Backtracking

- most general technique used for searching
- search for a set of solutions or an optimal solution satisfying some constraints can be solved using backtracking
- named by D. H. Lehmer in 1950
- desired solution expressed using n tuple $(x_1, x_2, \ldots, x_n)$
- x_i is chosen from some finite set S_i
- the problem solution - finding one vector that maximizes or minimizes or satisfies a criterion function $P(x_1, \ldots, x_n)$
- sometime it seeks all vectors that satisfy P
- say sorting problem $a[1:n]$
- n tuple x_i index in a of ith smallest element
- P criterion function is inequality $a[x_i] < a[x_{i+1}]$

Backtracking

- m_i is the size of set S_i
- $m = m_1 m_2 \cdot m_n$ n tuples; possible candidates satisfying the function P
- brute force approach - form all n tuples, evaluate each one with P and save which yield the optimum
- backtracking algorithm ability to answer with fewer than m trials
- build up solution vector one component at a time
- use modified criterion functions $P_i(x_1, \ldots, x_i)$ - called bounding functions
- test whether the vector being formed has any chance of success
- advantage: if it is realized that the partial vector $(x_1, \ldots, x_i)$ - no way lead to an optimal solution
- then $m_{i+1} \cdots m_n$ possible test vectors can be ignored entirely

Backtracking

- the problems solved using backtracking require that the solutions satisfy a complex set of constraints
- constraints divided into two categories: explicit and implicit
- explicit constraints are rules - restrict each x_i to take on values from a given set
- example:

$$
\begin{array}{lclcl}
x_i \geq 0 & \text{or} & S_i & = & \{\text{all nonnegative real numbers}\} \\
x_i = 0 \text{ or } 1 & \text{or} & S_i & = & \{0, 1\} \\
l_i \leq x_i \leq u_i & \text{or} & S_i & = & \{a : l_i \leq a \leq u_i\}
\end{array}
$$

- depend on the particular instance I of the problem being solved
- all tuples that satisfy the explicit constraints define a possible solution space for I

Backtracking

- implicit constraints are rules that determine which of the tuples in the solution space of I satisfy the criterion function
- describe the way in which the x_i must relate to each other
- 8-queens - a classic combinatorial problem
- place 8 queens on an 8×8 chessboard so that no two "attack"
- no two of them are on the same row, column or diagonal

			Q				
					Q		
							Q
	Q						
						Q	
Q							
		Q					
				Q			

- represents one of solution

8-queen problem

- each queen must be on a different row
- assume queen i is to be placed on row i
- all solutions can be represented as 8-tuples $(x_1, \ldots, x_8)$
- x_i is the column on which queen i is placed
- explicit constraints $S_i = \{1, 2, 3, 4, 5, 6, 7, 8\}$, $1 \leq i \leq 8$
- solution space consists of 8^8 8-tuples
- implicit constraints are that no two x_i's can be the same, i.e, all queens must be on different columns and no two queens can be on the same diagonal
- it reduces solution space from 8^8 to 8! tuples
- solution in example $(4, 6, 8, 2, 7, 1, 3, 5)$

Sum of subsets problem

- Given positive numbers w_i $(1 \leq i \leq n)$ and m
- finds all subsets of the w_i whose sums are m
- $n = 4$ $(w_1, w_2, w_3, w_4) = (11, 13, 24, 7)$ and $m = 31$
- subsets are vectors $(11, 13, 7)$ and $(24, 7)$
- solution vector can be represented by indices of these w_i
- now, solutions vectors $(1, 2, 4)$ and $(3, 4)$
- all solutions are k tuples $(x_1, x_2, \ldots, x_k)$ $1 \leq k \leq n$
- different solutions - different sized tuples
- explicit constraints $x_i \in \{j | j \text{ is an integer and } 1 \leq j \leq n\}$
- implicit constraints - no two be the same and that the sum of the corresponding w_i's be m
- to avoid multiple instances of the same subset $(1, 2, 4)$ and $(1, 4, 2)$ - another implicit constraint imposed is that $x_i < x_{i+1}$ $1 \leq i < k$

