Chapter 7

Recursive Backtracking in Practice

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 three main strategies, which can be combined: reducing the size of the search space, eliminating portions of the search space from consideration (known as *pruning*), and avoiding repeated subproblems. This chapter will focus on the first two, leaving avoiding repeated subproblems to chapter 8.

7.1 Reducing the Search Space

Reducing the search space involves formulating the algorithm to reduce b and/or h.

The length of the longest solution path (h) is determined by the length of the solution for produce output algorithms and the size of the input for process input algorithms, while the branching factor (b) stems from what choice is being made and how many alternatives there are. As a result, there is often more of an opportunity to reduce b, but it is still worth considering both.

Some common avenues —

- *Consider process input vs produce output.*

  Reducing the length of the longest solution path (h) favors the produce output strategy if the output is typically much smaller than the input, such as in "find a subset" problems where the resulting subsets tend to be small.

  Reducing the branching factor (b) favors the process input strategy if there are substantially fewer alternatives than the number of items, since produce output often needs to choose the next item from amongst all remaining items. For example, for "find a subset" problems, the branching factor is 2 (choose or not choose) in the process input approach but $n-k$ (the number of remaining items) in the produce output approach.

- *Limit redundant paths.*

  Redundant paths exist when only the set of choices made matters rather than the order in which they are made, such as in a "find a subset" problem — only the subset matters, not the order in which elements were added.

  - *Impose an ordering on the series of choices.*

    For example, for a produce output "find a subset" algorithm, the elements can be ordered and the task of finding the next element from those not yet in the subset can be reduced to finding the next element in the list — the number of alternatives is reduced from $n-k$ (everything not yet in the subset) to the rest of the list ($\leq n-k$, and decreasing much faster than $n-k$ if most elements are not in the subset).

– *Leverage something problem-specific.*
This, of course, is impossible to produce general advice for. As an example, observe that in the n queens problem, the fact that queens placed in the same column can attack each other so that n queens on an $n \times n$ board means there will be exactly one queen per column. Thus, instead of considering all n^2 board positions as possible locations for the next queen, it is only necessary to consider the n rows within the current column — the algorithm is now a series of "which row?" choices rather than "which position?".

7.2 Pruning

Only so much can be done to limit the branching factor and the length of the solution paths — it is also necessary to avoid exploring the whole search space. The idea is to stop exploring a branch when it is safe to do so — because no (legal) complete solution contains the current partial solution, or no complete solution containing the current partial solution is the desired solution, or pruning the current branch doesn't eliminate all of the desired solutions.

Some common avenues —

- *Don't pursue dead ends.*
  Stop exploring branches where there are no (legal) complete solutions containing the current partial solution. This is easily achieved by only considering legal alternatives, and is already a routine part of a recursive backtracking algorithms.
- *Don't pursue solutions that are already bad.*
  For optimization problems, once one solution has been found, you can stop exploring branches where the current partial solution is already worse than the solution already found — extending the current solution will not make it better.
- *Explore branches most likely to lead to a good solution first.*
  (Next possibilities do not need to be explored in the same order for every subproblem.) While this isn't in itself a pruning technique, it can enhance pruning opportunities by establishing a better solution-so-far.
- *Exploit symmetries.*
  For example, for any solution to the n queens problem, another can be found by flipping the board around the horizontal axis — that is, swap the top and bottom rows of the board, the next-to-top and next-to-bottom rows, and so forth. As a result, it is only necessary to consider the top (or bottom) four rows for the first queen's placement — skipping the other rows means some valid solutions won't be found, but there's at least one left (if there's one at all) and we only need one...

In every case, the decision about whether or not to prune must be a local decision — 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.

7.3 Branch and Bound

In optimization problems, there isn't any point in continuing to explore branches where none of the complete solutions stemming from the current partial solution will be better than the best complete solution known so far. *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. In the trivial case, where the bound is simply the cost of the partial solution so far, this strategy reduces to the pruning strategy described above where a branch is abandoned if the partial solution is worse than the current best complete solution. The advantage of branch-and-bound comes in identifying better estimate functions which allow for earlier pruning.

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* local*, that is, 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 ideas 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. A local strategy is to consider the alternatives for each choice according to some criteria, such as the value of the bound function. A global strategy is to employ best-first search and the A* heuristic.

7.4 Best-First Search and the A* Heuristic

Pruning seeks to avoid exploring sections of the search space by skipping branches that don't have the desired solution; another strategy is to find the desired solution quickly so that then the search can stop.

Best-first search is a variation of breadth-first search which uses a priority queue instead of a queue, with the elements in the priority ordered so that the most promising ones are first. Dijkstra's algorithm is best-first search applied to shortest paths — instead of putting each vertex in a queue (breadth-first search) or a stack (depth-first search, non-recursive version) as it is discovered, it goes into a priority queue ordered by the distance from the start. The advantage of Dijkstra's algorithm over breadth-first or depth-first search for shortest paths is that because the priority queue is ordered by the length of the shortest path to that vertex, once a vertex comes out of the queue, there aren't any shorter not-yet-found paths because such a path would have to go through a vertex still in the queue and all of those are farther from the start than the current vertex.

Recursive backtracking is depth-first search applied to the space of partial and complete solutions. An iterative version can be formulated using a stack:

```
push the initial subproblem onto the stack
repeat as long as the stack isn't empty
  pop the top of the stack
  if the solution is complete, ...
  otherwise for each legal alternative for the next choice,
    push a copy of subproblem with that alternative added to the partial solution
     onto the stack
```

The ... depends on the task — for a "find a solution" task, a solution has been found and the loop can end immediately, while for a "find the best solution" task, the new solution would be compared to the best-so-far and the best-so-far updated as appropriate.

Best-first search replaces the stack with a priority queue ordered (from best to worst) by the cost of the partial solution so far —

```
add the initial subproblem to the PQ
repeat as long as the PQ isn't empty
  remove the first element from the PQ
  if the cost of the partial solution is greater than the cost of the best solution
   so far,
    break
  otherwise if the solution is complete,
    update the best-so-far with the better of it and the current solution
  otherwise for each legal alternative for the next choice,
    add a copy of subproblem with that alternative added to the partial solution to
     the PQ
```

As with pruning only when the cost of a partial solution is worse than the cost of a complete solution, best-first search can end up exploring most of the search space. The *A* heuristic* incorporates a bound function so that the priority queue is ordered by the cost of the best complete solution involving the current partial solution — this means that the most promising partial solutions are explored first, not just the shortest ones. This bound function needs to have the same properties as the branch-and-bound bound function: it must be safe, local, and relatively efficient to compute.

The big advantage of best-first search (with A*) is the potential for finding a solution much faster than pruning would. However, a significant drawback to queue-based searches (like best-first and breadth-first search) is space — remember that the (priority) queue potentially has to hold every subproblem at depth d from the start, making for b^d subproblems. By contrast, depth-first search only has to keep track of one subproblem per level at a time.

A* often works well in practice, but if space is a problem, there are other algorithms that seek to balance computation time with storage space.