Final Report:
Chess End Game Puzzle

Caherinet Benoit

December 10, 2008

1 Problem Description and interest

The problem I plan to investigate for this report is end game chess puzzles, particularly the one called direct mate. An end game chess puzzle is a puzzle which starts from a given configuration of pieces on the chess board, and plays with the standard rules, and the player tries to win the game after beginning at the specified configuration. The number of moves the player has to make in order to win is specified in the puzzle and is the minimal number to force the other player into checkmate. Since the player has no opponent, the win must be independent of any move his opponent may be able to do. This implies that after each move of his opponent, the move of the player is unique. Generally such puzzles are in one, two or three moves (one move meaning that the player and his opponent each have a turn). An example of such a puzzle is in figure 1 where it is the white player's turn and he is required to win in two moves. The solution for this particular configuration is: white queen move to c6, black pawn to c6 and finally white bishop to a6.

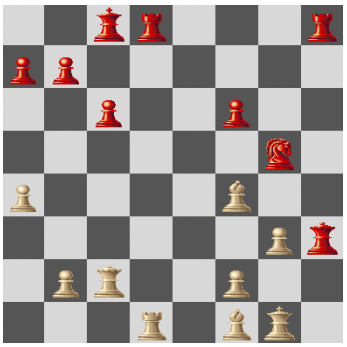

Figure 1: chess puzzle

This particular problem is an interesting problem to solve on several levels. First it can be approached in the idea of using chess artificial intelligence. The problem of chess artificial intelligence and this problem have a lot of similarities: the moves and the branching are exactly the same. The main difference between these two problems is the goal. Artificial intelligence relies on an evaluation function to determine a partial goal because there are too many different moves to be considered to be able to consider all the different possibilities. In my particular problem, I don't have such a problem and I can go through all the possibilities. Another interest of this problem is to see how well ASP performs in solving larger programs because this program present a lot different rules. It may also make this problem quite challenging.

2 Problem modelisation

In order to create a model for this problem, each piece is represented by a number. Four set of atoms will be needed. The first set of fluents is $piece(n, c, l, t)$ which means that at the turn t, the piece n is in column c, line l. For the white piece $n \in [1, Nbwhite]$ and for the black piece $n \in [Nbwhite + 1, Nbwhite + NbBlack]$. The second set of fluents is used to determine what type the piece n is. For this we use $queen(n)$ to mean that the piece n is a queen, $king(n)$ to mean that the piece n is king, etc. Furthermore we use the action $play(n, c, l, t)$ to mean that at the turn n we play the piece n and that the destination cell is column c, line l. Finally, I use a last set of fluents $alive(n, t)$ to mean that at the turn t the piece n is alive.

3 Play generation and constraints

The first thing we need to do in order to solve this planning problem is to generate the play. For each turn, there must only be a single play. Furthermore during an even turn it must be a white piece which is played and during the odd turn a black piece:

$$1 \leq \{play(w, c, l, t) : w \in White : \{c, l\} \in [1, 8]\} \leq 1 \qquad t \in \{0, 2, ..\}$$

$$1 \leq \{play(b, c, l, t) : w \in Black : \{c, l\} \in [1, 8]\} \leq 1 \qquad t \in \{1, 3, ..\}$$

When you play a piece, the piece must move and does not stay in the same place :

$$\leftarrow piece(p, c, l, t), play(p, c, l, t)$$

Another constraint is that the only piece which can jump over another piece is the knight:

$$\leftarrow piece(p1, c1, l1, t), piece(p2, c1, l2, t), play(p1, c1, l3, t) \qquad (l1 < l2 < l3) \qquad (1)$$

$$\leftarrow piece(p1, c1, l1, t), piece(p2, c2, l1, t), play(p1, c3, l1, t) \qquad (c1 < c2 < c3) \qquad (2)$$

$$\leftarrow piece(p1, c1, l1, t), piece(p2, c1, l2, t), play(p1, c3, l3, t) \qquad (|c2 - c1| = |l2 - l1|, \qquad (3)$$
$$|C3 - C1| = |L3 - L1|, L1 < L2 < L3, C1 < C2 < C3.)$$

Here p1 is the piece which is moving and p2 the piece which will be jumped over. The rule (**??**) is that in the case where the play is on a column, (**??**) that it is on a line and (**??**) that it is on a diagonal. The knight destination is not on the same line, neither on the same column and neither on the same diagonal and thus will not be affected by this rules.

