Automated reasoning about retrograde chess problems using Coq

Marko Maliković, Ph.D.
The Faculty of Humanities and Social Sciences
University of Rijeka, CROATIA

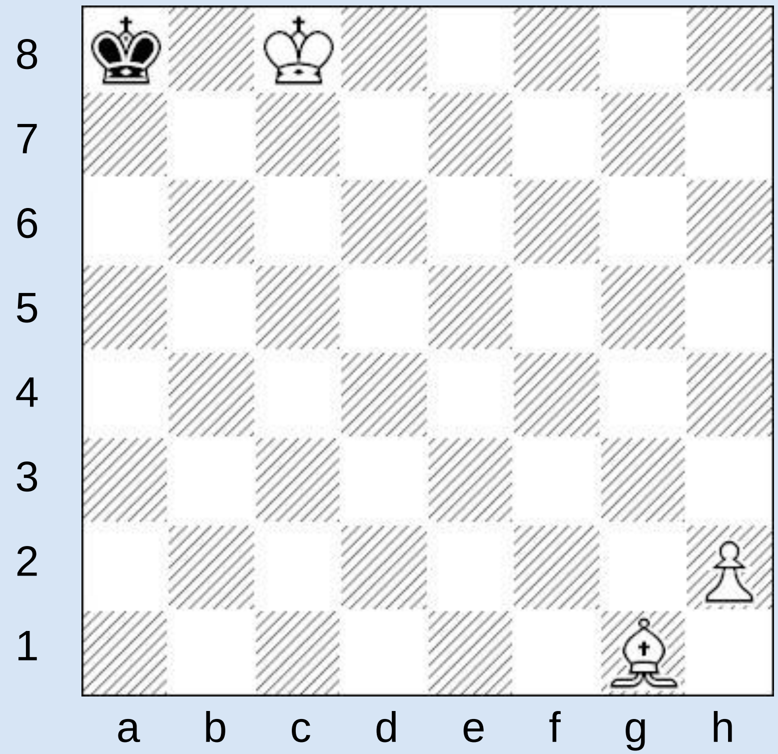
Retrograde chess analysis

Method that determine which moves:

1. have to be 2. could be

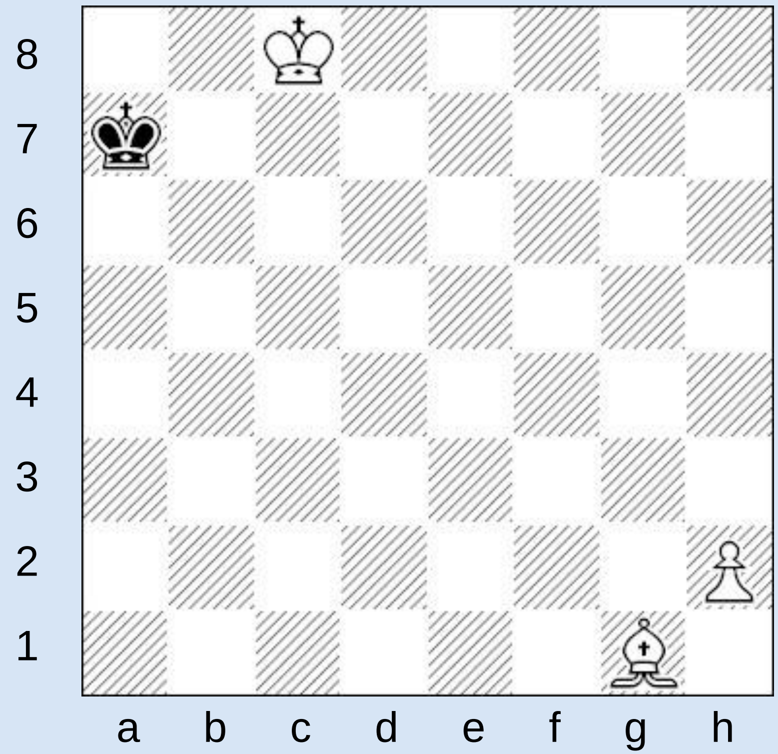
played leading up to a given chess position

What did Black just play?
What was white's move before that?



