Automated Discovery of Search-Extension Features

Pálmi Skowronski[1], Yngvi Björnsson[1], and Mark H.M. Winands[2]

[1] Reykjavík University, School of Computer Science,
Menntavegi 1, Reykjavík 101, Icleand
{palmis01,yngvi}@ru.is

[2] Games and AI Group, Department of Knowledge Engineering,
Maastricht University, Maastricht, The Netherlands
m.winands@maastrichtuniversity.nl

Abstract. One of the main challenges with selective search extensions is designing effective move categories (features). Usually, it is a manual trial-and-error task, which requires both intuition and expert human knowledge. Automating this task potentially enables the discovery of both more complex and more effective move categories. The current work introduces *Gradual Focus*, an algorithm for automatically discovering interesting move categories for selective search extensions. The algorithm iteratively creates new more refined move categories by combining features from an atomic feature set. Empirical data is presented for the game Breakthrough showing that Gradual Focus looks at a number of combinations that is two orders of magnitude fewer than a brute-force method does, while preserving adequate precision and recall.

1 Introduction

The $\alpha\beta$ algorithm is one of the fundamental and most effective search techniques used by game-playing programs for playing two-person adversary board games, such as chess and checkers. Over the years many enhancements have been proposed to improve its efficiency. For instance, we know that the standard strategy of exploring all alternatives to the same fixed depth is not most effective. Instead, various techniques have been proposed for searching the game tree more selectively, where some lines of play are terminated prematurely whereas others are explored more deeply. The former scenario is referred to as search reductions (or speculative pruning) and the latter as search extensions. In chess, for example, it is common to resolve forced situations, such as checks and recaptures, by searching them more deeply.

The move-decision quality of the alpha-beta algorithm is greatly influenced by the choices of which lines are investigated deeply [1,2]. Therefore, the design of an effective search-extension scheme is fundamental to any high-performance $\alpha\beta$-based game-playing program. The typical approach for incorporating search extensions into a game-playing program is to predefine a set of move categories (e.g., checks and recaptures), and then associate a different cost weight to each

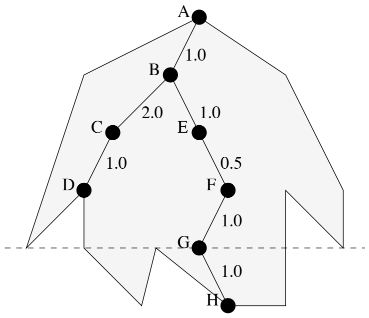

Fig. 1. Fractional-ply example

category. During the search, each move is categorized as belonging to one of the predefined move classes, and the depth of the current search path then becomes the sum of the weights of the moves on the path. If all move categories have the same weight, one would obtain the regular behaviour of a fixed-depth search. However, by assigning a weight of less than one to selected move categories, e.g., checking moves, such lines of play will be explored more deeply. This scheme is commonly referred to as fractional-ply extensions [3,4], depicted in Figure 1.

The weight of each move category, i.e., its fractional-ply value, is either manually assigned a value based on trial and error, or, alternatively, automatically tuned from game records [5,6], test-suites [7], or during play [7,8,9]. For creating the move categories the standard practice is to do it manually based on intuition and domain expertise. In this work we investigate ways for automatically discovering useful move categories for use in game-playing programs. The main contribution is a new method for automatically discovering such features, called *Gradual Focus*. We experiment with it in the game Breakthrough, where there exists little knowledge of what comprises good moves to extend on.

The paper is structured as follows. In Section 2 we give an overview of relevant background material, followed by a description of the new Gradual Focus feature-discovery algorithm in Section 3. The algorithm is empirically evaluated in Section 4, and finally we conclude and discuss future work in Section 5.

2 Background

The general approach to automated feature discovery is to start with a set of so-called *atomic* features. The atomic features are typically simple features expressing trivial facts about the problem domain, for example, type and placement of pieces. As a standalone, these features are not necessary effective, for example, because they might be too general. More sophisticated features are then constructed by combining the atomic features in various ways, e.g., by using the logical operators $\wedge$ and $\vee$. This may be done in an iterative fashion, that is, first pair-wise combinations are created, then three-wise, etc. The problem ,though, is that the number of possible feature combinations grows exponentially in each iteration. For example, a brute-force power set method would generate in total $2^n - 1$ features from an atomic feature set of size n. Consequently, to limit the growth rate, a selective mechanism judging the merits of newly create features may be applied, carefully choosing which features to evolve further.

