Learning to play chess selectively by acquiring move patterns

Lev Finkelstein and Shaul Markovitch
Computer Science Department
Technion, Haifa 32000
Israel
shaulm@cs.technion.ac.il

Abstract

Several researchers have noted that human chess players do not perceive a position as a static entity, but as a collection of potential actions. Indeed, it looks as if human chess players are able to follow promising moves without considering all the alternatives. This work studies the possibility of incorporating such capabilities into chess programs. We present a methodology for representing *move patterns*. A move pattern is a structure consisting of a board pattern and a move that can be applied in that pattern. Move patterns are used for selecting promising branches of the search tree, allowing a narrower, and therefore deeper, search. Move patterns are learned during training games and are stored in an hierarchical structure to enable fast retrieval. The paper describes a language for representing move patterns, and algorithms for learning, storing, retrieving and using them.

1 Introduction

Assume that a human chess player faces the board shown in Figure 1. Even a novice would realize that the most promising action is to make a fork. The human player can do so immediately without considering all the alternative moves [9]. Some people attribute such capability to intuition. Others to experience. But is it possible to endow computers with such capability: Selecting a good action without necessarily resorting to search?

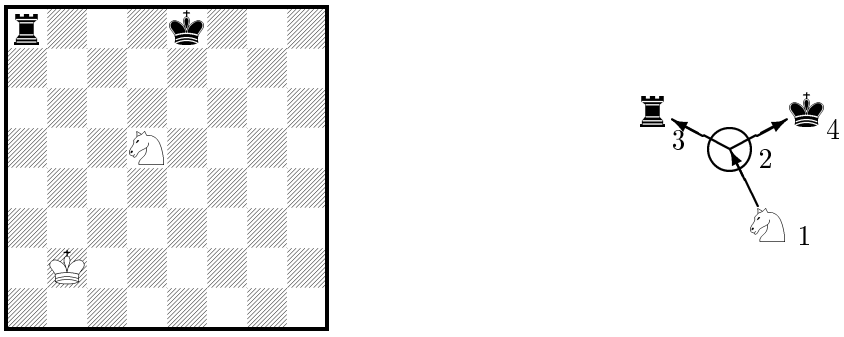


Figure 1: An example of a fork. A human player would identify the pattern illustrated to the right of the board.

The basic algorithm used by most of the current chess playing programs is the minimax algorithm proposed by Shannon [26, 27], assisted by the $\alpha\beta$ pruning method [23]. Chess search trees are growing very fast and various methods, mostly domain independent, have been developed for their selective deepening and pruning such as *quiescence* search [27, 29, 2], *singular extensions* [1] and B* [4]. Human players commonly use domain-specific knowledge to decide which branches of the search tree should be pruned [9]. Such an approach, when used by computer programs, is called *plausible-moves generation*. Using this approach, we can reduce the branching factor significantly, therefore allowing the search procedure to reach much deeper levels of the game tree. Wilkins' PARADISE [32], for example, can reach a depth of 19 plies.

Selecting a set of plausible moves out of the set of possible moves is a generalization of the very common problem where an agent must choose between a set of alternative actions. One approach to solve this problem is to pre-program the agent with a domain-specific decision procedure. In complex domains, such as chess, however, such an approach is extremely difficult. An alternative approach is to endow the agent with the capability of learning this decision procedure from its experience. This problem of learning to act is the primary interest of researchers in the field of *reinforcement learning* [17, 30]. However, since most of the algorithms developed in this field require explicit representation of all the possible states, they are not applicable to complex domains with very large state spaces such as chess. Attempts have been made to generalize over states and actions using feature vectors to allow handling a larger number of states. However, feature-based representation is not powerful enough to describe domains with highly-structured states.

A comprehensive survey of recent work in the area of learning in chess can be found in [12]. An interesting early attempt of learning to act in chess was undertaken by Pitrat [24], whose system tried to learn winning moves by analyzing game trees. Minton [22] developed a method, based on Pitrat's approach, for recognizing "good" moves. A learning procedure is supplied with examples of moves that are considered to be beneficial and tries to determine the weakest preconditions that enable the application of such moves. The learned knowledge is encoded into production rules which recommend a set of moves for positions that match the preconditions. Minton noted that his method was not efficient for chess learning due to the complexity of the game, and therefore applied it to the simpler domain of Go-Moku. Other programs are able to learn moves [10, 31, 25], but they focus on small sub-domains of chess.

CHUMP [14] uses a pattern-based approach to acquire patterns of boards with associated moves. The patterns are extracted during learning using an eye-movement simulator [28]. The Tal program [11] also acquires patterns called *chunks* with associated moves. The patterns used by CHUMP are perceptual and are therefore limited in their generalization capabilities [14]. A recent work by Levinson [19, 18, 20] introduces a new approach for representing generalizations of structural knowledge. His approach assumes that a state can be described by a set of objects and relations between them. Such a state is represented by a general graph, and the subgraph relation is used to generate generalizations (called *patterns*). The graphs are organized in a hierarchy, allowing a fast retrieval of states. This representation scheme was tested with the Morph system which uses patterns to learn evaluation functions for chess [20].

The work presented in this paper takes the pattern-based move-oriented approach similarly to Gobet and Jansen [14], but uses an extension of the powerful pattern language designed by Levinson for representing move patterns. This language allows us to learn generalizations that save learning time, storage space and utilization time. For example, Figure 2 of Section 2.2

shows how to encode moving into a fork position (see Figure 1) regardless of the particular location of the position on the board.

The basic principle behind the move-oriented approaches is to allow evaluation of moves without an explicit exploration of their outcomes. Therefore, whereas the basic knowledge representation element of Morph is a *board* pattern, the basic element of our approach is a *move* pattern. A move pattern is a structure consisting of a board pattern and a move that can be applied in that pattern. Each move pattern has an associated weight indicating the potential benefit of applying the move in the given context.

Move patterns allow a chess program to reason about moves in a way that is similar to the way that human players reason [8, 9]. In Table 1 we show several simple situations where using a plausible-move generator can save from one to three levels of search. The arrows stand for attack/defend relationships and the circles stand for empty squares. The particular representation will be further explained in Section 2. Here we just wish to illustrate the intuition behind move patterns.

Move patterns are used for pruning branches of game trees by having the search procedure explore only promising branches. The patterns are acquired during training games. They are either extracted from the boards using an explanation-based generalization (EBG) approach similar in spirit to Morph's method, or are generated by intersecting existing patterns. Weights are updated according to the change in the board value.

The rest of the paper is organized as follows. Section 2 describes the representation language used for describing move patterns. Section 3 discusses the way that move patterns are organized and utilized. Section 4 describes learning algorithms for acquiring move patterns. Section 5 describes an implementation of the move pattern methodology. Finally, Section 6 concludes.

2 A language for representing move patterns

In this section we present a new pattern language which allows us to generalize over moves and their context. We start by defining *board patterns* – for representing the context of moves and continue with defining move patterns. Our language for representing board patterns is an extension of the knowledge representation scheme proposed by Levinson [19, 18].

2.1 Representing the context of a move pattern

A pattern is a generalization of an object that contains its essential features. The essential features of a chess board are the relationships within the set of pieces and squares. A natural method for representing relationships within a set of objects is a graph (or a hypergraph). The vertices of such a structure are the pieces and squares, and the edges are sets of vertices that describe their relations.

The basic relationships between chess pieces are *defense* and *attack*. A finer classification will distinguish between *direct* and *indirect* attack/defense. These types of relationships were used in Levinson's *Morph* system and are sufficient for representing the current static state of the board. For example, look at the board shown in Table 2. The left graph shows the above relations for this board: the black rook attacks a white pawn, the white king defends the white rook etc. Since these relations reflect only the current board, we call them *static* relations and the corresponding patterns will be called *static*. Table 3 shows the two types of edges in static patterns.

