N-Queens

Background

- Problem surfaced in 1848 by chess player Max Bezzel as 8 queens (regulation board size)
- Premise is to place N queens on N x N board so that they are non-attacking
- A queen is one of six different chess pieces
- It can move forward & back, side to side and to its diagonal and anti-diagonals
- ***The problem: given N, find all solutions of queen sets and return either the number of solutions and/or the patterned boards.***

■ **Example I:** The n-queens problem

■ How can we place n queens on an n×n chessboard so that no two queens can capture each other?

A queen can move any number of squares horizontally, vertically, and diagonally.

Here, the possible target squares of the queen Q are marked with an x.

x			x			x	
	x		x		x		
		x	x	x			
x	x	x	Q	x	x	x	x
		x	x	x			
	x		x		x		
x			x			x	
			x				x

Basic Algorithm

Generate a list of free cells on the next row.

– If no free cells, backtrack (step 2 of previous row)

2) Place a queen on next free cell & proceed with step 1 for next row

– If no next row, proceed to step 3

3) At this point, you have filled all rows, so count/store as a solution

Basic Algorithm (example)

F	F	F	F	F

Q				

Q				
		F	F	F

Q				
		Q		

Basic Algorithm (example)

Q				
		Q		
				F

Q				
		Q		
				Q

Q				
		Q		
				Q
	F			

Q				
		Q		
				Q
	Q			

Basic Algorithm (example)

Q				
		Q		
				Q
	Q			
			F	

Q				
		Q		
				Q
	Q			
			Q	

DONE

Optimizing the Algorithm

- Further look ahead—not just next row
  - Keep track of *total* free spaces per row
    - If any row ahead has a zero count, backtrack
- Jump to other rows
  - To prevent unnecessary future backtracks, jump to rows with fewest free spots and place queens there first
- Cell blocking
  - Prevention of placing queens on cells that would lead to repeating of previously found solution and/or putting the board in an unsolvable situation

Optimizing the Algorithm (ex)

5	F	F	F	F	F
5	F	F	F	F	F
5	F	F	F	F	F
5	F	F	F	F	F
5	F	F	F	F	F

-		Q			
2	-	-	-	F	F
3	F	-	F	-	F
3	F	-	F	F	-
4	F	-	F	F	F

-		Q			
-				Q	
1	F	-	-	-	-
2	F	-	F	-	-
3	F	-	F	-	F

-		Q			
-				Q	
-	Q				
1	-	-	F	-	-
1	-	-	-	-	F

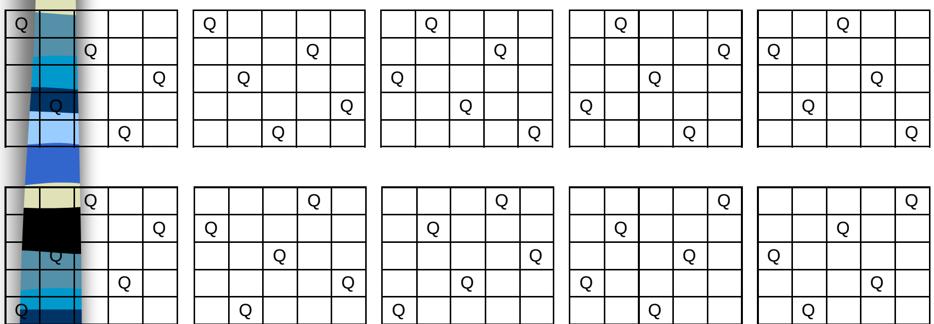
-		Q			
-				Q	
-	Q				
-			Q		
-					Q

Optimizing the Algorithm (ex)

