# **Chapter 5: Backtracking Algorithms**

## **Introduction**

Backtracking is kind of like **trying to find your way out of a maze**.  You try heading in one direction, and if you hit a dead end, you go back to the last intersection and try a different direction.  In the worst case, you end up trying every possible passage in the maze. Now imagine if half the branches that lead to dead ends had signs on them saying, "Dead end this way".  You could avoid those passages and reduce the amount of time it would take you to find your way out of the maze.

Backtracking algorithms do a similar type of thing.  They try to make decisions as to whether a particular solution is going to be good before they compute it. This is best pictured if we think of the sequence of decisions that must be made to solve a problem as a tree.  One of the leaf nodes is the best solution.  A brute force algorithm would simply search every node in the tree until it found the best solution.  **A backtracking algorithm tries to decide whether a particular branch will lead to the best solution before it gets all the way to the leaf nodes.**

An example of this would be the 0-1 Knapsack Problem.  We all remember the dynamic programming solution to this problem.  There is also a backtracking solution.   We can picture the solution tree for a problem with 3 items.  The root is before we pick any items.  If we pick item 1, we go to the left child of the root.   If we do not pick item 1, we go to the right child of the root.  If we can find some indicator that a particular branch is no good, we can avoid computing that possible solution. We'll talk in more detail about the backtracking solution to the 0-1 Knapsack Problem a little later.

If you think back to talking about graphs, you can see that backtracking is actually performing a depth-first search on the tree.  **It isn't a complete depth-first search, because we try to avoid visiting every node in the tree, but the general approach is depth first.**  If you remember way back to talking about tree traversal, the preorder traversal is actually one way to perform a depth-first search on a tree.

## **N-Queens Problem**

**Backtracking is used to solve problems where a sequence of items is chosen from a set of possible items such that the sequence satisfies some criteria**.

The 0-1 Knapsack Problem is clearly this type of problem.  Finding your way through a maze is also the same type of problem.   You must pick a sequence of directions to go such that the sequence ends at the exit to the maze. The N-Queens problem is also the same type of problem.  The idea is to place N queens on a board that is N by N squares, such that none of the queens could capture another queen.  Queens can move horizontally, vertically, or diagonally any distance. The sequence to be found are the N positions for the queens.  The set from which those positions are chosen is the N2 possible positions on the board and the criteria the sequence must satisfy is that none of the queens can capture another.

To solve this problem with backtracking, we have to do several things.  First, we must visualize a tree model for our solution space (the set of all solutions).  We can simplify our tree by recognizing that **no two queens can be on the same row** (they can move horizontally any distance).  **So, the children of the root will show the position of the queen on row 1.**  For an N by N size board, there will be N children of the root. Let's consider the 4-Queens problem, and draw part of the tree representing the solution space (we start with the 4-Queens problem, because there is no solution for smaller values of N). Second, we must come up with ways of eliminating branches before we actually traverse them. **In the N-Queens problem, one way of eliminating a branch is if it would put two queens in the same column** (queens can travel any distance vertically).  So, in our sample tree we have drawn, we know that if we choose **1,1** as the position for the **queen in row 1**, then **we do not need to traverse it's child that contains 2,1.** That would put two queens in the same column, so the entire portion of the tree beneath that 2,1 node does not need to be traversed.

**Another way of eliminating branches is to recognize that no two queens can be on the same diagonal.**  So, if we choose **1,1** for the queen in row 1, then we do not need to evaluate **2,2** for the queen in row 2. Using these two rules, we can eliminate quite a bit of the solution space.  **When we eliminate a node and its branch, we say that the node was nonpromising, meaning that node could never lead to the solution.  A node is promising if it may lead to the solution.**When we identify a node as **nonpromising**, we go back to the parent and try another branch.  Eventually, we will get to one or more leaf nodes and have several candidate sequences to evaluate (you can reach a solution before getting to a leaf node if the sequence thus far satisfies the problem criteria).  **In the N-Queens problem, if we get to any leaf node, we have our solution because the problem is just to find one way to put N-Queens on the board.**

When we eliminate a branch of the tree because a node is **nonpromising**, we say that we have **pruned** the solution tree.  **This pruning is one of the things that makes backtracking more efficient than a simple brute force algorithm**.  Designing your tree to eliminate obviously illegal solutions is another way to make it more efficient.

Page 209 in your book shows how the backtracking algorithm checks spaces on the chessboard.  Page 215 has a table that shows the savings between checking every node in the tree (algorithm 1 in the table) and backtracking.  You can see that the higher the value of N, the more the savings by backtracking.

