Proceedings of the Twenty-Sixth AAAI Conference on Artificial Intelligence

Research Challenges in Combinatorial Search

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

Abstract

I provide a personal view of some of the major research challenges in the area of combinatorial search. These include solving and playing games with chance, hidden information, and multiple players, optimally solving larger instances of well-known single-agent toy problems, applying search techniques to more realistic problem domains, analyzing the time complexity of heuristic search algorithms, and capitalizing on advances in computing hardware, such as very large external memories and multi-core processors.[1]

Introduction

I found the challenge of writing this paper a little daunting. The task, as I understand it, is both to predict future research problems in the field, and also to provide an opinion about what problems the field should address. My main difficulty is that I don't feel qualified to do this. What research will be done will ultimately be decided by the researchers who will actually do the work. In any case, here is my best effort along these lines. I should point out that this is not a consensus view of the entire field. While I did solicit comments from some members of the field, this is ultimately a personal view of where I think the field will and should go.

First, I'd like to circumscribe what I plan to cover, and also what I plan to leave out of this treatment. I use the term "combinatorial search" in the title to distinguish this area from other common uses of the term "search", in particular web search and database search, each of which is an important field in its own right, but have little in common with combinatorial search, other than sharing the word "search". By combinatorial search, I mean the kind of search involved in playing a game like chess, or solving a Rubik's Cube. What characterizes combinatorial search is that the problem space is implicitly defined by a function that generates the neighbors of a state, rather than an explicit listing of the states as in a database or the world-wide web.

Traditionally, combinatorial search has encompassed three different problem types: single-agent problems such as Rubik's Cube and the sliding-tile puzzles, games such as chess and checkers, and constraint-satisfaction problems such as Sudoku. To a large extent, each of these subareas has become a separate research field, with it's own conference(s), and even journals in the case of the latter two. I will ignore constraint-satisfaction problems, as they are covered by another paper in these proceedings. I will say a few words about game playing, since it is not otherwise represented, but I don't have much expertise in this area. The bulk of this paper will focus on single-agent problems.

In particular, I will focus on systematic search algorithms, which are guaranteed to find a solution if one exists, and often find optimal solutions. This is in contrast to stochastic local search algorithms, such as simulated annealing, genetic algorithms, tabu search, etc. The main reason for this choice is that my expertise is entirely within the area of systematic search algorithms, the field of stochastic search has become a subfield in own right, and others are much more qualified to comment on its research challenges.

Finally, this is not a survey of current active research in combinatorial search. I leave that to the excellent paper by Nathan Sturtevant in these proceedings. Rather, this is an attempt to look into the future.

Game Playing

The term "computer game playing" has become ambiguous. Originally it referred to computers playing games such as chess and checkers, against people or other computers. Now, it also refers to people playing computer games, including war games, fantasy games, sports games, etc., an industry that is reputed to be larger than the motion picture industry.

An important practical application, which is currently receiving significant attention in the search community, is building automated agents to control characters in computer games. As one example, path planning by such automated agents makes use of search algorithms, and usually has the additional complication that multiple agents must plan their paths without colliding with each other. This is an active current research area, and will do doubt continue to be so. Given the amount of money in this industry, however, it is likely that significant advances will be made by game developers, and will not be published due to proprietary considerations, or simply because there is little motivation for, nor a culture of scientific publication in that industry.

Returning to computers playing traditional games, there

[1]This paper was invited as a challenge paper to the AAAI'12 Sub-Area Spotlights track.

are two primary research challenges. The first is building programs to beat the best humans at various games, and the second is "solving" a game, which means to determine an optimal strategy for playing the game, and the win, loss, or draw value of a game given optimal play by all players.

Building a program that can defeat the best humans is the easier of these two tasks. The first non-trivial games to be addressed, chess and checkers, are two-player perfect-information games, and the signature achievement of this effort was the defeat of Gary Kasparov by the Deep Blue machine in 1997 (Campbell, Hoane, and Hsu 2002). Currently, the most high-profile challenge in this area is the game of Go, a two-player perfect-information game, but one where the best computers are not yet competitive with the best people. As a general observation, computers tend to play better relative to people in games with smaller branching factors, where search is most effective.

