Branch and Bound

CS 149 Staff

December 1, 2006

1 Introduction

When confronted with an integer program which is not totally unimodular, there are two general lines of attack. Incomplete methods, including simulated annealing, tabu search, and genetic algorithms, run in polynomial time but are not guaranteed to return an optimal solution. The gold standard complete method is called branch and bound, which is a divide and conquer technique. The most general branch and bound technique is:

Replace the optimization over a set S with optimization over $S_1, S_2...S_k$, where $S_1 \cup S_2...S_k = S$ (usually these sets are chosen to be disjoint). Recurse until either the solution within this set is obvious or one can prove that there must be an optimum in one of the other sets being considered.

The most common way to prove that there's an optimum in another set is to solve the problem over a larger set $U \supset S$ (a relaxation) and then compare the result to the current best feasible solution known. However, there are other ways to prove that an optimum exists outside this set. For example, given two knapsacks subproblems with the same remaining weight and items, the one with the smaller profit can be discarded since there always exists a corresponding solution in the other sub problem that is better.

2 Runtime

The worst-case recursion relation for branch and bound is usually of the form $T(n) = 2T(n-1) + n^c$ for some constant c. If $T(0) = 1$ then repeated application of the recurrence relation yields $T(n) = n^c + 2(n-1)^c + 2^2(n-2)^c + ...2^n$ which is obviously exponential. Contrast this to quicksort, which has $T(n) = 2T(n/2) + n = n + 2(n/2) + 4(n/4)... = \Theta(n \log n)$ and insertion sort $T(n) = T(n-1) + n = n + (n-1) + ...1 = \Theta(n^2)$.

Remember that $2^{n+1} = 2 \cdot 2^n$, so if it takes a year to solve an instance of size 100 it takes 1024 years to solve a slightly larger instance of size 110.

3 Search Strategies

Best first search, where the next node to expand is the one with the best relaxation value, has the advantage that it never expands a node it wouldn't have to if it used another ordering. Unfortunately, best first search has two major flaws that make it in applicable in practice. Storing the current fringe is memory intensive, leading to potential memory exhaustion. Even if memory doesn't run out, moving from one part of the tree to it a completely different part limits the use of efficient incremental data structures, dramatically slowing search in many cases. Depth first search has a small memory footprint and makes good use incremental data structures, but risks wasting time on an uninteresting part of the search space. For example, when trying to design a good transportation vehicle, depth first search might first consider how to make a good horse drawn carriage, determine the optimal design for one, and only then reconsider the horse-drawn idea and consider internal combustion engines.

3.1 Branching Variables

When branching it is necessary to choose how to partition the search space. The most common way is to add a constraint of the form $x \geq \ell + 1$ to define one piece and $x \leq \ell$ to define the remainder, for some integer-valued variable x and constant ℓ. Given this technique, one must choose which variable to branch on. In practice balance search trees seemed to work better, so it makes sense to branch on a variable that is fractional in the relaxation (since otherwise one of the children nodes would have the same relaxation value). For many problems specialized partitioning strategies are more effective.

3.2 Value Ordering

Once they branching variable has been selected, one must decide which value for that variable to explore first. (Best first search doesn't have to make this choice, but depth first search and variants do.) There are many heuristics to solve this problem, such as choosing the child with the better relaxation value first, rounding the branching variable to the nearest integer and trying that side first, or various problem-dependant heuristics.