On Achieving History-based Move Ordering in Adversarial Board Games using Adaptive Data Structures*

Spencer Polk† and B. John Oommen‡

Abstract

This paper concerns the problem of enhancing the well-known alpha-beta search technique for intelligent game playing. It is a well-established principle that the alpha-beta technique benefits greatly, that is to say, achieves more efficient tree pruning, if the moves to be examined are ordered properly. This refers to placing the best moves in such a way that they are searched first. However, if the superior moves were known *a priori*, there would be no need to search at all. Many move ordering heuristics, such as the Killer Moves technique and the History Heuristic, have been developed in an attempt to address this problem. Formerly unrelated to game playing, the field of Adaptive Data Structures (ADSs) is concerned with the optimization of queries over time within a data structure, and provides techniques to achieve this through dynamic reordering of its internal elements, in response to queries. In earlier works, we had proposed the Threat-ADS heuristic for multi-player games, based on the concept of employing efficient ranking mechanisms provided by ADSs in the context of game playing. Based on its previous success, in this work we propose the concept of using an ADS to order moves themselves, rather than opponents. We call this new technique the History-ADS heuristic. We examine the History-ADS heuristic in both two-player and multi-player environments, and investigate its possible refinements. These involve providing a bound on the size of the ADS, based on the hypothesis that it can retain most of its benefits with a smaller list, and examining the possibility of using a different ADS for each level of the tree. We demonstrate conclusively that the History-ADS heuristic can produce drastic improvements in tree pruning in both two-player and multi-player games, and the majority of its benefits remain even when it is limited to a very small list.

Keywords: *Alpha-Beta Search, Adaptive Data Structures, Move Ordering, History Heuristic, Killer Moves*

1 Introduction

The problem of achieving robust game play, in a strategic board game such as Chess or Go, against an intelligent opponent is a canonical one within the field of Artifical Intelligence (AI), and has seen a great deal

*The second author is grateful for the partial support provided by NSERC, the Natural Sciences and Engineering Research Council of Canada. A preliminary version of this paper was presented at ICCCI'15, the *7th International Conference on Computational Collective Intelligence Technologies and Applications*, in Madrid, Spain, in September 2015.

†This author can be contacted at: School of Computer Science, Carleton University, Ottawa, Canada : K1S 5B6. E-mail: andrewpolk@cmail.carleton.ca.

‡Author's status: *Chancellor's Professor*, *Fellow: IEEE* and *Fellow: IAPR*. This author can be contacted at: School of Computer Science, Carleton University, Ottawa, Canada : K1S 5B6. The author is also an Adjunct Professor with University of Agder, Grimstad, Norway. E-mail: oommen@scs.carleton.ca.

of research throughout the history of the field [20, 25]. A substantial portion of the vast body of literature present in this field is based on the highly effective alpha-beta search technique, which provides an efficient way to intelligently search a potentially very large number of moves ahead in a game tree, while pruning large sections of the tree which have been found to be irrelevant [7, 20]. Using this technique, great strides have been made over the years in competitively playing many strategic board games at the level of top human players.

It is well known that the performance of the alpha-beta search is greatly impacted by a proper move ordering. This involves arranging possible moves so that the best move is likely to be searched first. Based on this knowledge, a substantial body of literature exists that spans a wide variety of move ordering heuristics that attempt to achieve this [7, 17, 23]. Examples of these techniques include the well-known Killer Moves strategy, and the History Heuristic, which serve as domain-independent approaches, that operate by remembering those moves that have performed well earlier in the search, and prioritizing them later [23].

The formerly unrelated field of Adaptive Data Structures (ADSs) is concerned with the problem of query optimization within a data structure, based on the knowledge that not all elements are accessed with the same frequency [3, 5, 6]. This problem is addressed through dynamic reorganization of the data structure's internal order, in an attempt to place elements accessed with a higher frequency nearer to the head of the list [1]. This reordering is accomplished in response to queries as they are received, and the field of ADSs proposes a number of possible mechanisms by which a data structure can be reordered in response to the queries, such as the Move-to-Front or Transposition rules for adaptive lists [1, 5, 6].

Observing that there is an intuitive link between the dynamic reordering of elements of an ADS in response to queries, and move ordering strategies in games, we had previously proposed the Threat-ADS heuristic, for multi-player games, which employs an adaptive list to rank *opponents* based on their relative threats [12]. The specific case of multi-player game playing is relatively unexplored in the literature dealing with intelligent game playing, and poses a number of unique challenges, which prevent existing multi-player techniques from achieving a performance comparable to their two-player counterparts [9, 22, 28, 29, 30]. The Threat-ADS was shown to be able to achieve statistically significant gains in terms of tree pruning, in a wide range of configurations, by considering different ADS update mechanisms and starting positions of the game [13, 15].

Based on the success of the Threat-ADS heuristic in the multi-player domain, we hypothesize that ADS-based ranking may be applied in other areas in the context of game playing. Specifically, based on the Killer Moves and History Heuristic techniques, we propose a related move ordering heuristic, which we refer to as the History-ADS heuristic, which uses the qualities of an ADS to rank individual moves, augmenting their position in the ADS when the move is found to produce a cut. In this work, we show that, while using lightweight, efficient ranking techniques associated with an ADS, the History-ADS is able to obtain substantial gains in tree pruning in both the two-player and multi-player cases, in a variety of games.

Preliminary results related to this work were presented in [14] and [16]. The remainder of the paper is laid out as follows. Section 2 presents a background on game playing, and a fairly brief description of the Killer Moves and History Heuristic techniques. Section 3 introduces the field of ADSs, and the techniques from that field that we employ in this work. Section 4 details our previous work, the Threat-ADS heuristic, based on which we describe the novel History-ADS heuristic. Section 6 describes possible refinements to the History-ADS. Section 7 describes our experimental configuration and game models, and Sections 8, 9, 10

and 11 present our results. Section 12 provides our discussion and analysis of these results, and Section 13 concludes the paper.

2 Game Playing Background

Historically, the primary technique for achieving competent game play against an adversarial opponent has been based on the Minimax strategy, which has been successfully applied to the problem of intelligent game playing from the theoretical roots of the discipline, to the modern era [10, 20, 25]. The Minimax strategy, as its name implies, attempts to maximize the perspective player's possible gain, when considering each possible move or action he could take, while assuming that the opponent does the opposite, or attempts to minimize the perspective player's returns.

When applied to a two-player, turn-based board game, such as Chess, the Minimax technique achieves intelligent and informed game play by searching a number of moves ahead in the game tree that represents all possible paths of moves in the game, or to a given depth, usually referred to as a ply [20]. The game tree is explored in a depth-first manner, until the desired ply is reached, at which point the game state is evaluated according to some form of refined heuristic, assigning a value to the position for the max, or perspective, player [20]. These values are then passed upwards through the tree, assuming that in positions where the perspective player is making a move, the maximum of these values will be selected, and the minimum will be chosen when the opponent can make a decision. In normal games, these options generally alternate with each level of the tree. Upon completion of the search, the root of the tree, representing the current turn, is assigned a value. This represents our best estimation, according to our available search depth and heuristic, of the best possible move available to the perspective player. A simple example of a game tree explored according to the Minimax strategy is presented in Figure 1.

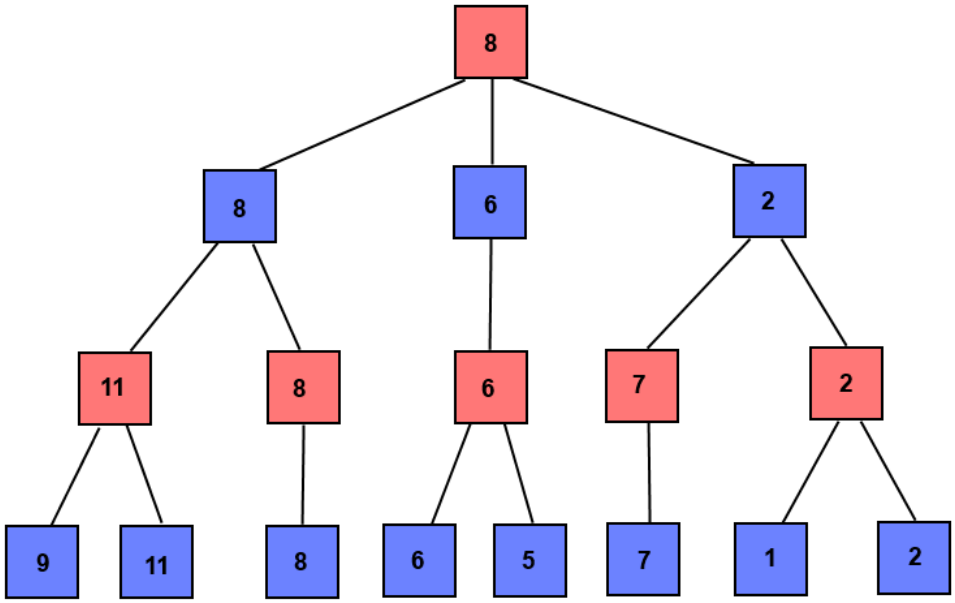

Figure 1: A simple example of a game tree explored according to the Minimax strategy. The red nodes represent the perspective or max player, and blue nodes represent the opponent, or min player.

From the above explanation, one can intuitively see that the strength of the Minimax technique is dependent upon two major factors. The first of these is the strength of the evaluation heuristic employed in leaf positions, as a weak heuristic will necessarily lead to an ill-informed understanding of the game state, while the use of highly refined heuristics employing expert knowledge is a standard method of insuring the strong performance of a game engine [10, 20, 22]. The other factor is the maximum possible ply depth that the engine can search to, since the ability to search deeper represents a higher degree of lookahead in the game. This allows for the formulation of more complex strategies, and takes care of avoiding possible pitfalls or traps set by the opponent. Maximum achievable search can be increased, as expected, through improvements to available hardware, or refinements to the Minimax technique.

Over the course of its history, a number of refinements, modifications, and improvements to the Minimax algorithm have been proposed, including techniques to improve gameplay logic, such as quiescence search, and extensions of the Minimax technique to environments other than those that involve perfect information for two player strategic games, such as multi-player environments, and games of incomplete information [20, 28]. However, a major focal point of improvements to the Minimax technique is in achieving a greater lookahead, via a more efficient search, including arguably the most well-known enhancement, alpha-beta pruning.

The well-known alpha-beta search (which refers to a Minimax search employing alpha-beta pruning) is based upon the observation that not all moves available at different levels of the game tree will impact the its value. Some of them are so poor that they will never be reasonably selected, while others are so strong that the opponent will never allow a situation where they can come to pass [7]. Furthermore, it is possible, through the construction of upper and lower bounds on the possible values at a given node, commonly called the alpha and beta values, to prove that a given node can never impact the value of the tree, and thus, its

children no longer need to be searched [7]. Given that there may be many possible descendants of this pruned node, huge sections of the search tree can be eliminated, greatly increasing efficiency, and the possible ply depth that can be searched with a specific set of resources can be increased. Due to its well-known nature, we will not discuss here the technicalities of the alpha-beta search in any greater detail.

It is well-known that the performance of alpha-beta search can be substantially improved by correct move ordering, that is, by involving methods by which the best possible move is searched first, leading to stricter bounds being constructed, and thus, more efficient pruning [20, 23, 28]. However, without perfect information about the game tree, it is intuitively impossible to know, with certainty, the identity of the superior moves. Thus, a wide range of move ordering heuristics have been proposed over the years. Some of these employ expert knowledge of the game to insure that strategically good moves are examined first. Others are domain independent strategies that apply to a wider range of strategic board games [10, 22, 23, 24]. Two well-known examples of these, upon which the present work is based, are described in detail in the following section.

