Hindawi Publishing Corporation
Advances in Artificial Intelligence
Volume 2013, Article ID 357068, 9 pages
http://dx.doi.org/10.1155/2013/357068

Research Article

Tree Pruning for New Search Techniques in Computer Games

Kieran Greer

Distributed Computing Systems, Belfast, UK

Correspondence should be addressed to Kieran Greer; kgreer@distributedcomputingsystems.co.uk

Received 28 May 2012; Revised 10 October 2012; Accepted 24 October 2012

Academic Editor: Srinivas Bangalore

This paper proposes a new mechanism for pruning a search game tree in computer chess. The algorithm stores and then reuses chains or sequences of moves, built up from previous searches. These move sequences have a built-in forward-pruning mechanism that can radically reduce the search space. A typical search process might retrieve a move from a Transposition Table, where the decision of what move to retrieve would be based on the position itself. This algorithm stores move sequences based on what previous sequences were better, or caused cutoffs. The sequence is then returned based on the current move only. This is therefore position independent and could also be useful in games with imperfect information or uncertainty, where the whole situation is not known at any one time. Over a small set of tests, the algorithm was shown to clearly out perform Transposition Tables, both in terms of search reduction and game-play results. Finally, a completely new search process will be suggested for computer chess or games in general.

1. Introduction

This paper describes a new way of dynamically linking moves into sequences that can be used to optimise a search process. The context is to optimise the search process for the game of computer chess. Move sequences are returned during the searching of the chess game tree that cause a cutoff, or determine that certain parts of the tree do not need to be searched further. These move sequences are usually stored in Transposition Tables [1, 2], but instead, they can be stored in a dynamic linking structure and reused in the same way. They can return an already evaluated sequence of moves, which removes the need to search the tree structure that would have resulted in this move sequence. The term "chain" instead of "sequence" will be used to describe the new structure specifically. Move sequence is a more general term that can be used to describe any searched move sequence.

This research has been carried out using an existing computer chess game-playing program called Chessmaps. The Chessmaps Heuristic [3] was created as part of a DPhil research project that was completed in 1998. The intention was to try and add some intelligence into a chess game-playing program. If the goal is to build the best possible chess program, then the experience-based approach has probably solved the problem already, as the best programs are now better or at least equal to the best human players. Computer chess can also be used, however, simply as a domain for testing AI-related algorithms. It is still an interesting platform for trying to mimic the human thought process or add human-like intelligence to a game-playing program. Exactly this argument, along with some other points made in this paper, are also written or thought about in [4, 5]. While chess provides too much variability for the whole game to be defined, it is still a small enough domain to make it possible to accurately evaluate different kinds of search and evaluation processes. It provides complete information about its environment, meaning that the evaluation functions can be reliable, and is not so complex that trying to encapsulate the process in a comprehensive manner would be impossible.

The rest of the paper is structured as follows: Section 2 describes the original Chessmaps Heuristic that can be used to order moves as part of a search process. Section 3 describes the dynamic move-linking mechanism that is the new research of this paper. Section 4 describes some other AI-related chess game-playing programs. Section 5 describes some test results from tests performed using the new linking mechanism. Section 6 describes how the linking mechanism could lead to other types of research, while Section 7 gives some conclusions on the work.

2. Chessmaps Heuristic

The Chessmaps Heuristic [3] uses a neural network as a positional evaluator for a search heuristic. The neural network is taught to recognise what areas of the board a piece should be moved to, based on what areas of the board each side controls. The piece move is actually calculated based on what squares it attacks, or influences, after it has moved, which therefore includes the long range influence of the piece.

