3J1-4

探索アルゴリズムにおける枝刈り手法の結合

Combining Pruning Methods in Search Algorithms

福永アレックス
Alex Fukunaga

東京工業大学
Tokyo Institute of Technology

Given a search algorithm and a set of pruning techniques, which pruning mechanism should be applied at each node in the search tree? We define and analyze some straightforward *heuristic combinations strategies* for combining multiple heuristics. We then experimentally evaluate the heuristic comination strategies on a new, state-of-the-art branch-and-bound solver for the multiple knapsack problem.

1. Introduction

Search algorithms commonly used in AI, such as backtracking, branch-and-bound, and A* depend on various techniques to control the size of the search space. Pruning techniques try to prune portions of the search tree by proving that the pruned subtree cannot contain a solution which is any better than (i) the best solution which has been found so far, or (ii) a solution which can be found in a subtree which will be fully explored by the algorithm (but has not necessarily been explored yet). The former case is the well-known branch-and-bound method. The latter case, which actually subsumes the former, is the broader class of *dominance*-baseed pruning mechanisms. An *admissible* or *exact* pruning mechanism is guaranteed to never prune an optimmal solution. For brevity, we use the term *pruning mechanism* to refer to an admissible, dominance-based pruning mechanism.

Pruning mechanisms are usually applied at every node in the search tree. For example, in branch-and-bound for a maximization problem, an upper bound heuristic value U is computed at every node and compared with a lower bound L, and the subtree is pruned if $U \leq L$. In general, there is a trade-off between pruning mechanism power and complexity: More powerful heuristics (which prune more nodes)require more time to comput. Thus, in order to for a pruning mechanism to be considered useful, it must prune the tree sufficiently to overcome the overhead of applying the mechanism. This paper investigates the following question: Given a set of pruning mechanisms H_1, H_2, etc., how should resources be allocated among them, i.e., which pruning mechanism should be applied at each node in order to minimize runtime?

2. Strategies for Combining Pruning Heuristics

Let H be an admissible, pruning mechanism.We say that a search node N is *pruned by* H, denoted $N \dagger H$, if the subtree under N will be pruned by an application of H.

We say that a pruning mechanism H_2 *subsumes* another pruning mechanism H_1 if $N \dagger H_1 \Rightarrow N \dagger H_2$ for all nodes N (in other words, H_2 is strictly more powerful than H_1.)

The simplest strategy is to choose only one mechanism, say H_a, and apply it at every node. This *pure* strategy is the most common strategy used in practice.

A *depth-based switching* strategy $\{(H_a, d), H_b\}$ applies H_a at every node in the search tree with depth less than or equal to d, and applies H_b at every node at depths greater than d. This strategy is most commonly applied when H_a subsumes H_b, and is significantly more expensive than H_b. The intuition is to apply the powerful but expensive H_b where we can get the maximum benefit, and avoid calling it deep in the tree where its utility is minimal. A problem with this approach is that *a priori*, it is not clear what the depth cutoff d should be.

A *Chain* strategy $\{H_a, H_b\}$ first applies mechanism H_a. If the node is pruned by H_a, then we backtrack. Otherwise, H_b is applied, and we backtrack if the node is pruned by H_b. In cases where the cost of computing H_a is significantly smaller than the cost to compute H_b, this translates to the intuitively obvious strategy: apply a very cheap pruning mechanism and try to prune quickly, otherwise, run the slower pruning mechanism to see if that will result in pruning the node.

Let us consider the pure and chain strategies above. We define a *terminal node* as a node where the search algorthm backtracks due to some reason other than pruning by one of the pruning mechanisms under consideration. For example, a node where we backtrack because we reach the natural, backtrack limit for the problem (e.g., the remaining subproblem is empty) is a terminal node.

Let H_1 and H_2 be two pruning mechanisms, where H_2 subsumes H_2. For every non-terminal node in the search tree, there are only three possibilities if H_2 subsumes H_1: (a) $N \dagger H_2$ and $N \dagger H_1$ (N can be pruned by both H_1 or H_2. (b) $\neg(N \dagger H_1)$ and $\neg(N \dagger H_2)$ (neither mechanism prunes N) (c) $N \dagger H_2$ and $\neg(N \dagger H_1)$.

If H_1 does not subsume H_2, then there is a fourth possibility: (d) $\neg(N \dagger H_1)$ and $N \dagger H_2$. However, meaningful

連絡先: 福永アレックス　東京工業大学　東京都目黒区大岡山 2-12-1 S-18 fukunaga@is.titech.ac.jp

analysis of this case seems to require estimations of the amount of pruning performed, which is a difficult problem. Thus, we assume that H_2 subsumes H_1.

First, consider the pure strategy which applies H_2 at each node N. The cost of executing this strategy is $C_{H_2}(N)$, or more succinctly, C_{H_2}.