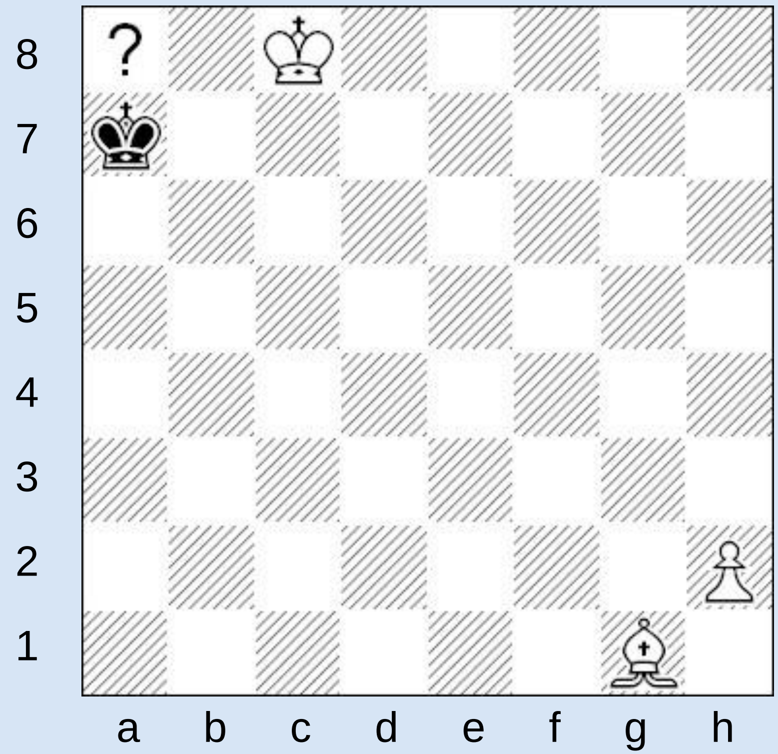
Black did move his King (his only piece) from *a7* (only possible square)!

What did Black just play?
What was white's move before that?



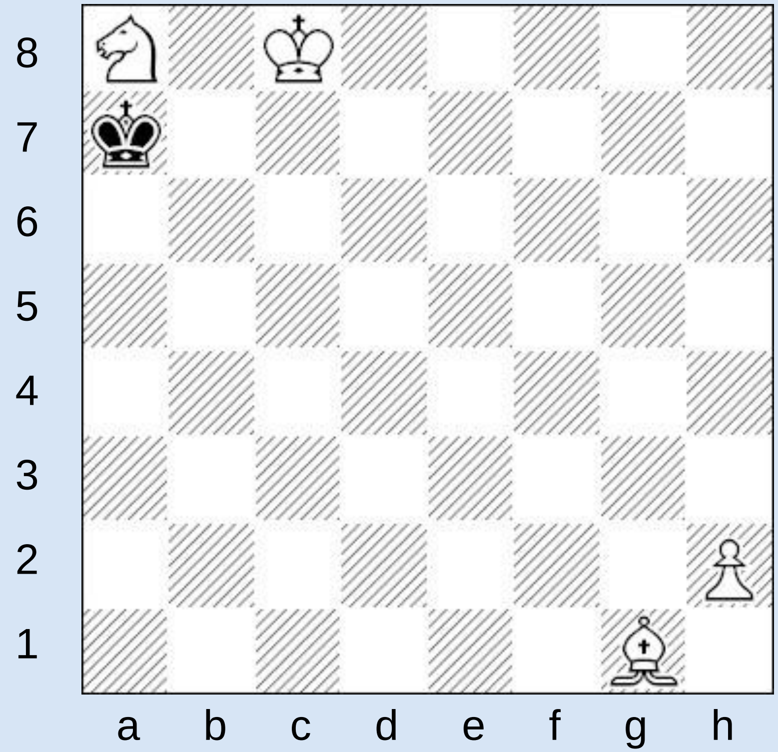
Black king was in check by white bishop!
How white made the last checking move?
Bishop is blocked => Some white piece must have moved to discover the check!

What did Black just play?
What was white's move before that?



