Theoretical Computer Science 644 (2016) 76–91

Contents lists available at ScienceDirect

Theoretical Computer Science

www.elsevier.com/locate/tcs

A program for Progressive chess

Vito Janko [a], Matej Guid [b,*]

[a] Jožef Stefan Institute, Ljubljana, Slovenia
[b] Faculty of Computer and Information Science, University of Ljubljana, Slovenia

ARTICLE INFO

Article history:
Received 22 January 2016
Received in revised form 26 May 2016
Accepted 20 June 2016
Available online 27 June 2016

Keywords:
Progressive chess
Chess
Heuristic search
Heuristics
Checkmate search
A* algorithm
Minimax search
Combinatorial complexity

ABSTRACT

In Progressive chess, rather than just making one move per turn, players play progressively longer series of moves. Combinatorial complexity generated by many sequential moves represents a difficult challenge for classic search algorithms. In this article, we present the design of a state-of-the-art program for Progressive chess. The program follows the generally recommended strategy for this game, which consists of three phases: looking for possibilities to checkmate the opponent, playing sequences of generally good moves when checkmate is not available, and preventing checkmates from the opponent. For efficient and effective checkmate search we considered two versions of the A* algorithm, and developed five different heuristics for guiding the search. For finding promising sequences of moves we developed another set of heuristics, and combined the A* algorithm with minimax search, in order to fight the combinatorial complexity. We constructed an opening book, and designed specialized heuristics for playing Progressive chess endgames. An application with a graphical user interface was implemented in order to enable human players to play Progressive chess against the computer, and to use the computer to analyze their games. The program performed excellently in experiments with checkmate search, and won both mini-matches against a human chess master. We also present the findings of self-play experiments between different versions of the program.

1. Introduction

Game playing has traditionally been described as an excellent test bed for designing and studying new algorithms. Some games are particularly challenging because the number of reachable, indistinguishable states far exceeds the storage and computational abilities of present-day computers, thus representing difficult problems even for state-of-the-art algorithms. In this paper, we focus on one such challenging game: a chess variant called Progressive chess. In this game, rather than just making one move per turn, players play progressively longer series of moves. White starts with one move, Black plays two consecutive moves, White then plays three moves, and so on.

Chess variants comprise a family of strategy board games that are related to, inspired by, or similar to the game of chess. They can differ from standard chess in various ways such as the board layout, the number of players, the movements of the pieces, the pieces themselves, the starting positions of the pieces, the dynamics of the game, etc. Progressive chess is one of the most popular chess variants [1]; probably hundreds of Progressive chess tournaments have been held during the past fifty years [2], and several aspects of the game have been researched and documented [3–6].

* Corresponding author.
E-mail address: matej.guid@fri.uni-lj.si (M. Guid).

http://dx.doi.org/10.1016/j.tcs.2016.06.028
0304-3975/

From a game-theoretic perspective, Progressive chess shares many properties with chess. It is a finite, sequential, perfect information, deterministic, and zero-sum two-player game. The state-space complexity of a game (defined as the number of game states that can be reached through legal play) is comparable to that of chess, which has been estimated to be around 10^{46} [7]. However, the per-turn branching factor is extremely large in Progressive chess, due to the combinatorial possibilities produced by having several steps per turn. In another chess variant, Arimaa, where "only" four steps per turn are allowed, on average there are about 17,000 legal moves per turn. Up until the previous year, human players prevailed over computers in every annual "Arimaa Challenge" competition, and the high branching factor is considered as the main reason why Arimaa is difficult for computer engines [8]. We thus expect Progressive chess to provide a challenging new domain in which to test new algorithms, ideas, and approaches.

We set our goal to develop a strong program for Progressive chess that would eventually be able to achieve (and possibly surpass) the human world champion's level. While conventional chess programs can be easily adapted to some chess variants [9], in Progressive chess this is not the case. The main reason is that the combinatorial complexity, caused by having many sequential moves represents a difficult challenge for classic search algorithms. We know of no past attempts to build Progressive chess playing programs. In the 90's, a strong Progressive chess player from Italy, Deumo Polacco, developed *Esaù*, a program for searching for checkmates in Progressive chess. According to the program's distributor, AISE (Italian Association of Chess Variants), it was written in Borland Turbo-Basic, and it sometimes required several hours to find a checkmate. To the best of our knowledge, there are no documented reports about the author's approach, nor whether there were any attempts to extend *Esaù* to a complete Progressive chess playing program.

The course of the article is as follows. We start with a description of the game of Progressive chess. In Section 3, we describe the design of our Progressive chess playing program. In Section 4, we focus on the specific challenge of searching for checkmates. We continue with a more generally problem of finding generally promising sequences of moves (Section 5). In Section 6, we describe how the program was optimized for opening and endgame play. Experimental design and results of the experiments are presented in Sections 7 and 8, respectively.

An earlier version of the program was presented at the 14th International Conference on Advances in Computer Games (ACG 2015) [10]. The program has been significantly improved since then. In this extended article we present the reasons for these improvements. In particular, a lot of attention has been payed to finding generally promising sequences of moves (when checkmate is not available), while searching for checkmates has also been substantially improved. This article also presents a wider range of experiments, and gives a more detailed introduction to the game of Progressive chess.

2. Progressive chess

2.1. The rules of the game

Rules for chess [11] apply, with the following exceptions:

- Players alternately make a sequence of moves of increasing number.
- A check can be given only on the last move of a turn.
- A player may not expose his king to check at any time during his turn.
- A king in check must get out of check with the first move of the sequence.
- A player who has no legal move or who runs out of legal moves during his turn is stalemated and the game is drawn.
- En passant capture is admissible on the first move of a turn only.

There is an additional rule, rarely invoked: the game is a draw if during ten consecutive turns there is neither a capture nor a pawn move, and neither player can show an impending mate.

There are two main variants of Progressive chess: Italian Progressive chess and Scottish Progressive chess. The former has been researched to a greater extent, and a large database of games (called "PRBASE") has been assembled. In Italian Progressive chess, a check may only be given on the last move of a *complete* series of moves. In particular, if the only way to escape a check is to give check on the first move of the series, then the game is lost by the player in check. In Scottish Progressive chess, check may be given on any move of a series, but a check also ends the series. It has been shown that the difference very rarely affects the result of the game [12].

2.2. Strategy

The general strategy for both players can be summarized as follows. Firstly, look for a way to checkmate the opponent's king. Secondly, if a checkmate cannot be found, aim to destroy the opponent's most dangerous pieces whilst maximizing the survival chances of your own. Finally, before executing an intended sequence of moves, ensure that the opponent cannot checkmate your own king on the next turn.