-				Q				
-	Q							
-			Q					
-					Q			
2	-	F	-	-	-	-	F	-
2	-	F	-	-	-	-	-	F
0	-	-	-	-	-	-	-	-

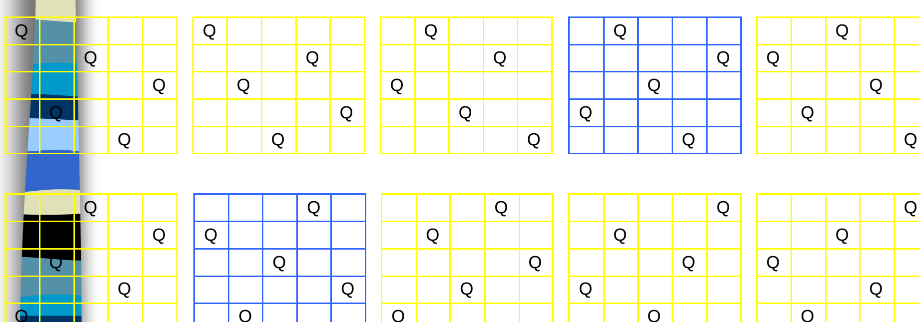
*example of when to backtrack
due to looking ahead (row 7)

Possible Solution Set



All possible solutions for N=5

Possible Solution Set



Only 2 *unique* solutions for N=5
(notice transformations & reflections)

Backtracking

- Suppose you have to make a series of *decisions*, among various *choices*, where
  - You don't have enough information to know what to choose
  - Each decision leads to a new set of choices
  - Some sequence of choices (possibly more than one) may be a solution to your problem
- Backtracking is a methodical way of trying out various sequences of decisions, until you find one that "works"

Backtracking Algorithm

- Based on depth-first recursive search
- Approach
  1. Tests whether solution has been found
  2. If found solution, return it
  3. Else for each choice that can be made
     a) Make that choice
     b) Recur
     c) If recursion returns a solution, return it
  4. If no choices remain, return failure
- Some times called "search tree"

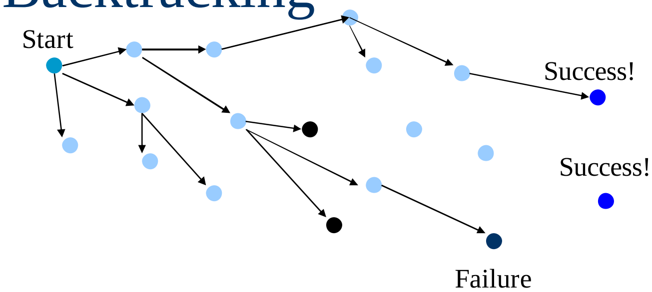
Backtracking Algorithm – Example

- Find path through maze
  - Start at beginning of maze
  - If at exit, return true
  - Else for each step from current location
    - Recursively find path
    - Return with first successful step
    - Return false if all steps fail

Backtracking Algorithm – Example

- Color a map with no more than four colors
  - If all countries have been colored return success
  - Else for each color c of four colors and country n
    - If country n is not adjacent to a country that has been colored c
      - Color country n with color c
      - Recursively color country n+1
      - If successful, return success
  - Return failure

Backtracking



Problem space consists of states (nodes) and actions (paths that lead to new states). When in a node can can only see paths to connected nodes

If a node only leads to failure go back to its "parent" node. Try other alternatives. If these all lead to failure then more backtracking may be necessary.

Recursive Backtracking

Pseudo code for recursive backtracking algorithms

If at a solution, return success
for(every possible choice from current state / node)
- Make that choice and take one step along path
- Use recursion to solve the problem for the new node / state
- If the recursive call succeeds, report the success to the next high level
- Back out of the current choice to restore the state at the beginning of the loop.

