Automatic Generation of Search Engines

Markian Hlynka and Jonathan Schaeffer

Department of Computing Science
University of Alberta, Edmonton, Alberta, Canada
{markian, jonathan}@cs.ualberta.ca

Abstract. A plethora of enhancements are available to be used together with the $\alpha\beta$ search algorithm. There are so many, that their selection and implementation is a non-trivial task, even for the expert. Every domain has its specifics which affect the search tree Even seemingly minute changes to an evaluation function can have an impact on the characteristics of a search tree. In turn, different tree characteristics must be addressed by selecting different enhancements. This paper introduces PILOT, a system for automatically selecting enhancements for $\alpha\beta$ search. PILOT generates its own test data and then uses a greedy search to explore the space of possible enhancements. Experiments with multiple domains show differing enhancement selections. Tournament results are presented for two games to demonstrate that automatically generated $\alpha\beta$ search performs at least on a par with what is achievable by hand-crafted search engines, but with orders of magnitude less effort in its creation.

1 Introduction

Programs which play two-player games of perfect information typically rely on two things to achieve high performance: an evaluation function and a search algorithm. The evaluation function estimates the desirability of a static board position. The search algorithm, using the evaluation function as a guide, acquires knowledge dynamically: it looks ahead through the possible lines of play and selects the one leading to the highest achievable evaluation given the search constraints. $\alpha\beta$ pruning [10] is the mainstay of this sort of game program. However, designers of high-performance search engines have long observed that while the pseudo code for $\alpha\beta$ is approximately 20 lines, an actual high-performance implementation of $\alpha\beta$ can run upwards of 20 pages of code. The reason for this disparity is straightforward. Since the development of $\alpha\beta$ pruning and the decision to pursue search as the cornerstone of two-player game research, most work in two-player games has been in the area of search enhancements [15]. While the choice of $\alpha\beta$ for a two-player perfect information game is trivial, the implementation for maximum performance on a particular problem is not. In practice, there is a plethora of enhancements which allow the basic search algorithm to focus its attention further on the areas of the search space most likely to produce useful information, while eliminating areas of the search which seem irrelevant.

H.J. van den Herik et al. (Eds.): ACG11, LNCS 4250, pp. 23–38, 2006.
© Springer-Verlag Berlin Heidelberg 2006

Search enhancements fall roughly into four categories: those that change the order in which successors are considered at each node in the tree; those that focus the search by dynamically extending or reducing the search depth; those that adjust the $\alpha\beta$ bounds to reduce the amount of search required; and those that employ some combination of memory or caching to avoid repeating work or to access knowledge acquired outside of the current search.

For each of these categories there are numerous enhancements in the literature. Some seem to work well in most domains. Others seem to be more specific, having their greatest effect on a class of domains, and less or detrimental impact on other domains. It is also pertinent to note that the word "domain" in this sense does not necessarily have a one-to-one correspondence with "game". That is, games typically pass through several phases in the course of normal play. These phases may differ significantly in their distinguishing features, and thus could be said to correspond to different domains. This is why high performance game programs often have multiple classes of settings corresponding to this changing nature of a game. Thus, different enhancements may be applicable in different phases.

When building a high-performance search engine for a particular game domain, it is not the choice of the algorithm that the programmer finds costly and time consuming. Rather, it is in the selection, implementation, testing, debugging, and interaction of the available search enhancements that the real work lies.

Consider the scenario of a programmer developing a high-performance search engine to play a new game. Currently, such a programmer is faced with a daunting task. The programmer must painstakingly sort through all the known search enhancements for the ones which will most benefit the new domain. The enhancements are sensitive to many potential differences between domains. They usually have parameters which require tuning for good performance. If the domain happens to be relatively new and unknown, this further exacerbates the difficulty of the task. Thus, the work is driven by intuition and experimentation, resulting in hundreds of hours of programmer and computer processing time [9,14].

The next problem our imaginary programmer faces is that different enhancements interact in various ways. Sometimes these interactions are well defined. However, more than enough enhancements exist to make a thorough list of interactions infeasible if not actually impossible. Of course, like the enhancements themselves, these interactions are also tied up with the domain to which they are applied. To make the problem interesting, the areas covered by each enhancement are not discrete and self-contained; different enhancements overlap in their goals as well as their methods.

The final problem is that most enhancements contain parameters that must be further tuned for maximum performance in the domain. Poor parameter selection could result in a highly beneficial enhancement being perceived as useless. Selecting the correct parameters is frequently at least as difficult as choosing the enhancement in the first place. And, of course, this must be done not just individually, but also in the context of all the other enhancements present:

their parameters may interact in increasingly complicated ways. For example, enhancements A and B may individually produce their best results with parameter sets A' and B' respectively. However, using A and B simultaneously with these same parameters may produce unsatisfactory results. Not only might the combined effect fail to increase performance, it may even decrease it.