In the context of game-playing, automatic feature discovery has first and foremost been applied to the learning of evaluation functions, as opposed to search-control features. One of the first such approaches was introduced in the system Zenith [10]. The system works in a way backwards to the general approach described above: it starts with a single feature, a logical formula describing the goal of the game. It then gradually breaks the goal down into simpler sub-goals by using predefined generic actions in the form of decomposition, abstraction, goal regression, and specialization. Logical features of this kind have also been successfully used to extract patterns that can be used as features for general game playing [11]. In contrast, the system GLEM [12], creates new features by gradually combining mutually exclusive atomic features along the lines described above. The method is additionally capable of learning an importance weight for each of the newly created features. This method was used to construct a high-quality evaluation function for the Othello program Logistello [13], although, in that case, the features were provided manually and GLEM used only for tuning their relative weights. A different feature-combination approach was used to learn an evaluation function for a program to play the card game Hearts [14]. All possible pair-wise, three-wise, and four-wise combinations were created in more or less a brute-force manner and a reinforcement-learning approach then used to learn their relative importance. Finally, learning of move-patterns for a plausible move generation in chess is presented in [15].

3 Gradual Focus

Gradual focus (GF), the method we introduce here, is a more intelligent way of constructing interesting features than an exhaustive power-set method. GF combines atomic features in an iterative fashion, where each iteration creates a set of more refined features, gradually narrowing their focus, using a variety of pruning methods to reduce the number of possible combinations.

3.1 GF Overview

Given a set of atomic features GF combines these features using an $\wedge$ operator. The features are combined one level at a time, i.e., one-wise, two-wise, three-wise, etc., and their quality evaluated. Those features that do not show an improvement are pruned off and prevented from occurring as subsets in later feature combinations. This process is repeated until no more combinations are formed.

Figure 2 shows an example of this process. The first level consists of the atomic features, called *Base* set, and the second level shows the feature set generated by the first iteration, where all two-wise combinations of the *Base* set are created. Each feature in the set is evaluated individually and those that perform

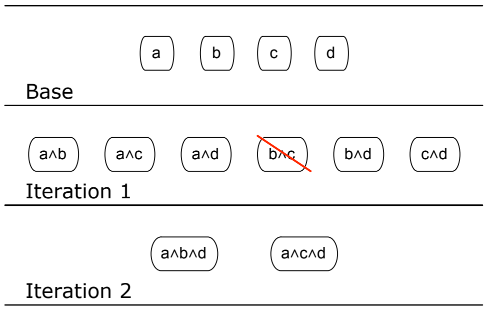

Fig. 2. Overview of Gradual Focus

worse than either parent are discarded, as feature $b \wedge c$ in this example. The next iteration evolves the surviving features further by combing the two-wise features with the features in the *Base* set. As the $b \wedge c$ feature has been discarded it is not evolved any further, nor are any evolving features containing $b \wedge c$ allowed. This results in only two three-wise combinations. Four-wise combinations cannot be formed either as $b \wedge c$ is forbidden as a subfeature, so the process halts.

3.2 Implementation Details

A pseudo-code for the GF algorithm is shown as Algorithm 1. The variable *BlackList* maintains disproven features, while the *Output* variable maintains the viable features. The *Neutral* variable is the empty feature; it is evaluated in *line 3* to establish a baseline of how the tree search behaves without search extensions. The function *evaluateFitness*, which will be discussed in detail in Section 3.6, returns a numerical value indicating the quality, i.e., the fitness of a feature. Each of *Base's* features is also evaluated *(line 5)*. Those features that perform below an expected level of quality *(lines 7-13)* can optionally be removed from the *Base* set, reducing GF's branching factor. This pruning method, called *threshold pruning*, is discussed in a more detail in Section 3.5.

The iterative feature-evolution process is done in *lines 17-26*. The function *evolveFeatures (line 19)* evolves the features in the *workSet* by combining them with the *Base* set, and using the *BlackList* to remove disallowed combinations. Both the *Base* and the *workSet* sets must be sorted in a descending order by evaluation score *(lines 14 and 18)* for *evolveFeatures* to work as intended.

