## BACKTRACKING -CO4

**General Method:**

Backtracking is used to solve problem in which a sequence of objects is chosen from a specified set so that the sequence satisfies some criterion. The desired solution is expressed as an n-tuple (x1, , xn) where each xi Є S, S being a finite set.

The solution is based on finding one or more vectors that maximize, minimize, or satisfy a criterion function P (x1, , xn). Form a solution and check at every step

if this has any chance of success. If the solution at any point seems not promising, ignore it. All solutions requires a set of constraints divided into two categories: explicit and implicit constraints.

Definition 1: Explicit constraints are rules that restrict each xi to take on values only from a given set. Explicit constraints depend on the particular instance I of problem being solved. All tuples that satisfy the explicit constraints define a possible solution space for I.

Definition 2: Implicit constraints are rules that determine which of the tuples in the solution space of I satisfy the criterion function. Thus, implicit constraints describe the way in which the xi’s must relate to each other.

* For 8-queens problem:

Explicit constraints using 8-tuple formation, for this problem are S= {1, 2, 3, 4, 5, 6, 7, 8}.

The implicit constraints for this problem are that no two queens can be the same (i.e., all queens must be on different columns) and no two queens can be on the same diagonal.

Backtracking is a modified depth first search of a tree. Backtracking algorithms determine problem solutions by systematically searching the solution space for the given problem instance. This search is facilitated by using a tree organization for the solution space.

Backtracking is the procedure where by, after determining that a node can lead to nothing but dead end, we go back (backtrack) to the nodes parent and proceed with the search on the next child.

A backtracking algorithm need not actually create a tree. Rather, it only needs to keep track of the values in the current branch being investigated. This is the way we implement backtracking algorithm. We say that the state space tree exists implicitly in the algorithm because it is not actually constructed.

## Terminology:

***Problem state*** is each node in the depth first search tree.

***Solution states*** are the problem states ‘S’ for which the path from the root node to

‘S’ defines a tuple in the solution space.

***Answer states*** are those solution states for which the path from root node to s defines a tuple that is a member of the set of solutions.

***State space*** is the set of paths from root node to other nodes. *State space* tree is the tree organization of the solution space. The state space trees are called static trees. This terminology follows from the observation that the tree organizations are independent of the problem instance being solved. For some problems it is advantageous to use different tree organizations for different problem instance. In this case the tree organization is determined dynamically as the solution space is being searched. Tree organizations that are problem instance dependent are called dynamic trees.

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

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

***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.

***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.

Depth first node generation with bounding functions is called ***backtracking***. State generation methods in which the E-node remains the E-node until it is dead, lead to branch and bound methods.

## Planar Graphs:

When drawing a graph on a piece of a paper, we often find it convenient to permit edges to intersect at points other than at vertices of the graph. These points of interactions are called crossovers.

A graph G is said to be planar if it can be drawn on a plane without any crossovers; otherwise G is said to be non-planar i.e., A graph is said to be planar iff it can be drawn in a plane in such a way that no two edges cross each other.

## N-Queens Problem:

Let us consider, N = 8. Then 8-Queens Problem is to place eight queens on an 8 x 8 chessboard so that no two “attack”, that is, no two of them are on the same row, column, or diagonal.

All solutions to the 8-queens problem can be represented as 8-tuples (x1, . . . . , x8), where xi is the column of the ith row where the ith queen is placed.

The explicit constraints using this formulation are Si = {1, 2, 3, 4, 5, 6, 7, 8}, 1 < i <

1. Therefore the solution space consists of 88 8-tuples.

The implicit constraints for this problem are that no two xi’s can be the same (i.e., all queens must be on different columns) and no two queens can be on the same diagonal.

This realization reduces the size of the solution space from 88 tuples to 8! Tuples.

The promising function must check whether two queens are in the same column or

diagonal:

Suppose two queens are placed at positions (i, j) and (k, l) Then:

* + Column Conflicts: Two queens conflict if their xi values are identical.
  + Diag 45 conflict: Two queens i and j are on the same 450 diagonal if:

i – j = k – l.

This implies, j – l = i – k

* + Diag 135 conflict:

i + j = k + l.

This implies, j – l = k – i

Therefore, two queens lie on the same diagonal if and only if:

j - l = i – k 

Where, j be the column of object in row i for the ith queen and l be the column of object in row ‘k’ for the kth queen.

