Solving Shortest Proof Games by Generating Trajectories using Coq Proof Management System

Marko Maliković
The Faculty of Humanities and Social Sciences
University of Rijeka
Omladinska 14, 51000 Rijeka, Croatia
marko.malikovic@ffri.hr

Mirko Čubrilo
Faculty of Organization and Informatics
University of Zagreb
Pavlinska 2, 42000 Varaždin, Croatia
mirko.cubrilo@foi.hr

Abstract. *In this paper we focus on Shortest Proof Games (SPG) as one important genre of retrograde chess analysis. SPG's serve to establish the legality of a position in given chess problems by searching for the shortest sequence of moves that lead to the initial chess position. First we give an overview of existing computer programs for solving SPG's, but due to the absence of any research papers on the topic, we provide informal descriptions obtained by the authors via e-mail and from partial information from programs' Web sites. In the second part of the paper we propose some systematic ideas for the establishment of a formal system for solving SPG's by using Coq - a formal proof management system. Our approach is based on the shortest trajectories (shortest planning paths which certain pieces might follow from initial square to achieve the target square), admissible trajectories (trajectories longer than the shortest trajectory) and bundles of trajectories. We show how these forms can be recursively generated using Coq and how they can be used in order to solve an SPG.*

Keywords. Retrograde chess analysis, Shortest proof games, Trajectories, Coq

1 Introduction

Retrograde chess analysis (RCA) is a method to determine which moves were played leading up to a given chess position. There are several main types of retrograde chess problems which can be classified on the basis of different criteria. In this paper we focus on Shortest Proof Games (SPG).[1] SPG can serve to establish the legality of a position. The problem of SPGs is the search for the shortest sequence of moves leading from the given to the initial chess position. Here we give a general definition of retrograde chess moves. Due to the inability to axiomatization of retrograde chess, definition is indirect and based on a definition of moves in "standard" chess: "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'* of move *m* is the *movement* of piece *p* due to the position P_n arising from position P_{n+1}." If we do not restrict ourselves only to chess then we can say that the retrograde analysis may be applied to any system that can display different states and in which a set of rules defines the change of one state to another. The purpose of such retrograde analysis can be: avoidance of undesirable final states or determination of action sequences which lead to some of the desirable final states.

For a build-up model for solving SPGs we use Coq, a computer tool for verifying theorem proofs in higher-order logic, whose complete theory and possibility of practical applications is given in [1] and [18]. The underlying theory of the Coq is the Calculus of Inductive Constructions [5], a formalism that combines logic from the point of view of λ-calculus and typing. OCaml is the implementation language for Coq [9]. Concerning a proposition that one wants to prove, the Coq system proposes tactics, to construct a proof, using elements taken from a context, namely, declarations, definitions, axioms, hypotheses, lemmas, and already proven theorems. In addition, the Coq system provides the language Ltac of operators called tacticals which make it possible to combine tactics and, in such a way, to build more complex tactics that

[1] For other types see [8] or [13].

can be defined as integral function called Ltac functions [2], [1, 61], [18, 213].

Why use Coq for solving retrograde chess problems? We offer a short answer by quoting from [20, 5]: “Even most problem composers feel that the basis of retrograde chess analysis is rather mathematical logic than the game of chess.” Apart from [10], [11] and [12], so far the Coq system has not been applied in the field of chess. But, in these three last publications Coq is applied to different types of retrograde chess problems than the problems we deal with in this paper, and this was reason for applying different methodology for solving these problems here. Specifically, the publications mentioned deal with problems of proving invalidity of any given position, such as determining if castling is disallowed or an *en passant* capture is possible or determining a certain number of moves played leading up to a given position, but under the premise of the lengths of the entire paths of the figures from initial to the given position being irrelevant.