Each type of piece corresponds to an additional constraint. For example to the knight corresponds the following constraint:

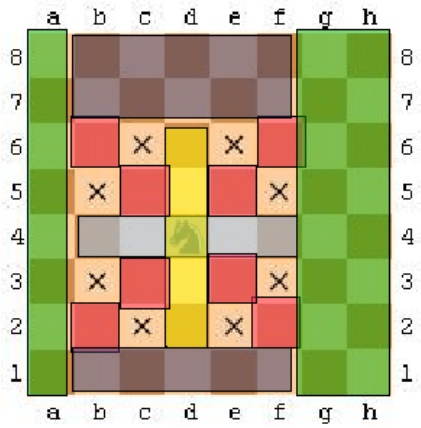

Figure 2: Possible knight move

$$\leftarrow knight(p), piece(p, c1, l1, t), play(p, c2, l2, t) \qquad (|c2 - c1| > 2) \tag{4}$$
$$\leftarrow knight(p), piece(p, c1, l1, t), play(p, c2, l2, t) \qquad (|l2 - l1| > 2, |c2 - c1| <= 2) \tag{5}$$
$$\leftarrow knight(p), piece(p, c1, l1, t), play(p, c1, l2, t) \tag{6}$$
$$\leftarrow knight(p), piece(p, c1, l1, t), play(p, c2, l1, t) \tag{7}$$
$$\leftarrow knight(p), piece(p, c1, l1, t), play(p, c2, l2, t) \qquad (|c2 - c1| = |l2 - l1|) \tag{8}$$

The rules (**??**) and (**??**) prevent the knight from jumping over more than two squares away in both directions. The rules (**??**), (**??**) and (**??**) prevent the knight from jumping on the same column, on the same line, and on the same diagonal respectively. Similar rules are created for the king, the queen, knights, bishops, rooks and pawns (see appendix for more details).

Now that all the basic constraints have been set, the fact that a piece is alive or dead needs to be represented. In order to express that with a simple move, the piece stays in the same state (alive or dead), true negation needs to be used.

$$alive(p, t + 1) \leftarrow alive(p, t), not \sim alive(p, t + 1) \tag{9}$$
$$\sim alive(p, t + 1) \leftarrow \sim alive(p, t) \tag{10}$$

The rule (**??**) means that if the piece was alive and there is no information to the contrary, the piece will remain alive. The rule (**??**) means that if the piece is dead, it will stay dead no matter what. Furthermore if there is a piece on the destination of a move, then this piece become dead:

$$\sim alive(p2, t + 1) \leftarrow play(p1, c, l, t), piece(p2, c, l, t)$$

And all the pieces which are not dead and are still alive, are on the board for the next turn:

$$piece(p, c, l, t + 1) \leftarrow piece(p, c, l, t), not\ playPiece(p, t), alive(p, t)$$

To finish the goal (for now) is that the last play kills the opponent king:

$$\leftarrow not \sim alive(b, lastTurn + 1), king(b) \qquad b \in black$$

4 Description of the problem of the above implementation

In the implementation above, the goal was to find black and white moves such that at the end the black king is in checkmate. The white and black played exactly the same role. But it's not really what we want. What we really want is that for every valid balck move, there is a mean for the white to win. This can be translated in the following formula where w and b represent the white and balck moves. In the program above what we were doing was:

$$\exists w1 \exists b1 \exists w2 \exists b2 \exists b3 \; black \; king \; dead$$

when what we really want is:

$$\exists w1 \forall b1 \exists w2 \forall b2 \exists b3 \; black \; king \; check \; mate$$

Solving this problem seems to be quite a big challenge since such problems have a complexity of σ^{3p}. ASP is only able to solve σ^{2p} so this problem is probably not solvable with only one ASP program. So after searching for a solution with only one program I tried to solve it in several. The solution I have come up with is explained in the following paragraph.

5 Detecting Checkmate

A first problem in solving this problem in several programs is the fact that the number of answer sets was too big. The preceeding implementation returned thousand of answer sets since the constraint were very weak. Exploiting so many answer sets in order to determine which is the right one is thus not practical. It's why it is needed to first reduce the number of answer sets present.

In order to do that, the best aproach seams to change the goal of the problem. In spite of checking if the black king is dead, the idea is now to check that the black king is checkmate (that the king can't move without beeing killed) two turns before. This concept relies on the following formula:

$$\exists w1 \forall b1 \exists w2 \forall b2 \exists b3 \; black \; king \; dead = \exists w1 \forall b1 \exists w2 \; black \; king \; check \; mate$$

This change will thus take care of $\forall b2$. In order to decide if the king is check mate I turn this part into a static problem. The king is check mate if all the square around it are reachable by a white piece. This is described in the following picture:

