5. Backtracking

Backtracking is a type of algorithm that is a refinement of brute force search. In backtracking, multiple solutions can be eliminated without being explicitly examined, by using specific properties of the problem. It systematically searches for a solution to a problem among all available options. It does so by assuming that the solutions are represented by vectors $(x_1, ..., x_n)$ of values and by traversing, in a depth first manner, the domains of the vectors until the solutions are found.

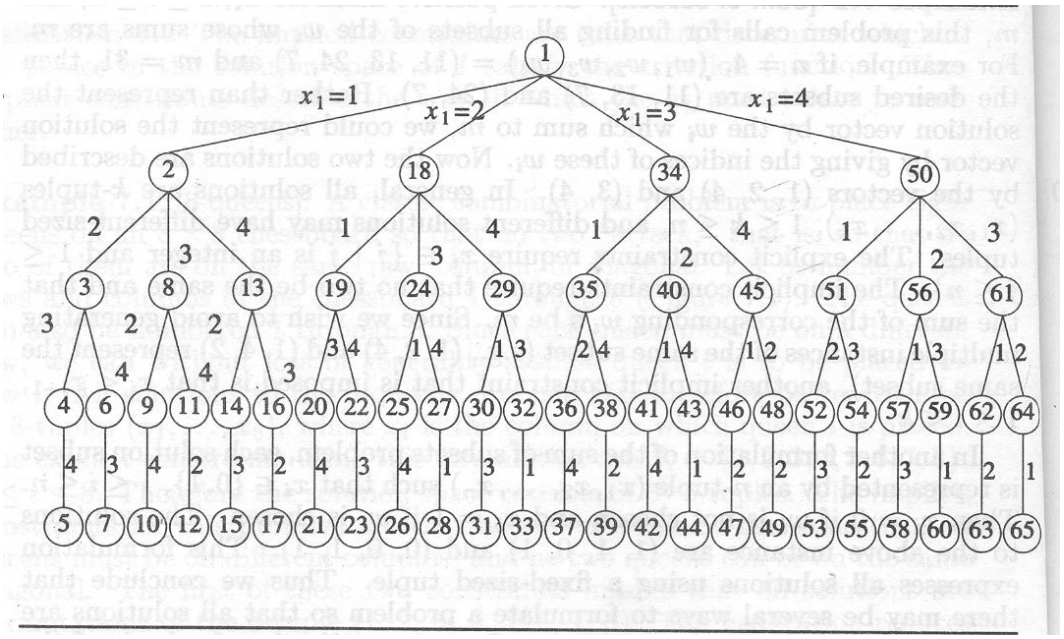
When invoked, the algorithm starts with an empty vector. At each stage it extends the partial vector with a new value. Upon reaching a partial vector $(x_1, ..., x_i)$ which can't represent a partial solution, the algorithm backtracks by removing the trailing value from the vector, and then proceeds by trying to extend the vector with alternative values. The term "backtrack" was coined by American mathematician D. H. Lehmer in the 1950s.

The constraints are divided into implicit constraints and explicit constraints. Implicit constraints are rules which determine which of the tuples in solution space of I satisfy the criteria function. And explicit constraints restrict x_i take values from a given set. e.g., $x_i \geq 0$, $x_i \in \{0,1\}$ etc.

Backtracking algorithm determine the problem solution by systematically searching the problem space with a tree representation.

Example 5.1
N-Queens Problem: N queens are to be placed on an nxn chessboard so that no two of them are in attacking position. i.e., no two queens are in same row or same row or same diagonal.
Let us assume that queen i is placed in row i. So the problem can be represented as n-tuple $(x_1, ..., x_n)$ where each x_i represents the column in which we have to place queen i.
Hence the explicit constraints is $S_i = \{1,2,..,n\}$. No two x_i can be same and no two queens can be in same diagonal is the implicit constraint. Since we fixed the row number the solution space reduce from n^n to n!.

Consider the case where n=4. A permutation tree shows all possible elements in solution space. Edge from level i to level i+1 specify the value of x_i. Hence there are 4! = 24 leaves for the tree.



Here nodes are labeled in Depth First Search order. Each node defines a *problem state*. All path from root to other nodes defines *state space* of problem. *Solution states* are those problem state s for which the path from root to s defines a tuple in solution space. *Answer states* are those solution states s for which pat from root to s defines a tuple that is member of solution (satisfies implicit constraints). The tree representation of solution space is called *state space tree*.

A node which has been generated and all of whose children are not yet been generated is called a *live node*. A live node whose children are currently been generated is called an *E-node* (node being expanded). A generated node with all its children expanded is called a *dead node*. There are two different ways to generate problem state

1. Given a list of live nodes, as soon as new child *C* of current E-node *R* is generated, the child becomes new E-node. When sub tree *C* got fully explored, *R* becomes the next E-node. Hence it will result in a depth first generation.
2. Given a list of live nodes, E-node remains as E-node until it is dead.

In both the cases, bounding functions are used to kill live nodes without generating all their children. First case with bounding function result in *backtracking* and the second case with bounding function result in *Branch and bound method*.