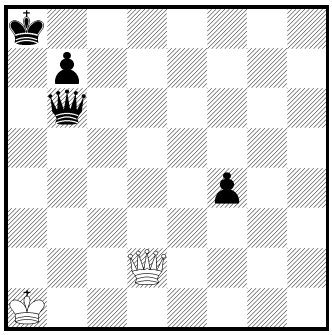
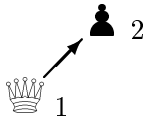
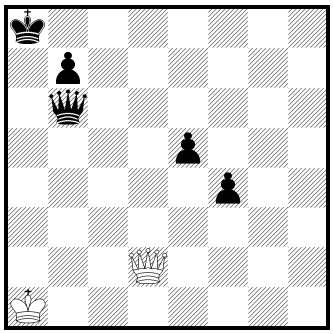
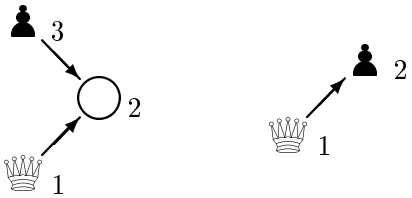
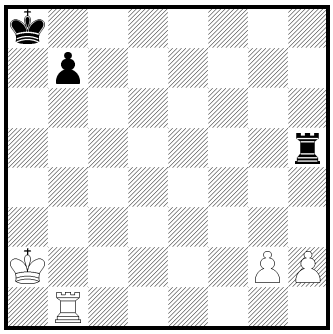
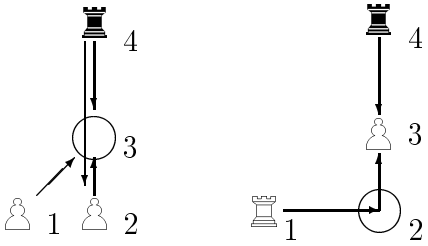
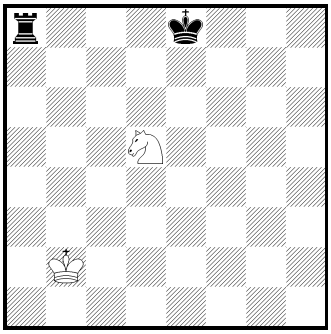
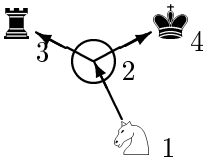
		White can win a pawn by identifying the pattern to the right of the board without exploring all the 24 legal moves.
		White can see that capturing a pawn will cause the loss of the queen and prunes this branch by combining the two patterns. It saves 2 levels of search.
		White can identify the two options available for defending his pawn, saving a search of three levels.
		An example of a fork. A human player will identify the shown pattern. Minimax will find this move only after 3-level analysis.

Table 1: The intuition behind the move pattern approach.

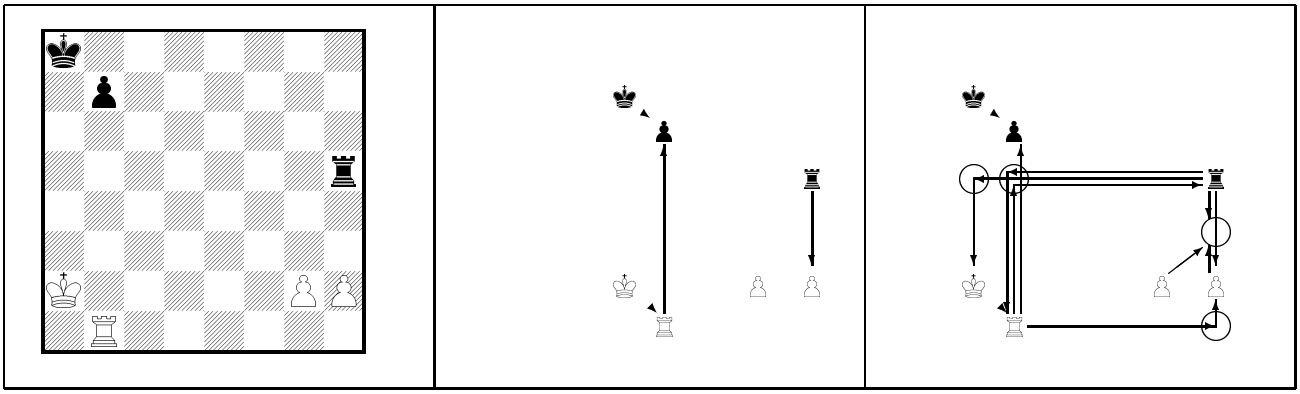


Table 2: An example of a board with its associated patterns. The patterns in the center cell are static. The pattern in the right cell is dynamic.

Edge type	Arity	Meaning	Example
F_1F_2	2	F_1 directly attacks/defends F_2	
$F_1F_2F_3$	3	F_1 indirectly attacks/defends F_3 through F_2	

Table 3: The two types of edges in static patterns. F stands for "Figure" (or pawn).

Static patterns, however, have limited representation power. For example, while the static language enables the representation of the black's ability to capture the pawn at h2, it is unable to represent the two ways for the white to avoid it (♖h1 or h3). Therefore, we extend the pattern language to allow representation of *dynamic* relations. Such relations represent the ability to move and enable us to predict the consequences of move application. There are many possible ways to define dynamic relations. To keep the patterns reasonably compact, we restrict our language to one-level extension of the static relations: the set of relations sufficient to describe all the static relations after performing one move. For example, after playing the move ♖b1–h1 in the position described by the board in Figure 2, the static relation "white rook defends white pawn" (of the type FF) becomes true. The corresponding dynamic relation in the current board will be "a white rook can move to a square from which it defends a white pawn". Table 4 describes the set of dynamic relations. It is possible to show that this set is sufficient for describing all the preconditions of static patterns. The right graph in Table 2 shows some of the significant dynamic edges corresponding to the board at the left.

Given a set of relations such as the one described in Table 4, we define the *full graph* of a board to be the graph describing *all* the relations in the board. A *board pattern* is any subgraph of the full graph. The subgraph relation defines a generalization hierarchy over the set of board patterns.

2.2 Representing moves of move patterns

The main goal of this research is to endow a system with the capability of learning move-preferences. Such learning requires a generalization over board-move pairs. The previous

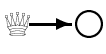
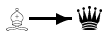
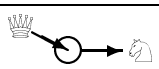
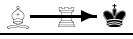
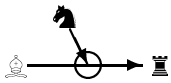
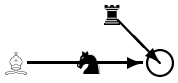
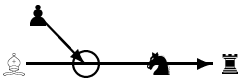
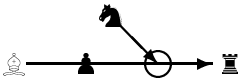
Edge type	Arity	Meaning	Example
FS	2	F controls square S	
F_1F_2	2	F_1 directly attacks/defends F_2	
F_1SF_2	3	F_1 controls square S, from where it would attack/defend directly F_2	
$F_1F_2F_3$	3	F_1 attacks/defends F_3 indirectly through F_2	
$F_1SF_2F_3$	4	F_1 can move to square S, from where it will attack/defend F_3 indirectly through F_2	
$F_1F_2SF_3$	4	F_1 directly attacks/defends F_3, and F_2 can be wedged between them on square S	
$F_1F_2F_3S$	4	F_3 can move to square S, from where it will be attacked/defended by F_1 indirectly through F_2	
$F_1F_2F_3F_4$	4	F_1 attacks/defends F_4 indirectly through F_2 and F_3	
$F_1F_2SF_3F_4$	5	F_1 attacks/defends F_4 indirectly through F_3, and F_2 can be wedged between F_1 and F_3 on square S	
$F_1F_2F_3SF_4$	5	F_1 attacks/defends F_4 indirectly through F_2, and F_3 can be wedged between F_2 and F_4 on square S	

Table 4: The types of edges in dynamic patterns. F stands for "Figure" (or pawn) and S stands for "Square".

subsection presented a methodology for generalizing boards. In this subsection we describe a representation scheme for a generalization over moves within a context of a certain board. A move pattern is a pair, where the first element is a board pattern with its vertices uniquely labeled, and the second element is a pair of the labels corresponding to the source and target vertices. A move pattern can be generalized by generalizing over its board graph component. The only restriction is that the nodes appearing in the move component of the move pattern should not be deleted. In order to be able to use move patterns for determining precedence, we associate a weight with each move pattern such that the larger the weight is, the more attractive we consider this move pattern to be. The way that weights are assigned to move patterns will be discussed in Section 4. Figure 2 shows an example of a move pattern with its associated weight. We use the notation $1 \xrightarrow{+R} 2$ to represent a move of a piece labeled by (1) to a square labeled by (2) with a weight of $+Rook$.