To check the diagonal clashes, let us take the following tile configuration:

In this example, we have:

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| i | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 |
| xi | 2 | 5 | 1 | 8 | 4 | 7 | 3 | 6 |

Let us consider for the case whether the queens on 3rd row and 8th row are conflicting or not. In this

case (i, j) = (3, 1) and (k, l) = (8, 6). Therefore:

j - l = i – k   1 - 6 = 3 – 8 

 5 = 5

In the above example we have, j - l = i – k , so the two queens are attacking. This is not a solution.

## Example:

Suppose we start with the feasible sequence 7, 5, 3, 1.

Step 1:

Add to the sequence the next number in the sequence 1, 2, . . . , 8 not yet used.

Step 2:

If this new sequence is feasible and has length 8 then STOP with a solution. If the new sequence is feasible and has length less then 8, repeat Step 1.

Step 3:

If the sequence is not feasible, then *backtrack* through the sequence until we find the *most recent* place at which we can exchange a value. Go back to Step 1.

On a chessboard, the **solution** will look like:

## 4 – Queens Problem:

Let us see how backtracking works on the 4-queens problem. We start with the root node as the only live node. This becomes the E-node. We generate one child. Let us assume that the children are generated in ascending order. Let us assume that the children are generated in ascending order. Thus node number 2 of figure is generated and the path is now (1). This corresponds to placing queen 1 on column 1. Node 2 becomes the E-node. Node 3 is generated and immediately killed. The next node generated is node 8 and the path becomes (1, 3). Node 8 becomes the E-node. However, it gets killed as all its children represent board configurations that cannot lead to an answer node. We backtrack to node 2 and generate another child, node 13. The path is now (1, 4). The board configurations as backtracking proceeds is as follows:

|  |  |  |  |
| --- | --- | --- | --- |
| 1 |  |  |  |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |

|  |  |  |  |
| --- | --- | --- | --- |
| 1 |  |  |  |
| **.** | **.** | 2 |  |
|  |  |  |  |
|  |  |  |  |

|  |  |  |  |
| --- | --- | --- | --- |
| 1 |  |  |  |
|  |  | 2 |  |
| **.** | **.** | **.** |  |
|  |  |  |  |

|  |  |  |  |
| --- | --- | --- | --- |
| 1 |  |  |  |
|  |  |  | 2 |
| **.** | 3 |  |  |
|  |  |  |  |

(a) (b) (c) (d)

|  |  |  |  |
| --- | --- | --- | --- |
| 1 |  |  |  |
|  |  |  | 2 |
|  | 3 |  |  |
| **.** | **.** | **.** | **.** |

|  |  |  |  |
| --- | --- | --- | --- |
|  | 1 |  |  |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |

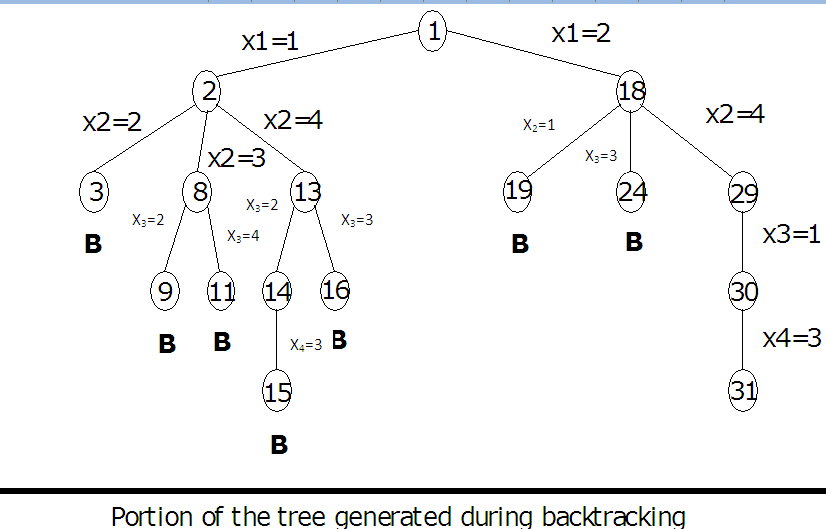
|  |  |  |  |
| --- | --- | --- | --- |
|  | 1 |  |  |
| **.** | **.** | **.** | 2 |
|  |  |  |  |
|  |  |  |  |

|  |  |  |  |
| --- | --- | --- | --- |
|  | 1 |  |  |
|  |  |  | 2 |
| 3 |  |  |  |
| **.** | **.** | 4 |  |

(e) (f) (g) (h)

The above figure shows graphically the steps that the backtracking algorithm goes through as it tries to find a solution. The dots indicate placements of a queen, which were tried and rejected because another queen was attacking.