Current methods (i.e.,"by hand"), however cumbersome, do work. Experts painstakingly "hill-climb", carefully implementing one enhancement at a time, evaluating it, and only accepting it if it yields better performance. That this approach works in practice is evidenced by such stunning successes as CHINOOK, DEEP BLUE, LOGISTELLO, and others.

Fürnkranz [7] identifies the learning of techniques to control search as an area which, "surprisingly... is more or less still open research in game-playing." Little work has been done to automate the task of devising a search algorithm along with its various enhancements for a particular purpose. Rather, most work has been focused in the area of search control, which refers to methods that allow a finer granularity of control over the search algorithm. The main forms of learning search control are focusing the search with extensions and reductions, modifying parameters which control the search, and using acquired knowledge to direct the search. These are exemplified by the work of [2] and [5].

Cook and Varnell [6] present an algorithm in which changing certain parameters modifies the basic behaviour of single agent parallel A* search. They call these variations search strategies, and their work is probably most similar to that presented herein. Indeed, Schaeffer et al. [18] posit that single-agent and two-player search are different sides of the same coin. Other related attempts have been made, such as that of ZILLIONS OF GAMES [19], which builds an evaluation function based on a game description. ZILLIONS's approach might complement the presented work, PILOT, nicely: we are working on high performance search, while ZILLIONS seems to focus on the evaluation. Unfortunately, further details on how ZILLIONS functions is unavailable.

The problem is simply stated: how does one find the right combination of enhancements, along with their associated parameters, to maximize the search performance of a game-playing program? Why not devise a system to automate this difficult task? Such a system could characterize the features of new search domains. It could then use this acquired knowledge to learn correlations between various features of the domain and search enhancements. The system would then test its hypotheses to confirm their accuracy. Upon completion of processing, the end result of the system would be a search algorithm, complete with enhancements, thoroughly tested, and specifically tailored to the required domain. Most importantly, all this would be accomplished without human intervention![1]

To this lofty goal this paper contributes the following. An implementation framework for two-player perfect-information games, wherein the search algorithm is separated from the domain-specific knowledge. This gives a game programmer two immediate advantages: implementing new games takes only one

[1] Save for the obvious human work required to define the new domain, determine the bounds, requirements, and evaluation function.

or two days, and each game immediately has access to the full set of available $\alpha\beta$ search enhancements.

A prototype program called PILOT is presented. PILOT is able to choose a set of search enhancements, compile a game program with these enhancements, automatically generate test sets for a specific domain (game), and perform a greedy search through the space of search enhancements to determine the best set for the particular domain in question.

Results are presented for two non-trivial games, Ataxx and Lose Checkers, as well as reports on some experiments with the (also non-trivial) games of Awari and Critical Mass.

2 PILOT

PILOT is the name given to a system which oversees the choice of $\alpha\beta$ search enhancements appropriate to a specific domain. PILOT is the control program responsible for *piloting* the various components which comprise the system. PILOT's job is (1) to generate a search engine by selecting search enhancements, (2) to create and execute tests to determine their effectiveness, and (3) to generate further search engines based on the data thus acquired. This process iterates until a user-imposed constraint is met, or until PILOT reaches a steady-state beyond which it cannot improve search performance.

2.1 Interfacing with PILOT

As stated in the introduction, PILOT includes an implementation framework which separates the search algorithm from game specific knowledge. This greatly speeds the process of developing a new game. To this end, a programming interface is provided to the developer of a new game. The programmer must provide the system with a number of standard functions and data types which are specific to the application. These fall into three main categories: rules, knowledge, and performance hooks.

Functions in the rules category implement the rules of the game. They include functions for making moves, unmaking moves, and move generation. The game evaluation is the main member of the knowledge category. While building a good evaluation function is difficult, this is not in the scope of our work. Search enhancements can be implemented in a generic fashion, but this can severely impact performance. Performance hooks rectify this by allowing the programmer to use domain-specific knowledge. Providing an incremental hash function for a transposition table is an example of this.

2.2 Components

PILOT's main modules are illustrated in Fig. 1. The modules are code generation, evaluation, analysis and learning, and enhancement selection.

