Partial Order Bounding: A new Approach to Evaluation in Game Tree Search

Martin Müller

Electrotechnical Laboratory, Umezono 1-1-4, Tsukuba, 305 Japan
Current address: Department of Computing Science, University of Alberta, Edmonton, Canada T6G 2E8

Abstract

In computer game-playing, the established method for constructing an evaluation function uses a scalar value computed as a weighted sum of features. This paper advocates the use of partial order evaluation, and describes an efficient new search method called *partial order bounding (POB)*.

Previous tree search algorithms using a partial order evaluation have attempted to propagate partially ordered values through the search tree, which leads to many problems in practice, such as the complexity of backing up sets of incomparable evaluations. POB compares partially ordered values only in the leaves of a game tree, and backs up boolean values through the tree. A closely related new algorithm, *linear extension partial order bounding (LE-POB)*, uses a standard scalar alpha-beta search with values from a suitably chosen linear extension of the partial order evaluation. As an application, the effectiveness of partial order evaluation is shown in the case of modeling capturing races called *semeai* in the game of Go.

Key words: Game tree search, evaluation functions, partial order evaluation, partial order bounding, computer Go, semeai

1 Evaluating and Comparing Game Positions in Game Tree Search

Researchers building game-playing programs based on minimax tree search seek practical answers to the two main questions of *when* and *how* to evaluate game positions. The usual answer to the question "When?" is: the later the better. Although games with pathological behavior, where deeper search is detrimental, have been studied extensively [29,31], in practice deep search works very well, and yields programs that avoid most tactical and many strategical mistakes.

The standard answer to the second question, "How to evaluate?", is to use a scalar evaluation such as an integer or real number, which is computed as the weighted sum of feature values $\sum w_i v_i$. Nonlinear feature combinators such as neural networks [4,6,39,41] and a generalized linear evaluation model [9] have also gained some popularity.

Individual evaluation features are usually designed by a programmer in interaction with an expert player. Feature weights are carefully tuned using a combination of engineering and automatic parameter tuning techniques [2].

The basic fixed-depth search scheme has some well-known weaknesses, such as the horizon effect, and the large errors introduced by the static evaluation of unstable positions. Furthermore, search tends to spend a lot of time exploring obscure variations that no human would consider worth thinking about. A large amount of theoretical and practical work has addressed these weaknesses of the minimax search model, leading to refinements that mostly address the question of which positions should be evaluated. The problem of unstable positions is typically solved by not evaluating such positions at all, and performing a quiescence search instead, in the hope of reaching states that are easier to evaluate. A common-sense approach is to try to search interesting variations deeper and less interesting variations less deeply. Search extensions on one hand, and pruning methods based on the result of shallow searches such as null move pruning [5] and ProbCut [8] on the other hand, are used for this purpose.

This paper describes a different, more radical approach to the basic questions of "when" and "how" to compare evaluations. Since many pairs of positions are really incomparable from the limited knowledge and accuracy incorporated in an evaluation function, it seems that forcing a comparison by mapping all positions to the same numeric scale does more harm than good. From this point of view, using only a *partial* ordering of positions seems to be a natural approach. The main technical contribution of this paper is an efficient method for combining partial order evaluation with minimax search. *Partial order bounding (POB)* provides a way to use partial order position evaluation in conjunction with any minimax-based search engine such as alpha-beta or proof-number search.

The structure of this paper is as follows: Section 2 reviews the problem of constructing an evaluation function for game tree search, points out some of the problems with the usual approach of using a weighted sum of features, and argues in favor of using partial order evaluation instead. Section 3 contains the main contribution of the paper, a simple new method called *partial order bounding (POB)* for search in game trees using partial order evaluations. It is shown that with this method efficient search engines based on standard methods such as alpha-beta or proof number search can be combined with the

higher expressive power of an evaluation function based on a partial order of values. Subsection 3.3 introduces *linear extension partial order bounding (LEPOB)* which performs a scalar alpha-beta search using values from a linear extension of the partial order evaluation.

Section 4 studies some examples of partial order evaluation functions and systematic ways to construct new evaluations by modifying or combining existing ones. Section 5 contains a case study and experimental results for partial order evaluation in capturing races called *semeai* in the game of Go. Section 6 discusses several related previous search methods in detail, and Section 7 summarizes the approach and discusses future work. The rules of Go are described briefly in the Appendix.

2 Evaluation by a Weighted Sum of Features

In the standard model of computer game-playing, position evaluation is a two step process. The first step maps a game position to an abstract representation. A number of relevant attributes are computed and collected in a high-dimensional feature vector $\mathbf{v}$. Within such a vector, a single feature value v_i is usually simple: 0 or 1 to encode the truth value of a boolean predicate, a small integer counting pieces of a specific kind in board games, or a real number measuring influence, mobility, or the probability of achieving some (sub-)goal in the game. Given a feature vector $\mathbf{v}$, in the second step a scalar-valued evaluation is computed as the weighted sum of feature values by $eval(\mathbf{v}) = \sum w_i v_i$.

2.1 Classical Weighted Sum Evaluation Functions: Strengths and Weaknesses

To motivate the introduction of partial order evaluation, we first discuss the strengths and weaknesses of the weighted sum evaluation scheme. For a recent survey of other alternatives to standard alpha-beta minimax search and evaluation, see Junghanns [18].

2.1.1 Strengths of Weighted Sum Evaluation

The weighted sum approach to evaluation has been very successful in practice. It has proven to be a useful abstraction mechanism, with many desirable properties, such as simplicity, and ease of use in efficient minimax-based algorithms. Furthermore, in some games there is a natural mapping of positions to a numerical evaluation, for example the expected number of captured pieces

in Awari or the balance of territory in Go. In games that end in a simpler outcome such as win, loss or draw, a scalar evaluation can be interpreted as a measure of the relative chance of winning. It has been shown [34] that even if the final outcomes are restricted to only the two values win or loss, a more fine-grained evaluation at interior nodes is beneficial.

A big advantage of scalar evaluations is that they can also be used for move ordering. Good move ordering is essential for the speed of alpha-beta [19]. In iterative deepening search, using the search result of the previous iteration yields a move ordering that is often close to optimal [32].

Finally, scalar evaluation is quite robust against errors in the evaluation function as long as they are systematic. An evaluation function needs only relative consistency for the positions that are compared with each other, it does not require absolute accuracy of the values. In the case of a complex game with relatively restricted moves, it is likely that within one search, many positions are very similar to each other. If all but a few features are the same, then relative evaluation is reduced to comparing those few differing features. In such a case it is easier for an evaluation function to be relatively consistent.

2.1.2 Criticizing the Weighted-Sum Approach

Despite the great success of the weighted sum approach to evaluation, the method has quite a few weaknesses, and many of the methods discussed in Junghanns' survey [18] were designed to address such weaknesses. The main drawback of using a single number for evaluation is that information is lost. All kinds of features are weighted, added and compared, even those for which addition and comparison do not really make sense. In the following we look at a few of the ensuing problems.

2.1.2.1 Unstable Positions A simple, stable position and a highly dynamic, unstable one can end up with the same static evaluation, though their semantics are very different. Highly unstable features such as passed pawns in chess have no good static evaluation. Whichever way the weights are chosen, the evaluation will be wrong in a large percentage of cases. The traditional remedy is to not evaluate *unstable* positions and perform a quiescence search to try to reach a stable position. Stable positions are considered 'easy' to evaluate, while unstable ones are considered 'hard'. Search extensions for quiescence search are usually decided by a feature vector which may contain other features than the evaluation function.

Quiescence search cannot solve all problems of unstable features. In a case such as passed pawns in chess, such features can remain on the board for a long

time. Shogi endgames often feature a large number of pieces left hanging while both players launch an all-out attack on the opponent's king. An example in Go are large groups with unclear status, which are both difficult to attack and hard to defend.

2.1.2.2 Long Term Strategic Features One problem of the weighted sum model is that in games with a mix of short-term tactical and long-term strategic features, it is difficult to maintain control over long-term features. Human experts are admired for their ability to formulate long range plans, follow them through many tactical complications and eventually make them succeed. In a weighted sum evaluation, it is difficult to assign good weights to such features. Giving them a big weight does not work, since it causes a program to cling too strongly to such far-away goals and it will therefore lose tactically. However, using small weights does not work either since small values are very likely to get lost among the inevitable score fluctuations caused by short-lived tactics.

From a strategic point of view, the program behavior resulting from a weighted sum evaluation is not human-like. However, it is debatable whether that is a good or a bad thing. The reduction of all tactical and strategical features of a position to a single number does lead to flexible play, since all positions with the same evaluation are treated equally by a program, no matter how different they really are. It seems that following long-term strategies needs to be modeled explicitly by a method outside the scope of weighted sum evaluation.

2.1.2.3 Close-to-Terminal Positions Besides the problem of unstable evaluations, another case where the loss of information resulting from condensing the feature vector into a single number is critical are positions that can 'almost' be evaluated statically. In positions that have only a few distinguished destabilizing features, it makes sense to employ a *simplest-first* policy and initially focus search on those moves that can be resolved exactly with little work. In other words, if radical simplification leads to a sure win, there is no need to search the complicated variations. Human players often use this kind of reasoning, but standard search methods cannot take advantage of such information even if it is contained in the evaluation vector.

2.2 Goals of Partial Order Evaluation

The main goal of partial order evaluation is to make comparisons between positions only when they are meaningful. In contrast, standard scalar eval-

uations are applied and used to compare positions regardless of whether the underlying positions are comparable. By refraining from judgment in doubtful cases, partial order evaluation aims at increasing the confidence in the validity of *better* and *worse* judgments derived by search.

One major target are games with long-term strategic features, which are difficult to model by a weighted sum evaluation. Another suitable class of games are those with partially decomposable positions, such as middle game positions in Go or Amazons. Such positions can often be split into one main part and several independent small parts. The values of such small parts often influence a search of the main part in subtle ways that cannot be summarized by a single number, but can be expressed in terms of a partial order bound, which determines which of the possible search outcomes are overall wins. An example of this kind of search in Go is studied in Section 5.

2.3 Problems of Using a Partial Order Evaluation in Minimax Tree Search

When using partially ordered evaluations, the result of a search cannot be just a single value from the partially ordered set. (We will sometimes use the equivalent but shorter term poset for partially ordered set.) Computing minima and maxima of such values is an ill-defined problem. A totally ordered set such as the integers or reals is closed under the application of the operators *min* and *max*: if $x_1, \ldots, x_n$ are values from a totally ordered set T, then both $\min(x_1, \ldots, x_n)$ and $\max(x_1, \ldots, x_n)$ are again elements of T, with the properties $\min(x_1, \ldots, x_n) \leq x_i$ and $\max(x_1, \ldots, x_n) \geq x_i$ for all $1 \leq i \leq n$. Furthermore, the minimum and the maximum coincide with one of these values. For values from a partially ordered set, it is no longer possible to define a *min* or *max* operator with these properties.

