How Do You Know Your Search Algorithm and Code Are Correct?

Richard E. Korf
Computer Science Department
University of California, Los Angeles
Los Angeles, CA 90095
korf@cs.ucla.edu

Abstract

Algorithm design and implementation are notoriously error-prone. As researchers, it is incumbent upon us to maximize the probability that our algorithms, their implementations, and the results we report are correct. In this position paper, I argue that the main technique for doing this is confirmation of results from multiple independent sources, and provide a number of concrete suggestions for how to achieve this in the context of combinatorial search algorithms.

Introduction and Overview

Combinatorial search results can be theoretical or experimental. Theoretical results often consist of correctness, completeness, the quality of solutions returned, and asymptotic time and space complexities. Experimental results typically consist of the number and quality of the solutions returned, the number of nodes generated, and the running time of the algorithm. In the remainder of this paper, we first consider the role of proofs in verifying results, then the issues of solution correctness, both optimal and sub-optimal solution quality, nodes generated, and finally running time.

Mathematical Proofs

One approach to this problem is to prove various properties of an algorithm and implementation. While a proof is certainly very strong evidence of the correctness of an algorithm, it shouldn't be considered absolute. Constructing a proof is also a complex process, and error-prone as well, but at least a formal proof can be checked automatically. In any case, the process of proving an algorithm correct is sufficiently different from the process of designing the algorithm that proving correctness is extremely useful, since the kinds of errors made in a proof are likely to be different from those made in designing an algorithm.

The notion of a correctness proof can also be extended to the code used to implement the algorithm. In this case, the task is to show formally that the implementation corresponds to the specification of the algorithm. Unfortunately, this is generally a more difficult task than proving an algorithm correct, and may not be feasible for a large program.

Is a Given Solution Correct?

The first question to ask of a search algorithm is whether the candidate solutions it returns are valid solutions. The algorithm should output each solution, and a separate program should check its correctness. For any problem in NP, checking candidate solutions can be done in polynomial time.

Is a Given Solution Optimal?

Next we consider whether the solutions returned are optimal. In most cases, there are multiple very different algorithms that compute optimal solutions, starting with simple brute-force algorithms, and progressing through increasingly complex and more efficient algorithms. Thus, one can compare the solution costs returned by the different algorithms, which should all be the same. This can be done in a bootstrapping process, where the simplest algorithms are only run on smaller problem instances, and more efficient algorithms are run on larger instances as well. In general, simpler algorithms are more likely to be correct. Furthermore, if the algorithms are significantly different from each other, they are unlikely to make the same errors.

This method can be enhanced by computing all optimal solutions. This can often be done with only a small change to a search algorithm, such as continuing to search after the first optimal solution is found, and pruning partial solutions only when their cost strictly exceeds that of the best solution found so far. For a best-first search such as A*, one would need to maintain a pointer to each optimal parent of a given node, and then trace all optimal paths back from the goal nodes to the start node. Given all optimal solutions, one can check that different algorithms all return the same set of solutions. Even if this is not feasible on large problems, it can be done on small problems for testing purposes.

Many algorithms present choices that affect efficiency, but not correctness. For example, in alpha-beta minimax, the order in which children are explored affects performance, but not the minimax value computed. One can check that the same value is computed under different orderings. This is often useful because many programs contain "silent bugs" which never manifest themselves, because they are masked by other properties of the code. For example, a program may work correctly if an array is sorted in increasing order, but fail if it is sorted in decreasing order. Changing a sorted order is usually as simple as reversing a comparison operator.

Sub-Optimal Solution Quality Correctness

Finding and verifying optimal solutions is computationally much more expensive than finding sub-optimal solutions. However, it is usually easier to check that an implementation of an optimal algorithm is indeed returning optimal solutions, than it is to check that an implementation of a sub-optimal algorithm is returning solutions of the expected quality, given a correct implementation of the algorithm. The reason is that while there are often multiple very different algorithms that find optimal solutions, it is rare to find multiple different suboptimal algorithms that return solutions of the same cost to compare against each other.

