Are There Practical Alternatives To Alpha-Beta in Computer Chess?

Andreas Junghanns*

January 29, 1998

Abstract

The success of the alpha-beta algorithm in game-playing has shown its value for problem solving in artificial intelligence, especially in the domain of two-person zero-sum games with perfect-information. However, there exist different algorithms for game-tree search. This paper describes and assesses those proposed alternatives according to how they try to overcome the limitations of alpha-beta. We conclude that for computer chess no practical alternative exists, but many promising ideas have good potential to change that in the future.

1 Introduction and Motivation

Conventional search methods, such as A* or alpha-beta, are powerful artificial intelligence (AI) techniques. They are appealing because of their algorithmic simplicity and clear separation of search and knowledge. Describing the basic alpha-beta algorithm takes only a few lines of code, and all the domain-dependent knowledge is encoded in a few functions called by a generic search engine. Additionally, the depth-first manner of exploring the search tree imposes only linear with the depth of the tree - an appealing property.

However, several (perceived) problems with the text-book description (see [KM75] for alpha-beta) of the conventional methods led researchers to explore better ways to traverse search spaces. Focusing on two-player search domains, particularly chess, this paper surveys these proposals made during the past decades. Special attention is paid to the practicability of these approaches and their impact on high-performance programs.

This paper is not intended to be an evaluation of the ideas and proposals made, neither empirically, nor theoretically. In the authors opinion, neither is possible in a meaningful way. Empirical evaluation would need fully tuned and enhanced versions of these algorithms and even solving this startling task, the results would be domain specific. The often suggested comparison on artificial game trees would produce equally artificial results of little importance to implementations for real-world domains, since each of the underlying search spaces exhibits very special properties. A theoretical comparison could give insight into the general complexities of the algorithms. However, they are of little meaning since conveniently ignored constants might be the key issue to success for practical implementations.

What is left? A critical review of many promising ideas as they relate to alpha-beta. An exact description of each algorithm is beyond the scope of the paper and the interested reader is referred to the original paper(s). The author hopes that this survey will help to sustain, maybe even revitalize, interest in those promising ideas that have, due to their immaturity, difficulty to compete with the domain-monopolizing, many-fold enhanced and fine-tuned alpha-beta algorithm.

1.1 Problems of Alpha-Beta

The simplicity and elegance of alpha-beta has a price; certain assumptions and simplifications are made that may cause disadvantageous properties. Every two-person zero-sum perfect-information game has a game-theoretic value, the result of the game if both players play perfectly. By traversing the entire game tree using

*University of Alberta, Department of Computing Science, 615 GSB, Edmonton, AB, CANADA T6G 2H1, Email: andreas@cs.ualberta.ca

alpha-beta and using the game results at the leaf nodes (win, loss and possibly draw), this game-theoretic value can be determined.

The alpha-beta algorithm reflects the assumption of a complete game tree traversal: perfect evaluation of the leaf nodes and a simple minimax propagation of those values back up the tree. Because of time constraints however, alpha-beta faces a depth limit when used in practice, voiding the key assumption of a complete game tree traversal. This also results in the loss of the game-theoretic meaning of the propagated values, since heuristic functions are used to estimate the game theoretic value of a node at the fringe of the search tree. This raises the following, related issues:

Heuristic Error: The search engine is not aware of the possible error contained in the heuristic evaluation of problem states. It assumes those values to be perfect, but if they were, search would not be necessary.

Scalar Value: All the domain-dependent knowledge is compressed into one scalar value. This necessarily loses information that is potentially useful for the search.

Expand Next: Alpha-beta expands nodes in a depth-first manner that depends only on the order the successors of a node are generated. The depth-first manner of exploring the tree, although restricting the space requirements, limits the choice of which node to expand next[1].

Bad Lines: Alpha-beta search, in its effort to *prove* the minimax value of the root node, expands at least a solution tree [KM75] of a fixed depth, thereby pursuing relevant lines as well as irrelevant lines to the same depth. One would like to expand lines to lesser depth that are irrelevant or "bad" and instead invest the effort into searching relevant or "interesting" lines deeper.

Insurance: By proving the minimax value of a search tree to a certain depth, alpha-beta runs no risk of missing anything that is visible within this search depth. Every successor is searched to the same depth - insurance that alpha-beta does not miss anything to that depth. Selective algorithms distribute their effort among different parts of the tree, concentrating effort in "relevant" lines. Falsely judging a line as "irrelevant" can prevent the search from correctly assessing the situation [2]. It seems that insurance is a strength of alpha-beta.

Value Backup: Conventional minimax backup rules (as used by alpha-beta) take only the maximum (minimum) of all the values of the successors into account. This is correct if game-theoretic values are propagated. However, since those values may be erroneous (see Heuristic Error), lines that lead to several good alternatives are preferable over lines having only one. Simply calculating minima and maxima does not reflect this reasoning.

Stopping: If the task of the search is to suggest one of many possible successors, it is sufficient to determine the best. Even considering alpha-beta enhanced with iterative deepening selects such a "best" successor with every iteration. It has no measure of confidence for that selection that could be used to decide when to stop the search. For standard alpha-beta the question is simply how deep to search.

Opponent: Minimax-like algorithms assume that both players of the game use the same heuristics. Since the program's heuristics are not perfect, there might be better ones used by the opponent in which case we are doomed to make mistakes, which the opponent can exploit to win. In the other case (the opponent uses inferior heuristics), to assume the opponent is playing perfectly will cause us to miss chances to lead the opponent into traps.

Insurance might be an important advantage for alpha-beta. It appears that the poor quality of our knowledge favors methods with good insurance. On the other hand, methods that rely on high quality of knowledge to guide the search might find this knowledge inappropriately poor (over-generalized, erroneous, *etc.*).

[1] Again, this statement refers to alpha-beta in its text-book formulation, ignoring enhancements such as move ordering. On the other hand, traditional best-first searches can pick nodes anywhere in the tree to expand next.

[2] For example, if a blunder turns out to be a sacrifice.

1.2 Enhancements of Alpha-Beta

After listing all the above issues about alpha-beta, one could be surprised how well it performs in domains such as chess, checkers and Othello. However, it is necessary to stress again that the above listed deficiencies of alpha-beta are considering its basic formulation as given in [KM75]. The following enhancements of alpha-beta greatly increase the efficiency of practical alpha-beta implementations for many domains. Nonetheless, these problems still exist, they are merely masked.

Iterative Deepening: Instead of searching to full depth right away, iterative deepening increases the search depth by one at a time. This counterintuitive approach allows to store useful information from earlier iterations to be used in later iterations to increase efficiency and it helps to control the time spent for a move decision, since search times for a fixed search depth vary significantly for different problems. This is how practical implementations of alpha-beta address the issue of *Stopping*. This is by no means an optimal solution, since it only helps to control that a set time limit is not exceeded.

Move Ordering: Alpha-beta is most efficient if the best successor of each node is searched first. Information stored from previous iterations helps to achieve nearly optimal move ordering. A good move ordering allows the search to address the question about which node to *Expand Next*[3].

