Experimental Computer Science in Game Playing

Jonathan Schaeffer
Department of Computing Science
University of Alberta
Edmonton, Alberta
Canada T6G 2H1

1 Introduction

This short article attempts to illustrate the importance of experimental computer science in the development of game-playing programs. While game playing is no longer one of the major thrusts of artificial intelligence research, it still remains one of the most visible. For many areas of artificial intelligence, games provide a convenient domain for demonstrating the impact of new research results.

Game-playing programs operate under real-time constraints (usually they must play a fixed number of moves in a prescribed period of time). Since it has been experimentally demonstrated that there is a high correlation between search effort and performance, most programs are designed to maximize the information gained in the time allotted. The implication is obvious: practitioners want their programs to be as fast as possible. Every modification to the program must be judged not only by the static paper benefits of the modification, but also by its dynamic run-time costs.

Experimental work is at the heart of many aspects of game-playing program development. Game-playing practitioners have learned to base their conclusions almost exclusively on experimental data. History has taught them that mathematical models are too simplistic and simulations are inadequate. Instead most practitioners combine two experimental strategies:

1. pose a hypothesis and determine the truth of it by analyzing a statistically significant amount of data, and

2. generate data and sift through it looking for clues and patterns.

These approaches may seem rather low-level and primitive to some given the rich history of logic and mathematics in computing. However, experience has shown that these methods are effective. Although many have tried to improve and even automate this process, there is a large gap between what an experienced experimentalist can achieve versus anything achievable by any other approach (with current hardware and software technology). Many ideas have potential, but unless they can be demonstrated in practice on hard problems, the game-playing community will ignore them.

In some sense, high-performance game-playing is all experimental work. For example, in chess some would argue that the major research results have already been obtained and now it is only a matter of extending existing hardware and software technology before *Deep Blue* is better than all human chess players. Going that extra step from "one of the best" to "the best" is the most difficult part of building a high-performance system, and it is hard to justify it based on research results. Nevertheless, quantifying the effort required to achieve such a milestone is in itself an important research result. *Deep Blue* can be viewed as an experiment demonstrating the state-of-the-art capabilities (and limitations) of artificial intelligence and parallel processing technology.

2 Research

This section highlights of few of the areas in game playing where experimental research are critical to success. This is intended to be illustrative, not exhaustive.

Game-playing programs are often described as being a combination of search and knowledge. A simple model of such a program consists of a search algorithm (such as alpha-beta) that considers move sequences some number of moves ahead. When the search reaches the end of a sequence, "knowledge" is used to assess how good/bad the position is (the evaluation function). The evaluation result is backed-up and combined with other backed-up results until a value for the root is determined.

This is, in fact, an overly simplistic view. The search algorithms can be quite sophisticated and the knowledge can appear in many guises in the search. Further, most high-performance game-playing programs combine many features that depend on other technologies, such as parallel processing, databases, compiler optimizations, etc. These programs are usually complex pieces of software with all these technologies intertwined.

2.1 Search

The game-playing community was the first to extensively research high-performance search algorithms. Many search algorithms and their enhancements either originated in the game-playing world, or were first extensively assessed there. These include, for example, iterative deepening, caching data with hash tables, cycle elimination, and branch re-ordering; all now firmly established search techniques.

This work was motivated by the drive for high, real-time performance. Without this constraint, then the issues of search disappear. In theory, given sufficient resources, all minimax search algorithms (such as alpha-beta, SSS*, conspiracy numbers, etc.) are equivalent. What separates the algorithms has nothing to do with performance in the limit; it has to do with which one can best uses its time resources most effectively. This includes a number of things such as the algorithm overhead (which affects the number of positions evaluated), storage overhead, the decisions of which nodes to expand, and the order in which nodes are expanded. These search algorithms have different costs and search strategies which means that they can produce different answers given the same real-time constraints. There isn't a usable theory of optimization that allows one to scientifically analyze the myriad of possibilities and select the "best" one. Instead, extensive experimentation is the only solution. The user may use their own search algorithm (such as hill climbing or simulated annealing) to guide them through the space of possible experiments, but this, at best, can only guide them to an acceptable solution quickly, without ever knowing if a better one exists.