The neural network can be trained on saved game scores for any player, including grandmaster games. The chess position can be converted into a board that defines what squares each side controls and this is then used as the input to training the neural network. The move played in that position can be converted into the squares that the move played influences. These are the squares that the moved piece attacks, and these are used as the output that the neural network then tries to recognise. The theory behind this is that there is a definite relation between the squares that a player controls and the squares that the player moves his/her pieces to, or the areas the player then plays to, when formulating a plan. The neural network by itself proved not to be accurate enough, but as it required the control of the squares to be calculated, this allowed several other move types to be recognised. One division would be to split the moves into safe and unsafe. Unsafe moves would lead to loss of material on the square the piece was moved to, while safe moves would not. It was also possible to recognise capture, forced, and forcing moves. Forced moves would be ones where the piece would have to move because it could be captured with a gain of material. Forcing moves were moves that forced the opponent to move a piece because it could then be captured with a gain of material. This resulted in moves being looked at as part of a search process in the following order:

(1) safe capture moves;

(2) safe forced moves;

(3) safe forcing moves;

(4) safe other moves;

(5) unsafe capture moves;

(6) unsafe forced moves;

(7) unsafe forcing moves;

(8) unsafe other moves.

The Chessmaps Heuristic is therefore used as the move-ordering heuristic at each position in the tree search. The neural network was really only used to order the moves in the "other" moves category, although this would still be a large majority of the moves. The research therefore resulted in a heuristic that was knowledge based, but also still lightweight enough to be used as part of a brute-force alpha-beta (α-β) search. Test results showed that it would reduce the search by more than the History Heuristic [1], but because of its additional computational requirements; it would use more time to search for a move and would ultimately be inferior. The heuristic, however, proved difficult to optimize in code, for example, trying to create quick move generators through bitmap boards, and so the only way to reduce the search time would probably be to introduce more AI-related techniques into the program. This has led to the following new suggestion for dynamic move sequences.

3. Dynamic Move Sequences

Two of the most popular experience-based approaches to minimising the search process are the History Heuristic and Transposition Tables [1, 2]. Tests have shown that using combinations of heuristics can produce a search reduction that is close to the minimal tree. This means that it is possible to reduce the size of an exhaustive or full search, close to what a perfect move ordering would produce. Perfect move ordering would select the best move first, second best second, and so on, at every node in the search tree. Transposition Tables, however, replace part of the search process with a result that has already been stored and so in effect forward prune, or remove, parts of the search tree. This means that it would be able, in theory, to produce a search tree size that was less than this minimum value. The History Heuristic is attractive because of its simplicity and also its compact form. It can represent all of the moves in a single 64-square board array. Transposition Tables can become very large in size, therefore requiring an indexing system to search over thousands of entries or more, to find a position that relates to the one currently being searched. The question arises could it not be possible to represent the Transposition Table information in a more compact form, if all of the moves can be represented in a 64-square board array? This might not be possible for Transposition Tables, as they need to index position descriptions themselves, but the dynamic move sequences will use this sort of structure.

The main structure for storing dynamic move sequences is a 64-square board array. During a normal α-β search, when a move sequence is found that causes a cutoff in the search process, it can be stored in this array by storing the first move in the array element, representing the first move's "move from" square. The entry then also stores all of the other move information, such as the "move to" square or the piece type. Multiple entries for different pieces can then be stored individually for the same square. This first move then stores a link to the second move, which stores a link to the next move, and so on. Each move could also have a weight value associated with it that would be incremented or decremented, based on whether the move sequence is subsequently found to be useful or not. The current implementation, however, does not require this, as any stored moves are automatically used again.

There are differences and similarities between the move chains and Transposition Tables. One thing is that these move sequences can be removed as well as added. Another is that Transposition Tables retrieve a move sequence for the "position" being currently searched, whereas the move chains do not consider the position but retrieve a possible move sequence for the "move" currently being looked at. The transformation processes are related, however, where any move played will lead to a position that could be stored in