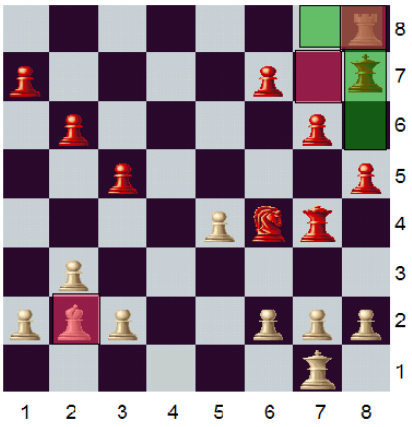

Figure 3: Checkmate detection

In order to create a model for this problem, I have added the fluent $threaten(n, c, l, t)$ which means that the square of column c and line l is "reachable" but the piece n at the time t. This fluent is quite similar to the fluent play. The way to restrict the move for the different piece is the same. But there is also many differences. The first big differences in how threats are generated. In order to detect that all the empty squares around the king are reachable by a white piece, not only a unique threat like for play but ensure that there is at least one:
If the black king is in (c1,l1) and t is the last turn before checkmate:

$$1 \leq \{threaten(w, c1, l1, t+1) : w \in White\} \tag{11}$$

$$1 \leq \{threaten(w, c, l, t+1) : w \in White\} \qquad |c - c1| < 2, |l - l1| < 2,\ (c, l)\ is\ inoccupied. \tag{12}$$

Another change is that a square is threatened even if the king is between the origin and destination of the play. For example, in figure 3 this change is needed so that the rook threatens column 8, line 6, if it wasn't, the king could have move away there. This change can only be made by adding $notking(..)$ in the coresponding rules.

Another thing we need to ensure is that when it is the white's turn to play, then it is not possible play to kill the black king. If there is an opportunity, it means that the preceeding black move was not valid. In order to check that, I didn't manage to retrieve the coresponding answer set but I managed to generate a new answer set similar to the one we want to retrieve but with bad sets in it. In order to do that I generate "bad" if the king is threaten: If the black king is in (c1,l1) and t is a white turn:

$$1 \leq \{threaten(w, c1, l1, t) : w \in White\} \tag{13}$$

$$bad \leftarrow threaten(w, c1, l1, t) \qquad w \in White \tag{14}$$

The answer set which I get now are the following:

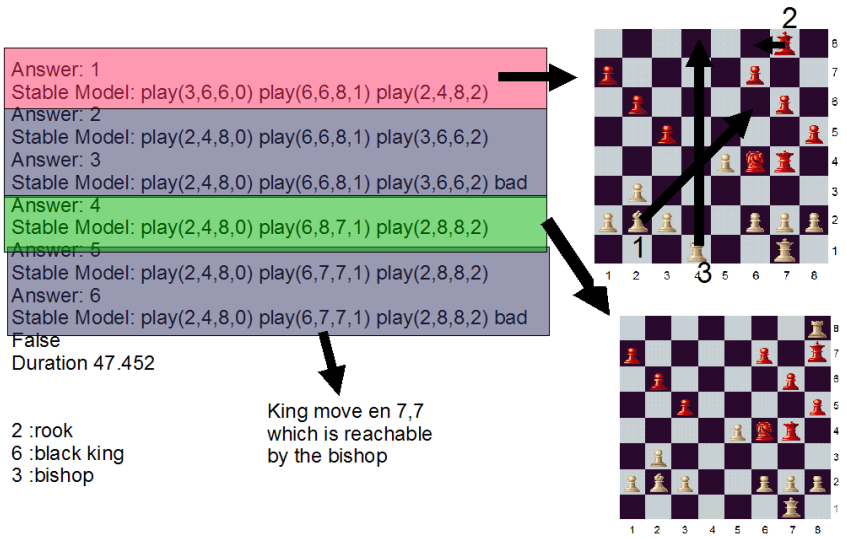

Figure 4: Answer set with check mate detection

There is now a reasonable number of answer sets in comparism to the number I previously got. In this figure the green answer set is the one we are hoping to get. The blue ones are the ones which are not good because because the second white move could have killed the king. The last answer set which is in red is the one which still is a problem. When playing the move of this answer set, in the resuling state the king is checkmate but the problem is that the black could have avoid that by moving their king to (7,7) or even not moving the king and playing any other piece. The reason we got this answer set is because of the $\forall b1$ which was presented above.

6 A "second" ASP program

Like it was mentioned above, solving this last one problem in only one ASP program is probably not possible. This is why I tried to do a second ASP program which looked to see if each previous answer set is valid. What I finally came up with is given the first white move, to be able to generate all the valid black moves. Then we need to compare the answer set from both our programs. If given the first white move, after retrieving all the answer set mark has bad we have the same answer set by both program then this is a good answer set otherwise it's not. The only major change with the preceeding program is that we retrieve the 3 line of generation of checkmate detection and we add the rule with the first white play. The examples are in anexe.