In Figure (b) the second queen is placed on columns 1 and 2 and finally settles on column 3. In figure (c) the algorithm tries all four columns and is unable to place the next queen on a square. Backtracking now takes place. In figure (d) the second queen is moved to the next possible column, column 4 and the third queen is placed on column 2. The boards in Figure (e), (f), (g), and (h) show the remaining steps that the algorithm goes through until a solution is found.



## Sum of Subsets-CO4

Given positive numbers wi, 1 ≤ i ≤ n, and m, this problem requires finding all subsets

of wi whose sums are ‘m’.

All solutions are k-tuples, 1 ≤ k ≤ n.

Explicit constraints:

* xi **Є** {j | j is an integer and 1 ≤ j ≤ n}.

Implicit constraints:

* No two xi can be the same.
* The sum of the corresponding wi’s be m.
* xi < xi+1 , 1 ≤ i < k (total order in indices) to avoid generating multiple instances of the same subset (for example, (1, 2, 4) and (1, 4, 2) represent the same subset).

A better formulation of the problem is where the solution subset is represented by an n-tuple (x1, , xn) such that xi **Є** {0, 1}.

The above solutions are then represented by (1, 1, 0, 1) and (0, 0, 1, 1). For both the above formulations, the solution space is 2n distinct tuples.

For example, n = 4, w = (11, 13, 24, 7) and m = 31, the desired subsets are (11,

13, 7) and (24, 7).

The tree corresponds to the variable tuple size formulation. The edges are labeled such that an edge from a level i node to a level i+1 node represents a value for xi. At each node, the solution space is partitioned into sub - solution spaces. All paths from the root node to any node in the tree define the solution space, since any such path corresponds to a subset satisfying the explicit constraints.

The possible paths are (1), (1, 2), (1, 2, 3), (1, 2, 3, 4), (1, 2, 4), (1, 3, 4), (2), (2,

3), and so on. Thus, the left mot sub-tree defines all subsets containing w1, the next sub-tree defines all subsets containing w2 but not w1, and so on.

## Graph Coloring (for planar graphs):

Let G be a graph and m be a given positive integer. We want to discover whether the nodes of G can be colored in such a way that no two adjacent nodes have the same color, yet only m colors are used. This is termed the m-colorabiltiy decision problem. The m-colorability optimization problem asks for the smallest integer m for which the graph G can be colored.

Given any map, if the regions are to be colored in such a way that no two adjacent regions have the same color, only four colors are needed.

For many years it was known that five colors were sufficient to color any map, but no map that required more than four colors had ever been found. After several hundred years, this problem was solved by a group of mathematicians with the help of a computer. They showed that in fact four colors are sufficient for planar graphs.

The function m-coloring will begin by first assigning the graph to its adjacency matrix, setting the array x [] to zero. The colors are represented by the integers 1, 2, . . . , m and the solutions are given by the n-tuple (x1, x2, . . ., xn), where xi is the color of node i.

A recursive backtracking algorithm for graph coloring is carried out by invoking the statement mcoloring(1);

**Algorithm mcoloring** (k)

// This algorithm was formed using the recursive backtracking schema. The graph is

// represented by its Boolean adjacency matrix G [1: n, 1: n]. All assignments of

// 1, 2, , m to the vertices of the graph such that adjacent vertices are assigned

// distinct integers are printed. k is the index of the next vertex to color.

{

repeat

{ // Generate all legal assignments for x[k].

NextValue (k); // Assign to x [k] a legal color. If (x [k] = 0) then return; // No new color possible If (k = n) then // at most m colors have been

// used to color the n vertices.

write (x [1: n]);

else mcoloring (k+1);

} until (false);

}

**Algorithm NextValue** (k)

// x [1] , x [k-1] have been assigned integer values in the range [1, m] such that

// adjacent vertices have distinct integers. A value for x [k] is determined in the range

// [0, m].x[k] is assigned the next highest numbered color while maintaining distinctness

// from the adjacent vertices of vertex k. If no such color exists, then x [k] is 0.

{

repeat

{

x [k]: = (x [k] +1) mod (m+1) // Next highest color.

If (x [k] = 0) then return; // All colors have been used for j := 1 to n do

{ // check if this color is distinct from adjacent colors if ((G [k, j]  0) and (x [k] = x [j]))

// If (k, j) is and edge and if adj. vertices have the same color. then break;

}

if (j = n+1) then return; // New color found

} until (false); // Otherwise try to find another color.

}

## Hamiltonian Cycles:

Let G = (V, E) be a connected graph with n vertices. A Hamiltonian cycle (suggested by William Hamilton) is a round-trip path along n edges of G that visits every vertex once and returns to its starting position. In other vertices of G are visited in the order v1, v2, . . . . . , vn+1, then the edges (vi, vi+1) are in E, 1 < i < n, and the vi are distinct expect for v1 and vn+1, which are equal. The graph G1 contains the Hamiltonian cycle 1, 2, 8, 7, 6, 5, 4, 3, 1. The graph G2 contains no Hamiltonian cycle.



