## BASIC SEARCH AND TRAVERSAL TECHNIQUES

### AND/OR Graphs

Breaking down a problem into several subproblems such that the solution of all or some of these subproblems results in a solution of the original problem is called **problem reduction**. If a subproblem is found to be still complex, it may be broken down further into sub-subproblems and so on until the only problems remaining are trivially solvable. Problem reduction is used in theorem proving, analysis of industrial schedule. Problem reduction can be represented by an AND/OR graph. It is a directed graph in which nodes represent problems and descendants of a node represent the subproblems associated with it.

Some definitions:

OR node: A node that can be solved by solving one of its descendants.

### BACKTRACKING

Backtracking is used for finding a set of solutions or an optimal solution satisfying some constraints.

The desired solution must be expressible as an n-tuple $(x_1, x_2,\ldots,x_n)$ where $x_i$ is chosen from some finite set $S_i$.

Often, we come across problems for which we have to find a solution vector which maximizes (or minimizes) a criterion function $P(x_1, x_2,\ldots,x_n)$.

Backtracking is mainly used to find solutions that satisfy a complex set of constraints:

i) Explicit constraints: These constraints restrict each $x_i$ to take on values only from a given set. E.g., in 0/1 Knapsack problem, $x_i = 0$ or 1, i.e., $S_i=\{0,1\}$.

All tuples that satisfy the explicit constraints define a possible solution space for the problem instance, I.

ii) Implicit constraints: These constraints determine which of the tuples in the solution space of I actually satisfy the criterion function.

E.g., for O/1 Knapsack, the tuple with maximum profit.

### The 8-queens problem

Place eight (8) queens on an 8x8 chessboard so that no two queens attack, i.e., no two queens are on the same row, column or diagonal. So the solution vector is $(x_1, x_2,\ldots,x_n)$ where n=8. Here, $x_i$ denotes the column position in which queen i is placed. We assume that queen i is placed on row i.

| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| | Q | | | | | | | |
| | Q | | | | | | | |
| Q | | | | | | | | |
| Q | | | | | | | | |
| Q | | | | | | | | |
| Q | | | | | | | | |
| Q | | | | | | | | |
| Q | | | | | | | | |

i) Explicit constraints: So, $x_i \in S_i$, where $S_i$={1,2,3,4,5,6,7,8}.
Thus, the solution space consists of $8^8$ tuples.
($x_1$,$x_2$,$x_3$,$x_4$,$x_5$,$x_6$,$x_7$,$x_8$)
1,1,1,1,1,1,1,1
1,1,1,1,1,1,1,2
1,1,1,1,1,1,1,3
……..
…….
8,8,8,8,8,8,8,8

8x8x8x8x8x8x8x8= $8^8$ tuples in the solution space.

ii) Implicit constraints: a) No two $x_i$'s are the same (i.e., no two queens on the same column).
b) No two queens are on the same diagonal.

The n-queens problem is a generalization the of the 8-queens problem.
To reduce the solution space, we can move some constraint from the implicit constraints to the explicit constraints. For instance for the 8-queens problem:

i) Explicit constraints: So, $x_i \in S_i$, where $S_i$={1,2,3,4,5,6,7,8} and $x_i \neq x_j$ if $i \neq j$, i.e., no two $x_i$'s are on the same (i.e., no two queens on the same column).
Thus, the solution space consists of 8! tuples.
($x_1$,$x_2$,$x_3$,$x_4$,$x_5$,$x_6$,$x_7$,$x_8$)
1,2,3,4,5,6,7,8
1,2,3,4,5,6,8,7
……
…..
8x7x6x5x……x1 = 8! tuples in the solution space.

ii) Implicit constraints: No two queens are on the same diagonal.

<u>0/1 Knapsack problem:</u>
i) Explicit constraints.= $x_i \in S_i$, $S_i$={0,1}.
What is the size of the solution space when n=4?
($x_1$,$x_2$,$x_3$,$x_4$)
2X2x2x2 = $2^4$ tuples in the solution space

ii) Implicit constraints= the tuple that gives maximum profit.
The solution space can be represented graphically by the state space tree.

Generate the problem states (nodes) in the tree. Determine which are <u>solution states</u> (nodes) and finally determine which solution states are <u>answer states</u> (nodes).
A solution node represents a tuple in the solution space.

<u>Live node</u>: A node which has been generated and all of whose children are not yet generated.
<u>E-node (node being expanded)</u>: The live node whose children are currently being generated.

Dead node: A generated node which is not to be expanded further or one for which all of its children have been generated.
Bounding functions will be used to kill nodes without generating all their children.

Backtracking involves generating the tree to get to the answer node(s).
Depth first generation of the state space tree with bounding functions is called Backtracking.

Why is backtracking called 'method of last resort'?