7 Conclusion

When trying to solve this problem I was looking for a challenge, and I think I found several. Probably more than I can handle. The first part was quite fastidious but interesting too. Even though I didn't encounter any major problem, determining all the rules of chess and creating the constraints for them was not ver easy. I am additionally quite surprised that despite the fact that there are so many rules, the grounding take less than one minute. The second part about determining if the king is in check mate was more a challenge. I tried different approach but which all failed before I finallycome up with my finall aproach. The reason they failed is quite similar to the reason I didn't managed to retrieve the answer set where the king can be killed before the end. The main problem being that it was not very easy to come up with a ASP program which answer an yes/no question by giving **only one** answer set if yes and no answer set if no. To finish I am partially disapointed by the last part, I spent a lot of time looking for a solution in the same program and never really managed to stop searching even knowing that it was not possible. I think my time would have been better spent to improve some other part of the program. Furthermore the solution I finally came up with seemed unsatisfying to me.

To conclude, the interesting point of this problem is that even though the program was big, ASP handle it quite well. Furthermore, the flexibility of ASP finally permits me to create the detection of checkmate by using the rules of play without changing them too much. In a traditional programming language, detecting checkmate would probably have been very different from playing moves. I have also managed, given this puzzle, to be able to detect what is the solution even if it is not done in one step.

8 Bibliography

Chess rules:
http://www.chess.com/learn-how-to-play-chess.html
Description of chess end game puzzle:
<http://en.wikipedia.org/wiki/Chess_endgame>
Example of chess end game puzzle:
<http://onlinechessstrategy.com/category/chess-endgame-puzzles/>
<http://homepage.ntlworld.com/adam.bozon/ChessPuzzles.htm>

9 Apendix A: source code

GeneralPlay.p:

```
%to be assert: white: the number of white piece; black:the number of black piece; t: the
numberWhite(1..white).
numberBlack(white+1..black+white).
number(1..black+white).

column(1..8).
line(1..8).

turn(0..t).

#domain numberWhite(W).
#domain numberBlack(B).
#domain number(I1;I2).
#domain column(C1;C2;C3).
#domain line(L1;L2;L3).
#domain turn(T).

%piece(n,c,l,t) at turn t the piece n is in column c and line l
%play(n,c,l,t) play the piece n at turn t and move it to column c, line l
%alive(n,t) the piece n is alive at time t

%at t=0 every piece is alive
alive(I1,0).

%if it is not kill, stay alive
alive(I1,T+1):- alive(I1,T),not -alive(I1,T+1).
%when it is dead, it stay dead
-alive(I1,T+1):- -alive(I1,T).

%one and only one play by turn
%T is even thus it is to the white to play
1{play(W1,C,L,T) :numberWhite(W1):column(C):line(L)}1 :- T==(T/2)*2.
%T is odd thus it is to the black to play
1{play(B1,C,L,T) :numberBlack(B1):column(C):line(L)}1 :- T!=(T/2)*2.

%when play don't stay at the same place
:- piece(I1,C2,L2,T),play(I1,C2,L2,T).

%if there is a piece between the origin and destination of the play, then play impossibl
:- piece(I1,C1,L1,T), piece(I2,C1,L2,T), play(I1,C1,L3,T), L1<L2, L2<L3.
```

```
:- piece(I1,C1,L1,T), piece(I2,C2,L1,T),play(I1,C3,L1,T), C1<C2, C2<C3.
:- piece(I1,C1,L1,T), piece(I2,C1,L2,T), play(I1,C1,L3,T), L3<L2, L2<L1.
:- piece(I1,C1,L1,T), piece(I2,C2,L1,T),play(I1,C3,L1,T), C3<C2, C2<C1.
:- piece(I1,C1,L1,T), piece(I2,C1,L2,T),play(I1,C3,L3,T), abs(C2-C1)==abs(L2-L1), abs(C3

%play the piece I1 to column C1, line L1
piece(I1,C1,L1,T+1):- play(I1,C1,L1,T).
playPiece(I1,T):-play(I1,C1,L1,T).

%the piece we don't play don't move if it is not dead
piece(I1,C1,L1,T+1):- piece(I1,C1,L1,T), not playPiece(I1,T), alive(I1,T).

%TODO : can't kill his own piece

%impossible to play a piece which is dead
:- play(I1,C1,L1,T), -alive(I1,T).

%if there is a piece at the destination of the play then this piece is dead
-alive(I2,T+1):- play(I1,C1,L1,T), piece(I2,C1,L1,T).

%the goal is to check mate the black king : goal of precedent implementation
%:- not -alive(B,t+1), king(B).

hide.
show play(_,_,_,_).
show bad.
%show alive(_,_).
```