Searching for checkmates — for both sides — is thus a very important task in this game. The left diagram in Fig. 1 shows an example of a typical challenge in Progressive chess: to find the sequence of moves that would result in checkmating the

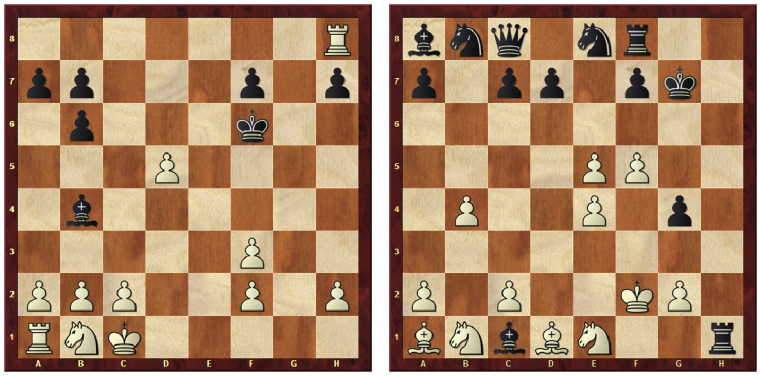

Fig. 1. Left: black to move checkmates in 8 consecutive moves. Right: white to move checkmates in 7 consecutive moves.

opponent. Black checkmates the opponent on the 8th consecutive move (note that white king should not be in check before the last move in the sequence). In the right-side diagram, the checkmate can be delivered by White in 7 moves.[1]

However, several more subtle points may be important to consider when making a decision about the most promising sequence of moves. Giving check on the last move of a turn often effectively reduces the opponent's sequence of moves by one. A king on the back rank or with only a few accessible squares at its disposal is likely to be at risk. Checks delivered by two pieces simultaneously can be especially dangerous, as the only possible response to a double check is a king move. Almost from the beginning of the game there is the ever-present risk of pawn promotions. Under-promotions are not uncommon, as it is often desirable to avoid premature checks. Interestingly, pawn promotions can often be prevented by placing the king so that they will give premature check. However, bringing the king too close to the enemy pieces may be dangerous. Putting the king in front of a friendly piece can be disastrous as well: the opponent may be able to put his own king further down the same line, and then give a check forcing an immediate discovered check in reply (see Sec. 5, Fig. 6).

The relative value of the pieces may differ significantly compared to ordinary chess, and is also far trickier to determine. For example, in the beginning of the game bishops are stronger than knights due to their long range abilities. However, in the endings knights are much better than bishops because of their ability to reach any square. Pawns may be of a high value when they threaten to reach promotion squares. Once the promotions are stopped, their value usually drops significantly.

2.3. Opening phase

The most common opening moves, as in orthodox chess, are 1.e2-e4 and 1.d2-d4.[2] Right from the beginning, Black must defend the square f7 and this largely determines the choice of acceptable initial moves. When making a decision about the sequence of moves to be executed, the typical dilemma is whether to recover material loss or to fight for an initiative, and whether to accept short-term disadvantages for potential long-term gains. Queens are particularly dangerous, so they are usually captured and taken off the board quickly. King safety should always be considered, and the king should be given air. Castling is almost never a sound option. Bringing the pieces into play may both increase their activity as well as the chance of being captured by the enemy pieces. An early advance of one or both wing pawns, bringing them closer to promotion squares and creating the way for the rooks to enter into the battle, often deserves attention.

Fig. 2 shows typical opening mistakes. The game went as follows: 1.e2-e4 2.d7-d5 Ng8-f6 3.Ng1-f3 e4xd5 Bf1-b5+ (left-side diagram). If Black responses with four moves by the bishop that win both the opponent's bishop and the queen (4.Bc8-d7 Bd7xb5 Bb5-e2 Be2xd1), White has a checkmate in 5 moves: 5.Nf3-e5 g2-g4 g4-g5 g5-g6 g6xf7# (diagram in the middle). However, in the position from the left-side diagram Black has a better sequence at his disposal: 4.c7-c6 Qd8-b6 Nf6-e4 Qb6xf2#, checkmating the white king. A much better sequence for White would be 3.d2-d4 e4-e5 Bf1-b5+, and Black already faces serious problems. Notice that giving check on the last move of a turn severely limits the opponent's options.

2.4. Endgames

In Progressive chess endgames, we assume that a player's turns have become long enough to execute sequences of arbitrary length. Not many games reach the ending, but those that do are often fascinating [6]. It is important to note that White only has odd-length sequences at his disposal, while Black's sequences are always of even length. As a non-trivial consequence, a king and two knights (K+2N) versus the lonely king K are wins for Black but not for White (see the left diagram in Fig. 3). As another example, king and queen (K+Q) to play against king is a simple win, while king and rook

[1] The solution of the left-side diagram is given in Fig. 4. The solution to the right-side diagram is 7.Ne1-f3 Nf3-d4 e5-e6 e6-e7 e7xf8R Rf8-h8 Nd4-e6#.

[2] We use standard *long algebraic notation* to describe chess moves. The starting and the ending squares of moves are separated by a hyphen, and captures are indicated using "x". The following letters are used for piece types: K for king, Q for queen, R for rook, B for bishop, and N for knight. The symbols "+" and "#" represent check and checkmate, respectively.

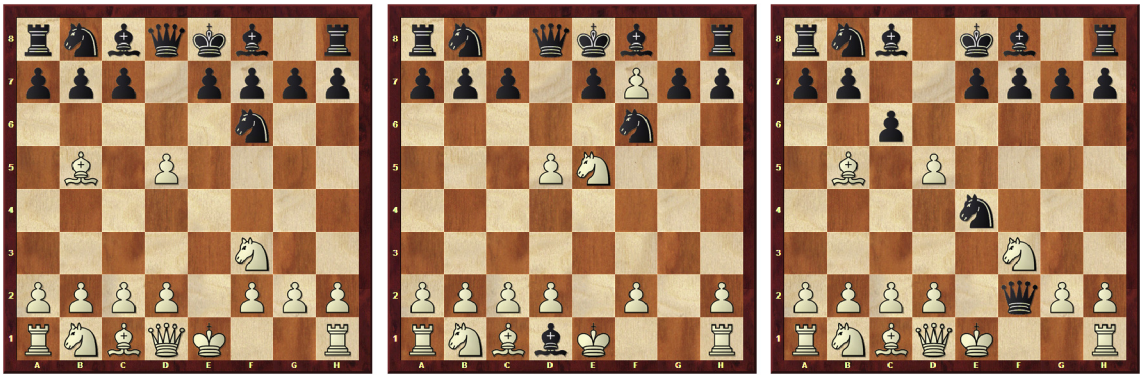

Fig. 2. In the left-side diagram it is 4th move (i.e., Black to move has to make four moves), and the king is in check. Taking the bishop and the queen with the bishop is not an option, as White has a checkmate at his disposal (diagram in the middle). However, Black can respond in a better way and delivers the checkmate to the white king (the right-side diagram).

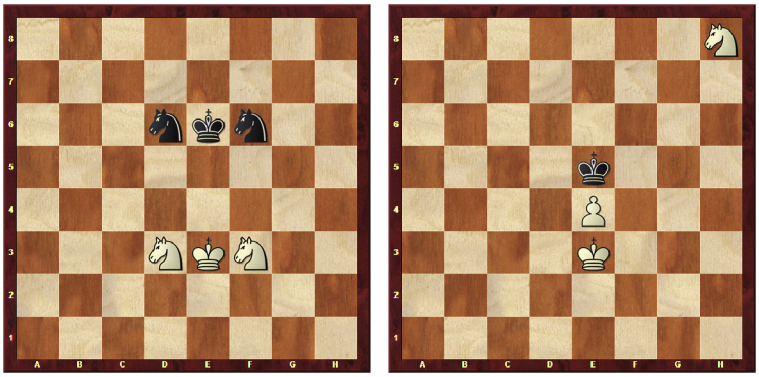

Fig. 3. Left: black to play wins, white to play only draws! Right: white to move.

(K+R) versus king is a win only if the defending king is already on the edge. The problem is that the check can only be delivered on the last move of the sequence and such a check leaves an undefended rook open to capture.

The right-side diagram in Fig. 3 shows an endgame that is easily winning for White in ordinary chess, while in Progressive chess the position is drawn. The white knight cannot escape the corner, as moving it results in a check to the black king and this can be done only at the end of the series of moves. The king cannot help the knight as this would leave the pawn undefended. The black king will capture the knight and safely return to the square in front of the white pawn. White cannot force Black to leave that square. The reason is that Black's turn always consists of an even number of moves.