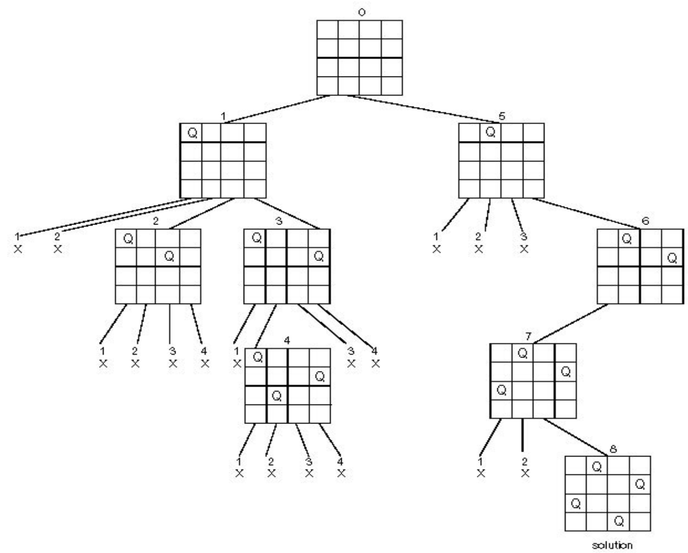
Report failure

Backtracking

- Construct the state space tree:
  - Root represents an initial state
  - Nodes reflect specific choices made for a solution's components.
    - Promising and nonpromising nodes
    - leaves
- Explore the state space tree using <u>depth-first search</u>
- "Prune" non-promising nodes
  - dfs stops exploring subtree rooted at nodes leading to no solutions and...
  - "backtracks" to its parent node

Example: The *n*-Queen problem

Place *n* queens on an *n* by *n* chess board so that no two of them are on the same row, column, or diagonal tate Space Tree of the Four-queens Problem



The backtracking algorithm

- Backtracking is really quite simple--we "explore" each node, as follows:
- To "explore" node N:
    1. If N is a goal node, return "success"
    2. If N is a leaf node, return "failure"
    3. For each child C of N,
        3.1. Explore C
            3.1.1. If C was successful, return "success"
    4. Return "failure"

Backtracking

Sum of Subsets
and
Knapsack

Backtracking

- Two versions of backtracking algorithms
  - Solution needs only to be feasible (satisfy problem's constraints)
    - sum of subsets
  - Solution needs also to be optimal
  - knapsack

The backtracking method

- A given problem has a set of constraints and possibly an objective function
- The solution optimizes an objective function, and/or is feasible.
- We can represent the solution space for the problem using a state space tree
  - The *root* of the tree represents 0 choices,
  - Nodes at depth 1 represent first choice
  - Nodes at depth 2 represent the second choice, etc.
  - In this tree a *path* from a root to a leaf represents a candidate solution

Sum of subsets

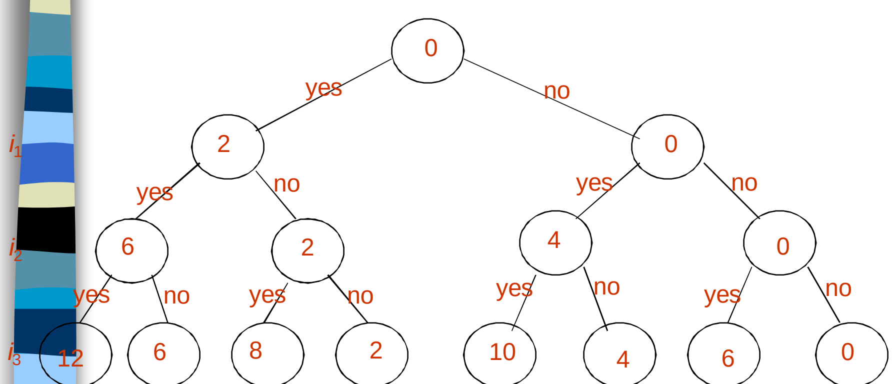
- Problem: Given n positive integers $w_{1,}$ ... w_n and a positive integer S. Find all subsets of $w_{1,}$ ... w_n that sum to S.
- Example:
  n=3, S=6, and $w_1=2$, $w_2=4$, $w_3=6$
- Solutions:
  {2,4} and {6}

Sum of subsets

- We will assume a binary state space tree. The nodes at depth 1 are for including (yes, no) item 1, the nodes at depth 2 are for item 2, etc. The left branch includes w_i, and the right branch excludes w_i.
- The nodes contain the sum of the weights included so far

Sum of subset Problem:
State SpaceTree for 3 items

$w_1 = 2, \quad w_2 = 4, \quad w_3 = 6$ and $S = 6$



The sum of the included integers is stored at the node.

A Depth First Search solution

- Problems can be solved using depth first search of the (implicit) state space tree.
- Each node will save its depth and its (possibly partial) current solution
  DFS can check whether node v is a leaf.
  - If it is a leaf then check if the current solution satisfies the constraints
  - Code can be added to find the optimal solution

A DFS solution

- Such a DFS algorithm will be very slow.
- It does not check for every solution state (node) whether a solution has been reached, or whether a *partial* solution can lead to a *feasible* solution
- Is there a more efficient solution?