2.1 The History Heuristic and Killer Moves

The number of techniques available to achieve efficient move ordering in a game playing engine is exhaustive, and to fully detail every one available in the literature would be outside the scope of this work. In this work, we specifically consider two well-known, commonly-used move ordering heuristics, which are the Killer Moves heuristic, and the History heuristic [23]. These techniques are related, in that both attempt to remember effective moves encountered ("effective" being defined as those likely to produce a cut, resulting in a smaller tree), and to explore them first if they are encountered elsewhere in the tree. Indeed, the History heuristic is regarded as a generalization of the Killer Moves heuristic, from a local to a global environment, within the tree [23]. However, both are still commonly employed in modern game engines [10, 22, 23].

The Killer Moves heuristic (also sometimes called the Killer heuristic) operates by prioritizing moves that were found to be good (that is, that produced a cut) in sibling nodes. For example, in the case of Chess, if it was found at some level of the game tree that White moving a bishop from C1 to A3 produced a cut, and that same move is encountered in another branch at the same level of the tree, it will be examined before other moves [23]. The heuristic is based on the assumption that each move does not change the board state that much. Therefore, if a move produced a cut in another position, it is likely to do well elsewhere, even if the preceding moves are different. Of course, this means that the Killer Moves heuristic can potentially be less effective in games where single moves do in fact produce large changes within the game.

The Killer Moves heuristic accomplishes this prioritization by maintaining a table in memory that is indexed by the depth. Within each memory location, a small number of "killer" moves are maintained (usually two), in a linked list or similar data structure [23]. If a new move produces a cut at a level of the tree where the list is full, older moves are replaced according to some arbitrary replacement scheme. When new moves are encountered, the "killer" moves in the table at the current depth are analyzed first, if they are applicable. Note that, as the algorithm can check if the killer moves are available when expanding a new node, and examine them immediately, the Killer Moves heuristic does not require the nodes to first be sorted. In fact additional time can be saved by not even generating the remaining moves if one of the killer moves produces a cut.

The History Heuristic is an attempt to apply the Killer Moves heuristic on a global scale, allowing moves

from other levels in the tree to influence decisions. While a simplistic approach would be to maintain only a single list of "killer" moves and to apply it at all levels of the tree, this would allow moves that produce cuts near the leaves (as there will be many more of them, due to the explosive nature of the game tree), to have a disproportionate effect on the moves within the list [23]. The history heuristic therefore employs a mechanism by which cuts produced higher in the game tree have a greater impact on deciding which move to analyze first.

This is accomplished by maintaining a large array, usually of three dimensions, where the first index is 0, for the maximizing player, and 1, for the minimizing player. The next two indices indicate a move in some way. For example, in the case of Chess, the array is normally indexed by [from][to] where each of [from] and [to] are one of the 64 squares on the board. Within each of these cells is a counter, which is incremented when the corresponding move is found to produce a cut [23]. This counter is incremented by the value $depth*depth$, or 2^{depth}, thereby insuring the value increases more if the cut is higher in the tree [23]. When moves are generated, they are ordered by their value in this array, from greatest to least. In this way, moves that have produced a cut more often, and moves that produced cuts higher in the tree, are examined first.

The History heuristic is noted as a particularly effective and efficient move ordering technique [23]. However, it does have some drawbacks. The array of counters it must store is relatively large, although not a practical concern for modern computers, being two 64-by-64 arrays in the case of Chess. More importantly, unlike the Killer Moves heuristic, moves cannot be generated from the History heuristic; they must be sorted, adding non-linear time at every node in the tree. Thus, it is desirable to look at other heuristics if they are capable of cutting branches of the tree without adding this sorting time. Furthermore, in very deep trees, the History heuristic is known to become less effective, to the point where some modern Chess-playing programs, in particular, either do not use it or limit its application [24].

3 Adaptive Data Structures

It is a well-known problem, in the field of data structures, that the access frequencies of elements within a data structure are not uniform [5, 6]. As an illustrating example, consider a linked list consisting of five elements, A, B, C, D, and E, in that order, where the corresponding access probabilities are 20%, 5%, 10%, 40% and 25%. Using a traditional singly-linked list, these access probabilities pose a problem, as the two elements accessed the most frequently, D and E, are located at the rear of the list, thus requiring a longer access time. We can intuitively see that another linked list, holding the same five elements, in the order D, E, A, C, B will achieve faster average performance. Thus, by restructuring the list, one can obtain an improved functionality for the data structure.

In the trivial example above, the reorganization is obvious, as the access probabilities are assumed to be known and stationary. However, in the real world, the access probabilities are, as one would expect, not known when the structure is first created. The field of ADSs concerns itself with finding good resolutions to this problem [1, 3, 5, 6]. As the access probabilities are not known, the data structure must learn them as queries proceed, and *adapt* to this changing information by altering its internal structure to better serve future queries [5]. Individual types of ADSs provide different methods to achieve this sort of behaviour for the specific data structure. An ADS may be of any type, typically a list or tree, with the well-known Splay Tree being an example of the second type [1, 11]. However, in this work, we will be focusing exclusively on

adaptive lists.

The method by which an ADS reorganizes its internal structure, in response to queries over time, must logically possess several qualities in order to be useful. Specifically, as the goal of an ADS is to improve the amortized runtime, by allowing more frequently accessed elements to be queried faster, the mechanism by which it reorders itself must itself be very efficient, or time lost on its execution would render benefits to query time irrelevant. Thus, methods developed in the fields of ADSs are typically simple, constant-time operations that do not require many memory accesses, comparison, or the use of counters.

In our previous work, we observed that the specific qualities of ADSs enable an ADS to be used as a highly efficient, dynamic *ranking* mechanism for other domains in game playing, provided two requirements can be met. The first is that the objects that we wish to rank can be represented in some way by the elements of the data structure, where the internal structure of the ADS can be seen to reflect their relative ranking. The second is that some method needs to exist to query the ADS when one of the ranked elements should be moved closer to the top position.

Given the wide range of potential objects that can be ranked within the domain of game playing, especially in the context of move ordering, we previously proposed that techniques from ADSs could be applied as an improving agent in the formerly-unrelated domain of game playing. This innovation led to the development of the Threat-ADS heuristic for multi-player games, where an ADS was employed to rank opponents, and this information was used to achieve move ordering in a state-of-the-art, multi-player technique. The Threat-ADS heuristic is described in more detail below.

3.1 ADS Update Mechanisms

The field of ADSs provides a wide range of techniques by which an ADS can reorganize its internal structure. We shall refer to these as *update mechanisms*. As alluded to in the previous section, these techniques tend to be very efficient, with a few constant-time operations, generally consisting of swapping the locations of a small number of elements in the list. In our previous work, we examined a wide range of known, ergodic, ADS update mechanisms, and found that they performed roughly equally well when applied to move ordering [13]. We have thus, to avoid repetition, restricted our analysis in this work to two very well known and frequently contrasted ADS update mechanisms, i.e., the Move-to-Front and Transposition rules.

Move-to-Front: The Move-to-Front update rule is one of the oldest and most well-studied update mechanisms in the field of ADSs [1, 5, 19, 26]. Not coincidentally, it is also one of the most intuitive. As its name suggests, when an element is accessed by a query, in a singly-linked list, it is moved to the head, or front, of the list. Thus, if an element is accessed with a very high frequency, it will tend to stay near the front of the list, and therefore will be less expensive to access. It is also intuitive to see that, for a list where elements have $O(1)$ pointers to the next element, performing the action of moving an element to the front of the list is also $O(1)$, thus making the update mechanism very inexpensive to implement.

Given that elements are always moved to the front of the list when using the Move-to-Front rule, the list changes quite dramatically in response to each query, and this can cause it to generate more expensive queries compared to its competitors in many circumstances [19]. However, unlike its competitors, the Move-to-Front update mechanism provides the valuable property of a lower bound on cost in relation to the optimal ordering. It has been shown that the Move-to-Front update mechanism will provide a system that costs no

more than twice that of the optimal ordering[1, 5]. This guarantee insures the Move-to-Front rule remains attractive, even when compared to competing update mechanisms, which can often outperform it.

Transposition: The most common competitor to the Move-to-Front rule, also studied extensively in the ADS literature, is the Transposition rule [1, 4, 5]. It is no more difficult to implement or understand than the Move-to-Front rule, and, like its chief competitor, offers a powerful performance gain with interesting properties. When an element is accessed, under the Transposition rule, it is *swapped* with the element immediately ahead of it in the list. Thus, as an element is accessed more and more frequently, it will slowly approach the head of the list, contrasted with Move-to-Front, where it is immediately placed there.

As can be deduced from its behaviour, the Transposition rule is less sensitive to change than the Move-to-Front rule, which, depending on the problem domain, can be a good or bad thing [1]. Under many circumstances, the Transposition rule will be much closer to the optimal rule than the Move-to-Front rule over a long period of time [4, 19]. Unfortunately, the Transposition rule does not offer any lower bound on cost in relation to the optimal ordering, and arguments have been made for either it or the Move-to-Front rule in different domains, leading to a historical lack of consensus in the field [1, 4, 26]. It is thus natural, when exploring a new domain of applicability with ADSs, to examine these two contrasting rules.

4 Previous Work: The Threat-ADS Heuristic

Our previous work focused exclusively on the domain of Multi-Player Game Playing (MPGP), which is a variant on traditional two-player game playing, where the number of opponents is greater than unity. Although multi-player games can be thought of as a generalization of the two-player environment to an N-player case, the majority of research has continued to focus on two-player games, with a substantially lesser focus on MPGP being present in the literature [9, 22, 27, 28, 31]. However, through the addition of multiple self-interested agents, such games present a number of complications and challenges that are not present in traditional two-player game playing. These include:

- One player's gain does not necessarily generate an equal loss amongst all opponents.
- Temporary coalitions of players can arise, even in games with only a solitary winner.
- The board state can change more between each of the perspective player's moves.
- A single-valued heuristic is not always sufficient to correctly evaluate the game state.
- Established, highly-efficient tree pruning techniques, such as alpha-beta pruning, are not always applicable.

Despite these challenges, due to the historical success of Mini-Max with alpha-beta pruning, in a wide range of environments, the majority of MPGP strategies have been based on its extensions to a multi-player environment [9, 22, 27, 28]. These include the Paranoid and Max-N algorithms, which operate by assuming a coalition of opponents against the perspective player, and by extending the heuristic to a tuple of values, one for each player, and where one assumes that each agent seeks to maximize his own value [28]. The details of these algorithms are omitted here in the interest of brevity and relevance.

In recent years, a novel MPGP technique, named the Best-Reply Search (BRS), has been proposed, which is capable of achieving substantially stronger performance than either the long-standing Paranoid or Max-N algorithms in a wide variety of environments [22]. Given its state-of-the-art nature, our previously proposed technique, the Threat-ADS heuristic, was designed with it in mind. The BRS, and the Threat-ADS heuristic, are detailed in following sections, as the Threat-ADS forms the basis of the new work that we present here.

4.1 Best-Reply Search

The BRS attempts to simplify the problem of a multi-player game back to a two-player game, to take advantage of the breadth of techniques available in the two-player context, and avoid the extremely large search space the Paranoid and Max-N algorithms must consider [22]. It achieves this by grouping all opponents together, and considering them to be a single,"super-opponent". During each Min phase of the tree, this "super-opponent" is only allowed to make a single move, or, in other words, only one opponent is permitted to act. This opponent is the one who has the most minimizing move, in relation to the perspective player, at this point in time, or the "Best Reply". Figure 2 shows a single level of a BRS tree (only a single level is shown for space considerations, as the branching factor is considerably higher for opponent turns in BRS), where the minimum of all opponent turns is being selected.