Backtracking: General principle

Backtracking find all answer nodes, not just one case. Let $(x_1, ..., x_i)$ be a path from root to a node in the state space tree. $T(x_1, ..., x_i)$ be the set of all possible values for x_{i+1} such that $(x_1, ..., x_i, x_{i+1})$ is also a path to problem state. So we know that $T(x_1, ..., x_n) = \varphi$. Let B_{i+1} be a bounding function such that if $B_{i+1}(x_1, ..., x_i, x_{i+1})$ is false for the path $(x_1, ..., x_i, x_{i+1})$ from the root to the problem state, then path cannot be extended to answer node. Then candidates for position i+1 are those generated by the T satisfying B_{i+1}.

Control Abstraction

```
Void Backtrack (int k)
{
```

```
    // While entering assume that first k-1 values x[1], x[2],…,x[k-1] of the solution
    //  vector x[1:n] have been assigned.

    for (each x[k] such that x[k] ∈ T(x[1],…,x[k-1]))
    {
            if (Bk(x[1],…,x[k-1]))
            {
                    if ((x[1],…,x[k]) is a path to an answer node)
                            display x[1:k];
                    if (k<n) Backtrack(k+1);
            }
    }

}
```

5.1 N Queens Problem

Problem: Given an nxn chessboard, place n queens in nom-attacking position i.e., no two queens are in same row or same row or same diagonal.

Let us assume that queen i is placed in row i. So the problem can be represented as n-tuple (x_1, ..., x_n) where each x_i represents the column in which we have to place queen i. Hence the explicit constraints is S_i ={1,2,..,n}. No two x_i can be same and no two queens can be in same diagonal is the implicit constraint. Since we fixed the row number the solution space reduce from n^n to n!. To check whether they are on the same diagobal, let chessboard be represented by an array a[1..n][1..n]. Every element with same diagonal that runs from upper left to lower right has same "row-column" value. E.g., consider the element a[4][2]. Elements a[3][1], a[5][3], a[6][4], a[7][5] and a[8][6] have row-column value 2. Similarly every element from same diagonal that goes from upper right to lower left has same "row+column" value. Tthe elements a[1][5], a[2][4], a[3][3], a[5][1] have same "row+column" value as that of element a[4][2] which is 6.

Hence two queens placed at (i,j) and (k,l) have same diagonal iff

$i - j = k - l$ or $i + j = k + l$

i.e., $j - l = i - k$ or $j - l = k - i$

$\mathbf{|j - l| = |i - k|}$

Algorithm 5.1

```
Bool Place (int k, int i)
{
    // Returns true if queen can be placed on kth row, ith column. Otherwise return false.
    // Assume that first k-1 elements of x is set.

    for( int j=1; j<k; j++)
            if ((x[j] == i) || (abs(x[j]-i) == abs(j-k)))  //same column or same diagonal
                    return false;
    return true;
}
Void NQueens (int k, int n)
```

```
{
    // Prints all possible permutations to place n queens on an nxn chess board so that
    // none of them are in attacking position.

    for (int i=1; i<=n; i++)
    {
            if (Place(k,i))
            {
                    x[k] = I;
                    if (k==n)
                            for(int j=1; j<=n; j++)
                                    print(x[j]);
                    else
                            NQueens(k+1, n);

            }
    }
```

5.2 Sum of Subsets

Problem: Given n distinct positive numbers w_i, $1 \le i \le n$ and m, find all subsets of w_i, with the sum of the elements equal to m.

e.g., Consider a set with n=4, (w_1, w_2, w_3, w_4) = (11, 13, 24, 7) and m=31. Then the subsets are (11,13,7) and (24, 7). This can also represented using their indices as (1,2,4) and (3,4). In general, all solutions are k-tuples (x_1, x_2, .., x_k), $1 \le k \le n$ and different solutions may have different sized tuples. Explicit constraints are $x_i = \{j \mid j$ is an integer and $1 \le j \le n\}$. Implicit constraints require that no two of them are same and sum of the corresponding w_i's be m.

Another formulation of subset sum problem is represented by an n-tuple (x_1, x_2, .., x_n) such that $x_i \in \{0,1\}$, $1 \le i \le n$. When $x_i = 0$, w_i is not chosen and when $x_i = 1$, w_i is chosen. Hence in the above example, solutions are (1,1,0,1) and (0,0,1,1). This method express solution using fixed-sized tuples. In both the cases, solution space consist of 2^n distinct tuples.