Backtracking

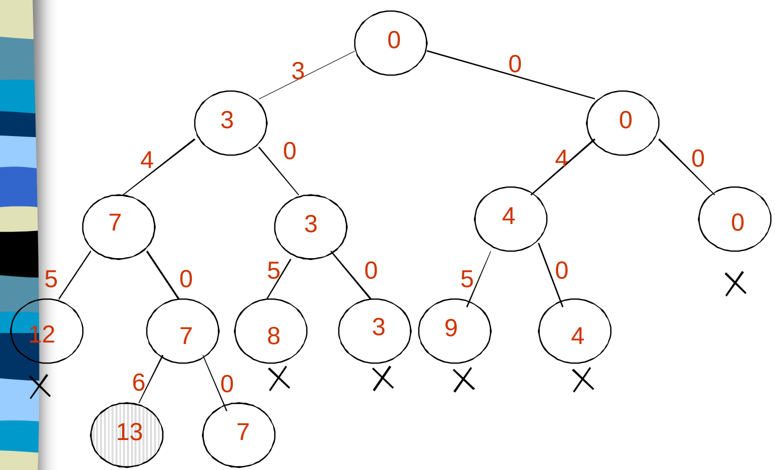
- Definition: We call a node *nonpromising* if it cannot lead to a feasible (or optimal) solution, otherwise it is *promising*
- Main idea: Backtracking consists of doing a DFS of the state space tree, checking whether each node is promising and if the node is nonpromising backtracking to the node's parent

Backtracking

- The state space tree consisting of expanded nodes only is called the *pruned state space tree*
- The following slide shows the pruned state space tree for the sum of subsets example
- There are only 15 nodes in the pruned state space tree
- The full state space tree has 31 nodes

A Pruned State Space Tree (find all solutions)

$w_1 = 3$, $w_2 = 4$, $w_3 = 5$, $w_4 = 6$; $S = 13$



Sum of subsets problem

Backtracking algorithm

void *checknode* (node *v*) {
  node *u*

  if (*promising* (*v*))
    if (*aSolutionAt*(*v*))
      write the solution
    else //expand the node
      for (each child *u* of *v*)
        checknode (*u*)

Checknode

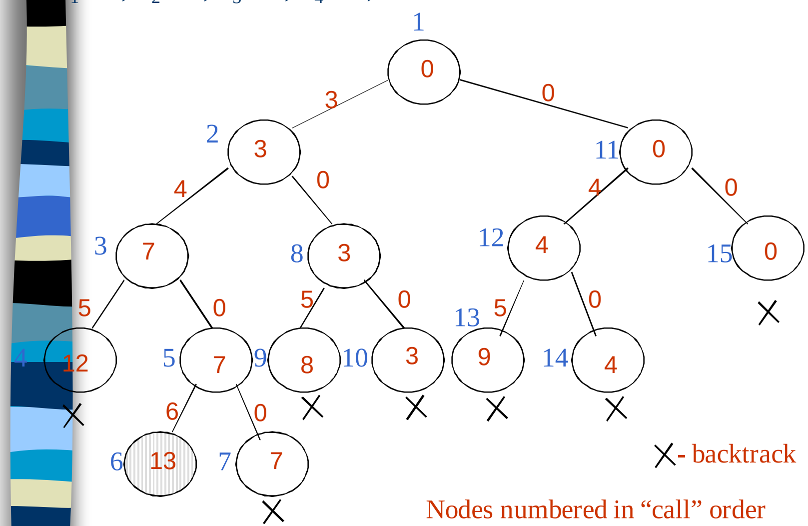
- Checknode uses the functions:
  - *promising*(v) which checks that the partial solution represented by v can lead to the required solution
  - *aSolutionAt*(v) which checks whether the partial solution represented by node v solves the problem.

Sum of subsets – when is a node "promising"?