The assessment of newly evolved features occurs in *line 20-25*, where each of the newly evolved features is evaluated *(line 22)* and its evolutionary direction assessed within *filterFeature (line 23)* as either ascending or descending. Descending combinations are added to the *BlackList*, thereby removing them and their descendants from the evolutionary process, while ascending combinations are returned from the function and added to GF's output *(line 23)*. Disproving a feature within *filterFeature* might also disprove other features in the *workSet* that are yet to be evaluated, which is why features must be compared against

Algorithm 1. *featureLearner(ref Base)*

```
 1: $BlackList \leftarrow \{\}$
 2: $Output \leftarrow \{\}$
 3: evaluateFitness( Neutral )
 4: for all $b \in Base$ do
 5:     evaluateFitness( b )
 6: end for
 7: if UseThreasholdPruning then
 8:     for all $b \in Base$ do
 9:         if $b_{value} < \delta$ then
10:             $Base \leftarrow Base \setminus \{b\}$
11:         end if
12:     end for
13: end if
14: sortDesc( Base )
15: $Output \leftarrow Neutral \cup Base$
16: $workSet \leftarrow Base$
17: while $workSet \neq \{\}$ do
18:     sortDesc( workSet )
19:     $workSet \leftarrow$ evolveFeatures( workSet, Base, BlackList )
20:     for all $f \in workSet$ do
21:         if isCompositionAllowed( f, BlackList ) then
22:             evaluateFitness( f )
23:             $Output \leftarrow Output \cup$ filterFeature( f, BlackList )
24:         end if
25:     end for
26: end while
27: sortDesc( Output )
28: display( Output )
```

the *BlackList* each time in the loop before they are evaluated *(line 21)*. This evolution process is iterated until no new features can be formed, the *evolveFeatures* function returns an empty set *(line 19 and 17)*, meaning that all promising evolution paths have been explored.

The implementation of the *evolveFeatures* routine is shown as Algorithm 2, where A is a set of features to be evolved and B is the set of features to combine with. The same feature can be composed in various ways with different first and second parent (i.e., $a \wedge b$ and $b \wedge a$ are considered the same feature). However, to simplify the evaluation of features we impose an order such that the first parent must have a greater fitness value than the second parent, which is why the input sets are sorted in a descending order before the evolution phase and then paired together from left to right. To prevent unnecessary combinations, the features can neither belong to the same group nor can a subset of the new feature c be on the *BlackList (line 4 and 6)*. Moreover, duplicate combinations are not allowed *(line 6)*. In the Sections 3.3, 3.4, and 3.5, we discuss three enhancements. Feature groups prevent illogical combinations and will be discussed in a more detail in

Algorithm 2. *evolveFeatures(A, B, BlackList)*

```
1: $new \leftarrow \{\}$
2: for all $a \in A$ do
3:    for all $b \in B$ do
4:       if $\neg$ belongToSameGroup( a, b) then
5:          $c \leftarrow$ combine( a, b )
6:          if isCompositionAllowed( c, BlackList ) $\wedge\ c \notin new$ then
7:             $new \leftarrow new \cup c$
8:          end if
9:       end if
10:   end for
11: end for
12: return new
```

Section 3.3. Combinations that are not pruned away are added to a new set *(line 7)* which is then returned *(line 12)* as a new generation of features.

The assessment for the evolution progress is shown in Algorithm 3. In *line 1* the new feature's value is compared against its first parent. The value of ϵ is domain-specific and is added to the feature's value to control the level of improvement needed for the feature to evolve further. Features that do not improve upon their parent are pruned (Lineair Tree Pruning) by adding them to the *BlackList (line 2)*, thus preventing them from occurring in future evolution sets. Features that surpass their parent, are returned *(line 11)*. Lines *3-8* in the algorithm are a part of the *linear tree pruning* method to be discussed in Section 3.4. Thereafter Section 3.5 discusses threshold pruning.