Current and future challenges include games with chance elements, such as backgammon, games with hidden information, including card games such as bridge and poker, and games with more than two players, particularly those that involve coalitions among groups of players. The challenge is simple to state: pick a game, and build a program that is better than the best humans.

Solving a game is even more difficult than beating the best humans, since it requires computing a perfect strategy. The most complex game solved to date is checkers (Schaeffer et al. 2007). It's a draw if both players play optimally. Again, the challenge is simple to state: pick a game that hasn't been solved, and solve it. For the near future, these efforts are likely to focus on two-player perfect-information games. For example, solving chess is an obvious challenge.

We now turn our attention to the main focus of this paper, which is single-agent problems.

Solving Larger Toy Problems

Much research in this area has used classical toy problems such as the 4×4 sliding-tile Fifteen Puzzle, or Rubik's Cube as problem domains. Other more recently popular problems include permutation puzzles such as Top-Spin or the Pancake Problem, or the 4-peg Towers of Hanoi problem. A commercially available version of Top-Spin consists of a circular ring of twenty distinct tokens, and a turnstile that can reverse the order of any four consecutive tokens in the ring. The pancake problem (Dweighter 1975) consists of a stack of n pancakes, all of different sizes, and a "spatula" that flips or reverses the order of the top k pancakes, the goal being to sort the pancakes in order of size. The 4-peg Towers of Hanoi problem has the same rules as the classical 3-peg problem, but adds one additional peg. It's an interesting search problem because the optimal solution to this problem is not known with certainty.

What characterizes all these problems is that for a given size, there is a single common search space. What distinguishes one instance from another is simply the choice of initial and goal states. With any such problem, there are at least four different research challenges, listed here in increasing order of difficulty.

The first challenge is to find any solution at all. For some problems, such as the Pancake Problem, this is trivial. For others, such as Rubik's Cube, it is far from trivial, as least for humans. For all of these problems, however, there exist deterministic algorithms that will efficiently generate suboptimal solutions to arbitrary problem instances.

The second and most common challenge is to find optimal or shortest solutions to random problem instances. This is generally accomplished using systematic search algorithms such as A* (Hart, Nilsson, and Raphael 1968) or IDA* (Korf 1985) with admissible or lower-bound heuristics. Since all these problems come in different sizes, the obvious research challenge is to find optimal solutions to larger instances than can currently be solved optimally. For the sliding-tile puzzles, currently the largest size problem for which random instances can be solved optimally is the 5×5 Twenty-Four Puzzle. For Rubik's Cube, the classic $3 \times 3 \times 3$ is the state of the art. The next research challenge is to optimally solve random instances of the 6×6 Thirty-Five Puzzle, or the $4 \times 4 \times 4$ Rubik's Cube. Since these represent enormous increases in the sizes of the corresponding search spaces, for the sliding-tile puzzles at least one can consider intermediate problems such as the 5×6 Twenty-Nine Puzzle.

Looking back, solving a significantly larger instance than the previous state of the art has generally required a new idea, rather than incremental improvements to existing techniques. For example, moving from optimal solutions to the 3×3 Eight Puzzle to the 4×4 Fifteen Puzzle required overcoming the memory limitation of the A* algorithm with IDA*. Moving from optimal solutions to the $2 \times 2 \times 2$ Rubik's Cube to the $3 \times 3 \times 3$ Rubik's Cube (Korf 1997) required the development of pattern database heuristics, which precompute and store in memory large tables of heuristic values for subparts of the problem in question (Culberson and Schaeffer 1998). Moving from the Fifteen Puzzle to the Twenty-Four Puzzle required the development of additive pattern databases (Korf and Felner 2002), which allow heuristic values for different parts of the puzzle to be added together while preserving the lower-bound property required for optimal solutions. For a recent example, the discovery of a better admissible heuristic for the Pancake Problem led to a dramatic increase in the size of problems that can be solved optimally (Helmert 2010).

The third and next harder challenge is to determine the worst-case optimal solution length for any instance of a given problem. For example, the hardest Fifteen Puzzle instance requires 80 moves to solve (Brungger et al. 1999). Similarly, a problem that was open for over 35 years and only recently solved was to determine the worst-case optimal solution for an instance of the $3 \times 3 \times 3$ Rubik's Cube. The answer is 20 moves, assuming that a 180-degree twist is counted as a single move (Rokicki et al. 2010). The obvious challenge is to determine these values for larger versions of these problems, such as the Twenty-Four Puzzle or the $4 \times 4 \times 4$ Rubik's Cube.