Several different approaches to overcome this fundamental problem have been tried. In some special cases such as probability distributions, it is possible to define meaningful *min* and *max* operators with similar but more restricted properties. Another solution which works if the poset is a lattice is to define the least upper (greatest lower) bound on a set of incomparable values as the maximum (minimum) of these values. However, this approach obviously loses information, and propagating such bounds by a tree backup makes the approximation weaker and weaker. Another solution [12] is keeping track of a set of nondominated sets of outcomes. However, this approach can lead to very high complexity in the case when there are many incomparable values. A viable search procedure seems to require extra assumptions, such as totally ordered private preferences of the players. The method of Dasgupta, Chakrabarti and DeSarkar [12], which uses such an approach, is discussed in more detail in Section 6.1.

Partial order bounding retains the first step of constructing an evaluation function from the scalar case, namely the mapping of a position to a high-dimensional feature vector. Such a vector can in principle contain all details of a position [9], but usually it involves some amount of preprocessing that leads to an abstract representation of a game position. In complex games such as chess and Go, the time used for computing good features dominates the overall processing time.

The second construction step for evaluations is different from the scalar case: instead of reducing the whole vector to a single scalar value, a partial order is defined over the feature vectors, and this partial order is used for comparing positions. In fact, the partial order bounding approach does not really depend on the intermediate construction of a feature vector, it works for any poset. A poset can be represented in other ways, such as an implicit function representation with lazy evaluation, or in terms of other, more dynamic data structures that are not easily mapped to a vector of fixed features [38].

3 Partial Order Bounding: A New Approach to Partial Order Search

As argued in the preceding section, partial order evaluations are potentially useful since they are more expressive than scalar evaluations. One reason for the fact that they have not been popular in game tree search so far seems to be that previous partial order search methods have tried to back up partially ordered values through the tree. Depending on the method used, this leads either to potentially large sets of incomparable options, or to a loss of information, or both. In addition, some methods are applicable only for restricted types or specific representations of partial orders. See Section 6 for a detailed discussion of previous work on search using partial order evaluations. *Partial order bounding (POB)*, as a new approach to partial order evaluation in search, avoids such problems by separating the comparison of partially ordered evaluations from the tree backup.

3.1 Null Window Search

Partial order bounding is based on the idea of null window searches, which have already become very popular in search using scalar evaluation through methods such as Pearl's SCOUT [31] and Plaat's MTD(f) [32]. Rather than directly computing minimax values, null window search is used to efficiently

compute bounds on the game value. These searches are used to establish that other moves are not better than the best known move in SCOUT, and to discover the minimax value by a series of null window searches in MTD(f). An important implementation detail of null window search is the use of *fail-soft alpha-beta* [13], which can return a value outside the alpha-beta window in the case that the search fails high or low. This method improves performance in the case of re-search with a different window.

To summarize, the goal of null window search is to establish an inequality between a given fixed bound and the unknown evaluation of a node. POB extends this idea to the case of partial order evaluation.

3.2 Definition of Partial Order Bounding

This section formally defines partial order bounding. For completeness, the required concepts from the theory of partially ordered sets are stated. Then POB is defined as a method for using a poset P as an evaluation for a minimax-based search method M.

3.2.1 Basic Definitions: Posets, Comparison Relations and Antichains

The definitions follow standard conventions. For more information on posets see textbooks such as [35,42].

Definition 1 *A* partially ordered set *or* poset P *is a set (also called P) together with a reflexive, antisymmetric and transitive binary relation $\leq$. The dual relation $\geq$ is defined by $x \geq y \iff y \leq x$. Two elements x and y of P are called* comparable *if $x \leq y$ or $y \leq x$, otherwise x and y are called* incomparable. *The relation $x < y$ is defined by $x < y \iff x \leq y \wedge x \neq y$, and $x > y$ is equivalent to $y < x$.*

A poset is graphically represented by drawing its *Hasse diagram*, which is a graph of the transitive reduction of the poset. A downward leading edge is drawn between two elements x and y if and only if $x > y$ and there is no element z such that $x > z > y$.

Example 2 *Figure 1 shows a poset $P = \{A, B, \ldots, K\}$. In P, relations between elements such as $J \geq F$ and $K \geq C$ hold, but for example J and G are incomparable elements. This can be seen from the diagram since there is neither an always downward leading path from J to G, nor one from G to J.*

Definition 3 *A nonempty subset A of P is called an* antichain *if and only if any two distinct elements of A are incomparable.*

Example 4 *Figure 2 shows an antichain in P consisting of the two elements E and G. Another example of an antichain would be $\{B, C, G\}$. Every one-element subset is also an antichain.*

3.2.2 Defining a Partial Order Bound for Search

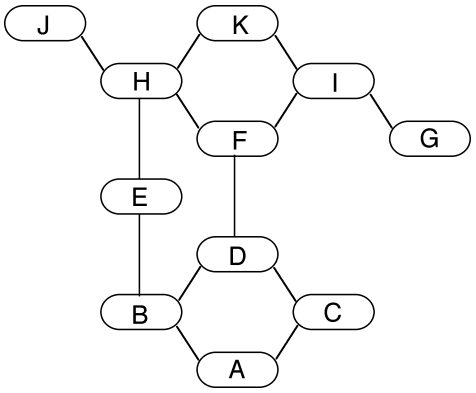


Fig. 1. Example of a partial order

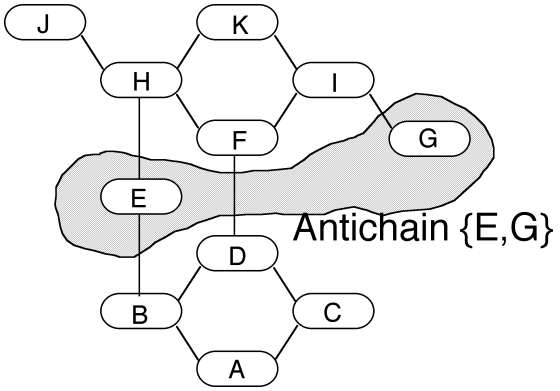


Fig. 2. Example of an antichain

In the case of a partially ordered set P, a bound $B \subseteq P$ can be given by an antichain in P that describes the minimal acceptable outcomes a player wants to achieve.

Definition 5 *Given a partially ordered set P and an antichain $B \subseteq P$, the* success set *of B in P is defined by $S(B) = \{x \in P \mid \exists b \in B : x \geq b\}$. The* failure set *of B in P is the complement of the success set: $F(B) = P - S(B)$.*

The success set contains all values that are "good enough" with respect to the given bound, while the failure set contains the remaining insufficient values. Search is used to decide whether the first player can achieve a result $x \in S(B)$, or whether the opponent can prevent this from happening.

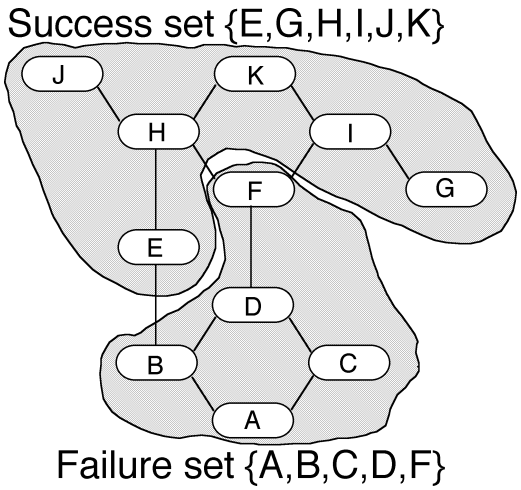


Fig. 3. Success set and failure set for the bound $\{E, G\}$

Example 6 *Figure 3 shows the partition of P into success and failure sets by the antichain bound $\{E, G\}$ of Figure 2.*

3.2.3 Partial Order Bounding

Definition 7 *Given a poset P, an antichain $B \subseteq P$, and a minimax tree search method M that uses a boolean or other scalar evaluation, the search method of* partial order bounding *$POB(M, P, B)$ is defined as follows:*

- *Replace the leaf evaluation of M as follows: evaluate leaf nodes using the partial order evaluation, resulting in a value $v \in P$.*
- *Classify the outcome as* success *if $v \in S(B)$ and as* failure *otherwise.*
- *Back up the value* success *or* failure *through the tree by method M, using the two totally ordered values success and failure with the ordering success > failure.*

Example 8 *The example tree shown in Figure 4 is taken from Fig. 2(a) of [12]. Leaves have been evaluated by pairs of integers. As the partial ordering, a vector dominance order (see Section 4.2.2) is used. In the diagram, as usual squares represent MAX nodes and circles MIN nodes. For illustration, we consider the following two out of the large number of possible bounds: $B_1 = \{(5, 7), (10, 3)\}$ and $B_2 = \{(5, 8), (6, 4)\}$. In this example, MAX can obtain the bound B_1 but fails to obtain the bound B_2. Leaf evaluations and backed up values are shown in the figure, with a plus sign representing success and a minus sign representing failure for MAX. Note that MAX would not succeed by selecting a single-element subset of B_1 in this example.*

The partial order bounding approach separates the comparison of partially

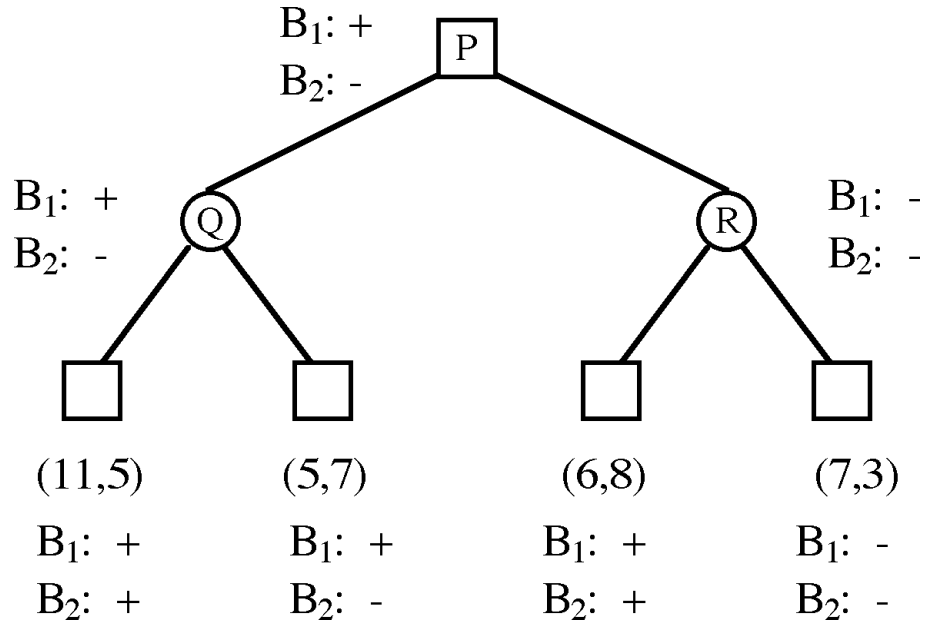


Fig. 4. Example of search using POB

ordered values from the value backup procedure in the tree or game graph. This simplifies the computation compared with previous approaches, since there are no sets of incomparable values that must be computed, stored, and backed up. Still, the full power of partial order evaluation is available for making decisions during the search, or rather for refraining from making doubtful decisions.