2.3 Null-move patterns

Null-move patterns describe the opponent's possible actions when our move does not change their local context. Unlike the move patterns described in the previous subsection, which can be used for both single-agent search and adversary search, null-move patterns are tailored

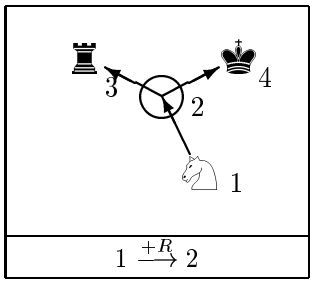


Figure 2: An example of a move pattern.

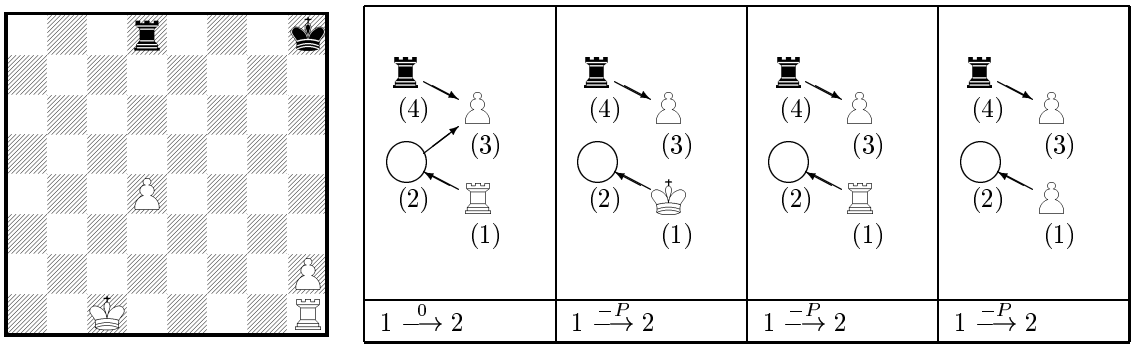


Figure 3: A set of move patterns describing that a pawn can be defended by a rook.

for adversary search. When a human player detects a threat, he implicitly assigns a negative weight to all the moves that do not cancel this threat (without gaining a higher advantage). Null-move patterns are an attempt to encode such reasoning into the move-pattern language. The best way to explain the need for null-move patterns is to consider the following examples.

Assume that we are given the board shown in the left part of Figure 3. How can we use move patterns to describe the situation, that black can capture pawn d4, and the only move which prevents it is ♖d1? If we use the method described in the previous section, we must have at least the 4 move patterns shown to the right of the board. From these patterns we can see that all the moves lead to loosing the pawn, except ♖d1 which saves it. This set

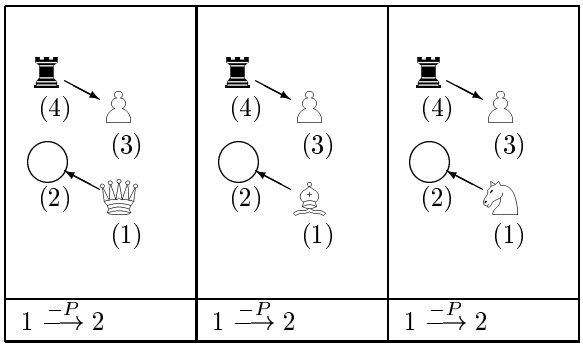


Figure 4: A set of additional move patterns describing that a pawn must be defended.

of move patterns does not cover positions with other white pieces. To cover such positions we need to add the 3 move patterns shown in Figure 4. We therefore need a set of 7 move patterns: 6 patterns telling that moves not related to the pawn lead to loosing this pawn, and one showing how to defend it.

Assume now that we have the position shown in Figure 5 where the attacked pawn is already defended by another pawn. The above 6 patterns give negative weight to each move that does not defend the attacked pawn. The result will be an attempt by the program to create a multiple defense of this piece. To prevent such incorrect behavior, the above patterns must be overridden by the more specific patterns shown in the right part of Figure 5.

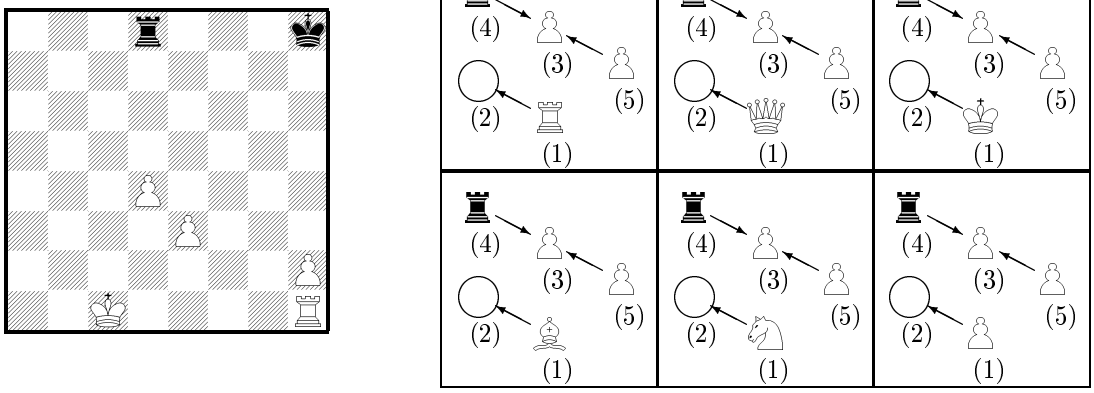


Figure 5: The above 6 patterns have an associated move $1 \xrightarrow{0} 2$. These patterns override the patterns of the previous figure to represent the fact that there is no need to defend a pawn, attacked by a rook, if it is already defended by another pawn.

It is easy to see that in each case when we specify the opponent's attack pattern, we need to add 6 additional patterns (for the six types of chess pieces). Multiplying the number of patterns by 6 would not be so harmful, if these patterns were not so general: since their graph structure is not connected (these patterns have at least 2 independent parts), the number of matches by such patterns for *any* board will be very large. As the result, the matching process will consume too much time, making the whole method inapplicable.

To overcome this problem we extend our representation scheme by a new type of move patterns, called *null-move patterns*, that allow considering opponent's threats. The idea is to evaluate the moves of the opponent as well as the moves of the player, and to add negative weight to all the player's moves that do not cancel the opponent's threat (if any). For example, for the board illustrated in Figure 3, only two move patterns are needed: "Black Rook can take a White Pawn" (a null-move pattern) and "White Rook can defend a White Pawn" (a regular move pattern). For the board in Figure 5, a single additional null-move pattern "Black Rook can take a defended White Pawn" is needed. These three patterns are shown in Figure 6.

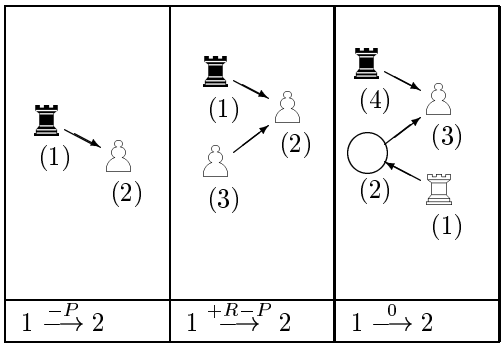


Figure 6: The three move patterns that are sufficient to describe a defense by a rook of a pawn attacked by a rook and a defended pawn. The left two are null-move patterns.

Now a question arises: How would we know that a player's move cancels an opponent's threat? Or, even more: How would we know that a player's move changes the evaluation of opponent's moves? To answer these questions, we further extend our representation scheme. In the extended scheme we associate with each move pattern the *explanation* for the weight given to it. For example, if we have a pattern "White Rook at (1) attacks Black Pawn at (2) defended by Black Pawn at (3)" (the reverted second pattern in Figure 6), we know that the *immediate* weight of the move "White Rook at (1) captures Black Pawn at (2)" is $+Pawn$, but the weight of the opponent's possible *reply* "Black Pawn at (3) captures White Rook at (2)" will be $-Rook$. [1]

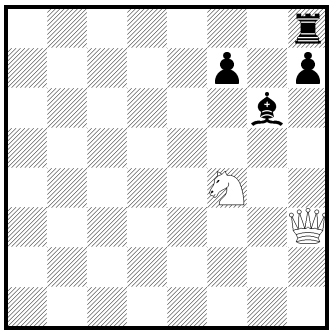
The proposed scheme provides a very powerful tool for conflict resolution. Look for example at the board and the corresponding patterns shown in Figure 7. The notation $1 \xrightarrow{+B} 2 \Longrightarrow \left\{3 \xrightarrow{-N} 2\right\}$ represents a move from (1) to (2) weighted $+Bishop$ with an associated reply from (3) to (2) weighted $-Knight$. If we do not separate a move from its possible replies, determining the weight of the move ♘f4×g6 becomes very tricky. For example, *Morph* would evaluate such a position as winning a bishop (see [20]), because a more general board pattern (the second from the left) is suppressed by a more specific one (the third from the left). If we use a reply information, however, we can see that in the case of ♘f4×g6 we *will not* win a bishop, since the reply f7×g6 matches the second pattern, but does not match the third one.