There are several computer programs for solving SPGs but there is no research paper in this field. Therefore, below we offer descriptions of existing programs given by the authors (by E-mail) and from programs’ web sites. ***Retractor*** [7] is an old program developed in 1991 in the Department of Computer Science at Stanford University, California. Retractor uses a simple, classical backtracking search. All possible retromoves are generated at each node, with backtracking when a position is hit that can be proven to be either illegal, or previously reached. If the search reaches an implied given maximum depth without hitting a position it can prove illegal, then that branch is counted as a solution. But this does not guarantee that the solution is correct, only that the preprogrammed ruleset isn't able to prove the position illegal. ***Natch*** [19] identifies pieces that have never moved (pawns first, and then pieces blocked by pawn). On a board where the squares identified in the previous phases cannot be used, or crossed, it builds many tables, where the minimum number of moves is given from one square to any other square, for all piece types. After this, it searches for all combinations of pieces. There are two constraints that must be respected. The number of moves must not exceed the total number of moves. The second constraint is only for pawns. When they are not on the same column, there must be enough pieces missing in the opposite camp. Then Natch tries to order moves. When cycle in the moves order is detected, the position is eliminated. “Natch isn't able to verify every SPG problem. There are many kinds of positions where Natch needs days, if not weeks, to find a solution.” [19]. ***Euclide*** [3] is a program divided in four major parts. The first one, called the preliminary analysis, tries to make obvious deductions directly from the position. Without going into details, Euclide counts the required moves for each piece to reach their possible destinations and then eliminates impossibilities. In the second part Euclide uses so-called strategies. For each of the initial 32 pieces it is determined: the final square of the piece, whether this piece was captured or not, the promotion square if any and the promotion piece, the order of the various captures made, the castling side if applicable. One subset of all these possible choices for each piece forms a strategy. All possible strategies are built by going through a large number of permutations. The possible permutations are built from data provided by the preliminary analysis, without further analysis. If Euclide failed to make many deductions in the first part, the total number of possible strategies is immensely huge, hence Euclide will run “forever”. The third part, in order to eliminate strategies, consists of a partial analysis of move dependencies. For each strategy built in the previous step, Euclide performs further counting deductions, exactly like in the first part but this time, for a given strategy, each piece has a known final square, known captures, etc. This can eliminate immediately a large number of possible strategies. Finally, the fourth part simply plays, from the initial position, moves until solutions are found or the move tree is exhausted. For each strategy, Euclide plays all possible games. Again, this can be very time consuming. The computations of the third part are carefully used to truncate huge branches of moves. Euclide fails to make obvious deductions that a human is able to make, hence the program sometimes considers a huge number of strategies that are obviously impossible. When there are many missing pieces, Euclide has much trouble finding where the captures have occurred and again considers a large number of strategies that are obviously not possible.

2 Bases of RCA using Coq

Here we load the *List* module [6, 51], since our model is going to use lists:

```
Require Import List.
```

The coordinates of the squares according to the orientation of chessboard are shown in Fig. 1. As we see, the standard labels of rows and columns of the chessboard are mapped into the natural numbers as follows: *a→1*, ..., *h→8*, *1→8*, ..., *8→1*. The set of squares on the chessboard can be defined in Coq as record type [1, 145] with two functions *column* and *row* of type *nat*:

```
Record chessboard : Set := square {column : nat; row : nat}.
```

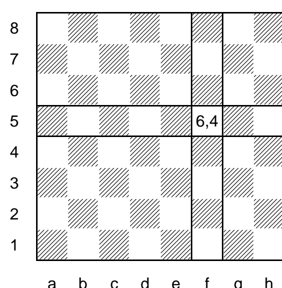

Figure 1. The coordinates of the squares

Set of pieces in order: king, knight, rook, bishop, queen and pawn, as well as pieces' colors (black and white) we introduce as enumerated inductive types without recursion [1, 137]:

```
Inductive piece : Set := k | n | r | b | q | p.
Inductive color : Set := W | B.
```

3 Trajectories in Coq

The Language of Trajectories is developed in Linguistic Geometry [17], as the lowest level language of the Hierarchy of Languages [14]. In accordance with the domain of this paper, we can informally describe trajectories as planning paths between two squares which certain pieces might follow to achieve the target square. We first want to consider *shortest* trajectories for an piece P of color C (abbreviation P_C) with the beginning at square (x_0,y_0) and the end at square (x_n,y_n). Trajectories will be defined as predicate whose attributes are: P, C and a list of squares which represent path. The set of shortest trajectories is a subset of set of *admissible* trajectories of some degree which are formally defined in [17, 51]. We define admissible trajectories as follows (in given definition is with $t_P((x_m,y_m),(x_n,y_n),l)$ assigned set of trajectories of piece P from (x_m,y_m) as the starting square and (x_n,y_n) as end square and of length l): "An admissible trajectory of degree 1 is a shortest trajectory. An admissible trajectory of degree k (k is an integer, k>1) is a trajectory $t \in t_P((x_0,y_0),(x_l,y_l),l)$ if there is a square (x_i,y_i) at chessboard such that t is a concatenation of an admissible trajectory of degree k-1 from $t_P((x_0,y_0),(x_i,y_i),l_1)$ and a shortest trajectory $t_P((x_i,y_i),(x_l,y_l),l_2)$, $l_1+l_2=l$."

Admissible trajectories are defined inductively and therefore, the first idea that arises is to define them in Coq as inductive type with recursion [1, 160]. But, for the starting and end square related to some piece there is generally more than one trajectory and we can't theoretically establish a whole system without differentiation of such trajectories. Therefore we just declare admissible trajectories of some degree (type *nat*) as predicate:

```
Parameter At : nat -> piece -> color -> list chessboard -> Prop.
```

Bundle of trajectories of some degree and for some piece, in accordance with [17, 50], is a set of trajectories which all have the same starting and end square:

```
Parameter bt :
nat -> piece -> color -> nat -> nat -> nat -> nat -> list (list chessboard) -> Prop.
```

4 Starting state of the system

Let us consider SPG problem given at Fig. 2.