Sum of subsets problem

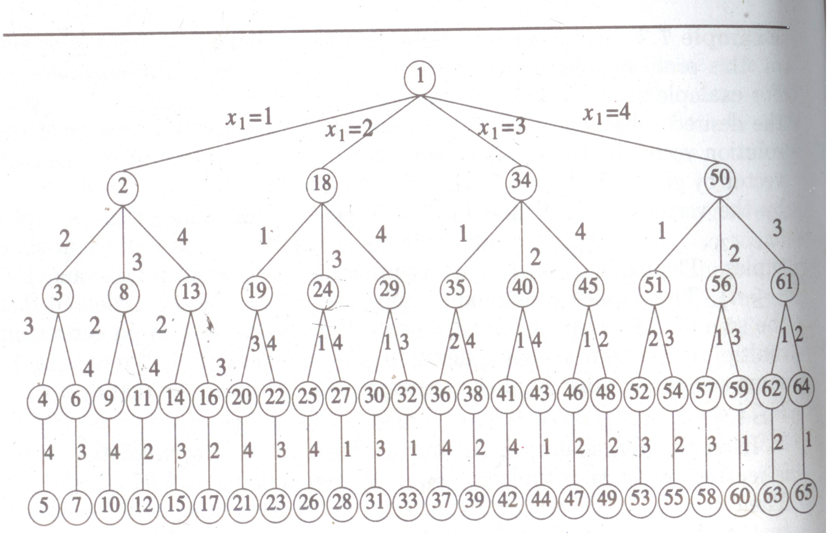
- another formulation - solution represented by n tuple $(x_1, x_2, \ldots, x_n)$ such that $x_i \in \{0, 1\}$ $1 \leq i < n$
- $x_i = 0$ if w_i is not chosen and $x_i = 1$ if w_i is chosen
- for example seen earlier - solution $(1, 1, 0, 1)$ and $(0, 0, 1, 1)$
- all solutions have a fixed sized tuple
- several ways to formulate a problem satisfy the same constraints
- solution space consists of 2^n distinct tuples
- backtracking determines solution by systematically searching the solution space for the given problem instance
- tree organization is used for solution space
- for a given solution space many tree organizations may be possible

n queens problem

- n queens are to be placed on an $n \times n$ chessboard so that no two attack
- solution space consists of all $n!$ permutations of the n-tuple $(1, 2, \ldots, n)$
- tree organization - permutation tree
- edges are labeled by possible values of x_i
- edges from level 1 to level 2 nodes specify the values for x_1
- leftmost subtree contains all solutions with $x_1 = 1$ and its leftmost subtree contains all solutions with $x_1 = 1$ and $x_2 = 2$ and so on
- edges from level i to level $i + 1$ are labeled with the values of x_i
- solution space is defined by all paths from the root node to a leaf node
- with $n = 4$, there are $4! = 24$ leaf nodes

