COSC2780: Intelligent Decision Making

Chess Puzzle Generation with Answer Set Programming

Axel Thor Ahmer [*]   Kee Chew Huang [†]

June 5, 2022

1 Introduction

This paper investigates the effectiveness of using answer set programming (ASP) to generate chess checkmate puzzles. ASP is a form of declarative logic programming that involves computing stable models as solutions (or answer sets). The ASP system used throughout this report is Clingo[1] - which includes both grounding and solving software [1].

The models we were interested in generating consisted of a set of chess pieces at various times, and a set of corresponding moves which will lead some initial chess position to a desired position where one player is in checkmate - this is considered a planning problem. Two declarative models with the same logic core were built: one to create models where only the white pieces move and try to achieve checkmate on black in the shortest time possible (static planning); with the other allowing both piece colors to move but with the same goal (dynamic planning with an adversary - considerably more difficult).

We succeeded in building correct models for both cases, however as is discussed within this paper, generating correct and fun chess puzzles is not equivalent to generating a model of the game of chess. Much future work exists in optimization and adding features. It is worth mentioning that this project appears to be the most ambitious chess related project using ASP - with other academic work focusing on much simpler planning problems within chess.

2 Literature

Chess has been researched and implemented in Prolog (a query-based declarative programming language) previously [2, 3, 4, 5]. Two open-source GitHub

[*]s3855518@student.rmit.edu.au

[†]s3808292@student.rmit.edu.au

[1]`https://potassco.org/`

repositories have shown that it is technically possible to logically declare the entire game of chess[4, 5], including special moves, e.g. en-passant[2] and castling[3]. One repository even incorporates a shallow game-tree search with a naive value based evaluation heuristic that can be used for move generation[4].

Literature investigating the use of ASP to model chess is much more scarce. Two famous chess computation puzzles have been solved using ASP: the n-queen problem, and the knight's tour problem[6] - however, both these puzzles are relatively simple - only involving a single piece type from one player, i.e. involving no adversarial planning. There has been analysis done of the n-queens problem and how chess problems have been approached previously[7].

Although ASP may not generally be suited to tasks involving building and pruning a game-tree, an overview of how logic programming can be applied to problems in different systems and applications suggests that it is at least conceptually possible[8].

Conventional methods for generating checkmate puzzles is to parse an input chess game or game database, searching for positions where it declares a 'checkmate in N' [9, 10]. From that point, it is a merely a matter of analyzing the positions further in order to ensure the puzzles are 'good puzzles'. Tord Romstad, one of the initial contributors to Stockfish [4], the current strongest open-source chess engine in the world, defines good chess puzzles have the following attributes [10]:

1. There is exactly one winning move.
2. The side to move is not in check. Positions where the side to move is in check tend to have too few legal moves to be interesting as puzzles.
3. The winning move is not a mate in 1, not a promotion to queen, and not a simple capture of a hanging piece. This is an attempt to eliminate puzzles that would be too easy.
4. The winning move is a sacrifice or was missed by at least one of the two players. This should usually ensure that the win is not entirely trivial to spot.

3 Implementation

3.1 Project Structure

This project's code can be found on GitHub[5]. The main branch contains: both old and new ASP model approaches; auxiliary ASP and planning notes; various python helper files for parsing chess notation to and from CLI, FEN, and LaTeX,

[2]`https://en.wikipedia.org/wiki/En_passant`
[3]`https://en.wikipedia.org/wiki/Castling`
[4]`https://stockfishchess.org/`
[5]`https://github.com/RMIT-COSC2780-IDM22/course-project-abp`

in addition to executing clingo programs and handling output files. The most important files and folders are listed and described below:

1. **asp** (Folder - Depricated):

   Contains ASP files for *brute-force* calculation of legal moves for static and dynamic planning. Correct but slow.

2. **asp2** (Folder):

   Contains ASP files for *declarative* inference of legal moves for static and dynamic planning. Correct and *less* slow.

   - **pieces.lp:** Non-linear piece movement definitions (king/knight).
   - **linear.lp:** Linear piece movement definitions (queen/rook/bishop).
   - **planner.lp:** Planning code relevant to both static and dynamic planning problems.
   - **static.lp:** Facts, rules, and integrity constraints specific to static planning.
   - **dynamic.lp:** Facts, rules and integrity constraints specific to dynamic planning.