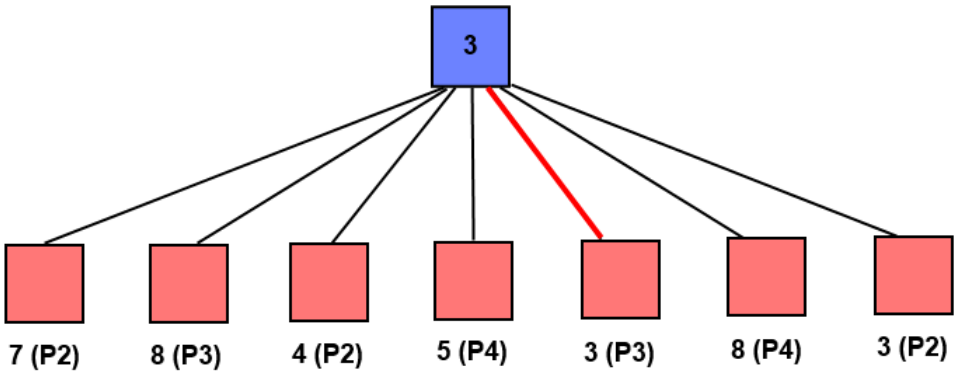

Figure 2: The operation of a single level of the Best-Reply Search. The scores that are reported have the opponent's player number listed next to them (in parenthesis) to assist in the clarification.

The immediate, glaring drawback of the BRS algorithm is that it considers illegal move states while searching. This is certainly a serious drawback, and in fact, limits the games to which BRS can be applied [22]. BRS can only be applied to those games where it is *meaningful* for players to act out of turn, and performs best when the board state does not change too dramatically in between turns [22]. An example of a game to which BRS can *not* by applied is Bridge, because scoring in Bridge is based on tricks, and thus, allowing players to act out of order renders the game tree to be void of meaning. In a game where the game state changes significantly between turns, there is a serious risk of the BRS arriving at a model of the game which is significantly different from reality.

However, in cases where it can be applied, the BRS has many benefits over the Paranoid and Max-N

algorithms, and often outperforms them quite dramatically [22]. By considering the multi-player game as if there were two players, issues related to pruning and potential lookahead for the perspective player are mitigated, which can lead to much better game play in certain games where the game state does not change much during each turn, such as Chinese Checkers, but where many opponents may be present [22].

4.2 The Threat-ADS Heuristic

A factor that is present in multi-player games, but which has no equivalent in the two-player case, is that of relative opponent threat, which we define as a ranking of opponents based on their potential to minimize the perspective player, either based on their current board position, or some knowledge we have gained about their skill. The idea of of considering opponent threat, both to model and predict their future actions, as well as to prioritize opponents, is a known concept within extant multi-player game playing literature [29, 30, 32]. As the BRS groups all opponents together, and must consider all possible moves they could make to find the best one, it presents an opportunity where a ranking of opponents based on their relative threats can be applied to move ordering. This is precisely how the Threat-ADS functions, employing an ADS to enable this ranking to take place.

Considering the execution of the BRS, we observe that, at each "Min" phase, the BRS determines which opponent has the most minimizing move. The phenomenon of having the most minimizing move against the perspective player, and also being characterized by possessing the relatively highest threat level against that player, are conceptually and intuitively linked. With that in mind, we query an ADS, which contains the identities of each opponent, with the identity of the opponent that is seen to possess the most minimizing move during a Min phase. This has the effect of advancing his position in the relative threat ranking, and allowing the ADS to "learn" a complete ranking over time.

With this ranking provided, we employ it by exploring the relevant moves, at each Min phase, in order from the most to least threatening opponent. As a threatening opponent is more likely to provide the greatest minimization to the perspective player, this improves move ordering, and thus the savings from alpha-beta pruning. We clarify this by means of an example in Figure 3. This figure shows how the ADS updates, based on which opponent was found to have the most minimizing move, at a certain level of the tree. Here, opponent "P4" has the most minimizing move, and thus the ADS is updated by moving him to the head of the list.

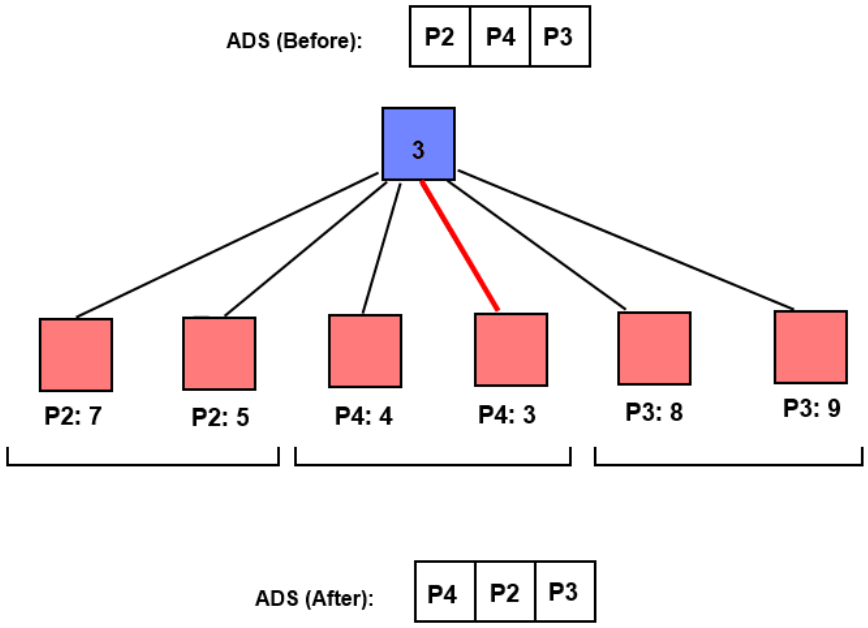

Figure 3: A demonstration of how the Threat-ADS heuristic operates over time.

We observe that the Threat-ADS heuristic is particularly lightweight, requiring only efficient, constant-time updates, and retains a list of a size equal to the number of opponents, which is likely to be a very small constant (typically no more then seven). Furthermore, the Threat-ADS heuristic has the quality of not requiring the moves to be sorted, similar to Killer Moves, as they may simply be generated in the order of the ADS. In our previous work, we demonstrated that the Threat-ADS heuristic was capable of producing meaningful, statistically significant gains in terms of tree pruning in a variety of multi-player games, and employing a range of update mechanisms, at different points in time within the game's progression, and for a variety of opponents [12, 13, 15].

5 The History-ADS Technique

As mentioned earlier, the novel technique we propose in this work is inspired by our previous Threat-ADS heuristic, and by the well-known Killer Moves and History Heuristic move ordering strategies. Specifically, we wish to employ the same metric for achieving move ordering that the established techniques employ, that of move history. Thus, we intend to create a ranking of moves based upon their previous performance within the search, or more specifically, by providing a higher ranking to those that have produced cuts previously. We then prioritize those moves which possess a higher rank when they are encountered later in the search, with the expectation that those that have produced a cut before will be more likely to do so again, leading to improvements in the efficiency of the alpha-beta search technique. As with the Threat-ADS heuristic, we

wish to accomplish this ranking by means of an ADS, given that ADSs provide efficient and dynamic ranking mechanisms.

Rather than utilizing an ADS whose elements are opponents, as in the case of the Threat-ADS, we, instead, employ an ADS containing moves. However, unlike the case with the Threat-ADS, we begin with an empty adaptive list. When a move is found to produce a cut, we query the ADS with the identity of that move. To illustrate this, in the case of Chess, we would query it with the co-ordinates of the square that the piece originated from, and the co-ordinates of its destination, as one does in invoking the History heuristic. If the ADS already contains the move's identity, its position within the ADS is changed according to the ADSs' update mechanisms. If it is *not* within the ADS, it is instead appended to the end, and immediately moved as if it were queried.

An example of how the ADS can manage move history over the course of the game is depicted in Figure 4, which showcases its learning process and application over a fragment of the search.

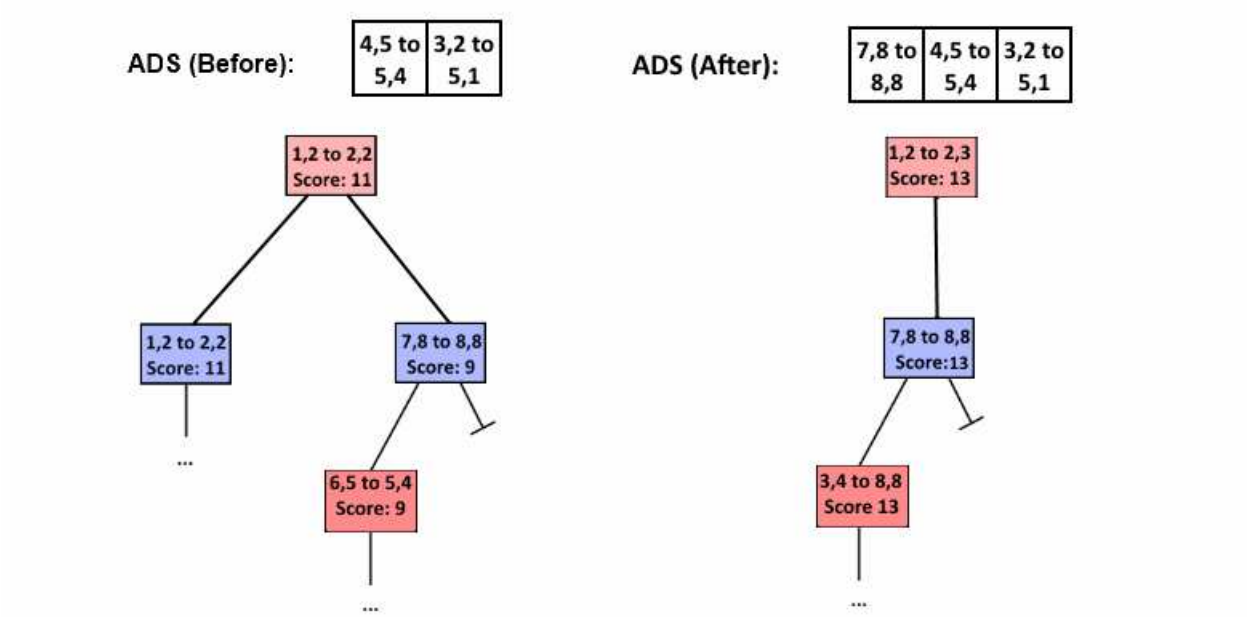

Figure 4: A demonstration of how an ADS can be used to manage move history over time. The move (7,8) to (8,8) produces a cut, and so it is moved to the head of the list, and informs the search later.

5.1 Specification of the History-ADS Heuristic

The precise execution of the History-ADS heuristic is very similar to that of our previously-introduced Threat-ADS heuristic. First of all, we must understand how to update the ADS at an appropriate time, i.e., through querying it with the identity of a move that has performed a cut. This is analogous to querying the ADS with the most threatening opponent within the context of the Threat-ADS. Then, at each Max and Min node, we must somehow order the moves based on the order of the ADS.

Fortunately, within the context of alpha-beta search, there is a very intuitive location to query the ADS, which is where an alpha or beta cutoff occurs, before terminating that branch of the search. This is, of course, analogous to the timing with which the History Heuristic updates its structure. To actually accomplish the move ordering, when we expand a node and gather the available moves, we explore them in the order proposed

by the ADS, again, similarly to how moves are ordered by their value according to the History Heuristic.

The last issue that must be considered is that, unlike in the Threat-ADS where opponent threats were only relevant on Max nodes, when considering move history, the information is relevant on both Max and Min nodes. Furthermore, we observe that a move that produces a cut on a Max node may not be likely to produce a cut on a Min node, and vice versa. This would occur, for example, if the perspective player and the opponent do not have analogous moves, such as in Chess or Checkers, or if they are some distance from each other. We thus employ two list-based ADSs within the History-ADS heuristic, one of which is used on Max nodes, while the other is used on Min nodes. Algorithm 1 shows the Mini-Max algorithm, employing the History-ADS heuristic.