A big advantage of partial order bounding is that it can be combined with any minimax-based search method, such as alpha-beta or proof-number search [1]. The new evaluation method can be added to a sophisticated state of the art search engine with minimal effort. All search enhancements of such an engine are automatically available to the new algorithm. Of course it must be checked how well they work with a partial order evaluation in each case.

The following are the main differences between POB and previous partial order search methods such as [12]:

- Comparison of partial order evaluations occurs only at leaf nodes. The evaluation is compared to a prespecified bound B, resulting in one of two outcomes: *success* or *failure*. There is no backup of (sets of) partial order evaluations through the search tree.
- POB requires an input parameter: a bound B in the form of an antichain in P which separates the acceptable outcomes, called the *success set*, from the unacceptable ones, the *failure set*.
- POB allows any representation of a partial order, not just vectors with a partial order defined by vector dominance.

The selection of a suitable bound B depends on both the application domain and the specific problem instance. In applications with exact partial order

specifications of possible solutions, such as some of the capturing races to be described in Section 5, a single bound is sufficient to completely specify a search problem. With a heuristic partial order evaluation function, an iterative process can be used to test different bounds. In this respect, POB is comparable to Plaat's MTD(f), which uses a series of null window calls to find a scalar minimax value. The big difference is that with partial order evaluation there is no single best solution. Many different incomparable bounds may be achievable for a given problem.

3.2.4 Combining Partial Order Bounding with Proof-Number Search Methods

Partial order bounding is a proof procedure: it backs up a boolean result. With a small modification, POB can be used with the family of search methods specialized for finding proofs and disproofs, such as proof-number search [1] and its refinements. These search methods use not only the two values *success* and *failure* but also a third value *unknown*. POB can be adapted accordingly to use two bounds that separate the success and failure sets from a third *unknown set* in the middle. Search repeatedly expands a most-proving node with value in the unknown set, until a proof or disproof of the root within the current success and failure bounds is obtained. Again, after a proof or disproof is obtained, a new search with different bounds can be started. As in [1], a heuristic initialization of proof and disproof numbers derived from the partial order evaluation of leaf nodes can be used to direct the search towards promising nodes.

3.3 LE-POB: Search Using Linear Extensions of a Partial Order

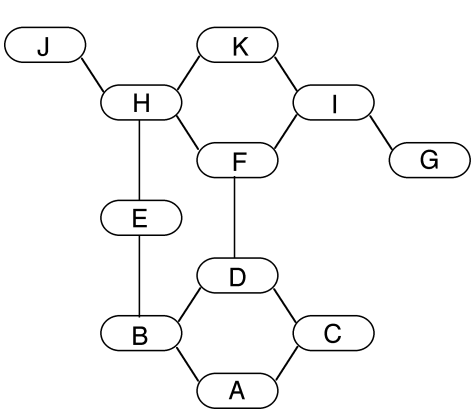
Linear extension partial order bounding (LE-POB) is another, closely related new search method that embeds POB within a standard integer-valued minimax search. Given a poset P and a bound B, LE-POB uses a suitable linear extension $E(P)$ as its evaluation for minimax search.

Definition 9 *Given an n-element poset P, a* linear extension*, or* extension to a total order*, of P is an order-preserving bijection between P and a set of n consecutive integers.*

Remark: the usual definition uses the fixed set of integers $\{1, \ldots, n\}$, but the definition given above is more convenient for the purpose of defining LE-POB.

Given a poset P and an antichain $B \subseteq P$, a linear extension $E(P)$ of P is useful if it can separate the success set from the failure set. This is guaranteed if the image of B, the set $\{E(b) | b \in B\}$, consists of consecutive integers.

Definition 10 *$E(P)$ is* dense with respect to *B if and only if all elements of B are mapped to consecutive integers. This is equivalent to the following condition: For all $x \in P$, $\min_{b \in B}\{E(b)\} \leq E(x) \leq \max_{b \in B}\{E(b)\} \Rightarrow x \in B$.*



P	E(P)
K	11
J	10
I	9
H	8
G	7
F	6
E	5
D	4
C	3
B	2
A	1

Fig. 5. Poset P and linear extension $E(P)$

Example 11 *Figure 5 shows the poset P and one of its linear extensions $E(P)$. In this example, $E(P)$ is dense with respect to the bound $B_1 = \{F, G\}$, which is mapped to successive numbers $\{6, 7\}$. However, $E(P)$ is not dense with respect to $B_2 = \{E, G\}$, since its image $\{5, 7\}$ has a gap at 6.*

3.3.1 The LE-POB Algorithm

Definition 12 *Given a poset P, an antichain $B \subseteq P$, a linear extension $E(P)$ that is dense with respect to B, and a fail-soft (see Section 3.1) alpha-beta based minimax tree search method M for integer evaluations,* linear extension partial order bounding *$LE - POB(M, P, B, E(P))$ is defined as follows:*

Compute $lb = \min\{E(b_i)|b_i \in B\}$, $ub = \max\{E(b_i)|b_i \in B\}$. Then perform a search using M with initial window $(lb - 1, ub + 1)$ and integer-valued leaf evaluation function $E(P)$.

The outcome of a LE-POB search is related to the outcome of POB searches in the following way:

(1) If the search does not fail low, then the initial bound B is achievable. Furthermore, if the return value v is larger than lb it proves the smaller success set $S' = \{x \in P|E(x) \geq v\}$. Therefore the stronger bound consisting of the minimal elements of S' is also achievable.
(2) If the LE-POB search fails low, the initial bound is not achievable. A fail-soft return value $v < lb$ disproves B and all other bounds for which all elements are mapped to values above v.

It is possible to use null window search with $(lb-1, lb)$ instead of $(lb-1, ub+1)$, but in this case, less information about possible smaller success sets is computed.

3.3.2 Constructing a Suitable Linear Extension

Given a bound B, finding a suitable linear extension that is dense with respect to B is easy:

(1) Determine $S(B)$ and $F(B)$ as in Definition 5. Partition P into the three sets $S(B)-B$, B and $F(B)$ with the induced partial order on the subsets.
(2) Run a standard algorithm for generating a linear extension such as [36,43] on each subset separately. The computation for B can be skipped since the ordering provided by any permutation of the elements of an antichain B is a linear extension.
(3) Concatenate the results, shifting the intervals if necessary so that they are adjacent with $F(B)$ lowest, B in the middle, and $S(B)-B$ highest.

3.3.3 Searching Several Bounds Simultaneously

It is possible to use LE-POB with several bounds $B_1, \ldots, B_n$ simultaneously, as long as the chosen linear extension is dense with respect to all of them, and the alpha-beta window contains all the images. The different bounds can for example represent a number of qualitative judgments, such as "about even", "a little better", "a little worse", "much better", "sure win", "disaster", etc.

3.4 Search Control with Partial Order Evaluation

To select bounds for POB, two cases must be distinguished: if the partial order evaluation is exact, the bound needed to accomplish a specific goal is known precisely. See Section 5 for the example of semeai in Go. In contrast, in a heuristic partial order evaluation, there is no such predetermined bound.

With partial order evaluation, search control becomes an iterative process involving a series of calls with progressively refined bounds. In this respect it is similar to the MT search framework of Plaat *et al.* [32]. However, with partial order evaluation the situation is less clear-cut than with scalar minimax search, where search converges to a single minimax value.

Specifying a partial order bound can be compared to the way a human player analyses a game position: he or she typically starts with a static position analysis, then formulates possible goals and plans, and searches selectively

within the framework of these goals and plans, all the while avoiding clearly bad positions and keeping an eye open for unexpected chances. If the initial goals turn out to be unrealistic, search is restarted with lowered aspirations. On the other hand, if the initial goals can be achieved, the player will strive for a little extra.

In a game with long-term strategic features, many components of a bound will remain the same from move to move. A natural choice of bound elements are therefore those from the previous move, with refinements reflecting the progress of the game since then.

How does POB address the problems of the weighted sum approach discussed in Section 2.1.2? How can it handle *unstable positions*, *long term strategies* and *close-to-terminal positions*? Unstable positions can be handled flexibly: In good positions, the success set can be restricted to eliminate risks. On the contrary, in bad-looking positions, bounds should exclude only the simple inferior positions, in order to direct play towards greater complications. In a similar manner, long term strategies and close-to-terminal positions can be dealt with. Evaluations where a desirable long-term feature is missing can be excluded from the success set, and a simplest-first search can be implemented by choosing small success (or failure) sets containing only simple, clear position evaluations.

4 Examples of Partial Order Evaluation Functions

Partially ordered sets offer a rich variety of possible structures, and their theory has been thoroughly researched. However, the use of partial order evaluation in game tree search is not yet well understood. The aim of this section is to start research in that direction by giving some examples of possible poset evaluations.

4.1 Efficiency of Partial Order Evaluation

Because scalars such as integers and reals are directly supported by current hardware, it seems appropriate to start the discussion with the issue of efficiency of posets. Although results on many special cases are known, the complexity of some basic algorithms for posets still remains unresolved [38]. For large posets, tradeoffs between storage and computation time must be considered.

In POB, only a single operation on posets is needed: the comparison of one

value with a fixed antichain bound. If the bound has only a few elements, this comparison can be conveniently expressed in terms of the primitive operation of comparing two values. If a bound contains a large or infinite number of elements, an efficient implicit test can still exist.

Example 13 *Let a partial order evaluation be given by intervals over the reals, with a comparison operator* $[x_1, y_1] \geq [x_2, y_2] \Leftrightarrow x_1 \geq x_2 \ \wedge \ y_1 \geq y_2$. *Then the set* $B = \{ [2-\delta, 2+\delta] \mid 0 < \delta \leq 1 \}$ *of intervals with center 2 and lower bound at least 1 is an antichain bound of infinite cardinality. However, it is easy to compare any given interval* $[x, y]$ *against* B *by using the above implicit definition of the set* B.

