array a[1 : n] of elements in nondecreasing
3   // order, n ≥ 0, determine whether x is present, and
4   // if so, return j such that x = a[j]; else return 0.
5   {
6       low := 1; high := n;
7       while (low ≤ high) do
8       {
9           mid := ⌊(low + high)/2⌋;
10          if (x < a[mid]) then high := mid - 1;
11          else if (x > a[mid]) then low := mid + 1;
12               else return mid;
13      }
14      return 0;
15  }
```

Algorithm 3.4 Iterative binary search

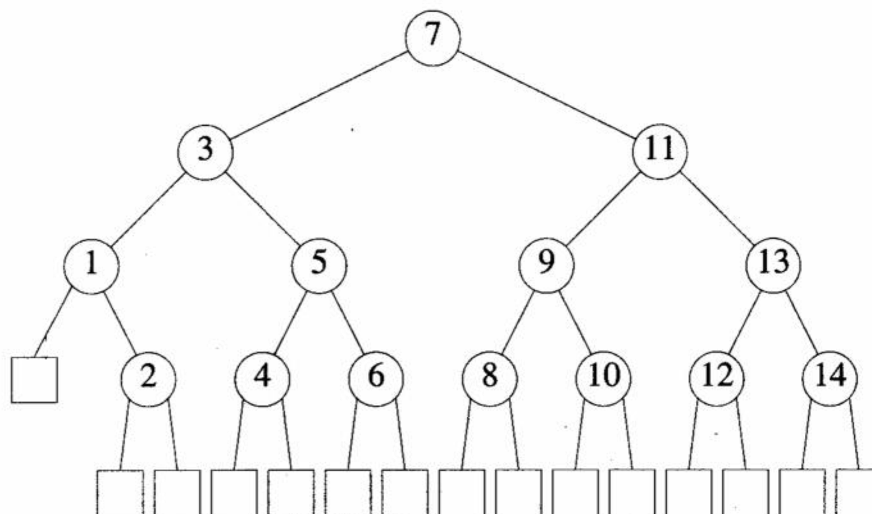


Figure 3.4 Binary decision tree for binary search, $n = 14$

Navlakhi®

```
1   Algorithm BinSearch1(a, n, x)
2   // Same specifications as BinSearch except n > 0
3   {
4       low := 1; high := n + 1;
5       // high is one more than possible.
6       while (low < (high - 1)) do
7       {
8           mid := ⌊(low + high)/2⌋;
9           if (x < a[mid]) then high := mid;
10              // Only one comparison in the loop.
11          else low := mid; // x ≥ a[mid]
12      }
13      if (x = a[low]) then return low; // x is present.
14      else return 0; // x is not present.
15  }
```

Algorithm 3.5 Binary search using one comparison per cycle

Navlakhi®

FINDING MIN MAX

The procedure is initially invoked by the statement

n = 9

$$\text{MaxMin}(1, n, x, y)$$

Suppose we simulate MaxMin on the following nine elements:

a:	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]
	22	13	−5	−8	15	60	17	31	47

A good way of keeping track of recursive calls is to build a tree by adding a node each time a new call is made. For this algorithm each node has four items of information: i, j, max, and min. On the array $a[\,]$ above, the tree of Figure 3.5 is produced.

Examining Figure 3.5, we see that the root node contains 1 and 9 as the values of i and j corresponding to the initial call to MaxMin. This execution produces two new calls to MaxMin, where i and j have the values 1, 5 and 6, 9, respectively, and thus split the set into two subsets of approximately the same size. From the tree we can immediately see that the maximum depth of recursion is four (including the first call). The circled numbers in the upper left corner of each node represent the orders in which max and min are assigned values.

Now what is the number of element comparisons needed for MaxMin? If $T(n)$ represents this number, then the resulting recurrence relation is

$$T(n) = \begin{cases} T(\lceil n/2 \rceil) + T(\lceil n/2 \rceil) + 2 & n > 2 \\ 1 & n = 2 \\ 0 & n = 1 \end{cases}$$

When n is a power of two, $n = 2^k$ for some positive integer k, then

Navlakhi®

```
1   Algorithm MaxMin(i, j, max, min)
2   // a[1 : n] is a global array. Parameters i and j are integers,
3   // 1 ≤ i ≤ j ≤ n. The effect is to set max and min to the
4   // largest and smallest values in a[i : j], respectively.
