REYKJAVÍK UNIVERSITY
HÁSKÓLINN Í REYKJAVÍK

Gradual Focus: A Method for Automated Feature Discovery in Selective Search

Pálmi Skowronski

Master of Science
January 2009

Reykjavík University - School of Computer Science

M.Sc. Thesis

REYKJAVÍK UNIVERSITY
HÁSKÓLINN Í REYKJAVÍK

Gradual Focus: A Method for Automated Feature Discovery in Selective Search

by

Pálmi Skowronski

Thesis submitted to the School of Computer Science
at Reykjavík University in partial fulfillment
of the requirements for the degree of
Master of Science

January 2009

Gradual Focus: A Method for Automated Feature Discovery in Selective Search

by

Pálmi Skowronski

January 2009

Abstract

The aim of selective search in adversary board games is to concentrate the search capacity on important lines of play, as to mimic the cognitive approach of humans. This is achieved by placing available moves into move categories, where interesting categories are examined more closely while less interesting ones are terminated early. One of the main challenges with selective search is designing effective move categories (features), which is a manual trial and error task that requires both intuition and expert human knowledge. Automating this task potentially enables the discovery of both more complex and more effective move categories. In this work we introduce *Gradual Focus*, an algorithm for automatically discovering interesting move categories for selective search. The algorithm iteratively creates new more refined move categories by combining mutually exclusive features from an atomic feature set. Each iteration selectively creates more detailed move categories. This enables the assessment of a move categories' evolutionary progress, making Gradual Focus a merit driven method which continues to evolve move categories while it is still beneficial. Empirical data is presented for the games *Breakthrough* and chess showing that Gradual Focus looks at two orders of magnitude less than a brute force method would do.

Gradual Focus: Sjálfvirk uppgötvun á valbundnum leitar sérkennum

eftir

Pálmi Skowronski

Janúar 2009

Útdráttur

Tilgangur valbundinnar andstæðings leitar (*e. selective adversary search*) í hefðbundnum leikjum á borð við skák er að stýra leitinni á þann hátt að áhugaverðir leikir eru skoðaðir nánar, sem er svipuð nálgun og við mennirnir notum til að leysa sambærileg vandamál. Þetta er útfært þannig að allar mögulegar aðgerðir (*e. moves*) eru flokkaðar í aðgerðaflokka eftir því hversu áhugaverðar aðgerðirnar eru. Með þessum hætti getur leitin eytt meiri tíma í að skoða áhugaverðar aðgerðir og minni tíma í að skoða óáhugaverðar aðgerðir. Það að skilgreina áhugaverða aðgerðaflokka er bæði flókið og villugjarnt ferli þar sem um er að ræða handvirkt ferli sem krefst sérfræði þekkingar mannanna. Ef ferlið yrði aftur á móti sjálfvirkt mundu opnast nýir möguleikar í að mynda flóknari og áhrifaríkari aðgerðaflokka en áður. Í þessari ritgerð lýsum við algrímnum *Gradual Focus* sem uppgötvar sjálfkrafa áhugaverða aðgerðaflokka fyrir valbundna leit. Með honum eru nýir og sértækari aðgerðaflokkar búnir til í ítruðu ferli þar sem aðgerðaflokkar úr síðustu ítrun eru sameinaðir við hóp af fyrirfram skilgreindum grunn (*e. atomic*) aðgerðaflokkum sem útiloka hvor aðra. Með þessu móti er hægt að meta þróun aðgerðaflokkanna sem gerir algrímnum kleift að þróa nýja aðgerðaflokka þangað til þeir eru ekki lengur gagnlegir. Gæði algrímsins eru sýnd í leikjunum *Breakthrough* og skák þar sem það kemur í ljós að Gradual Focus skoðar einungis tvær stærðargráður af aðgerðaflokkum til samanburðar við allar mögulegar samsetningar.

To Auður and our three beautifully gifted children, Eva María, Sara Lind, and Matthías.
Your love and support means the world to me.

Acknowledgements

First and foremost I would like to thank my supervisor Dr. Yngvi Björnsson for his guidance and thoughts, without him this work would not have been possible. It has been a true privilege to have worked with him. I would also like to give special thanks to Hilmar Finnsson, a friend and colleague, for his support and our numerous discussions about informative search over these last years, as well as proof reading this thesis. I am also grateful to Dr. Mark H.M. Winands for lending his program MIA, providing both answers as well as more questions.

I would also like to thank my friends and family for their support through the years, especially Eyjólfur Sigurðsson for his thoughts about statistical analysis.

Greatest thanks of all I give to Auður Hermannsdóttir for her love and support, as well as countless proof readings. I truly thank you because I could never have done this without you!

Contents

List of Figures

List of Tables

Chapter 1

Introduction

If I have ever made any valuable discoveries, it has been owing more to patient attention, than to any other talent.
Isaac Newton (1642 - 1727)

Enabling computers to play sophisticated board games such as chess is one of the oldest research areas in artificial intelligence. In 1950 Claude Shannon wrote a seminal paper (Shannon, 1950) where he successfully described two different search strategies for game playing programs called type A and B. Type A programs employ a brute force search, where all continuations are searched to a fixed depth. Type B programs try to mimic the playing abilities of humans by selectively examining "promising" and unstable lines of play.

Most chess programs pursued Shannon's strategy of type B by generating "plausible" moves until the mid 1970. At that time brute-force programs managed to reach depth of 5 plies and higher and with the introduction of iterative deepening, brute-force programs consistently won over plausible-move generation programs, creating a new trend in computer chess where knowledge and intuition for plausible move selection was next to ignored. Although regularly there would be some programs that tried to implement some level of knowledge based selective search, it was not until about 1990 with the emergence of null-move pruning (Beal, 1989, 1990; Goetsch & Campbell, 1990; Donninger, 1993) that the importance of speculative enhancements was again recognized, which has been the dominant approach since, in chess as well as other adversary games.

The purpose of selective enhancements is to enable the search to concentrate on the right areas of the game tree, just as humans would. Successfully constructing these schemas is a difficult multi dimensional task. One of these is choosing effective move categories to control the search, which is commonly done manually. This can be a tedious and time consuming trial and error prone process which requires intuition and expert human knowledge of the domain. Automating this process potentially enables more complex and efficient move categories to be created.

This thesis describes an automated feature discovery algorithm for selective search, called *Gradual Focus*. The method iteratively creates new move categories by combining mutually exclusive feature from a given atomic feature set. Each iteration selectively creates more refined move categories. This enables the assessment of a move categories' evolutionary progress, making Gradual Focus a merit driven method which continues to evolve move categories while still being beneficial.

In this work we use the Gradual Focus method to discover search extensions features for the game of *Breakthrough* as well as finding known extension features in chess. In Breakthrough the method successfully discovered interesting features that improved a game-playing programs playing strength, significantly.

The main contributions of this thesis are an introduction of a method to automatically discover search-control features as well as a set of effective extension features for the game of Breakthrough.

The structure of the thesis is as follows: Chapter 2 introduces the basics of adversary search, followed by a more detailed description of Selective Search and its effects on the search tree. Chapter 3 gives a detailed description of the Gradual Focus algorithm and its enhancements. In Chapter 4 experimental results are presented for the games Breakthrough and chess. Finally, conclusions and discussion of future work is given in Chapter 5.

Chapter 2

Background

All truths are easy to understand once they are discovered; the point is to discover them.

Galileo Galilei (1564 - 1642)

In this chapter we explain *selective search* in adversary games. The emphasis is placed on *search extensions*, since they are the main focus of the thesis.

2.1 Search Overview

The objective of adversary search is to come up with a move decision in games played against an opponent, such as Chess and Othello.

2.1.1 The Search Tree

The search space can be impicitly represented as a tree where each node is a state and the edges between states represent legal moves. A move made by a player is also referred to as a *ply*. The search tree is traversed in a depth-first manner from left to right. When a leaf node is reached, it is evaluated and assigned a numeric value indicating its quality. The larger the number, the higher the quality.

The value is passed up to the leaf's parent, that in turn passes up the best possible value of all its children which is not always the largest. This process is called *backing-up* the value and continues until the best possible value has been propagated up to the root node. Therefore an optimal path through the tree, from root to leaf, can be plotted by choosing

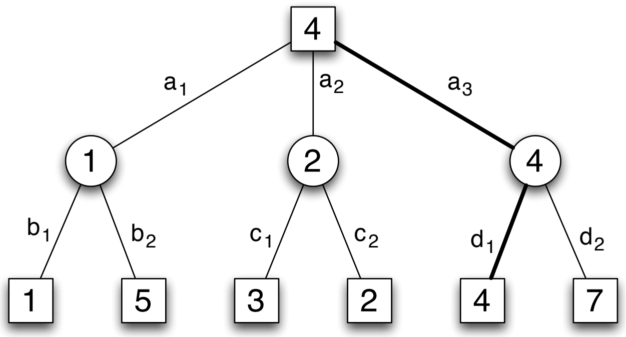


Figure 2.1: Optimal path throug a 2-ply MiniMax search tree

the next child, from current node, with the highest backed-up value. This can be seen in Figure 2.1 where the value *4* has been backed up to the root revealing an optimal path via moves a_2 and d_1.

In most games, the game tree is too deep, making true terminal nodes unreachable. Instead, the search relies on a limited look ahead for evaluating the *"best"* move. The deeper the look ahead, the more accurate the best move decision will be on average. The average number of children for each node in a given domain is called the *branching factor*. As each level of the tree is expanded, the number of nodes increases exponentially in proportion to the branching factor. This puts limitations to the look ahead depth, making the branching factor one of the main indicators of the search domain's complexity.

It is evident that for most games an exhaustive brute-force search is impractical as the number of nodes explored would be astronomical.

2.1.2 MiniMax Algorithm

The basic method for traversing game trees in adversary search is the *Minimax search* (Neuman & Morgenstern, 1944). The first player, known as *Max*, tries to maximize his gain by always choosing the move with the highest backed up value, while the other player, known as *Min*, tries to minimize the other player's gain by choosing the move with the lowest backed value. Given that the game is a *zero sum* game[1], only one player can win, *Min* increases his odds of winning by decreasing *Max's* chances of winning.

Figure 2.2 shows an example of a MiniMax tree where square nodes represent Max's turn to move and circle nodes represent Min's turn to move. The tree is traversed depth-first

[1] The term *Zero sum* describes the property that the sum of the scores for both players, always equals zero.

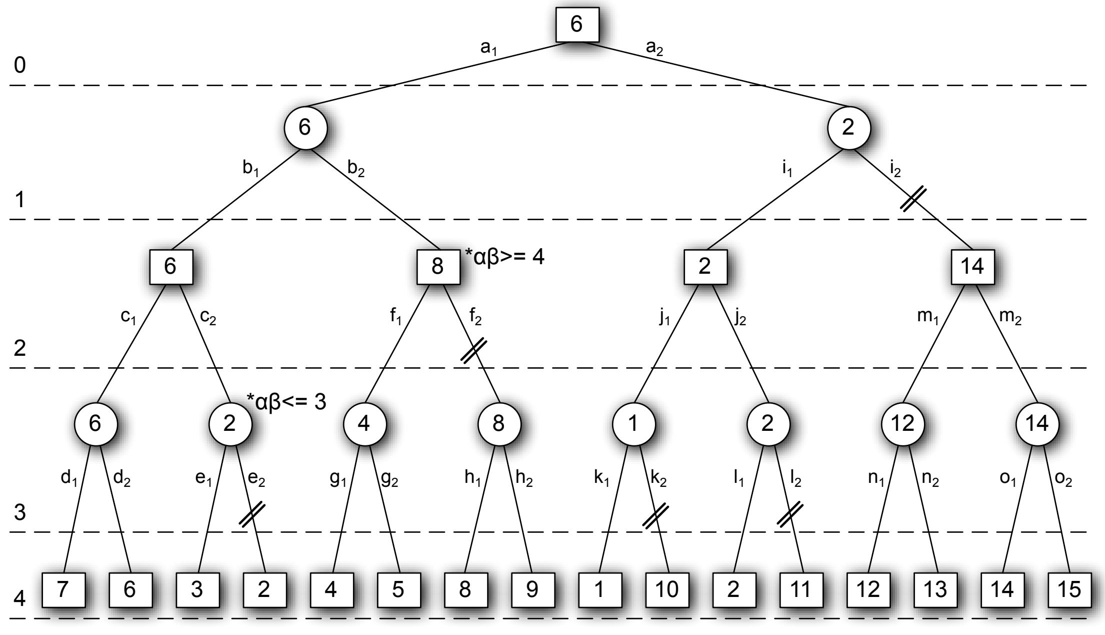


Figure 2.2: 4-ply MiniMax tree example
(Korf, 2007)

from left to right, expanding moves a_1, b_1, c_1, and d_1 where the leaf node is evaluated as 7, then move d_2 is expanded where the next leaf is evaluated as 6. At this point Min backs up the lowest possible value, 6. Next move expanded is c_2 where the same process is performed, backing up the lowest value 2. Next Max backs up the largest available value, 6. When all the sub-trees have been evaluated in this manner, Max at the root will back up the largest value, in this case 6, indicating that a_1 is the best move.

Although the minimax algorithm guarantees finding the best move based on the underlying evaluation function, it does so by traversing through the entire tree. This is unnecessary because only a so called *minimal* or *critical tree* needs to be explored to determine the best move. Continuations that are not part of the critical tree have no chance of influencing the outcome of the search, making their expansion unnecessary.

Moves and their continuations that are not a part of the critical tree in Figure 2.2, are shown with cutoff markers at moves e_2, f_2, k_2, l_2 and i_2. A detailed description of identifying the critical tree will be given in the next section.

2.1.3 $\alpha\beta$ Search

$\alpha\beta$ search (Knuth & Moore, 1975) is a widely practiced and proven search method that takes advantage of the fact that only the critical tree needs to be searched to find the best

Algorithm 1 *$\alpha\beta$(state, α, β, depth)*

```
1: if depth $\leq 0$ or isTerminal( state ) then
2:     return evaluate( state )
3: end if
4: best $\leftarrow \alpha$
5: moveList $\leftarrow$ generateMoves( state )
6: while move $\leftarrow$ nextMove( moveList ) do
7:     makeMove( state, move )
8:     value $= -\alpha\beta($ state, $-\beta, -\alpha, depth - 1)$
9:     retractMove( state, move )
10:    if value > best then
11:        best $\leftarrow$ value
12:        $\alpha \leftarrow$ value
13:        if best $\geq \beta$ then
14:            return best
15:        end if
16:    end if
17: end while
18: return best
```

move. This is done by creating a search window with two bounds that define the possible range of the best value/move, lower bound α and upper bound β. The scope between the α and β values is known as the *$\alpha\beta$ window*. Search continuations with backed up values outside the $\alpha\beta$ window, *fail-low* if less than α or *fail-high* if greater or equal than β, are not explored because they are not a part of the critical tree and as such have no chance of affecting the move decision at the root. This is called *pruning* as it cuts off a tree's branches. During the search the $\alpha\beta$ window is narrowed as new indications of the best move are found. By narrowing the $\alpha\beta$ window the chances of pruning increases, enabling a deeper search with the time saved, resulting in an increase in the search's quality.

The $\alpha\beta$ algorithm, shown as Algorithm 1, is a recursive algorithm that reduces the $depth$ by one for each recursion *(line 8)* until a terminal node or the tree's maximum height is reached *(line 1)*. For each recursion the α and β parameters are switched and negated. This and the negation of the returned evaluation are part of the *negamax* formulation (Reinefeld, 1983) which excludes special distinction between Min and Max by making both players back up the best value from the perspective of the player to move. The narrowing of the $\alpha\beta$ window can be seen in *lines 10-12*, as α is replaced by the new best value. If the value is greater than or equal to the upper bound β *(line 13)*, that node's remaining subtrees are pruned off, immediately returning the best value. Because of the negamax formulation the algorithm needs only consider fail-high instances when pruning off sub-trees.

Applying Algorithm 1 on Figure 2.2's search tree will return the same solution but by expanding only about half of the tree's nodes, as moves e_2,f_2, k_2, l_2, and i_2 and their subtrees are pruned off. Though the pruning might change the $\alpha\beta$ bounds for some nodes, in this case two nodes marked "*", it will never change the outcome of the search.

Over the years many enhancements have been added to $\alpha\beta$ search to increase its performance. The best known enhancements are probably *iterative deepening* (Slate & Atkin, 1977), *move ordering* (Slate & Atkin, 1977; Schaeffer, 1989), *null window search* (Marsland, 1983; Reinefeld, 1983), *transposition table* (Slate & Atkin, 1977), and *quiescence search* (Shannon, 1950). Most of these methods are designed to improve $\alpha\beta$ search's pruning mechanism so that the search tree will be as close as possible to the critical tree.