4.1.1 Relative Cost of Scalar and Partial Order Evaluation

At first sight, it seems unlikely that a partial order evaluation can be as efficient as integers or reals with their direct hardware support. However, for a typical complex game, the cost of comparing evaluations will be negligible compared to the cost of computing feature values. For this reason, many high performance game-playing programs already employ lazy evaluation [15]. At first, exact scores such as mate or repetition draws are detected. Next, only a few important features with high weight are computed, and the contribution of the remaining features is estimated by bounds. If the bounds suffice to cause a cutoff in the search, the time for computing the remaining features is saved. The same idea of lazy evaluation can be used with posets. The fine-grained comparison of partial order evaluations should allow at least comparable savings through lazy evaluation, and features which are not required for comparing a position with the current bound need not be computed at all.

4.2 Partial Order Evaluations that are Similar to Scalar Evaluation

A number of partial orders can be constructed in a straightforward manner from a given total order. We discuss the following cases:

- Exceptions: specific pairs of incomparable values
- Low-dimensional posets that can be represented by a small number of scalars
- Partial orders representing uncertainty about the true value of a scalar, such as intervals, triples, and probability distributions

4.2.1 Exceptions

Exceptions from a total ordering remove comparability for specific pairs of values. For example, exceptions can be used to make values such as *sure-draw*

and *balanced-active-position* incomparable.

4.2.2 Small Scalar Vectors as Low-Dimensional Posets

The standard approach to evaluation in multiobjective search [37,14,12,11] uses a m-dimensional vector of scalar values from domains $Y_1 \ldots Y_m$. A partial order on such vectors is defined by the following vector dominance relation:

$$\mathbf{y} \leq \mathbf{y}' \;\Leftrightarrow\; y_i \leq y_i' \;\; \forall i \in 1, \ldots, m.$$

In partial order terminology, the poset defined by the vector dominance relation is the direct (or cartesian) product of the totally ordered domains, $Y_1 \otimes \ldots \otimes Y_m$.

In general, each poset P can be represented as the direct product of $dim(P)$ total orders, where $dim(P)$ is the dimension of P [30]. However, it might be intractable to find such a representation for a given poset. Practical algorithms exist for posets of "modest size" [46], but no efficient general method is known.

4.2.3 Uncertain Scalar Values

Uncertainty about the true value of a scalar can be represented by a partial order. Intervals, "fat values" such as triples containing a lower bound, a realistic value and an upper bound [5], and probability distributions [3] are prominent examples. Different kinds of partial orders can be defined over such structures. One natural interpretation of an interval is as a pair of upper and lower bounds on the unknown true value. The corresponding partial order is a dominance order as in Section 4.2.2. Intervals can also be interpreted as representing different kinds of probability distributions, such as triangular, constant in the interval, as the points $\mu \pm k\sigma$ of a normal distribution over the reals, and others. In the safest but weakest ordering, two values are comparable only if their ranges are disjoint. An example of a stronger ordering of probability distributions is *stochastic dominance* [33].

Antichain bounds on intervals can be constructed for example by fixing the lower bound, the upper bound, or some linear combination of the two such as the center of the interval as in Example 13. In addition, a desired minimum or maximum interval width can be specified. Bounds defined implicitly in terms of a comparison function can contain a large or even infinite number of elements, but they are typically very easy to compare with a given value.

Textbooks on partial orders such as [35] describe a number of methods for constructing new posets from existing ones. All these techniques apply to the problem of constructing a partial order evaluation. Some examples are generating direct products of posets, with vector dominance as in Section 4.2.2 as a special case, as well as adding, deleting and merging values in a poset.

4.3.1 Adding New Values to an Evaluation

Completing a given evaluation An evaluation might be incomplete, since it might not be applicable to all possible positions. In this case a special value *unknown* can be added to the set of possible evaluations, which is incomparable with all other values, except for sure wins and losses.

Adding bounds on values to a poset For each value $x \in P$, bound values can be added as new elements to the set. Let x^+ represent such an unspecified value at least as good as x, $x^+ \geq x$, and let x^- stand for a value worse-or-equal to x, $x^- \leq x$. A partial order over the extended set $P \cup \{x^+, x^-\}$ can be defined as the transitive closure of the partial order on P plus the new relations $x^+ \geq x$ and $x \geq x^-$.

4.4 Developing Partial Order Evaluation Functions

There are several established ways to develop scalar evaluation functions, such as knowledge engineering with the help of human experts and textbooks, analysis of games played by a program, and automatic weight tuning using training samples from collections of test problems, master games, or minimax search scores. Some of these techniques can be adapted to construct partial order evaluation functions.

4.4.1 Knowledge Transfer from Human Experts

In knowledge engineering involving input from human experts, the step of expressing expert knowledge in terms of numeric evaluations has always been problematic. When developing a partial order evaluation, this step is no longer necessary. Relative evaluations such as "better", "worse", or "unclear", which correspond more closely to an expert's judgment, can be directly implemented in a partial order.

4.4.2 Learning Partial Order Evaluations

Constructing a partial order evaluation for a given set of elements can be viewed as a learning problem. The task is to learn whether $x \geq y$ is true or false for each pair of elements. There is a vast arsenal of classifier learning methods that are applicable to this problem. See [22] for an overview of methods for learning classifiers. The main problem is to assure that the learned classifier is consistent, so that it really represents a partial order.

Example 14 *Cohen, Schapire and Singer discuss "learning how to order things" [10]. In their work, a relative ordering between items is constructed by means of a graph with weighted directed arcs. A weight represents the degree of confidence that the first of two linked items is preferable to the second. Such a directed graph does not necessarily represent a poset, since the orientation of arcs might be inconsistent. However, if the graph is consistent, it provides a parameterized representation of a family of posets. For each minimum weight level w_{min}, the transitive closure of the subgraph consisting of all edges with weight $w \geq w_{min}$ is a partial order. In [10], the method was applied to the problem of ranking web pages according to their relevance, but it also seems to be directly applicable to the construction of evaluation functions in games.*

4.4.3 Converting an Existing Weighted Sum Evaluation

A simple proposal for how to convert an existing scalar weighted sum evaluation to a partial order is given by the following iterative method:

(1) Identify feature values and value combinations for which the value comparison is unstable and often overturned by deeper searches.
(2) Make unstable comparison instances incomparable in the partial order.
(3) Use the resulting new evaluation for tests, iterate steps (1) - (3).
(4) The remaining cases with high error rate indicate situations where new evaluation features must be found.

5 Partial Order Evaluation of Capturing Races in the Game of Go

Fig. 6. Simple semeai with 3 against 2 liberties

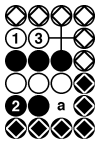


Fig. 7. Black wins the semeai

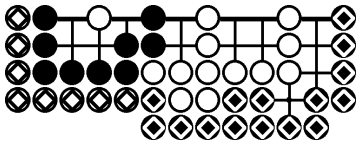


Fig. 8. Semeai with big eyes and many liberties

This section contains a case study in which partial order evaluation is used to model the state of capturing races or *semeai* in the game of Go. See the Appendix for a brief explanation of the rules of Go, and definitions of the terms *block*, *liberty*, *capture* and *eye*. For details on Go rules, see [7] or an online source such as *www.usgo.org*.

A *semeai* is a race to capture. Both players try to be the first to capture the opponent's stones, in order to save their own stones. Figures 6 and 8 show two examples of semeai situations. In the figures, by convention the stones on the outside marked by a diamond are assumed to be safe by having enough liberties outside the area shown. Such stones form the boundary of a problem and are not considered as possible targets for capture. In a semeai problem, players try to secure their own unmarked blocks of stones and try to capture the opponent's unmarked stones.

Example 15 *Figure 6 shows a black block consisting of three stones that has three liberties marked 'a', and a white three stone block that has two liberties marked 'b'. The surrounding large blocks are considered to be safe.*

Black wins this race even if White plays first. Figure 7 shows one possible continuation. After three moves, both players have only a single liberty left, but it is Black's turn, and the move at 'a' captures the three white stones and wins the fight. Removing the white stones gives the adjacent black block three new liberties and space that guarantees a connection to the safe black stones below.

In semeai, an evaluation based on two incomparable features called *liberty count* and *eye status* can be used to model the strength of each player's position. We will show that this partial order evaluation is more powerful than a single scalar value.

Skill in analyzing capturing races is an important component of Go playing strength for both humans and computer programs. Misjudging or misplaying a

semeai usually costs a large number of points and often decides a game. Techniques for analyzing semeai have been developed centuries ago, and passed on among Go players. Informal descriptions of semeai-related techniques, aimed at human players, are available in Go books and articles [17,21]. For a formal description and classification of semeai, and more discussion of game-specific details, see [25].

In the cases studied here, there are only two possible outcomes: either one player can win and take all the points in the local region, or with best play both players end up coexisting in a so-called *seki*. In general, semeai can be arbitrarily complex and involve many blocks of stones. The outcome of such semeai can also be complex, with each player winning some part of the semeai and losing the rest of the area. Semeai can also involve local move repetition called *ko*. In such cases, the situation cannot be properly analyzed by minimax-based searches alone. Techniques from combinatorial game theory are better suited to evaluate such situations [26,27].

Two types of semeai problems are considered here. The problem instances have been carefully selected to be challenging as semeai problems, yet avoid many other difficulties of Go such as *ko* fights, ill-defined boundaries and other non-local dependencies which typically appear in textbook problems designed for human players.

The first type of problem can be solved statically by a partial order evaluation. The second, more complex type of capturing races is solved by a search using partial order bounding. In this type of semeai, the configuration of one player's stones is fixed, and the question becomes whether the second player can obtain a sufficiently good combination of *liberty count* and *eye status*. Figure 14 in Section 5.4.1 shows some examples of this type of problem, which occurs frequently in practice when one player does not have enough room to make a living group but can counterattack against the opponent's surrounding stones. In the examples, the exact winning condition for the second player can be expressed by a partial order bound, and partial order bounding can be used to search for a winning move. If no winning move exists, a different bound can be used to determine whether coexistence in *seki* is possible.

5.1 Related Work on Analysis of Semeai

Techniques for analyzing and playing semeai have been known among Go players for many centuries. See Hunter [17] for a detailed tutorial of semeai-related techniques. The first concise description of the classical *semeai formula* in a western language was given by Lenz [21]. Both of these earlier works were written for a human audience and are not sufficiently precise and complete for

a computer implementation.

Landman [20] proposes a table-based algorithm for simple semeai involving one block of each player. Both blocks are allowed to have either no eyes or one single point eye. In similar fashion, Nakamura [28] analyses some cases of semeai, including one instance where one player has two blocks. Neither Landman's nor Nakamura's analysis is applicable in semeai with so-called *big eyes*, such as the example in Figure 8. Furthermore, the correctness of their basic semeai analysis methods depends on a rather large number of implicit assumptions.

In [25] we developed the first formal description and classification of semeai, including a number of technical conditions that must be fulfilled to ensure that a semeai analysis is valid. The two main conditions restrict the structure of empty areas and eyes to so-called *plain liberty regions* and *plain eyes*. The paper [25] contains a more in-depth discussion of these important game-specific details than is possible here.