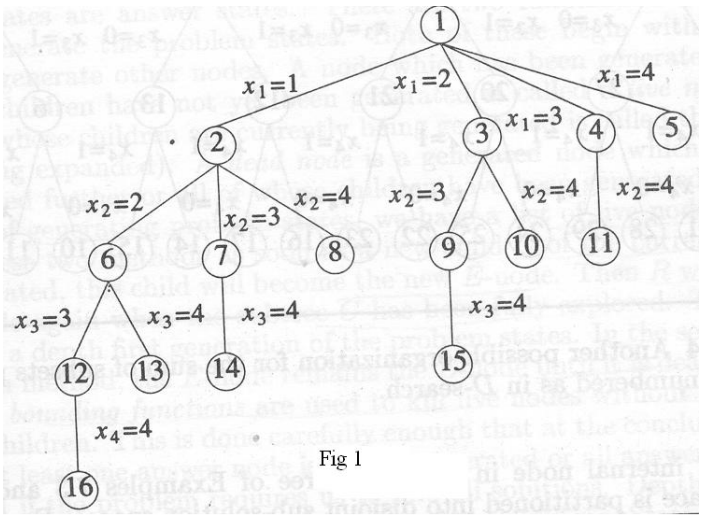


Fig 1

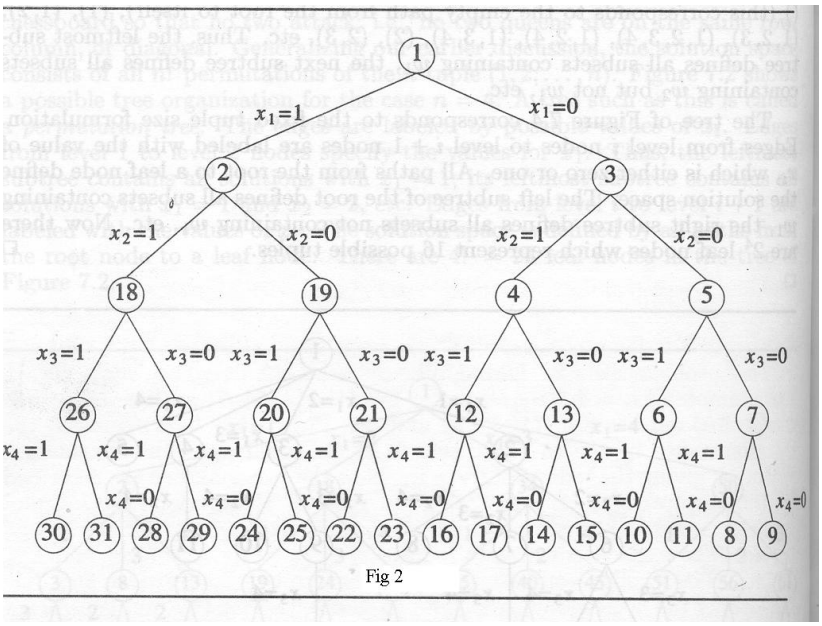


Fig 2

Fig1 and Fig2 show the Breadth first tree representation of the subset sum problem using queue and stack respectively. Fig1 uses variable size tuple and Fig2 uses fixed size tuple to represent the tree.

The bounding function chosen is $B_k(x_1, .., x_k)$ is true iff

$$\sum_{i=1}^{k} w_i x_i + \sum_{i=k+1}^{n} w_i \geq m$$

x_i's are initially in increasing order. In this case, $x_1, .., x_k$ cannot lead to an answer node if

$$\sum_{i=1}^{k} w_i x_i + w_{k+1} > m$$

Hence the bounding functions used are

$B_k(x_1, ..., x_k)$ is true iff $\sum_{i=1}^{k} w_i x_i + \sum_{i=k+1}^{n} w_i \geq m$ and $\sum_{i=1}^{k} w_i x_i + w_{k+1} \leq m$

Algorithm 5.2

```
void  SubsetSum (float s, int k, float r)
{
    //Values of x[j], 1≤j≤k, has already been determined.
//    s = Σ (j=1 to k-1) w[j]*x[j]  and  r = Σ (j=k to n) w[j]
```

// w[j]'s are in increasing order. Assume that w[1] ≥ m and $\sum_{i=1}^{n} w_i \geq m$

```
    // Generate left child. s+w[k] ≤ m, because B_{k-1} is true.
    x[k] = 1;
    if (s+w[k] == m)
            for (int j=1; j<=k; j++)
                    print(x[j]);

    else if (s+w[k]+w[k+1] <= m)
            SubsetSum(s+w[k], k+1, r-w[k]);

    // Generate right child and evaluate B_k
    if((s+r-w[k] >= m) && (s+w[k+1] <= m))
    {
            x[k] = 0;
            SubsetSum(s, k+1, r-w[k]);
    }
}
```

5.3 Knapsack Problem

Problem: Given n inputs and a knapsack or bag. Each object i is associated with a weight w_i and profit p_i. Choose a subset of items so as to fill the knapsack with maximum profit.

$$\text{Maximize} \sum_{1\leq i\leq n} p_i x_i$$

$$\text{Subject to} \sum_{1\leq i\leq n} w_i x_i \leq m \text{ and } x_i \in \{0,1\}$$

Solution space consist of 2^n distinct ways to assign 0 or 1 values to the x_i's

Bounding function: Find upper bound on the value of best feasible solution obtainable by expanding the given lie nodes and any of its descendants. If this upper bound is not higher than the value of best solution determined so far, then the remaining live nodes in the path can be killed.

Consider the fixed tuple representation of the above problem. If at node Z, the value of x_i, $1\leq i\leq k$, have been determined, then find the upper bound of all nodes from k+1 to n using Greedy method. There is a relaxation on the implicit constraint that it can take value in the range $0\leq x_i \leq 1$. Function Bound(cp, cw, k) find this. Assume that to the function we are passing items arranged in the descending order of per unit profit. i.e., $p[i]/w[i] \geq p[i+1]/w[i+1]$, $1\leq i\leq n$.

We know that the bound value for a feasible left child of a node Z is same as that of Z. Hence calculate the bound value for only the nodes in the right sub tree.