PieceMove.p:

```
%%%%%%% Restrict the different move of the different piece %%%%%%%%%%

%move impossible for a queen
:- queen(I1),piece(I1,C1,L1,T),play(I1,C2,L2,T), C1!=C2, L1!=L2, abs(C2-C1)!=abs(L2-L1).

%move impossible for a king
:- king(I1),piece(I1,C1,L1,T),play(I1,C2,L2,T), abs(C2-C1)>1.
:- king(I1),piece(I1,C1,L1,T),play(I1,C2,L2,T), abs(L2-L1)>1, abs(C2-C1)<=1.

%move impossible for a rook
:- rook(I1),piece(I1,C1,L1,T),play(I1,C2,L2,T), C1!=C2, L1!=L2.

%move impossible for a bishop
```

```
:- bishop(I1),piece(I1,C1,L1,T),play(I1,C2,L2,T), abs(C2-C1)!=abs(L2-L1).

%move impossible for a knight
:- knight(I1),piece(I1,C1,L1,T),play(I1,C2,L2,T), (C2-C1)*(C2-C1)+(L2-L1)*(L2-L1)!=5.
%:- knight(I1),piece(I1,C1,L1,T),play(I1,C2,L2,T), abs(C2-C1)>2.
%:- knight(I1),piece(I1,C1,L1,T),play(I1,C2,L2,T), abs(L2-L1)>2, abs(C2-C1)<=2.
%:- knight(I1),piece(I1,C1,L1,T),play(I1,C1,L2,T).
%:- knight(I1),piece(I1,C1,L1,T),play(I1,C2,L1,T).
%:- knight(I1),piece(I1,C1,L1,T),play(I1,C2,L2,T), abs(C2-C1)==abs(L2-L1).


%move impossible for a pawn
:- pawn(I1),piece(I1,C1,L1,T),play(I1,C2,L2,T), C2!=C1.
:- pawn(W),piece(W,C1,L1,T),play(W,C2,L2,T), L2!=L1+1, C2==C1.%the white
:- pawn(B),piece(B,C1,L1,T),play(B,C2,L2,T), L2!=L1-1, C2==C1.%the black
%TODO more accurate def
```

puzzle2simplified.p:

```
%5 white, 6 black


%%% white %%%%
piece(1,7,1,0).
king(1).

piece(2,4,1,0).
rook(2).



piece(3,2,2,0).
bishop(3).



piece(4,7,2,0).
pawn(4).
```

```
piece(5,5,4,0).
pawn(5).


%%% black%%%
piece(6,7,8,0).
king(6).



piece(7,6,7,0).
pawn(7).



piece(8,7,6,0).
pawn(8).

piece(9,8,5,0).
pawn(9).



piece(10,6,4,0).
knight(10).

piece(11,7,4,0).
queen(11).
```