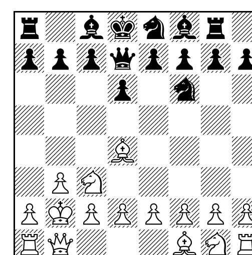

Figure 2. Shortest Proof Game in 8.0

In SPG problems every piece has to reach its own end square. Here one problem appears: Do we know what square is an end square for every piece? Generally, in all of SPG problems we only know what the end square is for kings because the other figures on the board can be promoted (except for pawns, of course, but pawns can change a column in which they are). But, in the problem given in Fig. 2 we know what the end square for more pieces is. This is because problem 2 is a simplified SPG problem for several reasons. First, all pieces from the initial chess position are still on the chessboard because no piece was captured and no piece was promoted in the previous moves. This means that we know what is the end square also for queens, bishops, and pawns. There are two squares that each of the knights could get to and this also applies to rooks. Moreover, some knight or rook can, in a given position, stay at the end square of

another knight or rook of the same color and this square can be just his temporarily square. Our current intention isn't to resolve such issues. Due to this we just discover to the reader that the knight at *f6* come from square *g8* and knight at *c3* from *b1*.[2] To setup starting state of problem shown in Fig. 2, we declare predicate *ON* where term *ON P C* x_0 y_0 x_n x_n means that peace P_C stays at square (x_0,y_0) and that square (x_n,y_n) is its target square. Then we just list all the instances of predicate *ON* for all pieces on the chessboard:

```
Parameter ON : piece -> color -> nat -> nat -> nat -> nat -> Prop.
Variable ON1 : ON k W 2 7 5 8.
Variable ON2 : ON q W 2 8 4 8.
...
Variable ON32 : ON p B 8 2 8 2.
```

5 Distances at chessboard

In order to construct the shortest trajectories, in accordance with [17, 52] and [16, 95], we define function *MAP* which gives the number of moves necessary for piece *P* staying at (x_0,y_0) to reach square (x_n,y_n) along the shortest path. For each kind of piece a different function exists, but these functions don't differ for the same kind of pieces of different colors apart for pawns. Let us first describe function *MAP* informally. For each kind of piece *P* we specify table 15×15 with number 0 on the central square of the table. The remaining squares are filled with the numbers equal to the number of moves necessary for piece *P* to reach the given square from the central square along the shortest path or, if the square is not reachable then with *2×the number of points in chessboard=128*. To define such tables in Coq for various kinds of pieces we use function *MAP_start* with five initial lists of lists over natural numbers which belong to kings, knights, rooks, bishops and queens (we show here only part of function which belongs to kings):

```
Definition MAP_start (P : piece) := match P with
| k => (7 :: 7 :: 7 :: 7 :: 7 :: 7 :: 7 :: 7 :: 7 :: 7 :: 7 :: 7 :: 7 :: 7 :: 7 :: nil) ::
     (7 :: 6 :: 6 :: 6 :: 6 :: 6 :: 6 :: 6 :: 6 :: 6 :: 6 :: 6 :: 6 :: 6 :: 7 :: nil) ::
     (7 :: 6 :: 5 :: 5 :: 5 :: 5 :: 5 :: 5 :: 5 :: 5 :: 5 :: 5 :: 5 :: 6 :: 7 :: nil) ::
     (7 :: 6 :: 5 :: 4 :: 4 :: 4 :: 4 :: 4 :: 4 :: 4 :: 4 :: 4 :: 5 :: 6 :: 7 :: nil) ::
     (7 :: 6 :: 5 :: 4 :: 3 :: 3 :: 3 :: 3 :: 3 :: 3 :: 3 :: 4 :: 5 :: 6 :: 7 :: nil) ::
     (7 :: 6 :: 5 :: 4 :: 3 :: 2 :: 2 :: 2 :: 2 :: 2 :: 3 :: 4 :: 5 :: 6 :: 7 :: nil) ::
     (7 :: 6 :: 5 :: 4 :: 3 :: 2 :: 1 :: 1 :: 1 :: 2 :: 3 :: 4 :: 5 :: 6 :: 7 :: nil) ::
     (7 :: 6 :: 5 :: 4 :: 3 :: 2 :: 1 :: 0 :: 1 :: 2 :: 3 :: 4 :: 5 :: 6 :: 7 :: nil) ::
     (7 :: 6 :: 5 :: 4 :: 3 :: 2 :: 1 :: 1 :: 1 :: 2 :: 3 :: 4 :: 5 :: 6 :: 7 :: nil) ::
     (7 :: 6 :: 5 :: 4 :: 3 :: 2 :: 2 :: 2 :: 2 :: 2 :: 3 :: 4 :: 5 :: 6 :: 7 :: nil) ::
     (7 :: 6 :: 5 :: 4 :: 3 :: 3 :: 3 :: 3 :: 3 :: 3 :: 3 :: 4 :: 5 :: 6 :: 7 :: nil) ::
     (7 :: 6 :: 5 :: 4 :: 4 :: 4 :: 4 :: 4 :: 4 :: 4 :: 4 :: 4 :: 5 :: 6 :: 7 :: nil) ::
     (7 :: 6 :: 5 :: 5 :: 5 :: 5 :: 5 :: 5 :: 5 :: 5 :: 5 :: 5 :: 5 :: 6 :: 7 :: nil) ::
     (7 :: 6 :: 6 :: 6 :: 6 :: 6 :: 6 :: 6 :: 6 :: 6 :: 6 :: 6 :: 6 :: 6 :: 7 :: nil) ::
     (7 :: 7 :: 7 :: 7 :: 7 :: 7 :: 7 :: 7 :: 7 :: 7 :: 7 :: 7 :: 7 :: 7 :: 7 :: nil) ::
     nil...
```