Algorithm 1 Mini-Max with History-ADS

```
Function BRS(node, depth, player)
 1: if node is terminal or depth ≤ 0 then
 2:   return heuristic value of node
 3: else
 4:   if node is max then
 5:     for all child of node in order of MaxADS do
 6:       α = max(α, minimax(child, depth − 1)
 7:       if β ≤ α then
 8:         break (Beta cutoff)
 9:         query MaxADS with cutoff move
10:       end if
11:     end for
12:     return α
13:   else
14:     for all child of node in order of MinADS do
15:       β = min(α, minimax(child, depth − 1)
16:       if β ≤ α then
17:         break (Alpha cutoff)
18:         query MinADS with cutoff move
19:       end if
20:     end for
21:     return β
22:   end if
23: end if
End Function Mini-Max with History-ADS
```

5.2 Qualities of the History-ADS Heuristic

As emphasized earlier, the History-ADS heuristic is very similar in terms of construction to the Threat-ADS heuristic. As with the Threat-ADS, it does not in any way alter the final value of the tree, and thus cannot deteriorate the decision-making capabilities of the Mini-Max or BRS algorithms. Similar to the Threat-ADS, the ADSs are added to the search algorithm's memory footprint, and their update mechanisms with regard to its running time.

Compared to the Threat-ADS, the History-ADS can be expected to employ a much larger data structure, as there will be many more possible moves than total opponents in any non-trivial game. Furthermore, as illustrated above, we maintain two separate ADSs in the case of the History-ADS, which rank minimizing and maximizing moves, respectively. The History-ADS, as described here, thus remembers any move that

produces a cut within its ADS for the entire search, even if it never produces a cut again and lingers near the end of the list. Further, we emphasize that new moves could be regularly added to the corresponding lists. However, while this may appear to suggest that the data structures are of unbound size, depending on how we identify a move, there are, in fact, a limited number of moves that can be made within a game.

Revisiting the example of Chess, if, as in the case of the History heuristic, we consider a move to be identified by the co-ordinates of the square in which the moved piece originated, and the co-ordinates of its destination square, we see that there are a maximum of 4096 possible moves. This figure serves as an upper bound on the size of the ADS. In the case of the History heuristic, an array of 4096 values, or whichever number is appropriate for the game, is maintained for both minimizing and maximizing moves, whereas the History-ADS only maintains information on those moves that have produced a cut. Unarguably, this is a significantly smaller subset, in most cases. We thus conclude that the memory requirements of the History-ADS are upper bounded by the History heuristic.

Perhaps more importantly, unlike the History heuristic, which requires moves to be sorted based on the values in its arrays, the History-ADS shares the advantageous quality of the Threat-ADS in that it does not require sorting. One can simply explore moves, if applicable, in the order specified by the ADS, thus allowing it to share the strengths of the Killer Moves heuristic, while simultaneously maintaining information on all those moves that have produced a cut.

Lastly, unlike the Threat-ADS, which was specific to the BRS, the History-ADS works within the context of the two-player Mini-Max algorithm. However, since the BRS views a multi-player game as a two-player game by virtue of it treating the opponents as a single entity, the History-ADS heuristic is also applicable to it, and it thus functions in both two-player and multi-player contexts. We will thus be investigating its performance in both these avenues in this work.

6 Refinements to the History-ADS Heuristic

As was discussed above, unlike in the case of the Threat-ADS, which, at most, contained only a few elements that represented the number of opponents, the ADS in the present setting could contain hundreds of elements, as many moves could produce a cut over the course of the search. Thus, it is worthwhile to investigate possible refinements or improvements to the History-ADS heuristic that can potentially mitigate this effect. This section describes two of such possible refinements, which are examined in this paper.

6.1 Bounding the Length of the ADS

Retaining all the information pertaining to moves that have produced a cut is logically beneficial. However, it is possible, and in fact very reasonable to hypothesize, that the majority of savings do not come from moves which are near the tail of the list, but rather near the front. Therefore, if we provide a maximum size on the list, and only retain elements in those positions, it may be possible to noticeably curtail the size of the list, providing some guarantees on its memory performance, while maintaining the vast majority of savings provided by the History-ADS. The way in which we will accomplish this is by forgetting any element of the list that falls to position $N + 1$, if the maximum is N. Otherwise, the History-ADS will operate as it was described above. An example of such an ADS updating over several queries, operating with a bounded list,

is presented in Figure 5.

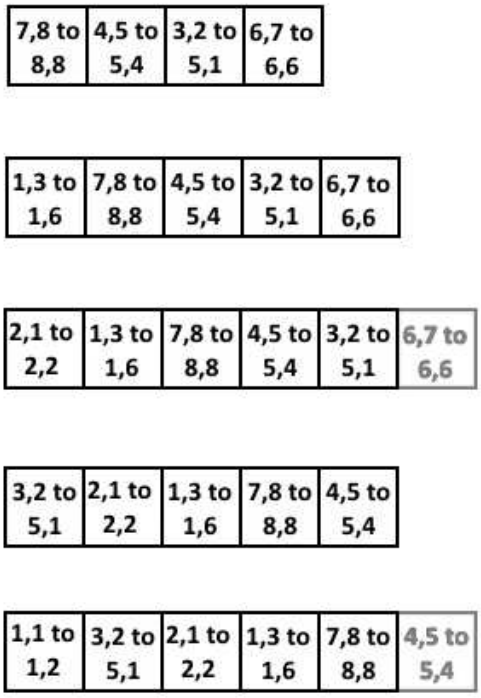

Figure 5: An example of a History-ADS's list sequence updating over several queries, with a maximum length of 5. The ADS starts with a list of length four, and is queried with the move (1,3) to (1,6), which it moves to the front. It is then queried with (2,1) to (2,2), and as (6,7) to (6,6) is pushed to the sixth position, it is forgotten (highlighted in grey). The process continues as it is queried with (3,2) to (5,1), causing only an internal change, and finally (1,1) to (1,2), pushing (4,5) to (5,4) off the end of the list.

Beyond limiting the memory usage of the History-ADS heuristic, if the developer is attempting to avoid sorting moves by generating them in the order of the ADS, having to traverse a very long list to do this could defeat the purpose of omitting sorting. Thus, demonstrating that the History-ADS can retain the majority of its savings with a smaller list can assist in managing implementation concerns, as well.

6.2 Multi-Level ADSs

The History-ADS heuristic as presented earlier maintains a single adaptive list, which is updated whenever a move produces a cut, and is used to order moves when they are encountered elsewhere in the tree. It performs this operation "blindly", without giving consideration to the location in the tree where the move produced a cut, relative to its current location. Thus, if moves are found to produce cuts at the lowest levels of the tree, they will be prioritized at the upper levels of the tree later in the search.

While this may lead to improved savings, as certain moves may be very strong regardless of which level of the tree they occur on, there is a potential weakness in such a blind invocation. Consider the case where a move produces a cut at the highest level of the tree, at node N. It is thus added to the adaptive list, and the search continues deeper into the tree, exploring it in a depth-first manner. Deeper in the tree, many moves are likely to produce cuts, and these will be added to the adaptive list ahead of the first move. When the search returns to the higher levels of the tree, and explores a neighbour of N, these moves will be prioritized

first, over the move that produced a cut at its neighbour. However, intuitively the move that produced a cut at N, which is a more similar game state compared to those deeper in the tree, is likely to be stronger at the current node.

Furthermore, by handling all moves equally, as there are many more nodes towards the bottom of the tree compared to the top, moves that are strong near the bottom of the tree will receive many more updates and thus a higher ranking in the adaptive list. This will occur even though cuts near the top of the tree are comparatively more valuable. Both the Killer Moves and History heuristics employ mechanisms to mitigate these effects [23]. Inspired by this, we augment the History-ADS heuristic with multiple ADSs, one for each level, and use them only within the contexts of their sibling nodes.

The use of multi-level ADSs may lead to a reduction in performance, given that learning cannot be applied at different levels of the tree. But given the precedence set by the existing techniques reported in the literature and the potential benefits, we consider it a meaningful avenue of inquiry.

7 Experimental Verification of the Strategy

As the two heuristics share many conceptual commonalities, it is logical to employ a similar set of experiments in analyzing the performance of the History-ADS heuristic that we used in our previous work on the Threat-ADS heuristic. We are interested in learning the improvement gained from using the History-ADS heuristic, when compared to a search that does not employ it. We accomplish this by taking an aggregate of the Node Count (NC) over several turns (where NC is the number of nodes at which computation takes place, omitting those that were pruned), which we then average over fifty trials. We will repeat this experiment with a variety of games, with the Move-to-Front and Transposition update mechanisms, and at varying ply depths, so to provide us with a clear picture of History-ADS, its benefits and drawbacks, and its overall efficacy. While it would seem intuitive to use the runtime as a metric of performance, it has been observed in the literature that CPU time can be a problematic metric for these sorts of experiments, as it is prone to be influenced by the platform used and by the specific implementation [23].

As with our work involving the Threat-ADS heuristic, we will employ the Virus Game, Focus, and Chinese Checkers when considering the multi-player case. However, as we are also considering the two-player case, we require an expanded testing set of games. While the Virus Game, Focus, and Chinese Checkers can all be played with two players, we have elected to also employ some more well-known two-player games, rather than using the same games in their two-player configurations. This is done so as to provide a wider testing base for the History-ADS. The new two-player games that we will employ are Othello, and the very well-known Checkers, or Draughts. The game models are briefly described in the next section.

Since the History-ADS heuristic may be able to retain its knowledge in subsequent turns, we will allow the game to proceed for several turns from the starting configuration. As the Threat-ADS heuristic does not influence the decisions of the BRS, but only its speed of execution, the end result of the game is not a fundamental concern in our experiments. Thus, we will not run the games to termination after this is done. Specifically, we will run the Virus Game for ten turns, Chinese Checkers for five, and Focus for three, which is consistent with our previously presented work. For Othello and Checkers, we allow the game to proceed for five turns in both cases.

As we did for the Threat-ADS heuristic in [15], rather than simply examine the History-ADS heuristic's

performance near the start of the game, we also examine its performance in intermediate board states. Compared to the initial configuration, intermediate board states represent a more challenging problem, for a number of reasons. These include a greater degree in the variability of intermediate board positions compared to those close to the start of the game, and the lack of "opening book" knowledge, if applicable, allowing intelligent play to more easily be achieved [8].

During turns within which measurement is taking place, other than the perspective player, all opponents made random moves, to cut down on experiment runtime, as we are interested in tree pruning rather than the final state of the game. However, this is clearly not a valid way to generate intermediate starting board positions, as these would be very unrealistic if only a single player was acting rationally. Thus, when considering the intermediate case, we progress the game initially by having each player use a simple 2-ply alpha-beta search or BRS for a set number of turns, after which we switch to the experimental configuration. The number of turns we advanced into the game in this way was fifteen for the Virus Game, ten for Chinese Checkers and Othello, five for Checkers, and, given its short duration, three for Focus.

In order to determine the statistical significance and impact of the History-ADS heuristic's benefits, we employ the non-parametric Mann-Whitney test to determine the statistical significance, as we do not assume that results follow a normal distribution. We also include the Effect Size measure, to illustrate the size of the effect and as a control against the possibility of over-sampling. In general, an Effect Size of 0.2 is small, 0.5 is medium, and 0.8 is large, with anything substantially larger than that representing an enormous, obvious impact [2].

We first present our results for two-player games, using the standard alpha-beta search technique enhanced with the History-ADS heuristic. We then present those results for the multi-player case, where the History-ADS heuristic is employed to augment the BRS. We then present our results for the refinement proposed in the previous section, that of providing a bound on the size of the adaptive list. Finally, we present our results for our second proposed refinement, namely that of providing an ADS for each level of the tree.

7.1 Game Models

In this section, we will briefly detail the games employed in our experiments, to give the reader a better understanding of their rules and game flow, and to contrast them against each other.