Graph G1 Graph G2

Two graphs to illustrate Hamiltonian cycle

The backtracking solution vector (x1, xn) is defined so that xi represents the ith

visited vertex of the proposed cycle. If k = 1, then x1 can be any of the n vertices. To avoid printing the same cycle n times, we require that x1 = 1. If 1 < k < n, then xk can be any vertex v that is distinct from x1, x2, . . . , xk–1 and v is connected by an edge to kx-1. The vertex xn can only be one remaining vertex and it must be connected to both xn-1 and x1.

Using NextValue algorithm we can particularize the recursive backtracking schema to find all Hamiltonian cycles. This algorithm is started by first initializing the adjacency matrix G[1: n, 1: n], then setting x[2: n] to zero and x[1] to 1, and then executing Hamiltonian(2).

The traveling salesperson problem using dynamic programming asked for a tour that has minimum cost. This tour is a Hamiltonian cycles. For the simple case of a graph all of whose edge costs are identical, Hamiltonian will find a minimum-cost tour if a tour exists.

**Algorithm NextValue** (k)

// x [1: k-1] is a path of k – 1 distinct vertices . If x[k] = 0, then no vertex has as yet been

// assigned to x [k]. After execution, x[k] is assigned to the next highest numbered vertex

// which does not already appear in x [1 : k – 1] and is connected by an edge to x [k – 1].

// Otherwise x [k] = 0. If k = n, then in addition x [k] is connected to x [1].

{

repeat

{

x [k] := (x [k] +1) mod (n+1); // Next vertex. If (x [k] = 0) then return;

If (G [x [k – 1], x [k]]  0) then

{ // Is there an edge?

for j := 1 to k – 1 do if (x [j] = x [k]) then break;

// check for distinctness.

If (j = k) then // If true, then the vertex is distinct. If ((k < n) or ((k = n) and G [x [n], x [1]]  0))

then return;

}

} until (false);

}

**Algorithm Hamiltonian** (k)

// This algorithm uses the recursive formulation of backtracking to find all the Hamiltonian

// cycles of a graph. The graph is stored as an adjacency matrix G [1: n, 1: n]. All cycles begin

// at node 1.

{

repeat

{ // Generate values for x [k].

NextValue (k); //Assign a legal Next value to x [k]. if (x [k] = 0) then return;

if (k = n) then write (x [1: n]); else Hamiltonian (k + 1)

} until (false);

}

## Branch and Bound General method: CO5

Branch and Bound is another method to systematically search a solution space. Just like backtracking, we will use bounding functions to avoid generating subtrees that do not contain an answer node. However branch and Bound differs from backtracking in two important manners:

1. It has a branching function, which can be a depth first search, breadth first search or based on bounding function.
2. It has a bounding function, which goes far beyond the feasibility test as a mean to prune efficiently the search tree.

Branch and Bound refers to all state space search methods in which all children of the E-node are generated before any other live node becomes the E-node

Branch and Bound is the generalization of both graph search strategies, BFS and D- search.

* + A BFS like state space search is called as FIFO (First in first out) search as the list of live nodes in a first in first out list (or queue).
  + A D search like state space search is called as LIFO (Last in first out) search as the list of live nodes in a last in first out (or stack).

*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-an-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.

*Definition 5:* The adjective "heuristic", means" related to improving problem solving performance". As a noun it is also used in regard to "any method or trick used to improve the efficiency of a problem solving problem". But imperfect methods are not necessarily heuristic or vice versa. "A heuristic (heuristic rule, heuristic method) is a rule of thumb, strategy, trick simplification or any other kind of device which drastically limits search for solutions in large problem spaces. Heuristics do not guarantee optimal solutions, they do not guarantee any solution at all. A useful heuristic offers solutions which are good enough most of the time.

173

## Least Cost (LC) search:

In both LIFO and FIFO Branch and Bound the selection rule for the next E-node in rigid and blind. The selection rule for the next E-node does not give any preference to a node that has a very good chance of getting the search to an answer node quickly.

The search for an answer node can be speeded by using an “intelligent” ranking function *c*( ) for live nodes. The next E-node is selected on the basis of this ranking function. The node x is assigned a rank using:

*c*( *x* ) = f(h(x)) + *g*( *x* ) where, *c*( *x* ) is the cost of x.

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

*g* ( *x* ) is an estimate of the additional effort needed to reach an answer node from x.