5   {
6       if (i = j) then max := min := a[i]; // Small(P)
7       else if (i = j - 1) then // Another case of Small(P)
8           {
9               if (a[i] < a[j]) then
10              {
11                  max := a[j]; min := a[i];
12              }
13              else
14              {
15                  max := a[i]; min := a[j];
16              }
17          }
18          else
19          {   // If P is not small, divide P into subproblems.
20              // Find where to split the set.
21                  mid := ⌊(i + j)/2⌋;
22              // Solve the subproblems.
23                  MaxMin(i, mid, max, min);
24                  MaxMin(mid + 1, j, max1, min1);
25              // Combine the solutions.
26                  if (max < max1) then max := max1;
27                  if (min > min1) then min := min1;
28          }
29  }
```

Algorithm 3.7 Recursively finding the maximum and minimum

Navlakhi®

$$
\begin{aligned}
T(n) &= 2T(n/2) + 2 \\
&= 2(2T(n/4) + 2) + 2 \\
&= 4T(n/4) + 4 + 2 \\
&\vdots \\
&= 2^{k-1}T(2) + \textstyle\sum_{1 \le i \le k-1} 2^i \\
&= 2^{k-1} + 2^k - 2 = 3n/2 - 2
\end{aligned}
$$

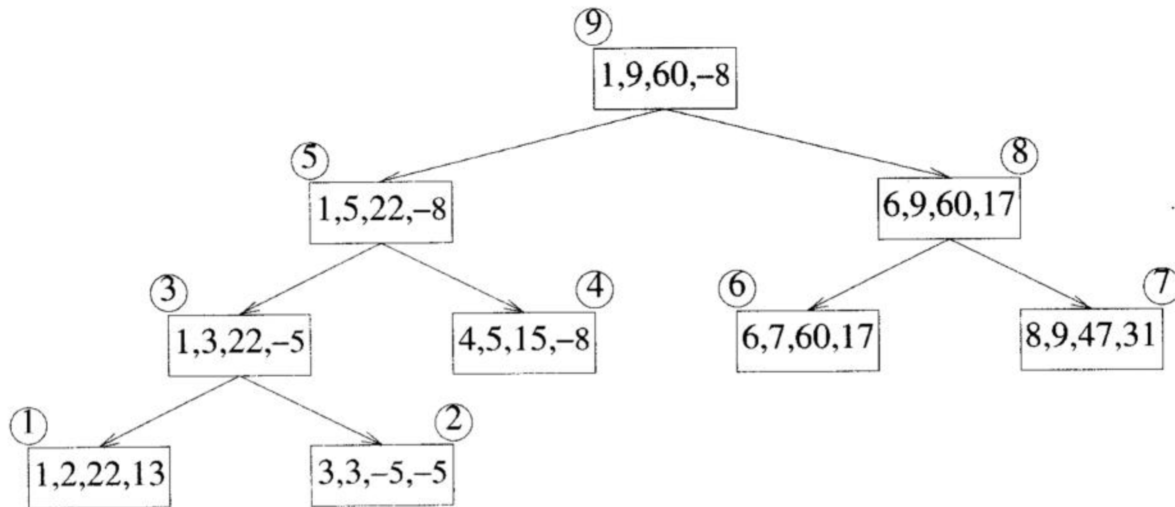


Figure 3.5 Trees of recursive calls of MaxMin

Note that $3n/2 - 2$ is the best-, average-, and worst-case number of comparisons when n is a power of two.

Compared with the $2n - 2$ comparisons for the straightforward method, this is a saving of 25% in comparisons. It can be shown that no algorithm based on comparisons uses less than $3n/2 - 2$ comparisons. So in this sense algorithm MaxMin is optimal (see Chapter 10 for more details). But does this imply that MaxMin is better in practice? Not necessarily. In terms of storage, MaxMin is worse than the straightforward algorithm because it requires stack space for i, j, $max, min, max1$, and $min1$. Given n elements, there will be $\lfloor \log_2 n \rfloor + 1$ levels of recursion and we need to save seven values for each recursive call (don't forget the return address is also needed).

Navlakhi®

MERGE SORT

QUICK SORT

STRASSENS MULTIPLICATION

MULTIPLICATION OF LONG INTEGRES

TENNIS TOURNAMENT