Checkers: Checkers, also known as draughts, is a very well-known board game, designed to be played by two players on an 8x8 checkerboard. During each player's turn, he may move any of his pieces one square diagonally left or right and forward, towards the opponent's side of the board. If the player's piece is adjacent to an opponent's piece, and the square directly across from it is unoccupied, the player may "jump" the opponent's piece, and capture it. If other subsequent jumps are possible, the player can continue to make them, chaining jumps together. If a piece reaches the opposite side of the board, it is promoted, and is no longer restricted to moving only forward. A player loses when his last piece is captured, or when he can make no legal moves. Under normal rules, the game of Checkers requires jumps, if available, to be taken. While this makes the game strategically interesting, it greatly decreases the variability of the game tree's size, and thus we have relaxed this rule, to generate greater search trees. We refer to our variant as Relaxed Checkers. The starting position for Checkers is shown in Figure 6.

Othello: Othello is a two-player board game also played on an 8x8 board, and based on the capture

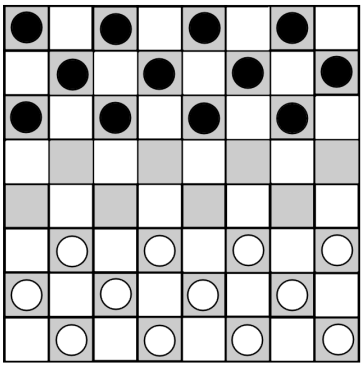

Figure 6: The starting position for Checkers.

of opponent pieces, although the mechanisms by which capturing takes place are quite different. Initially, each player has two pieces on the board, arranged as in Figure 7, During his turn, a player may place an additional piece on the board, in a position that "flanks" one or more opponent pieces in a line between the placed piece and another that the player controls. If the player cannot do this, his turn is passed. This captures the enclosed pieces, and they are flipped, or replaced by pieces of the color of the capturing player. Play continues until neither player can make a valid move, at which point the player who controls the most pieces is declared the winner.

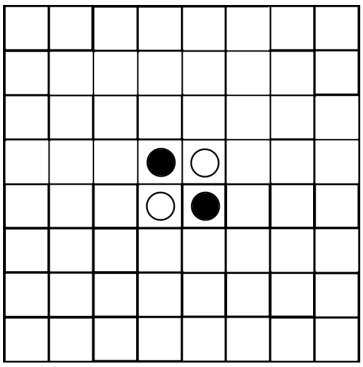

Figure 7: The starting position for Othello.

Focus: Focus is a board game based on piece capturing, designed to be played by two to four players, on an 8x8 board, with the three squares in each corner omitted. The game was originally developed by Sackson in 1969 and released since under many different names [21]. Unlike most other games of its type, Focus allows pieces to be "stacked" on top of each other. The player whose piece is on top of the stack is said to control it, and during his turn, may move one stack he controls, vertically or horizontally, by a number of squares equal to the height of the stack. When a stack is placed on top of another one, they are merged, and all pieces more than five from the top of the stack are removed from the board. If a player captures his own piece, he may place it back on the board in any position, rather than moving a stack. Starting positions for Focus are pre-determined to insure a fair board state. The starting positions for Focus are shown in Figure 8.

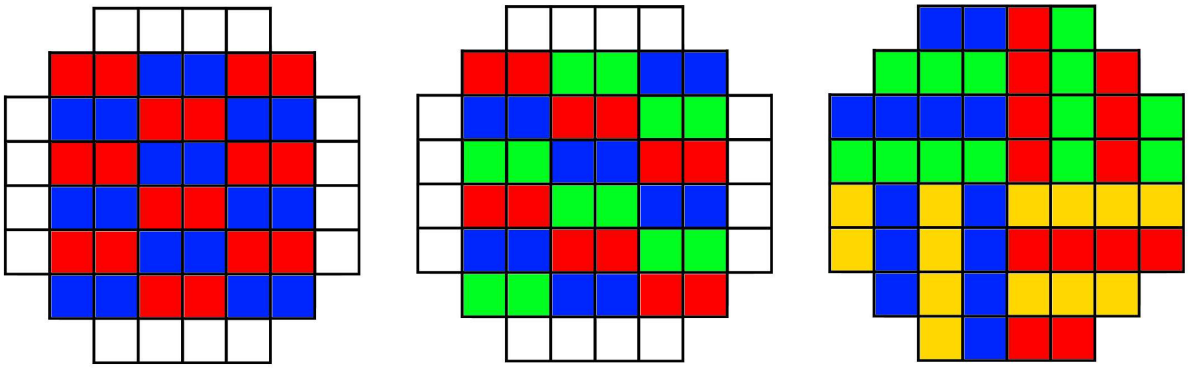

Figure 8: The two, three, and four player starting positions for Focus.

Virus Game: The Virus Game is a multi-player board game of our own creation, modeled after similar experimental games from previous works, based on a biological metaphor [18]. The Virus Game is, in essence, a "territory control" game where players vie for control of squares on a game board of configurable size (in this work, we use a 5x5 board). During his turn, a player may "infect" a square adjacent to one he controls, at which point he claims that square, and each square adjacent to it. The Virus Game is designed primarily as a highly-configurable testing environment, rather than a tactically interesting game, as it is easy for players to cancel each others' moves; however, it shares many elements in common with more complex games. A possible starting position, and an intermediate state of the Virus Game, are shown in Figure 9.

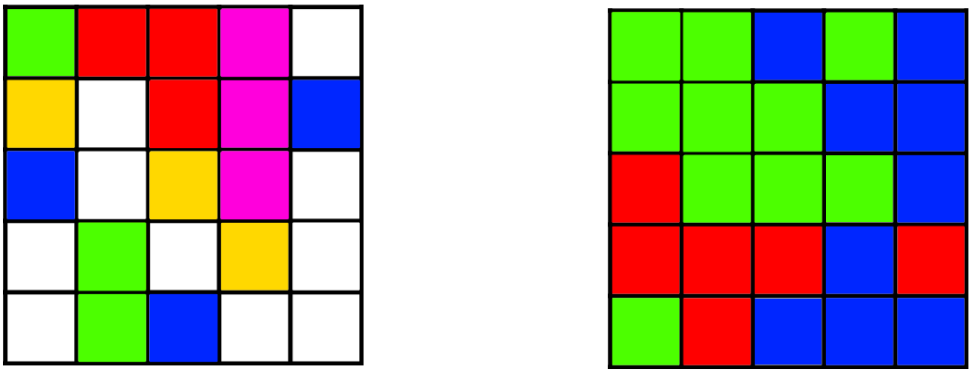

Figure 9: The Virus Game at its initial state, and ten turns into the game. Observe that two players have been eliminated, and the pieces are more closely grouped together.

Chinese Checkers: Chinese Checkers is a well-known multi-player board game, played by between two and six players, omitting five players, as it would give one an unfair advantage. The game is played on a star-shaped board, and the objective is to move all of one's pieces to the opposite corner from one's starting position. On his turn, a player may move one of his pieces to one of the adjacent six positions, or "jump" an adjacent piece, which could be either his or his opponent's. As in Checkers, jumps may be chained together as many times as possible, allowing substantial distances to be covered in a single move. The possible starting positions for Chinese Checkers are shown in Figure 10.

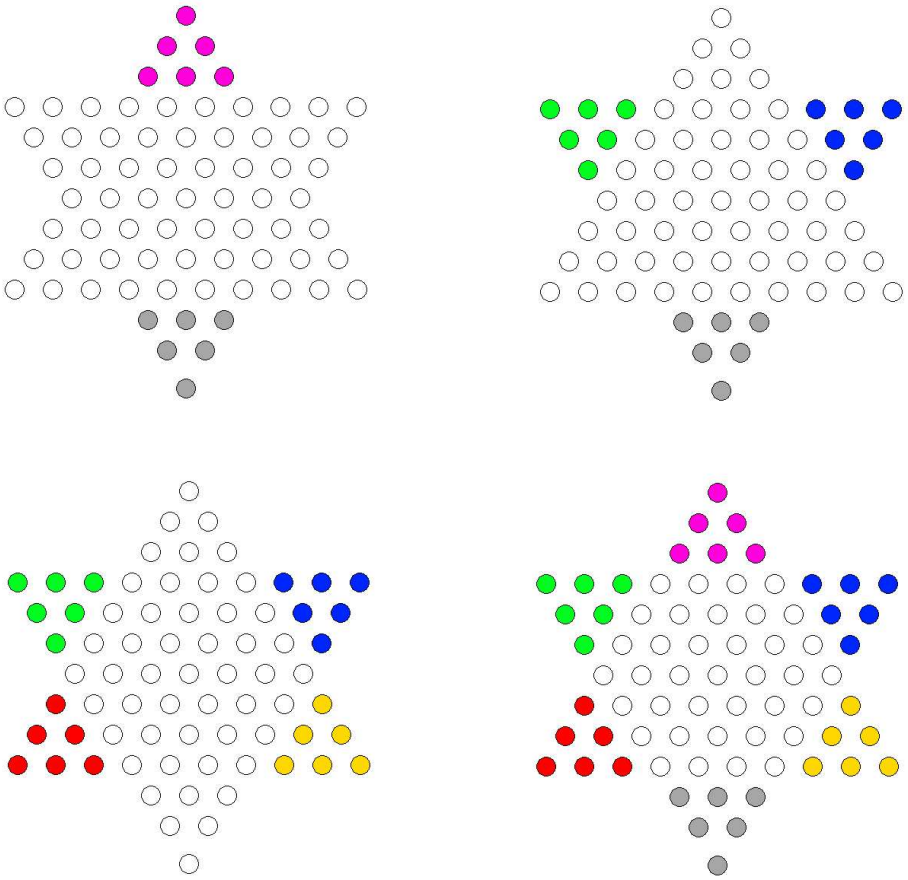

Figure 10: The two, three, four, and six player starting positions for Chinese Checkers.

8 Results for Two-Player Games

Table 1 presents our results for the two-player game Othello. We observe that in all cases, the History-ADS heuristic produced very strong improvements in terms of NC, compared to standard alpha-beta search. Furthermore, in each case, the Move-to-Front rule outperformed the Transposition rule. A higher proportion of savings generally correlates with a larger game tree, both in terms of a greater ply depth, and considering the more expansive intermediate case. Our best performance was in the 8-ply intermediate case, with savings of 47%. We observed an Effect Size ranging between 0.5 and 0.75, indicating a moderate to large effect [2].

Table 2 presents our results for Relaxed Checkers. We notice a very similar trend, compared to Othello, with the History-ADS heuristic generating substantial improvements to pruning in all cases, and generally doing better the larger the search space is, with Move-to-Front always outperforming Transposition. The best performance was again observed in the 8-ply intermediate case, with a 63% reduction in tree size, well over half the tree. In this case, the Effect Size ranged between 0.5 and, in cases with less variance, reached levels well over 2 or even 3, suggesting an extreme effect.

Results for our final two-player game, Focus, are shown in Table 3. Yet again, the History-ADS heuristic always produced substantial gains in terms of tree pruning, with larger savings, the Move-to-Front rule

Table 1: Results of applying the History-ADS heuristic for Othello in various configurations.

Ply Depth	Midgame	Update Mechanism	Avg. Node Count	Std. Dev	P-Value	Effect Size
4	No	None	669	205	-	-
4	No	Move-to-Front	523	162	2.2×10^{-4}	0.71
4	No	Transposition	572	225	0.016	0.47
6	No	None	5061	2385	-	-
6	No	Move-to-Front	3827	1692	6.0×10^{-3}	0.51
6	No	Transposition	4057	2096	0.015	0.42
8	No	None	38,800	20,300	-	-
8	No	Move-to-Front	26,800	12,000	1.2×10^{-3}	0.59
8	No	Transposition	29,700	12,300	0.014	0.45
4	Yes	None	2199	745	-	-
4	Yes	Move-to-Front	1699	633	2.2×10^{-3}	0.67
4	Yes	Transposition	1761	597	0.01	0.59
6	Yes	None	20,100	9899	-	-
6	Yes	Move-to-Front	14,500	6303	6.0×10^{-3}	0.57
6	Yes	Transposition	15,200	6751	0.015	0.49
8	Yes	None	182,000	114,000	-	-
8	Yes	Move-to-Front	95,600	50,200	$< 1.0 \times 10^{-5}$	0.76
8	Yes	Transposition	113,000	60,900	1.3×10^{-4}	0.60

always outperformed the Transposition rule, and it did best in larger trees. Our best performance was in the midgame case, with a 77% reduction in tree size. The Effect Size was over 2 in the more variable intermediate board case, and exceeded 10 in case of an initial board position, again indicating an extreme effect.