3. **printChessBoard.py** (Python File):

   A python file that queries the user to specify a series of inputs:

   - Board size (n).
   - Upper bound on number of white pieces (w).
   - Upper bound on number of black pieces (b).
   - Maximum planning time-steps (k).
   - Number of solutions to output (l).
   - Model to implement, i.e. static or dynamic (model).

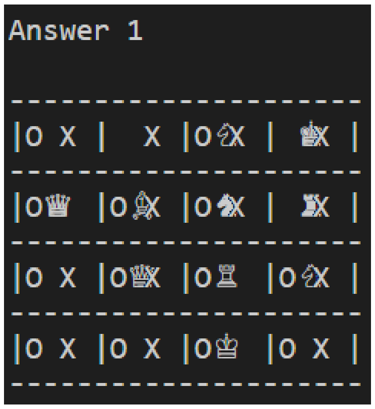
   The program then runs the follow operating system command, piping the result into a text file:

   ```
   clingo asp2\{model}.lp asp2\planner.lp asp2\linear.lp asp2\
       pieces.lp -c n={n} -c k={k} -c w={w} -c b={b} --opt-mode
       optN -n {l} > output.txt"
   ```

   This output file is then parsed and human interpretable CLI board visualizations 1 of the generated plans are printed to the console, in addition to debugging predicates, and creating an additional text file with source code capable of presenting the plans as nicely formatted chess puzzles.

4. **puzzle-sheets** (Folder):

   Contains downloadable pdf puzzle sheets generated by our planner you can share with your friends!



```
Answer 1


--------------------
|O X |  X |O♘X | ♚X |
--------------------
|O♛  |O♗X |O♞X | ♜X |
--------------------
|O X |O♛X |O♜  |O♘X |
--------------------
|O X |O X |O♚  |O X |
--------------------
```

Figure 1: CLI board visualization of an outputted checkmate plan. Pieces shown on their starting squares, with white/black attacks displayed with an O/X, respectively. Note: piece colors are inverted due to the black console.

3.2 Basic Predicates

This section describes the basic knowledge used to represent the most important chess principles. This knowledge is expressed as symbols that can be interpreted by Clingo to enable solving of posed problems. Some important predicates are listed below:

1. **chessman(Piece, Color, Row, Col, T)**

   Represents the state of a chess piece (chessman) on the chessboard at time T. A collection of chessman(_,_,_,_,T) describe the state of a chessboard at time T.

2. **guarded(Row, Col, Row0, Col0, Color, T)**

   Represents that the cell(Row, Col) is under guarded from the cell(Row0, Col0)) at time T. The Color of the attacking piece is also registered to avoid taking friendly pieces during planning in the later phases.

3. **check(Color, T)**

   Represents whether the king of the Color is in check at time T. This is induced by evaluating the intersections of guarded and the king's position at some time.

4. **linearAttack(Row, Col, Color, Direction, Row0, Col0, T)**

   A linear attack is we represent the movement/attacks of the linear chess pieces (rooks, bishops, and queens). This predicate represents the cell(Row, Col) is under a linear attack from the piece at cell(Row0, Col0) at time T. It is evaluated on every board by recursively propagating a linearAttack in every Direction that linear piece can attack - until it hits another chess piece or the edge of the board.

5. **dynamicMove(Row, Col, Row', Col', T)**

   Represents a move that complies with basic chess movement rules only i.e. it does not account for moving oneself into check being illegal. It asserts a chessman at cell(Row, Col) can move to cell(Row', Col') at time T, under basic movement rules only. With it's current implementation, it mimics very closely the guarded/5 predicate, however it will become useful once pawns are implemented (future work) - as they are they only chess pieces that move differently to how they attack/guard.

6. **legalMove(Row, Col, Row', Col', T)**

   The set of legalMoves is a subset of the set of dynamic dynamicMoves for each time T. It contains an extra constraint that says a move cannot be made that puts it's own king in check (an illegal move in chess), such a move could be considered a dynamicMove but not a legalMove.

7. **move(Row, Col, Row', Col', T)**

   A *move* is *one* of the set of legal moves that Clingo selects through a choice rule. It represents the move that happens in actual chess game we are modelling.

8. **checkmate(T)**

   A checkmate is defined as there is no legal move available at time T.