- A backtracking algorithm on one problem instance might generate only O(n) nodes while on a different instance it might generate almost all the nodes in the state space tree. If the number of nodes in the solution space is $2^n$ or n!, the worst case time for a backtracking algorithm will generally be $O(p(n)2^n)$ or O(q(n)n!) respectively, where, p(n) and q(n) are polynomials in n. The advantage of backtracking is the ability to solve some instances with large n in a very small amount of time. The only difficulty is in predicting the behaviour of the algorithm for the problem instance we wish to solve. Thus, when we use backtracking, if we are unlucky, we could end up taking time, of the order of $O(p(n)2^n)$ or O(q(n)n!), depending on the size of the state space tree.
This is the reason why backtracking is called a method of last resort. We try not to use it unless there are other options.

```
Nqueens(i,n) //Initial call is Nqueens(1,n)
begin
for j:=1 to n do //n columns
        if Place(i,j)==true then //Can I place a queen in i^th row at column j.
                x[i] = j;
                if (i==n) then //an answer found, all queens are placed
                        print(x[1:n]);
                else
                        Nqueens(i+1,n)
                endif
        endif
endfor
end
```

```
Place(i,j)
//returns true if x[i] can be assigned with j. Can ith queen be placed on column j?
//Otherwise false. x is a global array whose first i-1 values have been set.
begin
for l = 1 to i-1 do
        if (x[l]==j) or abs(l-i)==abs(x[l]-j) then
                        //on same col or same diagonal
                return false;
        endif
endfor
return true;
end
```

**Applications of N-queens problem:**
It has found several real-world applications, including practical task scheduling and assignment, computer resource management, and VLSI testing

Reference: P, S.S., 2011. New decision rules for exact search in n-queens. J. Global Optim. 497–514.

**Sums of Subsets**
Given n+1 numbers, $w_i$, 1<=i<=n and M, find all subsets of $w_i$, whose sum is M.
E.g., n =4, (11,13,24,7), M=31.
Solution subsets are (11,13,7) and (24,7).

Variable-size tuple formulation: A solution is represented by a tuple, ($x_1$, $x_2$,.., $x_k$), where 1<=k<=n and , $x_i$= {j|j is an integer, 1<=j<=n}.
Thus, the solutions for the above problem instance is denoted by ($x_1$, $x_2$, $x_3$) = (1,2,4), and ($x_1$, $x_2$)=(3,4).

i) Explicit constraints: $x_i$= {j|j is an integer, 1<=j<=n}. No two subsets should be the same. So, $x_i < x_{i+1}$, 1<=i<n.
NOTE: The corresponding state space tree consists of $2^n$ nodes. Each node, except the root node represents a solution node, or in other words, a tuple that satisfies the explicit constraints.

ii) Implicit constraints: Sum of the subset is equal to M.

Fixed-size tuple formulation: A solution is represented by a tuple, ($x_1$, $x_2$,.., $x_n$) and $x_i$= {0,1}, 1<=i<=n.
Thus, the solutions for the above problem instance is denoted by ($x_1$, $x_2$, $x_3$, $x_4$) = (1,1,0,1), and ($x_1$, $x_2$, $x_3$, $x_4$)=(0,0,1,1).

i) Explicit constraints: $x_i$= {0,1}.
NOTE: The corresponding state space tree consists of $2^n$ leaf nodes. Each leaf node represents a solution node, or in other words, a tuple that satisfies the explicit constraints.

ii) Implicit constraints: Sum of the subset is equal to M.

A bounding function is used to decide whether a particular assignment, $x_k$ will lead to a solution. Otherwise, the node that represents it in the state space tree is killed. Using the fixed-size tuple formulation, the bounding function is given by:

$$\sum_{i=1}^{k}(w_i x_i) + \sum_{i=k+1}^{n}(w_i) >= M$$

Only if the above is true, then the assignment so far, i.e., $x_{1.....}$ $x_k$ can lead to a solution. Otherwise it will not, since adding all the remaining weights to the solution found so far will yield a sum which is less than M.

```
sumofSubs(k,n) //Initial call is sumofSubs (1,n)
begin
for i:=0 to 1 do //only two choices: 0 or 1
        if Admissible(k,i)==true then
                x[k] = i;
                if (k==n) then //check whether the sum == M
                        sum = 0;
                        for j = 1 to n do
                                sum = sum + x[j].w[j];
                        endfor
                        if sum==M then  //answer found
                                print(x[1:n]);
                        endif
                else  sumofSubs(k+1,n)
                endif
        endif
endfor
end

Admissible (k,i)
//returns true if x[k] can be assigned with i.
//Otherwise false. x is a global array whose first k-1 values have been set.
begin
sum=0;
for j = 1 to k-1 do
        sum = sum + x[j].w[j];
endfor
sum = sum + i.w[k];
for j = k+1 to n do
        sum = sum + w[j];
endfor
if sum < M then
        return false;
else
        return true;
endif
end
```

**Graph Coloring**

Let G be a graph and m be a positive integer. Can nodes of G be colored in such a way that no two adjacent nodes have the same color yet only m colors are used? This is called the m-colorability decision problem? It is known that if the degree of the given graph is d, then it can be colored with d+1 colors.