9 Results for Multi-Player Games

Our results for the multi-player Virus Game are presented in Table 4. We observe very similar behaviour, in comparison to the two-player games. Again, the Move-to-Front rule always outperforms the Transposition rule, and the History-ADS produces substantial gains in all cases, tending towards greater savings in larger trees. The best result was a 55% reduction in NC, in the 6-ply initial board position case.

Table 5 holds our results for the multi-player variant of Focus. The trends observed are almost identical to the two-player case, and match expectations from patterns recognized there, with a slightly higher maximum of a 78% reduction in tree size in the intermediate case, given that tree sizes are larger in midgame searches.

Lastly, Table 6 presents our results for Chinese Checkers, our final multi-player game. Chinese Checkers deviated slightly from established patterns, as performance was very uniform, although the History-ADS heuristic produced large savings in every case, and Move-to-Front continues to outperform Transposition, if slightly in some situations. Our best results were observed in the four player, initial board position case, with a 65% reduction in tree size, which is, in fact, the smallest search.

Table 2: Results of applying the History-ADS heuristic to Relaxed Checkers in various configurations.

Ply Depth	Midgame	Update Mechanism	Avg. Node Count	Std. Dev	P-Value	Effect Size
4	No	None	5930	864	-	-
4	No	Move-to-Front	4712	461	$< 1.0 \times 10^{-5}$	1.41
4	No	Transposition	5148	675	$< 1.0 \times 10^{-5}$	0.90
6	No	None	78,600	10,600	-	-
6	No	Move-to-Front	40,800	5619	$< 1.0 \times 10^{-5}$	3.58
6	No	Transposition	48,600	6553	$< 1.0 \times 10^{-5}$	2.84
8	No	None	910,000	172,000	-	-
8	No	Move-to-Front	362,000	55,900	$< 1.0 \times 10^{-5}$	3.18
8	No	Transposition	435,000	68,400	$< 1.0 \times 10^{-5}$	2.76
4	Yes	None	5447	1859	-	-
4	Yes	Move-to-Front	3772	1257	$< 1.0 \times 10^{-5}$	0.90
4	Yes	Transposition	4497	1474	4.5×10^{-3}	0.51
6	Yes	None	64,000	25,700	-	-
6	Yes	Move-to-Front	36,100	12,700	$< 1.0 \times 10^{-5}$	1.08
6	Yes	Transposition	43,600	16,600	$< 1.0 \times 10^{-5}$	0.79
8	Yes	None	859,000	408,000	-	-
8	Yes	Move-to-Front	317,000	135,000	$< 1.0 \times 10^{-5}$	1.33
8	Yes	Transposition	422,000	161,000	$< 1.0 \times 10^{-5}$	1.07

Table 3: Results of applying the History-ADS heuristic to two-player Focus in initial and midgame states.

Ply Depth	Midgame	Update Mechanism	Avg. Node Count	Std. Dev	P-Value	Effect Size
4	No	None	5,250,000	381,000	-	-
4	No	Move-to-Front	1,290,000	88,000	$< 1.0 \times 10^{-5}$	10.39
4	No	Transposition	1,800,000	158,000	$< 1.0 \times 10^{-5}$	9.07
4	Yes	None	10,600,000	3,460,000	-	-
4	Yes	Move-to-Front	2,420,000	637,000	$< 1.0 \times 10^{-5}$	2.37
4	Yes	Transposition	2,910,000	760,000	$< 1.0 \times 10^{-5}$	2.22

The next section presents our results for the possible refinements to the History-ADS heuristic, described in Section 6. Given that, in the case of the History-ADS heuristic, the Move-to-Front rule outperformed the Transposition rule in every single case, sometimes by a large margin, we restrict our update mechanism to it going forward, given its demonstrated superiority.

Table 4: Results of applying the History-ADS heuristic for the Virus Game in various configurations.

Ply Depth	Midgame	Update Mechanism	Avg. Node Count	Std. Dev	P-Value	Effect Size
4	No	None	254,000	28,600	-	-
4	No	Move-to-Front	157,000	17,900	$< 1.0 \times 10^{-5}$	3.37
4	No	Transposition	165,000	22,800	$< 1.0 \times 10^{-5}$	3.11
6	No	None	10,500,000	1,260,000	-	-
6	No	Move-to-Front	4,690,000	1,010,000	$< 1.0 \times 10^{-5}$	4.57
6	No	Transposition	4,850,000	739,000	$< 1.0 \times 10^{-5}$	4.45
4	Yes	None	309,000	40,700	-	-
4	Yes	Move-to-Front	188,000	17,800	$< 1.0 \times 10^{-5}$	2.97
4	Yes	Transposition	199,000	20,700	$< 1.0 \times 10^{-5}$	2.69
6	Yes	None	12,800,000	1,950,000	-	-
6	Yes	Move-to-Front	5,940,000	832,000	$< 1.0 \times 10^{-5}$	3.51
6	Yes	Transposition	6,060,000	974,000	$< 1.0 \times 10^{-5}$	3.45

Table 5: Results of applying the History-ADS heuristic for multi-player Focus in various configurations.

Ply Depth	Midgame	Update Mechanism	Avg. Node Count	Std. Dev	P-Value	Effect Size
4	No	None	6,970,000	981,000	-	-
4	No	Move-to-Front	2,180,000	184,000	$< 1.0 \times 10^{-5}$	4.88
4	No	Transposition	2,740,000	271,000	$< 1.0 \times 10^{-5}$	4.32
4	Yes	None	14,200,000	8,400,000	-	-
4	Yes	Move-to-Front	3,240,000	1,730,000	$< 1.0 \times 10^{-5}$	1.30
4	Yes	Transposition	3,570,000	1,860,000	$< 1.0 \times 10^{-5}$	1.26

10 Results for Bounded ADSs

Table 7 presents our results for Othello, with a bounded ADS, in the same configurations as earlier. As is to be expected, due to the History-ADS heuristic being restricted from retaining as much information, some decay in performance was observed, however in even the worst case, the majority of savings were maintained even if the size of the ADS was limited by 5. In a very encouraging scenario, the decrease was by only 1% (from 26% to 25%), when the ply depth was 6 and the size of the list was bounded by 20, from the initial board position.

Table 8 presents our results for Checkers. A similar pattern was observed as in the case of Othello, where the smaller the length of the list, the less the improvement gleaned from the History-ADS heuristic, although the vast majority of savings remained. In the very best case, the reduction in savings was only 2%, when the size of the list was bounded by 20, from the initial board state (regardless of ply depth).

Consider Table 9, which presents our results for Focus with varying limits on the ADS' size. We observe the

Table 6: Results of applying the History-ADS heuristic Chinese Checkers in various configurations.

Ply (Players)	Midgame	Update Mechanism	Avg. Node Count	Std. Dev	P-Value	Effect Size
4 (4-play)	No	None	1,380,000	417,000	-	-
4 (4-play)	No	Move-to-Front	486,000	135,000	$< 1.0 \times 10^{-5}$	2.14
4 (4-play)	No	Transposition	505,000	131,000	$< 1.0 \times 10^{-5}$	2.09
4 (6-play)	No	None	3,370,000	1,100,000	-	-
4 (6-play)	No	Move-to-Front	1,250,000	316,000	$< 1.0 \times 10^{-5}$	1.92
4 (6-play)	No	Transposition	1,320,000	338,000	$< 1.0 \times 10^{-5}$	1.87
4 (4-play)	Yes	None	3,340,000	933,000	-	-
4 (4-play)	Yes	Move-to-Front	1,310,000	365,000	$< 1.0 \times 10^{-5}$	2.17
4 (4-play)	Yes	Transposition	1,360,000	314,000	$< 1.0 \times 10^{-5}$	2.12
4 (6-play)	Yes	None	8,260,000	2,640,000	-	-
4 (6-play)	Yes	Move-to-Front	3,400,000	899,000	$< 1.0 \times 10^{-5}$	1.84
4 (6-play)	Yes	Transposition	3,650,000	920,000	$< 1.0 \times 10^{-5}$	1.74

same pattern as we did in Othello and Checkers, in both the two-player and the multi-player cases, although the difference in savings between an unbounded list and a list of length 20 or 5 is quite a bit smaller, with even the worst case being 78% to 75% in the 4-ply midgame case, which was only a 3% difference.

Table 10 contains our results for the Virus Game. Our observations were analogous to those for the other three cases presented so far. While the History-ADS heuristic produced noticeable gains in all cases, they were lessened by a limit being placed on the maximum length of the ADS. In the very worst case, the change observed was a reduction in savings from 56% to 51% in the 6-ply case, with measurements taken from the initial board state, a very small change of only 5%.

Finally, Table 11 presents our results for Chinese Checkers. The patterns we observed were similar to those for the other four games, however in the case of four-player Chinese Checkers, from the initial board position, a maximum list size of 5 outperformed a list size of 20. As before, for this game, variability was quite small, such as 62% to 61% in the 4-ply, six player case, with a limit on the list of size of 20. Considering even the worst situation, the largest change observed was 66% to 62%, in the 4-ply, four player case, with measurements taken from the initial board position.

11 Results for Multi-Level ADSs

Table 12 presents our results for multi-level ADSs, in the domain of Othello. We observed that, similar to the limit on the ADS, the use of a multi-level ADS reduces performance by a consistent but small amount. Limiting the multi-level ADS's size further reduces the performance, in the majority of cases. In the best case, savings were reduced from 39% to 36%, when a multi-level ADS with no size limitation was employed, in the 8-ply case, from the initial board position.

Our results for Relaxed Checkers are shown in Table 8. The patterns observed were similar to those for Othello, with the use of multi-level ADSs performing slightly worse than the original version. The smallest

Table 7: Results of applying the History-ADS heuristic to Othello with a varying maximum length on the ADS.

Ply Depth	Midgame	Limit	Avg. Node Count	Std. Dev	P-Value	Effect Size
4	No	No ADS	669	205	-	-
4	No	Unbound	525	175	3.2×10^{-4}	0.70
4	No	20	572	196	5.8×10^{-3}	0.47
4	No	5	596	196	0.041	0.36
6	No	No ADS	5061	2385	-	-
6	No	Unbound	3727	1552	1.7×10^{-3}	0.56
6	No	20	3779	1884	3.3×10^{-3}	0.54
6	No	5	3961	1603	0.013	0.46
8	No	No ADS	38,800	20,300	-	-
8	No	Unbound	23,600	10,800	$< 1.0 \times 10^{-5}$	0.75
8	No	20	24,900	11,000	1.1×10^{-5}	0.68
8	No	5	28,600	12,200	5.8×10^{-3}	0.50
4	Yes	No ADS	2199	745	-	-
4	Yes	Unbound	1702	570	2.7×10^{-4}	0.67
4	Yes	20	1787	663	2.6×10^{-3}	0.55
4	Yes	5	1840	576	0.010	0.48
6	Yes	No ADS	20,100	9899	-	-
6	Yes	Unbound	13,300	6916	7.0×10^{-5}	0.69
6	Yes	20	13,900	6846	6.9×10^{-4}	0.63
6	Yes	5	14,800	7916	1.7×10^{-3}	0.54
8	Yes	No ADS	182,000	114,000	-	-
8	Yes	Unbound	92,900	49,700	$< 1.0 \times 10^{-5}$	0.79
8	Yes	20	109,000	63,800	7.0×10^{-5}	0.65
8	Yes	5	116,000	59,800	3.1×10^{-5}	0.58

change was from 46% to 40%, in the 6-ply case, where measurements were taken from a midgame board position.