Algorithm 5.3

```
float Bound (float cp, float cw, int k)
{
    // m is the size of the knapsack

    float b=cp, c=cw;
    for (int i=k+1; i<=n; i++)
    {
            c += w[i];
            if (c<m) b += p[i];
            else return (b+(1-(c-m)/w[i]) * p[i]);
    }
    return (b);
}

void BKnap(int k, float cp, float cw)
{
    // fw is the final weight of knapsack, fp is the final maximum profit.
    // x[k]=0, if w[k] not in knapsack else x[k]=1. Initially set fp=-1.

    //Generate left child
    if (cw+w[k] <= m)
    {
            y[k] = 1;
            if (k<n) BKnap(k+1, cp+p[k], cw+w[k]);
            if((cp+p[k] >fp) && (k==n))
            {
                    fp = cp+p[k];
                    fw = cw + w[k];
                    for (int j=1; j<=k; j++)
                            x[j] = y[j];
            }
    }

    //Generate right child
    if (Bound(cp, cw, k) >= fp)
    {
            y[k] = 0;
            if (k<n) BKnap(k+1, cp, cw);
            if ((cp>fp) && (k==n))
            {
                    fp = cp;
                    fw = cw;
                    for (int j=1; j<=k; j++) x[j] = y[j];
            }
    }
}
```

If the solution generated by the greedy method has all x_i's equal to zero or one, then it is also an optimal solution to 0/1 Knapsack problem. If not, exactly one x_i has the value between zero and one.

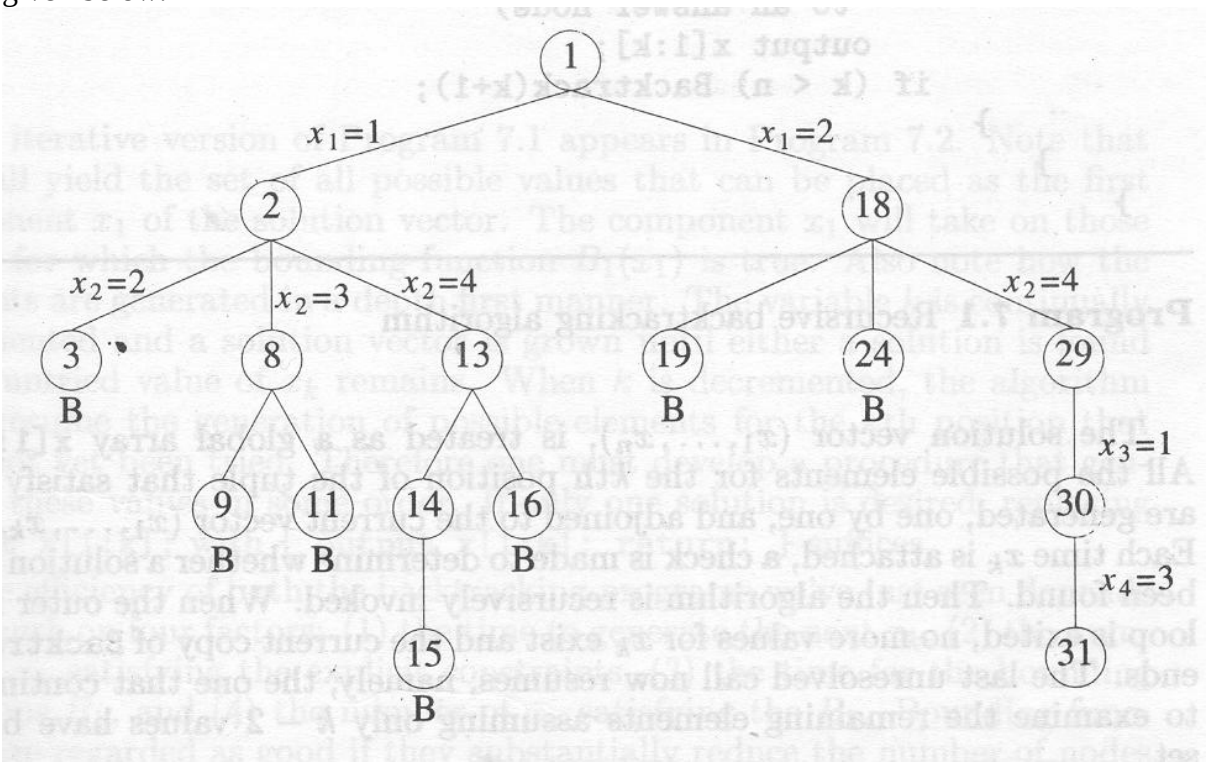
6. Branch and Bound

Branch and bound is a general algorithm for finding optimal solutions of various optimization problems. It consists of a systematic enumeration of all candidate solutions, where large subsets of fruitless candidates are discarded, by using upper and lower estimated bounds of the quantity being optimized. The method was first proposed by A. H. Land and A. G. Doig in 1960 for linear programming.

It is a counter-part of the backtracking search algorithm which, in the absence of a cost criteria, the algorithm traverses a spanning tree of the solution space using the breadth-first approach. It is a state space search method in which all children of E-node is generated before any other love node can become the E-node. Here, a BFS like state space search in which the live nodes are maintained using a queue is called **FIFO** and a D-Search like state space search in which the live nodes are maintained using a stack is called **LIFO**. As in the case of backtracking, bounding functions are used to avoid the generation of subtrees that do not contain an answer node.

6.1 Least Cost (LC) Search

Consider the portion of 4-Queens state space tree generated by FIFO branch and bound given below.



In both FIFO and LIFO branch and bound, selection rule for next E-node does not give preference to a node that has a very good chance of getting the search to an answer node quickly. The search can be speeded up by adding a ranking function for live nodes. Next E-node is selected on the basis of ranking function. Rank is created on the basis of

additional computational effort needed to reach an answer node from live node. For any node x, cost could be

- Number of nodes in the sub tree x that need to be generated before the answer node is generated
- Number of levels from x to the nearest answer node

The difficulty of using the above two process is that the computing the cost requires searching the subtree for answer node. So to find out the cost we have to explore the whole subtree.