2.1.4 Null Window Search

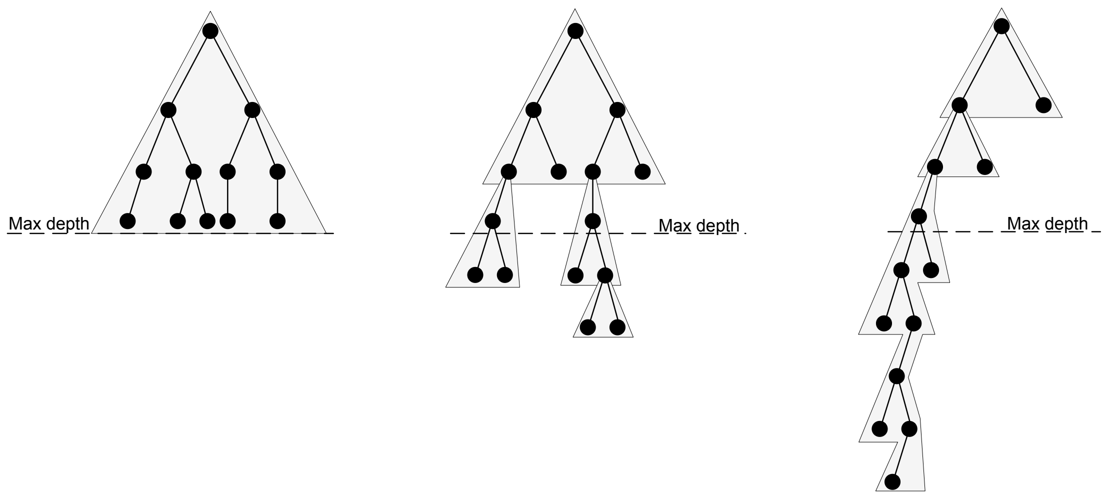
The smaller the $\alpha\beta$ window is the more continuations will be pruned off. In $\alpha\beta$ search the window is initialized with $-\infty$,∞ and gradually narrowed as the search continues. This is done because the best value is not known and we don't want to run the risk of overseeing it by choosing a smaller search window that does not contain the best value.

Null window search (Marsland, 1983; Reinefeld, 1983), which is a $\alpha\beta$ variation algorithm, approaches this by performing only a regular search with full $\alpha\beta$ bounds for the first move, expecting it to be the best move. All other moves are searched with the smallest possible $\alpha\beta$ window (*null window*) around the backed up value of the first move, *-(value+1),-value*. The narrow search is expecting to fail-low and thereby proving that the first move is the best. If one of the alternative moves proves the opposite, i.e. *value* $> \alpha$, its true value is determined by re-searching the move with a larger search window, establishing it as the new best move.

The overhead of performing re-searches is not significant in comparison with the increase in pruned instances, making null window search a standard in adversary search. Re-search instances can be decreased greatly with other standardized enhancements that make promising moves expanded early, e.g. move ordering and transposition table.

2.2 Selective Search

The capability of games programs to calculate deep lines of play exceeds that of most human capabilities. Despite this, it is not given that a game-playing program will win a



(a) Even Search Tree (b) Varible Depth Search (c) Agressive Extension

Figure 2.3: Search Tree Expantion Types

match against a human. Humans have an unexplained ability to ”see” what part of the game-tree matters the most and focus on examining only relevant moves. While computer’s conventional traversal of a search trees considers all moves to be of equal interest, reducing the search depth by one for each move *(i.e. line 8 in Algorithm 1)*. This results in an even expansion of the search tree to the tree’s max depth, as shown in Figure 2.3(a).

Of course all moves are not of equal interest. The goal of *selective search* is to guide the search to focus in areas of the search tree that matter the most, as to mimic the abilities of humans. This is done by examining interesting moves more closely while less interesting moves are terminated early. A move is deemed interesting if expanding it has a good potential for changing the backed up value significantly. Other moves that give no new insight to the search are of little interest. This creates an unbalanced *variable depth* search tree where beneficial continuations grow beyond the tree’s max depth while potentially unfavorable paths are terminated early, not reaching the tree’s max depth. See Figure 2.3(b).

Selective search is categorised into two groups, *speculative pruning* and *search extensions*. Speculative pruning applies to the case of terminating uninteresting moves early while search extensions apply to the former, where interesting moves are examined more closely.

As stated in Section 2.1.1, a search tree grows exponentially in accordance with its branching factor. It is evident that domains with a large branching factor require a greater effort to reach depths needed for adequate level of play. For these domains it becomes necessary to focus the search on the right areas.

Most selective search schemas use intensive knowledge of the search space to achieve maximum results. This makes the schemas domain dependent and hinders their usage on different games. Games with similar properties are likely to gain from the same selective search schema, but equal efficiency cannot be guaranteed, as the games' criteria might not be the same.

In Chapter 3 we propose a method for learning a selective search schema automatically, thereby circumventing many of the known difficulties.

2.2.1 Speculative Pruning

The objective of speculative pruning, also known as *forward pruning* or *search reductions*, is to prune off search continuations that are estimated to be irrelevant, within the critical tree as well as outside. The notion is that a shallower search can sufficiently predict the outcome of a full depth search without significantly distorting the quality of the search. Because this category of methods may prune off parts of the critical tree, they run the risk of wrongfully pruning away good moves that a deeper search would otherwise reveal. This property is part of what is called the *horizon effect*. The risk of overseeing the best possible move is of course grave, but if this occurs only rarely the time saved is worth the risk as it is used to search interesting moves more closely, increasing the overall quality of the search.

Null move pruning (Beal, 1989, 1990; Goetsch & Campbell, 1990; Donninger, 1993) is probably the most used speculative pruning method. Other methods worth mentioning are *razoring* (Birmingham & Kent, 1988), *futility pruning* (Slate & Atkin, 1977), *late move reductions* (Romstad, 2006), *multi-cut* (Björnsson & Marsland, 1998), and *AEL pruning* (Heinz, 2000) which combines three enhanced methods, *adaptive null move pruning* (Heinz, 1999), *extended futility pruning* (Heinz, 1998), and *limited razoring* (Heinz, 1998), and has shown good performance in the domain of chess.

2.2.2 Search Extentions

The focus of search-extension methods is to increase the quality of the search by examining interesting lines more deeply than others. This can be done by not reducing the depth for interesting moves *(i.e.* $depth - 1$ *decrement, line 8 in Algorithm 1).* This enables the height of the tree to grow beyond the max depth for lines of play with interesting moves, which introduces the concept of *aggressive extensions*. If the search were to extend too aggressively, it might follow a single line of play without the equivalent quality gain, wasting time by leaving large parts of the search tree unexamined, thereby missing out on other interesting lines of play. See Figure 2.3(c).

Search extensions are crucial in guiding the search in the right direction, but can be tedious to design. It requires the discovery of move categories, known as *features*, *labels* or *classes*, that will contribute the most to the search's advance and tuning the categories extension schema, while being careful not to extend too aggressively. The schema's computation overhead must also be considered, making a successful design far from easy.

As an example of a successful search extensions in the domain of chess, the program Fruit 2.1 by Fabien Letouzey (Letouzey, 2005) extends all moves of the categories: *single reply*, *recapture*, *pawn promotion* and *checking moves*, by a reduction of $depth - 0$. Though the same extensions might also be applied in chess like games, i.e. games with similar properties and goals, they are not guaranteed to be as efficient, as the game criteria might not be the same. This emphasizes previous discussion on domain independence in Chapter 2.2.

2.2.3 Fractional Ply

Fractional ply (Hyatt, 1996; Levy, Broughton, & Taylor, 1989), is a *variable depth* extension that grades moves with fractional ply values (FP values) to control the search strategy. The FP value is a real number that indicates the depth decrement for a given move, instead of the conventional one ply reduction $depth - 1$ *(i.e. line 8 in Algorithm 1)*, it is replaced by $depth - fp$ where fp is the move's decrement value. This means that interesting moves receive a FP value between 0 and 1, regular moves receive a FP value of 1, and uninteresting moves get a FP value > 1. The assignment of FP-values to moves are done online by classifying each move into a category with pre-calculated FP-values.

Figure 2.4 shows the expansion of a search tree where moves are decremented based on a labels FP value.

The online classification of moves is a clear drawback, it requires extra computation time which makes the search slower. While a complicated classification method might give more insight into the moves quality and the accuracy of FP values, the extra time overhead increases as well, gradually consuming the advantage of fractional ply.

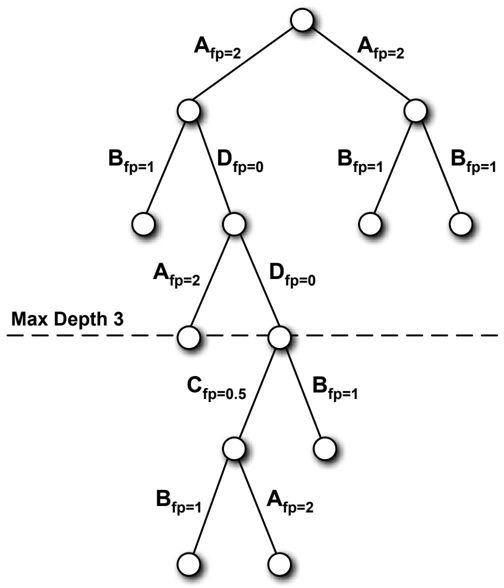


Figure 2.4: Fractional ply example

Another challenge with fractional ply is tuning the FP values, which is a time consuming task largely based on trial and error, making the discovery of descriptive FP values for large and complex categories impractical. To improve this process Björnsson and Marsland presented a framework based on *gradient-descent* to tune FP values automatically (Björnsson, 2002; Björnsson & Marsland, 2001).

In theory, fractional ply has both the characteristics of forward pruning and search extensions. But in practice, it is only used as a search extension schema, extending interesting lines of play. Reducing the search for uninteresting moves with fractional ply is not deemed feasible because it makes the search subject to the horizon effect. The horizon effect is when lines of play are evaluated wrongfully where a deeper search would have returned the correct value, as it lies just beyond the "horizon". Levy, Broughton and Taylor (Levy et al., 1989) tried to compensate for the horizon effect by identifying the cases it occurred in and created special schemas for those instances. This of course increased the complexity and computation of their program, presumably making it slower.

2.2.4 Realization Probability Search

Realization probability search (RPS) (Tsuruoka, Yokoyama, & Chikayama, 2002) is a search method that is based on the fractional-ply framework. The fractional-ply values are calculated by examining games played by expert players. Given a classification method that categorizes each move to a specific category *c*, the probability that a given category, *Pc*, should be played is calculated from records of actual games played by expert players as in Formula 2.1:

$$Pc = np/nc \tag{2.1}$$

where *nc* is the number of positions where it was possible to play a move of category *c* and *np* is the number of times a move of category *c* was actually played. The probability *Pc* is called *transition probability* and is used to calculate the *realization probability* of a node, but it can also be used to calculate fractional-ply weights for each category *c* using Formula 2.2:

$$FP = log(Pc)/log(C) \tag{2.2}$$

In Formula 2.2, *C* is a game dependent constant between 0 and 1 that needs to be tuned for each type of game-playing programs. If a move belongs to multiple categories, an optimistic approach is taken, choosing the category with the lowest FP weight, hoping the move is an interesting move worth investigating further. Algorithm 2 explicates the *RPS* algorithm.

As mentioned earlier with fractional ply (see Section 2.2.3), prematurely terminated lines of play can be evaluated positively, *updating the best value*, while a deeper search would reveal it not to be. To counter this horizon effect *RPS* implements a re-search clause to confirm the outcome of tactical unstable moves that have updated the $best$ value.

If the FP value of a given move is larger than a predefined FP threshold, that move is deemed unstable as shown in algorithm 2 *(line 9)*. At first the move is evaluated with a shallow *null window search* using the move's FP value to reduce the depth, *(line 11)*. If the shallow search evaluates the move to be the best seen so far, *(line 13)*, the outcome is confirmed with a deeper re-search, *(line 14)*, where the depth is reduced by the largest safe reduction, $fp_{threshold}$. On the other hand if the move's FP value is within the tactical safety limits, a normal FP search would be performed, *(line 18)*, reducing the depth by the FP value un-changed.

2.2.5 Enhanced Realization Probability Search

Winands and Björnsson (Winands & Björnsson, 2008) have pointed out two main problems with the RPS algorithm, one regarding a horizon effect for fail-low instances, and the other regarding the re-search.

Algorithm 2 *rps(state, α, β, depth)*

```
 1: if $depth \leq 0$ or isTerminal( state ) then
 2:    return evaluate( state )
 3: end if
 4: $best \leftarrow \alpha$
 5: $moveList \leftarrow$ generateMoves( state )
 6: while $move \leftarrow$ nextMove( moveList ) do
 7:    $fp_{weight} \leftarrow$ fpWeight( move )
 8:    makeMove( state, move )
 9:    if $fp_{weight} > fp_{threshold}$ then
10:       // Null window search
11:       $value = -rps( state, -(\alpha + 1), -\alpha, depth - fp_{weight} )$
12:       // Re-search
13:       if $value > best$ then
14:          $value = -rps( state, -\beta, -\alpha, depth - fp_{threshold} )$
15:       end if
16:    else
17:       // Normal fp search
18:       $value = -rps( state, -\beta, -\alpha, depth - fp_{weight} )$
19:    end if
20:    retractMove( state, move )
21:    if $value > best$ then
22:       $best \leftarrow value$
23:       $\alpha \leftarrow value$
24:       if $best \geq \beta$ then
25:          return best
26:       end if
27:    end if
28: end while
29: return best
```

Fail-low instances are usually not a concern with conventional *$\alpha\beta$ search* but can become troublesome when using fractional ply. When continuations are terminated prematurely there is a risk of wrongfully pruning away beneficial moves due to the horizon effect. RPS addresses this problem by re-searching fail-low moves that have a large reduction value, but always ignoring fail-low instances.

The second problem with RPS is the re-search implementation. Search reduction moves that have updated the best value are always re-searched with a normal FP value, a relative full depth. Not all moves need to be re-searched that deeply and doing so can be a waste of time. Winands and Björnsson compensate for these problems with their algorithm *Enhanced Realization Probability Search (ERPS)* (Winands & Björnsson, 2008).

2.3 Feature Discovery

To come up with the move categories to extend electively is a difficult task which is commonly done manually and requires both intuition and domain-specific knowledge. This is a tedious and time consuming trial and error prone process.

For example, in the game *Lines of Action* the move categories for RPS was constructed by combining tree intuitive feature groups, creating a set 277 mutually exclusive features which cover all possible moves in the search (Winands & Björnsson, 2008). A more common approach is to identify a smaller set of more isolated features, as discussed for chess in Section 2.2.2. Automating this process potentially enables more complex and efficient features.

To the author's knowledge there has little or no work been done on automatic feature discovery for selective search, though there has been on automatically discovering evaluation features. Evaluation features are different from selective search features in a way that evaluation features describe a state within the search whereas selective search features describes moves within the search. The main approach for automatically constructing evaluation feature is by combining predefined features through an iterative process, where each iteration increases the number of combined feature per compilation by one *i.e. iter 1 {f1, f2, f3}, iter 2* $\{(f_1 \wedge f_2), (f_1 \wedge f_3), (f_2 \wedge f_3)\}$, and *iter 3* $\{f_1 \wedge f_2 \wedge f_3\}$. Note that the combination operator $\wedge$ could just as easily be $\vee$ or $\otimes$.

One such method is *Generalized Linear Evaluation Model (GLEM)* (Buro, 1998), which is a well suited for most board games. It is a semi-automatic process that combines mutually exclusive atomic features as described above, with the addition to learning the weights of newly formed features and evaluating their performance. This is a time consuming process which continues until the user is satisfied with the results and halts the process. To reduce the evaluation time, GLEM only evaluates *active* combinations, where a combination is active if it occurs at least n-times in a set of training positions. GLEM has been used successfully in Othello where it added considerable playing strength to Logistello, a program that has beaten the best human players. Besides discovering new features and fitting their weights, GLEM also provides a procedure for generating training positions.

Sturtevant and White (Sturtevant & White, 2006) use a similar approach when learning features for the card game Hearts. As the features needed to play cards are quite complex, they defined a set of useful perfect-information features describing the game. These features where then used to create higher level features by exhaustively creating all possible combinations as described above. Unlike GLEM, this method has no strategy of limiting the feature growth other than restricting the combinations to $\wedge$ operations, hampering the method from creating larger than four-wise combinations. TD-learning was used to learn and evaluate the features. A game-playing program using these features was able to beat one of the best search-based hearts program, but what is interesting is that the more complicated features did not always result in the best performance. For instance the four-wise features were by far the best for avoiding the Q$\spadesuit$ whereas little difference was between two- and tree-wise feature combinations when avoiding $\heartsuit$.

Fawcett and Utgoff take a different approach with their system Zenith (Fawcett & Utgoff, 1992), which starts with a single feature describing the system's goal, from which other features are derived creating sub-goals of a sort. This approach has been implemented with some success for general game playing, where the features are derived from a logical description of the game (Tomoyuki Kaneko & Kawai, 2001).

A more detailed compilation of automatic evaluation feature discovery can be read in (Utgoff, 2001).