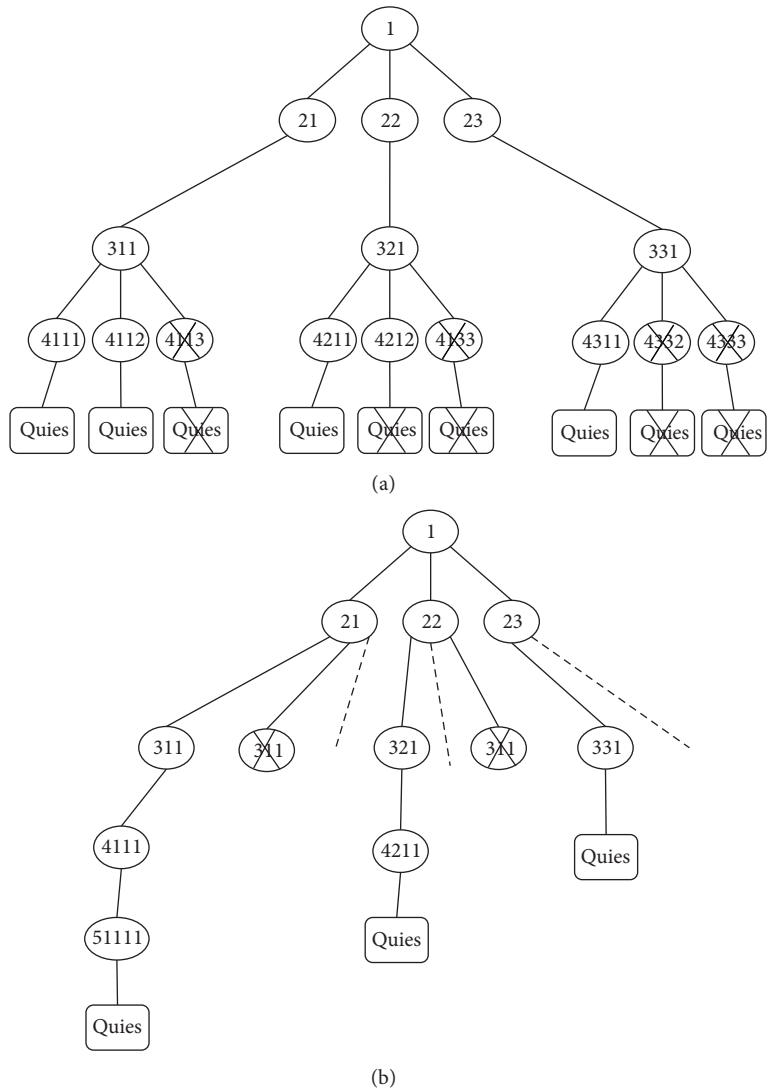

Figure 1: Diagram showing the difference in the number of nodes searched for a standard α-β search (a) or a search with dynamic move sequences (b).

a table, or any position evaluation retrieved from a table will result in a move that could be stored in a move chain. The difference then is that the position evaluation is exact and static, whereas the move path evaluation is dynamic and needs to be checked again before it can be used.

The argument for storing general information in experience-based heuristics is the fact that positions in a search tree vary only slightly from one ply to the next and so if some information is retrieved, it is likely to be useful again, even if it does not exactly match the current position. Dynamic move sequences can also use this argument. The positions that the move chains are used in (over a single search) should be closely related to each other, so if the move sequence is legal, it might be useful in a different position as well. Unreliable sequences then need to be removed when necessary, and this is helped by constantly updating the information on the stored move chains. Figure 1 is a diagram showing the potential difference in the number of nodes searched using a standard α-β search or using dynamic move sequences. Note that this is only illustrative and is not meant to be accurate. Table 1 gives more accurate numbers. The figure is intended to show simply that the standard search is much broader and exhaustive. It will prune more, from the first branch to the last, as the window of acceptable moves narrows. The new algorithm will produce much narrower and deeper searches on every branch. The dashed lines show where possibly forward pruning has taken place. With fewer moves being considered; this is prone to miss critical moves, however.

Figure 1(a) shows a game tree searched to a particular depth, with certain nodes being pruned through the α-β minimax algorithm. These nodes are crossed out with Xs. Considering the nodes in the subtree of the node labeled "21", for example, the diagram shows that there are 5 standard node evaluations and a further two quiescence searches after this. Note that a node must be evaluated before its sub tree can be discarded. The quiescence search is much deeper but

Table 1: Comparison of node count for 98 game positions with the Chessmaps Heuristic, plus either transposition tables or dynamic move sequences.

