A short note on Branch and Bound

David Helmbold, Spring 2005

This document reviews Backtracking and Branch and Bound. Both are heuristics that attempt to quickly perform exponential searches. Although can dramatically decrease runtime on average, it is usually not too hard to find particular instances where they don’t speed things up (or even slow it down). Both methods work by consider the search space as a tree and perform a structured exploration of the tree. See Chapter 9 of Fundamentals of Algorithmics by Brassard and Bratley for more on these topics (two copies are on reserve in the Science Library). We will start by describing Backtracking, and then move on to Branch and Bound.

**Backtracking**

Backtracking is appropriate for a combinatorial problem were we are interested in any solution (or if no solution exists).

A classic example of Backtracking is boolean CNF (conjunctive normal form) satisfiability. A CNF formula is an ”and-of-ors” like \((v_1 \lor v_2) \land (\overline{v_2} \lor v_3)\)

Given a boolean formula over some variables \(v_1, v_2, \ldots, v_n\) find an assignment of ”true” or ”false” to the variables making the formula true (or report that no such truth assignment exists).

Here we have a space of candidate solutions (the \(2^n\) ways of of assigning each variable either ”true” or ”false”) and are attempting to find a feasible solution that satisfies the conditions. A brute force technique might examine each of the \(2^n\) candidate solutions in turn to see if that solution is feasible, and is likely to be computationally complex.

An alternative is to consider a tree containing the solutions are the leaves and ”partial solutions” at the internal nodes. Recall that a solution is an assignment of true (1) or false (0) to each of the \(n\) variables. Thus a solution can be represented as list of \(n\) 0’s and 1’s (the first bit in the list is the assignment to \(v_1\), and so on). Consider building up a solution assignment by assignment. Initially we have the empty list [], then if we assign \(v_1\) true then we get the vector [1], and if we also assign \(v_2\) false we get the vector [1, 0] and so on. We can construct a binary tree out of these lists (or partial solutions) where the root of the tree is labeled [], the left children of the root is list [0] and the right child is list [1]. In general, if \(L\) is the list at some node then the lists \(L0\) and \(L1\) (here I am using \(L0\) to mean the list created by concatenating 0 onto the end of list \(L\)) are the lists for the left and right children respectively. With this tree structure, a pre-order traversal printing out the leaves enumerates all of the candidate solutions.

The big insight for backtracking is that sometimes a whole subtree can be eliminated with a single test. For example, if the boolean formula is \((v_1 \lor v_2) \land (...)\) and both \(v_1\) and \(v_2\) are false, then the first term is false and the formula cannot be satisfied no matter what values the other variables take on. Therefore once we get to the list [0, 0] we can shortcut exploring the whole subtree rooted at the node labeled [0, 0] and backup to try another list (probably [0, 1]). In general, whenever the algorithm can tell that the subtree rooted at the current node cannot contain a feasible solution, then you backup to the current node’s parent and try another child. If all of the parent’s children have already been tried, then you backup to the grandparent, etc.
Exercise: Consider the boolean formula \((v_1 \lor v_2) \land (\overline{v_2} \lor v_3)\). Draw the tree of partial solutions for 3 variables. Which solution is found first by the backtracking algorithm?

Exercise: Consider the boolean formula \((v_1 \lor v_2) \land (\overline{v_1} \lor v_2 \lor v_3) \land (v_1 \lor \overline{v_2} \lor v_3)\)

Use backtracking on the tree of partial solutions to verify that this CNF has no feasible solution.

Backtracking can also be used for other problems, like the 0-1 Knapsack problem, the N-queens problem (place N queens on an \(N \times N\) chessboard so that no two queens are on the same row, column, or diagonal), graph k-coloring (find an assignment of the colors \(\{1, 2, \ldots k\}\) to the vertices so that adjacent vertices are given different colors), and many others. The trick is to find a tree of partial solutions so that for many partial solutions (like when two queens are in the column or two adjacent vertices are given the same color) there is no completion of the partial solution that is feasible.

Backtracking can stop when it finds any feasible solution, or it can continue if a listing of all feasible solutions is desired.

Branch and Bound

Branch and bound uses a similar tree structure as backtracking, but the search process is more like a breadth first search than a depth first search. In fact, it might be more accurate to call branch and bound a "best-first-search".

Recall the (0,1) knapsack problem:

Given a set of items numbered 1 to \(n\) with weights \(w(i)\) and values \(v_i\) and a capacity \(C\), find a subset of the items \(S\) with \(\sum_{i \in S} w(i) \leq C\) and \(\sum_{i \in S} v(i)\) as large as possible.

The utility of an item \(i\) is \(v(i)/w(i)\). For convenience we assume that the items are ordered by decreasing utility, so if \(1 \leq i < j \leq n\) then \(v(i)/w(i) \geq v(j)/w(j)\).

As in backtracking we can view partial solutions as lists indicating which of the first items (those with highest utility) are taken or left. Complete solutions are \(n\)-vectors indicating for every item whether that item is taken or left. Each partial solution is feasible if the items taken so far have total weight at most \(C\), and infeasible otherwise. We also have the tree of solutions with the empty list at the root, the complete solutions at the leaves, and each child represents an extension of the parents partial solution to the next item.

The extra ingredient for branch-and-bound is a bounding function for partial solutions that overestimates of the value (or an under estimate of the cost if the problem is to minimize cost) for the best feasible complete solution that can be built starting from the partial solution.

For the knapsack problem we can a bounding function based on the best utility of the remaining (unconsidered) items. For example, if \(C = 20\) and a partial solution has taken items with total value 15 and total weight 10, and the next unconsidered item has a utility of 1 then every feasible extension of the partial solution will have value at most 25 (the 15 already taken, and at most 1 unit of value for each remaining unit of capacity). There might not be a feasible extension having value 25, but no feasible extension of the partial solution can have value more than 25.

We can now describe how the branch-and-bound algorithm works. It uses a priority queue of partial solutions where the priority of a partial solution is the value of the bounding function on that partial solution. For this problem we are trying to maximize the value so larger bounds have more priority (in a minimization problem, smaller bounds on the cost
would have more priority). Initially the priority queue contains just the empty solution (at the root of the tree) with value of its bound.

Each iteration of the algorithm first extracts the (partial) solution with the best bound from the priority queue. If that is a complete solution, then it is returned as an optimal solution for the problem. All the children of that partial solution are created, and any infeasible children are discarded. Any children that are complete solutions have their bounds set to their actual values, and bounds for children that are incomplete solutions are computed using the bounding function. All of the feasible children are then inserted back into the priority queue. The algorithm then moves to the next iteration, extracting another (partial) solution from the priority queue.

**Exercise:** Run this branch and bound algorithm on the following knapsack instance with $C = 10$:

<table>
<thead>
<tr>
<th>item</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
</tr>
</thead>
<tbody>
<tr>
<td>value</td>
<td>8</td>
<td>6</td>
<td>5</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>weight</td>
<td>6</td>
<td>5</td>
<td>5</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>utility</td>
<td>$4/3$</td>
<td>1.2</td>
<td>1</td>
<td>0.5</td>
<td>0.5</td>
</tr>
</tbody>
</table>