[2] It is clear that all rooks are at their side and no castling is done.

The required results of function *MAP* for the king standing at *f6* is represented by a 8×8 sub-table (internal shaded table) in which number 0 is located in the square where observed piece stands.[3]

To define function *MAP* over function *MAP_start* we first define several auxiliary recursive functions [4, 27], [18, 38], [1, 164].

```
Fixpoint beginning_of_linear_list (x0 : nat) (l : list nat) {struct l} : list nat := match l with
nil => nil | l'::l1 => match 16-x0 with 0 => nil | S x0' => l'::beginning_of_linear_list
(x0+1) l1 end end.

Fixpoint rest_of_linear_list (x0 : nat) (l : list nat) {struct l} : list nat := match l with nil =>
nil | l'::l1 => match 8-x0 with 0 => l | S x0' => rest_of_linear_list (x0+1) l1 end end.

Fixpoint beginning_of_2_dim_list (y0 : nat) (l : list (list nat)) {struct l} : list (list nat) :=
match l with nil => nil | l'::l1 => match 16-y0 with 0 => nil | S y0' =>
l'::beginning_of_2_dim_list (y0+1) l1 end end.

Fixpoint rest_of_2_dim_list (y0 : nat) (l : list (list nat)) {struct l} : list (list nat) := match l
with nil => nil | l' :: l1 => match 8-y0 with 0 => l | S y0' => rest_of_2_dim_list (y0+1) l1
end end.

Fixpoint MAP_temp (x0 y0 : nat) (l : list (list nat)) {struct l} : list (list nat) := match l with
  nil => nil | l' :: l1 => app (0 :: nil) (rest_of_linear_list x0 (beginning_of_linear_list x0 l'))
:: MAP_temp x0 y0 l1 end.
```

The attributes of function *MAP* are the coordinates of the starting square of piece *P* and starting 15×15 table which belong to pieces of the type *P*:

```
Definition MAP x0 y0 l := app ((0 :: nil) :: nil) (MAP_temp x0 y0 (rest_of_2_dim_list y0
(beginning_of_2_dim_list y0 l))).
```

6 Generating shortest trajectories

In this section we show how in Coq all shortest trajectories for piece *P* from (x_0,y_0) to (x_n,y_n) can be iteratively generated, mostly in accordance with the steps given in [17, 196]. For example, in the position shown at Fig. 3 we have to find all the shortest trajectories for the white king from *c7=(3,2)* to his target square *e1=(5,8)*.

Figure 3. By which shortest paths can white king from *c7=(3,2)* come to his target square *e1=(5,8)*?

First we compute the shortest distance l_0 between the starting and end square by computing *MAP* for given starting square (x_0,y_0)

[3] Now we can see that asymmetry is not only the reason for problem of creating lists for pawns. The situation is more complicated also because of the following: impossibility of standing of pawns in 1th and 8th row, because of move two squares backward only from 4th row (for white) or from 5th row (for black) and because of diagonally moves (retrograde capturing).

and given piece. In our example we have to compute *MAP 3 2 (MAP_start k)*. The value in the resulting list which corresponds to the end square is the shortest distance l_0 (in our example l_0=6). Further, we compute *MAP* for end square (x_n,y_n) and same piece (in our example *MAP 5 8 (MAP_start k)*). Now, we find the set of squares such as *MAP* x_0 y_0 *(MAP_start k)+MAP* x_n y_n *(MAP_start k)=*l_0. To find this set which we denote by *SUM*, we define function *Sum* which is the sum of two 2-dimensional matrices:

```
Fixpoint Sum_linear (M1 M2 : list nat) {struct M1} : list nat := match M1 with nil => nil | M :: M1' => match M2 with nil => nil | N :: M2' => nth 0 M1 0 + nth 0 M2 0 :: Sum_linear M1' M2' end end.
```

```
Fixpoint Sum (M1 M2 : list (list nat)) {struct M1} : list (list nat) := match M1 with nil => nil | M :: M1' => match M2 with nil => nil | N :: M2' => Sum_linear M N :: Sum M1' M2' end end.
```