Despite this seemingly unscientific approach, there are a few research results that have resulted from the work on games that clearly stand out since they contradict the intuition of what many would conclude based on their theoretical models. Perhaps the most important result has been the realization that brute-force search is a viable option. Theory would discount such an approach, since search trees typically grow exponentially with depth. Nevertheless, the realization that a naive search algorithm that attempts to exhaustively examine a search space, subject to real-time and resource constraints, is effective has been a major surprise.

Numerous simulations of game-tree search algorithms have been reported in the literature. In almost every case the results have been ignored by practitioners; they know better than to trust any simulation-based conclusion. For example, Aske Plaat in his Ph.D. thesis refutes many of the conclusions in the literature on the well-known SSS* algorithm. Most of these conclusions were derived from simulations.

Plaat writes that (Aske Plaat, "Research Re: Search and Research," Ph.D. thesis, Erasmus University, 1996, page 82):

> [...] the trees generated in practice are highly complex and dynamic entities, whose structure is influenced by the techniques that make use of [four features of the search space]. Acquiring data on these factors and the way they relate seems a formidable task. [...] These causes and effects are all interconnected, yielding a picture of great complexity that does not look very inviting to disentangle. [...]
>
> In the light of the highly-complex nature of real-life game trees, simulations can only be regarded as approximations, whose results may not be accurate for real-life applications. We feel that simulations provide a feeble basis for conclusions on the relative merit of search algorithms as used in practice. The gap between the trees searched in practice and in simulations is large. Simulating search on artificial trees that have little relationship with real trees runs the danger of producing misleading or incorrect conclusions. It would take a considerable amount of work to build a program that can properly simulate real game trees. Since there are already a large number of quality game-playing programs available, we feel that the case for simulations of minimax search algorithms is weak.

The consequence is that the only ideas that have merit in the community are those that are experimentally demonstrated in a real program. A new search idea must be tested over a sufficiently large test set and either demonstrate efficiency benefits (smaller tree size) and/or improved application performance (larger percent of correct moves).

Games were the first artificial-intelligence applications to address the critical issues of real-time performance. Real-time constraints, of necessity, requires experimental work. Game-playing programs must make dynamic decisions where to allocate their time resources during a game. The only way to properly simulate the eventualities is extensive testing. Even then it is not enough. There is no usable theory on which can draw on; "what works well in practice" is the only heuristic applicable for practitioners. Nevertheless, the artificial intelligence community is researching this area (anytime algorithms) and some of their results may one day be applicable to game-playing programs. So far the work on anytime algorithms is largely theory, and the few experiments that test the theoretical assumptions often show that these can be wrong. In effect, some of the lessons learned by the game-playing community are being rediscovered elsewhere.

2.2 Knowledge

Critical to the success of most game-playing programs is "knowledge." Note the qualification on the word knowledge. While some of the information in the program is factual (for example, it follows from the rules of the game), much of it is heuristic. Heuristics are rules of thumb: usually right, sometimes wrong. Many heuristics have little relationship to human knowledge; they tend to be either loose mathematical approximations of human concepts, or non-human-like terms that have proven to empirically work well in practice. For example, experiments in chess and checkers have shown that an evaluation function consisting of material (piece count) plus a small random number results in reasonable play!

When you are designing or improving an evaluation function, how do you know whether a heuristic is good or not? How often is it relevant? How important is the heuristic when it is relevant? What is the cost of computing the heuristic? All of these questions must be answered and the results weighed before anything is added to an evaluation function. None of these questions can be answered analytically; no good theoretical models exist (yet). These questions must be answered experimentally. And, once the knowledge is in the program, experiments are then necessary to determine whether the benefits of the new addition (reduced search, improved evaluation) offset the costs (increased search, increased execution time, exceptions).

The above would be simple if the effect of every change to an evaluation function could be assessed independently of the previous ones. Regrettably, this is not the case. An evaluation function may contain hundreds of heuristics, each of which is assigned an importance measure. Adding, deleting or changing a heuristic or changing a heuristic's importance can dramatically affect the dynamic co-operative balance that is essential in a good evaluation function. Extensive experimentation is necessary to understand the affect of any change on the balance. Various main-stream, artificial intelligence techniques have been explored in attempts to automate this above process. Regrettably, the successes are few. Building a quality evaluation function is usually a lot of hard (manual) work and luck.

The most common approach to improving the knowledge in a program is to perform a large number of experiments (play games, analyze positions), comb through the data, identify any deficiencies in the program's performance and then use that information to improve the program's knowledge (or search). In other words, the experiments drive the knowledge engineering.