- Consider a node at depth i
- *weightSoFar* = weight of node, i.e., sum of numbers included in partial solution node represents
- *totalPossibleLeft* = weight of the remaining items i+1 to n (for a node at depth i)
- A node at depth i is non-promising
  if (*weightSoFar* + *totalPossibleLeft* < S)
  or (*weightSoFar* + w[i+1] > S)
- To be able to use this "promising function" the w_i must be sorted in non-decreasing order

A Pruned State Space Tree

$w_1 = 3$, $w_2 = 4$, $w_3 = 5$, $w_4 = 6$; $S = 13$




```
sumOfSubsets ( i, weightSoFar, totalPossibleLeft )
   1) if (promising ( i ))                          //may lead to solution
  2)     then if ( weightSoFar == S )
   3)          then print include[ 1 ] to include[ i ]      //found solution
   4)     else       //expand the node when weightSoFar < S
   5)          include [ i + 1 ] = "yes"                   //try including
   6)          sumOfSubsets ( i + 1,
                         weightSoFar + w[i + 1],
                         totalPossibleLeft - w[i + 1] )
   7)          include [ i + 1 ] = "no"                    //try excluding
   8)          sumOfSubsets ( i + 1,   weightSoFar ,
                                totalPossibleLeft - w[i + 1] )

boolean promising (i )
   1) return ( weightSoFar + totalPossibleLeft ≥ S)  &&
             ( weightSoFar == S  ||  weightSoFar +  w[i + 1] ≤ S )
```

Prints all solutions!

Initial call sumOfSubsets(0, 0, $\sum_{i=1}^{n} w_i$

Backtracking for optimization problems

- To deal with optimization we compute:
  - *best* - value of best solution achieved so far
  - *value*(v) - the value of the solution at node v
  - Modify *promising*(v)
- *Best* is initialized to a value that is equal to a candidate solution or worse than any possible solution.
- *Best* is updated to *value*(v) if the solution at v is "better"
- By "better" we mean:
  - larger in the case of maximization and
  - smaller in the case of minimization

Modifying promising

- A node is *promising* when
  - it is feasible and can lead to a feasible solution and
  - "there is a chance that a better solution than *best* can be achieved by expanding it"

How is it determined?

- Otherwise it is *nonpromising*
  A *bound* on the best solution that can be achieved by expanding the node is computed and compared to *best*
- If the *bound* > *best* for maximization, (< *best* for minimization) the node is promising

Modifying promising for Maximization Problems

- For a *maximization* problem the bound is an *upper bound*,
  - the largest possible solution that can be achieved by expanding the node is less or equal to the *upper bound*
- If *upper bound > best* so far, a better solution may be found by expanding the node and the feasible node is *promising*

Modifying promising for Minimization Problems

- For *minimization* the bound is a *lower bound*,
  - the smallest possible solution that can be achieved by expanding the node is less or equal to the *lower bound*
- If *lower bound* < *best* a better solution may be found and the feasible node is *promising*

Template for backtracking in the case of optimization problems.

```
Procedure checknode (node v ) {
   node u ;

   if ( value(v) is better than best )
        best = value(v);
   if (promising (v) )
        for (each child u of v)
                checknode (u );
}
```

- *best* is the best value so far and is initialized to a value that is equal or worse than any possible solution.
- *value(v)* is the value of the solution at the node.

Notation for knapsack

- We use *maxprofit* to denote *best*
- *profit(v)* to denote *value*(*v*)

The state space tree for knapsack

- Each *node v* will include 3 values:
  - *profit* (v) = sum of profits of all items included in the knapsack (on a path from root to *v*)
  - *weight* (v)= the sum of the weights of all items included in the knapsack (on a path from root to *v*)
  - *upper*Bound*(v)*. *upperBound(v)* is greater or equal to the maximum benefit that can be found by expanding the whole subtree of the state space tree with root *v*.
- The nodes are numbered in the order of expansion

Promising nodes for 0/1 knapsack

- Node *v* is *promising* if *weight(v)* < *C*, and *upperBound(v)*>*maxprofit*
- Otherwise it is not promising
- Note that when weight(v) = C, or maxprofit = upperbound(v) the node is non promising

Main idea for upper bound

- **Theorem**: The optimal profit for 0/1 knapsack ≤ optimal profit for *KWF*
- *Proof:*
- Clearly the optimal solution to 0/1 knapsack is a possible solution to *KWF*. So the optimal profit of *KWF* is greater or equal to that of 0/1 knapsack
- **Main idea**: *KWF* can be used for computing the upper bounds

Computing the upper bound for 0/1 knapsack

- Given node *v* at depth *i*.
- *UpperBound(v)* =

  KWF2(i+1, weight(v), profit(v), w, p, C, n)
- *KWF2* *requires* that the items be ordered by non increasing p_i / w_i, so if we arrange the items in this order before applying the backtracking algorithm, *KWF2* will pick the remaining items in the required order.