3. The design of the program

The graphical user interface of our Progressive chess playing program is shown in Fig. 4. The application provides the several functionalities, including playing against the computer, searching for checkmates, saving games, and watching saved games. The user is also allowed to input an arbitrary (but legal) initial position, both for playing (or analyzing) and for discovering sequences of moves that lead to a checkmate. We implemented the Italian Progressive chess rules. The application and related material are available online [13].

3.1. Search framework

As indicated in the introduction, one of the greatest challenges for AI in this game is its combinatorial complexity. For example, on turn five (White to move has five consecutive moves at his disposal) one can play on average around 10^7 different series of moves. Games usually end between turns 5 and 8, but may lengthen considerably as the skills of the two players increases. Generating and evaluating all possible series for the side to move quickly becomes infeasible as the game progresses. Moreover, searching through all possible responses after each series is even less feasible, rendering conventional algorithms such as minimax or alpha–beta rather useless for successfully playing this game.

Generally speaking, our program is based on heuristic search. However, the search is mainly focused on sequences of moves for the side to move, and to a much lesser extent on considering possible responses by the opponent. In accordance with the aforementioned general strategy of the game, searching for the best series of moves consists of three phases:

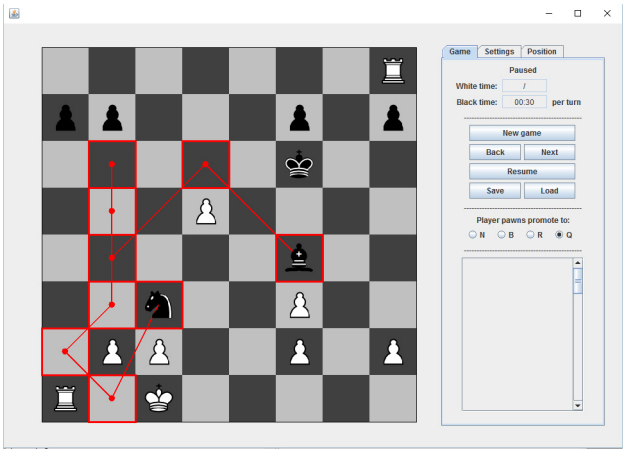

Fig. 4. The interface of our progressive chess playing program. Black's last turn moves are indicated; they represent the solution to the problem in the left-side diagram of Fig. 1: 8.Bb4-d6 b6-b5 b5-b4 b4-b3 b3xa2 a2xb1N Nb1-c3 Bd6-f4#.

Searching for checkmate In the first phase, the aim of the search is to discover whether there is a checkmate available. If one is found, the relevant series of moves is executed and the rest of the search is skipped. Checkmates occur rather often, thus finding them efficiently is crucial for successfully playing this game.

Searching for generally good moves Another search is performed, this time trying to maximally improve the position. Usually, the aim of this phase is to eliminate the opponent's most dangerous pieces, and to maximize the survival chances of own pieces. For example, giving check on the last move of a turn is considered a good tactic, as it effectively reduces the opponent's sequence of moves by one. King safety and pawn promotions are also important factors to consider. It is often possible to prevent inconvenient opponent's moves by placing the king so that they will give premature check. It is computationally prohibitive to traverse all possible sequences, so it is important that the heuristic values guide the search towards the most promising ones. The search in this phase includes a subset of possibilities both for the player to move as well as for the opponent.

Preventing checkmate The previous phase generates a number of sequences and their respective evaluation. It is infeasible to perform a search of all possible opponent replies for each sequence. However, it is advisable to verify whether we are getting mated in the following turn. The most promising sequence of moves is checked for opposing checkmate. In case it is not found, this sequence of moves is then executed. Otherwise the search proceeds with the next-best sequence, and the process then repeats until a safe move is found, or the time runs out. In the latter case, the best sequence according to the heuristic evaluation is chosen. In this phase, again a quick and reliable method for finding checkmates is required.

The main reason for splitting the search into three phases, as opposed to one search that finds good sequences with or without checkmate, is that in each phase the heuristics used are fundamentally different and also lead to exploring different parts of the search space. A similar split is made by human players when they play the game.

4. Searching for checkmates

Searching for checkmates efficiently is crucial for playing this game and is therefore given a lot of focus in this paper. In this section, we explore various attempts to achieve this goal. It can be considered as a single agent search problem, where the goal is to find a checkmate in a given position. An alternative problem setting would be to find *all* checkmates in the position, or to conclude that one does not exist, without exploring all possibilities.

The A* algorithm was used for this task. We considered various heuristics for guiding the search. In the experiments, we observed the performance of two different versions of the algorithm (see Section 4.1), and of five different heuristics (see Section 4.2). Experimental design is described in Section 7.

4.1. The algorithm

The task of finding checkmates in Italian Progressive chess has a particular property — all solutions of a particular problem (position) lie at a fixed depth. Check and checkmate can only be delivered on the last move of the player's turn, so any existing checkmate must be at the depth equal to the turn number. A* uses the distance of the node as an additional term added to the heuristic evaluation, guiding the search towards shorter paths. In positions with a high turn number (where a longer sequence of moves is required) this may not be preferred, as traversing longer variations first is likely to be more promising (as they are the only ones with a solution). One possibility to resolve this problem is to remove the distance

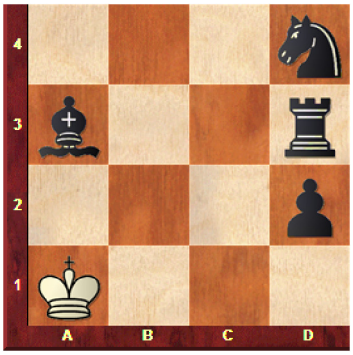

Fig. 5. The values of heuristics listed in Table 1 for this position are as follows. Manhattan: 13, Covering: 1, Ghost: 7, Squares: 8.

Table 1
Heuristics for guiding the search for checkmates.

Name	Description
Baseline	Depth-first search without using any heuristic values.
Manhattan	The sum of Manhattan distances between pieces and the opponent's king.
Covering	The number of mating squares already covered. Multiple coverings count multiple times.
Ghost	The minimum number of legal moves pieces require to reach any mating square, if they were moving like "ghosts" (ignoring the obstacles).
Squares	The sum of the number of moves that are required for each individual piece to reach every single mating square.

Table 2
Additional heuristics that can be combined with the existing ones.

Name	Description
Promotion	How far are pawns to the square of promotion, rewards extra queens.
Pin	How far is the king to the square where self-pin could be exploited.

term completely, degrading the algorithm into best-first search. An alternative is to weight the distance term according to the known length of the solution. Weight $a/length$ was used for this purpose, where the constant a was set arbitrarily for each individual heuristic, and *length* is the length of the solution. In all versions we acknowledged the symmetry of different move orders and treated them accordingly. In the experiments, we used both versions of the algorithm: *best-first search* and *weighted A**.

4.2. The heuristics

For the purpose of guiding the search towards checkmate positions, we tried an array of different heuristics with different complexities, aiming to find the best trade-off between the speed of evaluation and the reliability of the guidance. This corresponds to the well-known search-knowledge tradeoff in game-playing programs [14]. All the heuristics reward maximal value to the checkmate positions.

It is particularly important to observe that in such positions, all the squares in the immediate proximity of the opponent's king must be covered, including the king's square itself (in the future they will be referenced as the *mating squares*). This observation served as the basis for the design of the heuristics. They are listed in Table 1.