3.3 Planning Overview

```
% initial board at T=1.
{chessman(...,T=1), chessman(...,T=1)...}

% white always move first.
color_to_move(white, 1).

% alternating turns.
color_to_move(Color, T) :- color_to_move(Color2, T-1), enemy(Color,
     Color2).

% picking a move from all the legal moves.
{move(R,C,R',C',T) : legalMove(R,C, R',C',T)}=1 :-
    time(T),
    T<N : checkmate(N).

% update the chessman positions after the move
chessman(..., T+1) :- chessman(..., T), move(...,T),...

% evaluation of guarded, linearAttack, dynamicMove, legalMove
    induced from chessman(...,T+1).
```

3.4 Legal Moves

3.4.1 Brute Force Approach

Our first approach to implement Legal Moves was to exhaustively define every situation that a chess player can be in: they are in check by two pieces, check by one piece, or they are not in check (it is impossible to be in check by three or more pieces by definition). Then we introduced a predicate to define if a piece was *pinned* (a pinned chess piece cannot move in a direction that expose the king to attack), and additionally a *block* predicate to define the locations friendly pieces can move to *block* an enemy check.

These possible situations were mapped to one of the actions: the king fleeing from an attack, a friendly piece taking the single checking piece, or a friendly piece blocking the single checking piece. The approach is computationally correct, but it is heavy and there were many edge cases during implementation. To give an indication of the code verbosity, a code segment for the situation where it is check from *one piece* is attached below:

```
% friendly piece taking or blocking a sole attacker
possibleMove(R,C,R',C',T) :-
numChecks(1,T),
chessman(king,Col,RK,CK,T),
(RK, CK) != (R,C),
color_to_move(T,Col),
guarded(R',C',R,C,Col,T),
not chessman(_,Col,R',C',T),
row(R'), col(C'),
chessman(EPiece,ECol,ER,EC, T),
guarded(RK,CK,ER,EC,ECol,T),
enemy(Col,ECol),
blockSquare(R',C',T) : linPiece(EPiece, _);
pinMoves(R,C,R',C',T) : pinned(R,C,_,_,_,T),
time(T).
```

3.4.2 Declarative Approach

Our second approach is to utilise a key rule in chess: a player may never make a move such that their king is under attack - i.e. a *legal move* is a move that does not expose the king to an attack. However, our implementation to this point was not capable of this because we only evaluated the next board once a move was selected. The declarative approach used to enforce the above rule, was to always look one move ahead in order to tell if a move is legal or not by checking if the moving player's king is in check in the successor board state.

With this in mind, we modified parts of our original implementation to enforce this new declarative logic. While the previously mentioned logic stays untouched, we put in DMove (a dynamic move) as a term within the chessman predicate - as we would be doing a one-step look-ahead for all dynamic moves in order to determine if they are legal moves or not. Evaluation of predicates including linearAttack, guarded and others are then induced from the new chessman/6 predicate. In other words, originally we had *Time* as the sole identifier for each board, however we now identify each board state with *(Time, DMove)* pairs.

This model works fluently and produces the result that we expected. We presumed this model may work slower since it is evaluating all the successor boards in one step. However, the outcome ended up approximately double the speed of the brute-force approach, and is a good case-study as to the usefulness and elegance of ASP for certain tasks.

3.5 Static Checkmate Planning

In static planning, our model is generating checkmate puzzles where only the white pieces are moving while the black pieces stay fixed in place. The key problem is defined as "Given some initial board state, what is the shortest checkmate for white if the black pieces cannot move?". See figure 2 for model details.

Since the black king cannot move away from attack under this setting, we enforced an integrity constraint such that the black king cannot be in check before the checkmate happens. Minimization is applied to find the shortest checkmate for each puzzle the model generated.

The following code segment shows some important rules and integrity constraints related to static planning:

```
% there must be a checkmate.
:- not {checkmate(N) : time(N)}=1.

% minimize the number of moves until checkmate.
#minimize{N:checkmate(N)}.

% there must be no check at time 1 for white or black.
:- check(_, white, 1).
:- check(_, black, 1).

% colors do not alternate until checkmate.
color_to_move(1, white).
color_to_move(T, white) :-
    color_to_move(T-1, white),
    time(T),
    T < N : checkmate(N).
color_to_move(T, black) :-
    time(T),
    checkmate(T).

% black is not in check until they are checkmated.
:- check(black, T), checkmate(N), T < N.
```