2.4 Summary

The chapter gave an introduction to *adversary search*, *game tree*, *minimax algorithm* and the widely used *$\alpha\beta$ algorithm*. The $\alpha\beta$ algorithm and many of its enhancements improve the search by gradually confining the search to the critical tree. But it does so considering all explored moves to be of equal interest.

Selective search methods take a more intelligent approach by examining interesting moves more closely, called *search extensions*, while less interesting moves are terminated early, known as *speculative pruning*. *Fractional ply* is one such method that can both extend and reduce the search. To prevent fractional ply from wrongfully pruning away important moves it is mainly used as a search extension. *Realization probability search (RPS)* improves on fractional ply's shortcomings by re-searching tactically unsafe moves, enabling reduced search for uninteresting moves with fractional ply. RPS also introduced a method for learning fractional ply values based on human expert knowledge. *Enhanced realization probability search (ERPS)* is a more stable implementation of RPS that considers a larger scale of tactically unsafe moves and limits unnecessary re-searches.

We also gave an introduction to methods for automatic feature discovery in games. However, these methods all concentrate on learning features for heuristic evaluation functions. There are no such methods for search-control feature discovery.

In the next chapter we will introduce a new method called *Gradual Focus*, which automatically discovers search-control features.

Chapter 3

Method

The more original a discovery, the more obvious it seems afterwards.
Arthur Koestler (1905 - 1983)

This chapter describes the method *Gradual Focus (GF)*, which is a new approach for discovering search-control features.

3.1 Search-Control Features

When choosing search control features, their properties and the effect they have on the search tree must be considered. This is a combination of the feature's frequency, fractional ply (FP) value, and other properties.

For example, when extending on frequent features the nominal search depth might diminish too much, whereas extending on uncommon features has insignificant effects. The FP value can be used to increase or decrease a feature's effect on the search tree. However, features with different properties are affected differently. The most beneficial features to extend on are those where such an extension commonly reveals some new truth, but at the same time is neither too frequent nor uncommon.

To employ an effective extension scheme that utilizes a range of different features, the issues described above must be considered for the features collectively as well as individually, increasing the level of complexity even more.

The Gradual-focus method addresses these issues by discovering interesting features automatically through feature combinations.

3.2 Feature Combinations

Atomic features are basic features of a domain e.g. *move pawn* or *move piece to top row*. Other features are a combination of two or more atomic features e.g. *move pawn to the top row*, which is the *pawn promotion* feature in chess. Some feature combinations are natural because of preexisting domain knowledge, others are not. Given a set of atomic features, in theory all possible combinations can be created by calculating the power set. The output is a set of $2^n - 1$ features, excluding the empty set, where n is the number of atomic features in the original set.

Evaluating all possible features would reveal the most effective combination, but unfortunately the evaluation process is too time consuming. Even a relatively small set of 16 atomic features would take approximately 90 days given a 2 minute evaluation of each new feature, making this method unfeasible for even for relatively small sets.

3.3 Gradual Focus

Gradual focus (GF) is a more intelligent way of constructing interesting features than the exhaustive power-set method. GF combines mutually exclusive features in an iterative fashion, where each iteration creates a set of more detailed features, gradually narrowing their focus, using a variety of pruning methods to reduce the number of evaluations needed.

3.3.1 Overview

GF discovers new features from a predefined set of mutually exclusive atomic features that are provided by the user. Combining these features with an $\wedge$ operator creates more detailed features, i.e. one-wise, two-wise, three-wise, etc. combinations. To keep track of a feature's evolution, GF iteratively evolves the features one level at a time and measures their progress. Those features that do not increase the efficiency are pruned off and prevented from occurring as subsets in other features. This process is repeated until there are no more possible combinations, at which time GF outputs its results and halts.

Figure 3.1 shows an example of this process where the first level consist of the predefined atomic features provided by the user, which is called the *Base* set. The second level shows the feature set generated by the first iteration, were all two-wise combinations of the *Base* set are created. Each feature in the set is evaluated individually and those that perform

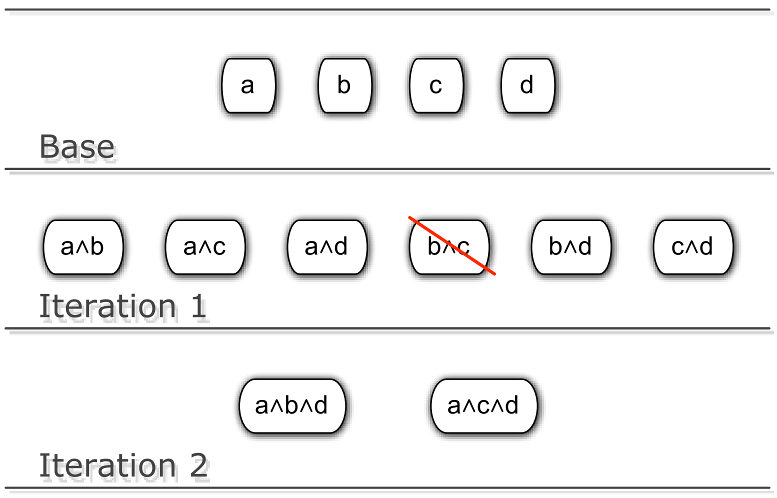


Figure 3.1: Overview of Gradual Focus

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, as well as preventing four-wise combinations from being formed.

3.3.2 Implementation

The GF algorithm is shown as *Algorithm 3*, where it starts by initializing two variables, *Output* and *Neutral*. The *Output* variable maintains a collection of promising selective search features and their effect on the search, while the *Neutral* variable is an empty feature.

The *Neutral* feature is evaluated in *line 2* to get a sense for the search tree's behavioral pattern without the use of an search extensions. This serves as a baseline for evaluating the quality of other features created by GF, as well a possible baseline for *threshold pruning*, which is explained in more detail in Section 3.6.

Evaluating a feature gives it a value indicating its quality, the higher the value the greater the quality. The evaluation also collects information about the feature's effect on the search tree, i.e. frequency in search and mean iterative depth, which is stored as part of the feature, along with its value. The evaluation function will be examined more closely in Section 3.7.

Algorithm 3 *featurLearner(ref Base, ref BlackList)*

```
1:  Output ← {}
2:  evaluate( Neutral )
3:  for all $b \in Base$ do
4:      evaluate( b )
5:  end for
6:  if UseThreasholdPruning then
7:      for all $b \in Base$ do
8:          if $b_{value} < \delta$ then
9:              Base ← Base \ {b}
10:         end if
11:     end for
12: end if
13: sortDesc( Base )
14: Output ← Neutral ∪ Base
15: newSet ← Base
16: while newSet ≠ {} do
17:     sortDesc( newSet )
18:     newSet ← featureEvolution( newSet, Base, BlackList )
19:     for all feature ∈ newSet do
20:         if ¬ isCompositionAllowed( feature, BlackList ) then
21:             evaluate( feature )
22:             Output ← Output ∪ filterEvolution( feature, BlackList )
23:         end if
24:     end for
25: end while
26: sortDesc( Output )
27: display( Output )
```

GF's main input is *Base*, which is a set of mutually exclusive atomic features. These features are defined by the user and are the base from which other features will evolve from through a series of combinations. Each of *Base's* features is evaluated *(line 4)* to establish a base line for the evolution process. Those features that are evaluated below an expected level of quality *(lines 6-12)* can be removed from the *Base* set, reducing GF's branching factor. This is an optional pruning method called *threshold pruning* which is discussed in more detail in Section 3.6. The *Neutral* feature and the *Base* set are both added to GF's output *(line 14)*: *Neutral* for its comparison properties, and *Base* to ensure that features that will not benefit from combining with other features will not be cut off.

The feature evolution process occurs in *lines 16-25*. This is an iterative process that gradually evolves the features in all possible dimensions by one step. A feature is evolved by combining it with a feature from the *Base* set, creating a new feature. The number of

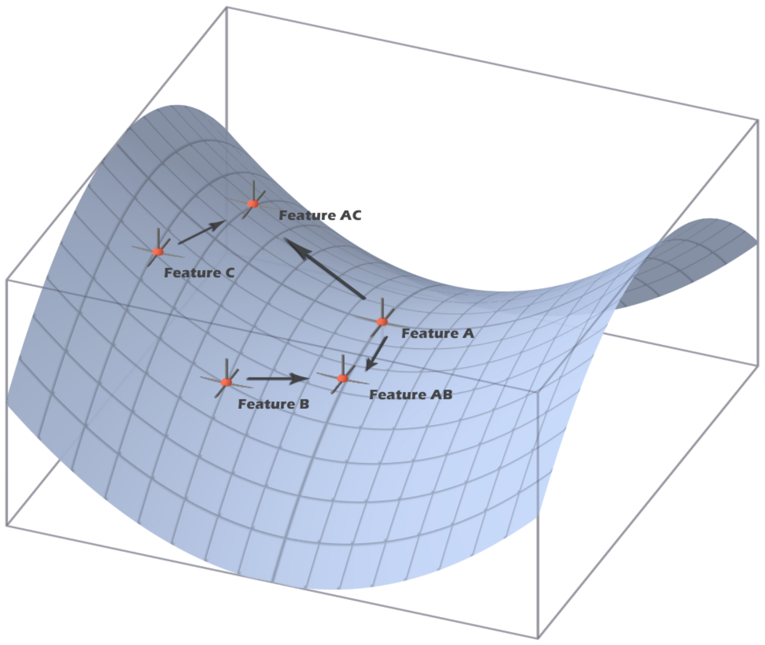


Figure 3.2: A conceptual evolution slope
(Alexandrov, 2008)

possible evolutionary dimensions for a given feature is relative to the number of available combinations with the *Base* set.

The evolution phase of each iteration is performed within the *featureEvolution* function *(line 18)*, where all the features in the *newSet* are evolved in accordance with the *Base* set and the *BlackList*. The *BlackList*, GF's second input parameter, is a pruning reference of disproven features that should not occur as subsets in other combinations. It is within the *featureEvolution* function that *BlackList* is used to prune away irrelevant combinations, all other combinations are returned as a set, updating the *newSet* with the newly evolved features. A more detailed description of the *featureEvolution* function will be given later in this chapter. Note that both the *Base* and the *newSet* sets must be sorted in a descending order by evaluation score *(lines 13 and 17)* before the evolution phase to prevent false-positive assessments of a feature's quality.

GF presumes the usage of mutually exclusive features, making newly formed features the intersection of its parents. This creates more detailed features with each iteration, gradually narrowing the features applicability and decreasing their frequency in a search, as shown in Formula 3.1. This enables the assessment of a feature' s evolutionary progress, as newly evolved features should have a higher evaluation value than their parents.

$$(A \cap B)_{frequency} < min(A_{frequency}, B_{frequency}) \tag{3.1}$$

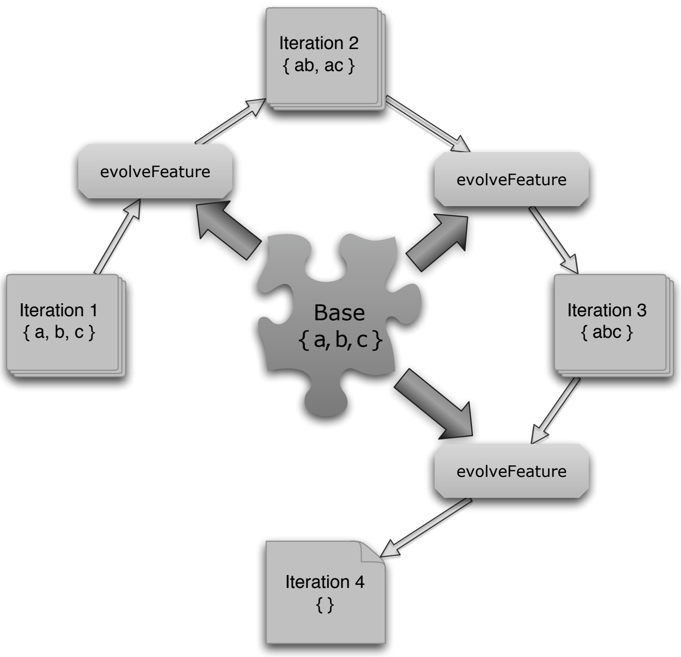


Figure 3.3: Evolution iteration

Figure 3.2 shows a conceptual representation how a feature evolves into a given dimension by combining it with other features, where the height of the surface corresponds to a feature's value. When a feature's evolution results in a decreasing value, the evolutionary direction descends down the slope, indicating that the feature advancement in a given direction should be terminated, pruning off further unnecessary evaluations. A feature has reached a local maximum when there are no more evolutionary advancements available, which is the features saddle point.

This occurs in *line 19-24*, where each of the newly evolved features are evaluated *(line 21)* and their evolutionary direction assessed within *filterEvolution (line 22)* 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 22)*. Disproving a feature within *filterEvolution* might also disprove other features in the *newSet* that are yet to be evaluated, which is why features are compared against the *BlackList* each time in the loop before they are evaluated *(line 20)*.

This evolution process is iterated until no new features can be formed, the *featureEvolution* function returns an empty set *(line 18 and 16)*, meaning that all interesting evolution paths have been explored. Figure 3.3 shows an example of this whole process where the *Base* set consist of three features *a, b,* and *c*.

Algorithm 4 *featureEvolution(ref A, ref B, ref 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 $\neg$ 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
```

Algorithm 5 *filterEvolution(ref newFeature, ref BlackList)*