5.2 Partial Order Semeai Evaluation

In semeai, having many liberties is always a good thing. However, having an *eye* is a second important factor that determines the relative strength of blocks in a semeai. The contribution of an eye to one player's strength in a semeai is measured by the *eye status*, which is related to the size of the eye. Blocks with no eye are assigned an eye status of 0. All blocks with so-called *small eyes* of size 1, 2 or 3 behave the same way in a semeai and are therefore assigned an eye status of 1. Larger eyes of size 4 to 7 all behave differently, and are assigned their respective size as an eye status. There are no plain eyes larger than size 7. The possible values for eye status are therefore {0,1,4,5,6,7}. Having the better eye status can be a huge advantage in a semeai in some cases, but it is less important in other cases. For more details on semeai, see [17,25].

Example 16 *Figure 8 shows a semeai where each of the players has surrounded a small area. On the left, Black has surrounded four empty points and a single white stone. On the right, White has surrounded a* 2×2 *area. In this example Black has an eye status of 5 and White has an eye status of 4.*

The number of liberties and the eye status together determine the relative strength of a block in a semeai. However, they cannot be reduced to a single scalar evaluation. Figure 9 shows the Hasse diagram of the partial order of (liberty, eye status) pairs. To save space, the lines between vertically aligned items are not drawn in the figure. For example, $(14, 0) > (13, 0)$ and $(6, 5) > (8, 1)$, but $(6, 5)$ and $(10, 1)$ are incomparable.

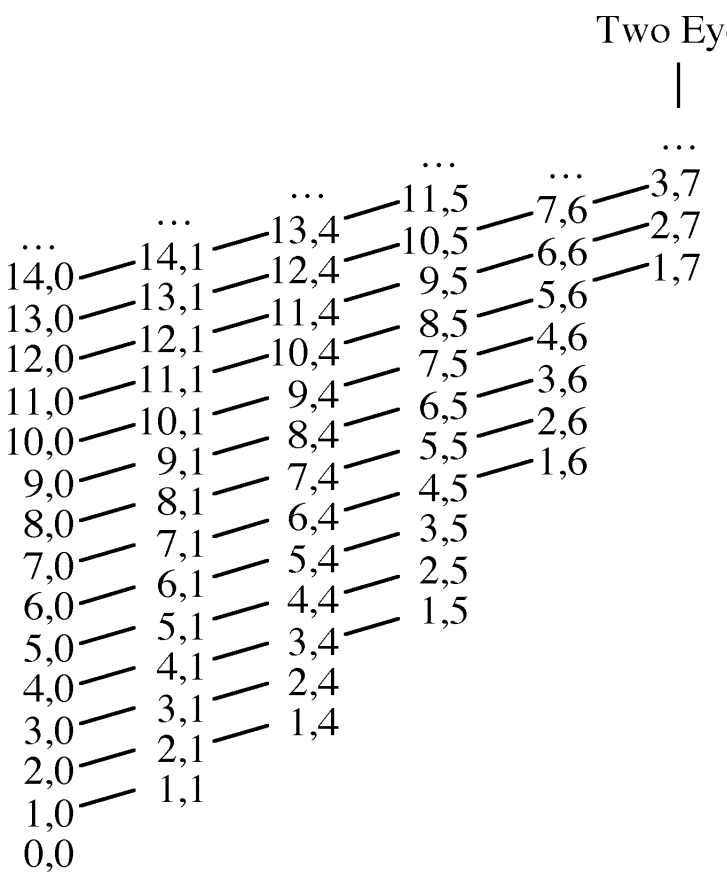


Fig. 9. Partial order evaluation of (liberty, eye status) pairs

5.3 Using POB to Solve Semeai Problems

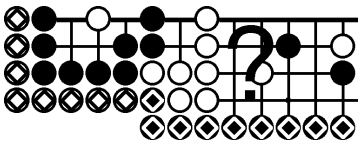


Fig. 10. A semeai search problem

Figure 10 shows a semeai at an earlier stage. The left side is the same as in Figure 8, and contains a black block which has four liberties inside an eye with an eye status of 5. Furthermore, there are two liberties shared between the black and the large white block to its right. The eye status and liberty count of the white block are not yet settled, indicated by the big question mark in the open-ended area to the right of the white block. Search in this area can be used to find out whether White can survive. By using game-specific semeai knowledge [25], the minimal strength that the white stones must achieve in a position with White to play can be specified by the partial order bound $B = \{(9,0),(2,5)\}$. Figure 11 shows this bound and the corresponding success set. A search can use partial order bounding with bound B to greatly constrain

the search space.

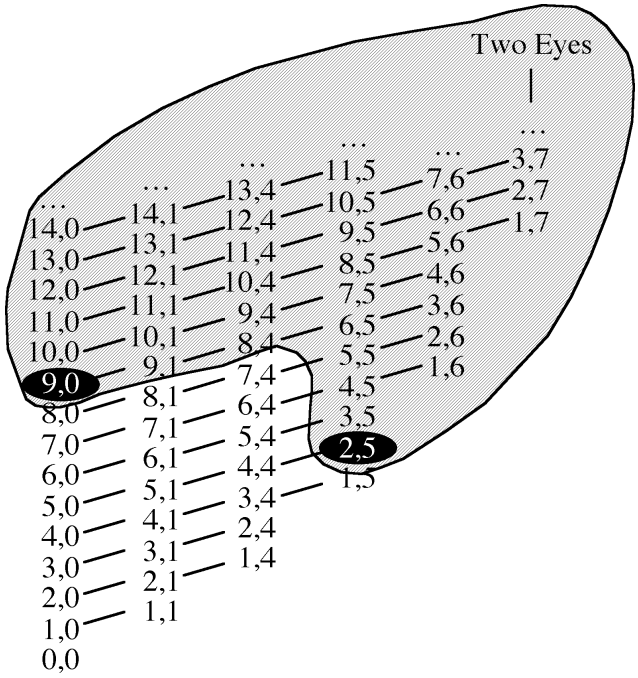


Fig. 11. Partial order bound and success set for semeai problem

5.4 Experimental Results

We tested the idea of partial order evaluation and search using partial order bounding in three experiments. The first two experiments compare the semeai problem-solving capability of the new method against the technique proposed by Landman [20]. The first experiment uses a set of basic semeai examples to illustrate the power of a static evaluation function based on the partially ordered features of liberty count and eye status. The second experiment incorporates *POB* into a standard search framework in order to solve more complex semeai instances, which cannot be solved statically by our techniques. *POB* is compared to a search using Landman's evaluation. The third experiment is a practical test pitting our Go program *Explorer* against two of the leading commercial programs in full-board situations containing many semeai problems.

5.4.1 Experiments 1 and 2: Semeai Problem-Solving

Three methods labeled *L1, L3* and *POB* are compared in these tests: *L1* uses Landman's original semeai evaluation which recognizes semeai with single-point eyes. *L3* uses an improved Landman-style evaluation for all small plain eyes up to a size of three points, which is the limit to where his method is

applicable. *POB* uses the static semeai evaluation based on the partial ordering of (liberty, eye status) pairs as in Figure 9. The implementation of all three methods is identical in all other aspects, including search enhancements such as iterative deepening and transposition tables. Most significantly, all implementations share the same semeai move generator, which encodes a large amount of Go-specific knowledge about which moves are equivalent, and which moves are dominated by other moves in a semeai situation. For all three tested methods, the current implementation runs at a speed of about 1000 semeai evaluations per second on a Macintosh G4/450. A fully incremental implementation would probably be at least one order of magnitude faster. For more Go-specific details on move generation and search, see [25].

To obtain a better measure of the relation between problem complexity and solution effort, a series of intermediate search problems is obtained from each original problem by considering the simplified semeai problem after each step of one specific correct solution sequence. Problems starting near the end of the solution sequence are typically much easier to solve than the original problem. Each position is solved twice, once with Black moving first and once with White moving first.

The node counts for solving simplified problems often show a strong odd/even fluctuation. This is caused by the fact that the situation is unsettled in the starting position and after an even number of moves, whereas after an odd number of moves the situation favors the previous player. Therefore, odd and even ply starting positions are listed in separate columns in the tables.

For each position the table contains two entries: the number of nodes searched to solve the semeai problem if black moves first and if white moves first. For example, in the results for problem B in Table 2, the entry *13/6* in row *W9*, column *L3* is read as follows: consider the semeai problem after move White 9 (*W9*) in the sample solution of problem B given in Figure 13. Using method *L3*, this problem takes 13 nodes if Black moves first and 6 nodes if White goes first (*13/6*). By comparison, the same problem takes 106/73 nodes with method *L1* and is solved statically by the partial order evaluation.

Figure 12 shows three semeai problems A, B and C. In problems A and B, it is White's turn, whereas in Problem C it is Black's turn to play. Figure 13 shows sample solution sequences. and Tables 1, 2 and 3 the number of nodes taken by the three solution methods.

These rather basic semeai clearly demonstrate the advantage of being able to evaluate large eyes. Problem A, with a single eye of size 2, can be solved statically by both *POB* and *L3*. *L1* already requires search.

Problems B and C contain larger eyes of size 4 and 5, which are reduced to small eyes only a few moves before the end. Only partial order evaluation

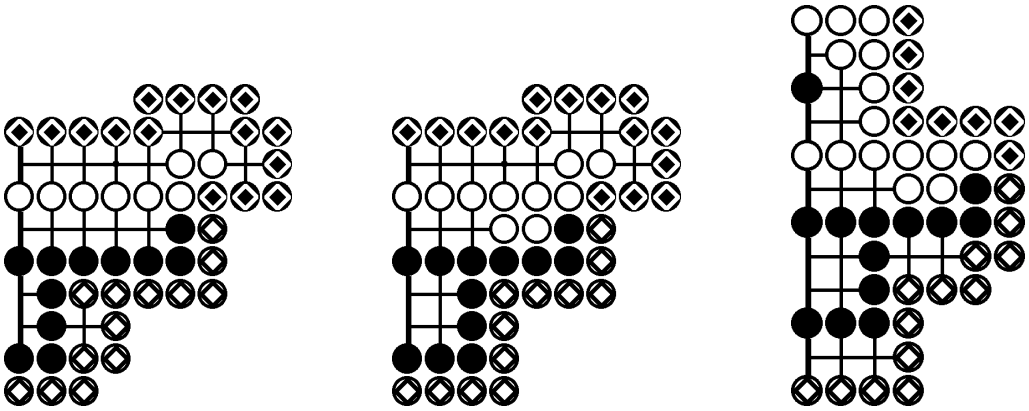


Fig. 12. Semeai test problems A, B and C

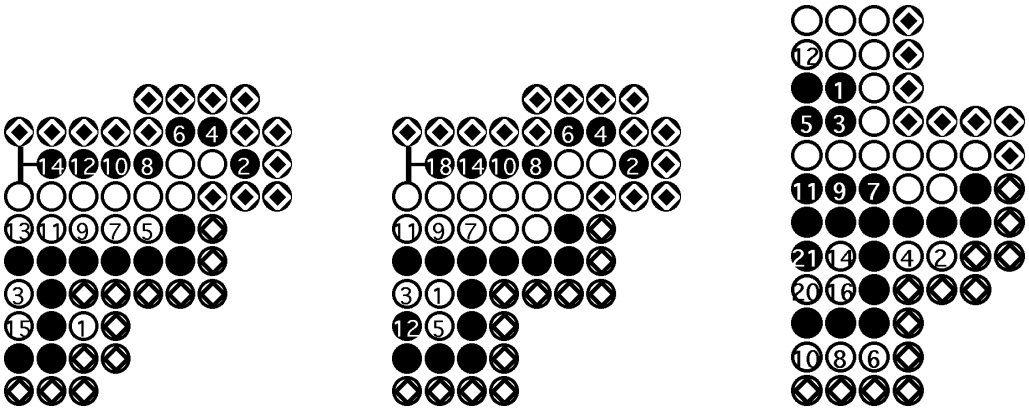


Fig. 13. Sample solution sequences for problems A, B and C. Solution B: 13 at 1, 15 at 5, 16 at 3, 17 at 5. Solution C: 13 at 1, 15 at 3, 17 at 5, 19 at 3, 22 at 16, 23 at 5, 24 at 1, 25 at 3

is able to solve these problems statically. In contrast, both *L1* and *L3* take hundreds of nodes to solve problem B and thousands of nodes to solve problem C.