A search strategy that uses a cost function *c*( *x* ) = f(h(x) + *g*( *x* ) to select the next

E-node would always choose for its next E-node a live node with least LC–search (Least Cost search)

*c*(.) is called a

BFS and D-search are special cases of LC-search. If *g*( *x* ) = 0 and f(h(x)) = level of node x, then an LC search generates nodes by levels. This is eventually the same as

a BFS. If f(h(x)) = 0 and essentially a D-search.

*g*( *x* ) > *g*( *y* ) whenever y is a child of x, then the search is

An LC-search coupled with bounding functions is called an LC-branch and bound search

We associate a cost c(x) with each node x in the state space tree. It is not possible to easily compute the function c(x). So we compute a estimate *c*( *x* ) of c(x).

## Control Abstraction for LC-Search:

Let t be a state space tree and c() a cost function for the nodes in t. If x is a node in t, then c(x) is the minimum cost of any answer node in the subtree with root x. Thus, c(t) is the cost of a minimum-cost answer node in t.

A heuristic *c*(.) is used to estimate c(). This heuristic should be easy to compute and generally has the property that if x is either an answer node or a leaf node, then

c(x) = *c*( *x* ) .

LC-search uses *c* to find an answer node. The algorithm uses two functions Least() and Add() to delete and add a live node from or to the list of live nodes, respectively.

Least() finds a live node with least c(). This node is deleted from the list of live nodes and returned.

174

Add(x) adds the new live node x to the list of live nodes. The list of live nodes be implemented as a min-heap.

Algorithm LCSearch outputs the path from the answer node it finds to the root node

t. This is easy to do if with each node x that becomes live, we associate a field *parent* which gives the parent of node x. When the answer node g is found, the path from g to t can be determined by following a sequence of *parent* values starting from the current E-node (which is the parent of g) and ending at node t.

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);

}

The root node is the first, E-node. During the execution of LC search, this list contains all live nodes except the E-node. Initially this list should be empty. Examine all the children of the E-node, if one of the children is an answer node, then the algorithm outputs the path from x to t and terminates. If the child of E is not an answer node, then it becomes a live node. It is added to the list of live nodes and its parent field set to E. When all the children of E have been generated, E becomes a dead node. This happens only if none of E’s children is an answer node. Continue the search further until no live nodes found. Otherwise, Least(), by definition, correctly chooses the next E-node and the search continues from here.

LC search terminates only when either an answer node is found or the entire state space tree has been generated and searched.

## Bounding:

A branch and bound method searches a state space tree using any search mechanism in which all the children of the E-node are generated before another node becomes the E-node. We assume that each answer node x has a cost c(x) associated with it and that a minimum-cost answer node is to be found. Three common search strategies are FIFO, LIFO, and LC. The three search methods differ only in the selection rule used to obtain the next E-node.

175

A good bounding helps to prune efficiently the tree, leading to a faster exploration of the solution space.

A cost function *c*(.) such that *c*( *x* ) < c(x) is used to provide lower bounds on solutions obtainable from any node x. If upper is an upper bound on the cost of a minimum-cost solution, then all live nodes x with c(x) > *c*( *x* ) > upper. The starting value for upper can be obtained by some heuristic or can be set to  .

As long as the initial value for upper is not less than the cost of a minimum-cost answer node, the above rules to kill live nodes will not result in the killing of a live node that can reach a minimum-cost answer node. Each time a new answer node is found, the value of upper can be updated.

Branch-and-bound algorithms are used for optimization problems where, we deal directly only with minimization problems. A maximization problem is easily converted to a minimization problem by changing the sign of the objective function.

To formulate the search for an optimal solution for a least-cost answer node in a state space tree, it is necessary to define the cost function c(.), such that c(x) is minimum for all nodes representing an optimal solution. The easiest way to do this is to use the objective function itself for c(.).

* For nodes representing feasible solutions, c(x) is the value of the objective function for that feasible solution.
* For nodes representing infeasible solutions, c(x) = .
* For nodes representing partial solutions, c(x) is the cost of the minimum-cost node in the subtree with root x.

Since, c(x) is generally hard to compute, the branch-and-bound algorithm will use an estimate *c*( *x* ) such that *c*( *x* ) < c(x) for all x.

## FIFO Branch and Bound:CO5

A FIFO branch-and-bound algorithm for the job sequencing problem can begin with upper =  as an upper bound on the cost of a minimum-cost answer node.

Starting with node 1 as the E-node and using the variable tuple size formulation of Figure 8.4, nodes 2, 3, 4, and 5 are generated. Then u(2) = 19, u(3) = 14, u(4) =

18, and u(5) = 21.

