BACKTRACKING

For a problem P, let C be the set of constraints for P.
Let D be the set containing all solutions satisfying C.

1. Is there a feasible solution?
   $D \neq \emptyset$?
   Decision Problem

2. What is the best solution?
   Optimization Problem

3. List all feasible solutions.
   Enumeration Problem

The Examination Scheduling Problem

Problem 1: Is there a schedule for all courses that uses at most k days?

Problem 2: Find a schedule that uses the minimum number of days.

Problem 3: List all examination schedules that use at most k days.

What are the relationships among these problems?

The Backtracking Method

Usually in the backtracking method

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

- the solution maximizes (or minimizes or satisfies) a criteron function $C(x_1, \ldots, x_n)$.

Problem Categories

- Max/Min – Optimization problem

- Satisfies – decision problem

- All solutions – enumeration problem

Examples

$$\left.\begin{array}{l}\text{1. Sorting}\\ \text{2. Knapsack}\\ \text{3. Activity Scheduling}\end{array}\right\}\text{Representations?}$$

The Eight-Queen Problem

	1	2	3	4	5	6	7	8
1				Q				
2						Q		
3								
4		Q						Q
5							Q	
6	Q							
7			Q					
8					Q			

What are the different representations?
1.
2.
3.
4.

Explicit vs Implicit?
Search space size?

The Subset Sum Problem

Given positive numbers $w_i, 1 \le i \le n$ and m.
Find all subsets of w_i whose sums are m.

For example:
$n = 4$, $m = 31$, $(w_1, w_2, w_3, w_4) = (11, 13, 24, 7)$

Representation?
Using set or a n-bit vector?

Possible solutions using sets:
$(11, 13, 7)$ and $(24, 7)$ or
(w_1, w_2, w_4) and (w_3, w_4)

Possible Solutions using a n-bit vector:
$(1, 1, 0, 1)$ and $(0, 0, 1, 1)$
Solution is represented by $(x_1, x_2, \ldots, x_n)$

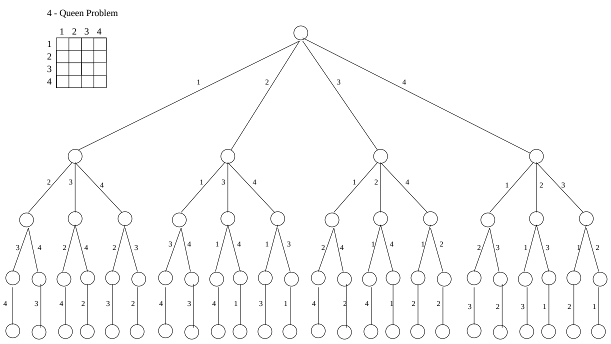
$$x_i = \begin{cases} x_i = 0 \text{ if } w_i \text{ is not chosen} \\ x_i = 1 \text{ if } w_i \text{ is chosen} \end{cases}$$

Problem? Size Space Size?

Systematic Search

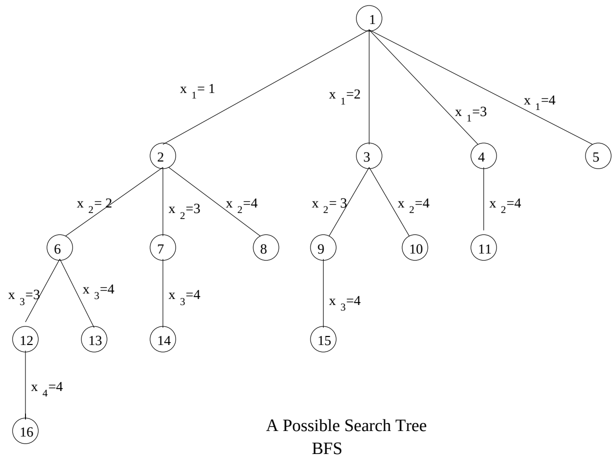
Backtracking algorithms determine problem solutions by systematically searching the solution space using a tree structure.

Using the Four-Queen Problem:



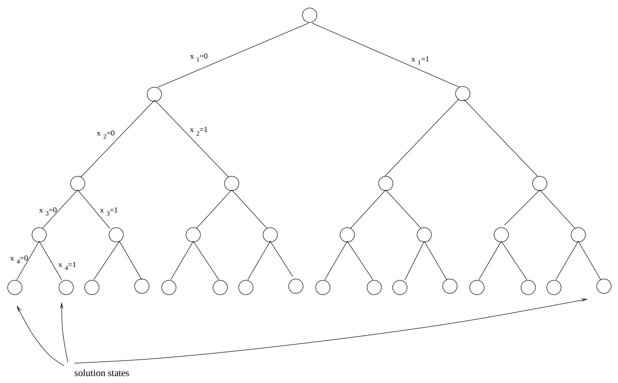
Sum of Subset

A possible search tree:



A Possible Search Tree
BFS

Another State Space Search Tree



- Nodes are also known as problem states
- Answers states are those solution states that satisfy the constraints

Live, Dead, E-Node

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.