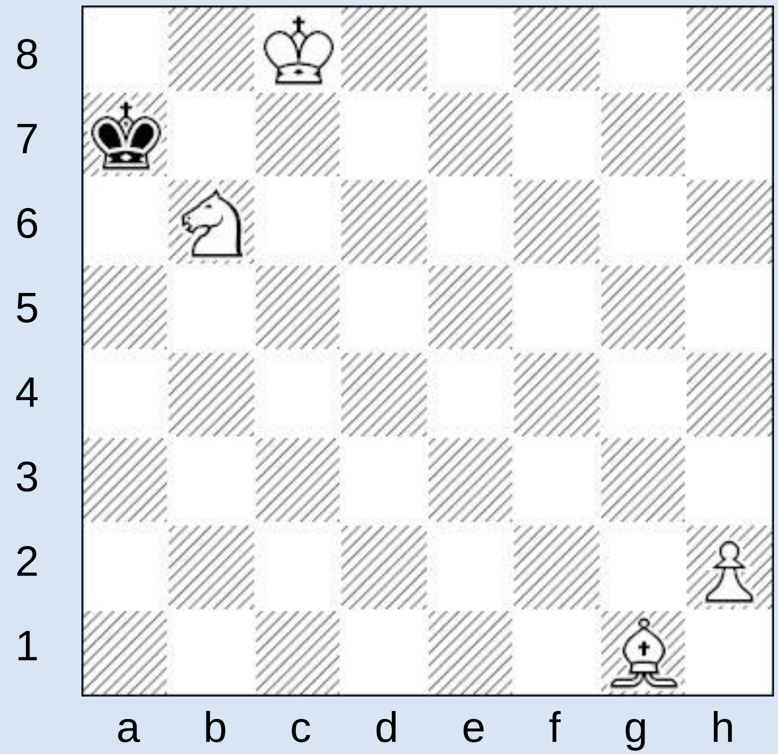
White had some piece on *a8* which black king captured by last move!

What did Black just play?
What was white's move before that?



Only white piece which can discover the check is white knight!

The position two moves before the given position

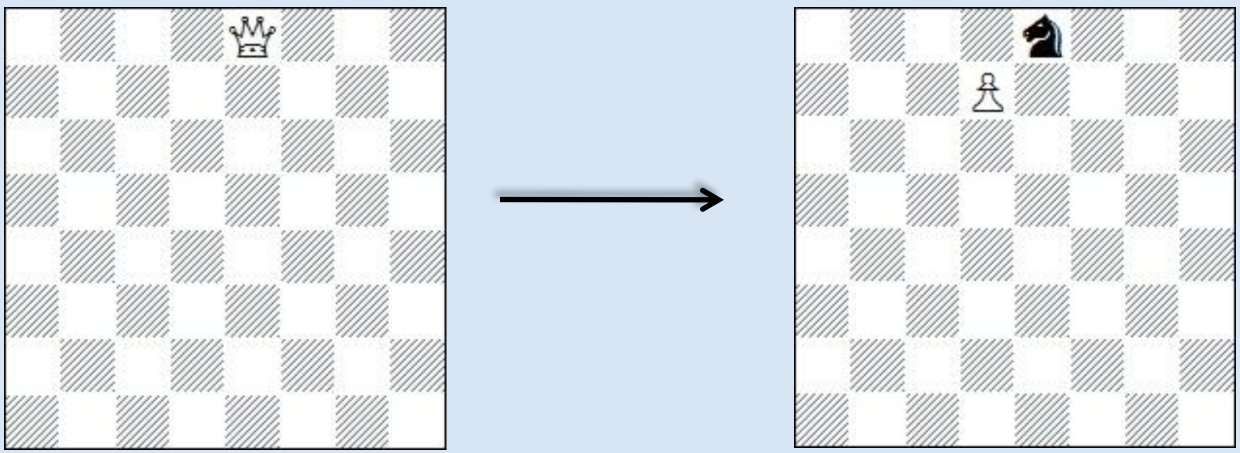


Retrograde chess analysis is a matter of deductive reasoning

Retrograde chess move

- Definition: "If in accordance with the laws of chess, position P_{n+1} arises from position P_n due to the move *m* of piece *p*, then the retrograde chess move m_1 of move *m* is the movement of piece *p* due to the position P_n arising from position P_{n+1}"

Different types of retrograde chess moves can have very different properties



Retrograde promotion with capturing

Basic formal system in Coq

M. Maliković. A formal system for automated reasoning about retrograde chess problems using Coq. *Proceedings of 19th Central European Conference on Information and Intelligent Systems*, 2008, pp. 465-475. Varaždin, Croatia

- Chess pieces as enumerated inductive type:

```
Inductive pieces : Set := P | B | R | Q | N | K | p | b | r | q | n | k | O | v.
```