	Chessmaps	Chessmaps + TT	Chessmaps + DMS
Depth 5	653134.3918	438067.3505	15365.72165
Reduction	0%	33%	97.5%
Depth 6	3263162.34	2467462.031	18920.76289
Reduction	0%	24.5%	99%

considers only forcing or capture moves, until a quiescent or stable position is reached. If Figure 1(b) represents the same tree, but including dynamic move sequences, then the valid move sequences forward prune the tree by replacing large parts of it. Only the paths in the move chains are considered and not the full sub tree with multiple branches. This can occur at any level in the tree. It is also dynamic, where an incorrect evaluation will remove the chain and allow for a full search the next time the move is encountered.

4. Related Work

The brute-force programs are still the most successful. They rely more on program optimization and speed, to search more positions and therefore obtain a more accurate evaluation. There are also, however, a number of strong AI-related programs, some of which are listed here. A summary of recent advances in AI game playing can be found in [6]. KnightCap [7] is possibly one of the best AI attempts at playing computer chess, based on its playing strength. It uses a variation of the Temporal Difference machine learning algorithm called TD Leaf (λ). The core idea of TD learning is that we adjust predictions to match other, more accurate, predictions about the feature. Accuracy can be obtained through deeper searches (more time) or evaluations of models that are closer to the known result. A similar mechanism was used in the NeuroChess program [8]. Blondie25 [9] is another example of a chess game-playing program that uses an evolutionary method to learn to play, as does [5]. These programs try to learn the evaluation function through feedback from playing a number of games, to adjust weight values related to features, to fine tune the evaluation criteria. They are therefore flexible in the sense that the criteria are not hard coded beforehand and are improved through machine learning. In [4] they use a Bayesian network to adapt the program's game-playing strategy to its opponent, as well as to adapt the evaluation function.

As the tree search process is almost optimised, the most important aspect of the computer chess program now is probably the evaluation function. The optimisation results, however, are compared to the brute-force α-β search, but with forward pruning it is possible to produce even smaller search trees. The smaller the search tree, the more information that can be added to the evaluation function, and so trying to minimise the search further is still important. While not directly related to computer chess, the paper [10] describes recent advances in dynamic search processes. Their algorithm called Learning Depth-First Search (LDFS) aims to be both general and effective. It performs iterated depth-first searches enhanced with learning. They claim that LDFS shows that the key idea underlying several existing heuristic-search algorithms over various settings is the use of iterated depth-first searches with memory, for improving the heuristics over certain relevant states until they become optimal. Storing move chains is also a memory structure that is dynamic and built up over time. The dynamic moves approach also probably has similar intentions to the UCT search algorithms [11, 12]. These search to variable depths, select what branches to expand further, and are able to deal with large search trees or uncertainty. The paper [11] describes how TD learning is offline, while UCT is online. It also states

> *At every time-step the UCT algorithm builds a search tree, estimating the value function at each state by Monte-Carlo simulation. After each simulated episode, the values in the tree are updated online and the simulation policy is improved with respect to the new values. These values represent local knowledge that is highly specialised to the current state.*

The move chains mechanism, however, appears to be very different. The process does not require the intelligence that something comparing states might do but is more automatic. In this research that means simply: if they are legal move sequences and evaluated as better through the evaluation function. The bandit algorithms, such as UCT, appear to rely on aggregating statistical values, to build up a more knowledgeable picture of what nodes are good or bad.

5. Testing

5.1. Search to Fixed Depth for Different Algorithm Versions. This set of tests tried three different search algorithms on the same set of positions but to fixed depths of 5 or 6 ply for each position. The search process automatically applies a minimal window and iterative deepening to each search that is performed. As well as this, the three algorithms tested included: either the Chessmaps Heuristic by itself, the Chessmaps Heuristic and Transposition Tables, with 1000000 entries for each side, or the Chessmaps Heuristic with Dynamic Move Sequences. Table 1 gives the results of the test searches.