3.3 Groups

Some features are not compatible in a sense that combining them is meaningless in the context of the game at hand, e.g., in chess: *capture the king* or *move a pawn and rook*. Features can thus be put in advance into logical groups, where

Algorithm 3. *filterFeature(newFeature, ref BlackList)*

```
1: if $newFeature_{value} < newFeature_{firstParentValue} + \epsilon$ then
2:    $BlackList \leftarrow BlackList \cup newFeature$
3:    if UseLinearTreePruning then
4:       for all $child \in$ getChildren( $newFeature_{secondParent}$ ) do
5:          $banned \leftarrow$ combine( $newFeature_{firstParent}$, child )
6:          $BlackList \leftarrow BlackList \cup banned$
7:       end for
8:    end if
9:    return {}
10: else
11:   return newFeature
12: end if
```

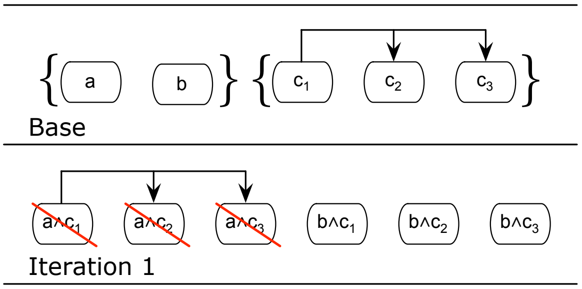

Fig. 3. Linear Tree Pruning example

combining features within a group is not allowed (*line 4* in Algorithm 2). This prevents many useless combinations from being formed. As an example, a set of six features divided into two equal groups, creates 63 different feature combinations without groups, but only 15 if groups are used. Combined features inherit the groups of their parents, making them belong to more than one group.

3.4 Linear Tree Pruning

Features within a group can form a natural hierarchy, where some features are subsets of others. This tree hierarchy can be used to predict the outcome of future evaluations and prune off those that are expected to be inferior. The logic is that if a new combination formed with a parent feature is pruned, then other combinations with that feature's children can also be eliminated.

Figure 3 shows an example of linear tree pruning. The *Base* set has been divided into two groups, and the second group has a hierarchy with c_2 and c_3 being children of c_1. In the first iteration GF creates all two-wise combinations which are then evaluated. Assuming the combination $a \wedge c_1$ is pruned, then if linear tree pruning is used the features $a \wedge c_2$ and $a \wedge c_3$ would also be pruned.

This method cannot guarantee that interesting combinations would not be pruned. The reason is that combinations with a highly frequent parent feature might extend too aggressively, whereas a combination with its less frequent children might not. The method is thus optional, as shown in Algorithm 3 in *lines 3-8*. It retrieves all the children of the second parent feature *(line 4)* and creates a new combination with the first parent and each of the retrieved children features *(line 5)*, which are then added to the *BlackList (line 6)*.

3.5 Threshold Pruning

Threshold pruning is an attractive optional pruning method available for GF. It removes all features from the *Base* set that are below a provided quality threshold, determining those features immediately as disadvantageous and therefore not eligible to participate in the evolution process (*lines 7-13* in Algorithm 1).

This is potentially a very effective pruning method as it reduces the exponential growth of the number of combined features at all iterative levels, but at the risk of wrongfully pruning away potentially good candidates. The risk can be lessened by marking which features can be safely pruned; this however requires knowledge of the search domain. A sensible choice for the threshold parameter δ *(line 9)* would be to approximate it around the value of the *Neutral* feature.

3.6 Fitness Evaluation

The true quality of a search-extension feature can be found only by using it in actual game-play. However, playing games is time-consuming, so instead we use a suite of selected test positions where the best move is known. Information about the feature's effect on the search is collected: number of solved positions, mean iteration depth, mean height, and feature's frequency in the search. Various methods to evaluate the feature's quality can be formed based on the gathered information, but we choose to use straightforwardly the number of solved positions (i.e., best move played) as it directly measures a feature's effectiveness.

4 Empirical Results

We empirically evaluate the GF method by discovering search-extension features for the game Breakthrough. The experiments were run on a Linux CentOS 5 Intel(R) Xeon (TM) 3.00GHz CPU machine with 2GB of memory.

4.1 Breakthrough

Breakthrough [16] is a two-person perfect-information game created by Dan Troyka in 2001. The game is played on a chess-board where each player has 16 pawn-like pieces that fill the two front and back rows of the board. The objective of the game is to break through the opponent's ranks and advance a piece to the opponents back rank. Despite its simple rules, which are given below, the game requires a sophisticated strategy to play at an expert level.