3.6 Dynamic/Adversarial Checkmate Planning

In dynamic planning, we want to generate puzzles where both colors can move. Following similar logic to static planning but enabling black to move will work, i.e. clingo will generate stable models which satisfy all the constraints - however, if the models are interpreted as chess puzzles, they are either incorrect or are not

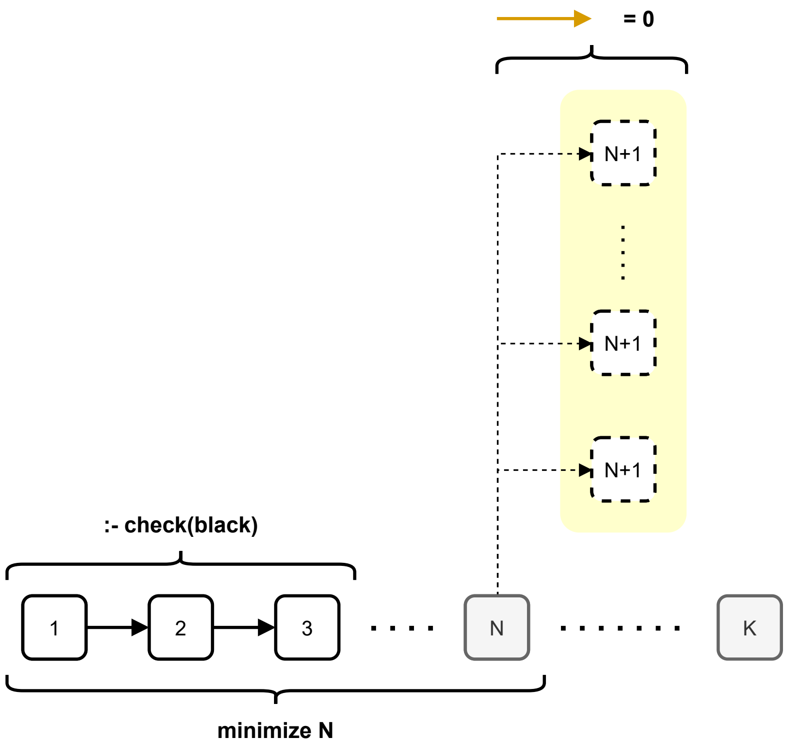


Figure 2: Static Planning: Successor boards of all dynamic moves are generated in order to infer legal moves for all time steps (figure only showing successor boards for black, but exist for both colors). Planning time has a constant upper bound K, for which no moves can be generated after. We enforce there must be a checkmate at time N (the number of legal moves at time N is zero), and it is blacks turn at time N. Additionally we enforce there must be no check on black before they are in checkmate. Legal move: solid arrow. Dynamic move: dotted arrow.

‘good’ puzzles. This problem is due to the fact that the model is not actually looking into the future and evaluating min-max expected value for each action taken, rather Clingo is making choice decisions about which moves get taken at all time-steps seemingly at random.

Our approach around this issue was to constrain the game-tree to limit black from making any choice decisions - flattening the possible board states by enforcing there should be *only one* legal move available for black every black turn. This approach avoids generating models where black purposely plays terrible moves to expedite getting checkmated. See figure 3 for model details.

The following code segment shows some important rules and integrity constraints related to dynamic planning:

```
% minimize the number of moves until checkmate.
#minimize{N:checkmate(N)}.

% colors alternate every move.
color_to_move(1, white).
color_to_move(T, white) :- color_to_move(T-1, black), time(T).
color_to_move(T, black) :- color_to_move(T-1, white), time(T).

% there can not be more than 1 legal move every black turn.
:-  time(T),
    color_to_move(T, black),
    legalMove(R1,C1,R1',C1',T),
    legalMove(R2,C2,R2',C2',T),
    (R1,C1,R1',C1') != (R2,C2,R2',C2').

% *all* legal moves for white *before* checkmate must lead to black
     not in check.
:-  color_to_move(T,white),
    legalMove(R,C,R',C',T),
    check(dynamicMove(R,C,R',C',T), black, T+1),
    T < N-1 : checkmate(N).

% soft constraints: min white piece values, max black piece values.
:~ #sum{V,R,C : chessman(P,white,R,C,1), value(P,V)}=T. [T@10]
:~ #sum{V,R,C : chessman(P,black,R,C,1), value(P,V)}=T. [-T@5]
```

4 Results and Discussion

The main problems we encountered in the project were how to generate a puzzle that is challenging, correct, and fun. This is because for a 2 player turn-based game, each player is trying to maximise his winning share while minimizing the opponent's. However, we could not achieve the min-max evaluation with clingo. This is because:

1. We cannot define a proper heuristic in clingo.
2. We cannot minimize and maximize the steps to checkmate at once.
3. The exponential nature of evaluating a chess game-tree.