checkmate

```
turn1(0..(t+1)).
#domain turn1(Tplus).


%piece(n,c,l,t) at turn t the piece n is in column c and line l
%threaten(n,c,l,t) the piece n at turn t threaten the column c and line l

%the goal should not be reachable befre
%When it is white to play the black king should not be threaten
{threaten(W1,C1,L1,T) : numberWhite(W1)}:-king(B),piece(B,C1,L1,T),T==(T/2)*2.
bad:-threaten(W,C1,L1,T).
```

```
%the black king need to stay alive before the end of the game
:- -alive(B,t+1), king(B).

occupiedBlack(C1,L1,t+1):-1{piece(B1,C1,L1,t+1) : numberBlack(B1)}.
1{threaten(W1,C1,L1,t+1) : numberWhite(W1)}:-king(B),piece(B,C1,L1,t+1).
1{threaten(W1,C2,L2,t+1) : numberWhite(W1)}:-king(B),piece(B,C1,L1,t+1), not occupiedBla


%general restriction

%impossible to threaten a piece which is dead
:- threaten(I1,C1,L1,Tplus), -alive(I1,Tplus).

%when threaten don't stay at the same place
:- piece(I1,C2,L2,Tplus),threaten(I1,C2,L2,Tplus).

% if there is a piece between the origin and destination of the threat, then threaten im
% (L3-L1)/(C3-C1)=k=(L2-L1)/(C2-C1) and 0<k<1 => abs(k)<1 &  k<
%:- piece(I1,C1,L1,Tplus), piece(I2,C1,L2,Tplus), threaten(I1,C1,L3,Tplus),not king(I2),
:- piece(I1,C1,L1,Tplus), piece(I2,C1,L2,Tplus), threaten(I1,C1,L3,Tplus),not king(I2),
:- piece(I1,C1,L1,Tplus), piece(I2,C2,L1,Tplus),threaten(I1,C3,L1,Tplus),not king(I2), C
:- piece(I1,C1,L1,Tplus), piece(I2,C1,L2,Tplus), threaten(I1,C1,L3,Tplus),not king(I2),
:- piece(I1,C1,L1,Tplus), piece(I2,C2,L1,Tplus),threaten(I1,C3,L1,Tplus),not king(I2), C
:- piece(I1,C1,L1,Tplus), piece(I2,C1,L2,Tplus),threaten(I1,C3,L3,Tplus),not king(I2), a

%%%%%%% Restrict the different move of the different piece %%%%%%%%%

%move impossible for a queen
:- queen(I1),piece(I1,C1,L1,Tplus),threaten(I1,C2,L2,Tplus), C1!=C2, L1!=L2, abs(C2-C1)!

%move impossible for a king
:- king(I1),piece(I1,C1,L1,Tplus),threaten(I1,C2,L2,Tplus), abs(C2-C1)>1.
:- king(I1),piece(I1,C1,L1,Tplus),threaten(I1,C2,L2,Tplus), abs(L2-L1)>1, abs(C2-C1)<=1.

%move impossible for a rook
:- rook(I1),piece(I1,C1,L1,Tplus),threaten(I1,C2,L2,Tplus), C1!=C2, L1!=L2.

%move impossible for a bishop
:- bishop(I1),piece(I1,C1,L1,Tplus),threaten(I1,C2,L2,Tplus), abs(C2-C1)!=abs(L2-L1).

%move impossible for a knight
:- knight(I1),piece(I1,C1,L1,Tplus),threaten(I1,C2,L2,Tplus), (C2-C1)*(C2-C1)+(L2-L1)*(L
```

```
%:- knight(I1),piece(I1,C1,L1,Tplus),threaten(I1,C2,L2,Tplus), abs(C2-C1)>2.
%:- knight(I1),piece(I1,C1,L1,Tplus),threaten(I1,C2,L2,Tplus), abs(L2-L1)>2, abs(C2-C1)<
%:- knight(I1),piece(I1,C1,L1,Tplus),threaten(I1,C1,L2,Tplus).
%:- knight(I1),piece(I1,C1,L1,Tplus),threaten(I1,C2,L1,Tplus).
%:- knight(I1),piece(I1,C1,L1,Tplus),threaten(I1,C2,L2,Tplus), abs(C2-C1)==abs(L2-L1).


%move impossible for a pawn
:- pawn(I1),piece(I1,C1,L1,Tplus),threaten(I1,C2,L2,Tplus), C2!=C1.
:- pawn(W),piece(W,C1,L1,Tplus),threaten(W,C2,L2,Tplus), L2!=L1+1, C2==C1.%the white
:- pawn(B),piece(B,C1,L1,Tplus),threaten(B,C2,L2,Tplus), L2!=L1-1, C2==C1.%the black
%TODO more accurate def

%hide.
%show threaten(_,_,_,_).
```

10 Appendix B: Test

result of puzzle2:

```
lparse --tru
e-negation -c white=5 -c black=6 -c t=2 checkmate generalPlay.lp pieceMove.lp Simplified
smodels version 2.26. Reading...done
Answer: 1
Stable Model: play(3,6,6,0) play(6,6,8,1) play(2,4,8,2)
Answer: 2
Stable Model: play(2,4,8,0) play(6,6,8,1) play(3,6,6,2)
Answer: 3
Stable Model: play(2,4,8,0) play(6,6,8,1) play(3,6,6,2) bad
Answer: 4
Stable Model: play(2,4,8,0) play(6,8,7,1) play(2,8,8,2)
Answer: 5
Stable Model: play(2,4,8,0) play(6,7,7,1) play(2,8,8,2)
Answer: 6
Stable Model: play(2,4,8,0) play(6,7,7,1) play(2,8,8,2) bad
False
Duration 47.452
Number of choice points: 5
Number of wrong choices: 5
Number of atoms: 8185
Number of rules: 1833956
Number of picked atoms: 1691
```

```
Number of forced atoms: 391
Number of truth assignments: 269583
Size of searchspace (removed): 42 (38)
```