Transposition Tables: Transposition tables aid in the above mentioned task of move ordering by storing information from iteration to iteration. Furthermore, they allow for the detection of transpositions (a position in a search that can be reached by different move sequences) and thus eliminate duplicate search effort.

Partition Search [Gin96] generalizes the concept of transposition tables. Instead of storing information for individual positions, information for sets of positions is stored, increasing the usefulness of transposition tables. Ginsberg could show the merits of Partition Search for bridge, but no successful attempt to apply this algorithm to chess has been reported.

Forward Pruning: The null-move heuristic is one chess-specific forward-pruning heuristic that is derived from the observation that it is almost always better to make a move than to pass. This enhancement decreases the search depth if two moves in a row by one player do not help to bring the value of a position back into the alpha-beta search window. This powerful heuristic, used recursively throughout the search tree, effectively decreases the search depth in "bad" lines, thus addressing the issue of *Bad Lines*. Other domains favor different methods, such as ProbCut in Othello [Bur95].

Search Extensions: Domain-dependent (such as check extensions) and domain-independent (singular extensions) knowledge can be used to increase the search depth for certain lines of play. Search extensions also address the *Bad Line* problem.

Null-Window Searches: The narrower the search window, the more efficient alpha-beta is. Null-window searches ($\alpha + 1 = \beta$) combined with a good move ordering improve the efficiency of alpha-beta, effectively pruning or limiting the amount of search effort spent in "bad" lines and thus tackling the *Bad Line* problem.

1.3 Conclusions

The above enhancements reduce, to some extent, the shortcomings of the basic alpha-beta algorithm. However, the exponential growth of the search tree inherently limits how deep alpha-beta can search, even with all the enhancements above. Ever increasing machine power should not be the only source of performance improvements. With diminishing returns for additional search depth on the horizon [JSB+97], selective methods and additional knowledge become more and more interesting as either enhancements to alpha-beta or even as completely new algorithms.

[3] Plaat shows [PSPdB96] that for fixed-depth searches, one can implement a best-first minimax algorithm using alpha-beta with null-window searches and transposition tables.

2 Alternative Approaches

The literature provides a large body of search-related papers that suggest different ways to search two-person zero-sum perfect-information games. Here, each method is presented by its general idea, how the authors suggest to incorporate the idea in an algorithm and what the advantages/disadvantages of the proposed algorithm are.

The subsections divide the methods into five groups according to the dominant feature of the method: *Backup Rule*, *Node Value Representation*, *Expand Method* and *Opponent Modeling*. Note that this grouping is subjective, but allows a more structured presentation than purely chronological ordering would. Within each subsection the methods are described in chronological order according to publishing date of the earliest versions of those methods.

2.1 Backup Rule

This subsection discusses methods that propose new backup rules. Methods using different backup rules because they work with non-scalar values are described in section 2.2.

2.1.1 M&N-Backup Procedure

This proposal by Slagle and Dixon [SD69] was formulated to overcome the problem with the *Value Backup*. They propose a backup function that uses the M and N best successor values of a node for max and min nodes, respectively. Intuitively expressed, Slagle and Dixon tried to find a way to express the notion that it is more desirable to have more good choices at a node than less. The backup function can be any function suitable for the domain. Note that minimax is one special case of this method.

The algorithm is similar to the minimax algorithm. However, one could think of using this idea as a backup procedure in other search algorithms as well.

This approach addresses the issue of *Value Backup*. Unfortunately, enhancements such as alpha-beta cutoffs might not be usable, since all the node values need to be true values and not just bounds.

2.1.2 Product-Propagation Procedure

An evaluation function does not necessarily return a game-theoretic value nor an absolute value indicating the goodness of a position. It could also return the probability of this position to be a (forced) win. Under the assumption of independence of the values of sibling nodes it is possible to apply the rules of probability theory as suggested by Pearl [Pea81].

This method is a backup procedure only and not a complete algorithm. The backup rule, called *product-propagation rule*, is the following. For max nodes the probability of being a win is

$$1 - \prod_i (1 - p_i)$$

and for min nodes being a win node the probability is

$$\prod_i (p_i),$$

where the p_i are the probabilities for the successors to be a win node.

It is rather easy to transform the output of an evaluation function such that it returns (pseudo) probabilities between 0 and 1. However, the assumption of independence of the evaluation of sibling nodes is generally attacked as being unrealistic. Unfortunately, this backup rule is not useful if some of the successors are evaluated with bounds (as alpha-beta does).

2.1.3 Average Propagation

Nau, Purdon and Tzeng [NPH86] argue that since both minimax propagation and product propagation have their justification in actual game playing and search models, an average of both propagations could be useful.

Again, the backup procedure is just part of an algorithm. The proposal is to propagate the average of minimax backup value and the product-propagation value.

Nau, Purdon and Tzeng argue that the structure of the game and the evaluation function decide which backup procedure works best.

2.1.4 Min/Max-approximation

The basic idea behind Rivest's approach [Riv87] is to penalize each move in the search tree. The penalty is higher for "bad" moves than for "good" moves. The algorithm then finds the leaf node with the least cumulative penalty along the path from the root to this leaf and expands it, because this is the leaf the value of the root most heavily depends on. To allow one to compute the partial derivative of the root value as a function of the value of each of the leaf nodes, the traditional min and max functions are replaced by derivable approximations.

The algorithm traverses the tree iteratively, going down to the subtree where the leaf with the lowest cumulative penalty is situated. This leaf is expanded and going up the tree, necessary values in the tree are updated. This is similarly used in other algorithms as well, such as B* and conspiracy numbers. The difference is in using the penalty of the moves to find the leaf node to expand next.

How are the penalties for each move determined? Generalized mean values (p-mean) are used to approximate max and min functions in the following way[4]:

$$M_p(a_1, a_2, ..., a_n) = (\frac{1}{n}\sum_{i=1}^{n}(a_i^p))^{1/p}$$

With $p \to \infty$ this function approximates the max function and with $p \to -\infty$ it approximates the min function. The advantage of this generalized p-mean function is its sensitivity to the values of all the successors and not just to the largest (max) or smallest (min) values. Using generalized p-mean functions enables the search to select moves that are not necessarily best in the sense of minimax searches, but have many good alternatives (not just one best that might be mis-evaluated).

Since the values of all the successors of a node influence its value if the above formula is used, a dependency measure for each successor can be calculated. It expresses how much the parent depends on the value of each successor. A penalty is given to each successor; the less dependent a parent is on the value of a successor, the greater is this penalty. The algorithm will expand the leaf node with the smallest penalty summed over the path that leads from the root to the leaf.

This approach aims to overcome the *Value Backup* problem. The min-max approximation bases the value of a parent on the values of all its successors. Unfortunately, positions are penalized where horrible moves, such as queen blunders or help-mates, are possible, because *all* successors of a node influence the value of a node. The algorithm has a better understanding of which node to *Expand Next*, because of the iterative traversal of the tree and the penalties indicating where it is worthwhile to concentrate search effort. The expansion strategy proposed here assumes the change of a value of a leaf is equally likely throughout the tree, which is obviously not true. Therefore, the expansion will not necessarily make the value of the root more reliable.