The second experiment involves some more complex semeai that cannot be solved statically by the partial order evaluation. In this experiment, search using partial order bounding is compared to search using Landman's approach. In problem D, none of the intermediate problems on the sample solution sequence can be solved statically by *POB*, so search must be used in each instance. However, *POB* clearly outperforms *L1* and *L3* in all nontrivial situations. In problems E and F, only the original problem requires a search by *POB*, and this search needs orders of magnitude less nodes than *L1* and *L3*.

Start	L1	L3	POB	Start	L1	L3	POB
W 13	2/2	static	static	B 12	29/6	static	static
W 11	10/6	static	static	B 10	127/22	static	static
W 9	28/22	static	static	B 8	354/55	static	static
W 7	63/55	static	static	B 6	781/113	static	static
W 5	123/113	static	static	B 4	1496/204	static	static
W 3	216/204	static	static	B 2	1543/229	static	static
W 1	243/229	static	static	Orig.	1594/258	static	static

Table 1
Results for problem A

Start	L1	L3	POB	Start	L1	L3	POB
W 17	2/2	static	static	B 16	9/5	static	static
W 15	9/2	static	static	B 14	9/14	static	static
W 13	20/14	static	static	B 12	2/27	static	static
W 11	35/2	2/2	static	B 10	84/73	2/6	static
W 9	106/73	13/6	static	B 8	224/213	139/25	static
W 7	225/213	31/25	static	B 6	559/345	399/58	static
W 5	359/345	66/58	static	B 4	594/374	434/75	static
W 3	390/374	85/75	static	B 2	633/407	473/96	static
W 1	425/407	108/96	static	Orig.	676/444	516/121	static

Table 2
Results for problem B

5.4.2 Experiment 3: A Comparison With Commercial Go Programs

The partial order semeai module has been integrated into our Go program *Explorer* [23]. For testing how the semeai knowledge of our program compares to that of leading commercial Go programs, we constructed two full board Go positions that contain a large variety of semeai positions. Starting from these positions, shown in Figures 16 and 17, we played *Explorer* against two recent world championship winning programs, *The Many Faces of Go* and *Go 4++*. *Explorer* is able to play these types of positions perfectly, but its opponents made a number of serious mistakes. In most test games, *Explorer* turned around several semeai that it should have lost in theory, and thereby gained a large number of points over the game-theoretically optimal result, which was determined by self-play. Detailed results are listed in Tables 7 and 8. The test positions and game records in SGF format are available on the web at http://www.cs.ualberta.ca/~mmueller/cgo/semeai.html.

Start	L1	L3	POB	Start	L1	L3	POB
B 25	2/5	static	static	W 24	9/20	static	static
B 23	2/14	static	static	W 22	20/20	static	static
B 21	2/27	static	static	W 20	35/2	2/2	static
B 19	20/44	20/5	static	W 18	54/2	9/2	static
B 17	2/65	2/14	static	W 16	77/77	20/20	static
B 15	54/90	54/27	static	W 14	104/104	35/35	static
B 13	77/119	77/44	static	W 12	135/135	54/54	static
B 11	2/152	2/65	static	W 10	346/630	132/513	static
B 9	246/475	213/189	static	W 8	1292/1767	418/1396	static
B 7	873/1313	646/433	static	W 6	1784/3028	627/2433	static
B 5	1222/1807	916/644	static	W 4	1831/3083	662/2484	static
B 3	1265/1856	955/681	static	W 2	1882/3142	701/2539	static
B 1	1312/1909	998/722	static	Orig.	1937/3205	744/2598	static

Table 3
Results for problem C

Start	L1	L3	POB	Start	L1	L3	POB
W 21	22/2	18/2	18/2	B 20	12/44	2/32	2/32
W 19	90/56	64/32	64/32	B 18	2/120	2/84	2/84
W 17	129/2	93/2	93/2	B 16	89/195	15/159	2/119
W 15	261/205	222/153	175/113	B 14	180/345	40/305	2/207
W 13	429/305	385/294	268/187	B 12	285/537	77/490	2/306
W 11	552/2	505/2	321/2	B 10	383/764	126/717	2/365
W 9	740/686	766/704	459/334	B 8	532/897	187/915	2/509
W 7	984/745	1025/914	600/450	B 6	691/1166	260/1196	2/656
W 5	1242/933	1298/1199	757/580	B 4	868/1449	345/1491	2/819
W 3	1526/1135	1597/1500	930/724	B 2	1063/1758	442/1812	2/998
W 1	1827/1355	1924/1820	1119/882	Orig.	1272/2084	551/2161	2/1193

Table 4
Results for problem D

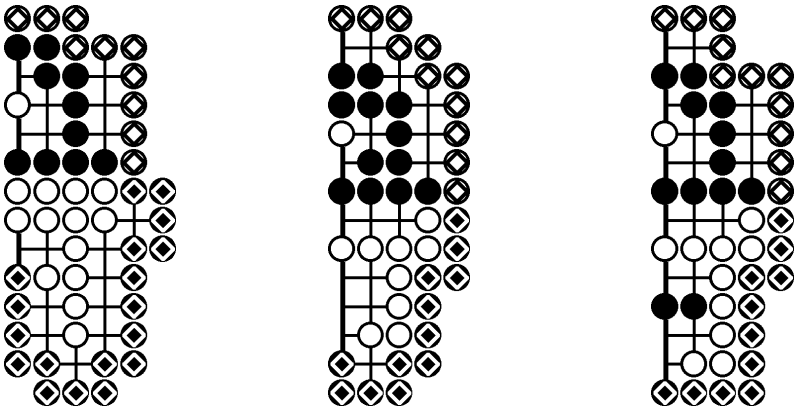


Fig. 14. Semeai test problems D, E and F

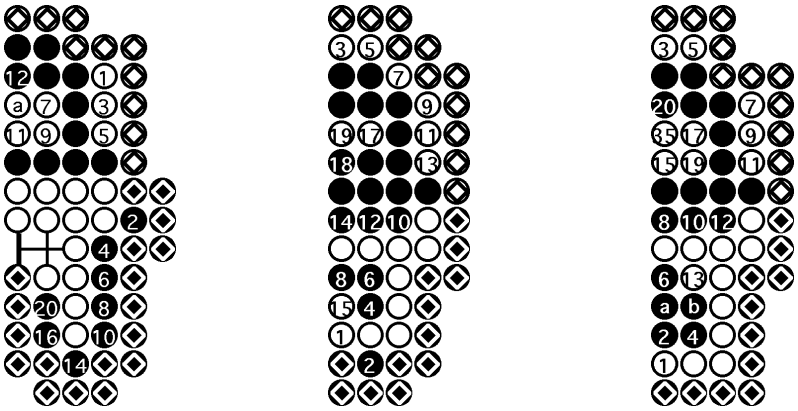


Fig. 15. Sample solution sequences for problems D, E and F. Solution D: 13 at 7, 15 at 9, 17 at a, 18 at 11, 19 at 7, 21 at 9. Solution E: 16 at 6. Solution F: 14 at a, 16 at 2, 18 at b, 21 at 35, 22 at 4, 23 at 6, 24 at a, 25 at 15, 26 at 2, 27 at 17, 28 at 19, 29 at 20, 30 at b, 31 at 4, 32 at a, 33 at 15, 34 at 17.

To summarize the experiments, by using partial order evaluation techniques a large variety of basic semeai situations can be evaluated much earlier than with previously proposed methods, and this ability leads to much more accurate semeai play than the techniques currently used by leading commercial programs.

5.5 Limitations of the Semeai Solver

The current semeai solver is limited in a number of ways:

- Each problem must be completely surrounded by a wall of safe stones. This is a severe restriction, but the same precondition is currently required by exact solvers for the domains of life and death problems [45] and endgames

Start	L1	L3	POB	Start	L1	L3	POB
W 19	5/2	static	static	B 18	20/9	static	static
W 17	14/2	static	static	B 16	20/20	static	static
W 15	27/2	static	static	B 14	2/35	2/2	static
W 13	178/146	28/14	static	B 12	93/218	6/38	static
W 11	517/354	133/78	static	B 10	273/533	25/143	static
W 9	1008/742	367/245	static	B 8	429/1028	58/381	static
W 7	1049/781	396/272	static	B 6	460/1071	75/412	static
W 5	1094/824	429/303	static	B 4	495/1118	96/447	static
W 3	1143/871	466/338	static	B 2	534/1169	121/486	static
W 1	1196/922	507/377	static	Orig.	5664/70072	465/2599	242/6

Table 5
Results for problem E

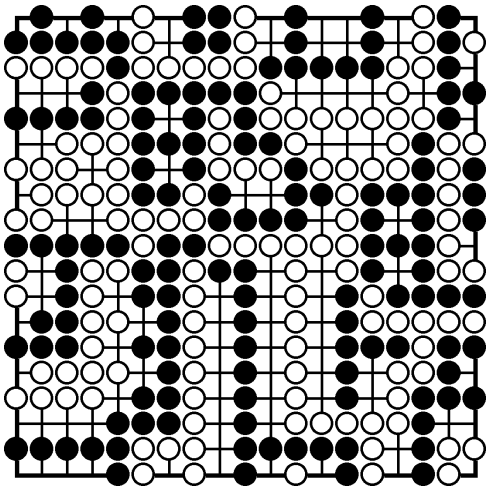


Fig. 16. Full board semeai problem 1

[24]. Wolf [44] discusses the issues involved in the generalization of a life and death problem solver to handle "open-ended" problems. The same issues arise in semeai.