T he output from the second ASP program for play(2,4,8,0). All the answer set but the one with play(6,8,7,1) are mark bad. Thus answerset 4 of the first program is a good answer set

```
>lparse -d no
ne --true-negation -c white=5 -c black=6 -c
t=1 generalPlay.lp pieceMove.lp Simp
lifiedPuzzle2.lp checkmatebis | smodels 0
smodels version 2.26. Reading...done
Answer: 1
Stable Model: play(2,4,8,0) play(6,8,8,1)
Answer: 2
Stable Model: play(2,4,8,0) play(6,8,8,1) bad
Answer: 3
Stable Model: play(2,4,8,0) play(6,8,8,1) bad
Answer: 4
Stable Model: play(2,4,8,0) play(6,8,8,1) bad
Answer: 5
Stable Model: play(2,4,8,0) play(6,6,8,1) bad
Answer: 6
Stable Model: play(2,4,8,0) play(6,7,7,1)
Answer: 7
Stable Model: play(2,4,8,0) play(6,7,7,1) bad
Answer: 8
Stable Model: play(2,4,8,0) play(6,6,7,1)
Answer: 9
Stable Model: play(2,4,8,0) play(6,8,7,1)
Answer: 10
Stable Model: play(2,4,8,0) play(6,6,8,1)
Answer: 11
Stable Model: play(2,4,8,0) play(11,6,4,1)
Answer: 12
Stable Model: play(2,4,8,0) play(11,6,4,1) bad
Answer: 13
Stable Model: play(2,4,8,0) play(9,8,4,1)
Answer: 14
Stable Model: play(2,4,8,0) play(9,8,4,1) bad
Answer: 15
Stable Model: play(2,4,8,0) play(11,5,6,1)
```

```
Answer: 16
Stable Model: play(2,4,8,0) play(11,5,6,1) bad
Answer: 17
Stable Model: play(2,4,8,0) play(11,6,5,1)
Answer: 18
Stable Model: play(2,4,8,0) play(11,6,5,1) bad
Answer: 19
Stable Model: play(2,4,8,0) play(7,6,6,1)
Answer: 20
Stable Model: play(2,4,8,0) play(7,6,6,1) bad
Answer: 21
Stable Model: play(2,4,8,0) play(8,7,5,1)
Answer: 22
Stable Model: play(2,4,8,0) play(8,7,5,1) bad
Answer: 23
Stable Model: play(2,4,8,0) play(10,4,3,1)
Answer: 24
Stable Model: play(2,4,8,0) play(10,4,3,1) bad
Answer: 25
Stable Model: play(2,4,8,0) play(11,7,5,1)
Answer: 26
Stable Model: play(2,4,8,0) play(11,7,5,1) bad
Answer: 27
Stable Model: play(2,4,8,0) play(11,8,5,1)
Answer: 28
Stable Model: play(2,4,8,0) play(11,8,5,1) bad
Answer: 29
Stable Model: play(2,4,8,0) play(10,8,5,1)
Answer: 30
Stable Model: play(2,4,8,0) play(10,8,5,1) bad
Answer: 31
Stable Model: play(2,4,8,0) play(11,8,4,1)
Answer: 32
Stable Model: play(2,4,8,0) play(11,8,4,1) bad
Answer: 33
Stable Model: play(2,4,8,0) play(10,5,6,1)
Answer: 34
Stable Model: play(2,4,8,0) play(10,5,6,1) bad
Answer: 35
Stable Model: play(2,4,8,0) play(11,6,3,1)
Answer: 36
Stable Model: play(2,4,8,0) play(11,6,3,1) bad
Answer: 37
```

```
Stable Model: play(2,4,8,0) play(11,8,3,1)
Answer: 38
Stable Model: play(2,4,8,0) play(11,8,3,1) bad
Answer: 39
Stable Model: play(2,4,8,0) play(10,8,3,1)
Answer: 40
Stable Model: play(2,4,8,0) play(10,8,3,1) bad
Answer: 41
Stable Model: play(2,4,8,0) play(11,7,6,1)
Answer: 42
Stable Model: play(2,4,8,0) play(11,7,6,1) bad
Answer: 43
Stable Model: play(2,4,8,0) play(10,7,6,1)
Answer: 44
Stable Model: play(2,4,8,0) play(10,7,6,1) bad
Answer: 45
Stable Model: play(2,4,8,0) play(11,7,2,1)
Answer: 46
Stable Model: play(2,4,8,0) play(11,7,2,1) bad
Answer: 47
Stable Model: play(2,4,8,0) play(10,7,2,1)
Answer: 48
Stable Model: play(2,4,8,0) play(10,7,2,1) bad
Answer: 49
Stable Model: play(2,4,8,0) play(11,7,3,1)
Answer: 50
Stable Model: play(2,4,8,0) play(11,7,3,1) bad
Answer: 51
Stable Model: play(2,4,8,0) play(10,4,5,1)
Answer: 52
Stable Model: play(2,4,8,0) play(10,4,5,1) bad
Answer: 53
Stable Model: play(2,4,8,0) play(11,4,1,1)
Answer: 54
Stable Model: play(2,4,8,0) play(11,4,1,1) bad
Answer: 55
Stable Model: play(2,4,8,0) play(11,4,7,1)
Answer: 56
Stable Model: play(2,4,8,0) play(11,4,7,1) bad
Answer: 57
Stable Model: play(2,4,8,0) play(11,5,2,1)
Answer: 58
Stable Model: play(2,4,8,0) play(11,5,2,1) bad
```

```
Answer: 59
Stable Model: play(2,4,8,0) play(11,3,8,1)
Answer: 60
Stable Model: play(2,4,8,0) play(11,3,8,1) bad
Answer: 61
Stable Model: play(2,4,8,0) play(10,5,2,1)
Answer: 62
Stable Model: play(2,4,8,0) play(10,5,2,1) bad
False
Duration 34.078
Number of choice points: 61
Number of wrong choices: 61
Number of atoms: 5726
Number of rules: 1301323
Number of picked atoms: 1199
Number of forced atoms: 0
Number of truth assignments: 36488
Size of searchspace (removed): 36 (11)
```

