Design and Analysis of Algorithms – CSE 101
Backtracking Algorithms : Design and Analysis

Search Problems The problems backtracking helps with are of the following form: If possible, find a solution of the following format that satisfies the following constraints. For example, in the n-queens problem, the solution is a placement of queens on the chess board. One format for presenting such a solution is: an array S, indexed by rows, with values from 1 to n. $S(i)$ is the column where there is a queen in the i'th row. The constraints would be: no two queens are in the same column, ($S(i) \neq S(i')$ for each $i \neq i'$) or in the same diagonal: $S(i) - S(i') \neq i - i', i' - i$ for any $i \neq i'$. The format should have a large but finite number of possible values, e.g., a small number of possible values in each position of an array means an exponentially large number of total values. It should be easy to verify when a constraint is broken.

Basic Design Steps

1. View finding a solution as making a series of *choices*. The format for candidate solutions is a big clue as to how to do this. For example, if the format is an array, for each index, we must choose the value of that array element. So in the queens example, the indices are rows. For each row, we must choose where to put the queen in that row.
2. Pick a single choice, and do a case analysis of the options. What options are possible at each choice? How do the constraints and previous choices limit the options? For example, in the queens example, for each row we have to choose a place to put the queen, out of n possible places. However, columns or diagonals used previously have been ruled out, so we really have somewhat fewer choices.
3. How does each option affect the problem? Write a recursive definition of the search as: In some order, try each option for the next choice. Recursively solve the simplified problem, given that option. If the search succeeds, output the results. Otherwise, "backtrack" and try the next option. If all options fail, return "search failed".

   Sometimes, once a choice is made, the problem is similar to the original one, on a smaller instance. Then this can be just recursive calls of the same algorithm. If it isn't, you can still use back-tracking by having a recursive procedure with a partial solution as a parameter. The recursive procedure calls itself with more and more of the solution filled in. In the queens example, we could carry forward a list of places that queens have been placed already.
4. Think of ways of pruning the search. Can you recognize when a partial solution violates a constraint? Then terminate without recursing. Are there choices that are forced, i.e., have only one option left? Then add that option to the partial solution. Is there symmetry that can be used, so that not all choices are necessary to explore? Is one option provably better than the others in some circumstance? Then there is no need to consider the other options.
5. Having answered these questions, you can fill in the following "generic format" for backtracking. Sometimes PartialSolution isn't necessary, if the original problem has "self-similarity".

   A generic format for a back-tracking algorithm might be as follows:

   (a) Backtrack(Problem, PartialSolution)

   (b) If PartialSolution is not consistent with all the constraints, return NUL

   (c) If all choices have been made, and PartialSolution is consistent with all the constraints, return PartialSolution

   (d) If there is a choice left with no consistent options, return NUL

   (e) (Optional Step): If there is a choice left with only one consistent option, add that to PartialSolution and recurse.

   (f) (Optional Step): If there is a choice with an option provably better than the others, add that to PartialSolution and recurse.

   (g) Pick a choice *choice* that needs to be made. Say $opt_1, ...opt_k$ are the possible options for the choice. Then for each, set $PartialSolutions_i$ to be $PartialSolution$ with opt_i added for the choice. For each $1 \leq i \leq k$, let $Solution_i = Backtrack(Problem, PartialSolution_i)$.

(h) Return $Best(Solution_1, ..Solution_k)$, where Best returns any non-null answer for search, or the non-null solution with highest optimality for optimization.

6. Example : n-Queens
   (a) Backtrack(n, PartialQueens: set of pairs (i, j) with $1 \leq i, j \leq n$ meaning places where queens are located)
   (b) If PartialQueens has two queens in the same row, column, or diagonal, return NUL
   (c) If PartialQueens has size n, return Partial Queens.
   (d) For each row i that doesn't have an element in PartialQueens, let $Options(i)$ be the set of columns j where (i, j) isn't on a column or diagonal with a point in PartialQueens.
   (e) If $Options(i)$ is empty for some row i, return NUL
   (f) If $Options(i) = \{j\}$ for some column j, add (i, j) to PartialQueens and return $Backtrack(n, PartialQueer$
   (g) Else: Pick i with no queen in the i'th row.
   (h) For each $j \in Options(i)$:
   (i) begin{for}
   (j) Let $Solution = Backtrack(Problem, PartialQueens \cup \{(i, j)\})$.
   (k) IF $Solution \neq NUL$ return $Solution$.
   (l) end{for}
   (m) Return NUL;

7. Analysis: A crude upper bound is as follows. Each recursion level, we make at least one choice, so the number of recursion levels is at most the number of choices. The number of recursive calls at a single level is at most the number of options. Thus, we can bound the total number of recursive calls by $(\#choices)^{(\#optionsperchoice)}$. If this is exponential, and the time to do the loops is polynomial, the recursion will be bottom-heavy, and the number of such calls dominates the total time. So in the queens problem, we have n choices (where to put a queen in a row), each with at most n options, for a total of at most time n^n. Actually, we can usually improve this, by noting that sometimes we must force more than one choice per level, or that the maximum number of options becomes smaller as the algorithm progresses. So for example, the k'th choice is n queens has at most $n - k$ options, since we eliminate at least one column each time. (In fact, it probably has between $n - 2k$ and $n - 3k$ options, because of diagonals.) This gives an upper bound of $n(n-1)(n-2)...2 \cdot 1 = n!$ on the time, but as mentioned above, this is still probably nowhere near tight because it does not count diagonals.