Finally, there is the challenge of exhaustively searching the entire problem space of a given problem. One compact way to present the output of such a search is to list the number of unique states at each depth from a designated initial

state. This was accomplished for the Eight Puzzle in the 1960s (Schofield 1967), and more recently for the Fifteen Puzzle (Korf and Schultze 2005). Performing such an exhaustive search for Rubik's Cube is perhaps the last remaining search challenge associated with the $3 \times 3 \times 3$ cube.

Beyond Toy Problems

Perhaps the sharpest criticism leveled at the area of combinatorial search is that many of the problem domains used are toy problems. I'm as guilty as anyone in this regard, but I'll argue here that it would behoove us to move beyond toy problems towards more realistic applications. I'll begin by examining the argument in favor of toy problems.

First, I want to make it clear that by "toy" problem I don't mean a trivial problem. Rubik's Cube has a problem space with over 10^{19} states, and the Twenty-Four Puzzle has over 10^{24} states. Finding optimal solutions, the worst-case optimal solution length, or performing an exhaustive search of such problems spaces are far from trivial, to say the least. I use the term "toy" here in the sense that a Rubik's Cube, for example, is literally a toy. Even to buy a chess set, one would probably go to a toy store.

There are several arguments for working on toy problems. One is that they are simple to describe, and easy to implement. In addition, these properties often motivate other researchers to work on the same problems, greatly facilitating the comparison of results. Finally, techniques developed on toy problems are often transferable to real-world problems.

While all these arguments have merit, the last is perhaps the most suspect. If the goal is to develop techniques that apply to real-world problems, then why not work on those problems directly? The usual response is that they are much more complex to describe and implement. I believe, however, that we can preserve the main advantages of toy problems by working on abstractions of real-world problems.

Many NP-hard problems, for example, are very simple to state, and directly model real-world problems. Furthermore, combinatorial search techniques are required to find optimal solutions to NP-hard problems. A concrete example from my own work is number partitioning (Korf 2011). Given a set of integers, divide them into a fixed number of subsets, so that the largest subset sum is minimized. Number partitioning directly models the problem of multi-processor scheduling, where each of a set of jobs with different durations must be assigned to one of a set of identical processors or cores, so that all jobs are completed in the shortest possible time. Implementing a simple search algorithm for number partitioning is much simpler than implementing an algorithm for Rubik's Cube. There are hundreds of simple NP-hard problems, many of which were invented as abstractions of real-world problems, and many of them have received little attention from researchers. Thus, we can have the simplicity advantages of toy problems, while solving problems that directly model real-world applications.

In addition to NP-hard problems, there are other search problems that are computationally challenging, even if their complexities are polynomial. One such problem, which has been as a test domain for search algorithms, is sequence alignment in computational biology. For example, given the genome of a human and that of a mouse, one would like to find regions of similarity between the two. The sequences are the strings of base pairs in their DNA, and the alignment process finds regions of similarity. Optimal pairwise alignment can be modelled as finding a lowest-cost corner-to-corner path in a two-dimensional grid, where each axis is labeled with the letters of one of the sequences, using a function that assigns a cost to each edge. This is a classic search problem. Its complexity is quadratic in the lengths of the sequences, but the sequences can be up to three billion base pairs long. Similarly, simultaneous alignment of three sequences can be modelled as finding a lowest-cost corner to corner path in a three-dimensional grid.

In addition to solving problems with important practical applications, moving beyond a small set of toy problems has another important advantage. When a single problem receives a lot of attention from researchers, further significant advances in the state-of-the-art become more difficult. This is true of both toy problems, such as the sliding-tile puzzles, and well-studied NP-complete problems, such as the travelling salesman problem. New problems that have not received extensive study often have new features that give rise to new techniques and algorithms, providing more opportunities for researchers to advance the state-of-the-art.

Optimal versus Suboptimal Solutions

I have focussed above almost exclusively on finding optimal solutions to combinatorial problems. For large instances of NP-hard problems, this is not feasible in practice. As a result, many researchers have switched their focus to algorithms that find suboptimal solutions. I have not done this, for several reasons.