Simply put, we cannot have both black and white players making "intelligent" decisions at once. Hence we tried multiple approach to at least make the puzzle more fun without doing a deeper and more advanced look-ahead. Since the problem was that the black player does not play the optimal move if we are minimizing the steps to checkmate, we decided to flatten the the game tree by enforcing black to have only one legal move at each turn. That allows black to make the *best* move since it is the *only* move.

We also tried to enforce that there should be no check at T=2 because the model could always generate a mate in one puzzle which is not very interesting to strong chess players. However, that brings out another problem - where the model generates a puzzle that could be checkmate in one, but white just

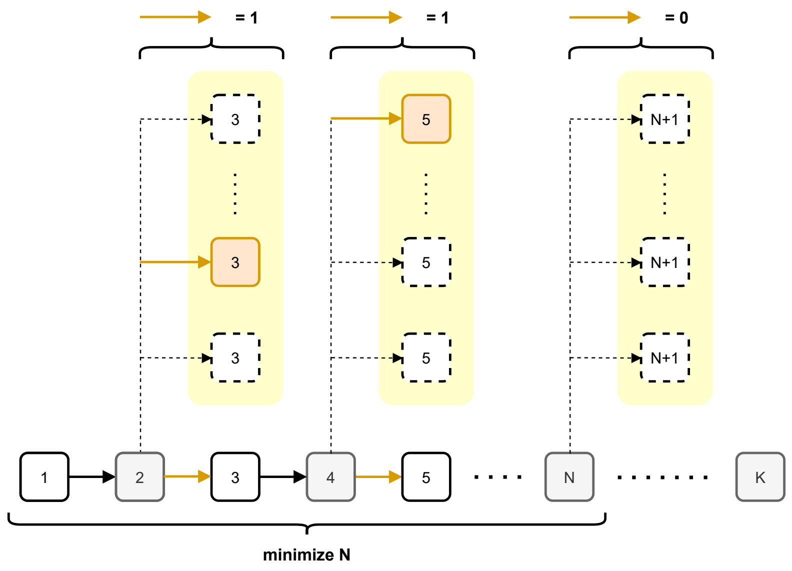


Figure 3: Dynamic Planning: Successor boards of all dynamic moves are generated in order to infer legal moves for all time steps (figure only showing successor boards for black, but exist for both colors). Planning time has a constant upper bound K, for which no moves can be generated after. We enforce there must be a checkmate at time N (the number of legal moves at time N is zero). We enforce for black moves before checkmate, the number of legal moves be one. Legal move: solid arrow. Dynamic move: dotted arrow.

purposely delays the checkmate just to follow the constraint, see figure 4 for a good example of this. We developed a general proposal to address this problem

1. Evaluate checkmate for each successor board
2. Constraint checkmate to be the shortest checkmate
3. Constraint there should not be checkmate for any legal move at T=2

However, given that our checkmate predicate is evaluated from the set of legal moves, which is induced from a successor board state, this proposal suggested that the model should look 2 steps ahead in the future. This would have generated way too many predicates and taken a lot of resources/time to run the model. Therefore we did not actualise this approach.

Instead, we implemented another approach; to have an integrity constraint limiting the model to not have any legal moves for white that would put black in check before black is checkmated. Since check is one of the prerequisites for a checkmate, it could never be the case there is a path to a different checkmate to the one in the model when generating *checkmate in 2* puzzles.

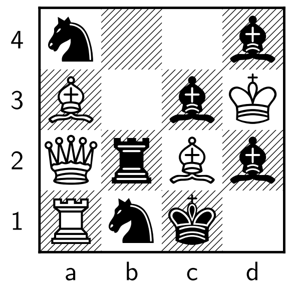


Figure 4: Checkmates in Branches Problem: While Clingo correctly planned for a *checkmate in 2 moves* given the constraints: there must be checkmate at time=4 (blacks second move), black only ever has 1 legal move, and white cannot check black on it's first move. There is a hidden problem: there is a *checkmate in 1 move* (Queen or Rook takes b1) - Oops! While Clingo is indeed showing a satisfiable answer set (there exists a checkmate in 2 given the constraints) - it is not a correct chess puzzle, as there is a faster checkmate than what was specified.