4-queen solution space - Tree organization

- nodes are numbered as in depth first search

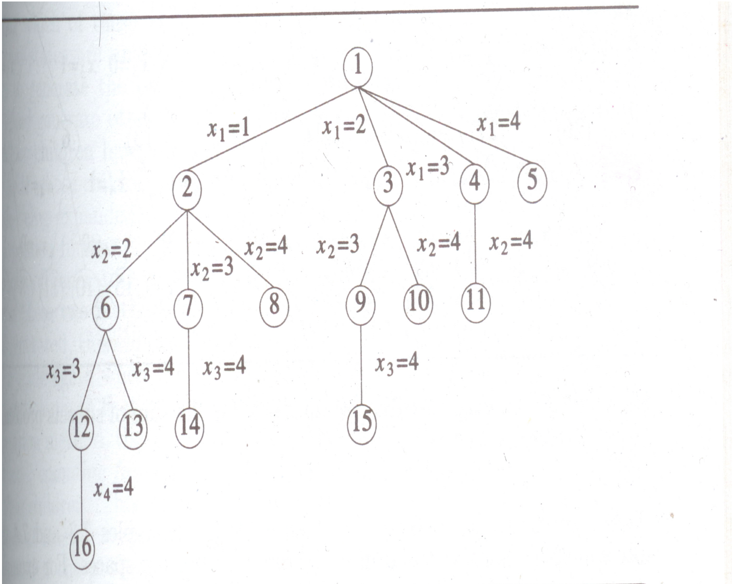


Tree organizations

- every element of the solution space represented by at least one node in the state space tree
- tree organizations are independent of the problem instance being solved - static trees
- it is advantageous to use different tree organizations for different problem instances
- tree organization is determined dynamically as the solution space is being searched
- tree organizations that are problem instances dependent are called dynamic trees
- once a state space tree conceived for any problem,
- problem can be solved by systematically generating the problem states, determining which of these are solutions states and finally determining which solution states are answer states

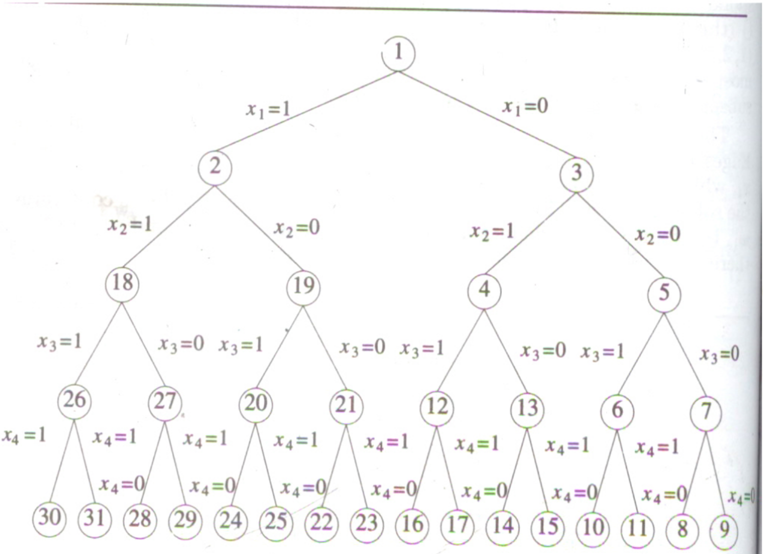
Sum of subset solution space organization

- nodes are numbered as in breadth first search



Sum of subset another organization

- nodes are numbered as in DF search



Backtracking

- two ways to generate the problem states
- both begin with root node and generate other nodes
- a node which has been generated and all of whose children have not yet been generated is called a live node
- the live node whose children are currently being generated is called the E-node (node being extended)
- a dead node is a generated node which is not to be expanded further or all of whose children have been generated
- in both methods of generating problem states, there is a list of live nodes
- first method - as soon as a new child C of the current E node R is generated, this child will become the new E node

Backtracking

- R will become the E node again when the subtree C has been fully explored
- this corresponds to a depth first generation of the problem states
- second state generation method - E node remains E node until it is dead
- in both methods, bounding functions are used to kill live nodes without generating all their children
- it is done carefully, so that at the conclusion of the process at least one answer node is always generated or all answer nodes are generated
- Depth first node generation with bounding functions is called backtracking
- second method is called branch-and-bound methods

Backtracking

- 4-queen problem - bounding function
- use criteria that if $(x_1, x_2, \ldots, x_i)$ is the path to the current E node then all children nodes with parent-child labeling x_{i+1} are such that $(x_1, \ldots, x_{i+1})$ represents a chessboard configuration in which no two queens are attacking