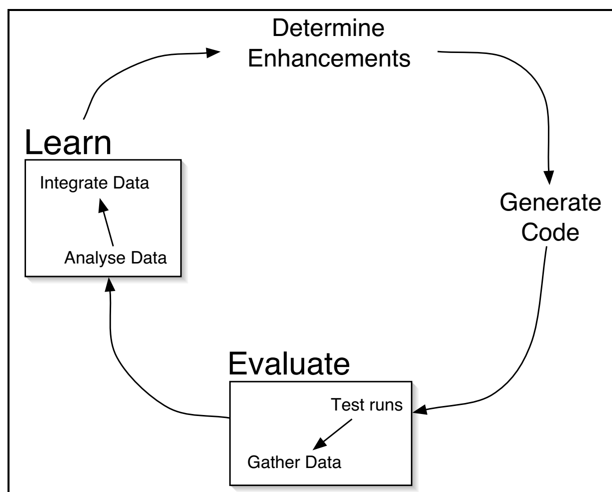

Fig. 1. The PILOT cycle

Code Generation. PILOT depends upon a code generator. Thus, the first task is to design a system in which the search algorithm and enhancements are written in a generic fashion, and can be easily enabled and disabled. It must also be easy to add new enhancements. Instrumentation must be embedded into this design to allow for the gathering of various measures of performance. Given such a framework, the process of designing a high-performance search engine for a specific application would be akin to fitting together pre-constructed blocks of code. Obviously there is a benefit in the fact that the code is already tested and the pieces are known to interact correctly. Nevertheless, there remains the problem of how to fit the provided pieces together in order to best accomplish a specific task.

PILOT currently generates code using the standard C/C++ preprocessor. Different enhancements are enabled and disabled via manipulation of a header file. The advantage of this approach is the speed of the generated code. While this approach seems the most straightforward, it is nevertheless challenging to code various enhancements independently such that any combination will produce correct code. This is not a simple matter of additive coding, as enhancements can have issues of dependence, precedence, mutual exclusion, and so forth. The prototype contains up to eight enhancements (with plans to increase this number), discussed in Subsection 2.3. Additionally, it has been instrumented to acquire a number of useful tree statistics.

The strength of this method is that it need not be merely a tool for automation with PILOT. The code generation can be leveraged for other tasks, such as more efficient evaluation of new enhancements, a starting point for expert programmers, or a tool for beginners with which to explore search algorithms.

Evaluation. Once the code generator produces a search engine for the task at hand, PILOT enters the evaluation phase. This phase consists of assessing the

search engine and gathering data from which intelligent decisions can subsequently be made.

Pilot can evaluate the search engine in several ways. Currently, assessment is based on an evaluation suite. Suites are comprised of a number of move sequences. Pilot has the capability to generate these suites for itself as needed, use test cases supplied by a human, or use a combination of both.

The purpose of the experiments detailed in this paper was to verify the effectiveness of the entire cycle in the context of tournament performance against programs written by humans. Thus, the following settings might be seen as modest. Evaluation suites were generated using a plain $\alpha\beta$ search engine for the given game. From the initial position, the search engine played some number of moves into the game using some consistent search depth. The position thus reached becomes the starting position for a sequence of related positions.[2] Several such sequences comprise an evaluation suite.

When the evaluation suite is used, each initial position in a sequence is set up, and the search engine evaluates each position in the sequence. It is possible that the search engine will generate a different move from the one that leads to the next position in the sequence. Pilot notes this and continues with the next *predetermined* position in the test set.[3] The detection of a different move selection may be used in the evaluation process at a later date. This process repeats for every sequence in the evaluation suite.

In carrying out the evaluation, two sorts of numbers can result: performance metrics and search tree statistics. Performance metrics evaluate the effectiveness of a particular search engine; that is, how well it performs. Search tree statistics quantify the characteristics of a search tree. The combination of these two types of data is used to suggest what enhancements to try on the next iteration of Pilot's learning cycle. The line between these two aspects is not necessarily sharp: a search tree statistic might be used as a performance metric. Nevertheless, performance metrics compare different search engines with an eye to which will win more frequently. Search tree statistics are values which *may* correlate to the performance.[4] Thus, performance metrics indicate which search engine will win, while the search tree statistics may tell us why.

For the experiments reported in this paper, search tree size was chosen as the performance metric. This choice was made for its initial simplicity: a smaller tree indicates a more efficient search.[5] Thus, as a baseline, a smaller tree is better. Clearly there are trade-offs. For example, if generating a smaller tree takes twice as much time as generating a larger tree, then the larger tree may

[2] This is important because some enhancements require a context in which to operate, and testing them on randomly generated, unrelated positions would be ineffectual.

[3] That is, each search engine is tested on the same sequence of related positions. This keeps testing consistent.

[4] Though how they correlate may not be readily apparent.

[5] It is significant to note that this depends on the enhancements used. The tree may become smaller due to enhancements which induce more $\alpha\beta$ cutoffs; or search-reduction enhancements may directly reduce the size of the tree. These are not necessarily equivalent, and this is under investigation.

arrive at the desired result first! Initial experiments with a generic hash function for the transposition table showed precisely this disparity, and indicated that a domain-specific incremental hash function was an essential addition to the code if transposition tables were to be evaluated fairly. Similarly, adding a search extension increases the tree size for a search of some nominal depth. In this case, the size of the search tree does not convey any gain: the tree appears larger!