Lastly, we present our results for the Virus Game with multi-level ADSs in Table 14. While we observed, as before, that the performance worsened when the multi-level approach was employed, the difference was quite a bit smaller in the context of the Virus Game. The loss in savings was very slight in the best case, from 56% to 55%, from the initial board position.

12 Discussion

The results presented in the previous sections strongly reinforce the hypothesis that an ADS managing move history, employed by the History-ADS heuristic, can achieve improvements in tree pruning through better

Table 8: Results of applying the History-ADS heuristic to Relaxed Checkers with a varying maximum length on the ADS.

Ply Depth	Midgame	Limit	Avg. Node Count	Std. Dev	P-Value	Effect Size
4	No	No ADS	5930	864	-	-
4	No	Unbound	4638	493	$> 1.0 \times 10^{-5}$	1.49
4	No	20	4755	406	$> 1.0 \times 10^{-5}$	1.36
4	No	5	4776	516	$> 1.0 \times 10^{-5}$	1.33
6	No	No ADS	78,600	10,600	-	-
6	No	Unbound	41,000	5588	$> 1.0 \times 10^{-5}$	3.55
6	No	20	42,700	5292	$> 1.0 \times 10^{-5}$	3.39
6	No	5	44,700	5545	$> 1.0 \times 10^{-5}$	3.20
8	No	No ADS	910,000	172,000	-	-
8	No	Unbound	358,000	53,200	$< 1.0 \times 10^{-5}$	3.20
8	No	20	375,000	55,700	$< 1.0 \times 10^{-5}$	3.11
8	No	5	398,000	78,300	$< 1.0 \times 10^{-5}$	2.97
4	Yes	No ADS	5447	1859	-	-
4	Yes	Unbound	3681	1138	$> 1.0 \times 10^{-5}$	0.94
4	Yes	20	4130	1304	1.9×10^{-4}	0.70
4	Yes	5	4217	1363	4.3×10^{-4}	0.66
6	Yes	No ADS	64,000	25,700	-	-
6	Yes	Unbound	34,400	12,400	$> 1.0 \times 10^{-5}$	1.15
6	Yes	20	36,700	14,200	$> 1.0 \times 10^{-5}$	1.06
6	Yes	5	39,500	16,800	$> 1.0 \times 10^{-5}$	0.96
8	Yes	No ADS	859,000	408,000	-	-
8	Yes	Unbound	293,000	100,000	$> 1.0 \times 10^{-5}$	1.39
8	Yes	20	333,000	133,000	$> 1.0 \times 10^{-5}$	1.29
8	Yes	5	397,000	193,000	$> 1.0 \times 10^{-5}$	1.13

move ordering, in both two-player and multi-player games. This confirms that such an ADS does, indeed, correctly prioritize the most effective moves, based on their previous performance elsewhere in the tree, as initially hypothesized.

Our results confirm that the History-ADS heuristic is able to achieve a statistically significant reduction in NC in three two-player games, Othello, Relaxed Checkers, and the two-player variant of Focus, as well as three multi-player games, the Virus Game, Chinese Checkers, and the multi-player variant of Focus. Compared to our previous work on the Threat-ADS heuristic, where the degree of reduction ranged between 5% and 20%, and which centered around 10%, we observe a much more drastic improvement in pruning when the History-ADS is employed, to a value as high as 78%. The History-ADS heuristic displayed quite variable performance, with higher savings generally correlated with the size of the tree. In the case of ply depth, this is intuitively appealing, as cuts made earlier higher in the tree can lead to large numbers of moves being

Table 9: Results of applying the History-ADS heuristic to Focus with a varying maximum length on the ADS.

Ply Depth	Midgame	Limit	Avg. Node Count	Std. Dev	P-Value	Effect Size
4	No	No ADS	5,250,000	381,000	-	-
4	No	Unbound	1,260,000	90,900	$> 1.0 \times 10^{-5}$	10.46
4	No	20	1,270,000	96,300	$> 1.0 \times 10^{-5}$	10.43
4	No	5	1,330,000	108,000	$> 1.0 \times 10^{-5}$	10.28
4 (Multi)	No	No ADS	6,970,000	981,000	-	-
4 (Multi)	No	Unbound	2,150,000	165,000	$> 1.0 \times 10^{-5}$	4.92
4 (Multi)	No	20	2,210,000	165,000	$> 1.0 \times 10^{-5}$	4.86
4 (Multi)	No	5	2,230,000	154,000	$> 1.0 \times 10^{-5}$	4.79
4	Yes	No ADS	10,600,000	3,460,000	-	-
4	Yes	Unbound	2,390,000	631,000	$> 1.0 \times 10^{-5}$	2.37
4	Yes	20	2,520,000	710,000	$> 1.0 \times 10^{-5}$	2.34
4	Yes	5	2,630,000	847,000	$> 1.0 \times 10^{-5}$	2.30
4 (Multi)	Yes	No ADS	14,200,000	8,400,000	-	-
4 (Multi)	Yes	Unbound	3,120,000	1,700,000	$> 1.0 \times 10^{-5}$	1.31
4 (Multi)	Yes	20	3,310,000	1,700,000	$> 1.0 \times 10^{-5}$	1.30
4 (Multi)	Yes	5	3,370,000	1,410,000	$> 1.0 \times 10^{-5}$	1.29

pruned, and if branching factor is higher, as in Focus, many moves are unlikely to be particularly strong, and so correct move ordering could, very reasonably, have a high impact on performance.

Our second observation is that in all cases, the Move-to-Front rule outperformed the Transposition rule. This is a reasonable outcome, as unlike with the Threat-ADS, the adaptive list may contain dozens to hundreds of elements, and it would take quite a bit of time for the Transposition rule to migrate a particularly strong move to the head of the list. As opposed to this, the Move-to-Front would migrate the move to the front quickly, and would likely keep it there. This phenomenon also confirms that the order of elements within the list matters to the move ordering. In other words, merely maintaining an unsorted collection of moves that have produced a cut will not perform as well as employing an adaptive list, further supporting the use of the History-ADS heuristic.

The exception to the above trend occurs in the case of Chinese Checkers. While Chinese Checkers, with its large branching factor, usually sees a larger reduction in tree size compared to Othello or Relaxed Checkers, savings for all the cases within the context of Chinese Checkers are roughly equivalent. Furthermore, despite having the smallest overall tree size, we observe that the best performance occurs for the the four player case, with measurements being taken from the initial board position. What this suggests is that moves that produce a cut in Chinese Checkers may not have a very strong natural ranking between them, and so attempting to rank them in the ADS does not help as much as it does for the other games. This would also explain why the Move-to-Front rule did not outperform the Transposition rule to the same extent as in the other cases.

Table 10: Results of applying the History-ADS heuristic to the Virus Game with a varying maximum length on the ADS.

Ply Depth	Midgame	Limit	Avg. Node Count	Std. Dev	P-Value	Effect Size
4	No	No ADS	254,000	28,600	-	-
4	No	Unbound	154,000	20,800	$> 1.0 \times 10^{-5}$	3.47
4	No	20	158,000	19,000	$> 1.0 \times 10^{-5}$	3.36
4	No	5	163,000	19,000	$> 1.0 \times 10^{-5}$	3.20
6	No	No ADS	10,500,000	1,260,000	-	-
6	No	Unbound	4,650,000	767,000	$> 1.0 \times 10^{-5}$	4.60
6	No	20	4,830,000	640,000	$> 1.0 \times 10^{-5}$	4.46
6	No	5	5,110,000	658,000	$> 1.0 \times 10^{-5}$	4.24
4	Yes	No ADS	308,000	40,700	-	-
4	Yes	Unbound	187,000	23,100	$> 1.0 \times 10^{-5}$	3.00
4	Yes	20	187,000	22,200	$> 1.0 \times 10^{-5}$	3.00
4	Yes	5	194,000	21,600	$> 1.0 \times 10^{-5}$	2.81
6	Yes	No ADS	12,800,000	1,950,000	-	-
6	Yes	Unbound	5,870,000	863,000	$> 1.0 \times 10^{-5}$	3.55
6	Yes	20	5,910,000	821,000	$> 1.0 \times 10^{-5}$	3.53
6	Yes	5	6,390,000	767,000	$> 1.0 \times 10^{-5}$	3.29

We found that when limiting the maximum size of the ADS, while there was some reduction in performance, as was expected, the loss was very slight in most cases. This confirms our hypothesis that elements near the head of the list tend to remain there, and provide the majority of the move ordering benefits, with diminishing returns as the list gets longer. The fact that the majority of savings are still maintained in all cases even when the list is limited to only contain five elements, successfully addresses one of the concerns we had with of the History-ADS heuristic discussed earlier, namely that the length of the adaptive list can potentially be quite large.

The multi-level ADS approach was found to do worse than the single ADS approach, suggesting that savings that may be gained for prioritizing moves that produced a cut on the current level of the tree, if they exist, are offset by the inability to apply what the algorithm has learned to other levels of the tree. We can thus conclude that the absolutist approach of having a separate ADS at each level of the tree is likely not the optimal way to address concerns of overvaluing moves that are strong near the bottom of the tree. However, the multi-level ADS approach may not be completely useless. If the History-ADS heuristic is employed alongside other, perhaps domain-specific move ordering heuristics, then prioritizing only the best moves at each level may be as effective. This approach begs for more investigation.

Despite the presence of multiple ADSs, limiting the size of the list to 5 reduced improvements even more. Observing the internal functioning of the search, the reason for this appears to be that the limit is only a factor at the lowest levels of the tree, where many more moves produce cuts and the limit impacts the ADS heavily. As opposed to this, levels closer to the root do not require as much space. This is especially visible

Table 11: Results of applying the History-ADS heuristic to Chinese Checkers with a varying maximum length on the ADS.

Ply Depth	Midgame	Limit	Avg. Node Count	Std. Dev	P-Value	Effect Size
4 (4-play)	No	No ADS	1,380,000	417,000	-	-
4 (4-play)	No	Unbound	476,000	125,000	$> 1.0 \times 10^{-5}$	2.16
4 (4-play)	No	20	531,000	125,000	$> 1.0 \times 10^{-5}$	2.03
4 (4-play)	No	5	525,000	157,000	$> 1.0 \times 10^{-5}$	2.04
4 (6-play)	No	No ADS	3,370,000	1,100,000	-	-
4 (6-play)	No	Unbound	1,280,000	368,000	$> 1.0 \times 10^{-5}$	1.90
4 (6-play)	No	20	1,330,000	332,000	$> 1.0 \times 10^{-5}$	1.85
4 (6-play)	No	5	1,360,000	297,000	$> 1.0 \times 10^{-5}$	1.83
4 (4-play)	Yes	No ADS	3,340,000	933,000	-	-
4 (4-play)	Yes	Unbound	1,240,000	335,000	$> 1.0 \times 10^{-5}$	2.25
4 (4-play)	Yes	20	1,330,000	330,000	$> 1.0 \times 10^{-5}$	2.16
4 (4-play)	Yes	5	1,370,000	455,000	$> 1.0 \times 10^{-5}$	2.11
4 (6-play)	Yes	No ADS	8,260,000	1,950,000	-	-
4 (6-play)	Yes	Unbound	3,200,000	863,000	$> 1.0 \times 10^{-5}$	1.92
4 (6-play)	Yes	20	3,290,000	821,000	$> 1.0 \times 10^{-5}$	1.88
4 (6-play)	Yes	5	3,340,000	767,000	$> 1.0 \times 10^{-5}$	1.86

in the case of 8-ply Othello from the midgame case, where a multi-level ADS with a maximum size of 5 did not, in fact, produce a statistically significant improvement in tree pruning, which is the first time that such an event has occurred for the History-ADS heuristic. Overall, however, the multi-level ADS' performance was close to the original version in the vast majority of cases.