After computing *SUM* we get:

```
Sum (MAP 3 2 (MAP_start k)) (MAP 5 8 MAP_start k)) =
(0 :: nil) ::
(0 :: 9 :: 8 :: 8 :: 8 :: 9 :: 10 :: 11 :: 12 :: nil) ::
(0 :: 8 :: 7 :: 6 :: 7 :: 8 :: 9 :: 10 :: 11 :: nil) ::
(0 :: 7 :: 6 :: 6 :: 6 :: 7 :: 8 :: 9 :: 10 :: nil) ::
(0 :: 6 :: 6 :: 6 :: 6 :: 6 :: 7 :: 8 :: 9 :: nil) ::
(0 :: 7 :: 6 :: 6 :: 6 :: 6 :: 6 :: 7 :: 8 :: nil) ::
(0 :: 8 :: 7 :: 6 :: 6 :: 6 :: 6 :: 6 :: 8 :: nil) ::
(0 :: 9 :: 8 :: 7 :: 6 :: 6 :: 6 :: 7 :: 8 :: nil) ::
(0 :: 10 :: 9 :: 8 :: 7 :: 6 :: 7 :: 8 :: 9 :: nil) :: nil
```

All the first possible moves from the starting square are from the intersection of three sets:

- *SUM*
- $ST_1(x_0,y_0)$*={square x y | nth x (nth y (MAP* x_0 y_0 *(MAP_start k)) nil) 0=1}*
- $ST_{l_0-l+1}(x_{l_0-l+1},y_{l_0-l+1})$*={square x y | nth x (nth y (MAP* x_{l0-l+1} y_{l0-l+1} *(MAP_start k)) nil) 0=l0-l+1} for l=l0*

For example, in the case shown in Fig. 3, *b6*, *c6* and *d6* are three end squares of all the possible first moves from *c7* related to *e1*. These squares are now *new starting squares* for the *next step* of generation. The *SUM* is the same for all steps of the generation, while ST_1 and ST_{l_0-l+1} iterative changed in the way that ST_1 in every next step as arguments takes coordinates of the new square, while in ST_{l_0-l+1} in every next step value l decreases by one. In such a way, after l_0 steps we get all possible shortest trajectories.

To iteratively generate bundles of shortest trajectories for all pieces on the chessboard we create very complex auxiliary function *GM*, function *GBT1* and main recursive function *GBT*:

```
Fixpoint GM (i : nat) (j : nat) (n : nat) (P : piece) (C : color) (t0 : list chessboard) (xl yl : nat) {struct n} : list (list chessboard) := match n with S n' => match j with S j' => match i with S i' => if eq_nat (nth i (nth j (MAP (column (last t0 (square 0 0))) (row (last t0 (square 0 0))) (MAP_start P)) nil) 0) 1 then if eq_nat (nth i (nth j (MAP (column (nth 0 t0 (square 0 0))) (row (nth 0 t0 (square 0 0))) (MAP_start P)) nil) 0) (length t0) then if eq_nat (nth i (nth j ((Sum (MAP (column (nth 0 t0 (square 0 0))) (row (nth 0 t0 (square 0 0))) (MAP_start P)) (MAP xl yl (MAP_start P)))) nil) 0) (nth xl (nth yl (MAP (column (nth 0 t0 (square 0 0))) (row (nth 0 t0 (square 0 0))) (MAP_start P)) nil) 0) then if eq_nat i xl then if eq_nat j yl then (app t0 (square i j :: nil)) :: nil else app (GM i' j n' P C t0 xl yl) ((app t0 (square i j :: nil)) :: nil) else app (GM i' j n' P C t0 xl yl) ((app t0 (square i j :: nil)) :: nil) else GM i' j n' P C t0 xl yl else GM i' j n' P C t0 xl yl else GM i' j n' P C t0 xl yl | _ => GM 8 j' n' P C t0 xl yl end | _ => nil end | _ => nil end.
```

```
Fixpoint GBT1 (P : piece) (C : color) (bt : list (list chessboard)) (xl yl : nat) {struct bt} := match bt with nil => nil | bt0 :: bt' => app (GM 8 8 128 P C bt0 xl yl) (GBT1 P C bt' xl yl) end.
```

```
Fixpoint GBT (P : piece) (C : color) (bt : list (list chessboard)) (xl yl n : nat) {struct n} := match n with | O => bt | S n' => GBT1 P C (GBT P C bt xl yl n') xl yl end.
```