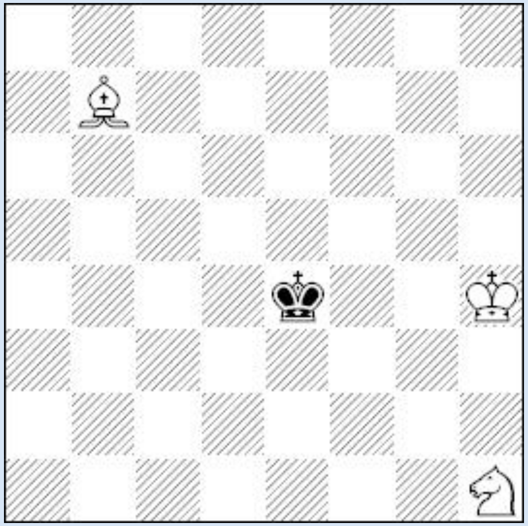
Position

```
Parameter position : nat -> list (list pieces).

Hypothesis H_position : position on =
(v :: nil) ::
(v :: k :: O :: K :: O :: O :: O :: O :: O :: nil) ::
(v :: O :: O :: O :: O :: Q :: O :: O :: O :: nil) ::
(v :: O :: O :: O :: O :: O :: O :: O :: O :: nil) ::
(v :: O :: O :: B :: O :: P :: O :: O :: O :: nil) ::
(v :: O :: O :: O :: P :: O :: O :: O :: O :: nil) ::
(v :: O :: O :: O :: O :: O :: O :: O :: P :: nil) ::
(v :: O :: O :: O :: O :: O :: O :: O :: O :: nil) ::
(v :: O :: O :: O :: O :: O :: O :: O :: B :: nil) ::
nil.
```

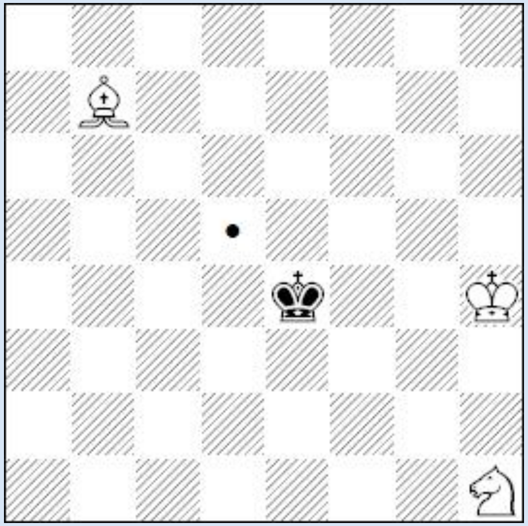
Functions for computing check positions

- Recursive for the bishop, rook and queen
- Non-recursive for knights and pawns



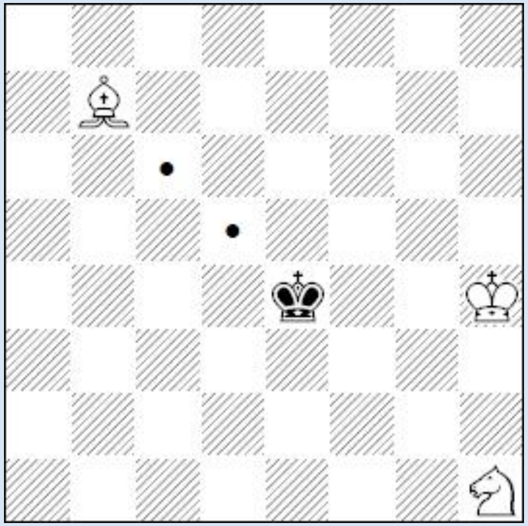
Recursive functions check the content of squares,
starting from the closest square of the king in all eight directions

Functions for computing check positions



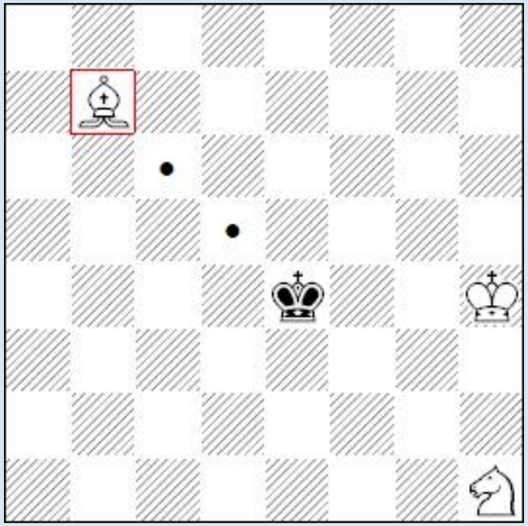
Square is empty -> Check next square!