The first line of numbers is the actual search results, while the second line is the percentage difference between Chessmaps by itself and the other mechanism that it is compared to. The reduction row describes the amount of extra search reduction that either the Transposition Tables or the move chains produce over just the Chessmaps Heuristic. The Transposition Tables that are implemented as part of the Chessmaps program can be considered as a standard variation of the mechanism. They produce a significant reduction in the number of nodes searched, as shown in the table. The move chains then produce an even larger search reduction by radically forward pruning the search tree. Most of the generated move chains were in fact only to a depth of 1 move but could be to depth of 3 or 4 moves. Algorithm 1

```
moveList = generateMoves();

while (moveList not empty) {
        nextMove = getNextMove();

        if (nextMove in moveChains) {
                moveSequence = getMoveSequence(nextMove, moveChains);
                if (isValid(moveSequence)) {
                        linkPath = moveSequence;
                        nextEval = evalSearch(linkPath, then rest of a-b plus quies);
                } else {
                        remove moveSquence from moveChains;
                        linkPath = null;
                }
        } else {
                linkPath = null;
        }

        if (linkPath == null) {
                nextEval = evalSearch(full a-b search + quies);
        }

        if (nextEval > bestEval) {
                bestEval = nextEval;
                bestMove = nextMove;
        }

        if (bestEval > alpha) {
                alpha = bestEval;
                bestPath = serchPath;
                if (linkPath != null) update linkPath in moveChains;
        }
}
always add final bestPath to moveChains;


function update_linkPath {
        if (path exists under move key) {
                replace with linkPath;
        } else {
                add linkPath as new path with first move as key;
        }
}
```

Algorithm 1: Search algorithm with move chains, highlighting where move chains are used.

describes how the move chains are used as part of a search process.

In words this could be:

(1) When playing a move at a node, check for an exact match in the move chains structure. This can include to/from squares, move type, and pieces on the squares.

(2) If a move sequence exists as a link path then:

  (a) retrieve the link path and play the sequence to make sure that it is legal;

  (b) if it is legal then:

    (i) perform a search that is: the link path first, then a standard α-β search plus quiescence. Search depths for full searches can be cut if they exceed the current maximum allowed;

  (c) if it is illegal then:

    (i) remove the path from the move chains structure;

    (ii) perform a standard α-β search to the desired depth plus quiescence.

(3) If the current evaluation is better than the best for this node:

(a) set the node's best value to the current evaluation;
(b) set the node's best path to the current search path;
(c) if the node's best value is better than alpha:
    (i) set alpha to the current best value;
    (ii) if the link path is valid, update the move chains structure to store the path.

(4) Always add any final path to the move chains structure.

This is probably just one variation of what could be a more flexible algorithm and mainly outlines where the move chains are used. This would be integrated as part of an α-β minimax algorithm, where the minimax values determine the cutoffs. In this version, the move chains replace almost exactly where the Transposition Table would be used. Any entry is automatically used if legal, and any new entry automatically overwrites an existing entry, as it is considered to be closer to the final solution. Any entry found not to be legal is automatically removed. The traditional assumption is that this should invalidate the search process, because the search tree can be different, and so critical moves are likely to be missed. The results of playing the two algorithms against each other, as described in Section 5.2, however, show that this is not the case. So can an argument be made for why these shallow move chain searches might work? Suppose that a move chain is stored that is only one move deep. It is only stored if the move was evaluated as better or caused a cutoff, which is only after a stabilizing quiescence search as well. Also, consider the fact that only the first positions in a search process are searched to the full depth. The leaf nodes are only searched to a depth of 1, plus the extensions. So this algorithm is treating more of the nodes like this. Because it is dynamic, the evaluations of the stored moves are constantly changing and being updated, which will help to maintain its accuracy. There might in fact be quite a lot of consistency between the nodes in the search tree, primarily because they originate from the same position and differ only slightly. The previous search results will then compensate for the shallow depth of a move chain plus forward pruning.

5.2. Playing the Different Algorithms against Each Other. The computer program is able to play different algorithms against each other. In these tests, the Chessmaps algorithm with Transposition Tables and a standard brute-force α-β search, was played against the Chessmaps algorithm with the Dynamic Move Sequences search algorithm. Both versions used an iterative deepening search with minimum window as well. There is also a random opening selector, so each game started from a different position. Tests were run over 100 games—50 with Chessmaps plus Transposition Tables as White and 50 with Chessmaps plus Dynamic Move Sequences as White. This test was repeated three times. Each side was allowed 30 minutes to play each game, with the results shown in Table 2.