```
1: if $newFeature_{value} < newFeature_{firstParentsValue} + \epsilon$ 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 return {}
10: else
11:    return return newFeature
12: end if
```

The implementation of the *featureEvolution* routine is shown as Algorithm 4 where *A* is a set of features that are to be evolved and *B* is its set of possible features to combine with. The same feature can be composed in various ways with different first and second parent, affecting its evolutionary assessment. To prevent false positive assessments of a feature's evolutionary progress, newly formed features' first parent must have a greater value then the second parent, which is why the input sets are sorted in descending order before the evolution phase *(line 13 and 17 in Algorithm 3)* and then paired together from left to right *(line 2 and 3 in Algorithm 4).* To prevent unnecessary combinations, the features can not belong to the same group or be on the *BlackList*, as well as removing duplicate features *(line 4 and 6).* Feature groups are to prevent illogical combinations and will be discussed in more detail in Section 3.4. 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 implementation for assessing the evolution progress is shown in Algorithm 5. In *line 1* the new feature's value is compared against its first parent. The variable ϵ is a domain specific constant to control the lenience of the evolution process. As stated earlier, features that do not enhance their parent are pruned off, which is done by adding the feature to the *BlackList (line 2)* hindering it from occurring in new evolution sets. On the other hand if the feature surpasses its parent, it is returned *(line 11)* and will be added to GF's output as a possible search control candidate. Lines *3-8* in the algorithm are part of the *linear tree pruning* method which is an optional pruning addition that will be discussed in Section 3.5.

GF's strength is its ability to assess the evolution process, pruning off unfavorable feature combinations, and remove redundant evaluations with the use of groups. Unfortunately the number of features may still grow exponentially, just as with other feature combination methods. By using GF's additional pruning methods, *linear tree pruning* and *threshold pruning*, the exponential growth can be further reduced.

3.4 Groups

Some features are not compatible in a sense that by combining them will result in a feature that can never occur in the domain e.g. for chess; *capture the king* or *move a pawn and move a rook*. Evaluating these features is the same as evaluating the *Neutral* feature and should therefore be pruned off. One solution would be to initialize the *BlackList* with all possible combinations that are not compatible. This is of course not feasible for large sets but can be used for deviations that are not to be evaluated for some reason. A more intuitive approach is to place features into logical groups, where combining features within a group is not allowed *(line 4 in Algorithm 4)*. As an example, a set of six features divided into two equal groups, creates 63 different feature combinations without the use of Groups, but only 15 if employed. Features created through combination inherit the groups of their parents, making them belong to more than one group.

3.5 Linear Tree Pruning

It is common for features within a group to form a natural hierarchy, e.g. where some features are subsets of others. Taking advantage of the tree hierarchy gives additional information that can be used to predict the outcome of future evaluations and prune off those that are expected to have a negative evaluation. Features belonging to a hierarchy of this type will be referred to as root, parent, and child features in accordance with regular tree definitions. The logic is that if a new combination formed with a parent

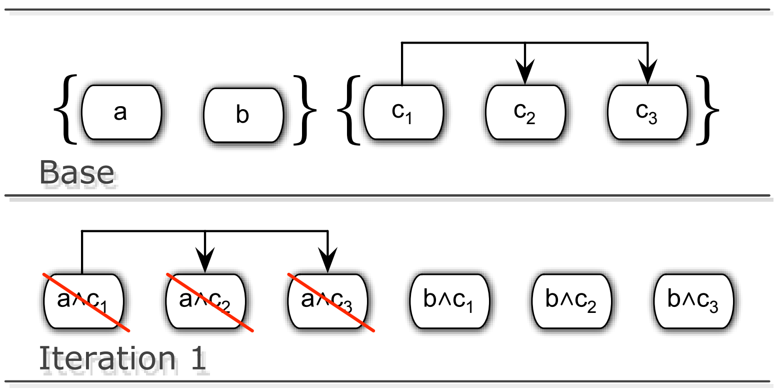


Figure 3.4: Linear tree pruning example

feature is pruned off, then other combinations with that feature's children can also be eliminated.

Figure 3.4 shows an example of this where the *Base* set has been divided into two groups, where the second group has a hierarchy where c_2 and c_3 are children of c_1. In the first iteration GF creates all two-wise combinations which are then evaluated. The combination $a \wedge c_1$ is negatively evaluated resulting in the more detailed features $a \wedge c_2$ and $a \wedge c_3$ also to be pruned off.

As a practical example envision an arbitrary game with an 8x8 board where the feature a is to move a piece into an undefended position, c_1 is the opponent's side of the board *(rank 5 to 8)*, c_2 is rank 5 and 6, and c_3 is rank 7 and 8. Given that piece count is an important property of the game and that moving a piece undefended onto the opponent's half of the board frequently results in the piece's capture, then the feature $a \wedge c_1$ will be pruned off as well as the more specialized child features $a \wedge c_2$ and $a \wedge c_3$.

Unfortunately this method cannot guarantee that interesting combinations won't be pruned off as combinations with a highly frequent parent feature might result in a negative evaluation assessment as it would extend too aggressively, whereas a combination with its less frequent children might not. It is the authors belief that this is an unlikely scenario because the union of two feature will be considerable less frequent than its parents, as shown by Formula 3.1. Nevertheless, the possibility of wrongfully pruning away interesting continuations exists, making this an optional speculative enhancement.

The implementation is an addition to *Algorithm 5* in *line 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 *Black list (line 6)*.

3.6 Threshold Pruning

Threshold pruning is the last of the optional pruning methods available for GF. It is a speculative pruning method which removes all features from the *Base* set that are below a given quality threshold, determining those features immediately as disadvantageous and therefore not eligible to participate in the evolution process *(line 6-12 in Algorithm 3)*.

This is potentially a very effective pruning method as it reduces the exponential growth in the number of combined features constructed by reducing the size of the *Base* set, but the risk of wrongfully pruning away potentially good candidates exists. The risk of wrongful pruning can be lessened by marking which features can be safely pruned; this however requires some knowledge of the search domain. A sensible choice for the threshold parameter δ *(line 8)*, would be to approximate it around the value of the *Neutral* feature.

3.7 Evaluation Function

Each feature is evaluated by extending on it in the search of a relevant game-playing program. Instead of playing actual games, which is time-consuming, a suite of carefully selected test positions is used, with the best move known. Through this process information about the feature's effect on the search is collected: number of solved positions, list of solved positions, mean iteration depth, mean height, and feature's frequency in the search.

The number of solved positions is used directly as an evaluation of the features quality, though it is recognized that a more complicated and precise evaluation function might be formed with the gathered information.

All of the collected information is attached to the features and displayed as part of the output, as it gives insight into a feature's behavioral pattern and makes the process of choosing features from GF's output more informative.

It should be noted that the collected information is an approximation of the true value depending on how representative the positions are of the entire search domain e.g. opening, middle, or endpositions. If needed, the correct value can be measured through a couple of self-play games.

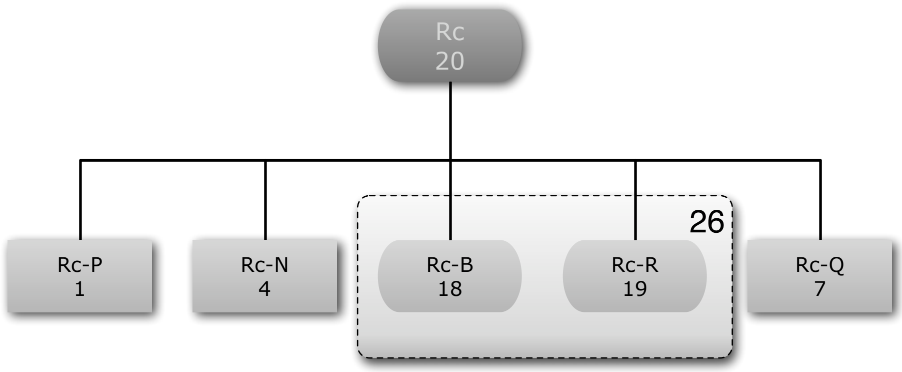


Figure 3.5: Example of non-mutually exclusive evolution

3.8 Mutually Exclusive Features

GF assumes the features in the *Base* set are all mutually exclusive, such that combining them always results in a more focused feature. Through this assumption GF can guarantee that it won't overlook relevant combinations.

This can be shown with a toy example in chess where GF combines overlapping features, meant to discover whether re-capture (Rc) is more important for some pieces than others. Figure 3.5 shows the evaluation for different re-capture combinations. First Rc is evaluated having an arbitrary value of 20. Rc is then evolved by combining it with all possible pieces, which is equivalent of splitting Rc into its discrete components. None of the newly evolved features surpass their parent and are therefore discarded. The mutually exclusive approach as discussed in Section 3.3 would on the other hand have discarded all re-captures except for Rooks and Bishops, when combined surpass the original Rc feature.

3.9 Summary

Gradual focus is an off-line learning method for learning new features and to predict their usefulness. The method starts with a set of general features that give a bird's eye view of the domain. The features are gradually evolved through feature combinations, creating more detailed features with each step, gradually narrowing the focus of the features. A key point is that features must be mutually exclusive, or the method cannot guaranty that relevant combinations are not overlooked. Grouping together related features and the ability to assess features progress, enables GF to prune away unnecessary and redundant evaluations, making feature combinations a possible approach. Additional pruning enhancements, *linear tree pruning* and *threshold pruning* further reduce the number of evaluations, but at the potential risk of wrongful pruning, and are therefore optional.

In the next chapter we evaluate the effectiveness of the GF method in practice.

Chapter 4

Results

The important thing is not to stop questioning.
Albert Einstein (1879 - 1955)

This chapter provides the results of using the Gradual-focus method for discovering search extension features for the games *Breakthrough* as well as preliminary results for chess.