This idea was implemented only for Connect-4. No chess related results are available.

2.2 Node Value Representation

2.2.1 B*

B* was proposed by Berliner [Ber79] and has since then been revised [BM95] to overcome problems found with the original formulation.

The fundamental insight leading to B* was that it is not necessary to have the exact minimax values of the successors of the root to find the best move. If we can generate bounds for the minimax value of those successors, then proving that the lower bound of the best successor is greater or equal to the upper bounds of all the alternatives is sufficient to determine the best successor of the root[5].

[4] In our case the a_i are the values of the successors of a node.

[5] However clever that might seem, alpha-beta is not computing the minimax values either, but generates upper bounds for all the moves but the first, effectively cashing in on most of the possible tree size reduction.

The goal of establishing a lower bound of the best successor that is greater or equal to the upper bound of all alternatives is called *separation*. There is no need to search any further, once separation is established, because the best successor of the root has been found. Furthermore, B* is a best-first search. The bound information is used to traverse the tree and expand the most relevant leaf node.

Over time this clear and simple idea was engineered to accommodate other enhancements to make the algorithm work. The basic idea remains to either push the best successor's lower bound above the best upper bound of the alternatives (PROVEBEST) or to lower the upper bounds of the alternatives (DISPROVEREST). The initial paper suggests static evaluation functions as optimistic/pessimistic estimators. Later, null-move searches were proposed.

This short sketch of the algorithm is not intended to be complete, but to merely reflect the basic ideas behind B*. The current B* implementation as given in [BM95] is elaborate and is largely due to attempts to overcome problems encountered with the initial design.

Separation is a natural way to solve *Stopping*. Once the optimistic values of all the alternatives are less than the pessimistic value of the best move there is no point to search further. The B*-algorithm gives also a solution to the *Expand Next* problem. To achieve separation, B* uses the potential of a state. The potential of a state is the likelihood that the value of a state improves if expanded further.

In the original paper, with the introduction of bounds, the B*-algorithm was the first to break with the tradition of single scalars as node evaluations in search algorithms. The current node representation in the algorithm of B* shows that Berliner does not consider the two bounds of the initial paper a solution to the *Single Value* issue. His current implementation uses the bounds, one realistic value and two additional probabilities. This opinion results from his experiences with the work of Palay ([Pal85], see section 2.2.2). Using searches as evaluation functions at the leaf nodes of the B*-tree increases the *insurance* against B* missing something.

2.2.2 Probabilistic B*

Palay picked the work of Berliner up and generalized the original B*-idea [Pal85]. Instead of using bounds[6] and thus implying a uniform distribution of the random variable *delphic value*[7], Probabilistic B* (PB*) uses a probability distribution to describe the location of the delphic value.

Palay introduced two phases to the B* algorithm, SELECT and VERIFY, instead of the two strategies in the original B*. First, the SELECT phase selects the best move and tries to improve its evaluation, then the VERIFY phase tries to find a better alternative, thereby hoping to lower the bounds of the alternative moves. Palay proposed to use null-move searches as an optimistic evaluation function.

Probability distributions are the most general way to address the *Scalar Value* problem. The intractability problems Palay discovered with his approach show that this amount of information is too much. The computational prize is too high compared to the benefits for the search.

Separation[8] was found to be rather hard to achieve to terminate the search. Therefore, Palay introduces a relaxed separation criterion, called *domination*. The domination is a statistical measure that indicates with what probability one move is better than the best alternative.

Many of Palay's ideas are used by the new B* algorithm and are implemented in *B*-Hitech*, a high performance chess program/machine.

2.2.3 Conspiracy Numbers

The basic idea of this approach proposed by McAllester [McA88] is to record for every node x in the search tree, how many leaf nodes in its subtree have to change their value in order to change the value of node x to another value v. These numbers (every possible value v for x has such a number) for node x are called *conspiracy numbers*, since this is the minimum number of leaf nodes that have to conspire to change the value of x to v.

The main goal of the algorithm is to make a change of the value of the root node unlikely if further leaf nodes are expanded. Unlikely means here that more than c nodes have to conspire to change the value v of

[6] See section 2.2.1 "B*".

[7] This term was coined by Palay himself and it refers to the answer of an oracle about the value of a node - such as a deep search.

[8] *Separation* is explained in section 2.2.1 B*.

node n to $w \neq v$. c is called the *conspiracy threshold*. The higher c is, the greater the confidence in the value of the root node. All the values v for a node x that have a conspiracy number less than c are called *plausible values*. To achieve a root node value that is unlikely to change, the algorithm can either try to increase or decrease the value at the root. It is almost irrelevant, whether the algorithm succeeds or not, because it likely increases the conspiracy numbers of some of the still plausible values of the root node. This leads to increased conspiracy numbers, possibly beyond c, and thus, narrowing the range of plausible values of the root node.

The algorithm in the formulation of Schaeffer [Sch90] chooses in each iteration between the options of increasing or decreasing the root value by looking at where more plausible values are, above or below the current minimax value of the root. A target value is set at either the biggest or smallest plausible value of the root node, depending on if increasing or decreasing the root node value was chosen, respectively. After deciding on this global strategy, the algorithm has to decide to which subtree to descend to find the leaf node to expand. It will always choose the subtree where the goal of increasing or decreasing of the root node value seems to be easiest, that is where the conspiracy number for the target value is smallest. The leaf node is expanded and its conspiracy numbers and minimax values are updated. Then the algorithm updates the conspiracy numbers and minimax values of all ancestors of the expanded leaf node.

The most appealing aspect of this algorithm is that domain-independent knowledge is used to grow the tree. This knowledge is used to find the next leaf to expand, providing one more possible way to address the *Expand Next* problem. The algorithm is highly aware of the problem that heuristic values might change. This is expressed in the idea of the conspiracy numbers—the algorithm's way to handle *Heuristic Error*. Because the conspiracy numbers contain much more information than a single scalar value (they can be viewed as probabilities), conspiracy-number search also gives an answer to the issue of *Scalar Values*. The problem of *Stopping* appears in a different shape: what is a suitable conspiracy threshold? Ideally, we would like to make it dependent on the quality of the evaluation function, but we have no idea about this quality either. Because this algorithm is a highly selective algorithm, *Insurance* becomes an issue. Some lines are followed to excessive depth, whereas others get neglected. The tendency to expand forced lines (lines with few responses for the opponent) can be a problem in solving a problem and/or terminating a search.

One fundamental problem with this algorithm in its current formulation is that the conspiracy numbers do not take into account that small value changes are more likely than large changes. Implementing conspiracy numbers soon reveals difficulties: Since we need a conspiracy number for every possible value a node can have, the space and time requirements are high if the number of possible values is large. Additionally, many nodes have similar conspirators for the same minimax values - making it difficult for the search to distinguish between alternatives.