A better method is to represent the cost as a function to two factors given by
c'(x) = f(h(x)) + g'(x)
where,
h(x) : cost of reaching x from root
f(.) : any decreasing function
g(x) : estimate of additional effort for reaching answer node from x.

Hence the next node to be selected will be the one with least c'(x) value. Hence it is called LC Search.
In c'(x), if g(x) = 0, f(h(x)) is the level of x, hence LC search generates nodes by level, or it transforms to BFS (FIFO).
In c'(x), if f(h(x))=0, and g'(x) >= g'(y) whenever y is a child of x, then LC search transforms to D-Search (LIFO).

Control Abstraction
Let t be the state space tree and c(.) be a cost function for nodes in t. If x is a node in t, c(x) is the minimum cost of any answer node in t with root x. Then c(t) is the minimum cost node in t. We use a heuristic function c'(t) to calculate c(t).
If x is an answer node or leaf node, then c(x) = c'(x).
Functions Least() and Add() deletes and adds from or to list of live nodes. Least() find nodes with least c'(x), delete the node from the list of live nodes and return the nodes. Live nodes are maintained as minheap. Path from answer node to t is printed.

```
struct listnode{
    strunct listnode *next, *parent;
    float cost;
}

LCSearch(struct listnode *t)
{
    struct listnode *x, *E;
    if (*t is an answer node)
            output *t and return;
    E = t;   //E-node
    Initialize list of live nodes to be empty;
    do
    {
            for (each child x of E)
            {
                    if x is an answer node, output the path from x to t and return;
                    Add(x);
```

```
            x->parent = E;
            }
            if there are more love nodes
            {
                  print("no answer nodes");
                  return;
            }
            E = Least();
      }
      while(1);
}
```

If the function Least() and Add() are Pop and Push function of a stack, then LC Search transforms to LIFO. If the function Least() and Add() are Delete and Insert function of a queue, then LC Search transforms to FIFO.

6.2 The 15-puzzle problem

This problem was invented by Sam Loyd in 1878. There are 15 numbered tiles placed on a square frame with capacity of 16 tiles. Given an initial arrangement and objective to transform any initial arrangement to a goal arrangement through a series of legal moves. A legal move is the one that move a tile adjacent to an empty space(ES) to ES. Each move creates a new arrangement of tile called *states* of the puzzle. Initial and final arrangements are called initial state and goal states respectively.

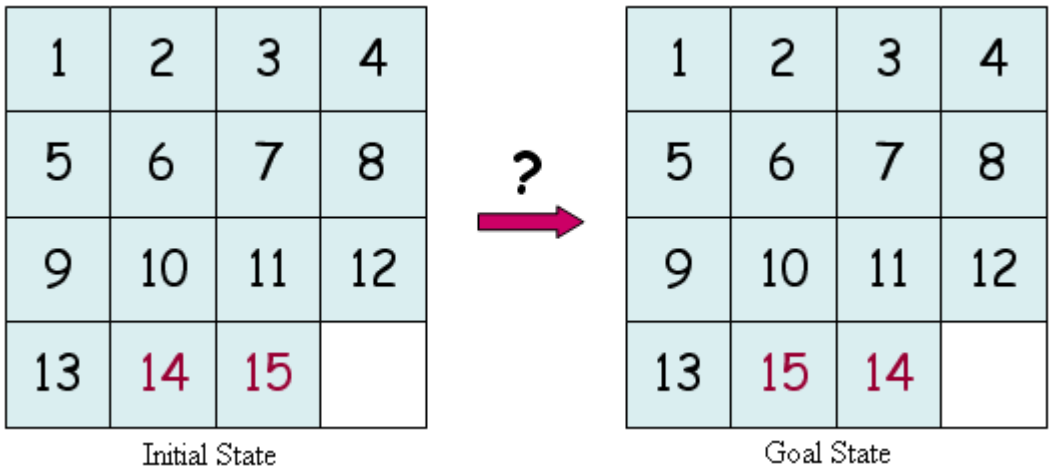
Given any initial state a goal state is reachable iff the value $\sum_{i=1}^{16} \text{less}(i) + r$ calculated for both the initial state and the goal state is of the same parity (i.e., either both odd or both even).

Here, less(i) is the number of tiles j such that j<i and position(j) > position(i).

r is the row number of empty state(ES)

Example 6.1

Check whether the goal state is achieved from the initial state given below.



Initial State

less(1) = 0	less(2) = 0	less(3) = 0	less(4) = 0	less(5) = 0
less(6) = 0	less(7) = 0	less(8) = 0	less(9) = 0	less(10) = 0
less(11) = 0	less(12) = 0	less(13) = 0	less(14) = 0	less(15) = 0

r = 4

$\sum = 15*0 + 4 = 4$

Goal State

less(1) = 0	less(2) = 0	less(3) = 0	less(4) = 0	less(5) = 0
less(6) = 0	less(7) = 0	less(8) = 0	less(9) = 0	less(10) = 0
less(11) = 0	less(12) = 0	less(13) = 0	less(14) = 0	less(15) = 1

r = 4

$\sum = 14*0 + 1 + 4 = 5$

Since the initial and the goal state are of different parity the goal state is not reachable from the initial state.

Example 6.2

Check whether the goal state is achieved from the initial state given below.