1			

1			
·	·	2	

1			
		2	
·	·	·	·

1			
			2
·	3		

1			
			2
	3		
·	·	·	·

	1		

	1		
·	·	·	2

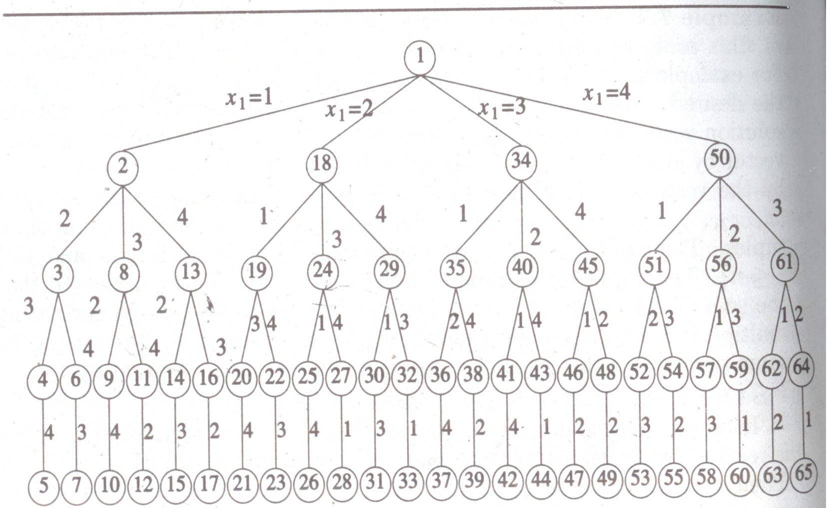
	1		
			2
3			
·	·	4	

4-queen problem using Backtracking

- start with root node as live node; it becomes E node and the path ()
- generate one child (ascending order) - node number 2 generated and the path (1)
- this corresponds to placing queen 1 on column 1 and node 2 becomes E node
- node 3 is generated and immediately killed
- node generated next is node 8 and the path becomes $(1, 3)$ and node 8 becomes the E node
- it gets killed as all its children represent board configurations that can not lead to an answer node
- backtrack to node 2 and generate another child node 13 and the path is $(1, 4)$
- dots indicate placements of a queen which were tried and rejected because another queen was attacking

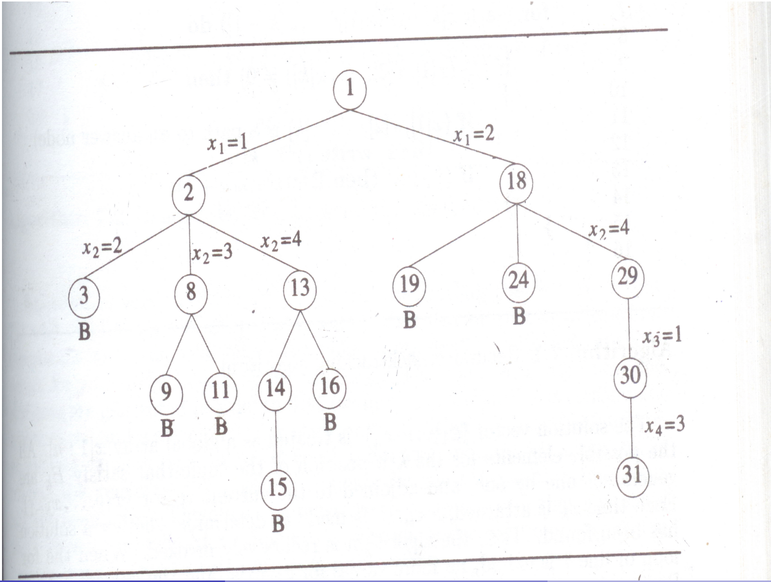
4-queen solution space - Tree organization

- nodes are numbered as in depth first search



4-queen problem

- portion of tree generated during backtracking



Backtracking formulation and process

- assume that all answer nodes are to be found
- let $(x_1, x_2, \ldots, x_i)$ be a path from the root to a node in a state space tree
- let $T(x_1, x_2, \ldots, x_i)$ be the set of all possible values for x_{i+1} such that $(x_1, x_2, \ldots, x_{i+1})$ is also a path to a problem state
- $T(x_1, x_2, \ldots, x_n) = \varnothing$
- the existence of bounding function B_{i+1} (expressed as predicted) such that
- if $B_{i+1}(x_1, x_2, \ldots, x_{i+1})$ is false for a path $(x_1, x_2, \ldots, x_{i+1})$ from the root node to a problem state then the path can not be extended to reach answer node
- the candidates for position $i+1$ of the solution vector $(x_1, \ldots, x_n)$ are those values which are generated by T and satisfy B_{i+1}
- recursive formulation - postorder traversal of a tree

backtrack(k)

for each $x[k] \in T(x[1], \ldots, x[k-1])$ do
    if $(B_k(x[1], x[2], \ldots, x[k]) \neq 0)$ then
    {
        if $(x[1], x[2], \ldots, x[k])$ is a path to an answer node
            then write $(x[1:k])$
        if $(k < n)$ then backtrack$(k+1)$
    }