The variable upper is updated to 14 when node 3 is generated. Since *c* (4) and *c*(5) are greater than upper, nodes 4 and 5 get killed. Only nodes 2 and 3 remain alive.

Node 2 becomes the next E-node. Its children, nodes 6, 7 and 8 are generated.

Then u(6) = 9 and so upper is updated to 9. The cost gets killed. Node 8 is infeasible and so it is killed.

*c*(7) = 10 > upper and node 7

Next, node 3 becomes the E-node. Nodes 9 and 10 are now generated. Then u(9) = 8 and so upper becomes 8. The cost *c*(10) = 11 > upper, and this node is killed.

The next E-node is node 6. Both its children are infeasible. Node 9’s only child is also infeasible. The minimum-cost answer node is node 9. It has a cost of 8.

When implementing a FIFO branch-and-bound algorithm, it is not economical to kill

live nodes with *c*(*x*) > upper each time upper is updated. This is so because live nodes are in the queue in the order in which they were generated. Hence, nodes with *c*(*x*) > upper are distributed in some random way in the queue. Instead, live nodes

with *c*(*x*) > upper can be killed when they are about to become E-nodes.

The FIFO-based branch-and-bound algorithm with an appropriate called FIFOBB.

*c*(.) and u(.) is

## LC Branch and Bound:

An LC Branch-and-Bound search of the tree of Figure 8.4 will begin with upper = 

and node 1 as the first E-node.

When node 1 is expanded, nodes 2, 3, 4 and 5 are generated in that order.

As in the case of FIFOBB, upper is updated to 14 when node 3 is generated and nodes 4 and 5 are killed as *c*(4) > upper and *c*(5) > upper.

Node 2 is the next E-node as *c*(2) = 0 and *c*(3) = 5. Nodes 6, 7 and 8 are generated and upper is updated to 9 when node 6 is generated. So, node 7 is killed as *c*(7) = 10

> upper. Node 8 is infeasible and so killed. The only live nodes now are nodes 3 and

6.

Node 6 is the next E-node as *c*(6) = 0 < *c*(3) . Both its children are infeasible.

Node 3 becomes the next E-node. When node 9 is generated, upper is updated to 8 as u(9) = 8. So, node 10 with *c*(10) = 11 is killed on generation.

Node 9 becomes the next E-node. Its only child is infeasible. No live nodes remain. The search terminates with node 9 representing the minimum-cost answernode.

2 3

The path = 1  3  9 = 5 + 3 = 8

## Traveling Sale Person Problem:

By using dynamic programming algorithm we can solve the problem with time complexity of O(n22n) for worst case. This can be solved by branch and bound technique using efficient bounding function. The time complexity of traveling sale person problem using LC branch and bound is O(n22n) which shows that there is no change or reduction of complexity than previous method.

We start at a particular node and visit all nodes exactly once and come back to initial node with minimum cost.

Let G = (V, E) is a connected graph. Let C(i, J) be the cost of edge <i, j>. cij =  if

<i, j> E and let |V| = n, the number of vertices. Every tour starts at vertex 1 and ends at the same vertex. So, the solution space is given by S = {1, , 1 |  is a

permutation of (2, 3, . . . , n)} and |S| = (n – 1)!. The size of S can be reduced by restricting S so that (1, i1, i2, . . . . in-1, 1)  S iff <ij, ij+1>  E, 0 < j < n - 1 and i0

= in =1.

Procedure for solving traveling sale person problem:

1. Reduce the given cost matrix. A matrix is reduced if every row and column is reduced. A row (column) is said to be reduced if it contain at least one zero and all-remaining entries are non-negative. This can be done as follows:
   1. *Row reduction:* Take the minimum element from first row, subtract it from all elements of first row, next take minimum element from the second row and subtract it from second row. Similarly apply the same procedure for all rows.
   2. Find the sum of elements, which were subtracted from rows.
   3. Apply column reductions for the matrix obtained after row reduction.

*Column reduction:* Take the minimum element from first column, subtract it from all elements of first column, next take minimum element from the second column and subtract it from second column. Similarly apply the same procedure for all columns.

* 1. Find the sum of elements, which were subtracted from columns.
  2. Obtain the cumulative sum of row wise reduction and column wise reduction.

Cumulative reduced sum = Row wise reduction sum + column wise reduction sum.

Associate the cumulative reduced sum to the starting state as lower bound and  as upper bound.