A **dead** node is a generated node which is not to be expanded further or all of whose children have been generated.

Two Problem State/Search Tree Generation Methods

Method 1 - Backtracking

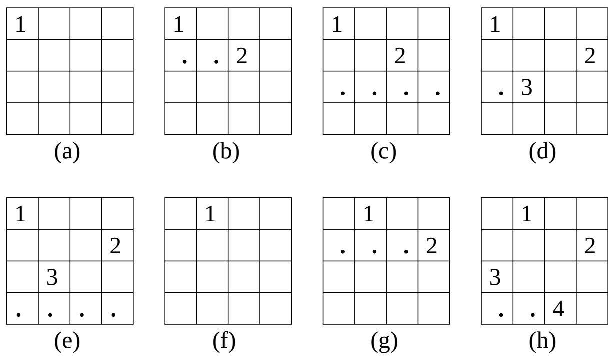
- As soon as a new child C of the current E-node R is generated, this child will be the new E-node.
- R will become the E-node again when the subtree C has been fully explored.

Method 2 - Branch and Bound

- E-node will remain as E-node until it is dead.

- Both methods have bounding functions that are used to kill live nodes without generating all their children.
- This must be done carefully such that at least one answer node or all answer nodes are obtained.

An Example : The Four Queen Problem



Generic Backtracking Algorithm

Algorithm Backtrack(t)
//Recursive Backtrack
//On Entering, the first $k-1$ values
//$(x_1, x_2, \ldots, x_{k-1})$ of the solution
//vector $X = (x_1, \ldots, x_n)$ have been assigned.
//We assume that X and n are global.
{
  for (each $x_k \in T(x_1, \ldots, x_{k-1})$ do {
    if ($B_k(x_1, \ldots, x_k)$ = true) {
      if $(x_1, \ldots, x_k)$ is a path to an answer node
      then output$(x_1, \ldots, x_k)$;
      if ($k < n$) Backtrack($k+1$);
    }
  }
}

- Let $T(x_1, x_2, \ldots, x_i)$ be the set of all possible values for x_{i+1} such that $(x_1, \ldots, x_{i+1})$ is is also a path to a problem state.
- $B_{i+1}(x_1, \ldots, x_{i+1})$ is false for a path $(x_1, \ldots, x_{i+1})$ from the root node to a problem state, then the path cannot be extended to reach an answer node.

Iterative Version

Algorithm IBacktrack(t)
//Iterative Backtrack
{
  $k := 1$;
  while ($k \neq 0$) do{
    if (there remains an untried
      $x_k \in T(x_1, \ldots, x_{k-1})$ and
      ($B_k(x_1, \ldots, x_k)$ = true)
    then
    {
        if $(x_1, \ldots, x_k)$ is a path to an answer node
        then output$(x_1, \ldots, x_k)$;
        $k := k+1$;
    }
    else
      $k := k-1$;
  }
}

The Eight-Queen Problem

Assuming that we place row by row, one per row.
We only need to check for column conflicts and diagonal conflicts.

1,1			1,4				1,8
	2,2	2,3		2,5		2,7	
	3,2	3,3			3,6		
4,1			4,4	4,5		4,7	
			5,4	5,5			5,8
		6,3			6,6		
	7,2					7,7	
8,1							8,8

Let position $P_1 = (i, j)$ and position $P_2 = (k, l)$.
P_1 and P_2 are on the same diagonal if $i + j = k + l$ or $i - j = k - l$.
But $i+j = k+l \equiv j-l = k-i$ and $i-j = k-l \equiv j-l = i-k$.
Therefore, P_1 and P_2 are on the same diagonal iff $|j-l| = |i-k|$.