Now we introduce axiom *AGBT* which will allow us to compute a bundle of shortest trajectories as result of the function *GBT*:

```
Axiom AGBT : forall x0 y0 xl yl k : nat, forall P : piece, forall C : color, ON P C x0 y0 xl yl -> bt k P C x0 y0 xl yl (GBT P C ((square x0 y0 :: nil) :: nil) xl yl (nth xl (nth yl (MAP x0 y0 (MAP_start P)) nil) 0)).
```

We can create the Ltac function *LGBST* which will from hypotheses in the initial state of system generate bundles of shortest trajectories for all pieces for which trajectories exists:

```
Ltac LGBST := repeat match goal with [h : ON ?P ?C ?X0 ?Y0 ?XL ?YL |- _ ] => apply AGBT with (k:=1) in h;compute in h end.
```

The resulting bundles will appear in context as lists of lists of squares. For example, bundle of shortest trajectories of the white king in Problem 2 will look like this:

```
bt 1 k W 2 7 5 8 ((square 2 7 :: square 3 6 :: square 4 7 :: square 5 8 :: nil) :: (square 2 7 :: square 3 7 :: square 4 7 :: square 5 8 :: nil) :: (square 2 7 :: square 3 7 :: square 4 8 :: square 5 8 :: nil) :: (square 2 7 :: square 3 8 :: square 4 7 :: square 5 8 :: nil) :: (square 2 7 :: square 3 8 :: square 4 8 :: square 5 8 :: nil) :: nil)
```

To generate single trajectories from such bundles we need one new recursive function, one axiom and one Ltac function as follows:

```
Fixpoint FGT (P : piece) (C : color) (bt : list (list chessboard)) {struct bt} : Prop := match bt with nil => True | bt0 :: bt' => At 1 P C bt0 /\ (FGT P C bt') end.
```

```
Axiom AGT : forall k x0 y0 xl yl : nat, forall P : piece, forall C : color, forall bt_list : list (list chessboard), bt k P C x0 y0 xl yl bt_list -> FGT P C bt_list.
```

```
Ltac LAGT := repeat match goal with [h : _ |- _ ] => apply AGT in h;compute in h end; repeat match goal with h : (_ /\ _) |- _ ] => case h;clear h;intros end; repeat match goal with [ h : True |- _ ] => clear h end.
```

7 Obstacles

In RCA a piece *P* can't reach some square (blocked destination) if it is occupied by another element *P'* and can't cross the square (blocked beam) if it is occupied by another element *P'* although *P* and *P'* can belong to the same or the opponents side. Blocked destinations and blocked beams are called *obstacles*. In some position two kinds of obstacles can exist for pieces to reach a square: unmovable and movable. Unmovable obstacles are pieces which stay at their target square and can't move anymore. Such obstacles have to be bypassed.

In the context we get after applying the function *LAGT*, some trajectories can be of length 1. Those trajectories correspond to unmovable obstacles. We introduce type *At_block*, axiom *A_block* and Ltac function *L_block_end_square* by whose application we

can assign all trajectories of length 1 as unmovable obstacles:

```
Parameter At_block : piece -> color -> list chessboard -> Prop.

Axiom A_block : forall P : piece, forall C : color, forall x y : nat, At 1 P C (square x y :: nil) -> At_block P C (square x y :: nil).

Ltac L_block_end_square := repeat match goal with | [h : At ?k ?P ?C (square ?X ?Y :: nil) |- _] => apply A_block in h end.
```

Movable obstacles are pieces that can still move and other pieces can wait their move. In order to consider movable and unmovable obstacles we create a new important function: function *obstacle* whose result will indicate whether a piece obstructs another piece to reach the planned square. The term *obstacle* $x\ y\ x_1\ y_1\ x_2\ y_2$ will return the boolean value *true* if square (x,y) is the obstacle for the move of a piece from square (x_1,y_1) to square (x_2,y_2) and the value *false* otherwise. The function *obstacle* is based on a comparison of the coordinates of the considered squares which we will not show here.

8 Goal

To solve an SPG means to find the shortest sequence of alternating white and black moves leading from the initial to a given chess position. Generally, one move is given by the kind of piece, its color and the starting and end squares. For the purposes of this paper, in the declaration of a move we have to add the attribute of type *nat* which indicates the ordinal number of moves in SPG:

```
Parameter move : nat -> piece -> color -> chessboard -> chessboard -> Prop.
```

To establish the order of moves first we have to know whose move was the last. In problem 2 it is known that the last move was the 16th move and it was black's move. Finally, now we can designate a goal which we have to prove according to problem 2 and this goal claims that there is a list of moves of length 16:

```
Goal exists P : piece, exists x1 : nat, exists y1 : nat, exists x2 : nat, exists y2 : nat, move 16 P W (square x1 y1) (square x2 y2).
```