KWF2(i, weight, profit, w, p, C, n)

```
bound = profit
for j=i to n
  x[j]=0  //initialize variables to 0
while (weight<C)&& (i<=n)        //not "full"and more items
   if weight+w[i]<=C              //room for next item
       x[i]=1                    //item i is added to knapsack
       weight=weight+w[i]; bound = bound +p[i]
  else
      x[i]=(C-weight)/w[i]  //fraction of  i added to knapsack
      weight=C; bound = bound + p[i]*x[i]
  i=i+1                          // next item
return bound
```

- KWF2 is in O(n) (assuming items sorted before applying backtracking)

C++ version

- The arrays *w*, *p*, *include* and *bestset* have size n+1.
- Location 0 is not used
- *include* contains the current solution
- *bestset* the best solution so far

Before calling Knapsack

```
numbest=0;    //number of items considered
maxprofit=0;
knapsack(0,0,0);
cout << maxprofit;
for (i=1; i<= numbest; i++)
    cout << bestset[i]; //the best solution
```

- *maxprofit* is initialized to \$0, which is the worst profit that can be achieved with positive p_is
- In Knapsack - before determining if node *v* is promising, *maxprofit* and *bestset* are updated

knapsack(i, profit, weight)

```
if ( weight <= C && profit > maxprofit)
  // save better solution
   maxprofit=profit //save new profit
   numbest= i; bestset = include//save solution
if promising(i)
  include [i + 1] = " yes"
  knapsack(i+1, profit+p[i+1], weight+ w[i+1])
  include[i+1] = "no"
  knapsack(i+1,profit,weight)
```

Promising(i)

```
promising(i)
   //Cannot get a solution by expanding node
   if weight >= C return false
   //Compute upper bound
   bound = KWF2(i+1, weight, profit, w, p, C, n)
return (bound>maxprofit)
```

Example from Neapolitan & Naimipour

- Suppose $n = 4$, $W = 16$, and we have the following:

i	p_i	w_i	p_i / w_i
1	\$40	2	\$20
2	\$30	5	\$6
3	\$50	10	\$5
4	\$10	5	\$2

- Note the the items are in the *correct order* needed by *KWF*

The calculation for node 1

maxprofit = \$0 (n = 4, C = 16)

Node 1

a) *profit* = \$ 0

weight = 0

b) *bound* = *profit* + $p_1 + p_2 + (\boldsymbol{C} - 7) * p_3 / w_3$

= \$0 + \$40 + \$30 + (16 -7) X \$50/10 =\$115

c) 1 is promising because its weight =0 < ***C*** = 16 and its bound \$115 > 0 the value of *maxprofit*.

The calculation for node 2

Item 1 with profit \$40 and weight 2 is included

maxprofit = \$40

a) *profit* = \$40

weight = 2

b) *bound* = *profit* + $p_2 + (\boldsymbol{C} - 7)$ X p_3 / w_3

= \$40 + \$30 + (16 -7) X \$50/10 =\$115

c) 2 is promising because its weight =2 < ***C*** = 16 and its bound \$115 > \$40 the value of *maxprofit*.

The calculation for node 13

Item 1 with profit $40 and weight 2 is not included

At this point maxprofit=$90 and is not changed

a) *profit* = $0

weight = 0

b) $bound = profit + p_2 + p_3 + (\mathbf{C} - 15) \text{ X } p_4 / w_4$

$= \$0 + \$30 + \$50 + (16 - 15) \text{ X } \$10/5$

$= \$82$

c) 13 is nonpromising because its bound $82 < $90 the value of *maxprofit.*