Fig. 5 gives the values of each heuristic from Table 1 for the pieces in the diagram. In *Manhattan*, pawns are not taken into account, resulting in the value of 13 $(2 + 5 + 6)$. The value of *Covering* is 1, as only the b2 square is covered. The *Covering* heuristic is the only one that needs to be maximized, which can be easily remedied by subtracting its value from some arbitrary large number. The *Ghost* heuristic obtains the value of 7 $(2 + 1 + 2 + 2)$: the rook needs two moves to reach a square immediately adjacent to the king, the bishop needs one, the knight needs two moves (to reach a1), the pawn needs one move to promote and one move with the (new) queen. The value of the *Squares* heuristic is 8 $(2 + 3 + 2 + 1)$, as there are four squares that need to be reached by the black pieces: the square a1 can be reached in two moves with the knight, a2 can be reached in three moves with the knight or rook, b1 can be reached in two moves with the rook, b2 can be reached in one move with the bishop.

Aside from covering squares around the opponent's king, there are two more useful heuristics that can be combined with the existing ones; we named them *Promotion* and *Pin* (see Table 2).

A majority of checkmates that occur later in the game include promoting one of the pawns, getting an extra attacker for delivering checkmate. Rewarding promotions of the pawns is therefore beneficial.

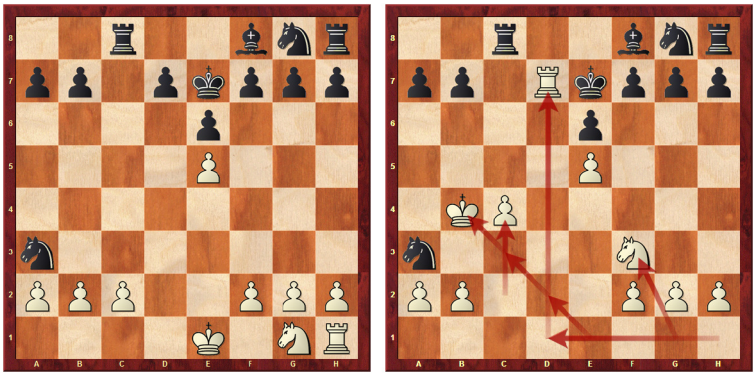

Fig. 6. Left: white to move checkmates in 7 moves. Right: the final position.

Another useful heuristic takes advantage of a "self-pin." Fig. 6 shows the controversial "Italian mate," which is enthusiastically championed by some but is felt by others to be undesirably artificial [12]. It occurs where the only way to escape a check is to give a check in return, making that move illegal. The position from the left diagram is from a game Boniface—Archer (played in the year 1993), where White played 7.c2-c4 Ke1-d2 Kd2-c3 Kc3-b4 Ng1-f3 Rh1-d1 Rd1xd7#. The final position (diagram on the right) is checkmate according to Italian rules. The solution shows the idea of exploiting the self-pin, moving the king to an appropriate square.

Let us briefly note two substantial improvements over our previous work [10] regarding checkmate search. Firstly, the *Covering* heuristic has been changed so that multiple coverings of the same square count multiple times. It turned out that this seemingly minor change significantly boosts the performance in some cases. Secondly, remaking the implementation using bitboards [15] contributed to a significant speed-up of the search.

5. Heuristic search for progressive chess

Our program first searches for a possible checkmate, as discussed in the previous section, and if one is discovered, the program executes the series of moves found. However, if no checkmate can be found, one must deal with the more general problem of finding a good series of moves. In this section, we discuss heuristics used for this task and also the choice of an appropriate algorithm.

5.1. Position heuristics

In this subsection, we explore the heuristics for evaluating positions. Given the huge amounts of available series, deep search is not feasible and we must therefore rely on relatively complex heuristics. Discovering good heuristics is challenging, as the game is relatively unexplored. Furthermore, a fair bit of what is known to be good in chess does not apply to Progressive chess. Defending pieces is an obvious example: while it may be a good idea in orthodox chess, it is often completely useless in Progressive chess (multiple sequential moves allow you to take a piece and then retreat away from defender's reach).

We hereby briefly describe the most important heuristics that our program uses for finding sensible sequences of moves. Note that these heuristics are used together (by summing their values), instead of using just one as when searching for checkmates (see Section 4).

Material count The Shannon value of pieces (Queen = 9, Rook = 5, etc.) hardly applies in Progressive chess. As already noted, bishops are better than knights in the early stages. In the ending, however, knights are much better than bishops because of their ability to reach any square. Pawns are much more dangerous than in the original game, since their promotions often cannot be prevented. Finally, queens are extremely dangerous, because of their huge potential for delivering checkmates. Additional experiments are still required to determine a suitable relative value of the pieces [16]. In this work, the Shannon values are used.

Pawn promotions Pawns can promote in five moves from their starting position. Stopping them becomes essential as the game progresses. It can be done by blockading them with pieces, placing pawns in such formation that opposing pawns cannot legally bypass them, or using the king to prevent promotions due to a premature check. The heuristic checks for both players the number of pawns that can promote next turn, and distance of the closest pawn to promotion. If the opponent does not have promotable pawns, an additional value is added to the position.

King safety Kings tend to be safe in the open air, preferably not at the edge of the board. Given the nature of the game it is usually trivial to checkmate a king that is enclosed with its own pieces, so the usual pawn defenses or castling are discouraged. Practice showed that a king is safest on the second rank; away from opponent pieces, but still safe from back rank mates. King safety can also be increased by placing the bishop in front of the king, since so placed bishop can be moved to block any lateral check.

Table 3
Table of heuristics and their weights.

Heuristic	Weights
Queen	9
Rook	5
Knight	3
Bishop	3
Pawn	1
n pawns that can promote	$n * 0.4$
No pawns can promote	-6
n squares to nearest promotion	$-n * 0.4$
King on second rank	0.8
n empty squares around the king	$n * 0.25$
Bishop in front of the king	1.5
Pieces on the last 2 ranks	2
n pieces, reachable to opponents bishop	$-n * 0.2$
Check	1
Checkmate	∞

Development Development is a risky proposition, since pieces in the center are more easily captured, and they can often be brought into action from their initial positions rather quickly. Nevertheless, pieces with higher mobility and pieces deeply in the opponent's territory (last 2 ranks) are positively rewarded.

Bishop avoidance If an opponent has only one bishop, it is desirable that your pieces stand on squares of opposite color, drastically diminishing its value.

Check Giving check to the opponents king in the last move of the series, forces him to spend his first move preventing it. Having one effective move less in his turn, might limit his options.

Opening book The opening book "is upgrading" based on results of previous games. The statistics are then used as a part of heuristic evaluation.

For completeness, we list in Table 3 our current weights used with the heuristics in the program.

Similar to regular chess, in an endgame the nature of the game changes and end game specific heuristics are used. They will be discussed in more detail in Section 6. There we also discuss opening moves used.

5.2. Algorithm

In the checkmate search, only series made in one turn are considered. In this search, however, one must also consider opponent replies if one is to achieve high level of play. The presented heuristic implicitly acknowledges options by the opponent, for example counting the number of his pawns with the ability to promote. Even so, a deeper search is probably required. In classical chess and many similar games, the minimax algorithm would be used to search the position few turns deep and return the best continuation. However, in this domain its use is infeasible, since we can hardly ever compute all possible series in a single turn, much less all possible replies to each of them (and similarly at greater depths).

Another popular alternative are Monte-Carlo algorithms [17], but in our practice they performed extremely poorly on this domain. It may be the fact that the game is extremely volatile, and often only one series wins, and most others lose, giving the random runs the algorithm a hard time to find the winning line. Additional experiments should be performed, however, before the algorithm is discarded completely.

Our approach was combining the A* search algorithm with minimax backward propagation. A* search is used to find promising series according to the listed heuristics. Then on each resulting position, the A* algorithm is run again for the next player. This can be repeated to any desirable depth, then the evaluation of the leaf nodes are back propagated according to the Min–Max principle. This procedure can be viewed as the minimax algorithm that instead of expanding all the possible nodes in one turn, expands only the most promising ones, and then greedily continues to explore only one of them.