3 Organization and utilization of move patterns

To make the above representation scheme usable, we must design a structure for efficient storage and retrieval of move patterns. The generalization relation *subgraph-of*, described in Section 2, defines a directed acyclic graph over the set of move patterns. We store move patterns and null-move patterns according to their board patterns. We therefore developed algorithms for inserting, updating and utilizing move patterns that are based on those developed by Levinson [19, 20]. We incorporated, however, many changes to handle the additional complexity of the move pattern hierarchy:

1. The database is stored in a directly acyclic graph structure (DAG). Each node corre-

[1] We mark White Rook in the reply by (2) since after capturing a Black Pawn it takes its place.



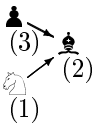
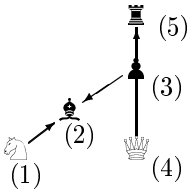
			
Without reply	$1 \xrightarrow{+B} 2$	$1 \xrightarrow{+B-N} 2$	$1 \xrightarrow{+B} 2$
With reply	$1 \xrightarrow{+B} 2 \Longrightarrow \{\}$	$1 \xrightarrow{+B} 2 \Longrightarrow \left\{3 \xrightarrow{-N} 2\right\}$	$1 \xrightarrow{+B} 2 \Longrightarrow \left\{3 \xrightarrow{+R-N} 2\right\}$

Figure 7: A partially specified board (white to play). Without the reply information, the move Nf4-g6 is erroneously considered to win a bishop. With the reply information it is correctly evaluated to exchange a knight for a bishop.

sponds to a set of weighted move patterns sharing a common board pattern.

2. Each node is linked to all of its immediate successors. A pattern, however, can be a generalization of another pattern in more than one way, depending on the mapping of the vertices and edges. To make the search more efficient and to avoid redundant pattern matching, we label each link with its associated mapping. In this way we preserve the pattern matching information obtained while moving down the pattern hierarchy.

3. Patterns are inserted into the database using the same principles as proposed by Levinson [19, 20]. The extended links, however, complicates the implementation of the algorithm.

4. To evaluate the moves for a particular board, we first represent it by its associated graph. We then perform the matching process from the top of the hierarchy in a depth-first fashion.

5. The central procedure of the whole process is pattern matching. Each edge is uniquely encoded as a 32-bit integer representing its type (from the list of Table 4) and the color and types of its participating figures. The edges of each pattern are sorted by their code. The encoding is designed to embed two heuristics into the order such that the more restricting an edge is and the more vertices it instantiates, the higher priority it gets. This implementation allows a very rapid pattern matching in most practical cases. In particular, more complex patterns are often matched faster than simple ones.

6. While traversing the pattern hierarchy we use the above method to match the graph representing the input board to the individual patterns. For each pattern, we get a list of mappings, representing how the graph associated with the input board can be mapped. These mappings prevent us from making errors due to the ambiguity of the subgraph relation. The moves available in the given pattern are then translated to the notation of the original board.

7. During the search, we mark the visited nodes to avoid repeated traversal of their descendents. We also make sure that the same move will not be propagated up through two paths.

8. If the same move gets different evaluations from different matching move patterns, then the algorithm should resolve the conflict.

   Look, for example, at the board and the associated knowledge-base shown at Figure 8. The move instances matching each one of the move patterns are shown in the table in the same figure. The utilization scheme that combines the estimates coming from different move patterns is rather complex and works using the following basic principles:

   - The gold rule of hierarchical systems: if the same move matches two move patterns connected by the relation subgraph-of, only the value for the most specific pattern is taken into account. For the example above, we do not consider the evaluation of the move ♕×f4 from the pattern "White Queen moves to a square".

     The same is correct for the null move patterns: Since a move pattern "White Queen can Defend White Pawn from Black Queen" is more specific than "Black Queen attacks White Pawn", we do not consider the evaluation of the latter while calculating weight of the move ♕e2.
   - Moves with positive and negative evaluations are evaluated separately, and mixed only at the final stage. If two move patterns, coming from different branches, match the same move, the move will be assigned the larger weight.

     For example, Figure 9 shows a position where the move ♘c7 gets a weight of $+Queen$ from one move pattern and a weight of $+Rook$ from the other. The move will be assigned the weight of $+Queen$.
   - A similar reasoning applies for the weights of moves obtained from null move patterns, except that the smaller weight is used.
   - At the final stage of the weight evaluation, the weight associated with the opponent's moves is subtracted from the weight associated with the player's moves.

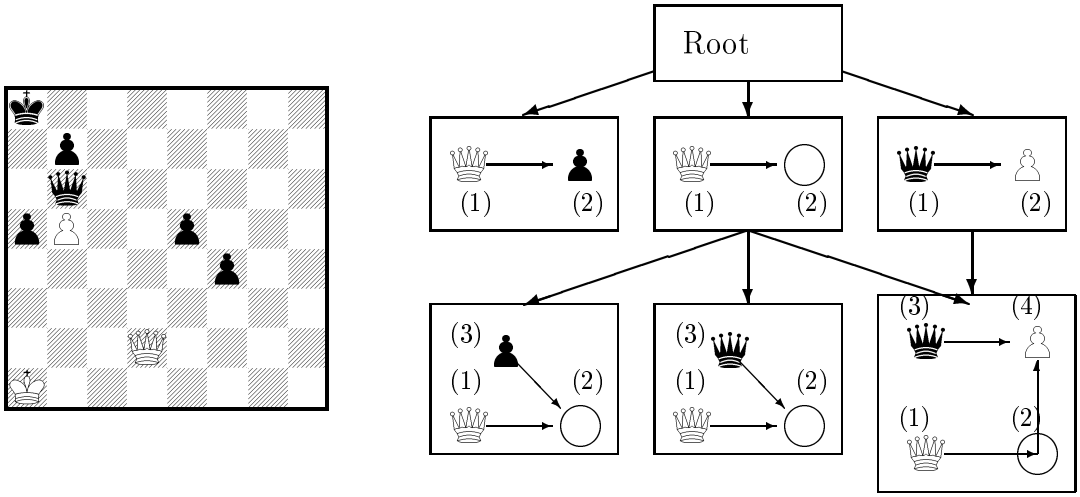
9. The list of returned moves is sorted by their weights. CHUMP [14] uses a similar method but also considers the number of patterns that support each move.

The following weights are assigned to the moves shown in Figure 8:

1. Moves ♕b4, ♕d4, ♕d6, ♕d8, ♕e3, ♕f2 get an estimate of $-Queen$.
2. Moves ♕×f4 and ♕×a5 get an estimate of $+Pawn - Queen$.
3. Moves ♕b2, ♕d3, ♕d5, ♕d7, ♕e2 get an estimate of 0.
4. All the rest of the moves get an estimate of $-Pawn$ since the response of black will be ♕×b5.

4 Learning move patterns

In the previous section we showed how can move patterns be utilized to decide which move to take. In this section we describe algorithms for learning move patterns from examples. Move patterns are learned while playing against an external teacher. This teacher can be a human, another chess program or the learning program itself. The reversed trace of the game is passed to the learner which extracts training examples and generalizes them in the following way:



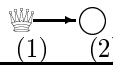
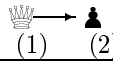
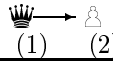
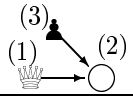
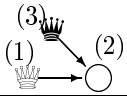
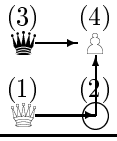
Patterns	Weighted moves and replies	Matching moves and replies
	$1 \xrightarrow{0} 2$	All white moves by queen
	$1 \xrightarrow{+P} 2$	1.Qf4, 1.Qa5
	$1 \xrightarrow{-P} 2$	1. ... Qb5
	$1 \xrightarrow{0} 2 \Longrightarrow \left\{3 \xrightarrow{-Q} 2\right\}$	1.Qb4 ab; 1.Qd4 ed; 1.Qf4 ef; 1.Qe3 fe
	$1 \xrightarrow{0} 2 \Longrightarrow \left\{3 \xrightarrow{-Q} 2\right\}$	1.Qa5 Qa5; 1.Qd4 Qd4; 1.Qd6 Qd6; 1.Qd8 Qd8; 1.Qe3 Qe3; 1.Qf2 Qf2
	$1 \xrightarrow{0} 2 \Longrightarrow \left\{3 \xrightarrow{-P+Q} 4\right\}$	1.Qa5 Qb5; 1.Qb4 Qb5; 1.Qb2 Qb5; 1.Qd3 Qb5; 1.Qd5 Qb5; 1.Qd7 Qb5; 1.Qe2 Qb5

Figure 8: An example of utilization of move patterns. The pattern database (board patterns only) is shown to the right of the board. The results of the matched move patterns are shown below.