These results are not meant to be definitive but should show that the new search algorithm is a viable alternative, with the idea of using dynamic move sequences being a sound one. These results show that using the dynamic move sequences with the Chessmaps Heuristic outperforms using Transposition Tables with the Chessmaps Heuristic, with the move chains version winning more games. The Chessmaps program is still only a prototype program and is missing modules or functions that would be included in a complete program. There is no real knowledge of endgame play, for example, where the program will tend to play more aimlessly and be happy to concede a draw by repetition, in a clearly winning position. On the other hand, in certain middlegame positions where its positional evaluator should work well, it does play strongly. It is still a bit buggy and can produce errors, but the testing and results have been performed under reliable circumstances. So a rough evaluation of playing strength might be a strong club player, although, this is not particularly strong for computer chess programs these days.

Table 2: Results of three hundred 30 minute games between transposition tables versus dynamic move sequences.

Draw	MinWin + CM + TT	MinWin + CM + DMS
102	80	118

5.3. Longer Games. In half-hour games, the dynamic linking mechanism was better. In hour-long games, however, the standard brute-force α-β search appears to be improved. While either variation could win, the general standard of play was possibly equal. Unfortunately, it was difficult to quantify this accurately through multiple runs, as a bug could creep in, particularly around king checks and further on in the game. Viewing play before the program became buggy, however, showed an equal strength, with only minor things changing the game state and final result. A limit on the maximum depth allowed for the move chains' α-β search, helped a bit, but did not remove the bug completely. These results also suggest improvements that could be made to the search process, which are described in Section 6.

Because the bug was relatively easy to spot, through watching the play, and it never happened during the half-hour games, this suggests that the half-hour games were played correctly. The variation that used move chains could search to a depth of 20 ply or more, but there is not much benefit in searching this deep. The evaluation function is unlikely to generate a better evaluation, especially from a narrow search. The standard broader search, however, would really benefit from searching one or two ply more. It would therefore be able to make more of the extra time and produce a better evaluation during the hour-long games. So if the move chains statistically won again, the conclusion would be definitive, but as the two variations are instead more equal, the argument is that this actually helps to validate the earlier results. It also suggests that there should be a limit put on the maximum search depth and points out some failings that would occur against stronger searches.

5.4. Other Statistical Results. Some other statistical results were as follows. If considering the move chains, a quiescence

search could search 15 to 40 times more nodes than the α-β search. For the search process, dynamic move chains would be retrieved only during the α-β search and for possibly 1/3 to 1/2 of the nodes searched in that part. This is, however, comparing moves to nodes but means that when a set of moves was evaluated at a node, a chain would be retrieved this number of times. It is estimated that on average, 6 moves are evaluated in a middlegame position, so a move chain would be retrieved possibly 1 in 12 to 1 in 18 times. The move chain would then be used possibly 1/2 to 3/4 of the time that a chain was retrieved. This is quite a high value and suggests some reliability behind the move chain and therefore also the relation between the nodes in the search tree. One other point to note is that the dynamic approach does not appear to use its time as well as the full α-β search and usually has more time left at the end of a game. Even for the half-hour games, the dynamic approach would have maybe 40–50% more time left over at the end. So it would be able to freely add extra evaluations without a time penalty. It would therefore benefit more from broadening its search and not deepening it further.

6. Future Work

The dynamic move sequences are a bit like dynamic chunking, where a player would recognise and play combinations he/she has seen before. In this case, the table stores move sequences instead of positions that can maybe be thought of as tactical chunks. Why compare to a tactical chunk? The aim of these chunks is to suggest relevant moves in a position. Tactics are about moves, not positional evaluation, and so move sequences known to be good could be thought of as chunks of tactical knowledge. Being able to react to individual moves would be useful for games with imperfect information as well.

6.1. Feature Extraction. Early work on knowledge-based approaches to playing computer chess tried to recognise key features in a chess position that would be used to determine what the best move might be. It is argued that stronger players are able to recognise these features, or chunks of knowledge [13], and use them better to analyse a position. One knowledge-based approach to building a computer chess program was to ask an expert to manually define these features in many different positions, but this proved impractical. One future research direction would be to store the positions that each move sequence was played in, with the move sequence itself. The set of positions can then be analysed to try and determine what the key features are. This is similar, but the process is slightly automated, and there is some help in determining what to analyse. The move sequences and the positions that the moves were played in are already defined, and so the problem is now to recognise the key difference or similarities in the related positions and not to redefine these key features from previous knowledge. This is still very challenging, however, and it is not clear how it might be done. Figure 2 describes this sort of process again.