McAllester, fully aware of the problems of conspiracy-number search, developed a new algorithm called *ABC Search* [MY93]. It is essentially the alpha-beta algorithm that uses the knowledge about conspirators to decide about the search depth. Another algorithm using the conspiracy idea is Lorenz' *Controlled Conspiracy Number Search* [LRFM95]. With the introduction of targets for conspiracy numbers for values at the root node, many of the drawbacks and problems of the original conspiracy-number search can be avoided. Experimental results are promising.

2.2.4 Uncertainty

Horacek [Hor87] proposed a special case of the original B* formulation, enhanced by a weighting factor. Whenever the evaluation function discovers difficulties (dynamic features) with the evaluation of a state, such as trapped or pinned pieces, it returns a weighted pair of values, where the weight expresses a tendency towards the value that is more likely. According to Horacek, weighted pairs should be used only in rare occasions.

The algorithm expands the tree in an alpha-beta-like way. A new backup rule is used to accommodate the weighted pairs. The move selection at the root will deal with the weighted pairs in case they reach the top of the search tree.

Positions evaluated with weighted pairs can be thought of as hard to understand. The algorithm understands which positions are easy to evaluate and which are not, enabling it to prefer those it understands. This is one way to address the *Heuristic Error* and *Scalar Value* problems. Both values are similar to pessimistic and optimistic evaluations and the weight determines where the realistic value is. It is not clear what the

advantages of this notation are compared to the new B* or fuzzy number notation.

This idea was implemented in the chess program *Merlin*.

2.2.5 Meta-Greedy Selective Search

The idea developed by Russel and Wefald [RW89] is to apply decision theory to search algorithms to give the algorithm a thorough understanding of the decisions it has to make. More precisely, if an algorithm knew the utility of the next search step compared to the cost of the time used to execute this search step, it could make an intelligent decision whether to stop the search using the result obtained so far or to continue searching.

The Meta-Greedy Selective Search (MGSS*) algorithm keeps the entire search tree in memory plus a list of *relevant* nodes ordered by their minimax value (a relevant node is a node that can, by changing its own value, change the best successor node of the root). Each iteration of the algorithm takes the first leaf node from the list and performs the test for termination: If the expected utility of the expansion of this node is less than the cost of the expected time for expanding this node then the algorithm stops. Otherwise, the node is expanded, values are propagated up the tree and the relevant successors are inserted into the ordered list of relevant nodes.

This algorithm deals with *Stopping* in a natural way. It has an inherent understanding of the value of time and can relate the value of expected computational results to the value of the expected time of the computation (given all these values are known). By expanding the node with the highest expected ratio between utility of the expansion to the time cost for this expansion, the algorithm solves which node to *Expand Next*.

To reason about cost of time, utility of search results and so on, different domain-dependent probabilities are needed. Extensive statistical experiments have to be conducted to get reliable data. Relevant leaf nodes exist only if the root node still has conspiracy numbers (see 2.2.3) of one. As soon as this number is greater than one for all alternative values, the list of relevant nodes is empty. This is an undesired effect. Moreover, this list does not even contain all conspirators, but only the ones that can change the best node - which are even fewer nodes.

Except for Russel and Wefald's implementation there is no other one known. The empirical results are not very reliable, since both versions, alpha-beta and the proposed algorithm, were "not very carefully engineered" [RW89].

2.2.6 Bayesian Game Tree Search

This approach (BP) by Baum and Smith [BS97] also uses an evaluation function returning probability distributions. Different from previous approaches using probabilities, the meaning of the probabilities here represents the expected change of the value if the search would expand this leaf node. The probability functions are propagated in the way proposed by Palay. A leaf relevance measure (QSS) is used to grow the tree such that relevant lines are explored deeper. This leaf relevance measure reflects the influence each leaf has on the decision theoretic utility of expanding every leaf of the tree.

The algorithm expands a set of nodes at each iteration. This is the top fraction of the nodes with highest relevance (called Q step size, QSS; for the definition see the original paper). The calculation of this utility requires influence functions for the changes of leaf nodes for changes at the root. The algorithm stops if the estimated expansion utility of all nodes is less than the estimated cost of time to expand them.

One innovation lies in the new interpretation of the probability distribution returned by the evaluation function. Baum and Smith develop a new argument to justify the assumption of independent probability functions of the successors. They assume that since their function expresses the possible changes of the value of a node if the node is expanded, which is the error of the evaluation function, this error is independent for siblings. They argue that even thought this assumption does not hold, the resulting procedure is stronger than alpha-beta and probability product.

The paper suggests an interesting way to train the evaluation function to return probability functions. The example positions are divided into *buckets* before the training and for each of the buckets (determined by predefined features) a probability function of how the value will change if the search would extend the node one ply is learned. Additionally developed engineering tricks help to make the proposal efficient in practise.

The authors extensively tested PB against alpha-beta and probability product. They conclude that, at least for Othello, Warri, Kalah and Pearl's P-Game, PB is the strongest of the three approaches. A number of questions remain to be answered. Even though transposition tables are not as important in Othello as they are in chess, why did the authors not implement them and would transposition tables change their results? Especially if one considers that alpha-beta using iterative deepening without transposition tables can hardly make use of previous iterations. So far, attempts to implement ProbCut in chess failed. Apparently it is hard to find reliable estimators for how deeper searches will change the current value of a position. Given this difficulty, it is hard to believe that one can easily construct an evaluation function for chess that BP would require.

2.2.7 Fuzzy Numbers

Since the knowledge used in search is imperfect but still gives a certain guideline, it should be possible to express ideas such as "The approximate value for this node is x". Furthermore, if we know that the approximate value favors one of the players, we should be able to express this. Certainly, probability functions, as Palay proposed, are one way of achieving the above desired properties. However, the computational effort incurred by them is high. Considering the low accuracy of the input probabilities at the leaf nodes it is questionable if the effort is justified to compute exact probabilities. Using fuzzy numbers [Jun94], offers a low computational method with reasonable expressiveness. A fuzzy number consists of three parameters that can be interpreted as pessimistic, realistic and optimistic value.

The new algorithm is a modified alpha-beta algorithm to accommodate the fuzzy numbers as node evaluations. A new backup procedure, cutoff criterion and best-node selection is needed. The backup procedure basically maximizes or minimizes (according to the type of node) all three parameters of the fuzzy numbers thereby creating a new fuzzy number for the parent node. A new best-node selection procedure was necessary to handle overlapping fuzzy numbers. A risk parameter is applied to resolve conflicts between several good alternatives. The risk can be chosen according to the game situation.

The low computational cost of dealing with fuzzy numbers[9] and the higher expressiveness of fuzzy numbers compared to scalar values addresses the *Single Value* problem.

The new backup procedure deals with the problem of *Value Backup*. The current backup formula is not satisfying, since other aspects of the *Value Backup* problem remain to be solved. However, fuzzy numbers seem to be flexible enough to accommodate more elaborate methods for backup procedures without losing the advantage of the low computational overhead. This approach overcomes the bonus/penalty problem that any evaluation function faces producing single scalar values: If it encounters positive and negative features in the position, the bonuses and penalties tend to cancel each other. Most importantly, fuzzy number search provides an elegant way to describe quiescence search: If pessimistic and optimistic evaluation are too far apart, the knowledge can be considered insufficient and a further expansion is beneficial. The main advantage is the dynamic way to define selectivity, no reiteration is necessary to decide, whether a node should be expanded. The quality of the evaluation of the node itself is the criterion.