Functions for computing check positions



Square is empty -> Check next square!

Functions for computing check positions



Square is engaged with opponent's bishop =>
King is in check in direction *left-up*

Example in *Coq*:
Function for direction *left-up*

```
Fixpoint check_lu_k (xkb ykb : nat) (pos : list (list pieces)) {struct xkb} : Prop :=
match xkb with
  S xkb' => match ykb with
                  S ykb' => match nth ykb' (nth xkb' pos nil) v with
                                   O => check_lu_k xkb' ykb' pos
                                  | Q => True
                                  | B => True
                                  | _ => False
                                 end
                 | _ => False
                end
  | _ => False
end.
```

Functions for computing new position after a retrograde move

```
position on =
(v :: nil) ::
(v :: k :: O :: K :: O :: O :: O :: O :: O :: nil) ::
(v :: O :: O :: O :: O :: Q :: O :: O :: O :: nil) ::
(v :: O :: O :: O :: O :: O :: O :: O :: O :: nil) ::
(v :: O :: O :: B :: O :: P :: O :: O :: O :: nil) ::
(v :: O :: O :: O :: P :: O :: O :: O :: O :: nil) ::
(v :: O :: O :: O :: O :: O :: O :: O :: P :: nil) ::
(v :: O :: O :: O :: O :: O :: O :: O :: O :: nil) ::
(v :: O :: O :: O :: O :: O :: O :: O :: B :: nil) ::
nil.
```

→

```
position (S on) =
(v :: nil) ::
(v :: N :: O :: K :: O :: O :: O :: O :: O :: nil) ::
(v :: k :: O :: O :: O :: Q :: O :: O :: O :: nil) ::
(v :: O :: O :: O :: O :: O :: O :: O :: O :: nil) ::
(v :: O :: O :: B :: O :: P :: O :: O :: O :: nil) ::
(v :: O :: O :: O :: P :: O :: O :: O :: O :: nil) ::
(v :: O :: O :: O :: O :: O :: O :: O :: P :: nil) ::
(v :: O :: O :: O :: O :: O :: O :: O :: O :: nil) ::
(v :: O :: O :: O :: O :: O :: O :: O :: B :: nil) ::
nil.
```

Retrograde move

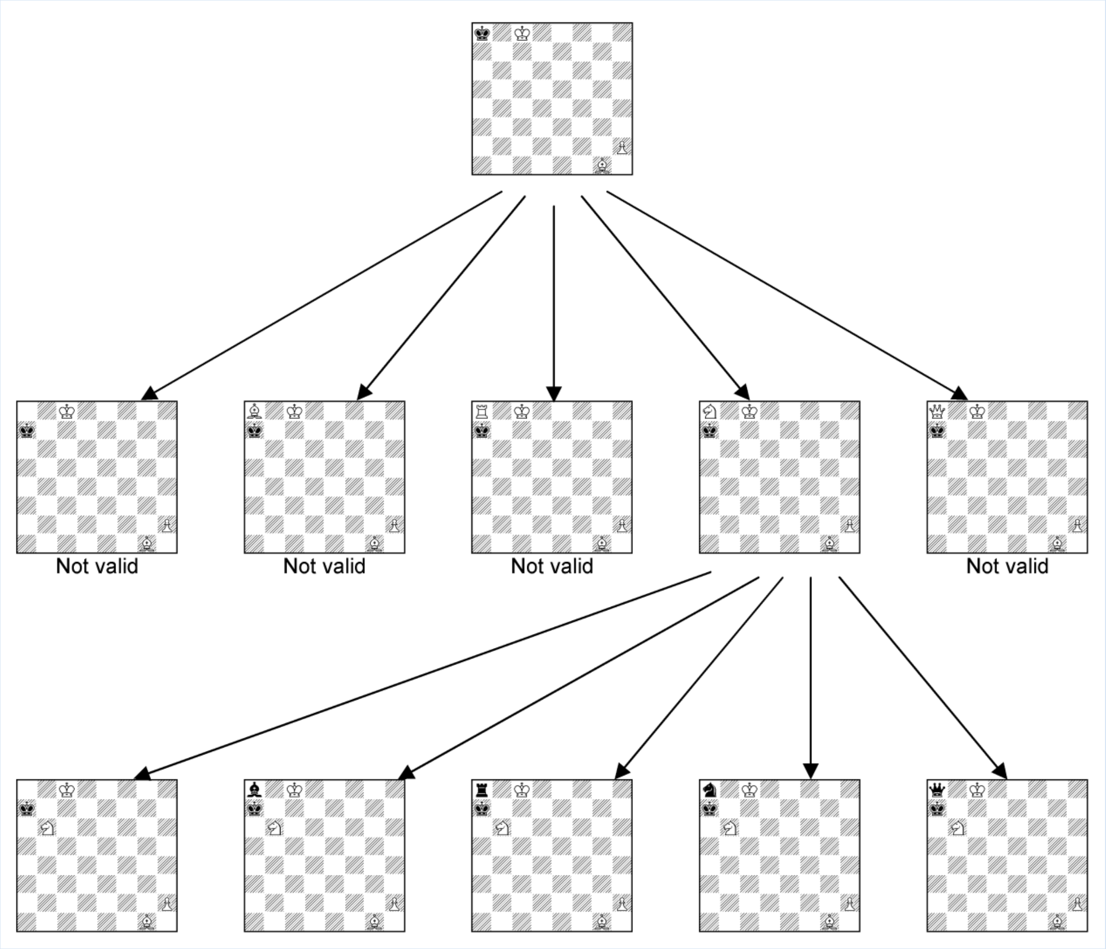
- Type of retrograde move:

Parameter move : nat -> pieces -> nat -> nat -> nat -> nat -> pieces -> type_of_move.

- Sequences of retrograde moves are stored on the list of moves:

H_list_moves : list_moves 2 =
moved 0 k 1 1 2 1 N standard_move ::
moved 1 A 1 1 3 2 b standard_move :: nil

Generating retrograde moves



Generating retrograde moves

M. Maliković; M. Čubrilo. What Were the Last Moves? *International Review on Computers and Software (IRECOS)*, Vol. 5, No. 1, 2010, pp. 59-70.

- Using Coq's tactics and *Ltac* language we create only one *Ltac* function *One_Move*
- We build up tree of retrograde chess moves and positions
- Every position as well as sequence of moves is stored in separate subgoal
- Thus, we use Coq's proof tree as tree of states and actions
- Our system is automated:
  - One_Move;One_Move;...
  - If all subgoals become proven => position is not legal
  - If only one subgoal remain unproven => it is a solution

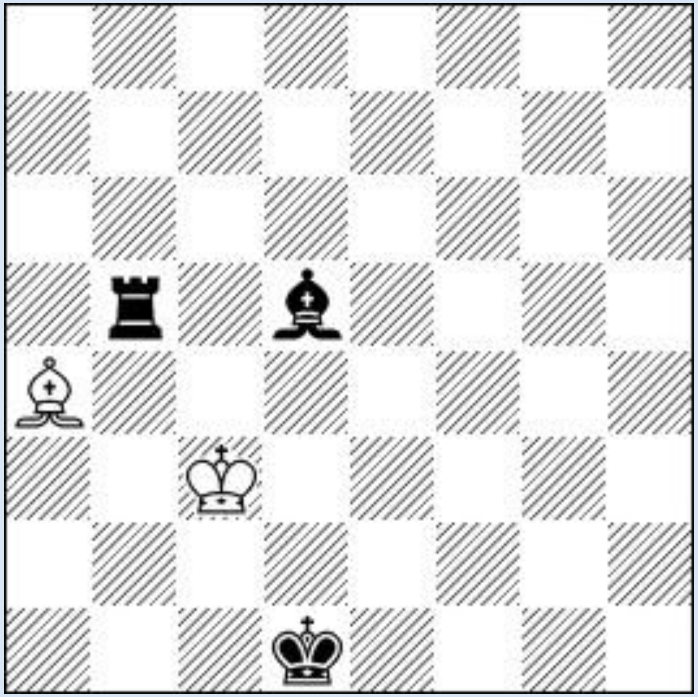
Generating retrograde moves

- with heuristic solutions obtained by observation -

- Each retrograde move must satisfy a number of conditions
- For example, the function *One_Move* check:
  - Is the player whose turn it is in check?
  - Is the player whose turn it isn't in check?
  - Determining eventually forced moves
    - e.g. because of the check positions by the pawn or knight
  - Eliminating the moves of the rook and king if retrograde castling has been already played by these pieces
  - So-called "imaginary check positions"
  - ...