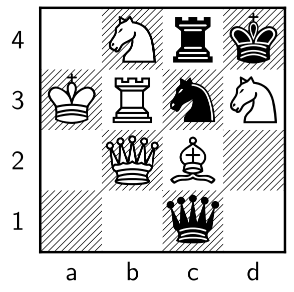
Optimization of piece values was another attempt to make the puzzles more enjoyable. Chess pieces are often associated with a strength value which can be used to relatively rank the strength of chess pieces against each other (queen-9, rook-5, bishop-3, knight-3, pawn-1). By adding soft constraints to our model we were able to generate puzzles where we prefer white to have less valuable pieces and black to have more valuable pieces (making the puzzle harder) - See figure 5 for an example of this.

The most prolific problem with puzzle generation was the case of similar puzzles being generated (see figure 6). Due to the way Clingo finds answers using backtrack-based enumeration, i.e. after computing one puzzle, a series of extremely similar puzzles would be generated with only small, irrelevant changes made to the piece placements - with the exact same moves as the solution. However, it was discovered that by forcing the solver to restart computation after each solution, and enforcing the enumeration style to be *solution recording* as opposed to *backtracking*, we were able to generate multiple, *conceptually different* puzzles in sequence. The appropriate Clingo flags for achieving this are:

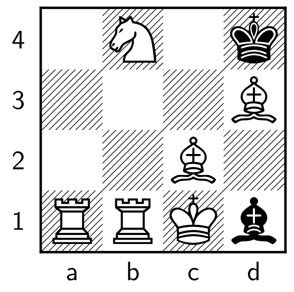
```
clingo --enum-mode record --restart-on-model [files]
```

5 Conclusion

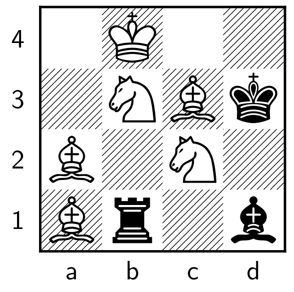
In conclusion we were able to use ASP to model chess (without pawns) effectively; such that, given some predefined starting condition, Clingo can indeed minimize the static and dynamic moves to checkmate under some limiting constraints. However, as previously stated, the project became much less about



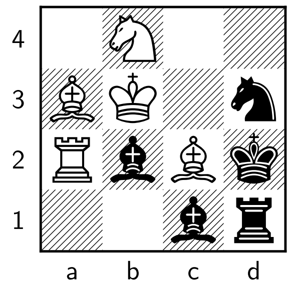
(a) Optimization: 23, -17



(b) Optimization: 19, -3



(c) Optimization: 15, -8



(d) Optimization: 14, -14

Figure 5: Optimization of piece values: the first four optimized models for dynamic *checkmate in 2* problems on a 4x4 board and max white/black pieces = 6. Optimization numbers represents the total white/black piece values on the board at time=1, respectively. Black numbers are negative as they are being maximized. Note minimizing white value has greater priority than maximizing black, and hence from (a) to (b) the total black piece value significantly reduces instead of increasing - while white's decreases (optimizes).

modelling chess, and more about modelling 'chess puzzles' - where there seems to be both a science and an art.

This project was very time consuming as our initial attempts were constantly being reformulated and improved. As mentioned previously, this project is a good case-study into the effectiveness and simplicity of declarative programming over imperative; while our original brute-force approach was correct, it's implementation was extremely long and complicated; while the declarative approach was simple to reason about and implement whilst also being more efficient.

There are still multiple improvements which could be made, and features to be added: e.g. adding pawns, and special chess rules. However, overall the project was very interesting and the authors become significantly more experienced with Clingo and ASP as a result of all the time spent working.