However, it is hard to come up with the optimistic and pessimistic values, just as Berliner discovered in his work. It is therefore questionable if this method will find its way into a practical application.

2.3 Expand Method

2.3.1 Bandwidth Search

Harris [Har74] departs from the classical formulation of best-first searches such as A* and adds constraints to the heuristics used. Whereas A* uses an admissible heuristic (a heuristic never overestimating the distance to the goal), his new constraint on the heuristics $h'(n)$, called bandwidth-heuristics, is the following:

$$h(n) - d \leq h'(n) \leq h(n) + e$$

[9] Creating the fuzzy number in the evaluation function is "just" a matter of combining the already calculated terms of the evaluation function in two more ways to get the optimistic and pessimistic values. The backup rule has to take only three numbers for each successor into account and is not operating over, for example, functions as in PB*.

where d and e are the error bounds of the heuristic. This means that the heuristic $h'(n)$ always guesses the *distance to the goal*[10] within a certain tolerance zone of the actual distance $h(n)$. Nodes with a value outside this zone can safely be cut off.

Harris' algorithm is an adaption of the A* idea to two-person game playing: Each open (not expanded or leaf) node has associated with it a heuristic value, representing a guess about the distance to the goal, and a cost, which is the distance from the root to this node. Closed (expanded or internal) nodes inherit the values of the "best" of its successors, min nodes receive the minimum and max nodes the maximum value of the successors. Here the player nodes are min nodes, because we are looking for the shortest path to the solution.

Expanding a node will expand *two* plies at a time[11]. This way, there will be only one kind of open nodes (min) and an order between them is possible. The node to expand next is chosen to be the leaf node of the principal variation. This is not the same as Korf's best-first minimax [KC94] (see section 2.3.5), because the cost of getting to the leaves is used here additionally, addressing the *Insurance* problem.

This algorithm has a very simple stopping criterion. If a node is found that lies within the defined range of $e + d$ from the optimal goal, the search stops. It is not clear how the idea of finding the shortest win ties into the picture of game playing. Shorter wins are surely preferable, but not the primary goal of the search and the cost of a move has no practical meaning (except maybe for time control issues).

Suppose the interpretation of path length to the goal makes sense in the framework of game playing. Then there is a drawback for this algorithm. We are finding any goal node that is within the range of $e + d$ from the optimal goal. We might not find an optimal goal and if losing and winning is close enough for the quality of our evaluation function (d and e are too large) then we might miss a winning move, picking a losing move instead.

Harris' algorithm does not assume a perfect evaluation function as alpha-beta does. Therefore, the algorithm can deal with imperfect heuristics assessing the quality of the goal found by knowing the quality of the heuristic function. The way the algorithm determines the node to expand next provides a better selectivity than alpha-beta and more insurance than most of the selective algorithms (such as best-first minimax), because the cost of a move is used to eventually prevent the deep branches from being expanded to excessive depth, giving shallow branches a chance to be looked at.

One more problem is how to determine d and e since we do not have any idea about h. However, if estimating d and e was possible, then Harris' proposal could be applicable, possibly cutting down the search space.

2.3.2 SSS*

Stockman [Sto79] proposes a best-first minimax algorithm called SSS*. SSS* is related to A*, a best-first search algorithm for single-agent problems (OR graphs). SSS* was adapted to search AND/OR graphs.

Similar to A*, SSS* manipulates an OPEN list of nodes which is sorted in descending order according to the merit of the nodes. The first node in the list has therefore the highest merit and is the one to expand next.

SSS* was criticized for being difficult to understand in the original description. But it was proven to evaluate no more nodes than alpha-beta would. Simulations show that SSS* evaluates significantly less nodes than alpha-beta. The disadvantages are obvious and the same as in A*: the large memory requirements and the overhead of maintaining the sorted OPEN list. However, Plaat *et.al.* [PSPdB96] gives a reformulation of SSS* using null-window alpha-beta calls (with transposition tables): MT-SSS*. This algorithm is searching the same leaf nodes in the same order as SSS*. The surprising message is the following: SSS* in the new formulation using alpha-beta with enhancements overcomes the two SSS* problems, the memory requirement and the overhead of the OPEN list. Thus, alpha-beta with enhancements searches just as few nodes as the best-first procedure SSS* if called with the right windows.

[10] We will come back to the questionable meaning of this term in the discussion.

[11] Leaf nodes in the first level are discarded and alpha-beta cutoffs are used.

2.3.3 Equi-Potential Search

Anantharaman [Ana90] proposes a selective search method, called Equi-Potential Search (EPS) that, unlike best-first methods that use a list to keep track of the best nodes to expand next, uses a depth-first strategy and still selectively expands leaf nodes. The algorithm grows the search tree such that all the leaf nodes have roughly the same potential of improving the move decision when expanded next. Equi-Potential Search decides whether to expand a leaf node solely on the (heuristic) information available at the leaf node itself to achieve a depth-first behavior.

Each successive iteration of EPS has a certain effort threshold E associated with it which increases from iteration to iteration. For each leaf of the search tree two measures are calculated:

S is the probability of a 1-ply extension of the leaf resulting in a change of the best top-level move multiplied by the expected benefit from such a move change.

P is the expected cost of a 1-ply extension of the leaf, including the expected cost of any resulting re-search of sibling nodes of any ancestor of this leaf.

The ratio S/P gives a cost-benefit measure that is compared against the effort threshold E of the current iteration. If $S/P < E$ the leaf node is expanded and the procedure is applied to its successors.

The appealing idea is the depth-first formulation of the algorithm, avoiding all the overheads of priority queues/lists. Furthermore, EPS can be run to tune itself. The tuning resulted in improved performance.

Some important problems are not addressed in Anantharaman's work: When should the algorithm stop increasing E? What is the maximum effort the algorithm should spend to determine a best move in a position? How does EPS perform in comparison with alpha-beta?

2.3.4 Singular Extensions

The motivation to the method of singular extensions proposed by Anantharaman, Campbell and Hsu [ACH90] was to find a domain-independent and dynamic way to formulate and employ search extensions. If a move is much better than any other alternative in a given position (it is singular) and is likely to affect the outcome of the search if its value changes, this move is researched with increased depth.

This method can be used as an algorithmic enhancement to alpha-beta. It is not an algorithm as such.

The most valuable idea here is the domain-independent way the search is guided to distribute effort in the search tree. Therefore it is a domain-independent approach to solve the *Bad Line* issue by adding more selectivity. However, additional search effort is needed to detect if a move is singular. In [Ana90] Anantharaman indicates that the large gains reported in the initial paper are only partially reproducible. He states that singular extensions "seem to just break even". The additional search effort cancels the gains for average searches.

2.3.5 Best-First Minimax

The basic idea behind Best-First Minimax as suggested by Korf [KC94] is to always expand the node at the end of the current principal variation.

