8.4 Making Recursive Backtracking Practical

The space of partial solutions is exponential in size ($O(b^h)$), where b is the branching factor and h is the length of the longest solution path), making the worst-case performance of recursive backtracking algorithms exponential. What can be done?

There are two main approaches, which can be combined. The first is to formulate the algorithm to reduce b and h, thus reducing the size of the search space.

- Address the branching factor.
  To reduce the branching factor, reduce the number of alternatives for each decision. For subset problems, for example, prefer the process input approach because "include this element in the solution or not" has a branching factor of 2 while the produce output approach ("which is the next element to take?") has a branching factor of up n.

  Sometimes specific aspects of the problem can be exploited to allow for a reformulation of the series of choices to reduce the branching factor. For example, for n queens, the observation that there is exactly one queen per column (and row) means that instead of the series of choices being where to place each queen on the board, it can be where to place each queen in its column, reducing the branching factor from n^2 to n.

- Address the longest path length.
  If the output is a subset of the input, for example, prefer a produce output approach, where the number of decisions made is the size of the solution, to a process input approach, where the number of decisions made is the number of input elements.

The second approach is to stop exploring a branch before a complete solution or dead end is reached i.e. pruning. Tactics include:

- Impose an ordering on the series of choices to limit redundant paths.
  A strategy for produce output "find a subset" problems is simply to order the elements in some way, and then only look forward in the ordering for the next element to include. All legal subsets can still be formed, but each subset can now only be formed by a single series of decisions, eliminating redundant ways of ending up with the same solution.

- Store solutions for repeated subproblems.
  This avoids exploring the same parts of the search space more than once, and can be very effective (even reducing the runtime to polynomial or pseudo-polynomial) if the number of unique states is small and there are many paths to each state.

- Explore branches most likely to lead to a solution first.
  (Next possibilities do not need to be explored in the same order for every subproblem.) This is only applicable for "find a solution" tasks, where the algorithm stops after the first solution is found.

- Avoid exploring branches which don't contain the solution.
  This could be because all options lead to a dead end (no possible solution from this point) or because pruning now won't eliminate all of the possible solutions). The latter strategy is only applicable to "find a solution" and "find the best solution" tasks. A variation of the strategy for optimization problems is to prune if the branch can't lead to a better solution than the best known so far, since then it also can't contain an optimal solution.

Effective strategies for many of these tactics tend to be highly problem-specific, though there are exceptions. For example, as noted for produce output "find a subset" problems, ordering the input elements is an effective way to eliminate many redundant paths. Also, pruning branches for next possibilities that lead to invalid partial solutions is always applicable.

In all cases, the decisions about which branches to explore and in which order must be local decisions — one cannot consider specific possible future scenarios of how the choice would play out. Furthermore, the decisions need to be relatively cheap to make since they will be made for every subproblem.

While the search space size (or how much of it is explored) is the dominating factor in the runtime of recursive algorithms, smaller factors can also have a significant effect. A key point here is to efficiently generate subproblems and their partial solutions — avoid copying! Instead, adopt a modify/restore approach. A clever representation can also help. For example, if the input items can be considered in a particular order, specifying which items are involved in the subproblem only means updating an index in the array instead of creating a new subset.

8.5 Branch and Bound

In optimization problems, branches can be pruned if no solutions in the subtree are good enough to be optimal — if the best solution that can be derived from the current partial solution is not better than the best solution found so far, there's no point in continuing with the current partial solution and the whole branch can be pruned. The challenge is that the cost of the best possible solution derivable from a given partial solution is not generally computable without actually searching the subtree and generating all those actual solutions.

Branch and bound works by having a *bound function* which returns an estimate of the best possible cost of any solution derived from the current partial solution. If the estimate is not better than the current best solution, the branch can be pruned.

The bound function must satisfy several key properties:

- It must be *safe*, that is, optimistic about the quality of the solutions.
  If the estimate is too optimistic, it claims that better solutions are possible than actually are — and the only consequence is wasting time searching that branch unnecessarily. However, if the estimate is too pessimistic and it claims that solutions are worse than they actually are, the consequence is possibly missing the optimal solution if the branch is pruned based on that pessimistic estimate.
- It must be based only on the partial solution / current state.
  We cannot consider how future choices play out — we're trying to avoid searching the subtree...
- It must be relatively efficient to compute.
  The bound function is evaluated for every subproblem, so while the effect on the big-Oh may not be so significant compared to the exponential number of subproblems, even a small amount multiplied by a big number has a significant impact on the actual time the algorithm takes.

The cost of the best solution so far can be initialized to ∞, but there's more potential for pruning if it can be initialized to a value closer to the optimal. As with the bound function, this will necessarily be an estimate. This best solution estimate function must satisfy two key properties:

- It must be *safe*, that is, pessimistic about the quality of the solutions.
  If the estimate is too pessimistic, it claims that the best solution is worse than it actually is — and the only consequence is wasting time searching branches that could have been pruned because they don't contain the optimal solution. However, if the estimate is too optimistic and it claims that the best solution is better than it actually is, the consequence is missing all of the solutions (including the optimal) because no branch is thought to be good enough to be worth exploring.
- It must be more efficient to compute than searching.
  Unlike the bound function, the best solution estimate function is only computed once — to initialize the best solution so far — so it can be more expensive than the bound function, but it still must be faster than the search problem that we're trying to speed up.

The goal of branch and bound is to prune enough of the search space to make a recursive backtracking approach practical to run. How successful this is depends on three factors:

- The quality of the bound function.
  A tighter bound means more and earlier pruning, but the function can't underestimate the true cost of the solutions in that subtree.

- The initial best solution so far value.
  Closer to optimal means more and earlier pruning, but the function can't overestimate the cost of the actual best solution.

- The time to compute the bound function.
  Better quality estimates are generally more expensive, and the bound function must be computed for each subproblem, whether or not the branch ends up being pruned. If the function is too expensive, there won't be enough work saved by pruning to make computing the bound worthwhile.

Good bound and best solution estimate functions tend to be highly problem specific. However, there are a few general tactics that may serve as starting points. For a bound function:

- the value of the partial solution so far + best single choice × the number of choices left

- the value of the partial solution so far + best single next possibility × the number of choices left
  (only safe if all choices are available at each stage)

- the value of the partial solution so far + greedy solution from that point for a relaxed version of the problem
  (only safe if a better solution is possible for the relaxed version, and the relaxed version is solvable with a greedy algorithm)

- consider a trivial bound and what is over/undercounted by the trival bound

For the best solution estimate, a greedy solution can be a good strategy. Another strategy, which can be combined with the initial estimate or used instead of making an initial estimate, is to focus on finding a good solution early by searching those branches first. One option is to use a modified depth first search where the next possibilities at each step are ordered by the most promising (based on the bound function) first. Another option is to use best first search, a variation of breadth first search where the most promising subproblem is handled next. (The BFS queue is replaced by a priority queue ordered by the bound function value.) The downside of a BFS-based approach is the potentially exponential size of the queue.