The output from the second ASP program for play(3,6,6,0). Here a lot of answer set are present and not bad which were not in the answer set of the first program thus the first answer set of the first program is not a good answer set.

```
>lparse -d no
ne --true-negation -c white=5 -c black=6 -c t=1 generalPlay.lp pieceMove.lp Simp
lifiedPuzzle2.lp checkmatebis | smodels 0
smodels version 2.26. Reading...done
Answer: 1
Stable Model: play(3,6,6,0) play(7,6,6,1)
Answer: 2
Stable Model: play(3,6,6,0) play(6,7,7,1)
Answer: 3
Stable Model: play(3,6,6,0) play(6,7,7,1) bad
Answer: 4
Stable Model: play(3,6,6,0) play(11,6,4,1)
Answer: 5
Stable Model: play(3,6,6,0) play(10,4,3,1)
Answer: 6
Stable Model: play(3,6,6,0) play(11,5,2,1)
Answer: 7
Stable Model: play(3,6,6,0) play(11,5,6,1)
Answer: 8
Stable Model: play(3,6,6,0) play(10,5,2,1)
```

```
Answer: 9
Stable Model: play(3,6,6,0) play(6,8,7,1)
Answer: 10
Stable Model: play(3,6,6,0) play(6,6,8,1)
Answer: 11
Stable Model: play(3,6,6,0) play(11,7,2,1)
Answer: 12
Stable Model: play(3,6,6,0) play(8,7,5,1)
Answer: 13
Stable Model: play(3,6,6,0) play(10,7,6,1)
Answer: 14
Stable Model: play(3,6,6,0) play(11,8,3,1)
Answer: 15
Stable Model: play(3,6,6,0) play(11,8,4,1)
Answer: 16
Stable Model: play(3,6,6,0) play(11,8,5,1)
Answer: 17
Stable Model: play(3,6,6,0) play(6,8,8,1)
Answer: 18
Stable Model: play(3,6,6,0) play(6,6,7,1)
Answer: 19
Stable Model: play(3,6,6,0) play(11,7,6,1)
Answer: 20
Stable Model: play(3,6,6,0) play(10,5,6,1)
Answer: 21
Stable Model: play(3,6,6,0) play(11,6,3,1)
Answer: 22
Stable Model: play(3,6,6,0) play(11,6,5,1)
Answer: 23
Stable Model: play(3,6,6,0) play(11,3,8,1)
Answer: 24
Stable Model: play(3,6,6,0) play(10,4,5,1)
Answer: 25
Stable Model: play(3,6,6,0) play(10,8,5,1)
Answer: 26
Stable Model: play(3,6,6,0) play(11,7,5,1)
Answer: 27
Stable Model: play(3,6,6,0) play(11,4,1,1)
Answer: 28
Stable Model: play(3,6,6,0) play(9,8,4,1)
Answer: 29
Stable Model: play(3,6,6,0) play(10,7,2,1)
Answer: 30
```

```
Stable Model: play(3,6,6,0) play(10,8,3,1)
Answer: 31
Stable Model: play(3,6,6,0) play(11,7,3,1)
Answer: 32
Stable Model: play(3,6,6,0) play(11,4,7,1)
False
Duration 34.234
Number of choice points: 31
Number of wrong choices: 31
Number of atoms: 5726
Number of rules: 1301323
Number of picked atoms: 1244
Number of forced atoms: 2
Number of truth assignments: 39307
Size of searchspace (removed): 32 (13)
```