1. Calculate the reduced cost matrix for every node R. Let A is 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 <i, j> in the tour. If S is not a leaf node, 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 .
   2. Set A (j, 1) to .
   3. Reduce all rows and columns in the resulting matrix except for rows and column containing only . Let r is the total amount subtracted to reduce the matrix.

c) Find *c**S*  *c**R*  *A* *i*, *j*  *r*, where ‘r’ is the total amount subtracted to reduce the matrix, *c**R* indicates the lower bound of the ith node in (i, j) path and *c* *S*  is called the cost function.

1. Repeat step 2 until all nodes are visited.

## 0/1 Knapsack Problem: CO5

Consider the instance: M = 15, n = 4, (P1, P2, P3, P4) = (10, 10, 12, 18) and

(w1, w2, w3, w4) = ( 2, 4, 6, 9).

0/1 knapsack problem can be solved by using branch and bound technique. In this problem we will calculate lower bound and upper bound for each node.

Place first item in knapsack. Remaining weight of knapsack is 15 – 2 = 13. Place next item w2 in knapsack and the remaining weight of knapsack is 13 – 4 = 9. Place next item w3 in knapsack then the remaining weight of knapsack is 9 – 6 = 3. No fractions are allowed in calculation of upper bound so w4 cannot be placed in knapsack.

Profit = P1 + P2 + P3 = 10 + 10 + 12

So, Upper bound = 32

To calculate lower bound we can place w4 in knapsack since fractions are allowed in

calculation of lower bound.

Lower bound = 10 + 10 + 12 + (

3 *X* 18) = 32 + 6 = 38

9

Knapsack problem is maximization problem but branch and bound technique is applicable for only minimization problems. In order to convert maximization problem into minimization problem we have to take negative sign for upper bound and lower bound.

## NP Hard and NP-Complete: CO 5 Basic concepts:

Nondeterministic Polynomial time

The problems has best algorithms for their solutions have “Computing times”, that cluster into two groups

|  |  |
| --- | --- |
| **Group 1** | **Group 2** |
| * Problems with solution time bound by a polynomial of a small degree. | * Problems with solution times not bound by polynomial (simply non polynomial ) |
| * It also called “Tractable Algorithms” * Most Searching & Sorting algorithms are polynomial time algorithms * **Ex:** Ordered Search **(O (log n)),** | * These are hard or intractable problems * None of the problems in this group has been solved by any polynomial time algorithm |
| Polynomial evaluation **O(n)** Sorting | * **Ex:** Traveling Sales Person **O(n2** |
| **O(n.log n)** | **2n)** Knapsack **O(2n/2)** |

No one has been able to develop a polynomial time algorithm for any problem in the 2nd group (i.e., group 2)

So, it is compulsory and finding algorithms whose computing times are greater than polynomial very quickly because such vast amounts of time to execute that even moderate size problems cannot be solved.

## Theory of NP-Completeness:

Show that may of the problems with no polynomial time algorithms are computational time algorithms are computationally related.

There are two classes of non-polynomial time problems

1. NP-Hard
2. NP-Complete

**NP Complete Problem:** A problem that is NP-Complete can solved in polynomial time if and only if (iff) all other NP-Complete problems can also be solved in polynomial time.

**NP-Hard:** Problem can be solved in polynomial time then all NP-Complete problems can be solved in polynomial time.

All NP-Complete problems are NP-Hard but some NP-Hard problems are not know to be NP- Complete.

## Nondeterministic Algorithms:

Algorithms with the property that the result of every operation is uniquely defined are termed as deterministic algorithms. Such algorithms agree with the way programs are executed on a computer.

Algorithms which contain operations whose outcomes are not uniquely defined but are limited to specified set of possibilities. Such algorithms are called nondeterministic algorithms.

The machine executing such operations is allowed to choose any one of these outcomes subject to a termination condition to be defined later.

To specify nondeterministic algorithms, there are 3 new



sful completion



## Example for Non Deterministic algorithms:

|  |  |
| --- | --- |
| **Algorithm Search(x){** | Whenever there is a set of choices |
| //Problem is to search an element x | that leads to a successful  completion then one such set of |
| //output J, such that A[J]=x; or J=0 if x is not | choices is always made and the |
| in A J:=Choice(1,n); | algorithm terminates. |
| if( A[J]:=x) then { | A Nondeterministic algorithm |
| Write(J); | terminates unsuccessfully if and |
| Success()  ; | only if (iff) there exists no set of  choices leading to a successful signal. |
| } |  |
| else{ |  |
| write(0) |  |
| ; |  |
| failure() |  |
| ; |  |
| } |  |

|  |  |
| --- | --- |
| **Nondeterministic Knapsack algorithm** | |
| **Algorithm DKP**(p, w, n, m, r, x){ |  |
| W:=0; |  |
| P:=0; |  |
| for i:=1 to n do{ | p or w) |
| x[i]:=choice(0, 1); |  |
| W:=W+x[i]\*w[i]; |  |
| P:=P+x[i]\*p[i]; |  |
| } |  |
| if( (W>m) or (P<r) ) then Failure(); |  |
| else Success(); |  |
| } |  |

**The Classes NP-Hard & NP-Complete:**

For measuring the complexity of an algorithm, we use the input length as the parameter. For example, An algorithm A is of polynomial complexity p() such that the computing time of A is O(p(n)) for every input of size n.