## **General Algorithm**

The general algorithm used for backtracking is as follows:

**checknode**(node v)  
{  
    if (v is promising)  
    {  
        if (*v is the end of a solution sequence*)  
            **display the solution**  
        else  
        {  
            for (each child of v)  
                **checknode**(next child)  
        }  
    }  
}

Like many of our problem solving techniques, **this is defined recursively**.  When actually implementing a backtracking algorithm, you'll have to determine whether to implement it recursively or iteratively. **Also note that it isn't necessary to actually construct a tree for the algorithm to traverse.  For example, in the N-Queens problem, the tree exists because of the recursive nature of the function calls.  We say that the tree exists implicitly in the algorithm itself.**

Page 212 in your book gives the actual algorithm for the N-Queens problem.The solution space tree will always contain at least an exponential number of nodes.   **The big Oh of most backtracking algorithms is O(2n).**  **Backtracking algorithms will run faster than other O(2n) algorithm that don't prune the tree.**  In the worst case, a backtracking algorithm may be no better than an intelligent brute force algorithm.

## **Graph Coloring**

The m-Coloring problem asks us to find all the ways that we could use m colors to color nodes in an undirected graph such that no adjacent nodes have the same color.  For example, given the following graph:

V(G) = {1, 2, 3, 4}  
E(G) = {(1,2), (1,4), (1,3), (2,3), (3,4)}

Is there a solution to the 2-Coloring problem?  In other words, how many ways can we use 2 colors to color nodes such that no adjacent nodes have the same color?  How about the 3-Coloring problem? For small values of m and a small number of nodes, we can visually solve this.   **However, as m and n get past 3 or 4, the number of possible solutions is too large.**  This problem is important in the coloring of maps.  You don't want adjacent countries to be the same color, and you may want certain countries to be certain colors. We convert a map to a graph in order to solve this problem for the map.  The map is converted to a planar graph -- a graph where no edge crosses another edge.

**Since the solution to this problem is a sequence of color choices (one for each vertex) that satisfy a given criteria (no adjacent vertices have the same color), we can use backtracking to solve it.**  Our first step is to visualize the solution space.   We can do this by having the children of the root be the color chosen for vertex 1.   The children of vertex 1 will be the color chosen for vertex 2, and so on.

The next step is to decide how we can eliminate branches (i.e., identify nonpromising nodes).  The criteria gives us **the fact that adjacent vertices cannot be the same color**, so we'll use that to eliminate branches.

Let's see what the tree would look like for the 3-Coloring problem for our graph above. We'll start with the root, and eliminate branches as we go, just like the backtracking algorithm. Page 229 in your book has the complete algorithm for the m-Coloring problem.  You can see that it follows the basic backtracking algorithm. An interesting side note is that while there are some graphs for which no solution to the 3-Coloring problem exists, you can find solutions to the 4-Coloring problem for any graph.  So, any map only needs 4 colors.

## **Sum-of-Subsets Problem**

The sum of subsets problem is a bit abstract, but relates back to the 0-1 Knapsack Problem.  In the sum of subsets problem, **we have n positive integers** (a single of these integers is wi).  **We also have a positive integer W**.  **The goal is to find all subsets of wi that sum to W.**  For example, let's say that our wi values are 5, 3, 2, 4, 1 and our W is 8.  We could make 8 out of 5+3, or 5+2+1, or 4+3+1.

**So, the problem is to find a sequence of wi values that satisfy the criteria of summing to W.  This is clearly the same sort of problem that backtracking is used for, so we can start by visualizing a tree for the solution space.**

In this tree, the children of the root will indicate whether the w1 has been picked or not.   **The left child indicates that we picked w1, while the right child indicates that we did not pick w1.**

In addition, we can make our life easier later on if we sort the weights in ascending order before visualizing our tree.  Let's write part of the tree for our weights from the previous example.

Next, we come up with **a way to eliminate branches**.  **Since the values of wi increase as we go down the tree, if we are at level i and adding the value of the weight at level i + 1 would cause the sum to be larger than W, we know that the node at level i is nonpromising.**  For example, imagine we are at a node that represents having picked a weight of 8, and the next weight to be picked is 10.  If W is 12, then picking the next weight would cause the sum to be larger than W.  Since the weights can only increase as we go down the tree, there are no other weights we could add to 8 that would cause it to equal 12, so we can eliminate the entire branch that starts with picking 8.