The first is that it is much more difficult to compare different algorithms that find suboptimal solutions. The reason is that there are two figures of merit for such algorithms, the solution quality and the running time. In order for one suboptimal algorithm to dominate another, it must find better solutions for all different running times. This rarely occurs in practice. Rather, one algorithm often will find better solutions for small running times, but other algorithms will find better solutions given longer running times. With optimal algorithms, we only have to compare the running times to find optimal solutions.

In addition, for most problems, there is another option for dealing with instances that are too large to solve optimally in practice, which is an anytime optimal algorithm. An anytime optimal algorithm returns a solution to a problem almost immediately, and as it continues to run it finds better solutions, until it generates, and then eventually verifies an optimal solution. Given any deadline to solve a problem, an anytime algorithm can be run up until the deadline, with the best solution found at that point returned. These algorithms can be compared against each other by the time they take to verify optimal solutions on smaller problem instances. In addition, many of these algorithms allow guarantees on the quality of the solutions generated, relative to optimal solutions. For example, weighted-A* allows one to bound solution quality to within an arbitrary constant of optimal, but the smaller the constant, the larger the running time (Pohl 1973).

Analyzing the Running Time of Heuristic Search Algorithms

An important research goal in any area of computer science is to be able to predict the running time of algorithms. For brute-force search algorithms, this is often straightforward, since the running time is proportional to the number of node generations, and it is often easy to calculate how many states there are in a complete problem space. As algorithms become more efficient, however, they become more complex, and it becomes more difficult to analyze their running times.

For example, depth-first branch-and-bound is a ubiquitous search algorithm, yet it is very difficult to predict its running time. One reason is that the amount of pruning depends on the sequence in which solutions are found. For example, the maximum amount of pruning only occurs once an optimal solution is found, and when that will occur during the search is very difficult to determine in advance.

As another example, the running time of any heuristic search depends on the quality of the heuristic function, in addition to the algorithm. This requires a way of characterizing the heuristic function in a way that allows predicting the performance of a search algorithm using that heuristic.

Early work in this area (Pearl 1984) was focussed on asymptotic complexity based on abstract models of search spaces, and high-level measures of heuristic accuracy. Unfortunately, those asymptotic results couldn't accurately predict the actual numbers of nodes generated on real problems with real heuristics.

More recently, there has been some success predicting the actual performance of IDA* on problems such as the sliding-tile puzzles and Rubik's Cube (Korf, Reid, and Edelkamp 2001). This remains an active area of research (Zahavi et al. 2010), but predicting the performance of most algorithms, using most heuristics, on most problems remains an important research challenge.

Capitalizing on New Hardware Capabilities

In addition to new problems, and new ideas for old problems, another generator of research in combinatorial search is new hardware technologies, or significant changes in the scale of existing technologies. Nobody would doubt that dramatic increases in processor speeds over the last 60 years has played a critical role in some of the biggest success stories in combinatorial search. Similarly, the dramatic increase in memory capacities has made pattern database heuristics practical and effective for a wide variety of problems.

A more recent example has been the explosion in the capacity of magnetic disk storage over the past decade. Traditionally, disk storage has been used primarily for file storage, rather than as temporary storage for computation. Many combinatorial search algorithms are limited by the available storage, since they save most of the nodes that they generate, in order to detect duplicate node generations and prevent multiple reexpansions of the same states. Recently search algorithms have been implemented that store their nodes on magnetic disk rather than main memory, in order to take advantage of their much larger capacity. The main research challenge has been to deal with the very long latencies of random disk accesses. Two approaches to this problem include delayed duplicate detection (Korf 2008) and structured duplicate detection (Zhou and Hansen 2004).

Looking forward, the biggest recent change in the evolution of computer hardware has been a levelling off of clock speeds, accompanied by an increase in the number of CPU cores on a chip. As with the rest of computer science, research in combinatorial search will have to figure out how to best make use of such parallel processing. In a sense, combinatorial search is well-suited to parallel processing, since search algorithms are characterized by relatively small atomic computations, such as a node generation or expansion, repeated many times in the course of a search. This property should make search easy to parallelize. On the other hand, parallel processing has been around for a long time, and some canonical algorithms, such as minimax search with alpha-beta pruning, have been notoriously difficult to parallelize.