The pseudocode is listed in Algorithm 1.

Algorithm 1 Minimax + A*(node, depth).

```
1: if depth = 0 then
2:     return heuristic value of node
3: end if
4: bestValue ← −∞
5: bestChilds ← best series of moves found in some time limit by the A* algorithm
6: for child : bestChilds do
7:     value ← -negamax(child,depth-1)
8:     bestValue ← max(bestValue, value)
9: end for
10: return bestValue
```

Table 4
Opening book for the first two turns.

1.e2-e4	2.e7-e5 f7-f6
1.e2-e4	2.d7-d5 Nb8-c6
1.e2-e4	2.d7-d5 d5xe4
1.e2-e4	2.e7-e5 Ng8-h6
1.d2-d4	2.d7-d5 c7-c6
1.d2-d4	2.d7-d5 h7-h5
1.d2-d4	2.d7-d5 Nb8-c6
1.d2-d4	2.c7-c5 c5xd4

This greedy approach could cause the algorithm to miss some relevant lines of play, but in practice it worked better than all tried alternatives. Different search depths were tested (see the results in Section 8).

6. Optimizing opening and endgame play

When playing regular chess openings or endings, one can use the knowledge specialized for that particular part of the game. The same approach was used in our Progressive chess program. In this section we present deviations from the normal play and reasons for them.

6.1. Openings

Most of chess-playing programs use a so called opening book [18]. That is a tree of the first few possible moves that have through accumulated experience of many games proven to be good. Instead of calculating best first moves every time, one of the saved moves is used. We used the same concept and saved good series of moves that one can play in the first two turns of the game (see Table 4). In every game, the computer player (semi)randomly chooses and plays one of them. The random choice (that nevertheless takes into account the statistics recorded in the tree) increases the variance of the games. Such an opening book can be easily extended and branched to any number of turns included into the tree.

Opening series were taken from an online tutorial [19], and are listed below. The openings and some alternatives were tested in the experiments presented in Sections 7 and 8.

From turn 2 on, the program hashes all reached positions and updates the game result in the corresponding nodes. So acquired statistics are then used to modify the heuristic evaluation of that position the next time it is reached. This approach creates even more varied games, as poorly performing openings tend to get avoided.

6.2. Endings

Similar to regular chess, the nature of the game changes in the endgame. With only a few pieces on the board, a danger of sudden checkmates diminishes, the power of the king increases, and stopping pawn promotions becomes one's first priority. Additionally, when the opponent is reduced to the king only, one must (just like in regular chess) push the opponent king to the edge of the board and deliver a checkmate. Doing so with some combinations of pieces (for example: two knights) requires a special pattern of moves that is not trivial to see, even for experienced players.

Using additional heuristics listed below with the regular ones in the second phase of the search, solves all the problems listed above.

Defence Calculates which squares are reachable by the opponent king, and which of them are undefended. Each reachable, but undefended position negatively impacts the position score. This heuristic implicitly rewards cutting the opponent king from the relevant side of the board.

Closeness to the edge This heuristic checks how close to the edge of the board will the opponent king be in the worst case, despite striving to get away from it. The closer to the edge the king is trapped, the higher is the heuristic value. It turns out that using this heuristic in conjunction with the checkmate search in the previous phase, the computer player is able to effectively find a checkmate with various combinations of pieces. No additional stored patterns are required for this task. Interestingly, the mates with two knights and a knight and a bishop can only be forced by the black player; as discussed earlier, this is the consequence of the fact that White always has an odd number of moves in a series, while Black's move number is always even. Thus only Black can force White to move to another square when only two squares are available to the opponent king, and this makes all the difference.

Closeness to the center Conversely, a small reward is given to the king for standing nearer to the center, which makes the king safer.

Stalemate The position is checked for a stalemate and ranked accordingly.

7. Experimental design

The goal of the experiments was to verify empirically how promising our approaches are for playing Progressive chess. The first point of interest was its ability to find checkmates in an efficient manner. In particular: (1) which of the two search algorithms performs better (*best-first search* or *weighted A**), (2) which is the most promising heuristic to guide the search (*Manhattan*, *Ghost*, *Covering*, or *Squares*), and (3) what is the contribution of the two additional two heuristics (*Promotion* and *Pin*; see Table 2).

Secondly, we wanted to evaluate heuristics proposed in Section 5 and experiment with different settings of our algorithm proposed in that section. Moreover, beside finding good configurations of individual program components, we also wanted to test the program's overall ability to play the game.

Another research question is who the advantage has in Progressive chess: White or Black (note that this is not so clear as in orthodox chess). *The Classified Encyclopedia of Chess Variants* claims that masters have disagreed on this question, but practice would indicate that White has a definite edge [2].

7.1. Experiments

Several sets of experiments were conducted. Firstly, we observed how quickly different versions of the program did find checkmates on a chosen data set of positions with different solution lengths (see Section 7.2). Both average times and success rates within various time constraints were measured. The search was limited to 60 seconds per position (for each version of the program).

Secondly, self-play experiments were performed with the programs with the same algorithm (weighted A* with the two additional heuristics) and various heuristics. The programs played each other in a round-robin fashion. The winning rates were observed for each version of the program. In the second phase of the game (see Section 3.1), a small random factor influenced the search so the games could be as diverse as possible. Four different, increasingly longer time settings were used in order to verify whether different time constrains affect the performance.

We wanted to test the ability of computers to find checkmates in comparison to the human ability in the same task. To do so, we looked at the tournament games from the PRBASE (see Section 7.3), and assumed they were played by good Progressive chess players. We observed how many times human players missed a checkmate that the computer found, and vice versa.

In the fourth experiment we tested different depths of search (Section 5), using the described combination of A* and the minimax algorithm. Even with the greedy nature of such combination, it is unreasonable to expect very deep search, since the A* search at every node is rather costly. Similarly to the second experiment, self-play experiments were performed: different versions of the program were playing at different search depths. We tested depths 0, 1 and 2, where 0 means that only the current turn is searched, without taking into account the opponent's replies (except the usual checkmate prevention). The experiment was repeated at different time settings to observe whether greater search depths benefit from longer available times.

Next, we tested the contribution of the two additional heuristics in the second phase: *Pawn promotion* and *King defense*. Again, self-play experiments were executed with four different versions of the program: one had the full set of heuristics, one was missing *Pawn promotion*, one was missing *King defense*, and the last one was missing both. The positive contribution of the *Material count* heuristic was obvious and was not explicitly tested.

In order to test the efficiency of our opening series (see Section 6), we conducted self-play experiments where each version of the program always picked its own opening. In addition to the opening series in the opening tree of the program, some additional series of opening moves were considered.

Lastly, the computer played a series of games against a chess FIDE master that also had substantial experience with Progressive chess. The goal was to determine the quality of the computer player as a whole. In the future we hope to increase the sample size of this experiment by entering a Progressive chess competition with our program.

7.2. The checkmates data set

We collected 900 checkmates from real simulated games between the programs. In each turn in the range from 4 to 12, there were 100 different checkmates included. The shortest checkmates in Progressive chess can be given on turn 3, however, they are few and rather trivial. Longer games are rare, and even then there are usually very few pieces left on the board, making the checkmate either trivial or impossible. The above distribution allowed us to observe how the length of the solution affects the search performance.

7.3. PRBASE data set

PRBASE [20] is a Progressive chess database, that is advertised as containing a few thousand games. The version of PRBASE that we managed to find, however, contained 654 games in total. Out of those games, 332 ended or should have ended in a checkmate (as opposed to a draw or more often, resignation). All the games were already annotated by the Progressive chess expert Deumo Polacco, and he also marked checkmates missed by the players. Checkmates in these games were slightly more difficult to find by the program, which implies a relatively high quality of the games in PRBASE.