13 Conclusions, Contributions and Future Work

In this work, we introduced the concept of ordering moves, in the alpha-beta search algorithm, based on an ADS, which is dynamically reorganized according to its update mechanism that ranks moves based on which moves have performed well earlier in the search. We have named this technique the History-ADS heuristic. Our results demonstrate conclusively that the History-ADS heuristic is able to obtain a substantial reduction in tree size, in a range of two-player and multi-player games. Specifically, its efficacy has been proven in Othello, Checkers, Focus, Chinese Checkers, and the Virus Game of our own invention, which represents a wide variety of games with different rules and strategies. The results strongly support the hypothesis that the History-ADS heuristic can perform well in the context of two-player and multi-player games.

Despite the power of the History-ADS heuristic, we observed that it retains a particularly large list of moves. We conjecture that the majority of those moves, especially near the tail of the list, may be pruned to save space and assist in implementation, without much loss of performance. Our results strongly support this hypothesis, and we found that in many games, even in the worst case, only a small proportion of savings was

Table 12: Results of applying the History-ADS heuristic to Othello with multi-level ADSs.

Ply Depth	Midgame	Limit	Avg. Node Count	Std. Dev	P-Value	Effect Size
6	No	No ADS	5061	2385	-	-
6	No	No	3727	1552	1.7×10^{-3}	0.56
6	No	Yes	4110	1828	0.023	0.40
6	No	Yes (5-limit)	4305	1843	0.089	0.37
8	No	No ADS	38,800	20,300	-	-
8	No	No	23,600	10,800	$< 1.0 \times 10^{-5}$	0.75
8	No	Yes	24,900	11,900	1.0×10^{-4}	0.68
8	No	Yes (5-limit)	25,600	13,100	2.9×10^{-4}	0.65
6	Yes	No ADS	20,100	9899	-	-
6	Yes	No	13,300	6916	7.0×10^{-5}	0.69
6	Yes	Yes	14,700	7279	1.8×10^{-3}	0.55
6	Yes	Yes (5-limit)	16,000	7283	0.017	0.42
8	Yes	No ADS	182,000	114,000	-	-
8	Yes	No	92,900	49,700	$< 1.0 \times 10^{-5}$	0.79
8	Yes	Yes	110,000	56,500	1.0×10^{-4}	0.64
8	Yes	Yes (5-limit)	105,000	50,600	$< 1.0 \times 10^{-5}$	0.68

lost when restricting the formerly unlimited list to a size of only five, thus overcoming one of the History-ADS heuristic's main drawbacks.

While the use of an ADS at each level of the tree performed similarly to the single ADS, it was outperformed by the single ADS in all cases. However, given the strong basis in the literature of techniques that consider moves based on the level of the tree where they are found, such as with the Killer Moves and History heuristics, we believe that it is worthwhile to investigate this area further. In the hope of striking a more reasonable balance between the two extremes, work is currently ongoing on methods to prioritize learning at the level of the tree where it was acquired, while not completely excluding information obtained elsewhere.

The History-ADS heuristic serves as a natural expansion upon our previous work, introducing the Threat-ADS heuristic, and demonstrates that the use of ADSs in the context of game playing has applicability outside of the window of ranking opponents based on their threats. This work provides an even stronger basis for further examination of the applicability of ADS-based techniques to game playing, and we hope it will inspire others to examine these possibilities in the future.

References

[1] S. Albers and J. Westbrook. Self-organizing data structures. In *Online Algorithms*, pages 13–51, 1998.

[2] R. Coe. It's the effect size, stupid: What effect size is and why it is important. In *Annual Conference of the British Educational Research Association*, University of Exeter, Exeter, Devon, 2002.

Table 13: Results of applying the History-ADS heuristic to Relaxed Checkers with multi-level ADSs.

Ply Depth	Midgame	Limit	Avg. Node Count	Std. Dev	P-Value	Effect Size
6	No	No ADS	78,600	10,600	-	-
6	No	No	41,000	5588	$> 1.0 \times 10^{-5}$	3.55
6	No	Yes	45,300	5977	$> 1.0 \times 10^{-5}$	3.14
6	No	Yes (5-limit)	46,900	6727	$> 1.0 \times 10^{-5}$	3.00
8	No	No ADS	910,000	172,000	-	-
8	No	No	358,000	53,200	$< 1.0 \times 10^{-5}$	3.20
8	No	Yes	394,000	53,500	$< 1.0 \times 10^{-5}$	2.99
8	No	Yes (5-limit)	416,000	68,300	$< 1.0 \times 10^{-5}$	2.86
6	Yes	No ADS	64,000	25,700	-	-
6	Yes	No	34,400	12,400	$> 1.0 \times 10^{-5}$	1.15
6	Yes	Yes	38,100	15,400	$> 1.0 \times 10^{-5}$	1.06
6	Yes	Yes (5-limit)	37,800	13,000	$> 1.0 \times 10^{-5}$	1.02
8	Yes	No ADS	859,000	408,000	-	-
8	Yes	No	293,000	100,000	$> 1.0 \times 10^{-5}$	1.39
8	Yes	Yes	350,000	163,000	$> 1.0 \times 10^{-5}$	1.25
8	Yes	Yes (5-limit)	404,000	150,000	$> 1.0 \times 10^{-5}$	1.11

Table 14: Results of applying the History-ADS heuristic to Focus with multi-level ADSs.

Ply Depth	Midgame	Limit	Avg. Node Count	Std. Dev	P-Value	Effect Size
6	No	No ADS	10,500,000	1,260,000	-	-
6	No	No	4,650,000	767,000	$> 1.0 \times 10^{-5}$	4.60
6	No	Yes	4,740,000	598,000	$> 1.0 \times 10^{-5}$	4.53
6	No	Yes (5-limit)	4,800,000	817,000	$> 1.0 \times 10^{-5}$	4.49
6	Yes	No ADS	12,800,000	1,950,000	-	-
6	Yes	No	5,870,000	863,000	$> 1.0 \times 10^{-5}$	3.55
6	Yes	Yes	5,990,000	653,000	$> 1.0 \times 10^{-5}$	3.49
6	Yes	Yes (5-limit)	6,160,000	616,000	$> 1.0 \times 10^{-5}$	3.40

[3] T. H. Corman, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to Algorithms*, pages 302–320. MIT Press, Upper Saddle River, NJ, USA, 3rd edition, 2009.

[4] V. Estivill-Castro. Move-To-End is best for double-linked lists. In *Proceedings of the Fourth International Conference on Computing and Information*, pages 84–87, 1992.

[5] G. H. Gonnet, J. I. Munro, and H. Suwanda. Towards self-organizing linear search. In *Proceedings of FOCS'79, the 1979 Annual Symposium on Foundations of Computer Science*, pages 169–171, 1979.

[6] J. H. Hester and D. S. Hirschberg. Self-organizing linear search. *ACM Computing Surveys*, 17:285–311, 1985.

[7] D. E. Knuth and R. W. Moore. An analysis of alpha-beta pruning. *Artificial Intelligence*, 6:293–326, 1975.

[8] M. Levene and J. Bar-Ilan. Comparing typical opening move choices made by humans and chess engines. *Computing Research Repository*, 2006.

[9] C. Luckhardt and K. Irani. An algorithmic solution of n-person games. In *Proceedings of the AAAI'86*, pages 158–162, 1986.

[10] A. Papadoupoulus. Exploring optimization strategies in board game abalone for alpha-beta search. In *In Proceedings of CIG'2012, the 2012 IEEE Conference on Computational Intelligence and Games*, pages 63–70, 2012.

[11] S. Pettie. Splay trees, davenport-schinzel sequences, and the deque conjecture. In *Proceedings of the nineteenth annual ACM-SIAM symposium on Discrete algorithms*, 2008.

[12] S. Polk and B. J. Oommen. On applying adaptive data structures to multi-player game playing. In *In Proceedings of AI'2013, the Thirty-Third SGAI Conference on Artificial Intelligence*, pages 125–138, 2013.

[13] S. Polk and B. J. Oommen. On enhancing recent multi-player game playing strategies using a spectrum of adaptive data structures. In *In Proceedings of TAAI'2013, the 2013 Conference on Technologies and Applications of Artificial Intelligence*, 2013.

[14] S. Polk and B. J. Oommen. Enhancing history-based move ordering in game playing using adaptive data structures. In *In Proceedings of ICCCI'2015, the 7th International Conference on Computational Collective Intelligence Technologies and Applications*, pages 201–211, 2015.

[15] S. Polk and B. J. Oommen. Novel AI strategies for multi-player games at intermediate board states. In *Proceedings of IEA/AIE'2015, the Twenty-Eighth International Conference on Industrial, Engineering, and Other Applications of Applied Intelligent Systems*, pages 33–42, 2015.

[16] S. Polk and B. J. Oommen. Space and depth-related enhancements of the history-ads strategy in game playing. In *In Proceedings of CIG'2015, the 2015 IEEE Conference on Computational Intelligence and Games*, 2015.

[17] A. Reinefeld and T. A. Marsland. Enhanced iterative-deepening search. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 16:701–710, 1994.

[18] P. Rendell. A universal Turing machine in Conway's Game of Life. In *Proceedings of HPCS'11, the 2011 International Conference on High Performance Computing and Simulation*, pages 764–772, 2011.

[19] R. L. Rivest. On self-organizing sequential search heuristics. In *Proceedings of the 1974 IEEE Symposium on Switching and Automata Theory*, pages 63–67, 1974.

[20] S. J. Russell and P. Norvig. *Artificial Intelligence: A Modern Approach*, pages 161–201. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 3rd edition, 2009.

[21] S. Sacksin. *A Gamut of Games*. Random House, 1969.

[22] M. P. D. Schadd and M. H. M. Winands. Best Reply Search for multiplayer games. *IEEE Transactions on Computational Intelligence and AI in Games*, 3:57–66, 2011.

[23] J Schaeffer. The history heuristic and alpha-beta search enhancements in practice. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 11:1203–1212, 1989.

[24] E. Schrder. Move ordering in rebel. Discussion of move ordering techniques used in REBEL, a powerful chess engine, 2007.

[25] C. E. Shannon. Programming a computer for playing Chess. *Philosophical Magazine*, 41:256–275, 1950.

[26] D. D. Sleator and R. E. Tarjan. Amortized efficiency of list update and paging rules. *Communications of the ACM*, 28:202–208, 1985.

[27] N. Sturtevant. A comparison of algorithms for multi-player games. In *Proceedings of the Third International Conference on Computers and Games*, pages 108–122, 2002.

[28] N. Sturtevant. *Multi-Player Games: Algorithms and Approaches*. PhD thesis, University of California, 2003.

[29] N. Sturtevant and M. Bowling. Robust game play against unknown opponents. In *Proceedings of AAMAS'06, the 2006 International Joint Conference on Autonomous Agents and Multiagent Systems*, pages 713–719, 2006.

[30] N. Sturtevant, M. Zinkevich, and M. Bowling. Prob-Maxn: Playing n-player games with opponent models. In *Proceedings of AAAI'06, the 2006 National Conference on Artificial Intelligence*, pages 1057–1063, 2006.

[31] I. Szita, C. Guillame, and P. Spronck. Monte-carlo tree search in settlers of catan. In *Proceedings of ACG'09, the 2009 Conference on Advances in Computer Games*, pages 21–32, 2009.

[32] I. Zuckerman, A. Felner, and S. Kraus. Mixing search strategies for multi-player games. In *Proceedings of IJCAI'09, the Twenty-first International Joint Conferences on Artificial Intelligence*, pages 646–651, 2009.