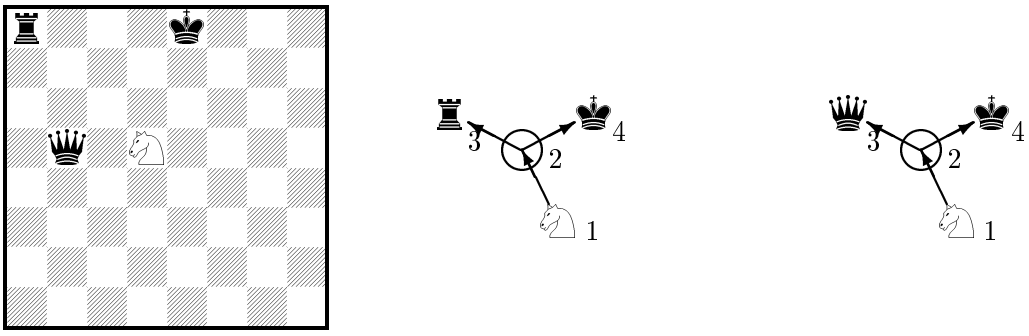


Figure 9: Although the two move patterns indicating the winning of queen and rook after Nc7+, only one figure can actually be captured. Therefore, the move will be assigned a weight of $+Queen$.

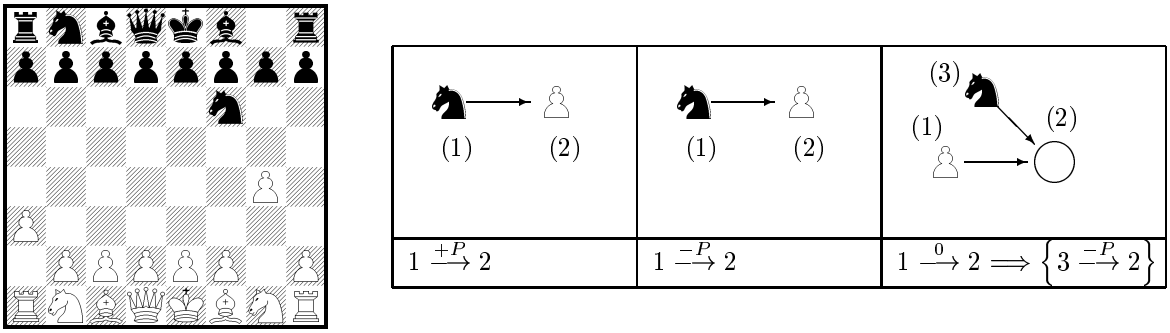


Figure 10: Move patterns which can be generated from the trace 1. a3 Nf6 2. g4 Ng4 ... and the board before 2. ... Ng4.

1. *Training examples:*

   (a) When the evaluation of a move before it is performed is very different from its evaluation *after* it is performed (the evaluation of the best opponent's response), we mark the board with its associated move as a training example. This is an instance of the known learning strategy *learning by surprise*. This method forces the system to learn increasingly complicated patterns trying to avoid being surprised. We call patterns generated from such examples *material patterns* since they usually represent a change in the material balance of the board. A fork pattern (see Figure 1) is an example of a material pattern.

   Assume that a learner gets the following trace of a game: 1. a3 ♘f6 2. g4 ♘g4 etc.[2] Since the trace is reversed, the learner will discover that the move 2. ... ♘g4 wins a pawn. The move pattern reflecting this fact and the corresponding null-move pattern (its reflection) are shown as the two leftmost move patterns in Figure 10. After the patterns are acquired, the learner can infer that the move 2. g4 was a mistake, and acquires an additional move pattern (with reply) shown as the rightmost pattern in Figure 10.

[2]The learner plays white. Such a game can take place, since at the beginning of the learning process the moves are almost random.

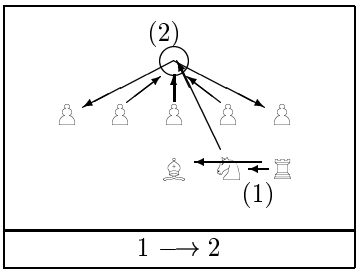


Figure 11: An example of a positional pattern.

(b) If a move is performed by a stronger opponent (a teacher), then the learner assumes that it is better than its alternatives, and takes it as a positive example. We call patterns generated from such boards *positional patterns*. An example of such a pattern is shown in Figure 11. This pattern is likely to be one of the first moves of gnuchess which was used as an external teacher in the experiments described in the following section and was learned by our program during its first game. Following the acquisition of this pattern the program used it to perform ♘f3 or ♘c3 as one of its first moves.

2. *Self-generated training examples:*

The training examples described above are all a result of actual moves played during the training games. Sometimes, we would like to learn from moves that were *not* played during the game, but are believed to potentially carry important information. In such a case, the learning algorithm simulates the unplayed move and generates an example that is then processed in a regular way.

(a) If the learning algorithm expects from the stronger opponent to make move X, but the opponent makes move Y, then move X is analyzed more carefully (a deeper search is performed). The algorithm assumes that the opponent knows something about X that is not known to the learner yet, and therefore X should be explored. The result is a move pattern for X that takes into account a deeper lookahead.

Assume that the learner knows some material patterns like those shown in Figure 10, but is not familiar with the pattern shown in Figure 12. Suppose now that it examines its game (as white) against a stronger opponent, and analyzes a game trace such as 1.e3 ♘f6 2. g4 d5 etc. While analyzing the move 2. ... d5 the learner will be surprised, because it would expect 2. ... ♘g4. The deeper analysis of 2. ... ♘g4 discovers the reply 3. ♕g4, and the pattern from Figure 12 will be acquired too.

(b) When the learner gives a negative evaluation to all possible moves, it considers the current board as a local minimum. To try escaping from the local minimum, a deeper search is performed. If an escape move is found, then the corresponding move pattern is acquired.

Suppose, for example, that the learner knows the patterns shown in Figure 10, but still does not know how to defend an attacked pawn. After 1.e4 ♘f6 the learner will realize that all the moves lose at least a pawn. The deeper analysis, however, shows that the pawn can be defended by moves 2.♘c3, 2.♕e2, 2.♕f3, 2.d3,

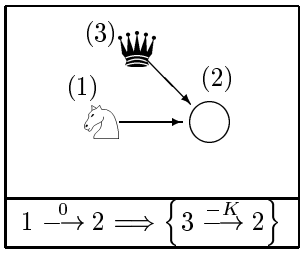


Figure 12: A move pattern that can be learned by self-generating training examples from the trace 1.e3 Nf6 2. g4 d5.

2.f3, 2.♗d3, and that it is possible to escape the treat by 2.e5. As a result, the patterns shown in Figure 13 will be added to the knowledge base.

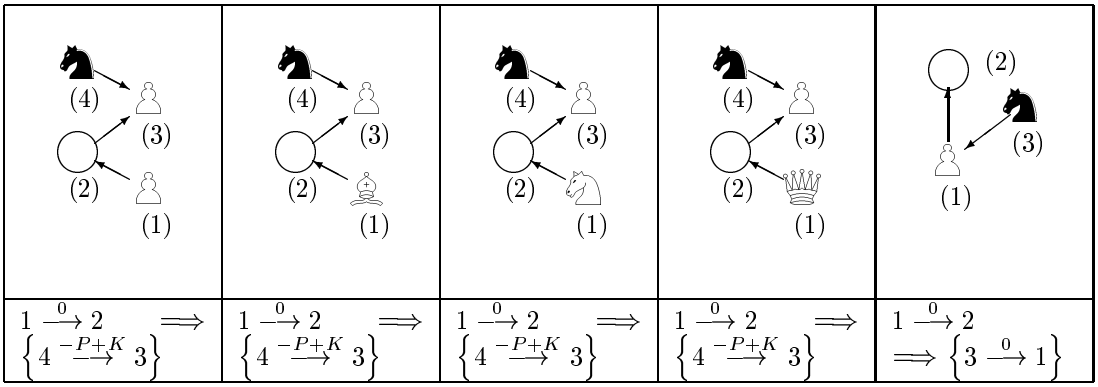


Figure 13: Move patterns that can be learned by self-generating training examples from the trace 1.e4 Nf6.