4.1 Breakthrough

Breakthrough is a board game developed in 2001 by Dan Troyka (Handscomb, 2001). It has a simple set of rules but a sophisticated strategy, making it an interesting test environment. See Appendix A for details about the game and the game-playing program we used for our experiments.

4.1.1 Positions

As mentioned in Section 3.7, the Gradual-focus method evaluates the quality of a feature by running a search using it on a set of representative tactical game positions. Unfortunately there exists no standard test-suite of positions for Breakthrough where the best move is known. We therefore created a set of approximately 300 endgame positions. The only best moves known for certain in Breakthrough are ones that lead to a forced win, which is why the positions we use are endgame positions. However, using only endgame positions will not give a correct representation of a feature's true frequency, as discussed in Section 3.7.

A searches' heuristic function can also be used to measure feature quality, enabling a wider variety of positions, but it would be a questionable approximation here because the heuristic functions in Breakthrough used here is is not very refined yet.

Each of the 300 positions where picked from a game of self-play where the terminal state could be reached in seven plies. Three programs using different heuristic evaluations were used to get a variety of endgame positions.

4.1.2 Experiments Setup

A description of the input features, which form GF's *Base* set, and their properties are shown in Table 4.1 and 4.2. Each feature was evaluated with a fixed FP value of 0.5, which allows us to only concentrate on choosing the set of relevant features. A FP value of 0.5 was chosen somewhat arbitrary, although such that it is neither too conservative nor too aggressive. Each feature was evaluated through a search of 500,000 nodes, which corresponds to approximately 5-ply search. GF's evolution parameter ϵ, which is used to control the lenience of the evolution process, was set to 3 to compensate for fluctuating evaluations, and the threshold pruning parameter δ equals to the *Neutral* feature's value, which was a value of 29.

The experiments were performed on Linux CentOS 5 machines with two Intel(R) Xeon (TM) 3.00GHz CPUs and 2GB in memory. Only a single processor was used per experiment.

4.1.3 Feature Evolution Results

Five different instantiations of GF were evaluated and compared against the exhaustive expansion of the power set. The results are shown in Table 4.3 where *No Groups* disregards previously presented grouping of feature, placing each feature in a group of its own. The *Groups* instance, groups features as described previously in Table 4.2 without any additional pruning, whereas *Linear Tree Pruning* and *Threshold Pruning & LTP* additionally employ the given pruning methods. The *Domain Knowledge* instance is the same as Threshold Pruning & LTP, but with the additional knowledge that some features should not be pruned off by threshold pruning.

Table 4.1: Description of Breakthrough's base features

Feature Id	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.

As can be seen by the *No Groups* instance, GF's ability to assess the evolution process reduces the the number of generated features combinations immensely without apparently overlooking any interesting combinations. Thirteen of these evaluations were incompatible combinations that can never occur in the game, which were prevented with the *Groups* instance. Adding linear tree pruning improves the pruning slightly further and without overlooking any previously interesting combinations. However, the possibility of wrongful pruning with linear tree pruning exists, which questions its usage for such little gains. A plausible explanation for its mediocre performance is that only 1 out of 8 groups were formalized as a linear tree.

Threshold pruning, on the other hand, further reduces the number of evaluations substantially, but at the cost of overlooking five of the top ten most interesting combination, *Ud-PP-Rdt*, *Ud-Rdt*, *PP-Rdt*, *PP-Rdt-Edg*, and *Rc-Rdb*. These features were overlooked because their parent features, *Rdb*, *Rdt*, and *Edg* were pruned off, preventing the combinations from being formed. This knowledge can be embedded into GF, the *Domain Knowledge* instance, by marking the features *Rdb*, *RdTt*, and *Edg* un-safe for threshold pruning, as discussed in Section 3.6. This will only prevent the oversight of these five features and at the cost of increasing the branching-factor substantially.

Table 4.2: Base features for Breakthrough

Feature Id	Group	Tree hierarchy, parent's name	Threshold pruning safe No Knowledge	Threshold pruning safe Knowledge
Ud	Ud	–	true	true
PP	PP	–	true	true
Rc	C	–	true	true
C	C	–	true	true
Ms	Ms	–	true	true
Rdb	Vertically	Root	true	*false*
RdBt	Vertically	Rdb	true	true
RdBb	Vertically	Rdb	true	true
Rdt	Vertically	Root	true	*false*
RdTt	Vertically	Rdt	true	true
RdTb	Vertically	Rdt	true	true
Edg	Horizontal	Root	true	*false*
Mr	Horizontal	Root	true	true
Udp	Udp	–	true	true
Bv2	Bv2	–	true	true

Table 4.3: Gradual focus evaluations in Breakthrough

Method type	Hours	Evaluations #	% of $\mathcal{P}$	Overlooked combinations
Power Set	-	32,768	100%	None
No Groups	30.6	138	0.42%	None
Groups	24.9	112	0.34%	None
Linear Tree Pruning (LTP)	23.7	105	0.32%	None
Threshold Pruning & LTP	5.8	27	0.08%	5
Domain Knowledge	10.0	45	0.13%	None

4.1.4 Precision and Recall

GF's findings were assessed by comparing them against a complete set of all one-, two-, and tree-wise combinations, in all 377 features. The top 25 features are shown in Table 4.4 where star symbols correspond to the top 10 features discovered by GF.

As expected, GF discovers the features with the highest number of solved positions, which is in accordance with the algorithm's design. But on the other hand it would seem that GF is overlooking a fair amount of effective features. However, these features are less effective descendants of GF's already discovered features and were therefore correctly pruned off. These features are redundant as their benefits are already obtained with the

Table 4.4: Tree-Wise Combinations in Breakthrough

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

use of their parent. Thus GF, at least in this domain, offers both good precision and recall rate.

4.1.5 Tournament Results

The top ten features suggested by GF as being interesting based on the number of solved positions were evaluated through self-play. Each of the players searched 500,000 nodes per move with a FP value of 0.5, in consistency with the evolution criteria. Table 4.5 shows the results, where all but three of the features increased the winning percentage, by them self and without any additional tuning of their FP values.

The features *Rc-Rdb* and *PP-Rdt-Edg* have a mediocre effect on the game which can be explained by their low frequency. Their effect on the search can be increased by lowering

Table 4.5: Features' result in Breakthrough

Feature Id	Games #	Wins %	Conf. Int.	Frequency	
				In GF	50 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%

their FP value, and it is known that *Rc-Rdb* with a FP value of 0.25 will increase the winning percentage to 60% ±3.92. A similar result could be expected for the *PP-Rdt-Edg* feature. However, tuning the FP values is outside the scope of this thesis, but for further reading see e.g. work by Björnsson (Björnsson, 2002; Björnsson & Marsland, 2001). The *Ud* feature on the other hand has a negative effect on the game, the opposite of what was observed during GF's evaluation. This is because GF's evaluation was based on end-game positions where *Ud*'s frequency was within suitable limits for fractional-ply extensions of 0.5, whereas *Ud*'s frequency in the whole game-tree is much higher, where a FP value of 0.5 results in an over aggressive extension as described in Section 2.2.2.

By extending on all of these features, except *Ud*, and tuning their FP value might result in even better performance than already shown, making GF a worthy addition to constructing selective search schemes.

4.2 Chess

In chess there exists a well known set of features to extend on that have been established over decades of experience with the development of chess-playing programs. These features are *re-capture*, *passing pawn*, and *check*. If the GF method could discover these features on its own it would strengthen the claim of its usefulness, besides satisfying our curiosity, and was therefore used with Fabien Letouzey's chess engine Fruit 2.1 (Letouzey, 2005), which is a formidable open source chess program.

4.2.1 Positions

The *Encyclopedia of Chess Middlegames (ECM)* was used to evaluate the performance of search extension features in chess. It is a set of 879 positions with a suggested best move for each position, and is one of a few such suits that has become somewhat of a de-facto standard for testing chess-program algorithms.

However, to minimize the evaluation turnaround time, the whole set was reduced to a bare minimum of 102 positions *(Appendix B.1)*. These 102 positions were identified as the positions Fruit could solve additionally when using its known extensions. For simplicity sake, all of the standard search additions were disabled in Fruit, except for the quiescence search (Shannon, 1950).

4.2.2 Experiments Setup

The *Base* features used for discovering *re-capture*, *passing pawn*, and *check* are shown in Table 4.6. However, GF requires its atomic features to be mutually exclusive or the methods integrity cannot be guaranteed. This places a restriction on the granularity of atomic features which prevents some of the known chess features from being discovered. This is a clear shortcoming of the GF method which might confine its usage to certain domains. Never the less, the chosen *Base* features should suffice for showing GF's abilities of finding interesting compositions.