- recursion is initially invoked by backtrack(1)
- solution vector $(x_1, \ldots, x_n)$ is treated as a global array $x[1:n]$
- all possible elements for the kth position of the tuple that satisfy B_k are generated, one by one, and adjoined to the current vector $(x_1, \ldots, x_{k-1})$
- each time x_k is attached, a check is made to determine whether a solution has been found then algorithm is recursively invoked

ibacktrack(n)

1. $k = 1$
2. while ($k \neq 0$) do
3. if (there remains an untried $x[k] \in T(x[1], x[2], \ldots, x[k-1])$ and $B_k(x[1], \ldots, x[k])$ is true) then
4. {
5. if ($x[1], \ldots, x[k]$) is a path to answer node then
6. write ($x[1:k]$)
7. $k = k + 1$
8. }
9. else $k = k - 1$

- $T()$ will yield the set of all possible values that can be placed as the first component x_1 of the solution vector
- component x_1 will take on those values for which the bounding function $B_1(x_1)$ is true

ibacktrack(n)

- elements are generated in a depth first manner
- k is continually incremented and a solution vector is grown until either a solution is found or notried value of x_k remains
- k is decremented - the algorithm resumes the generation of possible element for the kth position that have not yet been tried
- one must develop a procedure that generates these values in some order
- efficiency of both algorithms depends on
  - time to generate the next x_k
  - number of x_k satisfying the explicit constraints
  - time for the bounding functions B_k
  - number of x_k satisfying the B_k
- bounding functions are good if they substantially reduce the number of nodes that are generated

Backtracking

- good bounding function may take more time to evaluate
- desired is reduction in overall computing time and not just reducing number of nodes
- many problems, size of state space tree is too large to permit the generation of all nodes
- bounding function must be used to find solution in a reasonable time span
- no sophisticated bounding methods are known for many problems
- efficient searching through rearrangement is also used with dynamic state space tree
- allows to eliminate significant number of nodes at a particular level

Backtracking: Timing analysis

- four factors (stated earlier) determine the time required by an algorithm
- once a state space tree organization is selected, the first three of these are relatively independent of the problem instance being solved
- last factor, the number of nodes generated, varies from one problem instance to another
- for example, for one instance it may generate $O(n)$ nodes and for different instance it may generate all nodes in state space tree
- if the number of nodes in solution space 2^n or $n!$ - worst case time for an algorithm will be $O(p(n)2^n)$ or $O(q(n)n!)$ where $p(n)$ and $q(n)$ are polynomials
- estimate number of nodes generated on a certain instance using Monte Carlo methods

Backtracking

- general idea in the estimation method is to generate random path in the state space tree
- X be a node on this random path, X is at level i of the state space tree
- bounding functions are used at node X to determine the number of m_i of its children that do not get bounded
- interested in estimating the total number of nodes m in state space tree that will not get numbered
- bounding functions are static if do not change as information is gathered during its execution
- many cases, bounding functions get stronger as the search proceeds
- say, bounding functions are static and unbounded nodes on level 2 is m_1

Backtracking

- if search tree is such that the nodes on the same level have the same degree then each level 2 node to have on the average m_2 unbounded children
- this yields a total of $m_1 m_2$ nodes on level 3
- the expected number of unbounded nodes on level 4 is $m_1 m_2 m_3$
- at level $i + 1$ it is $m_1 m_2 \ldots m_i$
- estimated number m for a given problem instance I is $m = 1 + m_1 + m_1 m_2 + \cdots$
- select several different random paths and determine the average of m values to obtain better estimate of number of unbounded nodes