1. Players' two back rows are filled with their pieces at the start of the game.
2. Players choose which side starts (our program assumes that White starts).
3. Players alternate moving a piece.
   (a) One square forward or diagonally-forward to unoccupied squares.
   (b) One square diagonally-forward to a square containing an opponent's piece, capturing opponent's piece and removing it from the board.
4. Capture moves are not forced.
5. A game ends when a player's piece reaches the opponent's back rank.

Table 1. Description of Breakthrough's base features

Feature	Feature Description
Ud	Moved piece is not threatened on destination square.
PP	Piece's direction is unhindered towards opponent's back rank.
Rc	Capture previously moved piece.
C	Capture opponents piece.
Ms	Majority of squares surrounding moved piece is occupied by players pieces, forming a greater mass.
Rdb	Players half of the board.
RdBb	First and second rank of the board.
RdBt	Third and fourth rank of the board.
Rdt	Opponents half of the board.
RdTb	Fifth and sixth rank of the board.
RdTt	Seventh and eighth rank of the board.
Edg	The board's edges, columns *a, b, g,* and *h.*
Mr	The board's middle, columns *c, d, e,* and *f.*
Udp	Prepare to move around opponent's piece. Piece is not threatened and standing opposite opponent's piece.
Bv2	Block opponent's advancement by placing piece in front of it, creating a vertical defensive line of two pieces.

4.2 Experiments Setup

A description of the atomic features forming GF's *Base* set is shown in Table 1. Each feature belongs to its own group except the following larger groups: {*Rdb, RdBb, RdBt*}, {*Rdt, RdTb, RdTt*}, and {*Edg, Mr*}. Two of these have a tree-hierarchy: *RdBb* and *RdBt* are *Rdb's* children, and *RdTb* and *RdTt* are *Rdt's* children. The features are evaluated using a fixed fractional-ply value of 0.5, chosen somewhat arbitrary although such that it is neither too conservative nor too aggressive. Each feature was evaluated using our Breakthrough program searching 500,000 nodes per search, which corresponds to approximately 5-ply search. The program uses $\alpha\beta$ search with iterative deepening, and a simple but fairly effective heuristic based on material and bonuses for advanced non-attacked pieces. GF's evolution parameter ϵ is set to 3 to compensate for fluctuating evaluations, and the threshold pruning parameter δ equals to the *Neutral* feature's value. As there exists no standard position test-suite for Breakthrough we created a set of 302 positions which were picked from a game of self-play where the terminal state could be reached in 7 plies. This allowed us to identify unambiguously the best move, but has the drawback of all the positions being taken from the endgame. Three programs using different heuristic evaluations were used to obtain a variety of endgame positions.

4.3 Feature Evolution Results

Four different instantiations of GF were evaluated. The results are shown in Table 2 where *Default* disregards previously presented grouping of features, plac-

Table 2. Gradual focus evaluations in Breakthrough

Method type	Hours	Evaluations #	% of $\mathcal{P}$	Overlooked combinations
Power Set	-	32,768	100%	
Default	30.6	138	0.42%	
Default + G	24.9	112	0.34%	0
Default + G + LTP	23.7	105	0.32%	0
Default + G + LTP + TP	5.8	27	0.08%	5

ing each feature in a group of its own. The GF's enhancements, *Groups* (G), *Linear tree pruning* (LTP), and *Threshold pruning* (TP), are then cumulatively added using the previously described grouping of features. The calculated result of a power-set's feature expansion are also shown.

As can be seen by the *Default* instance, GF reduces the number of generated feature combinations immensely compared to the brute-force power-set method, and without overlooking any interesting combinations. Thirteen of these evaluations are incompatible combinations that can never occur in the game, which were prevented in the *G* instance. Adding *LTP* improves the pruning slightly further without overlooking any previously interesting combinations. Additionally, *TP* further reduces the number of evaluations substantially, but at the cost of overlooking five of *Default*'s top ten most interesting combinations, *Ud-PP-Rdt*, *Ud-Rdt*, *PP-Rdt*, *PP-Rdt-Edg*, and *Rc-Rdb*.