The *check* extension is represented through the feature AcK which is an atomic feature, meaning that the check extension cannot be a composition of two features or more without breaking GF's mutually exclusive rule.

The *passing pawn* extension should emerge as a combination of PcP and RdBt, though it is unlikely that the combination would emerge at all as the ECM positions are middlegame positions and passing pawns are more likely to occur in endgames.

The *re-capture* extension can be perceived as an atomic feature and is therefore a little tricky to composite through other features and not entirely "truthful", meaning that the composition is more of an approximation. Combining the Cc and Lc groups should revel re-captures of each piece type, LcP-CcP for pawns, LcN-CcN for knights, LcB-CcB for bishops, LcR-CcR for rooks, and LcQ-CcQ for the queen. The only certain re-capture instance is the LcQ-CcQ for there is only one queen, where as for knights, bishops, and rooks there is a 1/2 chance of the feature being a re-capture instance and only 1/8 chance

Table 4.6: Base features for Chess

Feature Id	Feature Description	Group	Tree hierarchy, parent's name	Threshold pruning safe
Pc*	Move a piece { * | K, Q, R, B, N, P }	Pc	–	false
Ac*	Attack a piece { * | K, Q, R, B, N, P }	Ac	–	false
Rc	Re-capture	Rc	–	false
Rdb	Players half of the board	Vertically	Root	false
RdBb	First and second rank of the board	Vertically	Rdb	false
RdBt	Third and fourth rank of the board	Vertically	Rdb	false
Rdt	Opponents half of the board	Vertically	Root	false
RdTb	Fifth and sixth rank of the board	Vertically	Rdt	false
RdTt	Seventh and eighth rank of the board	Vertically	Rdt	false
Cc*	Capture a piece { * | Q, R, B, N, P }	Cc	–	false
Lc*	Opponent previously moved a piece { * | Q, R, B, N, P }	Lc	–	false

for pawns. These features should stand out significantly never the less, except for pawns, emerging an interesting re-capture pattern.

There are some feature combinations in Table 4.6 that are obviously not logical. These combinations were initialized into the *BlackList* as discussed in Section 3.4 and are the following; Pc-RdBb, RcK-RdTt, PcK-AcK, and PcK-AcQ.

To reduce the turnaround time the learning process was divided by learning the re-capture composition, the Cc* and Lc* features, separately. Instead the true re-capture feature *(Rc)* was added to the base set to observe Rc's interaction with other features.

Each feature was evaluated through a search of 1,000,000 nodes, which corresponds to approximately 7-ply search. GF's evolution parameter ϵ was initialized with a value 0

Table 4.7: Gradual focus output in Chess

Feature Id	Solved Positions
AcK	44
PcK	29
Rc-Rdb	26
Rc	24
RdTt	17
PcQ-Rdt	17
RdBb	15
Rdt	15
PcN-RdTb	12
AcR	11

as Fruit's evaluation function is much more decisive than that of Breakthrough. The threshold parameter δ was also set to 0 as it was never used.

All of the experiments were performed on Linux CentOS 5 machines with two Intel(R) Xeon (TM) 3.00GHz CPUs and 2GB in memory. Only a single processor was used per experiment.

4.2.3 Feature Evolution Results

Table 4.7 shows the top 10 features GF suggest and the discovery of re-recapture features in Table 4.8. It comes as no surprise that the *check* feature AcK is among the highest ranking, nor that *passing pawn* feature PcP-RdTt is not. What is surprising is that the only certain *re-capture* feature, re-capturing the queen, does not emerge while re-capturing the knight, bishop, and rook does.

The second best feature is moving the king (PcK), which is logical as it is a response to the check feature. Another interesting result is that GF suggest that it is beneficial to extend on re-captures on the players half of the board. Other features, *RdTt* to *PcN-RdTb* are moving pieces onto the opponents half of the board, presumably leading up to a check event.

Taking a close look at the emergence of the re-capture extension through Table 4.8, shows that extending on true re-capture instances for all pieces results in a value of 24. The fact that the only emerging re-capture approximation is that of the knight, bishop and rook, indicate that these features are the main providers of re-captures success.

Table 4.8: Re-capture results in Chess

Feature Id	Solved Positions
Rc	24
CcR-LcR	15
CcB-LcB	14
CcR	13
CcB	11
CcKn-LcKn	11

Combining these into a feature that extends for re-capture of bishops, knights, and rooks will result in a value of 21, narrowing it even further to bishops and rooks, removing the weakest link, results in a value of 26. This is a very small increase, which is not enough to claim a new re-capture approach in chess, but it is a good example of GF's finesse in finding new interesting combinations.

4.3 Summary

The Gradual focus method was used with a Breakthrough program, showing its capabilities of finding interesting features combinations, while evaluating only 0.34% without additional pruning methods, compared to a brute force method. The number of evaluations was reduced even further with linear tree pruning and threshold pruning. Linear tree pruning reduced the number of evaluations only additionally to 0.32%, while threshold pruning further reduced the number of evaluations to 0.08%, but at the cost of overlooking five interesting combination. GF's suggestions for interesting labels in Breakthrough were evaluated one at a time, without tuning the FP value. All but three label showed positive results, which suggests that selecting a set of these labels and tuning their FP value would create a formidable extension method.

Gradual focus was also implemented in chess with the purpose of finding known extensions and there by validating the method. Here GF's shortcomings appear, due to the restriction of mutually exclusive features, some levels of granularity for atomic features cannot be reached, preventing some features from being discovered. Even so, GF confidently identified the *check* extensions and *re-capture* to some extent. Though the emerged re-capture combination was not as expected, it was a better fit to the data than the standard re-capture implementation. As the data set covered only middlegames, *passing pawns* could not emerge. Never the less, a confident statement can be given for GF's effectiveness.

Chapter 5

Conclusions

Chess is generally considered to require "thinking" for a skilful play. A solution to this problem will force us to admit that machine thinks, or further restrict the concept of "thinking".
Claude Shannon (1916-2001)

In this thesis we explained the importance of selective search. One of the control issues is choosing effective move categories to extend on, which can be a tedious and time consuming process. We therefore introduced a new method called *Gradual Focus* which is an automatic feature discovery method for selective search. The method was used for discovering search extension features for the game *Breakthrough* as well as finding known extension features in chess.

Gradual Focus, like many other feature discovery methods, creates new features by combining features from a given base set. This results in an exponential growth in the number of possible features where all combinations is the power set which consists of 2^n features where n is the number of features in the base set of atomic features. Evaluating a set of this magnitude is not feasible, which is why other feature discovery methods create only combinations of very few features. Gradual-focus on the other hand is a merit driven method which selectively continues to evolve features while still beneficial.

By presuming the usage of mutually exclusive features and combining them in an iterative fashion, Gradual Focus is enable to assess a feature's evolutionary progress, and consequently prune off unfavorable combinations. In practice Gradual Focus was shown to reduce the feature growth by two orders of magnitude without overlooking useful combinations. The number of features can be reduced even further with Gradual Focus' additional pruning enhancements: *Groups*, *Linear Tree Pruning*, and *Threshold Pruning*.

Groups prunes away redundant features by placing them into logical groups, whereas *Linear Tree Pruning*, and *Threshold Pruning* are speculative enhancements that further reduces the number of combinations, although at the risk of wrongfully pruning away interesting combinations. Features are evaluated by using them in searching a set of carefully selected tactical positions where the best move is known. Through this process information on the feature's effect on the search is collected, which is displayed as part of the output, as it gives insight into a feature's behavioral pattern and makes the process of choosing features from Gradual Focus's output more informative. To the author's knowledge, Gradual Focus is the only existing feature discovery method for selective search.

Gradual Focus was evaluated in the game Breakthrough as well as partially for chess. In Breakthrough the method discovered all interesting features, where all but three of the top ten features increased program playing strength without any additional tuning of FP values. In chess Gradual Focus was used with the purpose of finding known extensions and thereby further validating the method. Here Gradual Focus's shortcomings appear, due to the restriction of requiring mutually exclusive features; some levels of granularity for atomic features cannot be reached, preventing potentially useful features from being discovered. Even so, Gradual Focus confidently identified the check extensions and re-capture to some extent. Though the emerged re-capture combination was not as expected, it was a better fit to the data than the standard re-capture implementation.

There are several pending issues for further work that would improve the method. First of is finding a solution to Gradual Focus's main shortcomings, which is not being able to reach sufficient level of granularity for features in some domains as was seen in chess. It is not clear what the best approach would be or if it is possible at all without losing some of Gradual Focus's properties. This is because this issue proceeds from the presumption of combining only mutually exclusive features, which is the foundation for assessing the evolutionary progress.