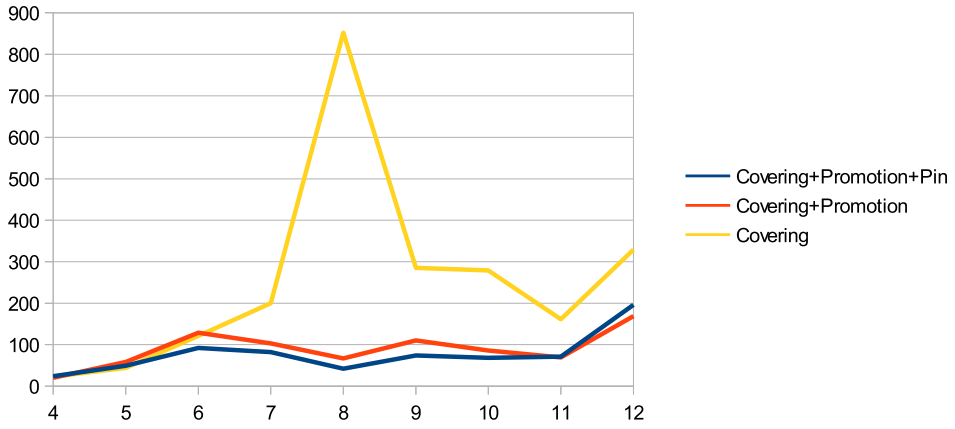

Fig. 7. Average times (in milliseconds) with the improved A* search algorithm. The horizontal axis represents the length of the solution.

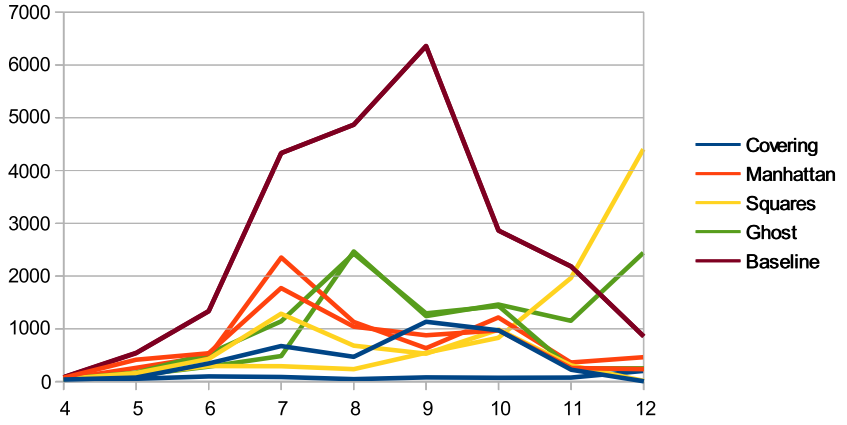

Fig. 8. Average times (in milliseconds) with the best-first search algorithm. The horizontal axis represents the length of the solution.

8. Results

In this section, we discuss the results of the experiments.

8.1. Average times for finding checkmates

First we show that combining the base heuristics with the two additional ones (*Promotion* and *Pin*) greatly improves the performance. In Fig. 7, we show average times for finding checkmates using the *Covering* heuristic and the improved A* search. The same pattern can be observed in all other combinations. We see that the *Promotion* heuristic dramatically shortens the average search time for checkmates in turns 7–9. That can be easily explained, since on those turns most of the promotions occur. The *Pin* heuristic gives slight improvements on basically all turns. These improvements are slight, as *Italian mates* do not occur often. The two additional heuristics were from now on included in all the experiments that followed.

Fig. 8 gives the average times for finding checkmates with the best-first search algorithm. It roughly outlines the difficulty of the task: finding checkmates is easier when the solution is short (turns 4–6), more difficult when the solutions are of medium length (turns 7–10), and easier again in the later stage (turns 11–12), as the material on the board diminishes. The *Covering* heuristic performed best at most solution lengths.

The average times with the improved A* algorithm are given in Fig. 9. For some heuristics, the average time increased greatly at the later stages. The main reason is that the heuristics occasionally fail to find the solutions in later stages (note that each failed attempt is “penalized” with 60,000 milliseconds, i.e., the time limit for each problem). The exception is the *Covering* heuristic, which outperformed others at all stages and turned out to be by far the most promising of all tried combinations.

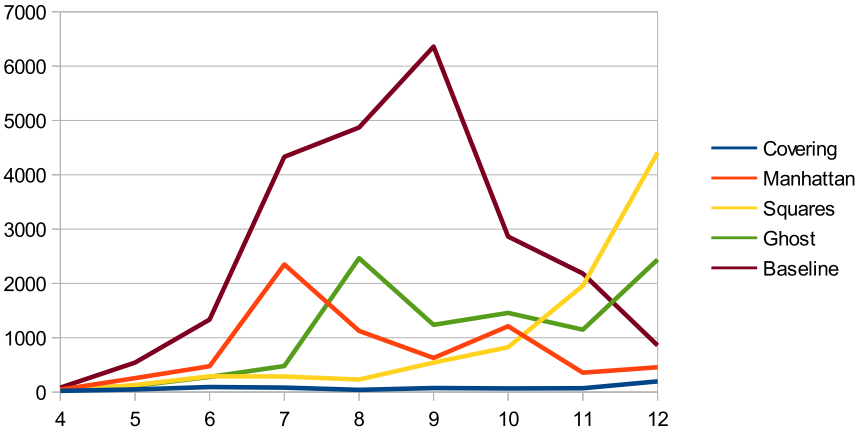

Fig. 9. Average times (in milliseconds) with the improved A* algorithm. The horizontal axis represents the length of the solution.

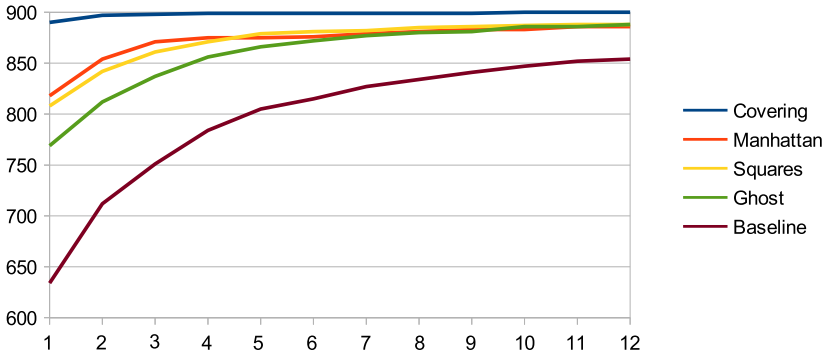

Fig. 10. Percentage of checkmates found in less than $1, 2, \ldots, 12$ seconds.

8.2. Finding checkmates

Fig. 10 demonstrates how many checkmates were found at any given point of time (in seconds). The *Covering* heuristic performed clearly best at every cutoff point. It found 98% of the checkmates in less than a second, and all of them within the time limit of 60 seconds. The improved A* algorithm (using the two additional heuristics) was used in the rest of the experiments.

Using the *Covering* heuristics and the improved A* algorithm we performed a test on the PRBASE database. On each position of each game, the computer was given 10 seconds to find a checkmate. Notice that the 10 seconds limit is probably much less time than the players and the annotator had spent on any given turn (tournaments are, to the best of our knowledge, usually played with the 15 minute per game limit). The program found 69 checkmates that were missed by the players, while it only missed 9 checkmates found by the players. This result indicates that even with such constrained time settings, the computer player at finding checkmates outperforms a typical player in the PRBASE database. Some of the checkmates found were also missed by the annotator.