In the end, we did manage to construct a chess puzzle model such that Clingo was able to generate interesting and novel puzzles that even experienced players

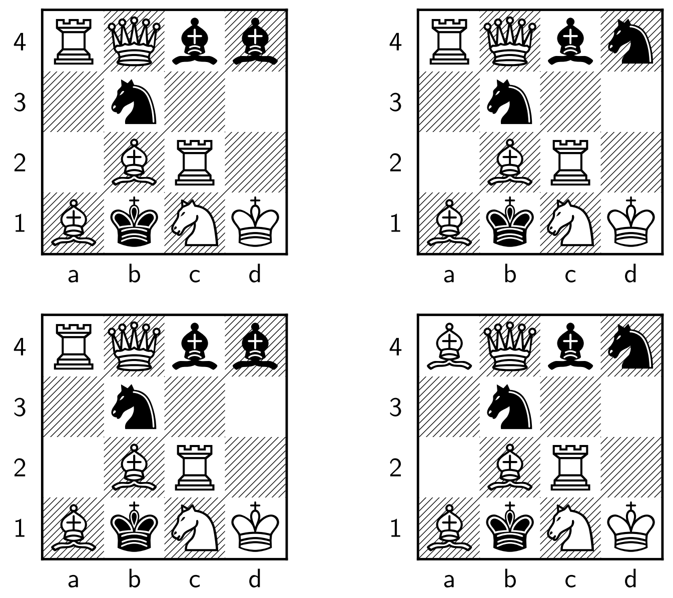


Figure 6: Similar Puzzle Problem: Similar puzzles being generated is common occurrence across both static and dynamic planning. All of the above puzzles were generated by clingo solving the dynamic checkmate in 2 move problem. Even a chess amateur can notice however, the boards are nearly all identical! This is due to Clingo exploiting the fact that quite often auxiliary changes can be made to pieces with zero effect on the essence of a puzzle. All the above puzzles have the same solution: b2d4, c4d3, b4xb3.

were able to routinely enjoy. However, one downside is that if one knows the puzzle constraints it greatly helps puzzle solving...

1. White checkmates in 2 moves.
2. White's first move is never a check.
3. Black only has 1 legal move on their first turn.

An example puzzle generated with this method is shown in figure 7. Please see the accompanying puzzle sheet for hours of fun, enjoy!

6 Acknowledgements

The authors would like to thank Prof. Sebastian Sardina for his insights into building a more declarative chess model, and allowing additional time to complete the task before the original submission date.

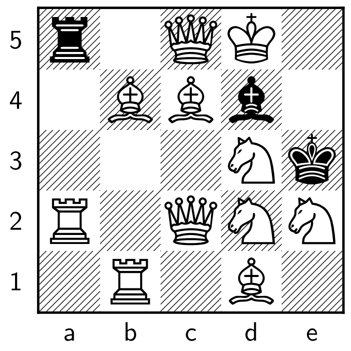


Figure 7: Example 5x5 Puzzle: white to move and checkmate in 2. Answer: a2a5,d4c5,b4c5

References

[1] Martin Gebser et al. *Potassco guide version 2.2.0*. `https://github.com/potassco/guide/releases/tag/v2.2.0`. Accessed: 5-5-2022. 2019.

[2] Tuhola H. *Modeling of Chess in Prolog*. `https://boxbase.org/entries/2018/nov/19/modeling-chess-in-prolog/`. Accessed: 2022–04-12. 2018.

[3] Barney Pell. "Metagame in symmetric chess-like games". In: (1992).

[4] Bin Hu and Marcus Chiu. *Prolog Chess Game*. `https://github.com/he7850/Prolog-chess-game`. Accessed: 5-5-2022. 2019.

[5] Erik Peldan. *Prolog Chess Game*. `https://github.com/epeld/pl-chess`. Accessed: 5-5-2022. 2019.

[6] Edeilson Milhomem da Silva Carvallho et al. "Two Classic Chess Problems Solved by Answer Set Programming". In: ().

[7] Jordan Bell and Brett Stevens. "A survey of known results and research areas for n-queens". In: *Discrete mathematics* 309.1 (2009), pp. 1–31.

[8] Enrico Pontelli. *A 25-Year Perspective on Logic Programming*. Springer, 2010.

[9] James Clarke. *How Training Puzzles are Generated*. `https://lichess.org/blog/U4sjakQAAEAAhH9d/how-training-puzzles-are-generated`. Accessed: 5-5-2022. 2014.

[10] Tord Romstad. *Prolog Chess Game*. `https://blog.playmagnus.com/generating-tactical-puzzles-with-stockfish-and-chess-jl-part-i/`. Accessed: 5-5-2022. 2020.