- The solver cannot yet handle semeai that include ko fights. A full solution to this problem seems to be outside the scope of minimax based search, as discussed at the start of this section.
- The solver is much less efficient for eyes and liberty regions that do not fulfill the strict conditions of *plain eye* and *plain liberty regions* [25]. In practice, such areas are often equivalent to some plain eye or plain liberty region. However, the current program does not contain any specialized knowledge or local search methods for these cases, and therefore reverts to a brute

Start	L1	L3	POB	Start	L1	L3	POB
W 35	5/2	static	static	B 34	20/9	static	static
W 33	14/2	static	static	B 32	20/20	static	static
W 31	27/2	static	static	B 30	2/35	2/2	static
W 29	44/20	5/20	static	B 28	2/54	2/9	static
W 27	65/2	14/2	static	B 26	77/77	20/20	static
W 25	90/54	27/54	static	B 24	104/104	35/35	static
W 23	119/77	44/77	static	B 22	2/135	2/54	static
W 21	152/104	65/104	static	B 20	170/170	77/77	static
W 19	189/2	90/2	static	B 18	209/209	104/104	static
W 17	230/170	119/170	static	B 16	252/252	135/135	static
W 15	275/209	152/209	static	B 14	299/299	170/170	static
W 13	2/1396	2/189	static	B 12	2/350	2/209	static
W 11	1615/2286	1487/1359	static	B 10	1007/2531	498/2029	static
W 9	11617/10397	11674/10176	static	B 8	6851/11651	2482/11708	static
W 7	20318/18891	20422/18743	static	B 6	9462/20356	3335/20460	static
W 5	20395/18966	20499/18818	static	B 4	9529/20435	3390/20539	static
W 3	20476/19045	20580/18897	static	B 2	9600/20518	3449/20622	static
W 1	20584/19109	20672/18945	static	Orig.	5316/37363	5316/37457	174/6

Table 6
Results for problem F

Opponent	B starts	Gain	W starts	Gain
Explorer	B+36	-	B+6	-
Many Faces (B)	W+103	+139	W+85	+91
Many Faces (W)	B+53	+17	B+117	+111
Go 4++ (B)	W+40	+76	W+40	+46
Go 4++ (W)	B+116	+80	B+129	+123

Table 7
Results of full board semeai problem 1

force approach in such regions. Augmenting the partial order evaluation by values representing upper and lower bounds as proposed in Section 4.3.1 would probably improve the performance dramatically in many of these cases. For example, problem D in Figure 14 could be solved statically with such a refined partial order evaluation.

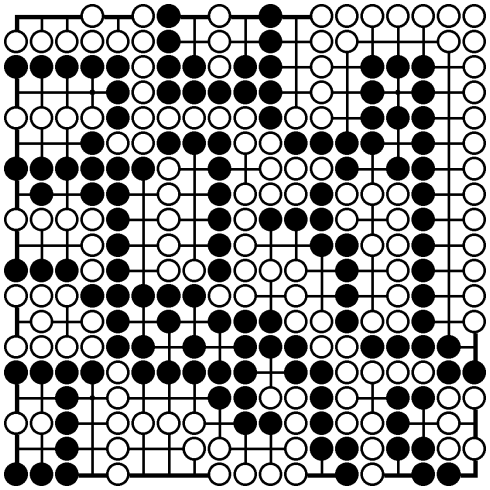


Fig. 17. Full board semeai problem 2

Opponent	B starts	Gain	W starts	Gain
Explorer	W+9	-	W+69	-
Many Faces (B)	W+93	+84	W+73	+4
Many Faces (W)	B+95	+104	B+139	+208
Go 4++ (B)	W+99	+90	W+99	+30
Go 4++ (W)	B+55	+64	B+51	+120

Table 8
Results of full board semeai problem 2

6 Related Work: Other Search Methods that Use Partial Order Evaluation

This section surveys a number of previous search methods that make use of partial order evaluation.

6.1 Multiobjective Search Methods

In the multiobjective approach to decision making, multiple potentially conflicting and incommensurate objectives are investigated simultaneously. Each single objective is evaluated by a scalar. All evaluations together form a vector, and a dominance relation between vectors is defined as described in Section 4.2.2.

Multiobjective search has been studied by a number of authors [11,12,14,37]. The first multiobjective search method, multiobjective A* (MOA*), was de-

veloped by Stewart and White [37]. Given an OR-graph with vector-valued edge costs, MOA* finds all nondominated paths from a start node to a given set of goal nodes.

Harikumar and Kumar introduced an iterative deepening version of MOA* called IDMOA* [14]. Dasgupta, Chakrabarti and DeSarkar developed a multi-objective heuristic search method for AND/OR graphs and partial order game tree search [11,12]. This method uses vector-valued leaf node evaluations, with a partial order defined by the vector dominance relation.

Despite using a partial order for evaluation, the method of [12] crucially relies on a total order, namely a players' private *preference*. A player's preference ϕ is a many-to-one mapping from the partially ordered set P to a number, which preserves the partial order on P. A player knows only his or her own preferences, and is indifferent regarding which of the outcomes among those with the same preference ϕ are chosen. Sets of possible outcomes which are incomparable in P are compared using the following procedure [12, p.240–241]:

Compare(S_1, S_2, ϕ)
To compare sets of outcomes S_1 and S_2
on the basis of preferences ϕ
1. If only S_1 is empty, declare S_2 as better.
Likewise, if only S_2 is empty, declare S_1 as better.
If both S_1 and S_2 are empty then
select S_1 or S_2 randomly and declare it to be better
2. let x_1 be the worst outcome in S_1 and
x_2 be the worst outcome in S_2 based on ϕ.
3. if x_1 and x_2 are of equal preference then
3.1 Drop all outcomes from S_1 and S_2 that are of
equal preference to x_1
3.2 Goto [Step 1]
4. If x_1 is better than x_2 based on ϕ,
then declare S_1 as better
else declare S_2 as better.

This approach to partial order evaluation needs a reasonably strong ordering by the player's preference ϕ. If too many values are mapped to the same preference the comparison breaks down. In the case studied in this paper, where a player has no a priori preference between incomparable values, the comparison of any two nonempty sets of outcomes degenerates to the final line of step 1 in *Compare()*: random selection.

6.1.1 Comparison of Dasgupta, Chakrabarti and DeSarkar's method with POB

Despite sharing the common goal of using partial order evaluation in game tree search, the approach of [12] and POB are very different in spirit. Dasgupta, Chakrabarti and DeSarkar assume that players have distinct, totally ordered private preferences. In contrast, in POB the partial order evaluation reflects a lack of knowledge of the player about the correct evaluation. The player simply cannot decide which option to choose and therefore prefers to avoid a decision at this point, hoping that deeper search will lead to better opportunities to make a meaningful comparison later.

In [12], backing up sets of nondominated options through the tree is expensive. Sets of options must be compared with each other, converted from a min-representation to a max-representation, and different kinds of dominated options must be eliminated. The size of option sets can grow very large. For example, in the case of 2-dimensional real-valued vectors, for each constant c and each set of reals X the set $\{(x, c-x)|x \in X\}$ is an antichain of cardinality $|X|$. In the case where there is a large number of incomparable options, the algorithm of [12] has to enumerate all of them. Although it is difficult to really compare the methods because of their different assumptions and goals, POB seems to have a large advantage in terms of efficiency since it only needs to propagate boolean values.

6.2 Barbara Huberman's "Program to Play Chess End Games"

In her 1968 Ph.D. thesis "A Program to Play Chess End Games" [16] Barbara Huberman develops a method that is similar to partial order evaluation. Huberman's work is a case study in knowledge engineering in the domain of chess endgames. Starting from rules and examples in chess textbooks, she develops evaluation rules which, in combination with shallow searches, guarantee a program's progress towards the goal of mating the opponent king. While not always optimal in the sense of finding the shortest way to mate, the program can find a win from every winning starting position in the endgames of king and rook against king, king and two bishops against king, and king-knight-bishop against king. Of course, on today's hardware such endgames can be exhaustively analyzed within a few seconds by means of retrograde analysis [40]. However, it is interesting to compare Huberman's search framework to POB.

In the *forcing tree model* of [16], a predicate $better(p, q)$ defined over game positions is used to track the progress of the attacker. From a starting position p with the attacker to play, search is used to build a *forcing tree* ending

in defender-to-play positions q_i where $better(p, q_i)$ holds for all q_i. Iterative deepening tree search is used to check whether a *better* position can be forced at depth 1, 3, 5, and so on. The search tree is pruned using another predicate $worse(p, q)$, which recognizes failures and likely failures of the attacker.

For chess endgames, *better* and *worse* predicates are defined in terms of *game stages* and *substages*, which represent high-level goals taken from textbooks, such as pushing the opponent king to the edge of the board. Progress in this model is defined as moving forward to the next (sub-)stage.

6.2.1 Comparison of Huberman's Approach with POB

The basic *better-worse* framework of Huberman's thesis can be regarded as a partial order evaluation, and is quite close in spirit to POB. Instead of explicitly defining search over partially ordered evaluations as in POB, a partial ordering of positions is implicit in the properties of the predicates used. The *better* and *worse* predicates lead only to a partial ordering of positions since there are many pairs of positions p, q for which none of the relations $better(p, q)$, $better(q, p)$, $worse(p, q)$ or $worse(q, p)$ hold.

A common feature of the forcing tree model and POB is that search is only used to satisfice, not to optimize. Search stops at the first move recognized as *better*, even though other, *much better* moves might exist. Similarly, POB does not distinguish between the different evaluations within a success set. LE-POB combines both satisficing and optimizing to some degree: it ensures a *better* result if possible, but it is also able to search for *much better* evaluations which are evaluated higher in the linear extension of the poset evaluation.

Huberman's stage-substage model with its linear progress towards a single, predefined goal is not as close to the spirit of POB. The strict definitions of *better* and *worse* in terms of (sub-)stages guarantee progress only in well-controlled, restricted domains such as the chess endgames studied. In the POB model, there is no single prescribed direction of play. The framework is flexible and allows different ways to make progress, within the limits of the current partial order bound. Also, in Huberman's model the *better* predicate always compares positions to a single value, namely the evaluation of the current position. POB has no such restriction, it allows any kind of partial order bound, including bounds containing many incomparable values. Finally, in the case that search with the initial bound fails, in POB it is possible to re-search with a more modest bound.

Many game tree search methods model uncertainty about the true value of a scalar evaluation by a probability distribution, or by a small set of scalars representing for example optimistic, pessimistic and realistic evaluations. See [3,18] for detailed discussions. Just like standard minimax search, these methods are based on the empirically well-founded assumption that deeper search can be used to reduce uncertainty about the true value.