Next, consider the chain strategy $\{H_1, H_2\}$, which applies H_1 first, and backtracks if N is pruned, and otherwise applies H_2. In case (a), the strategy will first apply H_1, prune the node, and backtrack, so the cost of the node is C_{H_1}. In case (b), H_1 and H_2 are both applied (unsuccessfully), and the cost is $C_{H_1} + C_{H_2}$. In case (c), H_1 is applied first and fails to prune the node, then H_2 is applied successfully, so the cost is $C_{H_1} + C_{H_2}$.

Since H_1 subsumes H_2, the set of nodes generated by the chaining strategy $\{H_2, H_1\}$ is exactly the same the set of nodes generated by the pure strategy H_1. The only difference is the runtimes of the two strategies. We can compare the runtimes of these two strategies by comparing the expected runtime cost per node.

For the pure H_1 strategy, the cost per node is simply C_{H_2}. For the $\{H_1, H_2\}$ chaining strategy, the cost per node is $P_aC_{H_1}+P_b(C_{H_1}+C_{H_2})+P_c(C_{H_1}+C_{H_2})$, where $P_a, ..., P_c$ are the probability of cases a-c defined above, respectively. The H_1, H_2 strategy is preferable to the pure H_1 strategy when the expected cost per node for the chaining strategy is lower than the cost per node of the pure strategy. That is: $C_{H_2} > P_aC_{H_1} + P_b(C_{H_1} + C_{H_2}) + P_c(C_{H_1} + C_{H_2})$.

Simplifying and solving for P_a (using the fact that $P_a + P_b + P_c = 1$), we obtain the result that the $\{H_2, H_1\}$ strategy has a lower cost per node than the pure H_1 strategy if and only if $P_a > C_{H_1}/C_{H_2}$.

This allows us to predict, for any particular node, whether to use the pure strategy H_2, or the chain strategy $\{H_1, H_2\}$. In addition, this suggests an *adaptive chain* strategy, where we decide upon the mechanism strategy (pure or chain) by applying the above inequality to data collected at runtime (e.g., the pruning probabilities and costs can be collected per search depth).

3. Experiments: The Multiple Knapsack Problem

We compared pure, chain, and adaptive chain strategies using a branch-and-bound-based solver for the Multiple Knapsack Problem (MKP).

The MKP is a classical combinatorial optimization problem, which generalizes the well-known 0-1 Knapsack problem to multiple knapsacks. Given m containers (bins) with capacities $c_1, ..., c_m$, and a set of n items, where each item has a weight $w_1, ..., w_n$ and profit $p_1, ..., p_n$. Assigning some subsets of the items to the containers to maximize the total profit of the items, such that the sum of the item weights in each container does not exceed the container's capacity, and each item is assigned to at most one container is the *0-1 Multiple Knapsack Problem*, or MKP. The MKP has numerous applications, including task allocation among autonomous agents continuous double-call auctions, multiprocessor scheduling, vehicle/container loading, and the assignment of files to storage devices in order to maximize the number of files stored in the fastest storage devices.

The MKP is strongly NP-complete. The previous, state-of-the-art, exact algorithm for the MKP is Pisinger's Mulknap algorithm [3]. Recently, we developed bin-completion, a new, branch-and-bound algorithm for the MKP which integrates a powerful dominance criterion, symmetry detection techniques, as well as bounding techniques used in previous solvers. Bin-completion is a general strategy which can be applied to the class of multi-container, assignment problems, including the MKP, bin-packing, and bin-covering problems. Experimental results showed that bin-completion significantly outperforms the previous state of the art algorithm on difficult benchmark instances [2].

We have developed a number of pruning mechanisms for the bin-completion algorithm. These are based on a dominance relationship between candidate assignments of items to bins, and symmetry relations betwen search nodes [1]. In particular, these include a powerful but complex pruning mechanism, path-symmetry, and a cheap, less powerful pruning mechanism, 2-swap symmetry (see [1] for a description of these mechanisms).

3.1 Summary of Results

We compared pure, chain, and adaptive chaining strategies for these two pruning mechanisms (path-symmetry and 2-swap symmetry). Experiments were performed on the same groups of benchmarks as [1]. As an example, on a set of 20 standard benchmark problems with 25 bins and 50 items where the item weights and profits are highly correlated (c.f. [3]), we found that the chain strategy solves problems on average 3 times faster than either pure strategy (pure path-symmetry and pure 2-swap). Thus, we have found that the chain strategy is an important component of the current, state-of-the-art algorithm for the MKP.

However, our current implementation of adaptive chaining yielded results which were approximately 25% slower than the chain strategy. Future work will focus on developing a robust, adaptive chain strategy which can outperform the chain strategy.

参考文献

[1] A. Fukunaga. Exploiting symmetry in multiple knapsack problems. In *Proc. CP-07 Workshop on symmetry and constraint satisfaction problems*, 2007.

[2] A. Fukunaga and R. Korf. Bin-completion algorithms for multicontainer packing, knapsack, and covering problems. *Journal of Artificial Intelligence Research*, 28:393–429, 2007.

[3] D. Pisinger. An exact algorithm for large multiple knapsack problems. *European Journal of Operational Research*, 114:528–541, 1999.