9 Impossible trajectories

All trajectories which contain a move with an unmovable obstacle have to be eliminated from consideration and we call these trajectories *impossible trajectories*. To find and eliminate impossible trajectories we create the recursive function *F_s_on_t*, axiom *A_block_t* and Ltac function *L_Block_t*. The term *F_s_on_t x y P C l* with the help of function *obstacle* gives as a result the boolean value of *true* if at the square *(x,y)* an unmovable obstacle for the trajectory *l* of the piece P_C is to be found. Also, axiom *A_block_t* claims that if some unmovable obstacle is blocking a trajectory then this trajectory corresponds to the blocked trajectory at its starting square while Ltac function *L_Block_t* provides the elimination of the trajectory if the conditions are satisfied:[4]

```
Fixpoint F_s_on_t (x y:nat) (P:piece) (C:color) (l:list chessboard) {struct l} : bool := match l with nil => false | l' :: l1 => match l1 with nil => false |_ => match obstacle x y (column l') (row l') (column (nth 0 l1 (square 0 0))) (row (nth 0 l1 (square 0 0))) with true => true | false => F_s_on_t x y P C l1 end end end.

Axiom A_block_t : forall P : piece, forall C : color, forall t : list chessboard, forall k x y : nat, At k P C t -> F_s_on_t x y P C t = true -> At_block P C (nth 0 t (square 0 0) :: nil).

Ltac L_Block_t := repeat match goal with [h1 : At 1 ?P1 ?C1 (square ?X1 ?Y1 :: ?t1), h2 : At_block ?P2 ?C2 (square ?X2 ?Y2 :: nil) |- _ ] => apply A_block_t with (P:=P1) (C:=C1) (t:=square X1 Y1 :: t1) (x:=X2) (y:=Y2) in h1; [compute in h1;try (match goal with [h3 : At 1 P1 C1 (square X1 Y1 :: ?t2) |- _ ] => clear h1 end) | tauto] end.
```

10 Generating SPGs

With the help of the shortest trajectories we can minimize the number of obvious moves in SPG. We can see that after the elimination of impossible trajectories from context of problem 2, the sum of number of moves in trajectories is equal to 16 provided that we take into account one trajectory for every piece. This is in accordance with the assumption of problem which is given as "SPG in 8.0". Now we have, in the correct order, to add up the moves from the remaining trajectories in context. If *At 1 P C (square x1 y1 :: square x2 y2 :: l :: nil)* is an trajectory of degree 1 for some piece P_C with first move from (x_1,y_1) to (x_2,y_2) and with *l* as rest of the path, then, if we want to add hypothesis *move i P C (square x1 y1) (square x2 y2)* to the context, the considered trajectory has to be reduced for its first square. This can be formally introduced in our system as the following axiom:

```
Axiom AGSPG : forall P : piece, forall C : color, forall x1 y1 x2 y2 i k : nat, forall l : list chessboard, At k P C (square x1 y1 :: square x2 y2 :: l) -> At k P C (square x2 y2 :: l) /\ move i P C (square x1 y1) (square x2 y2).
```

We now need to generate as many hypotheses about the possible first moves as there are possible moves for a player whose turn it is. For now, the moves have to satisfy the conditions not to skip any piece and not to arrive at a square occupied by another piece. So, we need the

[4] In addition, function *L_Block_t* does these steps: 1. If in context appears more than one trajectories with the same starting square and if one of them is blocked by a piece, then this trajectory can be cleared from the context (this step is iteratively repeated); 2. If after step 1 in context remain only one trajectory from a starting square and if this trajectory is blocked by an unmovable obstacle then this trajectory is designated as a blocked piece at its starting square; 3. Clearing double trajectories.

following axiom by which those moves that do not meet these conditions can be eliminated:

```
Axiom Move_Obstacle : forall P1 P2 : piece, forall C1 C2 : color, forall x y x1 y1 x2 y2 i k : nat, forall l : list chessboard, move i P1 C1 (square x1 y1) (square x2 y2) -> At k P2 C2 (square x y :: l) /\ obstacle x y x1 y1 x2 y2 = true -> False.
```