8.3. Contribution to the overall game skill

The results of the self-play experiments using the programs with different checkmate heuristics are given in Fig. 11, showing the percentage of wins for each heuristic. The number of games for each time setting is listed in Table 5. The *Covering* heuristic clearly outperformed all the other heuristics, showing that the ability to find checkmates is strongly linked to the overall ability to play the game well.

8.4. Search depth

The results of the self-play experiments using the programs searching to different search depths are given in Fig. 12. The percentage of wins are shown for each heuristic. The number of games is given in Table 6. Regardless of the time given, search with depth 1 performed best. Such a search devotes all its time calculating sequences of moves in the current turn and some of the opponent's possible replies. This strategy turned out to be more effective than investigating deeper at the cost of less search in the earlier turns.

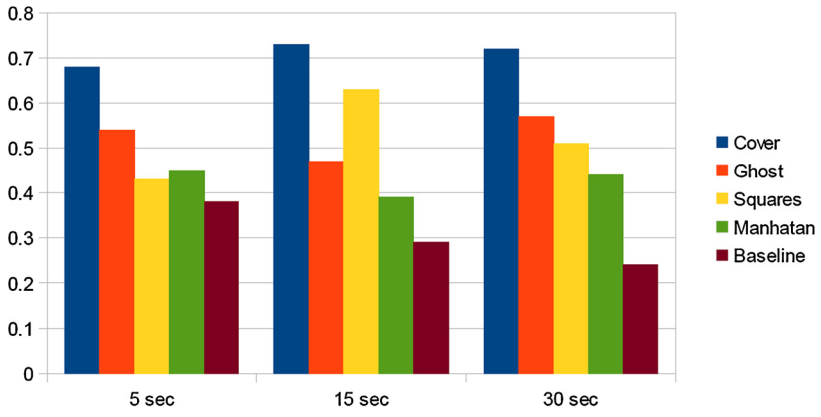

Fig. 11. Contribution to the overall game skill: the success rate for each program in different time settings.

Table 5
The number of games for each time setting.

5 sec	15 sec	30 sec
600	400	200

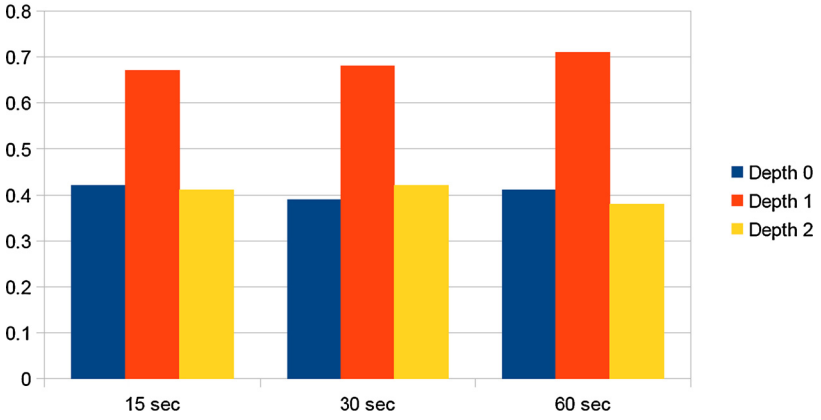

Fig. 12. Search depth: the success rate for each program in different time settings.

Table 6
The number of games for each time setting.

15 sec	30 sec	60 sec
300	250	200

8.5. Evaluating additional heuristic knowledge

Fig. 13 shows the results of the self-play experiments between the programs that had some of the heuristics removed. The experiment was performed on both search depths 0 and 1 (300 games per run). It is clearly visible that additional knowledge contributed to higher winning rates. The difference got smaller with a greater search depth, since in that case the algorithm is less heuristic dependent.

8.6. Evaluating openings

Table 7 shows the results of the self-play experiments between the programs that used different opening series. The last four options in the table do not use a recommended first move for white, and White's winning rate clearly decreases. Other opening choices appear relatively well balanced and give similar winning possibilities for both players. Nevertheless, the obtained ratios should be interpreted with care, in particular since Progressive chess openings are extremely volatile: there are well-known cases when a single improvement brought a sudden, radical change in the evaluation of entire variations.

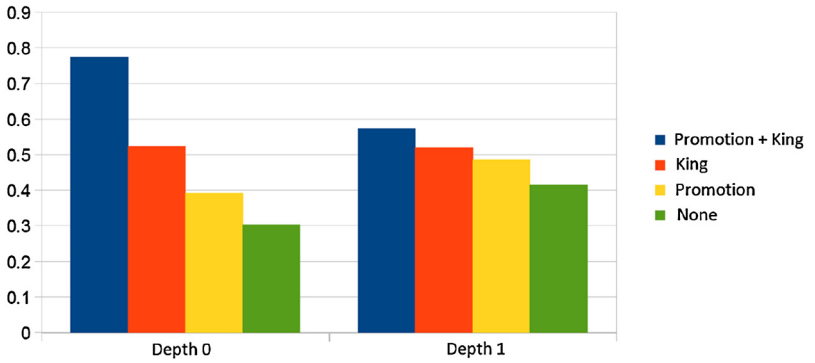

Fig. 13. Evaluating additional heuristic knowledge: the success rate for each program in different time settings.

Table 7
White win percentage at listed openings.

1.e2-e4	2.d7-d5 Nb8-c6	43%
1.e2-e4	2.d7-d5 d5xe4	50%
1.d2-d4	2.d7-d5 c7-c6	50%
1.d2-d4	2.d7-d5 h7-h5	44%
1.d2-d4	2.d7-d5 Nb8-c6	39%
1.d2-d4	2.c7-c5 c5xd4	64%
1.d2-d3	2.d7-d5 Ng8-f6	27%
1.e2-e3	2.e7-e5 Ng8-h6	20%
1.f2-f3	2.d7-d5 c7-c5	25%
1.h2-h4	2.e7-e5 e5-e4	29%

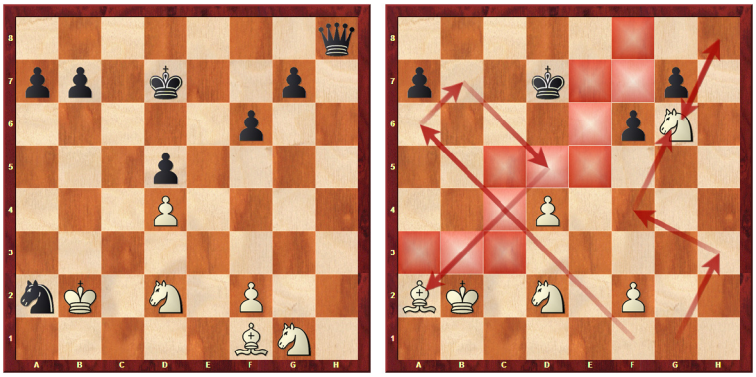

Fig. 14. Left: white to move wins, 9 moves left. Right: the final position, the program's moves are indicated. Black is lost, as his king cannot cross the barrier created by white pieces (the squares marked with the red color), and all pawn promotions are prevented.

8.7. Games against the human opponent

The final version of the program was matched against a chess FIDE master with substantial Progressive chess experience. Two matches were played: with the time limit of 10 minutes per game the computer won 6–0, while with the time limit of 15 minutes per game the result was 3.5–2.5 (the computer won). With such a limited number of games (moreover, against a single opponent) the experiment may not be of much scientific value, however, it was interesting to see that our computer program can win several games against a rather strong human opponent.