The resulting algorithm traverses the tree up and down: down to reach the leaf of the current principal variation and up to propagate the new values. Some observations help to understand why this algorithm works: If min had to move last in the current principal variation then the value of the root tends to decrease, since the last non-leaf node of the principal variation was a min-node which results in the selection of a small value. On the other hand, if max had to move last in the current principal variation, the value usually increases. Thus, regardless of what type of node was expanded last, the principal variation is likely to change, and therefore the place where the next node is expanded is changed too.

Korf formulates a new way to find the next node to expand. The simplicity of the algorithm is appealing, however, it is doubtful if this idea can solve the *Expand Next* problem, because of problems with the *Insurance*. The algorithm might miss something in subtrees that are off the principal variation because of the high selectivity. Korf's results consequently favor a hybrid version of best-first minimax and alpha-beta to combine the thoroughness of alpha-beta at the root and the selectivity of best-first minimax at the leaves. The algorithm practically depends on the odd-even effect. A nearly perfect (static) evaluation function (meaning:

not producing the odd-even effect) would not allow the algorithm to switch the principal variation often enough to distribute effort between different subtrees.

2.4 Opponent Modeling

Alpha-beta unrealistically assumes that the opponent uses the same methods for determining the best move. That includes, among others, the evaluation function, search extension strategies, back-up rules, search depths and other parameters of the decision making process of the opponent. Following are methods that try to overcome this simplifying assumption of alpha-beta.

2.4.1 *-Min

Whereas minimax-based algorithms assume that the opponent is minimizing the same values as the player maximizes using the same evaluation function, the *-Min procedure as proposed by Reibman and Ballard [RB83] treats min nodes as chance nodes. This means, because the opponent has a different evaluation function he will make different decisions than the player.

Instead of propagating the minimum of all successors as minimax does, at min nodes *-Min propagates the weighted sum of the values of all b successors

$$P_S M_1 + (1-P_S)P_S M_2 + ... + (1-P_S)^{b-1} P_S M_b = P_S(\sum_{n=1}^{b}(1-P_S)^{n-1} M_n),$$

where P_S is called the predicted strength of the weaker and fallible opponent. The predicted strength is the probability that given a choice of b moves, the opponent will choose the n^{th} best move over the $(n+1)^{th}$ best move.

This model is rather unrealistic. The randomization does not model the actual decision making process of the opponent, especially its own evaluation function. Bounds are not sufficient for this model, which makes alpha-beta cutoffs impossible.

2.4.2 ProbiMax

In an endgame it is sometimes important to reach mate positions within a certain number of moves (because of the 50-move rule). Jansen proposes an algorithm to exploit the human fallibility by deliberately playing suboptimal moves to make it harder for the human opponent to reach the game theoretic outcome of the position within the required number of moves. The program is assumed to have perfect knowledge by means of an endgame database.

Instead of modeling the opponent's move decision using the minimum as minimax-like algorithms do, Jansen [Jan93] proposes the probabilistic model ProbiMax. The value of a min node is modeled by

$$V(p) = \sum_i \omega_i V(p_i),$$

where $V(p)$ is the value of a position, p_i are the successors of p and ω_i is the probability of position p_i being chosen by the opponent.

The value of a max node is given by the following formula

$$V(p) = \max_{i:D(p_i) \geq D(p)-\delta} V(p_i).$$

$D(p_i)$ is the depth to the leaf node of the game tree, thus a mate or stale-mate position. The player is only considering moves that risk to lower the distance $D(p_i)$ to this goal node by a maximum of δ. The idea here is to avoid positions that seem to be good, but are too close to the undesired outcome of the game compared to the maximal distance.

The model of the opponent needs to be rather accurate, to determine all the probabilities ω_i for the possible positions p_i well enough for this proposal to work. It is not clear how to obtain even approximations of these probabilities.

2.4.3 Speculative Play

Jansen [Jan93] proposes speculative play for the program playing the weak side but with perfect knowledge provided by an endgame database. The proposal consists of choosing, out of several optimal moves, the one that has *the smallest relative number of optimal replies* for the opponent. Uiterwijk and van den Herik [UvdH94] present similar ideas. There a "bonus/malus" schema is developed which encourages to play moves that have only few good responses for the opponent in mate-like positions.

There is no complete algorithm to Jansen's idea, except for the mathematics. Uiterwijk's proposal to add a bonus can also be viewed as a special backup mechanism rather than a completely new algorithm.

Jansen's approach is only useful if perfect knowledge is available. For chess, this method is only useful in the last stages of the game, when endgame databases provide perfect information.

2.4.4 Opponent Model Search

Iida et al. [IUvdHH93] give a new algorithm using the knowledge about the fallible opponent. The idea is to model the opponent such that one can predict where the opponent makes mistakes and can exploit those mistakes.

The algorithm has two values for each node. They represent what the player and the opponent think the value of the node is. Leaf nodes are evaluated with both evaluation functions. The backup rules assume the opponent to play minimax using only the opponent's values. However, the player uses both, player and opponent values, to determine what value a parent node should have, thus exploiting possible mistakes of the opponent.

This algorithm assumes perfect knowledge about the behavior of the opponent by means of evaluation function and search depth. This is an idealized case.

2.4.5 M*

Carmel and Markovitch [CM96] try to exploit the known weaknesses of a fallible opponent. A player is modeled by its assumed search depth, evaluation function and possibly a model for its opponent.

Instead of modeling the opponent as the exact opposite to the player, as minimax searches, M* assumes a possibly different model for the opponent. This means, instead of minimizing over all the successors of an opponent node, M* is called recursively to return a move that the opponent would make at this opponent node. The player then assumes this move as given and makes it. The resulting node is treated as a player node again (like the root). The opponent model can again contain a model for the player, which in turn can contain a model for the opponent etc.

This schema leads to repeated retraversing of the tree and to multiple evaluation of leaf nodes. Carmel and Markovitch give a modified version of the algorithm that traverses the tree only once and does all the evaluations for the multiple models and the backups in one pass. In this algorithm a node has multiple evaluations and the backup rule keeps track of how the values are propagated such that the M*-value reaches the root node. Carmel and Markovitch also give a method how to learn the opponent's model.

There are several fundamental and technical problems to the proposed methods. Assuming that the players knew that their opponent is stronger they would either use the knowledge about the opponent's evaluation function to become stronger or would not be able to do so because of resource bounds. Therefore, weaker players cannot benefit from this method. To model the opponent by means of an evaluation function and search depth assumes the existence of a superset of evaluation features, which is rather unrealistic in today's diverse community of playing programs. None of the programs actually uses fixed depth searches, but relies on extensions and quiescence search. These are additional parameters of the search needed to adequately model an opponent. Falling into one's own trap due to insufficiently learned parameters is possible considering the complexity of the task of modeling the opponent.

Additionally, the extensive use of multiple evaluation functions is computationally expensive and the described pruning method is less efficient. It is questionable, even assuming all other problems are solved, if the benefits of modeling the opponent can outweighed the losses due to the higher computational costs. In addition, this method would fail when used against humans. If the opponent is insufficiently modeled, playing minimax risks the least.