In the figure, there is a move sequence and also the two positions that it was played in. These positions share one common feature shown by the hour-glass object. If this similarity can be automatically recognised, then the process of storing move chains gives an automatic process both for recognizing what the critical move sequences in a game are and also what the critical features that allowed the moves to be played are. One paper that describes using a predefined set of features is [4] where a relatively simplistic feature set contained 840 different variables, while nearly 6000 features are used in [7].

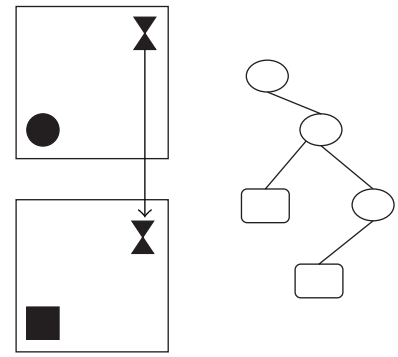

Figure 2: Move chain related to two different positions. The similarity in the positions is shown by the hour-glass feature.

6.2. New Search Process. Section 5.3 showed how the dynamic move sequences algorithm was inferior for larger searches. It also suggested that a broader search was required for it, while the standard α-β search benefited from a deeper one. As the aim is to produce a more intelligent (and human-like) program, the following suggestions can now be made. Iterative deepening is also an often used heuristic, where a search is performed to, for example, 3 ply. It returns a best path, which is then the first path searched at 4 ply, and so on. If the best path is searched first, then the search is reduced by the greatest amount. Dynamic move sequences search deep enough, so to broaden them, why not try another complete search, but one that largely "excludes" the first search path. The justification would be that the narrow search would miss other legitimate lines of play, and a human also performs this type of task. A player would evaluate a move sequence to some depth and determine if it was good or not. The player would then check a completely new or different line, to see if some other plan was in fact better. The dynamic move sequences search could be widened by performing more than one full search but with the rule that they are mostly exclusive. There might have to be some overlap for single moves or smaller move sequences, and so research would need to determine the practical details of that. This process is also appealing in uncertain games, because full searches are returned up to the time limit. This would allow for full evaluations of something, when alternative plans might be missed completely—a more probabilistic and human approach.

The resulting separate but related narrow full searches suggest a number of potential solutions that can then be evaluated further as the start to new searches. The process

could even use a genetic algorithm to take current solutions or searched paths and with evolution combine them into other ones. For this, both negative and positive search paths should be stored, so it would be paths with something critical or important in them. The resulting new paths would also have to be integrated with other full searches but starting from a more limited set of positions. So a different type of iterative process can be performed. The first phase, which is the traditional search using move chains, is almost a knowledge-extraction process that is then reasoned over in the second phase. Note that the positions still need to be stored, to define valid evolutions between existing move sequences, and feature selection would help to define when to crossover or what to mutate. This could still be largely a mechanical process and therefore implementable as part of a computer program.

So this is almost the opposite of what is currently done in more traditional searches, but of course, it uses an α-β variant for its tree searches first. These sorts of processes are also being attempted in general, see for example [10]. This also requires the idea of a "plan", possibly more than other search processes, to intelligently combine solution paths. To help with this, there should be more underlying knowledge, and statistics can also help, by removing bad or poor evolved plans automatically. More intelligence and reasoning appear to be part of new search processes now, in any case.

7. Conclusions

These tests are based on a relatively small number of examples, but they show the potential of the algorithm and confirm that dynamic move sequences are at least reliable. That is, they can be added to a search process and used without returning unreliable moves. Playing against the algorithm also confirms that it is not likely to make a poor move any more often than the other algorithms. So in this respect, move sequences or chains are a viable alternative to Transposition Tables. They also offer a compact solution to indexing the moves and offer new opportunities for research into other more knowledge-based or intelligent ways of playing the game of computer chess. The main danger is probably simply the loss in evaluation quality because of the reduction in the search space. Most of the generated move chains were only to depth 1 but could be to depth 3 or 4, for example. So it is the ability to dynamically update them, the similarity between positions in a search, and the quiescence search that keep the search result accurate.