More interesting performance metrics include move quality and time elapsed. However, this makes comparisons a non-trivial matter, especially as the quality of a move is not easy to define. Tournament results or self-play may also prove a good performance indicator. Thus, while these are beyond the scope of the current experiments, there is clearly potential for future work.

Analysis and Learning. In Fig. 1, PILOT's next phase consists of analysis and learning from the experimental data. This involves making intelligent decisions from the statistics generated in the evaluation phase and integrating it with the acquired knowledge of the system. The current prototype of PILOT, however, is much more simplistic. It executes a straightforward hill-climbing algorithm based on search tree size, which is exactly what a human would do. While PILOT *is* currently able to gather detailed search tree statistics, this ability is not used in these initial experiments. Thus, this section is provided to indicate some of the intended directions of future work. PILOT's current hill-climbing approach is discussed in more detail in the next section.

Enhancement Selection. In the enhancement selection phase, PILOT determines what set of search enhancements to try in the next iteration. This phase is closely related to both the code generation and learning phases. In the initial stage of the search, PILOT can start either from a plain, unenhanced $\alpha\beta$ search or from a user specified starting point – a 'suggestion'.

Future incarnations of PILOT, upon arriving at the enhancement selection phase, will use acquired knowledge to direct the choice of what to try next. However, for the sake of an initial deterministic and easily understood baseline, PILOT currently performs a greedy search: from any state S, each unused enhancement is tested in turn (currently without the benefit of parameter tuning). The enhancement, e, which results in the greatest reduction in the size of the search tree compared to S, is enabled. That is, a new state $S' = S + e$ is created. This process is iterated with S' as the new starting state until no further improvement is possible. Thus, PILOT hill-climbs in much the way a human expert would,[6] but without human intervention, and with the potential to run many more tests than would be feasible for a human.

[6] With one difference. A human will sometimes *disable* an enhancement, even after it has been shown to be beneficial. Such testing might find a combination of lesser enhancements which might increase performance when interaction with another enhancement is removed. The current version of PILOT will miss these opportunities: PILOT sees the starting state only as something to which enhancements can be added, not from which they can be removed. It was felt that this was a reasonable decision in the context of the current set of enhancements. However, PILOT is rapidly outgrowing this decision, and this capability will be added.

Thus, while we hope to see good enhancements selected, we also hope that not *all* enhancements are selected, or the benefits will be unclear compared to simply selecting all enhancements. However, this is partly a function of the evaluation method used. Since we use search tree sizes, and most enhancements reduce the size of the tree, a certain amount of correlation must be expected. Nevertheless, early experiments with the games of Awari and Critical Mass confirmed the effectiveness of this method by demonstrating different enhancements being selected depending on both the domain and the evaluation technique.[7] For example, the transposition tables in Awari were implemented generically, and were therefore computationally expensive. Therefore, when PILOT evaluated this domain using timing data, transposition tables were seen to degrade search performance, and PILOT decided against using them.

Having determined which (if any) enhancements to use, PILOT generates a header file to enable the appropriate enhancements. Then, the code generator is commanded to build a new search engine with the indicated set of enhancements. It is PILOT's job to ensure that enhancement dependencies and prerequisites are met for any set of enhancements it passes to the code generator.

2.3 Enhancements

PILOT currently supports the following set of basic $\alpha\beta$ enhancements: transposition tables (TT), transposition table move ordering (TTO), history heuristic (HH), killer heuristic (KH), iterative deepening by 1 or 2 ply (ID1 or ID2), extending the search by 1 ply in positions with only a single move (EXT), and Principal Variation Search (PVS).

Transposition tables are the most commonly used method of information caching [13]. They are an important enhancement because a transposition table entry can serve three main purposes: narrowing of $\alpha\beta$ bounds, a source of move ordering, and a source of immediate cutoffs in the search. Parameters that can be learnt include the size of the table,[8] the replacement scheme of the hash table, and when to reset the table. These are parameters which future versions of PILOT will be able to address in more detail.

As just mentioned, transposition table move ordering is one use of a transposition table. At its simplest, it involves searching the move retrieved from the transposition table first. However, enhancements like ETC [17] and Presearch [8] provide information which might be used to further order moves at a node. Additionally, there exists the possibility of determining precedence between different move-ordering enhancements. Presently, PILOT uses a static precedence among move-ordering techniques, though we intend to alter all these parameters in future experiments.