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

Initial State

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

Goal State

Initial State

less(1) = 0	less(2) = 0	less(3) = 0	less(4) = 0	less(5) = 0
less(6) = 0	less(7) = 0	less(8) = 1	less(9) = 0	less(10) = 0
less(11) = 0	less(12) = 0	less(13) = 1	less(14) = 1	less(15) = 1

r = 2

$\sum = 0+0+0+0+0+0+0+1+0+0+0+0+1+1+1+2 = 6$

Goal State

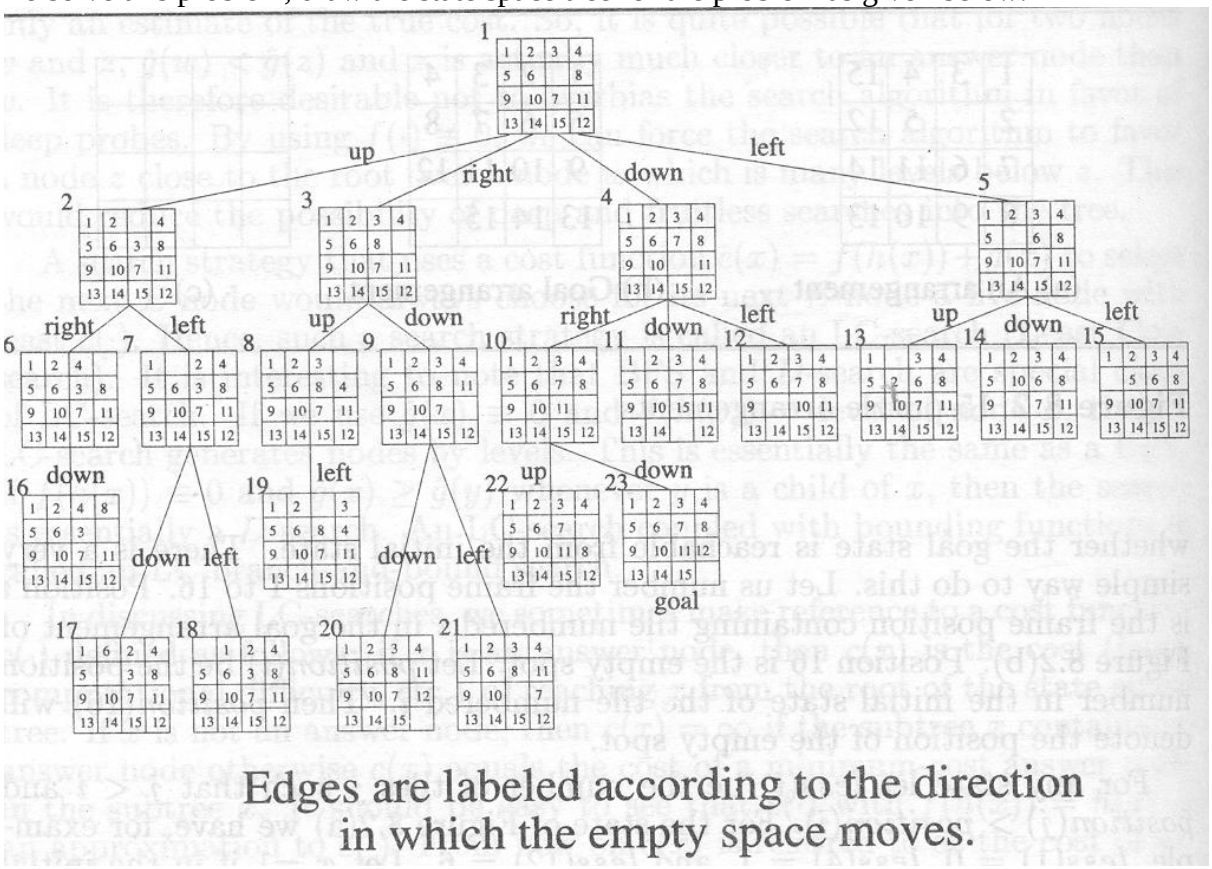
less(1) = 0	less(2) = 0	less(3) = 0	less(4) = 0	less(5) = 0
less(6) = 0	less(7) = 0	less(8) = 0	less(9) = 0	less(10) = 0
less(11) = 0	less(12) = 0	less(13) = 0	less(14) = 0	less(15) = 1

r = 4

$\sum = 15*0 + 4 = 4$

Since the initial and the goal state are even parity the goal state is reachable from the initial state.

To solve this problem, draw the state space tree for the problem as given below.



Here the children of the node x is the state reachable from x by one legal move. Consider the move as a move of empty space. Hence there are four possible moves, moving up, down, right and left. Here, no node p has a child state same as p's parent.

Here if we perform a breadth first search, the search will be blind. An intelligent move for search method will be one that seeks an answer node and adapt to the path after each move.

Instead of calculating c(x), calculate c'(x) as c'(x) = f(x) + g'(x).

f(x) : length of the path from root to x.

g'(x) : number of non blank tiles not in their goal position.

In the state space tree, node1 dies after leaving behind nodes 2, 3, 4 and 5.