The test results suggest that the potential of the move chains might be the fact that it can provide a basis for new ways to search a game tree and even under different circumstances. It is shown to be at least equal to the other variants tested, but the different way that it stores information could lead to different search mechanisms that might ultimately prove better in certain types of game. It is not completely clear if this process can be mapped to a human thought process when playing the game. Retrying a combination that a player thought of previously usually requires some positional judgment first. So it might simply be the statistical argument that supports the other experience-based heuristics. The position changes only slightly from one ply to the next and so if a move is found to be good, then there is a reasonable chance that it will be good in a future position as well. The intention of this research has been to reduce the search process so much that the evaluation function can be made more substantial, to make the whole game-playing process more human like. In this respect, the research has certainly achieved its goal. The future work section describes a new search process that requires additional directed intelligence, but this is now being looked at for other search processes as well.

Disclosure

This paper is a revised version of a white paper "Dynamic Move Chains—a Forward Pruning Approach to Tree Search in Computer Chess", originally published on DCS and Scribd, March 2011.

References

[1] J. Schaeffer, "The history heuristic and alpha-beta search enhancements in practice," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 11, no. 11, pp. 1203–1212, 1989.

[2] J. Schaeffer and A. Plaat, "New advances in alpha-beta searching," in *Proceedings of the 25th ACM Computer Science Conference*, pp. 124–130, February 1996.

[3] K. Greer, "Computer chess move-ordering schemes using move influence," *Artificial Intelligence*, vol. 120, no. 2, pp. 235–250, 2000.

[4] A. Fernández and A. Salmerón, "BayesChess: a computer chess program based on Bayesian networks," *Pattern Recognition Letters*, vol. 29, no. 8, pp. 1154–1159, 2008.

[5] A. Iqbal, "What computer chess still has to teach us—the game that will not go," *Electronic Journal of Computer Science and Information Technology*, vol. 2, no. 1, pp. 23–29, 2010.

[6] J. Fürnkranz, "Recent advances in machine learning and game playing," *Oesterreichische Gesellschaft fuer Artificial Intelligence Journal*, vol. 26, no. 2, pp. 19–28, 2007.

[7] J. Baxter, A. Tridgell, and L. Weaver, "KnightCap: a chess program that learns by combining TD(lambda) with gametree search," in *Proceedings of the 15th International Conference on Machine Learning*, pp. 28–36, 1998.

[8] S. Thrun, "Learning to play the game of chess," in *Advances in Neural Information Processing Systems 7*, G. Tesauro, D. Touretzky, and T. Leen, Eds., Morgen Kaufmann, San Fransisco, Calif, USA, 1995.

[9] D. B. Fogel, T. J. Hays, S. L. Hahn, and J. Quon, "A self-learning evolutionary chess program," *Proceedings of the IEEE*, vol. 92, no. 12, pp. 1947–1954, 2004.

[10] B. Bonet and H. Geffner, "Learning depth-first search: a unified approach to heuristic search in deterministic and non-deterministic settings, and its application to MDPs," in *Proceedings of the 16th International Conference on Automated Planning and Scheduling (ICAPS'06)*, D. Long, S. F. Smith, D. Borrajo, and L. Mc-Cluskey, Eds., pp. 142–151, June 2006.

[11] S. Gelly and D. Silver, "Combining online and offline knowledge in UCT," in *Proceedings of the 24th International Conference on Machine Learning (ICML'07)*, pp. 273–280, Corvallis, Ore, USA, June 2007.

[12] Y. Wang and S. Gelly, "Modifications of UCT and sequence-like simulations for Monte-Carlo Go," in *Proceedings of the IEEE Symposium on Computational Intelligence and Games (CIG'07)*, pp. 175–182, Honolulu, Hawaii, USA, April 2007.

[13] J. Fürnkranz, "Machine learning in computer chess: the next generation," *The International Computer-Chess Association Journal*, vol. 19, no. 3, pp. 147–161, 1995.