3. *Extracting patterns from examples:* The outcome of the previous stage is a specific move pattern which is used as the raw example. In the next stage the learner performs Explanation-based generalization similar to the one used by Minton [22]. The learner tries to find minimal preconditions that preserves the "surprise" (the difference between pre-evaluation and post-evaluation) of the move. For positional patterns we eliminate all pieces and squares that have no relation to the move. For material patterns we try to remove pieces one by one. When any removal of a piece changes the difference of evaluation, the process stops and the generalized pattern is acquired.

   When the learner gets as a training example the board from Figure 10 and the move 1. ... ♘g4, it deletes from the board almost all the pieces except the knight and the pawn, and gets the two first patterns shown to the right of the board. These patterns are acquired at the same step, because one of them is a reflection of the other.

4. *Assigning weights to the acquired move patterns:* In the case of a material pattern we set its weight to the value of the material difference. For positional patterns, we use the frequency of its usage by the teacher to determine its weight. To enhance the statistics, we employ a process called *augmentation*. A set of examples of moves, either from a set of games played between two copies of the teacher, or from master databases, is

passed through the pattern-hierarchy to update the statistics of the positional patterns (without allowing the acquisition of new patterns).

5. *Incorporating new move patterns into the knowledge base:* The new move patterns are inserted into their location in the pattern hierarchy. The learner performs generalization of the acquired move pattern by intersecting it with existing ones. An example for such generalization is shown in Figure 14.

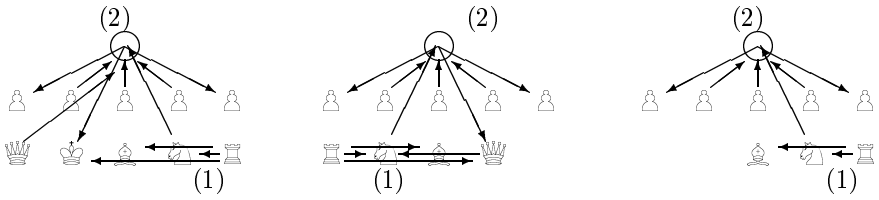


Figure 14: Generalization of the two positional patterns to the left yields the positional pattern to the right

5 SYLPH : a system that learns move patterns in chess

We have implemented the move-pattern learning methodology in a learning program called SYLPH[3]. The system starts with very little knowledge: The basic rules of chess, the set of predefined primitive relations (see Table 4) and the material value of pieces. SYLPH learns by playing against a stronger opponent and gradually acquiring move patterns, merging them into its move-pattern hierarchy.

We performed a set of experiments to test the plausibility of the move-pattern approach. We start by describing the experimental methodology and continue by showing the results obtained.

5.1 Experimental methodology

Our experimentation with SYLPH consists of a learning stage and a testing stage. The learning stage included two parts. First, SYLPH performed online learning through a sequence of 100 training games against a teacher (gnuchess). A snapshot of the acquired pattern knowledge-base was saved after every 10 games to allow us studying the progress of the system. This part was followed by an augmentation stage where the statistics of the acquired patterns were updated by observing 50 games between two copies of the teacher.

Since the goal of the move-pattern methodology is to allow selective search, we tested the acquired pattern knowledge-base by using it as a filter at the top level of the search tree. In Section 3, we describe the basic pattern-utilization function which, given a board, returns the set of possible moves ordered by their priority. We use this function to define two similar filters: f_k, which selects the best k moves, and f_γ, which selects the best γ portion of the moves. This experimentation methodology follows the approach taken by Gobet and Jansen [14].

[3]SYstem that Learns a Pattern Hierarchy.

A good filter is characterized by its ability to select a small number of moves without altering the decision that would have been taken without it. Therefore, the principle measurement used for most of the experiments was the estimated probability of the filter to preserve the decision of the search procedure. We accumulated the set of board-move pairs from 25 games played between two copies of gnuchess level 4 (a total of above 2400 pairs). To avoid bias, these games are different than the games used for learning. The probability of a filter to success is measured by counting the number of times that the move chosen by gnuchess is among the moves selected by the filter.

The independent variables used for the experiments are:

1. *The filtering method:* f_k and f_γ. For reference we also tested a random filter, and $\alpha\beta$ with depth limit of 1 to 4 (using material for evaluation).
2. k*:* Values of 1 to 10 were tried.
3. γ*:* Values of 0.1 to 1.0 were tried.
4. *Learning stage:* The number of training games.
5. *Augmentation stage:* The number of observed games used for augmentation.

5.2 The learning process

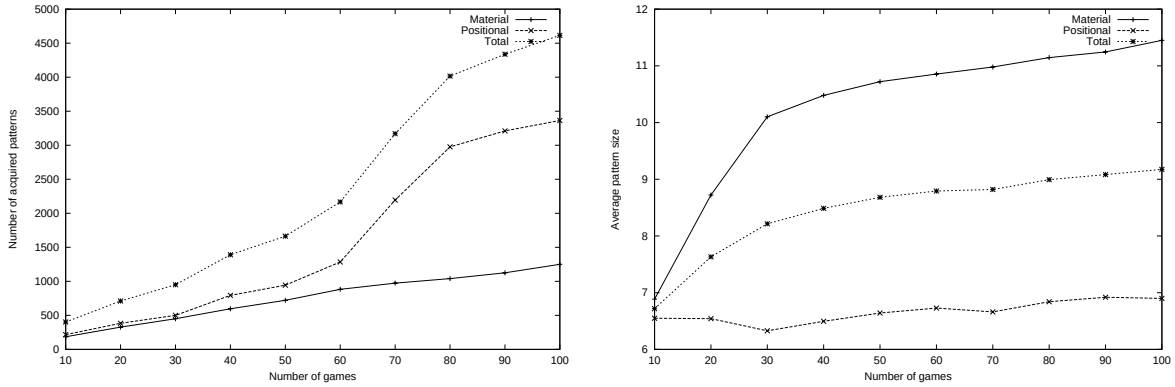


Figure 15: Progress during learning: The left figure shows the number of frames in the database as a function of the number of training games. The right figure shows the average pattern size as a function of the number of training games.

During the training games SYLPH acquired 4614 patterns. Figure 15 shows the number of patterns and the average pattern size as a function of the learning stage. We also show a separate statistics for material and positional patterns. We measure the size of a pattern by its number of edges. It is interesting to note that the average size of the material patterns increases faster than the average size of the positional patterns. The reason for this phenomenon is the different methods that material and positional patterns are acquired. The current version of SYLPH generates positional patterns by extracting the *close* neighborhood of the difference between the position before and after the move. This strategy yields limited-sized patterns. Material patterns are generated by an EBL algorithm that extracts the minimal graph which explains the unexpected outcome of the move. This strategy

leads to the generation of increasingly complex material patterns that explain more complex situations.