4.4 Precision and Recall

GF's findings were compared with the complete set of all one, two, and tree-wise combinations, in all 377 features. The resulting top 25 features are shown in Table 3 along with the number of positions they solve and their first parent. Features written in italics are less effective descendants of already discovered features and as such redundant as their benefits are already obtained by the use of their parent. Ignoring these features leaves only 11 of the original 25 features.

The top features that GF returns are exactly these 11 features. Thus, in this domain, GF offers both perfect precision (number of correctly identified features) and perfect recall (how large portion of the interesting features were discovered).

4.5 Tournament Results

The top-ten features suggested by GF as interesting were also evaluated through self-play. Table 4 shows the results, with the last two columns contrasting the feature frequency when (1) searching the test-suite and (2) in actual games.

All but three of the features lead to an improved play. The features *Rc-Rdb* and *PP-Rdt-Edg* have a little effect on the game, which can in part be explained by their low frequency. However, that alone is a not a sufficient explanation as *PP-RdTt* with even a lower frequency is doing well. That feature is extending

Table 3. Tree-Wise Combinations in Breakthrough

Feature	Solved	Parent	Feature	Solved	Parent
★ Ud-RdTt	235	Ud	*PP-Edg-RdTt*	*115*	*PP-RdTt*
Ud-PP-RdTt	*224*	*Ud-RdTt*	*Ud-Ms-RdTt*	*114*	*Ud-RdTt*
★ PP-RdTt	211	PP	*Ud-PP-RdTb*	*111*	*Ud-PP*
★ RdTt	202	–	*Ms-RdTt*	*110*	*RdTt*
★ Ud-PP-Rdt	173	Ud-PP	*Ud-PP-Mr*	*109*	*Ud-PP*
★ Ud-PP	155	Ud	★ PP-Rdt-Edg	108	PP-Rdt
Ud-Edg-RdTt	*149*	*Ud-RdTt*	*Ud-Rdt-Mr*	*105*	*Ud-Rdt*
Edg-RdTt	*142*	*RdTt*	★ Ud	102	–
Ud-RdTt-Mr	*141*	*Ud-RdTt*	*Ud-Rdt-Edg*	*102*	*Ud-Rdt*
PP-RdTt-Mr	*140*	*PP-RdTt*	★ PP-Rdt	98	PP
RdTt-Mr	*138*	*RdTt*	★ PP-Edg	97	PP
★ Ud-Rdt	125	Ud	★ Rc-Rdb	96	Rc
Ud-PP-Edg	*122*	*Ud-PP*			

Table 4. Features' result in Breakthrough

Feature	Games #	Wins %	Conf. Int.	Frequency Suite	Frequency Games
Ud-Rdt	2400	58.17%	±1.97	15.57%	4.71%
PP-RdTt	2400	57.04%	±1.98	2.87%	0.87%
RdTt	2400	56.54%	±1.98	8.75%	1.91%
Ud-RdTt	2400	55.96%	±1.99	7.76%	1.90%
Ud-PP	2400	53.92%	±1.99	8.44%	2.31%
Ud-PP-Rdt	2400	53.50%	±2.00	6.94%	1.99%
PP-Rdt	2400	52.03%	±2.00	12.33%	3.92%
Rc-Rdb	2400	49.29%	±2.00	1.55%	1.48%
PP-Rdt-Edg	2400	49.13%	±2.00	3.18%	1.18%
Ud	2400	46.46%	±2.00	32.18%	43.37%

only on safe pawn moves just about to reach the back rank and thus, even though infrequent, almost always results in a more accurate evaluation score. In contrast, the *Ud* feature has a negative effect on in-game performance, whereas it was slightly beneficial on the test-suite. We see that this type of extension is substantially more frequent in actual game play than in the test-suite, which might be enough to tilt the balance. This could probably be avoided by using a more diverse test-suite containing start, middle, and endgame positions.

5 Conclusions

We introduced a new method for learning search-extension features, called *Gradual Focus*. It iteratively creates new refined features by combining atomic features from a base set with the $\wedge$ operator, using various merit-based pruning techniques to select which features to elaborate. We evaluated the method in the game Breakthrough, where there exists little domain knowledge of what makes up good move categories to extend on. The method learned several promising search-extensions features from a suite of test positions. Moreover, it required several orders of magnitude less time than a brute-force approach while demonstrating both an excellent precision and recall. The learned features, when used in regular game play, significantly improved our program's playing strength. Also, the method does not require the learned features necessarily to be move categories, and GF could thus be used for other search-control features as well.