In contrast, POB does not assume the existence of a "true" scalar evaluation, which can be discovered by a deeper search. The degree of uncertainty in a position evaluation is modeled directly by the partial order. Searches are tailored to test whether it is possible to achieve a given level of evaluation.

In POB, search is controlled by the bound defining the success set. If the success set contains only high certainty evaluations, search goes deeper at unclear situations. On the other hand, if the player to move is in an unfavorable situation, the bound can be set in such a way that the success set includes unclear positions, and therefore search is used to avoid clear losses.

7 Conclusion and Future Work

Partial order bounding (POB) is a simple but powerful new partial order game tree search method. The method extends the idea of null window searches, which test whether an evaluation better than a given bound can be achieved, from the case of scalar evaluations to partially ordered sets, by using an antichain bound that divides a poset into success and failure sets. POB has been used to solve semeai problems in Go and validated by a comparison to a standard alpha-beta search.

There are many promising directions for future work, to extend the ideas developed in this paper:

- Build heuristic partial order evaluations for many specific games and apply POB.
- Develop software tools for constructing efficient representations of partial orders and bounds.
- Use the POB approach in single-agent search algorithms similar to IDMOA* [14], and thereby extend the scope of this type of search methods to arbitrary representations of partial orders.
- Use machine learning methods to learn partial order evaluations.
- Apply POB in the context of combinatorial game search.

A preliminary version of this research was presented at a talk at the AAAI 1999 Spring Symposium on *Search Techniques for Problem Solving under Uncertainty and Incomplete Information* in Stanford. I would like to thank the participants for useful comments, especially Murray Campbell for the example of passed pawns as an unstable long-term feature in chess, and Richard Korf who pointed me to Barbara Huberman's Ph.D. thesis [16]. I also want to thank my colleague Reijer Grimbergen at ETL for many discussions about problems of evaluation in shogi and other games, and the anonymous referees for their valuable suggestions on how to improve this paper, especially the experimental section.

Appendix: The Rules of Go

This is only a very brief summary of the Go rules. For details, see [7] or an online source such as *www.usgo.org*. Go is played between two players called Black and White, who alternatingly place a stone of their own color on an empty intersection on a square grid. Black plays first. The standard board has 19×19 lines, but smaller sizes are sometimes used. The goal of the game is to control a larger area than the opponent, by placing stones such that they surround empty points and unsafe opponent stones, and cannot be captured by the opponent.

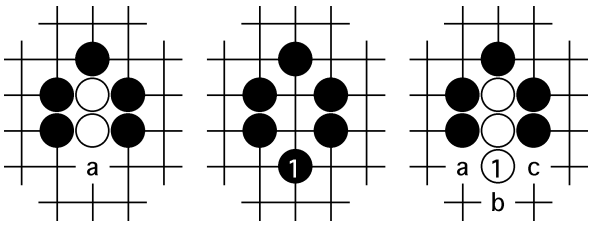


Fig. 18. The capturing rule

Stones of the same color connected horizontally or vertically form a unit called a *block*. Empty points adjacent to a block are called *liberties* of the block. A block is *captured* and removed from the board if its last liberty is occupied by the opponent. Figure 18 shows two white stones with a single liberty at 'a'. If Black plays there, the two white stones are captured and removed from the board. If White plays on the same point first, the white stones have three liberties at 'a', 'b' and 'c' and Black needs to fill all of them to capture the

three connected stones. Capturing and recapturing stones can potentially lead to the infinite repetition of positions, which is forbidden by the rules. Players can pass at any time; two or three consecutive passes end the game. An *eye* is a (small) area surrounded by one player. The area can contain stones of both colors and empty points. Blocks with two separate eyes are safe from capture. Blocks with a single eye can still be captured, but they are often stronger than blocks without an eye.

References

[1] L.V. Allis. *Searching for Solutions in Games and Artificial Intelligence.* PhD thesis, University of Limburg, Maastricht, 1994.

[2] T.S. Anantharaman. Evaluation tuning for computer chess: Linear discriminant methods. *ICCA Journal*, 20(4):224–242, 1997.

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

[4] J. Baxter, A. Tridgell, and L. Weaver. Experiments in parameter learning using temporal differences. *International Computer Chess Association Journal*, 21(2):84–99, 1998.

[5] D. Beal. *The Nature of Minimax Search.* PhD thesis, Universiteit Maastricht, 1999.

[6] D. F. Beal and M. C. Smith. Temporal coherence and prediction decay in TD learning. In *Proceedings of the 16th International Joint Conference on Artificial Intelligence (IJCAI-99)*, pages 564–569, 1999.

[7] R. Bozulich. *The Go Players Almanac.* Ishi Press, Tokyo, 1992.

[8] M. Buro. Experiments with Multi-ProbCut and a new high-quality evaluation function. Technical Report 96, NECI, Princeton, N.J., 1997.

[9] M. Buro. From simple features to sophisticated evaluation functions. In H.J.van den Herik and H.Iida, editors, *Computers and Games: Proceedings CG'98. LNCS 1558*, pages 126–145. Springer Verlag, 1999.

[10] W.W. Cohen, R.E. Schapire, and Y. Singer. Learning to order things. *Journal of Artificial Intelligence Research*, 10:243–270, 1999.

[11] P. Dasgupta, P. Chakrabarti, and S. DeSarkar. Multiobjective heuristic search in AND/OR graphs. *Journal of Algorithms*, 20(2):282–311, 1996.

[12] P. Dasgupta, P. Chakrabarti, and S. DeSarkar. Searching game trees under a partial order. *Artificial Intelligence*, 82(1–2):237–257, 1996.

[13] J. Fishburn. *Analysis of Speedup in Distributed Algorithms.* PhD thesis, University of Wisconsin, Madison, 1981.

[14] S. Harikumar and S. Kumar. Iterative deepening multiobjective A*. *Information Processing Letters*, 58(1):11–15, 1996.

[15] E. Heinz. Efficient interior-node recognition. *ICCA Journal*, 21(3):157–168, 1998.

[16] B. J. Huberman. *A Program to Play Chess End Games*. PhD thesis, Stanford University, 1968. Available from University Microfilms, Inc., www.umi.com, Nr. 69-8199.

[17] R. Hunter. Counting liberties, How to win capturing races. In R. Bozulich, editor, *The Second Book of Go*, chapter 7 and 8. Kiseido, Tokyo, 1998.

[18] A. Junghanns. Are there practical alternatives to alpha-beta? *ICCA Journal*, 21(1):14–32, January 1998.

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

[20] H. Landman. Eyespace values in Go. In R. Nowakowski, editor, *Games of No Chance*, pages 227–257. Cambridge University Press, 1996.

[21] K.F. Lenz. Die Semeai-Formel. *Deutsche Go-Zeitung*, 57(4), 1982.

[22] T. Mitchell. *Machine Learning*. McGraw-Hill, 1997.

[23] M. Müller. *Computer Go as a Sum of Local Games: An Application of Combinatorial Game Theory*. PhD thesis, ETH Zürich, 1995. Diss. ETH Nr. 11.006.

[24] M. Müller. Decomposition search: A combinatorial games approach to game tree search, with applications to solving Go endgames. In *IJCAI-99*, pages 578–583, 1999.

[25] M. Müller. Race to capture: Analyzing semeai in Go. In *Game Programming Workshop in Japan '99*, volume 99(14) of *IPSJ Symposium Series*, pages 61–68, 1999.

[26] M. Müller. Generalized thermography: A new approach to evaluation in computer Go. In J. van den Herik and H. Iida, editors, *Games in AI Research*, pages 203–219, Maastricht, 2000. Universiteit Maastricht.

[27] M. Müller, E. Berlekamp, and B. Spight. Generalized thermography: Algorithms, implementation, and application to Go endgames. Technical Report 96-030, ICSI Berkeley, 1996.

[28] K. Nakamura. Graph theoretic analyses of Go board phases. In J. van den Herik and H. Iida, editors, *Games in AI Research*, pages 239–249, Maastricht, 2000. Universiteit Maastricht.

[29] D.S. Nau. Pathology on Game Trees revisited and an alternative to minimaxing. *Artificial Intelligence*, 21(1-2):221–244, 1983.

[30] O. Ore. *Theory of Graphs*, volume 38 of *Colloquium Publications*. American Mathematical Society, Providence, 1962.

[31] J. Pearl. *Heuristics: Intelligent search strategies for computer problem solving.* Addison-Wesley Publishing Company, 1984.

[32] A. Plaat, J. Schaeffer, W. Pijls, and A. De Bruin. Best-first fixed-depth minimax algorithms. *Artificial Intelligence*, 87:255–293, 1996.

[33] S. Russell and P. Norvig. *Artificial Intelligence: a Modern Approach.* Prentice Hall, 1995.

[34] A. Scheucher and H. Kaindl. Benefits of using multivalued functions for minimaxing. *Artificial Intelligence*, 99:187–208, 1998.

[35] R. Stanley. *Enumerative Combinatorics Vol. 1.* Number 49 in Cambridge Studies in Advanced Mathematics. Cambridge University Press, 1997.

[36] G. Steiner. An algorithm to generate the ideals of a partial order. *Operations Research Letters*, 5:317–320, 1986.

[37] B. Stewart and C. White. Multiobjective A*. *Journal of the ACM*, 38(4):775–814, 1991.

[38] M. Talamo and P. Vocca. An efficient data structure for lattice operations. *SIAM Journal on Computing*, 28:1783–1805, 1999.

[39] G. Tesauro. TD-Gammon, a self-teaching backgammon program, achieves master-level play. *Neural Computation*, 6:215–219, 1994.

[40] K. Thompson. Retrograde analysis of certain endgames. *ICCA Journal*, 9(3):131–139, 1986.

[41] S. Thrun. Learning to play the game of chess. In G. Tesauro, D. Touretzky, and T. Leen, editors, *Advances in Neural Information Processing Systems (NIPS) 7*. MIT Press, 1995.

[42] W. Trotter. *Combinatorics and Partially Ordered Sets. Dimension Theory.* Johns Hopkins University Press, Baltimore, 1992.

[43] Y. Varol and D. Rotem. An algorithm to generate all topological sorting arrangements. *The Computer Journal*, 24:83–84, 1981.

[44] T. Wolf. About problems in generalizing a tsumego program to open positions. In H. Matsubara, editor, *Game Programming Workshop in Japan '96*, pages 20–26, Computer Shogi Association, Tokyo, Japan, 1996.

[45] T. Wolf. Forward pruning and other heuristic search techniques in tsume go. *Information Sciences*, 122:59–76, 2000.

[46] J. Yanez and J. Montero. A poset dimension algorithm. *Journal of Algorithms*, 30:185–208, 1999.