5.3 The effect of learning resources on the performance

We have tested the progress of SYLPH by measuring the quality of the filter with the pattern databases saved during the learning process. We have used the f_γ filter with $\gamma = \frac{1}{3}$. The left

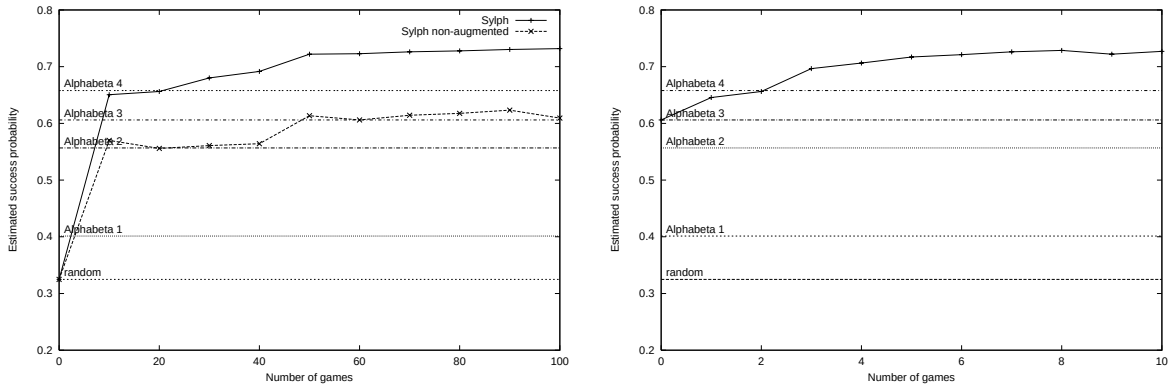


Figure 16: Progress during learning: the left graph shows the performance of SYLPH as a function of the number of training games. The right graph shows the performance of SYLPH as a function of the number of augmentation games.

graph of Figure 16 shows the performance of the filters as a function of the number of training games. The upper curve shows the performance with the augmented patterns and the lower graph with the non-augmented patterns. We also show the performance of the random filter and the performance of $\alpha\beta$ of depth 1, 2 ,3 and 4 filters. It is interesting to note that after 30 training games the SYLPH filter outperforms a filter that uses $\alpha\beta$ of depth 4. The right graph of Figure 16 shows the performance of SYLPH as a function of the number of augmentation games.

5.4 The effect of the filter selectivity on the performance

We have tested the effect of the filter selectivity on the performance of SYLPH. The performance of the f_k filter was tested with values of k between 1 and 10. The performance of the f_{gamma} filter was tested with values of γ between 0 and 1. Figure 17 shows the results obtained. Here, again, we can see the superiority of SYLPH over the depth 4 $\alpha\beta$ filter. Note that even when SYLPH rejects the move selected by the teacher, it is not necessary an error. It is quite possible that the moves selected by SYLPH are equivalent, but we could not check it, since we use the teacher as a black box and do not have access to its internals.

5.5 Testing the generality of the acquired pattern

One important question to ask after viewing the last results is how general are the acquired patterns. Since the patterns were acquired while playing against the teaching program, and were tested against the same program, it is quite possible that SYLPH merely acquired a *model* of its opponent [15, 16, 7] and is not able to use the patterns for filtering the moves

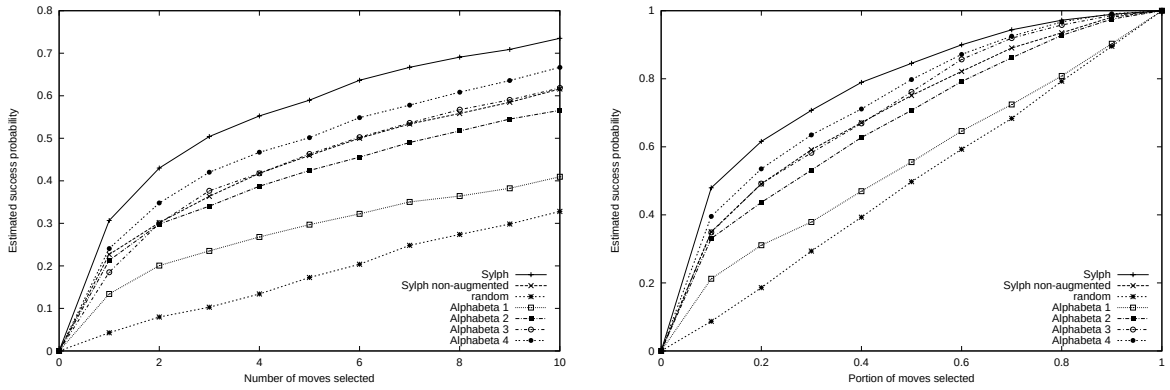


Figure 17: The performance of SYLPH as a function of the filter selectivity. $k = 5$ means that the filter is allowed to select 5 moves. $\gamma = 0.5$ means that he filter is allowed to select half of the available moves.

of another player. To verify the generality of the acquired move patterns, we tested SYLPH on 3830 moves performed by Mikhail Tal, taken from 100 games of the publicly available database of Tal's games. Figure 18 shows the results obtained. The two graphs are analogical

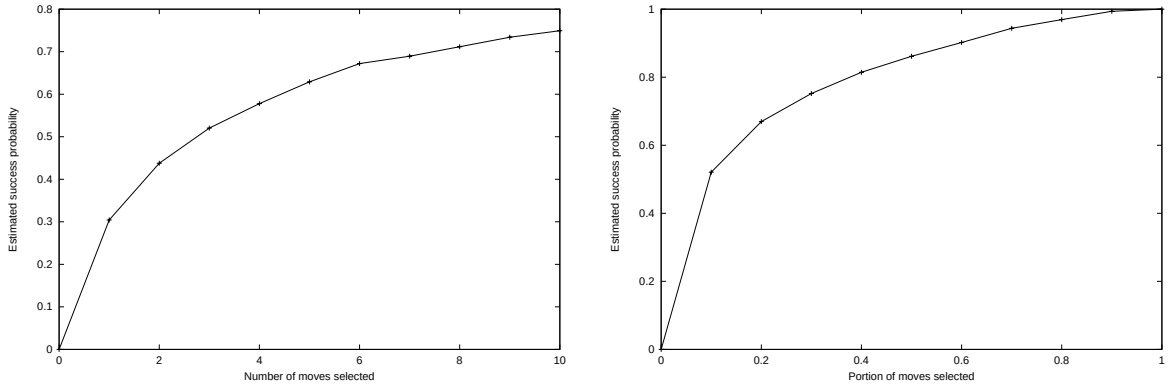


Figure 18: The performance of SYLPH as a function of the filter selectivity. The tests were done on 100 games taken from Tal's database.

to the two graphs in the previous subsection. One interesting observation is the difference in performance between SYLPH and CHUMP which was tried on the same data. For example, the result achieved by SYLPH for $k = 4$ is 0.577, while the best result achieved by CHUMP is 0.194. We believe that the main reason for this difference is the better generalization capabilities of abstract patterns compared with perceptual patterns [14].

5.6 Testing selective $\alpha\beta$ against non-selective one

To test the incorporation of the move filter into a search procedure, we performed 50 games by two versions of a depth-4 $\alpha\beta$ programs. The first is a regular non-selective $\alpha\beta$ which uses material for evaluation. The second is the same program with the pattern-based filter applied to the top level of the search tree. We used the f_k filter with $k = 5$, i.e., only 5 moves

are selected. The selective program managed to score 6 wins, 6 losses and 38 draws while searching much smaller trees. It had an average material advantage of 0.78 pawns at the end of the game.

6 Conclusions

Human players are known to have the ability of intuitively exploring only promising moves. This work studies the possibility of endowing chess programs with similar abilities by learning patterns of promising moves. Using move patterns, a chess playing program can evaluate chess *moves* without the exploration of their *outcomes*. Then, it can further explore only the outcome of promising moves.

We have defined a knowledge representation scheme for representing patterns of moves, devised a method for organizing move patterns in a hierarchical knowledge base, and developed algorithms for efficient manipulation of the knowledge base. We proceeded with developing a methodology for learning move patterns from experience. An interesting feature of the methodology presented here is its resemblance to the perception of human players. Researchers [9, 5, 14, 13] have noted that human chess players do not perceive positions as a static entity, but as a collection of potential actions.

Our methodology was implemented in the SYLPH chess learning system, and was tested experimentally. The system was able to acquire thousands of move patterns after playing only 100 training games. The quality of these patterns was tested by using the patterns for a move filter. The experiments show that, indeed, the probability of the filter throwing out the move selected by the testing program reduces when SYLPH gains experience. The performance of the pattern-based filter became better than the performance of a filter that uses a depth 4 $\alpha\beta$ with material evaluation function.

An alternative approach to the methodology presented here is to generate the set of next boards, to evaluate each of them by, for example, MORPH, and to select the moves leading to the best boards. It is possible to show that with a fixed number of moves selected at each level, the number of boards processed by this approach increases linearly with the branching factor, while the number of boards processed by SYLPH remains constant. On the other hand, the complexity of processing each board is higher for SYLPH than for MORPH since SYLPH uses a richer language and more complicated patterns. Therefore, we expect the benefit of the move-pattern approach to grow with the branching factor.