There is still much scope for improvements and we view this work as a first step in exploring automatic discovery of search-control features. In particular, currently we use the number of solved positions as the only indicator of a feature's quality. However, by also monitoring other statistics when evaluating a feature, such as its frequency in the search, one could pinpoint promising evolution paths more intelligently. For example, a feature that is much less frequent than another might be preferred even if it solves a slightly less number of test positions. A second issue is that we evaluate all features using a fixed FP value, which undeniably excludes the discovery of potentially useful features (we have observed that a feature's quality is quite sensitive to its FP value). Thus, combining learning of features with methods for learning FP values is a worthwhile avenue for future research. Finally, it would be interesting to explore the method in other game domains, and we have started preliminary work in chess.

Acknowledgments

This research was supported by grants from The Icelandic Centre for Research (RANNIS) and by a Marie Curie Fellowship of the European Community programme Structuring the ERA under contract MIRG-CT-2005-017284. We also thank the anonymous reviewers for their comments.

References

1. Anantharaman, T.S., Campbell, M.S., Hsu, F.: Singular extensions: adding selectivity to brute-force searching. Artificial Intelligence 43(1), 99–109 (1990)
2. Beal, D.F., Smith, M.C.: Quantification of search extension benefits. ICCA Journal 8(4), 205–218 (1995)
3. Hyatt, R.M.: Crafty. A chess program (1996) (March 27, 2008), `ftp://ftp.cis.uab.edu/pub/hyatt`
4. Levy, D., Broughton, D., Taylor, M.: The SEX algorithm in computer chess. ICCA Journal 12(1), 10–21 (1989)
5. Tsuruoka, Y., Yokoyama, D., Chikayama, T.: Game-tree search algorithm based on realization probability. ICGA Journal 25(3), 146–153 (2002)
6. Winands, M.H.M., Björnsson, Y.: Enhanced realization probability search. New Mathematics and Natural Computation 4(3), 329–342 (2008)

7. Björnsson, Y.: Selective Depth-First Game-Tree Search. Phd dissertation, University of Alberta (2002)
8. Björnsson, Y., Marsland, T.A.: Learning extension parameters in game-tree search. Information Sciences 154(3-4), 95–118 (2003)
9. Kocsis, L., Szepesvári, C., Winands, M.H.M.: RSPSA: Enhanced Parameter Optimization in Games. In: van den Herik, H.J., Hsu, S.-C., Hsu, T.-s., Donkers, H.H.L.M(J.) (eds.) CG 2005. LNCS, vol. 4250, pp. 39–56. Springer, Heidelberg (2006)
10. Fawcett, T.E., Utgoff, P.E.: Automatic feature generation for problem solving systems. In: Intern. Conf. on Machine Learning (ICML), pp. 144–153 (1992)
11. Kaneko, T., Yamaguchi, K., Kawai, S.: Automated identification of patterns in evaluation functions. In: Advances in Computer Games, vol. 10, pp. 279–298 (2003)
12. Buro, M.: From simple features to sophisticated evaluation functions. In: van den Herik, H.J., Iida, H. (eds.) CG 1998. LNCS, vol. 1558, pp. 126–145. Springer, Heidelberg (1999)
13. Buro, M.: Experiments with Multi-ProbCut and a new high-quality evaluation function for Othello. In: Games in AI Research, pp. 77–96 (1999)
14. Sturtevant, N.R., White, A.M.: Feature construction for reinforcement learning in hearts. In: van den Herik, H.J., Ciancarini, P., Donkers, H.H.L.M(J.) (eds.) CG 2006. LNCS, vol. 4630, pp. 122–134. Springer, Heidelberg (2007)
15. Finkelstein, L., Markovitch, S.: Learning to play chess selectively by acquiring move patterns. ICCA Journal 21(2), 100–119 (1998)
16. Handscomb, K.: 8×8 game design competition: The winning game: Breakthrough ... and two other favorites. Abstract Games Magazine 7 (2001)