Estimate unbounded nodes

1. $k = 1, m = 1, r = 1$
2. repeat
3. $\{$
4. $\quad T_k = \{x[k] | x[k] \in T(x[1], x[2], \ldots, x[k-1])$ and $B_k(x[1], \ldots, x[k])$ is true
5. $\quad$ if $(size(T_k) = 0)$ then return m
6. $\quad r = r \times size(T_k);\ m = m + r$
7. $\quad x[k] = \text{choose}(T_k);\ k = k + 1$
8. $\}$ until (false);

N-queens problem

- with problem formulation seen earlier
- how to test whether two queens are on the same diagonal
- 2D array $a[1:n, 1:n]$; every element on the same diagonal that runs from the upper left to the lower right has the same row-column value
- every element on the same diagonal that goes from the upper right to the lower left has the same row+column value
- say, two queens are placed at (i,j) and (k,l) then they are on the same diagonal only if

$$\begin{array}{rccc} & i-j=k-l & \text{or} & i+j=k+l \\ \Longrightarrow & j-l=i-k & \text{and} & j-l=k-i \text{ respectively} \end{array}$$

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

NQueens(k, n)

for $i = 1$ to n do
    if place(k, i) then
    {
        $x[k] = i$
        if $(k = n)$ then write($x[1:n]$)
        else NQueens($k + 1, n$)
    }

place(k, i)

for $j = 1$ to $k - 1$ do
    if $((x[j] = i)$ # two in the same column
        or $(\text{abs}(x[j] - i) = \text{abs}(j - k)))$ # or in the same diagonal
        then return false;

N-queens problem

- place(k, i) returns true if kth queen can be placed in column i
- it test whether i is distinct from all previous values $x[1]. \ldots, x[k-1]$ and whether there is no other queen on the same diagonal
- computing time is $O(k-1)$
- brute force approach 8×8 chessboard there are $\left(\begin{array}{c} 64 \\ 8 \end{array}\right)$ possible ways to place 8 queens
- approximately 4.4 billion 8 tuples to examine
- queens on distinct rows and columns require to examine at the most 8!, i.e., 40320 8-tuples

Sum of subsets problem

- backtracking solution using fixed tuple size
- element x_i of the solution vector is either one or zero depending on whether number w_i is included or not
- for a node at level i the left child corresponds to $x_i = 1$ and the right to $x_i = 0$
- bounding function may be $B_k(x_1, \ldots, x_k) =$ true iff

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

- if first condition is not satisfied then $x_1, \ldots, x_k$ can not lead to an answer
- bounding functions can be strengthened using second condition
- that is w_i's are initially in nondecreasing order

Sum of subsets problem

- no need of B_n, there is no w_{n+1}
- assume that $w[1] \leq m$ and $\sum_{i=1}^{n} w[i] \geq m$
- using s and r to store partial sum value for $\sum_{i=1}^{k-1} w[i]x[i]$ and $\sum_{k}^{n} w[i]$ respectively
- initial call sumofsub$(0, 1, \sum_{i=1}^{n} w[i])$ = sumofsub(s, k, r)
- no need to test $k > n$ to terminate recursion as on entry to the algorithm $s \neq m$ and $s + r \geq m$
- $r \neq 0$ and k can not be greater than n
- $s + w[k] < m$ and $s + r \geq m \implies r \neq w[k]$ and hence $k + 1 < n$

sumofsub(s, k, r)

1. # generate left child $s + w[k] \leq m$ since B_{k-1} is true
2. $x[k] = 1$
3. if $(s + w[k] = m)$ then write $(x[1:k])$ # subset found
4. else if $(s + w[k] + w[k+1]) \leq m)$
5.   then sumofsub$(s + w[k], k + 1, r - w[k])$
6. # right child and evaluate B_k
7. if $((s + r - w[k] \geq m)$ and $(s + w[k+1] \leq m)$ then
8. {
9. $x[k] = 0$
10.   sumofsub$(s, k + 1, r - w[k])$
11. }