2.3 Other Aspects

Besides search and knowledge, experimental computing is required in many other places in the program design. A few illustrative examples include:

1. Parallel processing. The parallelism in most search algorithms is non-deterministic. The real-time constraint introduces more non-determinism. The result is that parallel game-playing programs are very difficult to debug. There are usually a large number of parameters that influence parallel performance, and deciding on the winning combination is only possible by using experimental results to guide you through the maze of possibilities. In effect this is similar to the problem of tuning the evaluation function.

   Regrettably there are a large number of published parallel algorithms that promise high performance in theory, but fall far short when exposed to the reality of a real implementation. Again, mathematical analysis and simulations misrepresent what is possible.

2. Opening book. The search algorithm and knowledge in a program uniquely define the program's "personality." As with any game player, the program will have strength and weaknesses. The opening moves of a game are usually retrieved from a pre-computed database (often called the *book*). The choice of the opening moves must emphasize the program's strengths, not its weaknesses. It does not seem feasible to identify the personality of the program mathematically; it must be generated by careful observation by a trained eye that examines a large sample of the program's play. This involves considerable computing to generate the data and, having decided on the opening book strategy, verify that the strategy works well in practice.

3. Compiler performance. As a pragmatic issue, high performance also includes getting the most from a compiler. Many practitioners (particularly commercial efforts) devote considerable resources to restructuring code and data to maximize program speed. An enormous number of compute cycles are spent benchmarking various program changes looking for the combination that maximizes the program's speed. This type of code hacking, while not scientific, is nevertheless an integral part of the construction process for many programs.

3 Experimental Methodology

A major problem in the field has been a lack of widely-accepted, standardized benchmarks. Many have been proposed, but most suffer from one or more deficiency (too small, inadequate representation of the search space, biases, etc.). As well, many of the developers of game-playing programs have commercial aspirations. Misrepresentations occur because sometimes commercial programs are tuned to perform well on well-known benchmarks, yet demonstrate poor performance otherwise. The result is that there are no universal benchmarks, and all reported results must be regarded with suspicion unless they have been independently verified.

In practice, the only experimental data points that matter is program performance under tournament conditions. Since 1970, there have been annual computer chess tournaments. Most other games now have annual machine-machine and man-machine competitions as well. Developers must expose their software to public scrutiny; fakery does not survive. These competitions are ideal, in the sense that they provide a realistic and level playing field, yet unfair, since the number of data points for a program are small and the sample might not be representative. One could argue that the annual ACM computer chess events represent the longest ongoing experiment in computing science history (Regrettably, these tournaments have now become sporadic. The World Computer Chess Championship is ongoing, and has been held triennially since its inception in 1974.)

4 Conclusions

Why is experimental work so important in the development of game-playing programs? Regrettably, the answer is that the alternatives approaches are, in this domain, immature; the gap between theory and practice is large. That's not to say that things won't change, but I don't see that happening for a long time. Many aspects of program development will always be fueled by experimental results.

In February, 1996, the *Deep Blue* chess machine played the World Chess Champion Garry Kasparov a six-game match. Although the human won the match, *Deep Blue* crowned itself in glory by winning a single game. One could argue that this match was the most heavily-publicized artificial-intelligence/parallel-computing event in history. Although this demonstration of the capabilities of artificial intelligence was for a "mere" game, this event has probably done more for increasing our awareness of the difficulty and the need for experimental computing science than any previous event.

The enormous effort required to build game-playing programs such as *Deep Blue* (chess) or *Chinook* (the World Man-Machine Checker Champion) is worth discussing. I can speak personally of the *Chinook* project which took six years, during which as many as 200 computers were simultaneous running computing new results, validating others, experimentally testing hypothesis, etc. That so much effort is required to build a "mere" checker-playing program speaks volumes of the difficulty of creating machines capable of exceeding human intelligence. However much was involved in the *Chinook* project, the investment in the yet-incomplete *Deep Blue* project is many fold greater.

As a consequence of the heavy reliance on experimental science in developing high-performance game-playing programs, some people assume that this work is unscientific; a waste of effort. We contend that it is the only way to do this kind of science. The real waste occurs when the experiences of the program developer aren't used to further our understanding of the problems involved and their solutions. For example, the experimental work could be used as an input for theoretical work, to help construct models that would ease the work of the experimenter and benefit the scientific community.