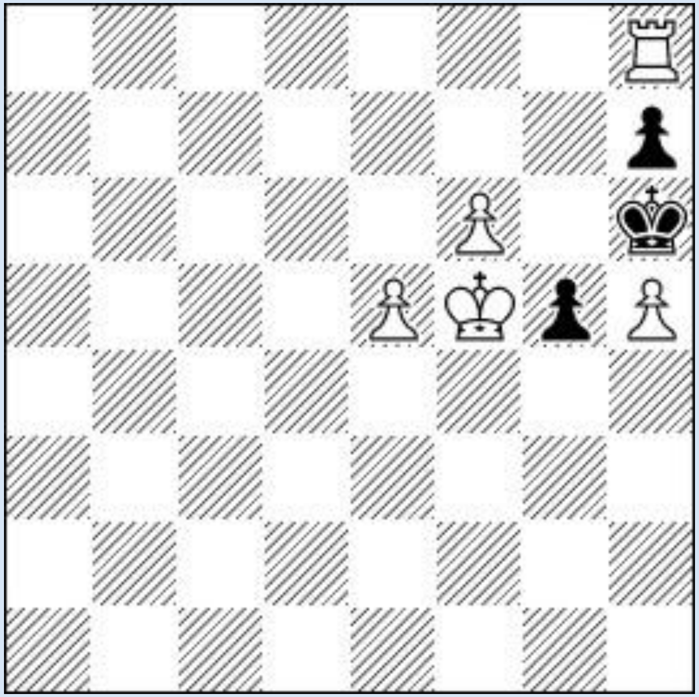
Purposes of RCA

What were the last 3 moves



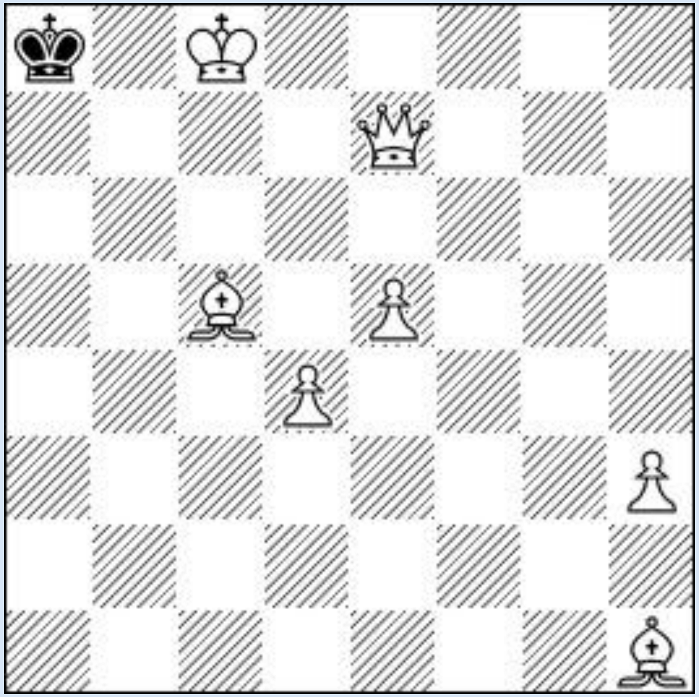
Mate in 2 moves!

Or: Is white's *en passant* capture legal?



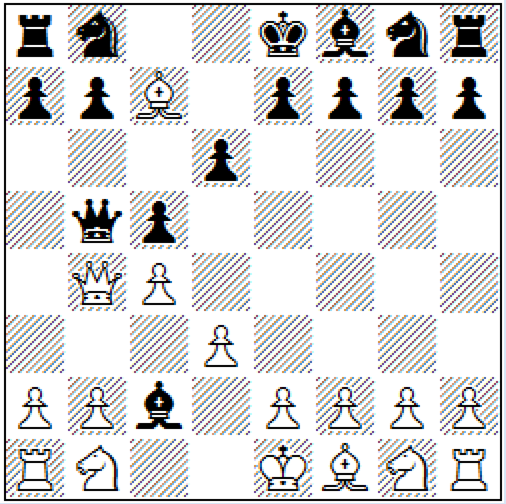
Can black castle?

Is position legal?