[7] For completeness, experiments were also performed on a 'random' game where all positions and moves were uncorrelated. A random position generated a random number of random moves which led to random successor states. PILOT was unable to determine any enhancements to be useful in this 'game'.

[8] Larger is not necessarily better [11,17].

The history heuristic [13] is a mechanism whereby the moves in the search have an associated global score which is incremented when a move is determined to be the best in some position. When a new position is encountered, the history score of each move available at that position is used as an indicator of which ones are likely to be best. Whereas a transposition table stores the exact context of a move, the history heuristic is a table of moves with associated scores indicating the frequency of their 'goodness'. Parameters which might be generated automatically include the increment factor, the scaling factor, and the frequency of scaling.

By contrast, the killer heuristic [1] stores some number of refuting moves for each level of the search tree. The killer heuristic can be viewed as a special case of the history heuristic. However, the killer heuristic uses less memory and might be preferred in certain situations. Parameters to tune for the killer heuristic are the number of killer moves stored per ply, and the replacement scheme.

Iterative deepening's main purposes are for move ordering at the root of the search, and for operating under time constraints. While iterative deepening might be used to assist in move ordering at interior nodes, the main parameter of interest is the amount of the increment between iterations: should the search iterate by one ply at a time, two, or possibly more? Changing this value can help to balance odd-even effects of the evaluation function. Being able to determine it automatically when providing a new evaluation function would be useful. Currently, Pilot chooses between iterating either one or two ply at a time.

Search extensions are useful in situations when it is clear that a position is unresolved, or when the choices available are too minimal to be likely to provide useful information. In these situations, search is extended an additional ply in an effort to overcome the horizon effect. A straightforward search extension is to extend the search in any position where there is only a single available move. Since it takes no effort to choose this move, further effort can be spent looking deeper instead. However, two parameters are of utmost importance: the amount of the extension (i.e., extend the search by *how many* ply), and the choice of a maximum bound. That is, if an extended position leads to a position which is also extendable, at some point this process must be curtailed, lest the current line be searched to unreasonable depth at the cost of the rest of the search.

Principal Variation Search, also known as minimal-window, null-window, or zero-window search, is a basic windowing enhancement which improves on alpha-beta by using the smallest possible bounds to search moves which are not in the main line. Re-search may be required if the bounds do not hold, but the minimal bounds allow for small reductions in the tree. The more stable the main line is, the better PVS fairs.

3 Experiments

To evaluate the feasibility and effectiveness of Pilot's approach, several games were written within the Pilot search framework. These include Tictactoe, Hexpawn, Awari, Critical Mass, Halma, Lose Checkers, and Ataxx. Additionally, a

random game was also created, as discussed briefly in Section 2.2. Of these, Tictactoe and Hexpawn are trivial games. The others, however, are non-trivial. It is also worth mentioning the increasing ease with which games can be added to this system. Each of the games here was implemented in day or two, and this is typically all that is now required to have a fully functioning game! The difficult part, of course, is coming up with a user-supplied evaluation function. For all games here, a simple evaluation function based on material difference was used. This allows PILOT to be evaluated solely in terms of the search it generates and not on the cleverness of the evaluation.

Awari is fairly well known in the games research community. It is played on a 2×6 board. Two players sit opposite the long sides of the board. Initially, each 'pit' contains 4 stones. Players take turns sowing their stones around the board in an effort to capture the most stones. Awari was solved by Romein and Bal [12].

Critical Mass is played on a 5×6 board, though other sizes are possible. Each square is assigned a 'critical mass' (CM) number which is equal to the number of adjacent horizontal and vertical squares. Starting with an empty board, players alternate in placing 'protons' on any square that is empty or already occupied by that player's own protons. When the number of protons on a square exceeds its CM, the square 'explodes': it becomes empty, and exactly one proton is added to each adjacent horizontal and vertical square. If any of the adjacent squares belonged to the opponent, they now belong to the current player. An explosion can lead to other squares exceeding their CM. Explosions are continued until no square is above critical mass. The game ends when one player owns all the squares on the board.

Lose Checkers is checkers where you play to lose all of your pieces.

Ataxx is played on an $n \times n$ board. Players take turns making either jump moves or cloning moves. The latter creates a new piece on an adjacent square, while the former moves a piece to a new location with a distance[9] of 2 from the starting square. The goal is to have the most pieces at the end of the game, and the game ends when there are no legal moves for either player, a position has repeated three times, or 50 jump moves have been made in a row [3].