Secondly, it would be an interesting task to construct a more informative evaluation function than the number of solved positions. During this work a minor attempt was made to construct such a function based on the information gathered during the feature evaluation. Needless to say this is a difficult task but we are convinced that the solution lies in that direction.

Thirdly, a natural continuation of creating an informative evaluation function would be to answer the general question "What makes a good feature?", which in turn would open up new dimensions in what features should be combined.

Fourth, it would be beneficial to classify features based on their contribution in the search e.g. attacking or defending features. This would increase Gradual Focus's variety of suggested features. One approach might be to compare features' solved positions, which could also be useful for constructing a new evaluation function as mentioned previously.

Another issue is that all the features are developed and evaluated with a fixed FP value which undeniably excludes the discovery of some features. A simple solution would be to run parallel instances of the process with different FP values and compare the output. It would also increase the method's effectiveness by automatically creating a variety of test positions that are representative of the entire search domain. Last but not least, it would be worthwhile to try Gradual Focus in more domains get a broader sense of its efficiency.

Bibliography

Alexandrov, O. (2008). *A saddle point image.* Availible October 17 2008 at http://upload.wikimedia.org/wikipedia/commons/4/40/Saddle_point.png.

Beal, D. F. (1989). Experiments with the null move. In *Advances in computer chess 5* (p. 65-79).

Beal, D. F. (1990). A generalised quiescence search algorithm. *Artificial Intelligence Journal*, *43*(1), 85-98.

Birmingham, J. A., & Kent, P. (1988). Tree-searching and tree-pruning techniques. , 123–128.

Björnsson, Y., & Marsland, T. A. (1998). Multi-cut pruning in alpha-beta search. In *Computers and Games, Proceedings of CG98, LNCS* (Vol. 1558, pp. 15–24).

Björnsson, Y., & Marsland, T. A. (2001). Learning search control in adversary games. In *proceedings of the Ninth International Advances in Computer Games Conference (ACG'99)* (pp. 157–174).

Björnsson, Y. (2002). *Selective depth-first game-tree search.* Phd disertation, University of Alberta.

Buro, M. (1998). From simple features to sophisticated evaluation functions. In *Computers and Games, Proceedings of CG98, LNCS* (Vol. 1558, pp. 126–145).

Donninger, C. (1993). Null move and deep search: Selective search heuristic for obruse chess programs. *ICCA Journal*, *16*(3), 137-143.

Fawcett, T. E., & Utgoff, P. E. (1992). *Automatic feature generation for problem solving systems* (Tech. Rep.).

Goetsch, G., & Campbell, M. S. (1990). Experiments with the null-move heuristic. In *Computers, chess and cognition* (p. 159-168).

Handscomb, K. (2001). 8x8 game design competition: The winning game: Breakthrough ...and two other favorites. *Abstract Games Magazine*, *7*.

Heinz, E. A. (1998). Extended futility pruning. *ICCA Journal*, *21*(2), 75–83.

Heinz, E. A. (1999). Adaptive null-move pruning. *ICCA Journal*, *22*(3), 123–132.

Heinz, E. A. (2000). Ael pruning. *ICCA Journal*, *23*(1), 21–32.

Hyatt, R. M. (1996). *Crafty.* A chess program availible March 27 2008 at ftp://ftp.cis.uab.edu/pub/hyatt.

Knuth, D., & Moore, R. W. (1975). An analysis of alpha-beta pruning. *Artificial Intelligence*, *6*(4), 293–326.

Korf, R. E. (2007). *Heuristic search.* (Manuscript)

Letouzey, F. (2005). *Fruit 2.1.* A chess probgram availible April 24 2007 at http://www.arctrix.com/nas/chess/fruit/.

Levy, D., Broughton, D., & Taylor, M. (1989). The SEX algorithm in computer chess. *ICCA Journal*, *12*(1), 10-21.

Marsland, T. A. (1983). Relative efficiency of alpha-beta implementations. In *Proceedings of the International Joint Conference on Artificial Intelligence (ijcai-83)* (pp. 763–766).

Neuman, J. von, & Morgenstern, O. (1944). *Theory of games and economic behaviour* (Second ed.). Princeton University Press.

Nikolai Krogius, B. P., A. Livsic, & Taimanov, M. (1980). *Encyclopedia of chess middlegames.*

Reinefeld, A. (1983). An improvement to the scout tree search algorithm. *ICCA Journal*, *6*(4), 4–14.

Romstad, T. (2006). *An introduction to late move reductions.* Availible January 13 2009 at http://www.glaurungchess.com/lmr.html.

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

Shannon, C. E. (1950). Programming a computer for playing ches. *Philosophical Magazine*, *41*(314), 256–275.

Slate, D. J., & Atkin, L. R. (1977). Chess 4.5 - northwestern university chess program. In *Chess skill in man and machine* (p. 82-118).

Sturtevant, N. R., & White, A. M. (2006). Feature construction for reinforcement learning in hearts. In *In 5th international conference on computers and games.*

Tomoyuki Kaneko, K. Y., & Kawai, S. (2001). Automatic feature construction and optimization for general game player. In *Proceedings of game programming workshop 2001* (pp. 25–32).

Tsuruoka, Y., Yokoyama, D., & Chikayama, T. (2002). Game-tree search algorithm based on realization probability. *ICGA Journal*, *25*(3), 146-153.

Utgoff, P. E. (2001). Feature construction for game playing. , 131–152.

Winands, M. H. M., & Björnsson, Y. (2008). Enhanced realization probability search. *New Mathematics and Natural Computation*, *4*(3), 329-342.

Appendix A

Breakthrough

Breakthrough (Handscomb, 2001) is a two person perfect information game created by Dan Troyka in 2001 as part of the "2001 8x8 Game Design Competition", which he won. 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.

Breakthrough has a very simple set of rules, (see A.1), which makes it an easy game to learn and implement. Compared to other games such as Chess, Breakthrough's state space is relative small making analysis of the game tree more convenient. Despite Breakthrough's simplicity it still has a sophisticated strategy which requires advanced search methods to play at expert level. This makes Breakthrough an ideal platform for developing new search methods and hopefully a good stepping stone to other more interesting domains such as Chess and Checkers. The downside of using Breakthrough as a development platform is that there is little literature to build on or to compare results against.

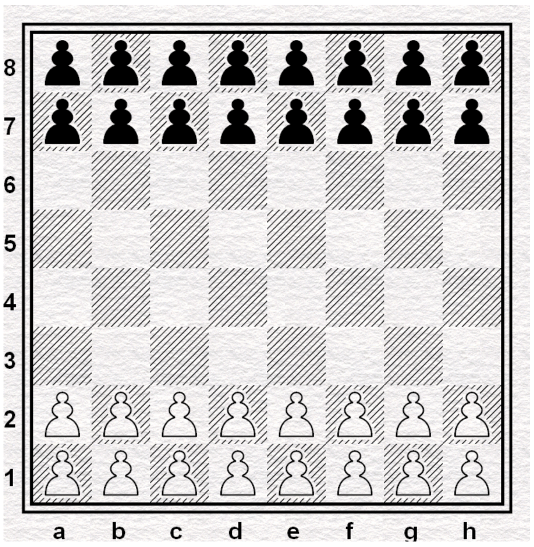
A.1 Rules

The rules of Breakthrough are as follows:

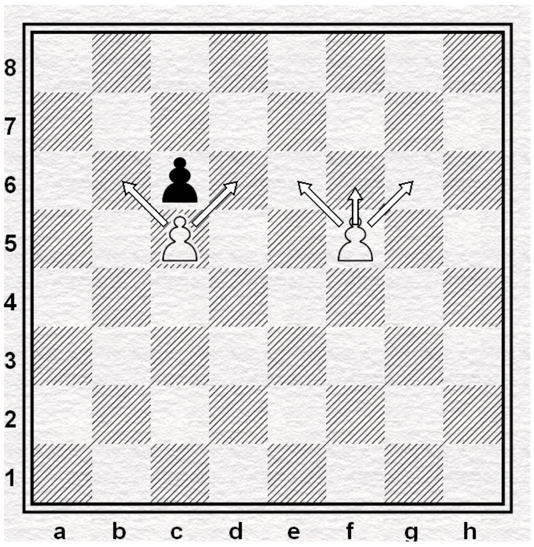
1. The start position is as shown in Figure A.1(a).
2. Players choose which side starts[1].

[1] Our implemented Breakthrough program chooses always White as the starting side, traditional Chess rules.

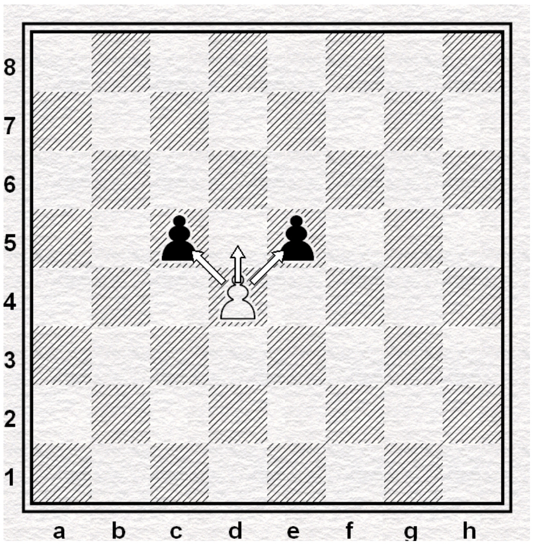
3. Players alternate move a piece.
   (a) Pieces can move one square forward or diagonally-forward to unoccupied squares, see Figure A.1(b).
   (b) Opponents piece can be captured by moving one square diagonally-forward to a square containing opponents piece, captured pieces are removed from the board. See Figure A.1(c).
4. Capture moves are not forced.
5. Game ends when player's piece reaches the opponent's back rank, see Figure A.1(d).



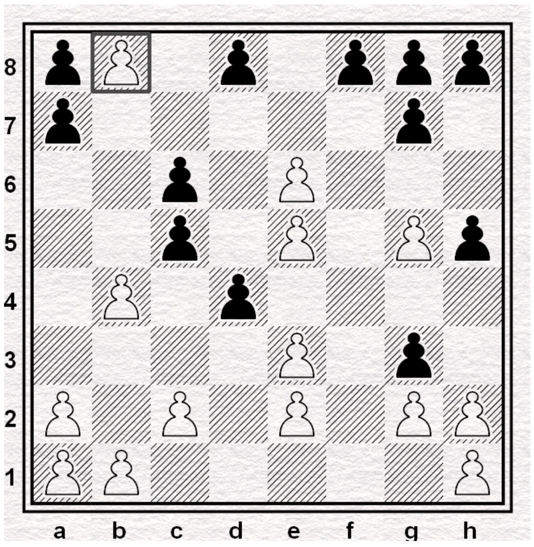
(a) Initial board position



(b) Legal moves



(c) Two legal capture moves and one regular move



(d) A terminal postition, through b8

Figure A.1: Breakthrough examples

A.2 Strategy

Although the best playing strategy for Breakthrough is unclear, there are still some useful heuristic to be exploited in creating an effective evaluation function for playing the game.

Material

Having more pieces is good as it improves both the offensive and defensive capabilities of a player.

Defense patterns

A single piece cannot stop the advancement of another piece, as the advancing piece can pass safely around by moving first in front of the defending piece and then diagonally beside it. To prevent this, defending pieces must form defense patters, the simplest being a vertical line of two pieces.

Attack force

A simple strategy for breaking through the opponent's defenses is concentrate the attack at a single point, preferably the weakest. The attacking group should be tightly formed to enable continuous attacks, preventing the opponent from regrouping.

Advancing pieces

The game itself is a race between players to get their piece to the opponent's back rank. By advancing your pieces as far as possible without sacrificing them needlessly brings them closer to that goal. However, there is a tradeoff between how far a piece can advance and how safe it is. By grouping many pieces together, the safer the pieces are and the chance of breaking though the opponent's ranks increases.

Attacking along edges

Attacking along the board's edges reduces the opponents response options for he can only counter the attack from one side instead of two. This might also force the opponent to use his middle pieces to aid against the attack, weakening the middle and the opposite side of the defenses.

A.3 Heuristics

Some of the above strategies are difficult to implement successfully, they are either computationally expensive or difficult to tune. A simple but fairly effective heuristic is the combination of piece capture and advancing pieces.

To encourage piece advancement, each piece that is not threatened receives a bonus, the further advanced the piece is the higher the bonus. Piece count is used to encourage capturing of opponent's pieces and to prevent pieces from advancing undefended up the board.

A.4 A Game Playing Program