Algorithm Place (k, i)

//Return true if a queen can be placed
//in the kth row and ith column
//otherwise it returns false
//$x[]$ is a global array whose first $(k-1)$
//values have been set
//$ABS(r)$ returns absolute value of r
{
  for $j := 1$ to $k-1$ do
    if ($x[j] = i$) //same column
    or ($ABS(x[j]-1) = ABS(j-k)$)
    then return false;
  return true;
}

```
Algorithm NQueens (k, n)
//Using backtracking, this procedure prints
//all possible placements of n queens on a
//n × n chessboard so that they are
//non-attacking

{
  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);
    }
  }
}
```


Sum of Subsets

Bounding Functions:

$B_k(x_1, ..., x_k)$ = true iff

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

$x_1, ..., x_k$ cannot lead to an answer if above is not satisfied.
Can be strengthened by first sorting w_i in non-decreasing order. In this case, $x_1, ..., x_k$ cannot lead to an answer node if

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

$B_k(x_1, ..., x_k)$ = 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 SumofSub(s, k, r)
//Finding all subsets of w[1 : n] that sum
//to m. The values of x[j], 1 ≤ j ≤ k
//have already been determined.
//s = Σ_{j=1}^{k-1} w[j] × x[j] and
//r = Σ_{j=k}^{n} w[j]
//w[j]'s are in non decreasing order
//Assume that w[1] ≤ m and Σ_{i=1}^{n} w[i] ≥ m
{
  //Generate left child
  x[k] := 1;
  if (s + w[k] = m);
    then write (x[1 : k]);
  else if (s + w[k] + w[k + 1] ≤ m)
    then SumofSub(s + w[k], k + 1, r − w[k]);
  //Generate right
  if ((s + r − w[k] ≥ m) and (s + w[k + 1] ≤ m))
  then
  {
    x[k] = 0;
    SumofSub(s, k + 1, r − w[k]);
  }
}
```

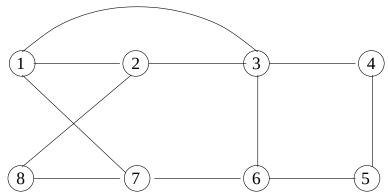

Graph Coloring

```
Algorithm mColoring(k)
//Recursive backtracking.
//Graph is given by adjacency matrix G[1 : n, 1 : n].
//Assignment 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
  {
    NextValue(k);
    if (x[k] = 0) then return;
    if (k = n)
      then write (x[1 : n]);
    else mColoring(k + 1);
  } until (false);
}
```

```
Algorithm NextValue(k)
{
  repeat
  {
    x[k] := (x[k] + 1) mod (m + 1);
    if (x[k] = 0) then return;
    for j := 1 to n do
    {
      if ((G[k, j] ≠ 0) and x[k] = x[j])) then break;
    }
    if (j = n + 1) then return;
  } until (false);
}
```


Hamiltonian Cycles



Hamiltonian Cycles

A hamiltonian cycle is 1,2,8,7,6,5,4,3,1

Let $(x_1, ..., x_n)$ be the solution vector, without any loss of generality, let $x_1 = 1$.

If $1 < k < n$ then x_k can be any vertex v that is distinct from $x_1, ..., x_{k-1}$ and v is connected by an edge to x_{k-1}.

Therefore, x_n is the last remaining vertex and must be connected to x_1 and x_{n-1}.


```
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
//x[k] must also be connected to x[1]

{
  repeat
  {
    x[k] := (x[k] + 1) mod (n + 1);
    if (x[k] = 0) then return;
    if (G[x[k − 1], x[k]] ≠ 0)
    then
    {
      for j := 1 to k − 1 do
        if (x[j] = x[k]) then break;
        if (k < n) or ((k = n) and G[x[n], x[1]] ≠ 0))
        then return;
    }
  } until (false);
}
```



```
Algorithm Hamiltonian (k)
{
  repeat
  {
    NextValue(k);
    if (x[k] = 0) then return;
    if (k = n) then write (x[1 : n]);
    else Hamiltonian(k + 1);
  } until (false)
}
```

How Slow or How Fast?

Analyze the search tree

$N(n)$ - How many nodes do we search?

$P(n)$ - How much time do you spend at each node?

Total time is $N(n) \times P(n)$

Tree size is usually more important

$N(n)$ usually grows exponentially with n.

$P(n)$ usually grows polynomially with n.

Important Insight

To improve any search tree based complete search, we must find ways to cut down the search tree without "omitting" any useful solutions.

Two-Prong Approach to Speed up:

1. Use a good representation - i.e. small underlying search tree
2. Prune effectively

Pruning Techniques

Preclusion : Omit dead vertices (and their subtrees).

Exploit Symmetry : Avoid duplicating symmetric solutions.

Fusion via Branch Merging : Avoid doing same work twice.

Search Rearrangement : Foster bigger pruning.

Static and Dynamic Search Tree

Static : independent of the problem instance

Dynamic: dependent on the problem instance. Tree organization is determined dynamically as the solution space is being searched.