Calculating c'(x) for these nodes, we have

c'(2) = 1+4 = 5

c'(3) = 1+4 = 5

c'(4) = 1+2 = 3

c'(5) = 1+4 = 5

Since c'(4) is the lowest value, next E-node is node 4. It will generates children 10, 11 and 12. Now the live nodes are 2,3,5,10,11 and 12.

c'(10) = 2+1 = 3

c'(11) = 2+3 = 5

c'(12) = 2+3 = 5

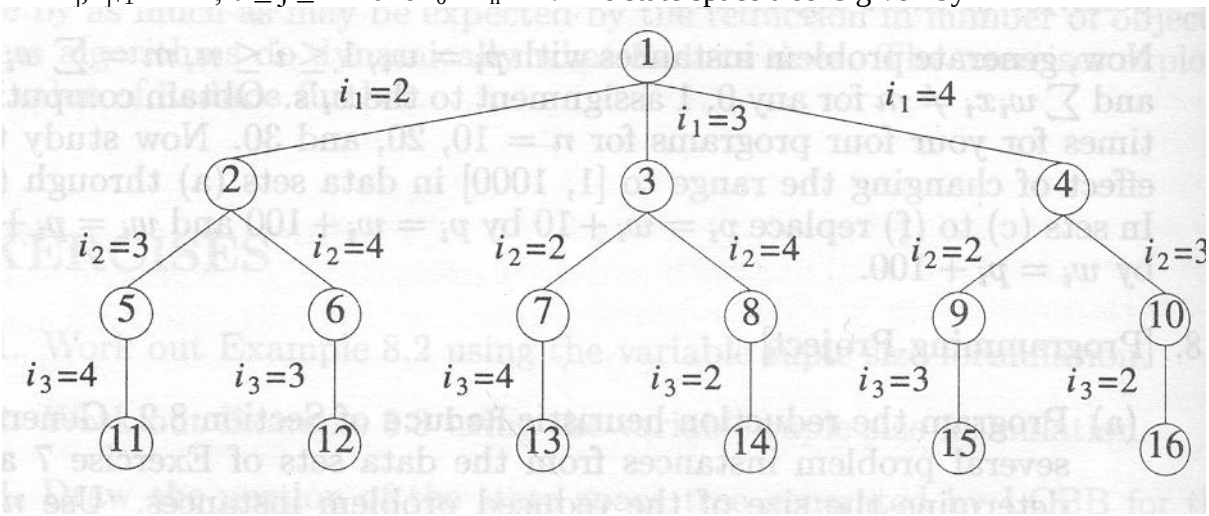
Now select node10. Continuing the same we get the next E-node as node 23 which is the answer node.

6.3 The traveling salesman problem

Problem: A salesman spends his time in visiting cities cyclically. In his tour he visits each city just once, and finishes up where he started. In which order should he visit to minimize the distance traveled.

The problem an be represented using a graph. Let G=(V,E) be a directed graph with cost c_{ij}, $c_{ij} > 0$ for all $\langle i\ j\rangle \in E$ and $c_{ij} = \infty$ for all $\langle j,t\rangle \notin E$. $|V| = n$ and $n>1$. We know that tour of a graph includes all vertices in V and cost of tour is the sum of the cost of all edges on the tour. Hence the traveling salesman problem is to minimize the cost.

The solution space is given by $S = \{1,S,1 \mid S \text{ is a permutation of } (2,3,\ldots,n)\}$.
Hence $|S| = (n-1)!$. Size of S can be reduced by restricting S so that $(1,i_1,\ldots i_{n-1}, 1) \in S$ iff $\langle i_j, i_{j+1}\rangle \in E$, $0 \le j \le n-1$ and $i_0 = i_n = 1$. The state space tree is given by



To use LCBB we need cost function c(.) such that $c'(r) \le c(r) \le u(r)$ for all nodes r.

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

Since both these methods are not practical, choose c'(x) as
c'(x) = reduced cost matrix for G.

A row or column in a given matrix is said to be reduced iff it contains at least one zero and all remaining entries as non-negative numbers. A matrix is said to be reduced if all rows and all columns are reduced.

Consider a graph with 5 vertices. Given the cost matrix of the graph. Since every tour in the graph includes one edge $\langle i, j\rangle$ with $1 \le I \le 5$ and exactly one edge $\langle i, j\rangle$ with $1 \le j \le 5$. We know that subtracting a common value from all the elements in a row or column will not change the minimum cost tour, even though it chances the minimum distance traveled.

Now consider the cost matrix of the graph G.

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

Reduced cost matrix of A is generated by

- Subtracting 10, 2, 2, 3 and 4 from rows 1, 2, 3, 4 and 5 respectively. Let the resultant matrix be A'