The m-colorability optimization problem seeks for the smallest integer m for which a graph G can be colored. This number is called the chromatic number of the graph.
A graph is planar iff it can be drawn on a plane such that no two edges cross each other.

Assume an m-colorability decision problem where the number of nodes in the graph is n.
Fixed-size tuple formulation: A solution is represented by an n-tuple, ($x_1$, $x_2$,.., $x_n$)

i) Explicit constraints: $x_i$= {j | j is a positive integer, 1<=j<=m}.
The size of the solution space is $m^n$. That is, there are $m^n$ leaf nodes in the state space tree.

ii) Implicit constraints: $x_i \neq x_j$ if there exists an edge (i,j). That is, no two adjacent nodes are given the same color.

```
mColoring(k,n,m) //Initial call is mColoring(1,n,m)
begin
for i:= 1 to m // m colors numbered 1 to m.
        if Colorable(k,i)==true then
                x[k] = i;
                if (k==n) then //all the nodes are colored
                        print(x[1:n]);
                else
                        mColoring(k+1,n,m);
                endif
        endif
endfor
end

Colorable(k,i)
//returns true if x[k] can be assigned with i, i.e., node k can be assigned color i.
//Otherwise false. x is a global array whose first k-1 values have been assigned already.
begin
sum=0;
for j = 1 to k-1 do //check whether i clashes with any other assigned colors
        if adjacent(j,k)==true and x[j]==i then
```

```
                    //node k is adjacent to j and same color
                return false;
        endif
endfor
return true;
end
```

4-color problem for planar graphs:

Given any map, can the regions be colored in such a way that no 2 adjacent regions have the same color, yet only four colors are used? For several years, 5 colors were thought to be sufficient. But after several hundred years, a group of mathematicians solved it with the help of a computer. Yes, 4 is sufficient.

**Hamiltonian Cycle**

Let G=(V,E) be a connected graph with n vertices. A Hamiltonian cycle (suggested by Sir William Hamilton) is a round–trip path along n edges of G that visits every vertex once and returns to its starting vertex.

A TSP tour is a Hamiltonian cycle with the least cost.

A Hamiltonian path is a path that visits every vertex exactly once.

Use backtracking to find all Hamiltonian cycles in a graph. Only distinct cycles are output.
Fixed-size tuple formulation: A solution is represented by an n-tuple, $(x_1, x_2,.., x_n)$, where $x_i$ is the ith visited vertex of the cycle. Let $x_1$=1 to avoid printing a cycle many times.

i) Explicit constraints: $x_i$= {j | j is a positive integer, 1<=j<=n} and $x_1 = 1$. $x_i \neq x_j$ if $i \neq j$.

The size of the solution space is 1.(n-1).(n-2)….1 = (n-1)!. That is, there are (n-1)! leaf nodes in the state space tree.
ii) Implicit constraints: $x_i$ is adjacent to $x_{i-1}$ and $x_n$ is adjacent to 1, for 1<=i<=n.

```
Hamiltonian(k,n) //Initial call is Hamiltonian(2,n)
//x[1] has been assigned 1 already in the main module before calling Hamiltonian(2,n)
Begin
for i:= 2 to n // vertex 2 to vertex n
        if Assign(k,i)==true then
                x[k] = i;
                if (k==n) then //all the nodes are visited once, go back to 1
                        print(x[1:n]);
                else
                        Hamiltonian (k+1,n);
                endif
        endif
endfor
end
```

```
Assign(k,i)
//returns true if x[k] can be assigned with i.
//Otherwise false. x is a global array whose first k-1 values have been assigned already.
Begin
for j = 1 to k-1 do //Check if i has been assigned earlier; if so, it cannot be assigned to x[k]
        if x[j] == i then
                return False
        endif
endfor
if adjacent(x[k-1],i)==true then
        if k==n then
                if adjacent(i,1)==true then
                        return true;
                endif
        else
                return true;

        endif
endif
return false;
end
```

We have used recursive algorithms for backtracking thus far. However, iterative backtracking algorithm is also an option.

```
Algorithm BackTrack(n)//Iterative general backtrack
begin
   k=1;
   while k≠0 do
        if there remains an untried x[k] ϵT(x[1],x[2],.,x[k-1])  && B(x[1],x[2],….x[k]) is true then
                if (x[1],x[2],….x[k]) is a path to an answer node, then
                        print (x[1],x[2],….x[k]);
                k = k +1; //consider the next set
        else
                k=k-1;
        endif
   dowhile
end
```

T(x[1],x[2],….x[k-1]) will yield the possible set of values that can be assigned to x[k] based on the values already assigned to x[1],x[2],…, x[k-1].  B(x[1],x[2],….x[k]) is the bounding function

which returns either true or false. It is used to kill nodes. Only if it returns true, we proceed. Otherwise, backtrack.

---