3 Conclusions

Over the last 40 years, researchers, aware of the fundamental deficiencies of the traditional alpha-beta algorithm, have been proposing fixes and new methods to overcome those difficulties. Did they result in practical alternatives to alpha-beta in computer chess? No.

Despite the multitude of papers concerned with the problems stated at the beginning, only few of the proposals have been implemented in a competitive program for an interesting game. Of all the presented approaches, B* is the one coming closest to being a complete and competitive solution for chess. Conspiracy numbers are used as well, but only in few programs. Singular extensions are used, but their value was reported to be less than initially thought. Outside computer chess, in the domain of two-player, zero-sum, perfect-information games, BP was shown to outperform alpha-beta in Othello, with both programs lacking a transposition table. Bridge, as a representative of imperfect-information games, is an example for success with Partition Search.

Many of the here examined approaches are related. For example, ideas formulated in MGSS*, PB* and Min/Max approximation can be found in later proposals such as EPS and BP[12]. It is unclear how successful those alternatives to alpha-beta will be in the future, because many problems remain to be solved.

Some methods try to backup more information than a single scalar value. This can quickly lead to high memory and computing resource demands, as in Conspiracy Numbers and PB*. The problem here is to find the right amount of information, such that gathering and processing it constitutes no problem and the search still gets enough information.

Many of the approaches try to find a better way of selecting the next node(s) to expand, hoping to use their computational effort in a more reasonable way. Those approaches are often faced with the *Insurance* problem, because they are too focussed on selected parts of the tree, as in Min/Max approximation, MGSS*, Best-First Minimax and B*, PB* to some extent. This is the old forward-pruning problem in a new disguise.

Other methods need much more domain-dependent knowledge, such as utilities, probabilities or costs, to solve problems like *Stopping* or which node to *Expand Next*. The already rather hard procedure of gathering knowledge will become more difficult using those new methods. Additionally, probability functions estimating how a value will change if a node is further expanded, depend on the evaluation function, search extensions and rules governing the quiescence search. If only one of the three is changed, it is likely that the probabilities change as well.

For the reasons given in section 1.2, alpha-beta is a very efficient and successful search algorithm. When new methods and/or algorithms are proposed, they are compared against alpha-beta, an algorithm that has been refined and enhanced over the last 40 years. One should not expect new methods to immediately surpass alpha-beta performance wise, especially in domains such as chess, where alpha-beta is very well established. Furthermore, researchers proposing a certain idea are possibly not familiar enough with all the enhancements of alpha-beta to be able to use them to the benefit of their new algorithms. This paper attempts to communicate these ideas.

4 Taxonomy

The two tables at the end summarize the methods of section 2. The entries in Table 1 give a short description of 4 important characteristics of search algorithms. The row for alpha-beta serves as a starting point. Missing entries stand for "no issue".

The second table describes how each method deals with the problems stated in the introduction and contains two additional columns: one for the amount of work at leaf nodes and the other for how many cutoffs in the tree are possible. The table reflects the author's opinion, some of the entries are without doubt arguable. However, more important than the individual entries is the trend, i.e. small differences between algorithms, many "+" or "-". Some methods should work better than alpha-beta, even better than alpha-beta with enhancements, only judged by the entries in the table. Common sense and this observation lead to the conclusion that some of the identified problems have much greater impact on the performance of an algorithm than others.

[12] See [BS97] for a short description of the relationship of those algorithms.

5 Acknowledgements

An overview like this one can never be complete. Some descriptions of algorithms capture work in progress and cannot be exact in every detail and were therefore deliberately given only with the basic ideas. The author would like to thank the GAMES research group at the University of Alberta for their support in providing (links to) papers and feedback on the here presented condensed view on the subject. Candid remarks of the referees led to necessary improvements. Special thanks are due to Yngvi Björnsson and my wife Manuela for patiently debugging drafts of this paper many times.

Financial support was provided by the German Academic Exchange Service (DAAD), the Killam Foundation and the Natural Sciences and Engineering Research Council (NSERC), for which the author is extremely grateful.

References

[ACH90] T. Anantharaman, M. Campbell, and F.H. Hsu. Singular extensions: Adding selectivity to brute-force searching. *Artificial Intelligence*, 43(1):99–110, 1990.

[Ana90] T. Anantharanam. *A Statistical Study of Selective Min-Max Search in Computer Chess*. PhD thesis, Carnegie Mellon University, Computer Science Department, Pittsburg, May 1990. also: technical report CMU-CS-90-173.

[Ber79] H.J. Berliner. The B* tree search algorithm: A best-first proof procedure. *Artificial Intelligence*, 12(1):23–40, 1979.

[BM95] H.J. Berliner and C. McConnell. B* probability based search. *Artificial Intelligence*, June 1995. Also: Technical Report CMU-CS-94-168, School of Computer Science, Carnegie Mellon University, Pittsburg, PA.

[BS97] E.B. Baum and W.D. Smith. A bayesian approach to relevance in game playing. *Artificial Intelligence*, 97(1–2):195–242, 1997.

[Bur95] M. Buro. ProbCut: A powerful selective extension of the $\alpha\beta$ algorithm. *ICCA Journal*, 18(2):71–81, June 1995.

[CM96] D. Carmel and S. Markovitch. Opponent modelling in adversary search. In *Proceedings of the National Conference on Artificial Intelligence (AAAI-96)*, pages 120–125, 1996.

[Gin96] M. Ginsberg. Partition search. In *Proceedings of the National Conference on Artificial Intelligence (AAAI-96)*, pages 228–233, 1996.

[Har74] L.R. Harris. The heuristic search under conditions of error. *Artificial Intelligence*, 5:217–234, 1974. Also in: Chess Skill in Man and Machine, Springer-Verlag.

[Hor87] H. Horacek. Reasoning with uncertainty. In D.F. Beal, editor, *Advances in Computer Chess 5*, Noortwijkhoot, The Netherlands, 1987.

[IUvdHH93] H. Iida, J.W.H.M. Uiterwijk, H.J. van den Herik, and I.S. Herschberg. Potential applications of opponent-model search. *ICCA Journal*, 16(4):201–208, 1993. Published also in the proceedings of the 7th Conference on Advances in Computer Chess, Maastricht, The Netherlands.

[Jan93] P.J. Jansen. KQKR: Speculatively thwarting a human opponent. *ICCA Journal*, 16(1):3–17, January 1993.

[JSB+97] A. Junghanns, J. Schaeffer, M. Brockington, Y. Björnsson, and T. Marsland. Diminishing returns for additional search in chess. In *Advances in Computer Chess 8*, pages 53–67, 1997.

[Jun94] A. Junghanns. Fuzzy numbers as a tool in chess programs. *ICCA Journal*, 17(3):41–48, September 1994.

[KC94] R.E. Korf and D.M. Chickering. Best-first minimax: Othello results. In *Proceedings of the National Conference on Artificial Intelligence (AAAI-94)*, August 1994. Also: to appear in AIJ.

[KM75] D.E. Knuth and R.W. Moore. An analysis of alpha-beta pruning. *Artificial Intelligence*, 6:293–326, 1975.