**Decision problem/ Decision algorithm:** Any problem for which the answer is either zero or one is decision problem. Any algorithm for a decision problem is termed a decision algorithm.

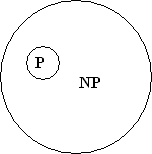
**Optimization problem/ Optimization algorithm:** Any problem that involves the identification of an optimal (either minimum or maximum) value of a given cost function is known as an optimization problem. An optimization algorithm is used to solve an optimization problem.

## P

polynomial time.

is the set of all decision problems solvable by nondeterministic algorithms in polynomial time.

Since deterministic algorithms are just a special case of nondeterministic, by this we concluded that **P** ⊆ **NP**



Commonly believed relationship between P & NP

The most famous unsolvable problems in Computer Science is Whether P=NP or

P≠NP In considering this problem, s.cook formulated the following question.

If there any single problem in NP, such that if we showed it to be in ‘P’ then that would imply that P=NP.

Cook answered this question with

**Theorem**: Satisfiability is in P if and only if (iff) P=NP



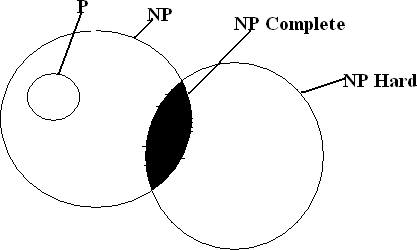
Let L1 and L2 be problems, Problem L1 reduces to L2 (written **L1 α L2**) iff there is a way to solve L1 by a deterministic polynomial time algorithm using a deterministic algorithm that solves L2 in polynomial time

This implies that, if we have a polynomial time algorithm for L2, Then we can solve L1 in polynomial time.

**L1 α L2** and **L2 α L3** then **L1 α L3**

A problem L is NP-Hard if and only if (iff) satisfiability reduces to L ie., **Statisfiability α L**

A problem L is NP-Complete if and only if (iff) L is NP-Hard and **L Є NP**



Commonly believed relationship among P, NP, NP-Complete and NP-Hard

Most natural problems in NP are either in P or NP-complete.

## Examples of NP-complete problems:

* Packing problems: SET-PACKING, INDEPENDENT-SET.
* Covering problems: SET-COVER, VERTEX-COVER.
* Sequencing problems: HAMILTONIAN-CYCLE, TSP.
* Partitioning problems: 3-COLOR, CLIQUE.
* Constraint satisfaction problems: SAT, 3-SAT.
* Numerical problems: SUBSET-SUM, PARTITION, KNAPSACK.

**Cook’s Theorem:** States that satisfiability is in P if and only if P=NP If P=NP then satisfiability is in P

If satisfiability is in P, then P=NP To do this

 

algorithm

Then formula Q(A, I), Such that Q is satisfiable iff ‘**A’** has a successful termination with Input **I**.

* + If the length of ‘**I**’ is ‘**n’** and the time complexity of **A** is **p(n)** for some polynomial **p()** then length of Q is **O(p3(n) log n)=O(p4(n))**

The time needed to construct Q is also **O(p3(n) log n).**

* + A deterministic algorithm ‘**Z’** to determine the outcome of ‘**A**’ on any input ‘**I**’

Algorithm **Z** computes ‘**Q**’ and then uses a deterministic algorithm for the satisfiability problem to determine whether ‘**Q**’ is satisfiable.

* + If **O(q(m))** is the time needed to determine whether a formula of length ‘**m**’ is satisfiable then the complexity of ‘**Z**’ is **O(p3(n) log n + q(p3(n)log n)).**
  + If satisfiability is ‘**p**’, then ‘**q(m)**’ is a polynomial function of ‘**m**’ and the complexity of ‘**Z**’ becomes ‘**O(r(n))’** for some polynomial ‘**r()’.**
  + Hence, if satisfiability is in **p**, then for every nondeterministic algorithm **A**

in **NP**, we can obtain a deterministic **Z** in **p.**

By this we shows that satisfiability is in **p** then **P=NP**