The framework proposed here is not specific to chess. To apply the framework to another game, one needs to define the set of relevant objects and set of relationships between them. The framework can also be applied to single agent search. Most of the research in reinforcement learning deals with assigning weights to actions in specific states. The move-pattern hierarchy can be viewed as a sophisticated method for representing associations between generalized classes and actions, replacing the rigid state-action table used by the traditional reinforcement learning algorithms.

Gobet and Jansen [14] classify the use of patterns in chess programs according to three dimensions: the type of patterns (abstract or perceptual), the way that the patterns are utilized (evaluation or move generation) and the way that the patterns are acquired (manually or by learning). In this classification, SYLPH *learns* and uses *abstract* patterns for *move generation*. We borrowed the table from Gobet and Jansen and inserted SYLPH into it to show its relationships to other pattern-oriented chess programs. The updated version is shown

in Table 5.

Use of patterns	Abstract		Perceptual	
	Learning	**No Learning**	**Learning**	**No Learning**
Move generation	SYLPH	Bratko/Michie [6], Zobrist/Carlson [33], Wilkins [32]	CHUMP	
Evaluation	MORPH	Zobrist/Carlson		CHUNKER I [3]

Table 5: A table by Gobet and Jansen that classifies pattern-oriented chess program with SYLPH added.

There are several directions for future research in the area of move patterns:

1. Rewrite and optimize SYLPH to allow experimentation with a larger number of patterns. Chase and Simon [8] claim that human chess experts use 80,000-100,000 patterns. We would like to test the performance of SYLPH with such a number of patterns. The current version of the system is experimental and its code is not optimized. Applying the filter takes about 300-500 milliseconds[4]. This figure makes it reasonable to be selective in the top 1-2 levels of the search tree. We believe that we can improve the performance of SYLPH significantly, which will allow us to apply the filter in deeper levels of the search.

2. Add *selective retention* [21] element to SYLPH, allowing it to delete patterns with low utility.

3. The above change will allow us to modify the positional pattern acquisition technique to learn larger patterns. The generalization element of SYLPH will intersect these larger patterns and create the more generalized (and smaller) patterns. If indeed the relevant pattern is small, the generalized pattern will remain and its descendents will be deleted by the selective retention filter. If, on the other hand, the larger pattern is relevant, it will contain information beyond its ancestor, and will be retained.

4. Test the effect of the teacher on the system performance. In particular, we intend to train SYLPH with master games.

5. Enriching the move-pattern language. The current language has some limitations that slows down the learning. We intend to add generalization hierarchy over the pieces to allow a generalized pattern such as "any white piece attacking any black piece". We also intend to introduce properties for the individual pieces, sets of pieces and squares such as the distance to the crowning row, the distance between two pieces, etc.

6. Applying the framework to other games. As we noted above, the framework is general and can be applied to other games.

7. Applying the framework to single-agent search.

We believe that the move-pattern approach has a potential of advancing the state-of-the-art in computer chess. Of course, to make it useful for real programs, we need to invest some efforts in making the manipulation of patterns fast by implementing fast pattern-manipulation code and maybe with the aid of special purpose hardware.

[4]SYLPH is written in Common Lisp and runs on a quite old Sun

References

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

[2] D. F. Beal. A generalized quiescence search algorithm. *Artificial Intelligence*, 43(1):85–98, 1990.

[3] Hans J. Berliner and M. Campbell. Using chunking to solve chess pawn endgames. *Artificial Intelligence*, 23:97–120, 1984.

[4] Hans J. Berliner and Chris McConnell. B* probability based search. *Artificial Intelligence*, 86:97–156, 1996.

[5] M. Botvinnik. *Computers in Chess.* Springer-Verlag, 1984.

[6] L. Bratko and D. Michie. Representation for pattern-knowledge in chess endgames. In M. R. B. Clarke, editor, *Advances in Computer Chess*, volume 2, pages 31–56. University Press, Edinburgh, 1989.

[7] David Carmel and Shaul Markovitch. Incorporating opponent models into adversary search. In *Proceedings of thirteenth National Conference on Artificial Intelligence (AAAI 96)*, Portland, Oregon, August 1996.

[8] W. G. Chase and H. A. Simon. The mind's eye in chess. In W. G. Chase, editor, *Visual Information Procesing*, pages 215–281. Academic Press, New York, 1973.

[9] A. D. de Groot. *Thought and Choice in Chess.* The Hague, 1965.

[10] N. S. Flann and T. G. Dietterich. A study of explanation-based methods for inductive learning. *Machine Learning*, 4:187–226, 1989.

[11] Stephen Flinter and Mark T. Keane. Using chunking for the automatic generation of cases in chess. In M. Veloso and A. Aamodt, editors, *Proceedings of the 1st International Conference on Case Based Reasoning (ICCBR-95).* Springer Verlag, 1995.

[12] Johannes Fürnkranz. Machine learning in computer chess: The next generation. *International Computer Chess Association Journal*, 19(3):147–160, September 1996.

[13] Fernand Gobet. A pattern-regocnition theory of search in expert problem solving. *Thinking and Reasoning*, 3(4):291–313, 1997.

[14] Fernand Gobet and P. Jansen. Towards a chess program based on a model of human memory. In H. J. van den Herik, I. S. Herschberg, and J. W. H. M. Uiterwijk, editors, *Advances in Computer Chess 7*, pages 35–60. University of Limburg, 1994.

[15] H. Iida, Jos W. H. M. Uiterwijk, H. J. van den Herik, and I. S. Herschberg. Potential applications of opponent-model search, part I: The domain of applicability. *ICCA Journal*, 16(4):201–208, 1993.

[16] H. Iida, Jos W. H. M. Uiterwijk, H. J. van den Herik, and I. S. Herschberg. Potential applications of opponent-model search, part II: Risks and strategies. *ICCA Journal*, 17(1):10–14, Mar 1994.

[17] Leslie Pack Kaelbling, Michael L. Littman, and Andrew W. Moore. Reinforcement learning: A survey. *Journal of Artificial Intelligence Research*, 4:237–285, 1996.

[18] R. Levinson. Experience-based creativity. Department of Computer and Information Sciences, University of California of Santa Cruz.

[19] R. Levinson. A self-organizing pattern retreival system and its applications. Technical Report UCSC-CRL-89-21, University of California of Santa Cruz, 1989.

[20] R. Levinson and R. Snyder. Adaptive pattern-oriented chess. In *Proc. of AAAI*, pages 601–605. Morgan Kaufman, 1991.

[21] S. Markovitch and P. D. Scott. Information filtering: Selection mechanisms in learning systems. *Machine Learning*, 10(2), 1993.

[22] S. Minton. Contstraint based generalization-learning game playing plans from single examples. In *Proc. of AAAI*. AAAI, 1984.

[23] A. Newell, J. Shaw, and H. Simon. Chess-playing programs and the problem of complexity. *IBM Journal of Research and Development*, 2:320–335, 1958.

[24] J. Pitrat. A program for learning to play chess. In *Pattern Recognition and Artificial Intelligence*. Academic Press, 1976.

[25] J. R. Quinlan. Learning effect classification procedures and their application to chess end games. In R. S. Michalski, J. G. Carbonell, and T. M. Mitchell, editors, *Machine Leraning*. Morgan Kaufmann, San Mateo, CA, 1983.

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

[27] C. E. Shannon. Programming a computer for playing chess. *Scientific American*, pages 48–51, 1950.

[28] A. H. Simon and M. Barenfeld. Information-processing analysis of perceptual processes in problem solving. *Psychological Review*, 76(5):473–483, 1969.

[29] D. J. Slate and L. R. Atkin. Chess 4.5 : The northwestern university chess program. In P. W. Frey, editor, *Chess Skill in Man and Machine*, pages 82–118. Springer-Verlag, 1977. 2nd edition, 1983.

[30] Richard Sutton. Learning to predict by the methods of temporal differences. *Machine Learning*, 3(1):9–44, August 1988.

[31] P. Tadepalli. Lazy explanation-based learning: A solution to the intractable theory problem. *Proc. of the International Joint Conference on Artificial Intelligence*, 1989.

[32] D. E. Wilkins. Using patterns and plans in chess. *Artificial Intelligence*, 14(2):165–203, 1980.

[33] A. Zobrist and R. Carlson. An advice-taking chess computer. *Scientific American*, 228:92–105, 1973.