[LRFM95] U. Lorenz, V. Rottman, R. Feldmann, and P. Mysliwietz. Controlled conspiracy-number search. *ICCA Journal*, 18(3):135–148, 1995.

[McA88] D.A. McAllester. Conspiracy numbers for Min-Max searching. *Artificial Intelligence*, 35:287–310, 1988.

[MY93] D.A. McAllester and D. Yuret. Alpha-beta-conspiracy search, 1993. URL: http://www.research.att.com/~dmac/abc.ps.

[NPH86] D.S. Nau, P. Purdom, and Tzeng H.C. An evaluation on two alternatives to minimax. In *Uncertainty in Artificial Intelligence*, pages 505–509, North Holland, Amsterdam, 1986.

[Pal85] A.J. Palay. *Searching with probabilities*. PhD thesis, Carnegie-Mellon Univ., Boston, Mass., 1983/85. See also (1985), book same title, Pitman.

[Pea81] J. Pearl. Heuristic search theory: A survey of recent results. In *Proceedings of the International Joint Conference on Artificial Intelligence (IJCAI-81)*, Vancouver, British Columbia, 24-28. Los Altos, Calif., 1981. Kaufmann.

[PSPdB96] A. Plaat, J. Schaeffer, W. Pijls, and A. de Bruin. Best-first fixed-depth minimax algorithms. *Artificial Intelligence*, 87(1–2):255–293, November 1996.

[RB83] A.L. Reibman and B.W. Ballard. Non-minimax strategies for use against falliable opponents. In *Proceedings of the international conference on artificial intelligence AAAI-83*, pages 338–343, Los Altos, CA, 1983. William Kaufman.

[Riv87] R.L. Rivest. Game tree searching by min/max approximation. *Artificial Intelligence*, 34(1):77–96, 1987.

[RW89] S. Russell and E. Wefald. On optimal game-tree search using rational meta-reasoning. In *Proceedings of the International Joint Conference on Artificial Intelligence (IJCAI-89)*, volume 1, pages 334–340, 1989. Also: Do the Right Thing.

[Sch90] J. Schaeffer. Conspiracy numbers. *Artificial Intelligence*, 43(1):67–84, 1990. Also in Advances in Computer Chess V, D.Beal (ed.), Elsevier Science Publisher, Amsterdam, Netherlands, pp. 199–218, 1989.

[SD69] J.R. Slagle and J.K. Dixon. Experiments with the M & N tree-searching procedure. *Journal of the ACM*, 16:189–207, April 1969.

[Sto79] G.C. Stockman. A minimax algorithm better than Alpha-Beta. *Artificial Intelligence*, 12(2):179–196, 1979.

[UvdH94] J.W.H.M. Uiterwijk and H.J. van den Herik. Speculative play in computer chess. In *Advances in Computer Chess 7*, pages 79–90, Maastricht, The Netherlands, 1994. University of Limburg.

	references	stop criteria	backup rules	node value representations	expand methods
alpha-beta	[KM75]	time	max (min)	scalar value	depth-first
M&N-Backup Procedure	[SD69]	-	function of M (N) best values	scalar value	-
Product Propagation	[Pea81]	-	product propagation	scalar value in $[0,1]$	-
Average Propagation	[NPH86]	-	average of max (min) and product propagation	scalar value in $[0,1]$	-
Min/Max approximation	[Riv87]	time	generalized p-mean	scalar value	best node (least commulative penalty)
B*	[Ber79, BM95]	separation of best and 2nd best move	max (min) of pessimistic, realistic and optimistic value, product propagation	pessimistic, realistic and optimistic value	most optimistic node for player/opponent
PB*	[Pal85]	domination of best over 2nd best move	product propagation of functions	probability distribution	most optimistic node for player/ opponent
Conspiracy Numbers	[McA88, Sch90]	only one plausible value left	according to conspiracy number definition	set of conspiracy numbers	conspirator to increase/ decrease root minimax value
Uncertainty	[Hor87]	time	max (min) or weighted max (min) according to node value	scalar value or weighted pair of scalar values	depth first
MGSS*	[RW89]	utility of expanding best node is less than time to expand it	max (min)	scalar value	best *relevant* node
BP	[BS97]	utility of further search is less than respective time cost	product propagation of functions	probability distribution	portion of nodes with best QSS
Fuzzy Numbers	[Jun94]	domination with certain risk	max (min) of pessimistic, realistic and optimistic value	pessimistic, realistic and optimistic value	depth first
Bandwidth Search	[Har74]	suboptimal goal found	max (min) value	scalar value	best node (g+h)
SSS*	[Sto79]	time	max (min)	scalar value	best first
EPS	[Ana90]	time	max (min)	scalar value	all leafs with smaller cost-benefit ratio than iteration effort limit
Singular Extensions	[ACH90]	-	max (min)	scalar value	depth first
Best-First Minimax	[KC94]	time	max (min)	scalar value	frontier node at PV
*-Min	[RB83]	time	max: max, min: with probabilities weighted sum	scalar value	depth first
ProbiMax	[Jan93]	time	min: probability weighted sum, max: max of low risk successors	scalar value	depth first
Speculative Play	[Jan93, UvdH94]	time	bonus/malus schema	scalar value	depth first
Opponent Model Search	[IUvdHH93]	time	exploiting mistakes of opponent	scalar value	depth first
M*	[CM96]	time	max (min) and exploiting mistakes of opponent	set of scalar values	depth first

	Heuristic Error	Scalar Value	Expand Next	Bad Line	Insurance	Value Backup	Stopping	Opponent	work at nodes	cutoffs possible
alpha-beta	-	-	-	-	++	-	-	-	++	+
alpha-beta enhanced	-	-	++	+	++	-	+	-	++	++
M&N-Backup Procedure						++				-
Product Propagation						+				-
Average Propagation						+			-	-
Min/Max approximation	-	-	+	++	+	++	-	-	-	-
B*	++	++	++	++	-	-	++	-	-	
PB*	++	++	++	++	-	-	++	-	- -	
Conspiracy Numbers	+	+	+	+	-	+	+	-	-	
Uncertainty	+	+	-	-	++	+	-	-	+	+
MGSS*	+	-	+	+	-	-	++	-	-	
BP	++	++	++	+	-	-	+		-	+
Fuzzy Numbers	++	+	-	+	+	-	++	-	+	+
Bandwidth Search	+	-	+	+	+	-	++	-	+	++
SSS*	-	-	++	+	++	-	+	-	++	++
EPS	+	-	++	+	+	-	-	-	-	
Singular Extensions			+	+	+				+	
Best-First Minimax	-	-	+	+	- -	-	-	-	++	
*-Min	+	-	-	-	-	+	-	+	-	-
ProbiMax						+		+	+	-
Speculative Play						+		+	+	
Opponent Model Search						+		+	+	-
M*	-	-	-	-	+	-	-	+	- -	-

Legend:	
method not working or severely handicapped because of this issue	- -
indicates that the issue is a problem	-
no issue	
method offers a fix to the specified issue	+
method offers a solution to the issue	++