Some search algorithms guarantee bounded sub-optimal solutions. For example, given a parameter w, weighted-A* (Pohl 1973) guarantees solutions that are no worse than w times optimal. For such algorithms the cost of solutions can be compared to w times the optimal solution cost. Unfortunately, this is often not a sufficient test of their solution quality, since many such algorithms actually return solutions much better than their guaranteed solution quality.

Are the Node Generations Correct?

The next question is whether a program generates the correct number of nodes, based on the algorithm it implements.

One technique to address this question is to analyze the asymptotic complexity of the algorithm. For example, if the complexity is $O(b^d)$, then the ratio of the number of nodes at successive depths d should approach b at large depths.

For some algorithms, it may be possible to compute the exact number of nodes generated in certain cases, such as the best or worst case. One can then generate these cases, and compare the experimental results to the theoretical prediction. For example, the exact number of leaf nodes evaluated by alpha-beta minimax in the best case for a tree with uniform branching factor and depth is easily computed. Furthermore, the best case occurs when all leaf nodes have the same value. Thus one can run alpha-beta on a uniform tree with identical leaf values, count the number of leaf nodes evaluated, and compare this to the predicted value. This would also expose the common error of pruning only on strict inequality, rather than on equality as well.

As another example, an exhaustive breadth-first search of a finite combinatorial problem outputs the number of unique states at each depth. For example, this has been done for the 15-Puzzle (Korf and Schultze 2005). Since there are several different ways to implement a breadth-first search, the results from shallow searches can be compared among alternative implementations. For large depths, however, most algorithms become infeasible due to memory limits. However, we often know the exact number of unique states. For example, the Fifteen Puzzle contains $16!/2$ reachable states. Thus, the sum of the number of unique states at each depth must equal the total number of states.

Are the Running Times Correct?

The next question is whether the running time of a program matches the expected running time of the algorithm. Given an asymptotic analysis of the running time of the algorithm, one can check that the actual running time of the program agrees with the analysis for large problems. For example, if an analysis indicates that the running time should be $O(n \log n)$, where n is the number of node generations, one can count the number of node generations n, and check to see that the running time is proportional to $n \log n$.

Independent Implementations

While many of the techniques above can be very useful, the gold standard for scientific accuracy is reproduction of results by independent researchers. This is often practical, even within a single research group, since many heuristic search algorithms are relatively simple, and can be implemented and debugged in a few weeks. It may be practical for a member of the group who is not directly involved in a project to independently implement the same algorithm. This allows comparison of all experimental results, including solution quality, node generations and running times. This valuable service could be compensated by co-authorship of the paper, and of course reciprocating when the roles are reversed.

An additional benefit of this strategy is that it can result in a better implementation of the algorithm. Given two different implementations of the same algorithm, any discrepancies in the number of node generations is likely due to one implementation pruning nodes it shouldn't, which is a correctness bug, or generating nodes that it shouldn't, which is a performance bug, or both. Since correctness bugs are easier to find, particularly for algorithms that return optimal solutions, most bugs are performance bugs. A similar argument applies to significant discrepancies in the running times of two implementations of the same algorithm. In general, having two people implement the same algorithm often results in new insights into how to do it more efficiently.

Summary and Conclusions

The design and implementation of algorithms, including search algorithms, is an inherently error-prone process. Our main defense against errors is to have multiple, redundant checks on all our results, as independent of each other as possible. These checks can include theoretical proofs of correctness, comparison of theoretical analyses to experimental results, experimental comparisons between different algorithms computing the same results, and experimental comparisons between independent implementations of the same algorithm. If you don't have a reasonably independent way of checking the correctness of your algorithm, code, or results, you should probably assume they are wrong, since that is the most likely scenario.

References

Korf, R., and Schultze, P. 2005. Large-scale, parallel breadth-first search. In *AAAI-2005*, 1380–1385.

Pohl, I. 1973. The avoidance of (relative) catastrophe, heuristic competence, genuine dynamic weighting and computational issues in heuristic problem solving. In *IJCAI-73*, 12–17.