Shortest proof games

- SPG's serve to establish the legality of a position in chess problems by searching for the shortest sequence of moves that lead from initial to given chess position



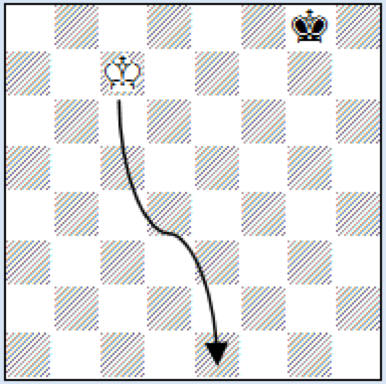
Formal bases of system for solving SPGs using Coq

M. Maliković; M. Čubrilo. Solving Shortest Proof Games by Generating Trajectories using Coq Proof Management System. *Proceedings of 21st Central European Conference on Information and Intelligent Systems*, 2010, pp. 11-18. Varaždin, Croatia

M. Maliković; M. Čubrilo. Formal System for Searching for the Shortest Proof Games using Coq. *International Review on Computers and Software (IRECOS)*, Vol. 5, No. 6, 2010, pp. 746-756.

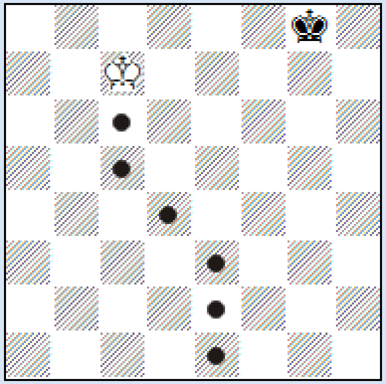
For given chess position we created recursive functions in Coq for generating:

- Trajectories - planing paths between two squares which certain pieces might follow to reach the target square



For given chess position we created recursive functions in Coq for generating:

- Shortest trajectories

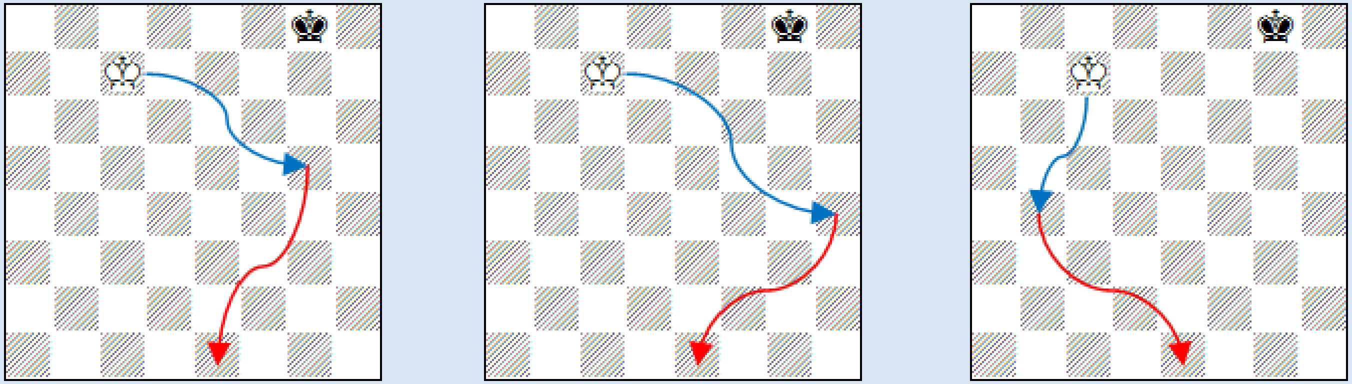


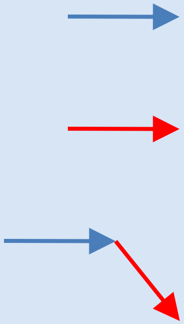
For given chess position we created recursive functions in Coq for generating:

- Admissible trajectories of some degree - defined inductively:
  - An admissible trajectory of degree 1 is a shortest trajectory
  - An admissible trajectory of degree $k > 1$ is a concatenation of an admissible trajectory of degree k-1 and one shortest trajectory

For given chess position we created recursive functions in Coq for generating:

- Admissible trajectories





Admissible trajectory of degree k-1

Shortest trajectory

Admissible trajectory of degree k

For given chess position we created recursive functions in Coq for generating:

- Circular trajectories - trajectory that’s starting and end square coincide
- Circular trajectories can be generated as admissible trajectories of some degree with same starting and end square

Thank you!