A simple Breakthrough game playing program was constructed to run experiments on. It used $\alpha\beta$ search with iterative deepening and has access to three different heuristics which were used to create end-game positions as described in Section 4.1.1. The first heuristic was a simple piece count heuristic, the second moved all pieces gradually up the board in a misguided attempt to form a large attacking force, and the third heuristic was the one described above in Appendix A.3. To compensate for indecisive heuristics, generated moves are always shuffled. This prevents the first generated move from frequently being chosen as the best move. A simple move ordering schema is employed which expands the PV move for each ply first. This is to provide consistency between iterations so that previous PV is not discarded prematurely before being examined further.

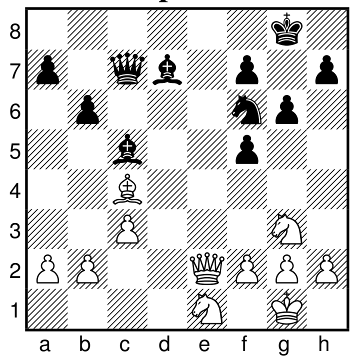
Appendix B

Test Suite

B.1 ECM Positions

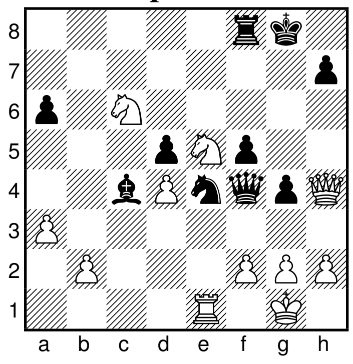
This test suit contains 102 positions from the *Encyclopedia of Chess Middlegames (ECM)* (Nikolai Krogius & Taimanov, 1980). The complete set consist of 879 positions but we used a subset of 102 positions for each experiment, chosen such that the positions were difficult to solve under the given time limit, unless using selective search extensions.

ECM position 5



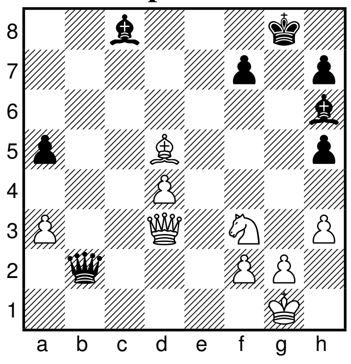
Black's turn
Best move: Bxf2+

ECM position 12



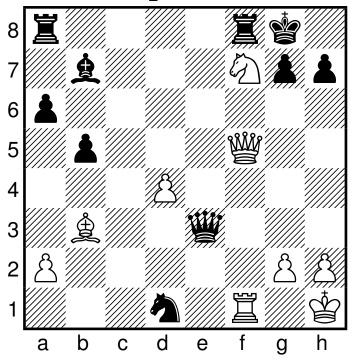
White's turn
Best move: Ne7+

ECM position 25



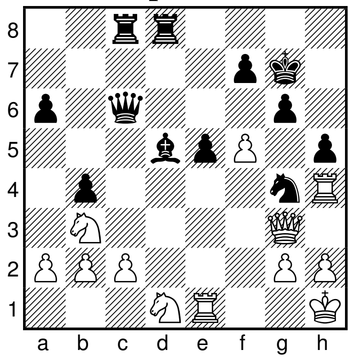
White's turn
Best move: Bxf7+

ECM position 27



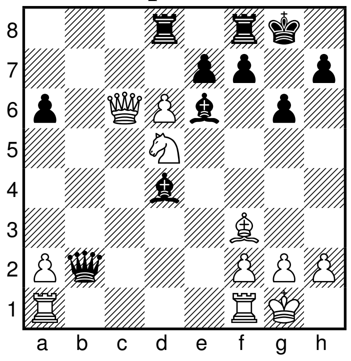
Black's turn
Best move: Bxg2+

ECM position 43



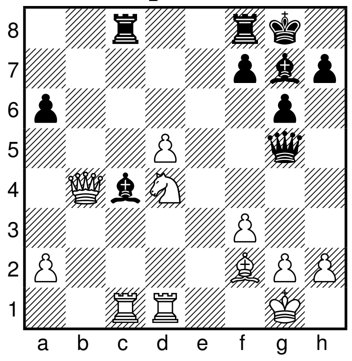
Black's turn
Best move: Bxg2+

ECM position 46



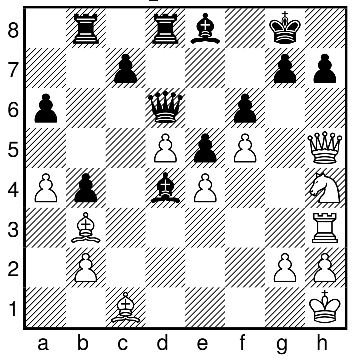
Black's turn
Best move: Bxd5

ECM position 72



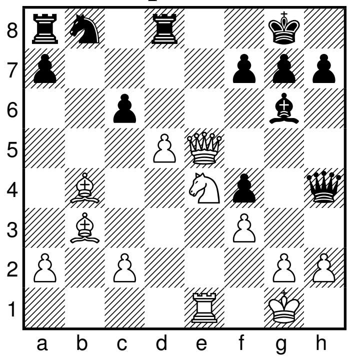
Black's turn
Best move: Bf1

ECM position 94



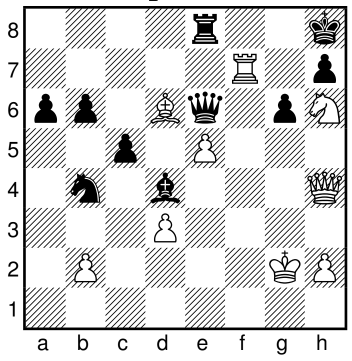
White's turn
Best move: Qxh7+

ECM position 96



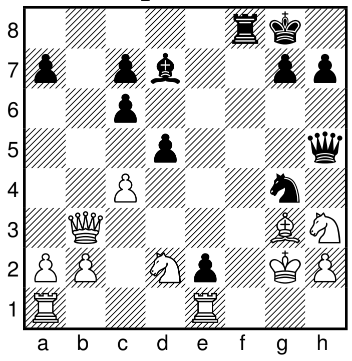
White's turn
Best move: Qe8+

ECM position 97



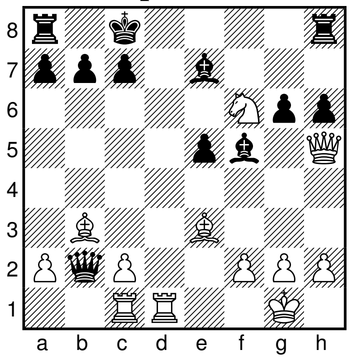
White's turn
Best move: Rxh7+

ECM position 103



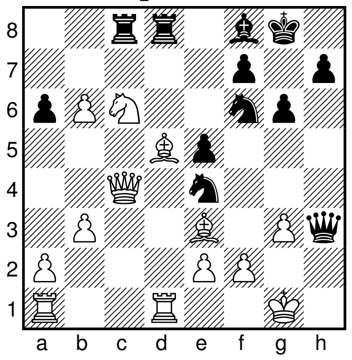
Black's turn
Best move: Qxh3+

ECM position 107



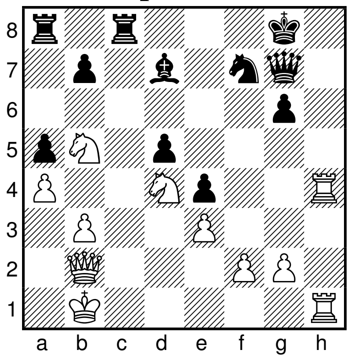
White's turn
Best move: Qxf5+

ECM position 138



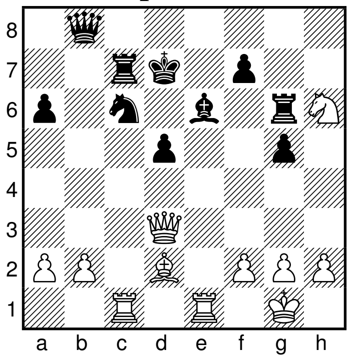
Black's turn
Best move: Rxd5

ECM position 159



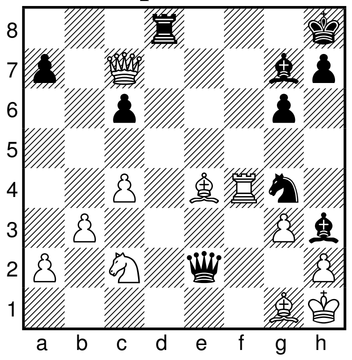
White's turn
Best move: Nd6

ECM position 176



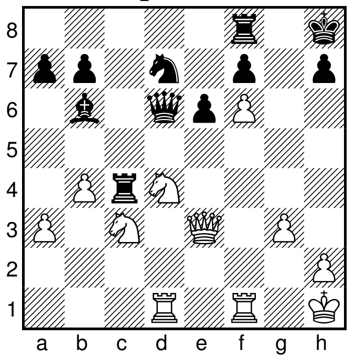
White's turn
Best move: Nxf7

ECM position 184



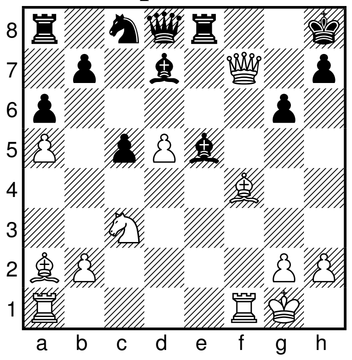
Black's turn
Best move: Nf2+

ECM position 192



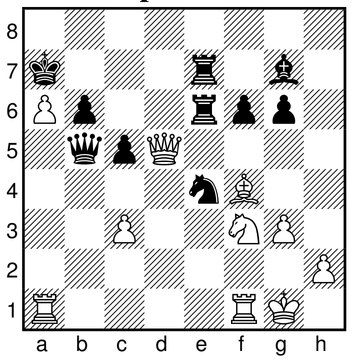
White's turn
Best move: Qh6

ECM position 199



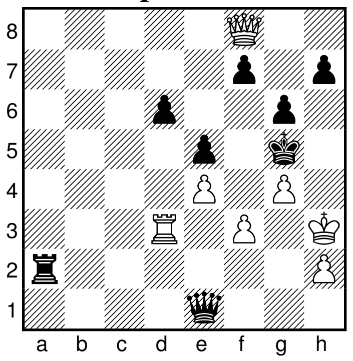
White's turn
Best move: Qf8+

ECM position 200



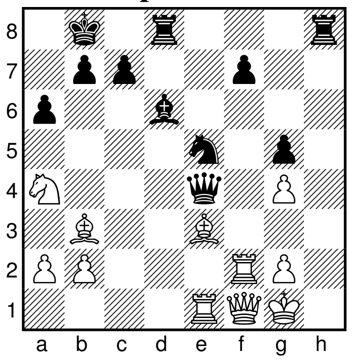
White's turn
Best move: Bb8+

ECM position 218



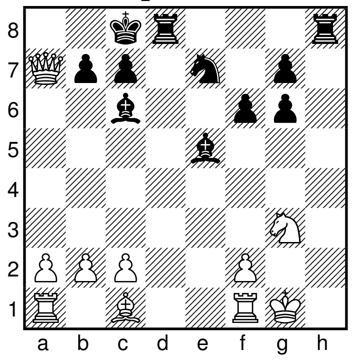
White's turn
Best move: f4+

ECM position 225



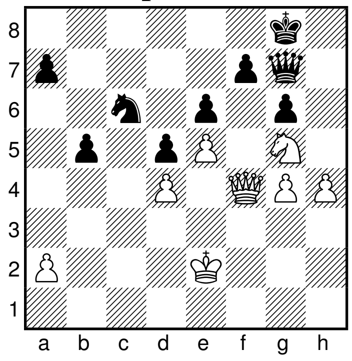
Black's turn
Best move: Rh1+

ECM position 232



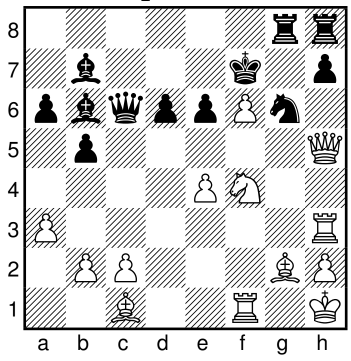
Black's turn
Best move: Rh1+

ECM position 238



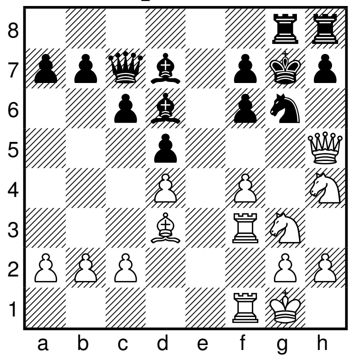
White's turn
Best move: Qc1

ECM position 243



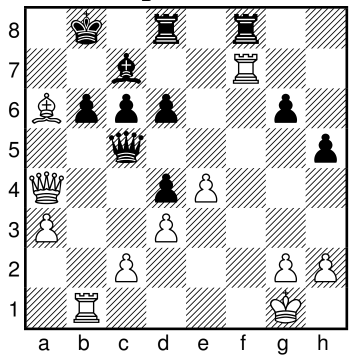
White's turn
Best move: Qxh7+

ECM position 246



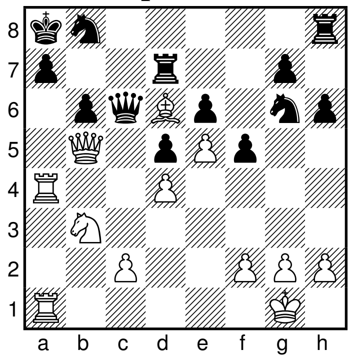
White's turn
Best move: Qh6+

ECM position 251



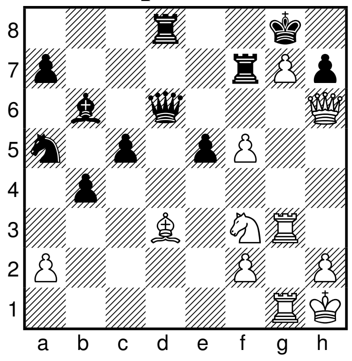
White's turn
Best move: Bb7

ECM position 256



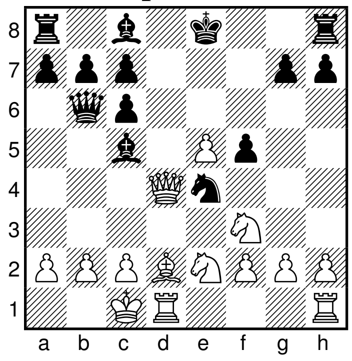
White's turn
Best move: Rxa7+

ECM position 263



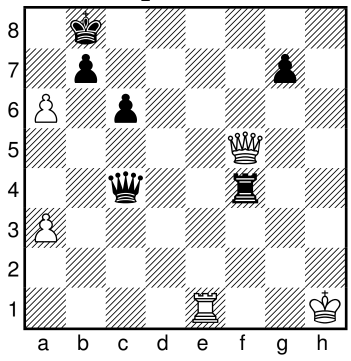
White's turn
Best move: Qxh7+

ECM position 268



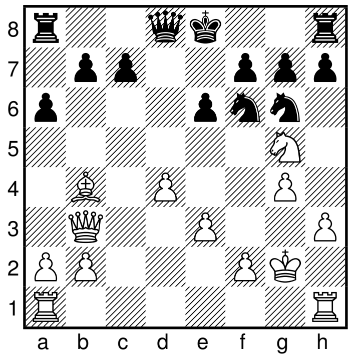
White's turn
Best move: Qd8+

ECM position 269



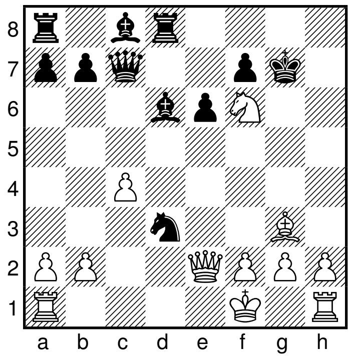
White's turn
Best move: Re8+

ECM position 272



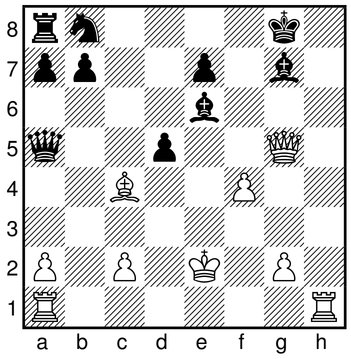
Black's turn
Best move: Nh4+

ECM position 275



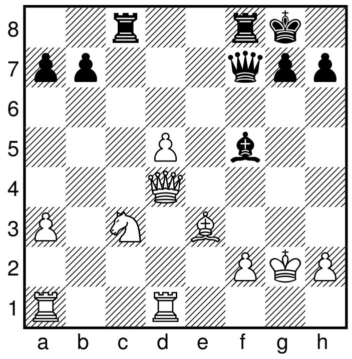
White's turn
Best move: Qg4+

ECM position 277



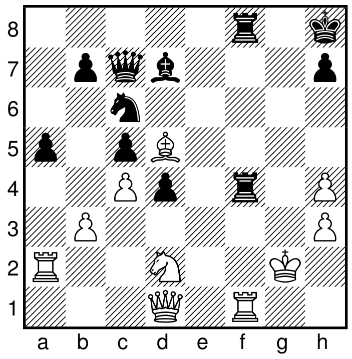
White's turn
Best move: Rh8+

ECM position 279



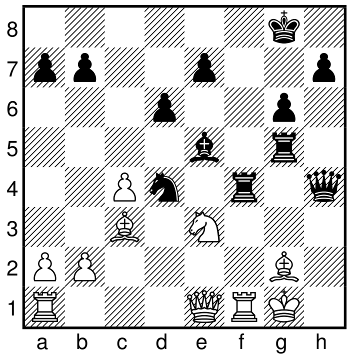
Black's turn
Best move: Rxc3

ECM position 282



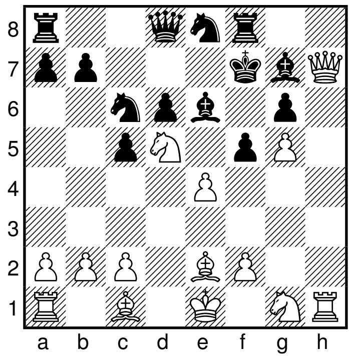
Black's turn
Best move: Bxh3+

ECM position 293



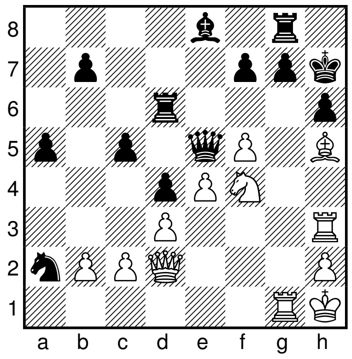
Black's turn
Best move: Qh2+

ECM position 294



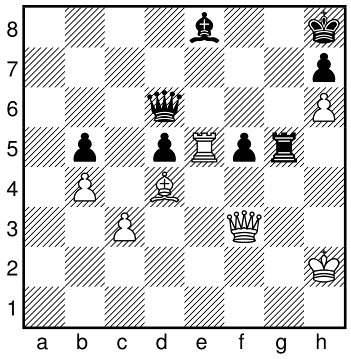
White's turn
Best move: Qxg6+

ECM position 301



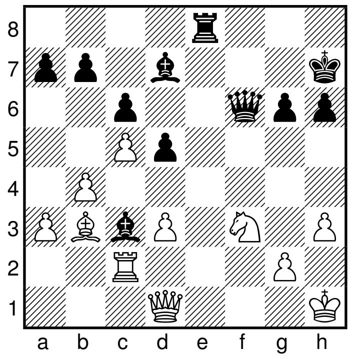
White's turn
Best move: Ne6

ECM position 302



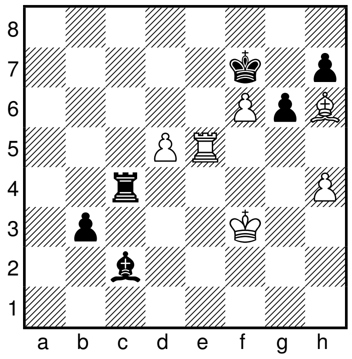
White's turn
Best move: Qg3

ECM position 317



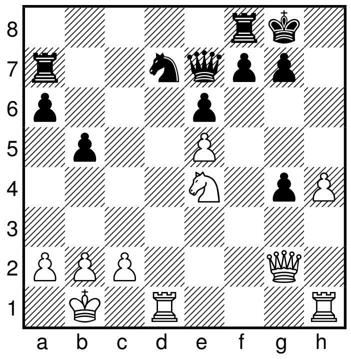
Black's turn
Best move: Re1+

ECM position 319



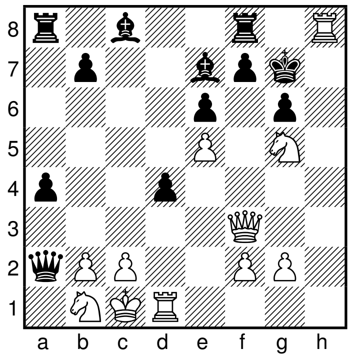
Black's turn
Best move: Re4

ECM position 328



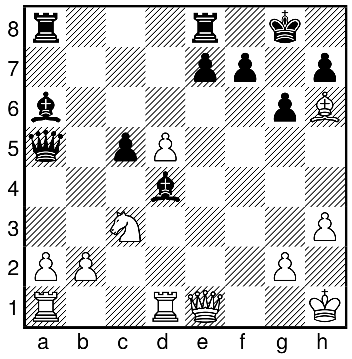
White's turn
Best move: Rxd7

ECM position 342



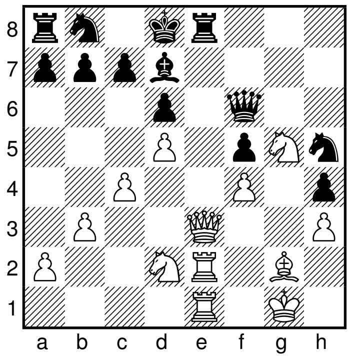
White's turn
Best move: Qf6+

ECM position 357



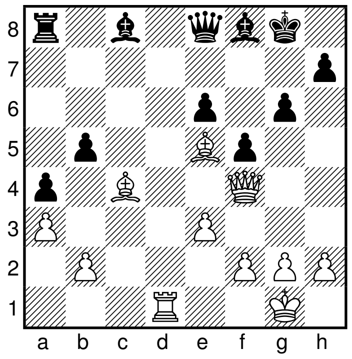
White's turn
Best move: Rxd4

ECM position 362



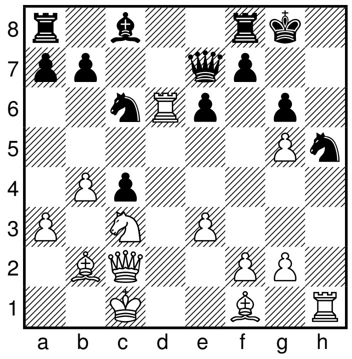
White's turn
Best move: Qxe8+

ECM position 380



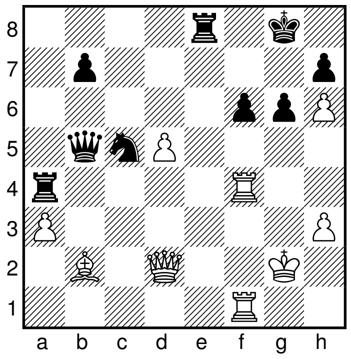
White's turn
Best move: Bc3

ECM position 403



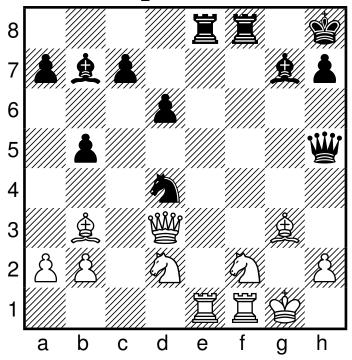
White's turn
Best move: Rxh5

ECM position 409



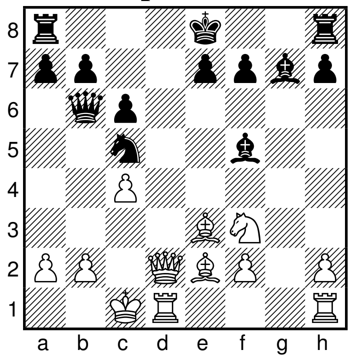
White's turn
Best move: Rxf6

ECM position 440



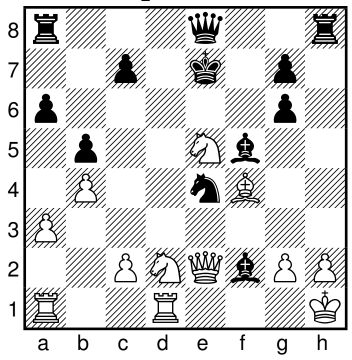
Black's turn
Best move: Rf3

ECM position 441



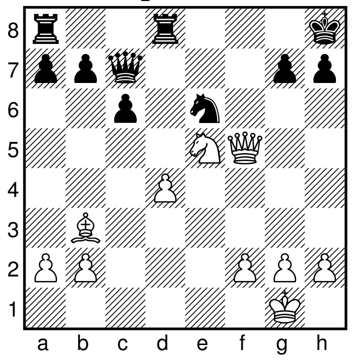
Black's turn
Best move: Nb3+

ECM position 449



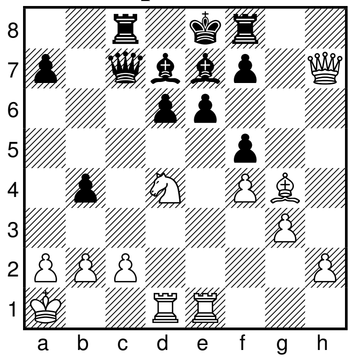
Black's turn
Best move: Rxh2+

ECM position 454



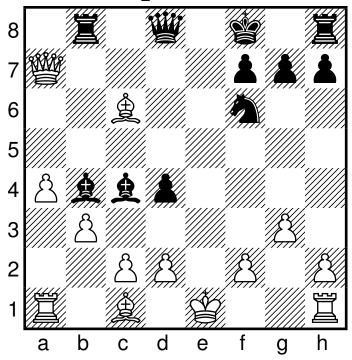
White's turn
Best move: Ng6+

ECM position 466



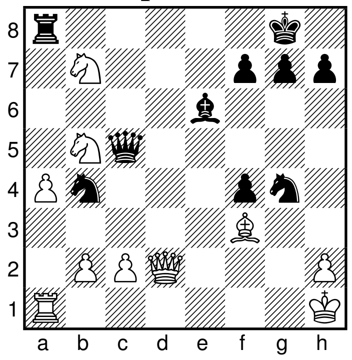
White's turn
Best move: Bxf5

ECM position 479



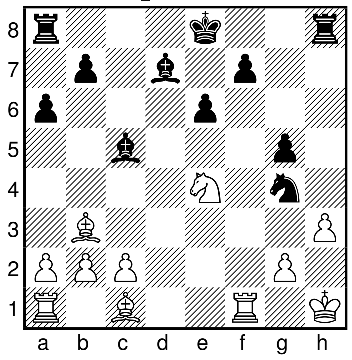
Black's turn
Best move: Qe8+

ECM position 481



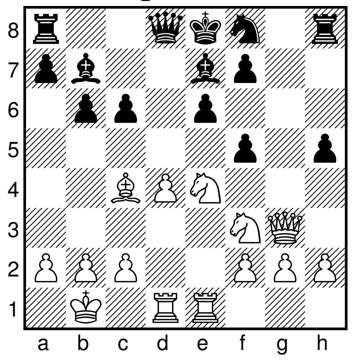
Black's turn
Best move: Qxb5

ECM position 506



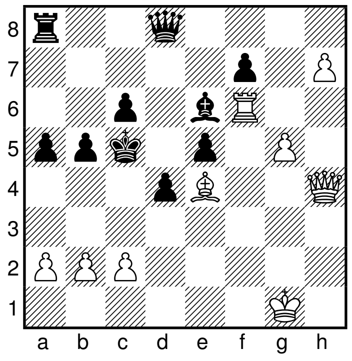
Black's turn
Best move: Bc6

ECM position 511



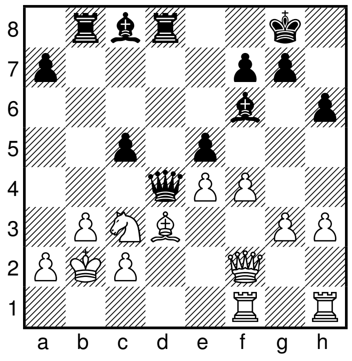
White's turn
Best move: d5

ECM position 521



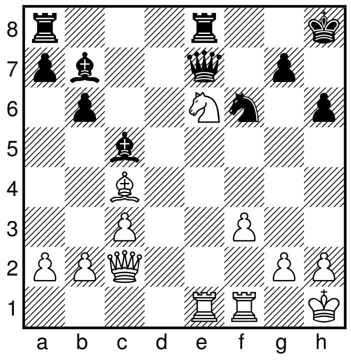
White's turn
Best move: g6

ECM position 523



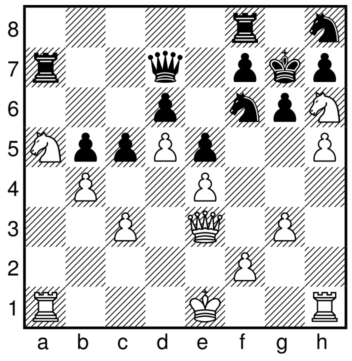
Black's turn
Best move: c4

ECM position 537



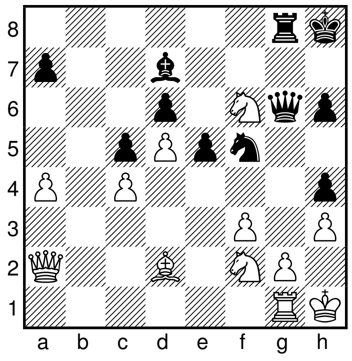
White's turn
Best move: Nf4

ECM position 540



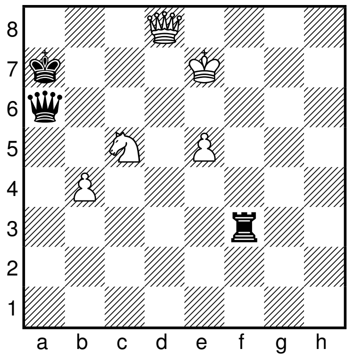
White's turn
Best move: Nf5+

ECM position 541



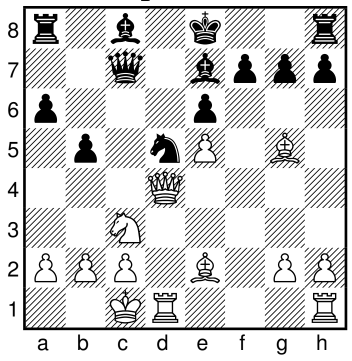
Black's turn
Best move: Ng3+

ECM position 550



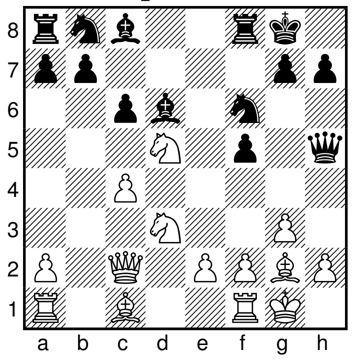
Black's turn
Best move: Rf7+

ECM position 594



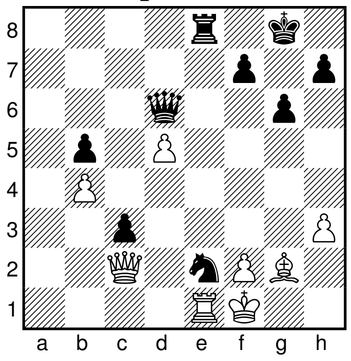
White's turn
Best move: Bxe7

ECM position 597



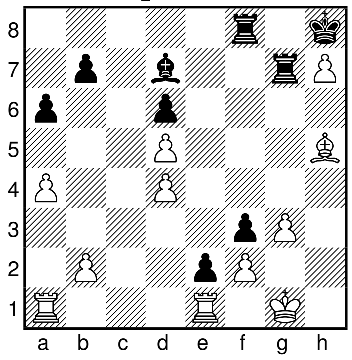
White's turn
Best move: Bf4

ECM position 612



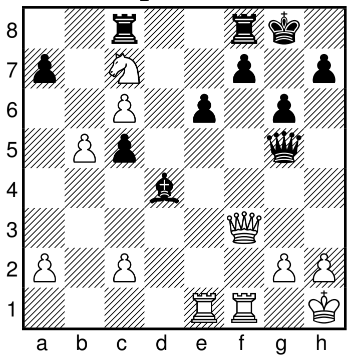
Black's turn
Best move: Ng3+

ECM position 620



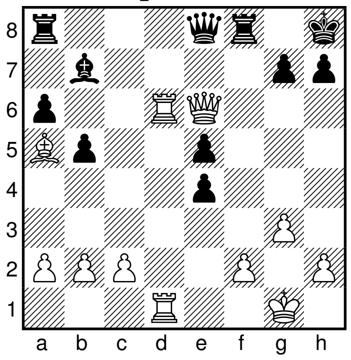
Black’s turn
Best move: Rxg3+

ECM position 625



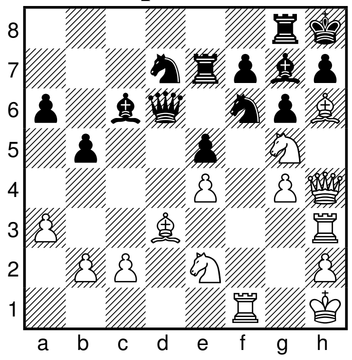
White’s turn
Best move: Nxe6

ECM position 637



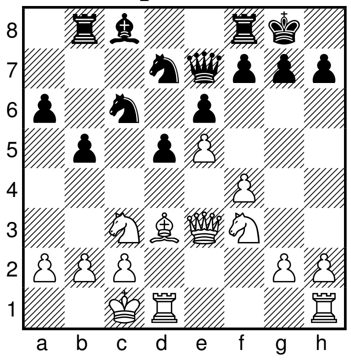
Black’s turn
Best move: Qh5

ECM position 655



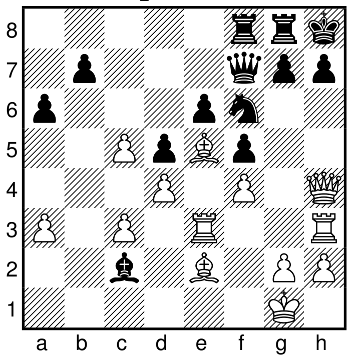
White’s turn
Best move: Nxh7

ECM position 656



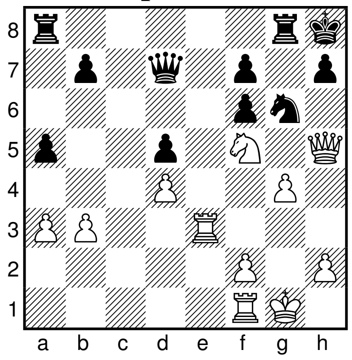
White’s turn
Best move: Bxh7+

ECM position 658



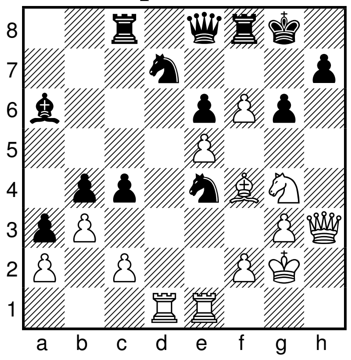
White’s turn
Best move: Qxh7+

ECM position 661



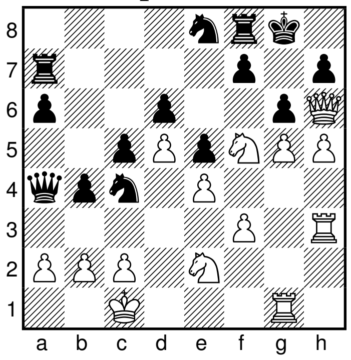
White’s turn
Best move: Qxh7+

ECM position 669



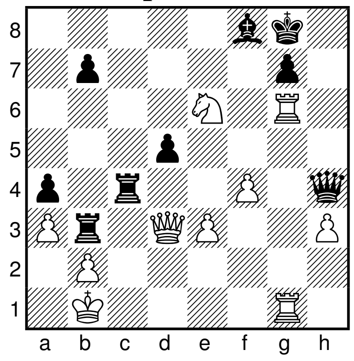
White’s turn
Best move: Qxh7+

ECM position 691



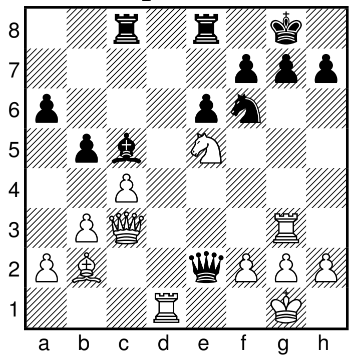
White’s turn
Best move: Qxh7+

ECM position 707



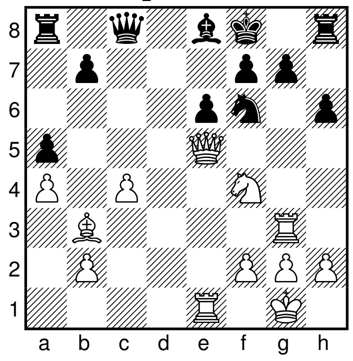
White's turn
Best move: Rxg7+

ECM position 708



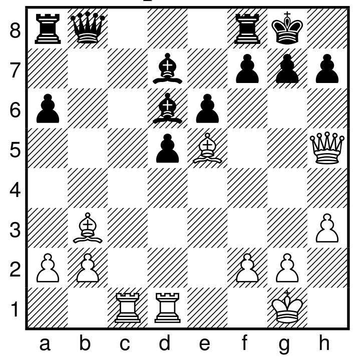
White's turn
Best move: Rxg7+

ECM position 710



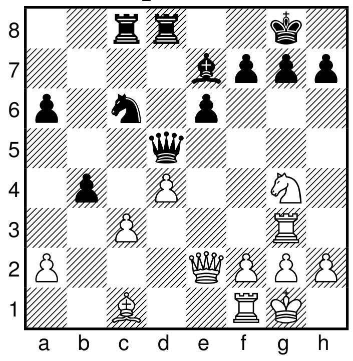
White's turn
Best move: Rxg7

ECM position 724



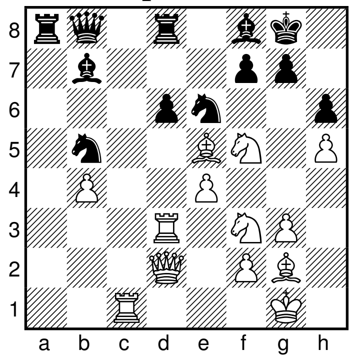
White's turn
Best move: Bxg7

ECM position 727



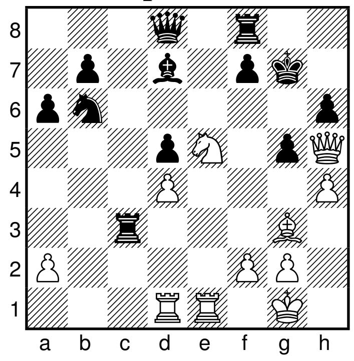
White's turn
Best move: Nh6+

ECM position 728



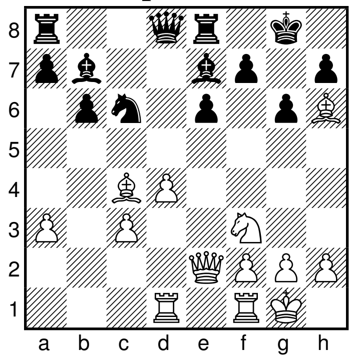
White's turn
Best move: Bxg7

ECM position 730



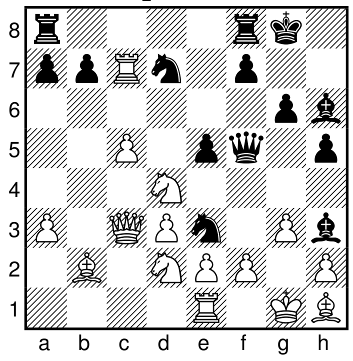
White's turn
Best move: Nxf7

ECM position 742



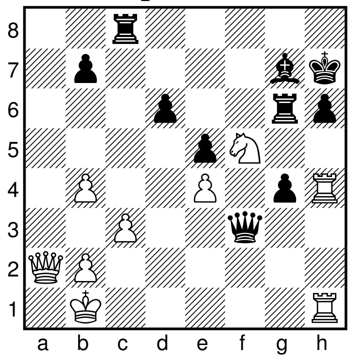
White's turn
Best move: d5

ECM position 751



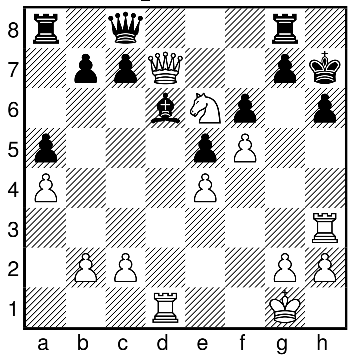
Black's turn
Best move: Qxf2+

ECM position 775



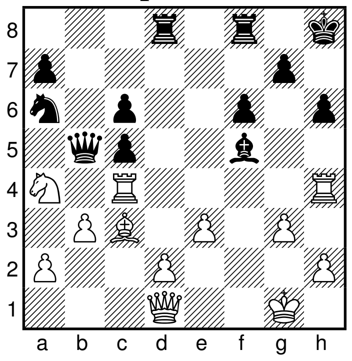
White's turn
Best move: Rxh6+

ECM position 777



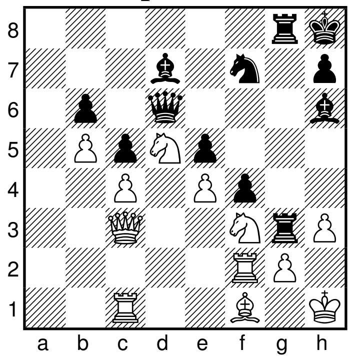
White's turn
Best move: Rxh6+

ECM position 781



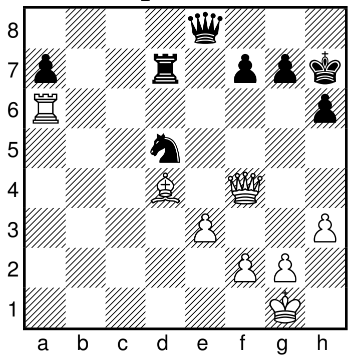
White's turn
Best move: Rxh6+

ECM position 783



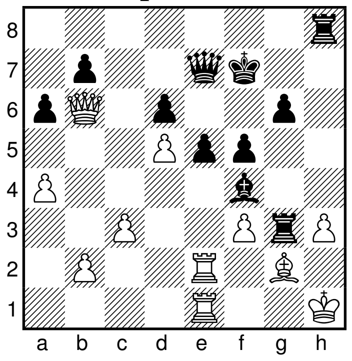
Black's turn
Best move: Bxh3

ECM position 786



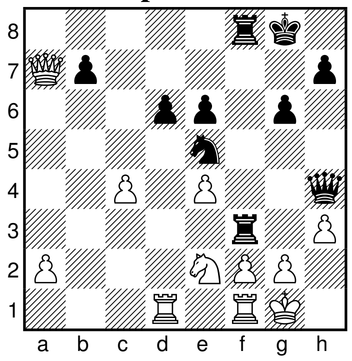
White's turn
Best move: Rxh6+

ECM position 787



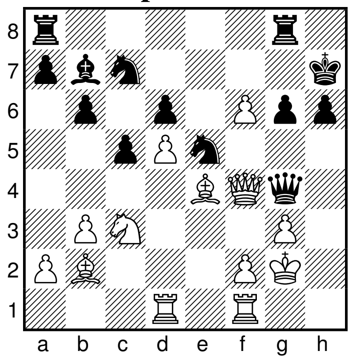
Black's turn
Best move: Rhxh3+

ECM position 788



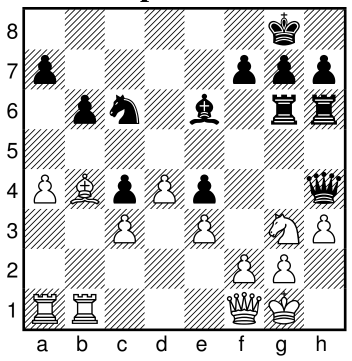
Black's turn
Best move: Rxh3

ECM position 793



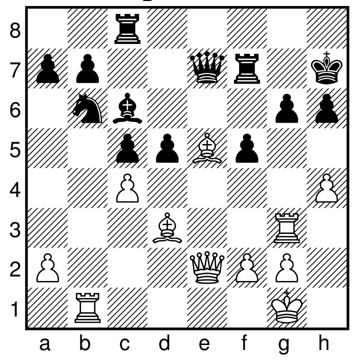
White's turn
Best move: Qxh6+

ECM position 795



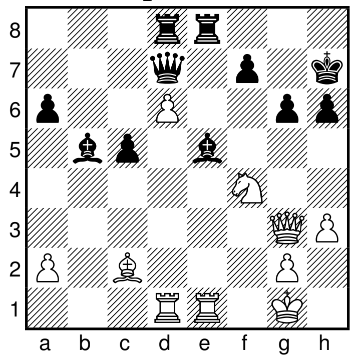
Black's turn
Best move: Bxh3

ECM position 811



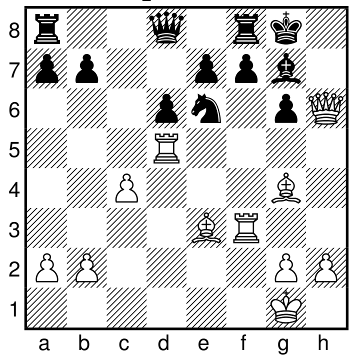
White’s turn
Best move: Rxg6

ECM position 812



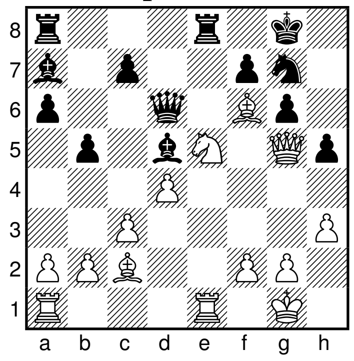
White’s turn
Best move: Bxg6+

ECM position 813



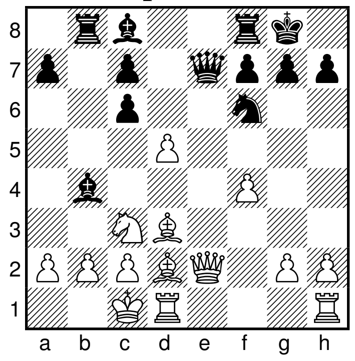
White’s turn
Best move: Qxg6

ECM position 819



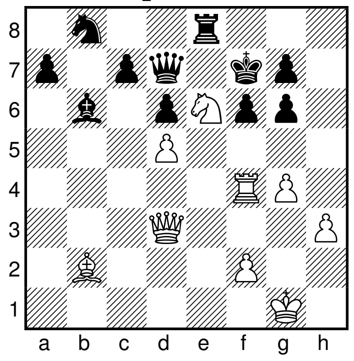
White’s turn
Best move: Nxg6

ECM position 822



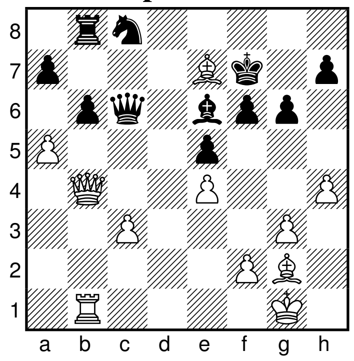
Black’s turn
Best move: Ba3

ECM position 830



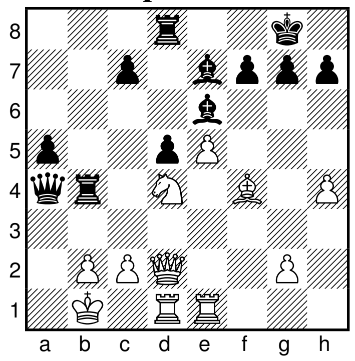
White’s turn
Best move: Bxf6

ECM position 835



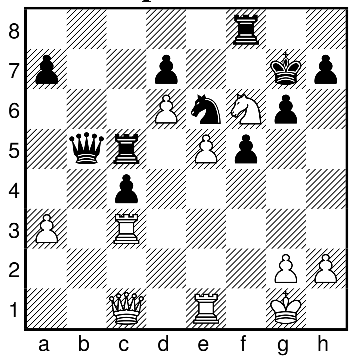
White’s turn
Best move: Bxf6

ECM position 850



Black’s turn
Best move: Rxb2+

ECM position 873



White’s turn
Best move: Qh6+