Fig. 14 demonstrates an instructive sequence of moves executed by the program in one of the games in the matches against the human FIDE master. In the diagram on the left, White has to make 9 moves. The most important tasks for him are (1) to capture the black queen, (2) to prevent black pawns from promoting, and (3) to capture the black knight. The diagram on the right shows the triumph of the program's strategy: not only that all the aforementioned goals were achieved, White also succeeded in preventing the black king to enter his camp by creating a barrier that the king is not allowed to cross. Black can do nothing to prevent losing all the material and/or getting mated. The human player thus resigned the game.

9. Conclusions

The aim of our research is to build a strong computer program for playing and learning Progressive chess. This chess variant was particularly popular among Italian players in the last two decades of the previous century [2]. By developing a strong computer program, we hope to revive the interest in this game both among human players, who may obtain a strong playing partner and an analysis tool, as well as among computer scientists. In particular, the extremely large branching factor due to the combinatorial explosion of possibilities produced by having several moves per turn makes Progressive chess both an interesting game and a very challenging environment for testing new algorithms and ideas.

We designed and implemented a Progressive chess program that follows the generally recommended strategy for this game, which consists of three phases: (1) looking for possibilities to checkmate the opponent, (2) playing sequences of generally good moves when checkmate is not available, and (3) preventing checkmates from the opponent. In order to adopt this recommended strategy, we developed two different sets of heuristics and search algorithms: one for checkmate search, and one for finding promising sequences of moves. Moreover, a set of specialized heuristics was crafted for the endgame phase, and an opening book was created in order to improve the opening play. The program and related material are available online [13].

For checkmate search, which is a very important task in Progressive chess, we slightly adapted the well-known A* algorithm, and introduced five heuristics for guiding the search. In particular, the *Covering* heuristic proved to be both efficient and effective when searching for checkmates. This heuristic gives an estimate how well the squares around the opponent's king are covered. In the experiments with 900 checkmate-in-N-moves problems, the program with the *Covering* heuristic found solutions in 98% of the cases within the first second, and all within one minute of search on regular hardware.

We adopted the Italian Progressive chess rules, where a check may only be given on the last move of a *complete* series of moves. The length of the solution (whether it is a checkmate or merely a promising sequence of moves) is therefore known in advance. As it was demonstrated experimentally, it was therefore beneficial to slightly adapt the A* algorithm. Concretely, the distance term of the A* algorithm was weighted with respect to the solution length.

The combination of the weighted A* algorithm and minimax search yielded best results for finding generally promising moves. Several heuristics were crafted for this purpose, including the following: material count, pawn promotions, king safety, and development. Interestingly, the minimax search performed best when searching to level 1, that is taking into account the player's current turn (a sequence of moves) as well as the opponent's replies (also a sequence of moves). While searching deeper in game-playing programs relying on the minimax principle should in general yield better results [21], this finding shows that the combinatorial explosion of possibilities produced by having several moves per turn makes it very difficult to cover all relevant parts of the search space.

In two mini matches against a rather strong human opponent — a chess FIDE master with substantial Progressive chess experience — the program not only won, but also demonstrated instructive play on several occasions. In this sense, by enabling a computer analysis for an arbitrary Progressive chess position, the program may represent a valuable teaching tool for an interested player.

Our program still requires further work to achieve (and possibly surpass) the level of the best human players. Firstly, the program would benefit from further improving the speed of finding checkmates. Beside trying to improve the *Covering* heuristic (or any other one), alternative approaches could be tried, such as nested Monte-Carlo tree search [22]. Particularly in the second phase of the game (which is not directly associated with searching for checkmates) we see a lot of room for improvements. One possibility to alleviate the problem of the combinatorial explosion of possibilities could be an inclusion of Monte-Carlo tree search techniques [23,24]. The question who has the advantage in Progressive chess is still open and could be the subject of further investigation.

References

[1] D. Pritchard, Popular Chess Variants, BT Batsford Limited, 2000.
[2] D.B. Pritchard, J.D. Beasley, The Classified Encyclopedia of Chess Variants, J. Beasley, 2007.
[3] M. Leoncini, R. Magari, Manuale di Scacchi Eterodossi (in Italian; "The Manual of Heterodox Chess"), Tipografia Senese, Siena-Italy, 1980.
[4] G. Dipilato, M. Leoncini, Fondamenti di Scacchi Progressivi (in Italian; "The Fundamentals of Progressive Chess"), Macerata-Italy, 1987.
[5] A. Castelli, Scacchi Progressivi. Matti Eccellenti (in Italian; "Progressive Chess. Excellent Checkmates"), Macerata-Italy, 1996.
[6] A. Castelli, Scacchi progressivi. Finali di partita (in Italian; "Progressive Chess. Endgames"), Macerata-Italy, 1997.
[7] S. Chinchalkar, An upper bound for the number of reachable positions, ICCA J. 19 (3) (1996) 181–183.
[8] D. Wu, Designing a winning Arimaa program, ICGA J. 38 (1) (2015) 19–40.
[9] E. van Reem, Shredder wins second Chess960 Computer Chess World Championship, ICGA J. 29 (4) (2006) 214–218.
[10] V. Janko, M. Guid, Development of a program for playing progressive chess, in: Advances in Computer Games (ACG 2015), in: Lecture Notes in Computer Science, vol. 9525, Springer, 2015, pp. 122–134.
[11] FIDE, The FIDE Handbook: Laws of Chess, http://www.fide.com/fide/handbook.html, 2016.
[12] J. Beasley, Progressive chess: How often does the "Italian rule" make a difference?, http://www.jsbeasley.co.uk/vchess/italian_rule.pdf, 2011, accessed 20-January-2016.
[13] V. Janko, M. Guid, http://www.ailab.si/progressive-chess/, 2016, accessed 20-January-2016.
[14] A. Junghanns, J. Schaeffer, Search versus knowledge in game-playing programs revisited, in: 15th International Joint Conference on Artificial Intelligence, Proceedings, vol. 1, Morgan Kaufmann, 1999, pp. 692–697.
[15] C.B. Browne, Bitboard methods for games, ICGA J. 37 (2) (2014) 67–84.

[16] S. Droste, J. Fürnkranz, Learning the piece values for three chess variants, ICGA J. 31 (4) (2008) 209–233.
[17] C.B. Browne, E. Powley, D. Whitehouse, S.M. Lucas, P. Cowling, P. Rohlfshagen, S. Tavener, D. Perez, S. Samothrakis, S. Colton, A survey of Monte Carlo tree search methods, IEEE Trans. Computat. Intell. AI Games 4 (1) (2012) 1–43.
[18] C. Donninger, U. Lorenz, Innovative opening-book handling, in: Advances in Computer Games, Springer, 2006, pp. 1–10.
[19] D. Hyatt, Progressive chess: introduction, https://www.youtube.com/watch?v=YPMRbodCwYg, 2010, accessed 20-January-2016.
[20] PRBASE (Italian progressive Chess database), http://www.oocities.org/colosseum/lodge/2483/att/soft.htm, accessed 20-January-2016.
[21] M. Luštrek, M. Gams, I. Bratko, Is real-valued minimax pathological?, Artificial Intelligence 170 (6) (2006) 620–642.
[22] T. Cazenave, Nested Monte-Carlo search, in: IJCAI, vol. 9, 2009, pp. 456–461.
[23] R. Coulom, Efficient selectivity and backup operators in Monte-Carlo tree search, in: Computers and Games (CG 2006), in: Lecture Notes in Computer Science, vol. 4630, Springer, 2007, pp. 72–83.
[24] L. Kocsis, C. Szepesvári, Bandit based Monte-Carlo planning, in: Machine Learning: ECML 2006, Springer, 2006, pp. 282–293.