3.1 Experiments Summary

The experiments reported here cover several phases in PILOT's development. The initial thrust was to test PILOT's greedy search strategy on two nontrivial games, Awari and Critical Mass. As PILOT was developed, more enhancements were added. After the initial stages of development, it was determined that it was necessary to evaluate PILOT in comparison to hand-crafted search engines. Assignments from a graduate course in heuristic search were made available for the games of Lose Checkers and Ataxx. These programs were hand-crafted for a course tournament, and thus represent an intensive month's work on the part of each student. Previous experience in this environment shows that the

[9] In at least one of the x or y components.

student programs typically range from extremely good (far outstripping any human players) through moderate to poor. Thus, it was determined that this was an excellent testbed in which to evaluate PILOT's automatic search generation.

We have two types of results for PILOT: the results of the enhancement selection process, and the results of the two tournaments. In the former, we would like to see different results for different games. In the latter, we would like confirmation that PILOT's automatic generation of a search algorithm does a comparable (or better) job than graduate students.

PILOT's results for the enhancement selection process are illustrated in Table 1. An interesting side-effect of the enhancement selection process is the order in which the enhancements are selected by the greedy search. This ranking, therefore, is an indication of the relative amount of benefit each enhancement adds to the particular domain. This has been indicated in the table. These experiments were produced by running PILOT through its cycle as discussed in Section 2. Specifically, PILOT generates and runs evaluation suites, as in Subsection 2.2. Evaluation suites were generated with fixed depth searches at one depth, and then used for search-engine assessment with an equal or deeper (but still fixed) depth. Since different domains can be searched to widely varying depths, for these experiments the depths were chosen manually for each domain in order to control the running time. As an example, Awari generated evaluation suites with a depth-5 search, and used the resulting suites with depth-10 searches.

Table 1. Enhancement selection by PILOT in a number of games and situations

	Enhancements							
Game	TT	ID1	ID2	TTO	HH	KH	EXT	PVS
Awari	1	3	—	2			—	—
Critical Mass	1	4	—	3	2	—	—	—
Critical Mass	2	5	—	4	3	1	—	—
LoseCheckers	1	+		3	2		—	—
LoseCheckers (PVS)	1		4	2	3			5
Ataxx (shallow)	3		2	4	1		5′	—
Ataxx (deep)	2	3		4	1	5	6′	—

Each row of Table 1 indicates a game. The columns indicate which enhancements are enabled. The enhancements are identified by the abbreviations noted in the list in Subsection 2.3. Each cell in the table has one of the following indicators:

$1 \cdots n$ A number indicates the order in which the enhancement was added to the search.

— Indicates an enhancement that was not yet implemented at the time of this experiment.

A blank space indicates an enhancement that was *not* selected for this experiment.

+ Indicates an enhancement not selected by PILOT, but turned on for the time-controlled tourments.

n' Indicates an enhancement that was chosen not because it caused an improvement, but because it was not detrimental in the tests. This is essentially an indication that the tests in these cases were not sufficiently broad.

It should be noted that there are multiple experiments reported for Critical Mass, Ataxx, and Lose Checkers. The difference in the Critical Mass experiments is that in the first one, the killer heuristic had not been implemented. Adding this enhancement to Critical Mass caused it to be selected first, which is noteworthy. Similarly, Lose Checkers (PVS) is the most recent addition, and in addition to the PVS enhancement it includes a re-written search engine, the combination of which account for the differences.

Due to Ataxx's high branching factor, a shallow run was done with PILOT where test positions were searched to a depth of 3. Later, a deeper test was done with positions searched to a depth of 6. The table shows two differences: a different iterative deepening increment, and the killer heuristic becoming useful in the deeper searches where it had been left out in the shallower tests. This is significant because it hints at the changing characteristics of the search tree as the depth increases, and also because the shallower search approximates the deeper one. (Only one enhancement is different, and the relative orders are the same.) This phenomenon bears further study, as it may be related to observations by [16] that for best performance the search depth at the time of learning should match the intended search depth (e.g., for tournaments).

It is also interesting to note that the transposition table is not always the first enhancement selected. When the games involved are considered, this may be explained. In the case of Ataxx, certain moves, or types of moves (jumps versus non-jumps) are extremely important. If the history heuristic is learning to favour one over the other, this total move ordering might be more effective than elimination of cycles, which are less prominent in Ataxx. Similarly in Critical Mass, transpositions are not as vital, since a single move can change the configuration of the entire board. Certain key moves will initiate a massive chain reaction across the entire board. Searching these moves first can save significant search effort.

By contrast, Lose Checkers is a more classical board game involving a diminishing number of moving pieces. We would expect transpositions to be prevalent, and indeed they are the first enhancement chosen. Since PILOT is selecting enhancements solely on their ability to reduce search-tree size (independent of time constraints), iterative deepening is at a disadvantage and therefore not selected.[10] Similarly, EXT could not be chosen by PILOT in "Lose Checkers (PVS)" because EXT typically increases the tree size in an effort to enhance the *quality* of the search; PILOT cannot yet understand this distinction.