**We have another way of eliminating branches.**  **If the sum of weights so far, plus the sum of picking all the remaining weights, would be less than W, then we know there is no solution in that branch.  Remember, the goal is to get a sum of exactly W.**   If picking all the remaining weights is less than W, then leaving out any of those weights is still less than W.

Page 223 in your book has an example of a tree for a given set of weights.  This tree has been pruned according to the rules we just discussed.  There is only one solution, found **at the leaf node 13**.  **Note that for this problem, solutions do not have to be at leaf nodes.  If we arrive at a sum of W in the middle of the tree, that's a valid solution as well.**

## **0-1 Knapsack Problem**

We said earlier that the 0-1 Knapsack Problem had a backtracking solution, so let's look at that. **The problem is to choose a sequence of items such that the profit is maximized and the weight of the items is not above W.  We're choosing a sequence according to a constraint, so backtracking is suitable.**

The tree we build for this problem is similar to the tree for the sum of subsets problem.  **It is a binary tree where choosing the left child means that you pick an item, and choosing the right child means that you do not pick an item.**

Our previous backtracking algorithms found all solutions.  **In the 0-1 Knapsack Problem, we have to find the best solution.** **So, we still have to find all solutions, but we also have to keep track of which one is best (maximum profit).**  In fact, we usually don't even know if a particular node is a solution or not. **We simply know whether it is better than other candidates.** For example, the **maximum profit** **might come from items whose total weight is less than W.** **We won't know that was the best solution until we eliminate the other possibilities.**

**In optimization problems, we change the meaning of a promising node.**  **Before, a node was promising if it might be part of the solution.**  **In optimization problems, a node is promising if its children might be part of the solution** (if we get to it, the node itself is assumed to possibly be part of the solution).  Here is the general backtracking algorithm for optimization problems.

checknode (node v)  
{  
    if (*value at v better than previous best value*)  
        **best\_value** = **value\_at\_v**  
  
    if (v is promising)  
    {  
        for (each child of v)  
            checknode (next child)  
    }  
}

**Similar to the sum of subsets problem, we order the tree.**  **In the 0-1 Knapsack Problem, we order the tree in descending order by the profit/weight ratio**.  **This makes our decisions to eliminate branches easier.**  Let's draw part of the tree for three items, with profits of $40, $30, $50 and weights of 2, 5, 10, and a capacity of 10.   We can indicate in the node itself the total profit so far and the weight of items in the knapsack.

**Now we need to decide how to eliminate branches.**  **One obvious way (rule 1) is to use the weight constraint (the weight of items must not exceed W).**  **So, we know that if the weight of items chosen is greater than W, we can eliminate that branch.**  Further, if the weight of items chosen so far is equal to W, we know that we do not have to check that node's children (because any additional weight would cause it to exceed W).  So, **we say that a node is nonpromising if the total weight thus far is greater than or equal to W.**

**Another way to eliminate branches is to use the greedy algorithm from the Fractional Knapsack Problem to determine an upper bound on profit.**  The profit realized in the Fractional Knapsack Problem will always be equal to or greater than the profit in the 0-1 Knapsack Problem.  So, if we are at a node with a total profit so far, and **we calculate the amount of profit we could get by greedily choosing from the remaining items, we end up with some value that represents an upper bound on profit if we count this node as part of the solution.**  **If that upper bound is less than the best solution we have picked so far (bound < maxProfit), we do not need to go down that branch.**  None of the solutions in that branch will be better than the solution we have already found. See the example in page 237 of the textbook.

## **Efficiency of Backtracking Algorithms**

We've said that **most backtracking algorithms are exponential**.  However, backtracking algorithms are extremely dependent on the data given to them.  **For example, with one set of data, the 0-1 Knapsack Problem might have to complete a depth-first search of the tree without eliminating any branches.**  With another set of data, the 0-1 Knapsack Problem might eliminate most of the branches.  This is because the decision to eliminate branches depends on the data you are processing.

**So, a given backtracking algorithm might run extremely quickly on one set of data and extremely slowly on another set of data**.  One way to compare backtracking algorithms to other types of algorithms for the same problem (for example, the dynamic programming and the backtracking algorithms for the 0-1 Knapsack Problem) is to simply run both algorithms a large number of times on random sets of data.  This will give you a good idea as to which one is more efficient.

For the 0-1 Knapsack Problem, the backtracking algorithm is, in general, more efficient than the dynamic programming algorithm.  However, for any single set of data, the dynamic programming algorithm might process it faster.