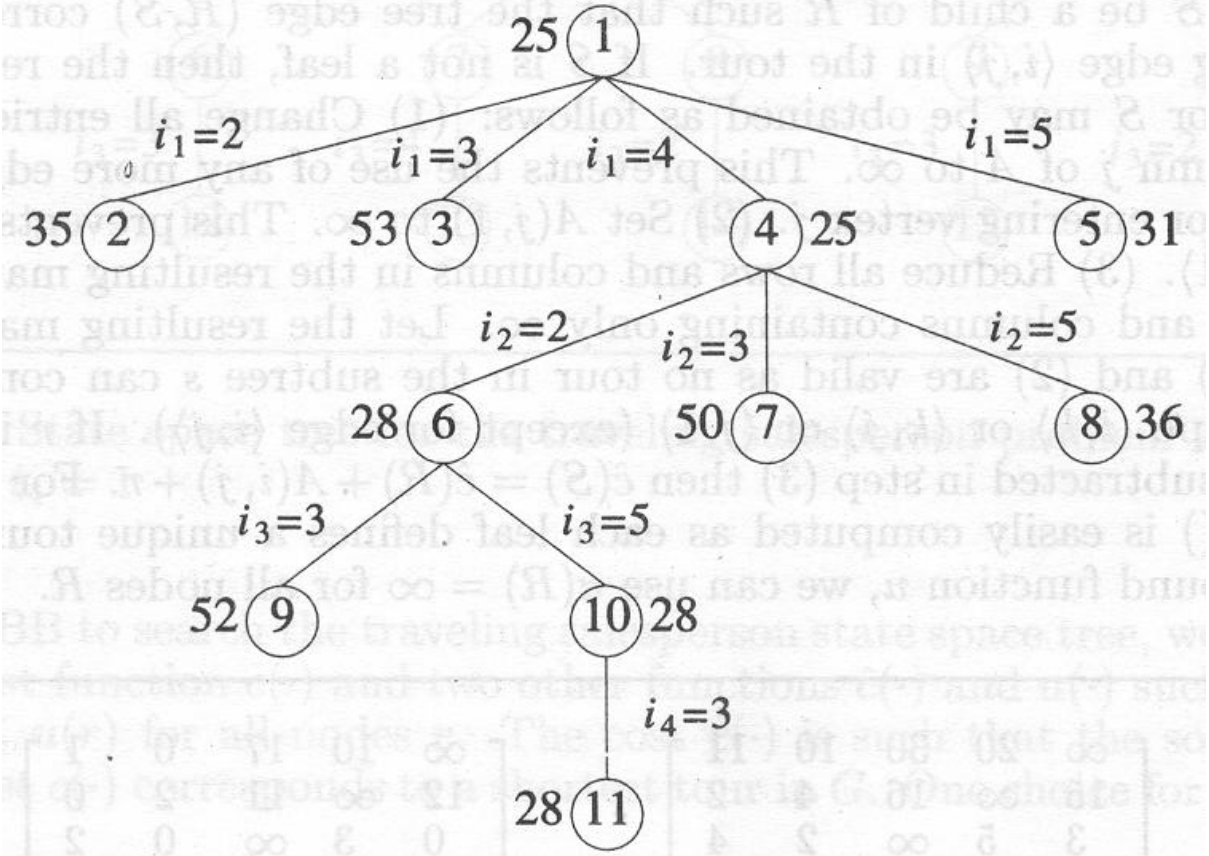
$$A' = \begin{bmatrix} \infty & 10 & 20 & 0 & 1 \\ 13 & \infty & 14 & 2 & 0 \\ 1 & 3 & \infty & 0 & 2 \\ 16 & 3 & 15 & \infty & 0 \\ 12 & 0 & 3 & 12 & \infty \end{bmatrix}$$

- And again subtract 1 and 3 from column 1 and column 3 respectively, we get the reduced cost matrix.

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

Find out the total value reduced from each row and column, which is 10+ 2+2+3+ 4+1+3 = 25.

Considering the state space tree, we are starting from node1, which means we started from vertex 1. Hence vertex 1 is the current E-node. Now generate children of node 1. Hence it generates nodes 2, 3 , 4 or 5 depending on whether we are selecting vertex 2, 3, 4 or 5. Calculate the reduced cost matrix for all the child nodes. The final state space tree will look like the one given below.



To calculate the reduced cost matrix of a child node C with parent P, if C corresponds to vertex j and P corresponds to vertex i, then reduced cost matrix is calcualted as per the following steps

- Change all entries in row i and column j to ∞ (Prevents the use of any more vertex leaving i or entering j)
- Set A(j,1) to ∞ (Prevents use of edge <j,1>)
- Reduce all rows and columns in the resulting matrix except the one with all ∞.

Let the resultant matrix be B. If r is the total amount subtracted in step 3,
c'(C) = c'(P) + A(i,j) + r.

Calculating the reduced cost matrix for node 2,3,4 and 5 we get

Path 1,2; node 2

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

C'(2) = 25 + 10 + 0 = 35

Path 1,3; node 3

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

C'(3) = 25+ 17 + 11 = 53

Path 1,4; node 4

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

C'(4) = 25+ 0 = 25

Path 1,5; node 5

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

C'(5) = 25 + 1 + 5 = 31

Since node 4 has got the least c'(.) value, select the next vertex as 4. Hence the current path is 1,4 and current E-node is node 4. Now generate the children of node 4. Then nodes 6, 7 and 8 are generated. Now the current live nodes are nodes 2,3,5,6,7 and8. Now find the c'(6), c'(7) and c'(9). Pick the next node as one with least c'(.) value.

Path 1,4,2; node 6

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

C'(6) = 28

Path 1,4,3; node 7

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

C'(7) = 50

Path 1,4,5; node 8

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

C'(8) = 36

Of the 6 live nodes, node 6 got the least C'(.) value, which means you selected vertex 2 as the next vertex. Hence the current path is 1,4,2. Now generate the children of node 6. So nodes 9 and 10 are generated. Calculate the C'() value of nodes 9 and 10.

Path 1,4,2,3; node 9

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

C'(9) = 52

Path 1,4,2,5; node 10

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

C'(10) = 28

Since node 9 has the least cost, select node 9, means select vertex 5. Now the path is 1,4,2,5. Since only one more vertex remaining select that as the next vertex. Hence the path is 1,4,2,5,3,1.