[10] A related issue is that early versions of the search engine may not have ordered root moves effectively when interacting with ID. This would affect ID's perceived usefulness. However, this favorably reflects on PILOT's sensitivity to the minutiae of implementation details. Later experiments bear out this sensitivity.

Thus, we see how PILOT not only suggests different combinations of enhancements for different games, but might give insights into why these choices are appropriate.

3.2 Lose Checkers Tournament Results

Lose Checkers has proved to be a most interesting game. PILOT's search-engine configurations are summarized in the previous section. For the reported tournament the configuration determined in "Lose Checkers (PVS)" in Table 1 was used.

The Lose Checkers tournament was played against sixteen programs. These programs had already played in a full double-round-robin tournament using 20 seconds per move. The most recent experiment had the following results.

Against 16 programs, PILOT scored 9 wins, 10 losses, and 13 draws, searching to an average depth of 21.6 ($\sigma = 4.3$).[11] Scores were assigned using the formula of one point per win and half a point per loss. Thus, PILOT scores 15.5 points. These data were then combined with the full-tournament result to determine PILOT's relative ranking.

PILOT ranked in seventh place in this combined result. Looking at the spread of points across the tournament shows that there are distinct groups of programs. The most exceptional programs scored between 27 and 32 points. The next group scored between 20 and 23, and a third group from 13.5 to 15.5. The next two groups scored between 10 and 11, and below 4.

Observations. Subsequent discussions with a few of the students who wrote the best programs were illuminating. The top programs, for example, used a random evaluation along with a parity checker which could solve one-on-one piece endgames. Their experiences indicate that a material evaluation such as PILOT's was frequently detrimental. It also appears that in Lose Checkers, deeper search is not necessarily correlated with success, particularly when using material evaluation.

PILOT ranks at the top of the third group of programs. Note that the second and third groups consisted of programs that are moderately strong, as opposed to the highly tuned programs of group one. Given PILOT's simple material-only evaluation, and that the other programs had significant time to come up with a search and evaluation combination that would be better than simple material, PILOT's performance is commendable. An examination of the games shows that PILOT generates deeper search than the majority of its opponents, and its limiting factor appears to be the evaluation. However, for these very reasons, Lose Checkers is an ideal testbed for PILOT: a subsequent challenge is to develop improved analysis and evaluation suites which better reflect tournament conditions.

3.3 Ataxx Tournament Results

For Ataxx, a simple depth extension was available to PILOT. Table 1 shows the learned result. The second ('deep') result indicated in the table was used for

[11] Each program plays black once and white once.

the tournament. It is rare in Ataxx that only a single move is available, and if it happens, it is usually in the endgame. It is clear from the fact that Pilot detected no change when using the depth extension that our automatic test generation is not comprehensive enough.

The Ataxx tournament originally consisted of 21 programs. Unfortunately, only 8 were available for this experiment. However, these 8 programs were a representative sample across the entire gamut. Thus, it is with reasonable confidence that this result is presented as representative.

The procedure was similar to Lose Checkers. The original Ataxx tournament was run with 15 seconds per move. The tournament with Pilot also used the same 15 second move limit. No draws occurred in either the Pilot tournament or the original Ataxx tournament.

Pilot tied for fourth place among nine programs with 10 wins and 6 losses, searching to an average depth of 7.9 ply ($\sigma = 0.94$). The top program had 16 wins, while the next group of four programs had from 10 to 12 wins. The final four programs had between 0 and 5 wins. Thus, Pilot again achieved a ranking in the second grouping: not outstanding, but in with the best. Once again, it is important to remember that Pilot is competing against programs which had their search and evaluation functions tuned specifically to Ataxx. Pilot is purposely handicapped by a simplistic material-difference evaluation in order to gauge the effectiveness of the search enhancements.

The rankings of the 8 opponents in the original tournament from best to worst, were: 1, 3, 5, 6, 9, 10, 16, and 20. Thus, the stronger programs were clearly better represented. It was somewhat surprising, therefore, when Pilot won and lost one game each against the third and fifth ranked programs; we were quite pleased.

4 Conclusions and Future Work

It is encouraging that even with some of the restrictive limitations placed on the prototype of Pilot, it was able to generate a search powerful enough to rank with the second class of hand-crafted search programs. When this is combined with the fact that Pilot's code, of necessity, must sacrifice a certain amount of speed for flexibility of implementation, and that domain-specific tuning of the evaluation function was nonexistent, the results seem even more promising.

Additionally, it has been shown that even using a greedy search across the space of available enhancements, Pilot offers insights into individual domains both by what it selects, and the manner in which it does it. Some of the results presented here suggest that Pilot might eventually make it possible to generate not just a single search engine for a game, but multiple search engines tailored to different phases of the same game. This is a practise that has remained difficult for humans.