Summary

I have provided a personal view of future research challenges in the area of single-agent combinatorial problems, and to a limited extent computer games. In the latter area these include automated characters in computer gaming, and solving and playing human games with chance elements, hidden information, and multiple players. In the former area the research challenges include finding optimal solutions to larger versions of classical toy problems, solving combinatorial problems that model real-world applications, analyzing the time complexity of search algorithms, and capitalizing on new hardware technologies.

Acknowledgments

Thanks to Carlos Linares Lopez, Wheeler Ruml, and Ariel Felner for suggesting research challenges for this paper. Thanks to Victoria Cortessis for helpful comments on a draft of this paper.

References

Brungger, A.; Marzetta, A.; Fukuda, K.; and Nievergelt, J. 1999. The parallel search bench ZRAM and its applications. *Annals of Operations Research* 90:45–63.

Campbell, M.; Hoane, A.; and Hsu, F. 2002. Deep blue. *Artificial Intelligence* 134(1-2):57–83.

Culberson, J., and Schaeffer, J. 1998. Pattern databases. *Computational Intelligence* 14(3):318–334.

Dweighter, H. 1975. Elementary problem e2569. *American Mathematical Monthly* 82(10):1010.

Hart, P. E.; Nilsson, N. J.; and Raphael, B. 1968. A formal basis for the heuristic determination of minimum cost paths. *IEEE Transactions on Systems Science and Cybernetics* SSC-4(2):100–107.

Helmert, M. 2010. Landmark heuristics for the pancake problem. In *Proceedings of the Symposium on Combinatorial Search (SOCS-10)*.

Korf, R. E., and Felner, A. 2002. Disjoint pattern database heuristics. *Artificial Intelligence* 134(1-2):9–22.

Korf, R. E., and Schultze, P. 2005. Large-scale, parallel breadth-first search. In *Proceedings of the 20th National Conference on Artificial Intelligence (AAAI-2005)*, 1380–1385.

Korf, R. E.; Reid, M.; and Edelkamp, S. 2001. Time complexity of Iterative-Deepening-A*. *Artificial Intelligence* 129(1-2):199–218.

Korf, R. E. 1985. Depth-first iterative-deepening: An optimal admissible tree search. *Artificial Intelligence* 27(1):97–109.

Korf, R. E. 1997. Finding optimal solutions to Rubik's cube using pattern databases. In *Proceedings of the 14th National Conference on Artificial Intelligence (AAAI-97)*, 700–705.

Korf, R. E. 2008. Linear-time disk-based implicit graph search. *Journal of the Association for Computing Machinery* 55(6):26–1 to 26–40.

Korf, R. E. 2011. A hybrid recursive multi-way number partitioning algorithm. In *Proceeding of the 22nd International Joint Conference on Artificial Intelligence ((IJCAI-11)*, 591–596.

Pearl, J. 1984. *Heuristics*. Reading, MA: Addison-Wesley.

Pohl, I. 1973. The avoidance of (relative) catastrophe, heuristic competence, genuine dynamic weighting and computational issues in heuristic problem solving. In *Proceedings of the Third International Joint Conference of Artificial Intelligence (IJCAI-73)*, 12–17.

Rokicki, T.; Kociemba, H.; Davidson, M.; and Dethridge, J. 2010. Brute-forcing rubik's cube: God's number is 20. http://tomas.rokicki.com/rubik20.pdf.

Schaeffer, J.; Culberson, J.; Treloar, N.; Knight, B.; Lu, P.; and Szafron, D. 2007. Checkers is solved. *Science* 317(5844):1518–1522.

Schofield, P. 1967. Complete solution of the eight puzzle. In Meltzer, B., and Michie, D., eds., *Machine Intelligence 3*. New York: American Elsevier. 125–133.

Zahavi, U.; Felner, A.; Burch, N.; and Holte, R. C. 2010. Predicting the performance of ida* using conditional dependencies. *Journal of Artificial Intelligence Research (JAIR)* 37:41–83.

Zhou, R., and Hansen, E. A. 2004. Structured duplicate detection in external-memory graph search. In *Proceedings of the 19th National Conference on Artificial Intelligence (AAAI-2004)*, 683–688.