Thus, there are plenty of areas for future exploration. With the demonstration of the basic soundness and effectiveness of Pilot's approach, several avenues present themselves. The most pressing requirements follow. While eight

enhancements are a good start, they are not yet representative of what a human programmer can use. More enhancements are needed. Also required is a new system of evaluation more powerful than search tree size. This new method might consider time, move quality, and tournament or self-play results. The current greedy search is effective, but simplistic. It is tempting to see how far it can go, but a better method of exploration must be found as the number and types of enhancements grows.

A second important area is parameter tuning. In the course of coding and experimenting, it has been observed that minor changes in one enhancement (intentional or otherwise) clearly impact upon the entire process of enhancement selection, and in the selection or discarding of *other* enhancements. Thus, even the current version of PILOT is able to effectively perceive these subtle differences. What currently lacks is a mechanism for explicitly testing such configurations.

Finally, PILOT has the potential to generate a new search engine on a per-evaluation-function basis. Current results have used minimalist evaluation functions, and it will be extremely interesting to see what is possible with an evaluation function that has been tuned for a specific game.

The ultimate goal is to free humans from the drudgery of creating a new search engine for every new game that comes along. Freed of the requirements of endless implementation and testing, we will be able to focus on an exciting new set of problems.

Acknowledgements

We would like to acknowledge the assistance of the following people and organizations, and express our sincere thanks: Akihiro Kishimoto, for his development of and assistance with the Generic Game Server clients for Lose Checkers and Ataxx; Michael Buro, for making his GGS code available to the world at large [4]. This research was supported by the Natural Sciences and Engineering Research council of Canada (NSERC) and Alberta's Informatics Circle of Research Excellence (iCORE).

References

1. A. G. Akl and M. M. Newborn. The Principle Continuation and the Killer Heuristic. In *ACM Annual Conference*, pages 466–473, 1977.
2. Y. Björnsson. *Selective Depth-First Game-Tree Search.* PhD thesis, University of Alberta, 2002.
3. M. Buro. Rules of Ataxx, 2004. http://www.cs.ualberta.ca/ ˜mburo/ggsa/ax.rules.
4. M. Buro. Generic Game Server, 2005. http://www.cs.ualberta.ca/˜mburo/.
5. M. Buro. ProbCut: An Effective Selective Extension of the $\alpha\beta$ Algorithm. *ICCA Journal*, 18(2):71–76, 1995.
6. D.J. Cook and R.C. Varnell. Adaptive Parallel Iterative Deepening Search. *Journal of Artificial Intelligence Research*, 9:167–194, 1999.

7. J. Fürnkranz. Machine Learning in Games: A Survey. In Johannes Fürnkranz and Miroslav Kubat, editors, *Machines that Learn to Play Games*, chapter 2, pages 11–59. Nova Science Publishers, Huntington, NY, 2001.
8. M. Hlynka and J. Schaeffer. Pre-Searching. *ICGA Journal*, 27(4):203–208, 2004.
9. P. Hoffman. *Archimedes' Revenge: The Joys and Perils of Mathematics*. W. W. Norton & Company, May 1988.
10. D.E. Knuth and R. Moore. An analysis of Alpha-Beta Pruning. *Artificial Intelligence*, 6(4):293–326, 1975.
11. A. Plaat. *Research Re: Search & Re-search*. PhD thesis, Erasmus University, Rotterdam, June 1996.
12. J.W. Romein and H.E. Bal. Solving Awari with Parallel Retrograde Analysis. *IEEE Computer*, 36(10):26–33, October 2003.
13. J. Schaeffer. The History Heuristic and Alpha-Beta Search Enhancements in Practice. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, PAMI-11(11):1203–1212, 1989.
14. J. Schaeffer. *One Jump Ahead*. Springer-Verlag, New York, 1997. ISBN 0387949305.
15. J. Schaeffer. A Gamut of Games. *AI Magazine*, 22(3):29–46, Fall 2001.
16. J. Schaeffer, M. Hlynka, and V. Jussila. Temporal Difference Learning Applied to a High Performance Game. In *International Joint Conference on Artificial Intelligence (IJCAI-01)*, pages 529–534, 2001.
17. J. Schaeffer and A. Plaat. New Advances in Alpha-Beta Searching. In *ACM Computer Science Conference*, pages 124–130, 1996.
18. J. Schaeffer, A. Plaat, and A. Junghanns. Unifying Single-Agent and Two-Player Search. *Information Sciences*, 135(3-4):151–175, 2001.
19. Zillions Development Corporation. Zillions of Games – Unlimited Board Games & Puzzles, 1998–2005. http://www.zillionsofgames.com/.