Each of the possible moves will be generated in a separate subgoal. From all of these subgoals we will generate new subgoals which will contain second moves which belong to the opponent. And so we continue to build a tree until we generate one subgoal which will contain the solution. For this purpose we create the following Ltac function *LGSPG* with the attributes *col* which indicates a players' color whose turn it is and attribute *i* which indicates what the ordinal number of moves is:[5]

```
Ltac LGSPG col I := assert (H_goal : exists P : piece, exists x1 : nat, exists y1 : nat, exists x2 : nat, exists y2 : nat, move 16 P W (square x1 y1) (square x2 y2)); [match goal with [h : At 1 ?P col (square ?X1 ?Y1 :: square ?X2 ?Y2 :: ?l) |- _ ] => apply AGSPG with (i:=I) in h;case h;clear h;intro h;pattern 1 in h;intro end | match goal with [h : At 1 ?P col (square ?X1 ?Y1 :: square ?X2 ?Y2 :: ?l) |- _ ] => pattern 1 in h end]; repeat match goal with [H_goal : exists P : piece, exists x1 : nat, exists y1 : nat, exists x2 : nat, exists y2 : nat, move 16 P W (square x1 y1) (square x2 y2) |- _ ] => clear H_goal; assert (H_goal : exists P : piece, exists x1 : nat, exists y1 : nat, exists x2 : nat, exists y2 : nat, move 16 P W (square x1 y1) (square x2 y2)); [match goal with [h : At 1 ?P col (square ?X1 ?Y1 :: square ?X2 ?Y2 :: ?l) |- _ ] => apply AGSPG with (i:=I) in h;case h;clear h;intro h;pattern 1 in h;intro end | match goal with [h : At 1 ?P col (square ?X1 ?Y1 :: square ?X2 ?Y2 :: ?l) |- _ ] => pattern 1 in h end] end; try assumption; simpl in * |-; try match goal with [h1 : move ?I ?p1 ?c1 (square ?X1 ?Y1) (square ?X2 ?Y2), h2 : At 1 ?p2 ?c2 (square ?X2 ?Y2 :: ?L2), h3 : At 1 ?p3 ?c3 (square ?X ?Y :: ?L) |- _ ] => apply Move_Obstacle with (P2:=p3) (C2:=c3) (x:=X) (y:=Y) (l:=L) in h1;[tauto | (split;[assumption | tauto])] end; match goal with [h1 : move I ?P col (square ?x1 ?y1) (square ?x2 ?y2) |- _] => repeat match goal with [h2 : At 1 P col (square x1 y1 :: ?l2) |- _] => clear h2 end end.
```

Whilst generating SPGs and reducing trajectories, new trajectories of length 1, which represent unmovable obstacles can appear in the context of new subgoals. Due to this, we have to try to use the function *L_block_end_square* again. After this it can occur that new blocked trajectories appear in context and they have to be eliminated from consideration by applying function *L_Block_t*. Now we have to find all the possible moves of the player whose turn it is and so on. In this way our system gives us a unique solution of the problem 2, and this solution is: *1. Pb3 Pd6 2. Bb2 Nd7 3. Bd4 Nf6 4. Nc3 Qd7 5. Qb1 Kd8 6. Kd1 Ne8 7. Kc1 Nf6 8. Kb2 Rg8.*

11 Admissible trajectories

Let us now consider problem showed in Fig. 4. By the problem it's not given number of moves of SPG. Due to this we don't know who made the last move. If we count all obvious white and black moves we get number 6 for both players.

[5] Note that at the end of the function LGSPG we clear trajectories that are no longer valid because the same piece has already made a move on another trajectory.

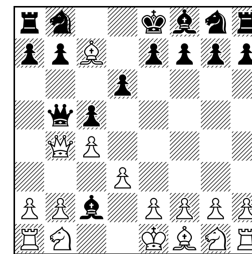

Figure 4. Markus Ott, feenschach 1982, 16+16. SPG?

So, we can conclude that the last move was made by black and that SPG is 12 moves long. If we try to solve the problem under these conditions and in the way described in previous sections, system finds a sequence of 9 moves as the longest sequence. A longer sequence doesn't exist because in all cases a piece remains blocked. So, something with our assumptions is wrong. First, the black did not make the last move since it there would have to exist a SPG with at least 12 moves long. This means that white made the first move. From this, it arises that SPG is longer than 12 moves, at least because the number of moves must be odd. This means that the trajectory of at least one piece is longer than the shortest trajectory.

Problem 5 has to be solved by constructing admissible trajectories of some degree *k*. First we have to try to solve problem 5 by generating at least one trajectory of degree 2. By definition, every admissible trajectory *t* of degree 2 can be generated by two shortest trajectories t_1 and t_2 where the end square of t_1 and starting square of t_2 correspond to each other and this square is called *dock*. So, we have to find one dock for trajectory for which we have to find an admissible trajectory of degree 2. The first question that arises is for what piece do we have to generate an admissible trajectory. We have to generate *bundles* of admissible trajectories of degree 2 for the white bishop from *(3,2)* to *(3,8)* because this bishop has to avoid some movable or unmovable obstacles. It can be shown that it is very useful to find all docks first, that is in our example, all squares (x_D,y_D) for which *MAP 3 2 (MAP_start b)+MAP 3 8 (MAP_start b)=3*. After that we have to generate for every dock (x_D,y_D) a bundle of shortest trajectories from (x_0,y_0) to (x_D,y_D) and a bundle of shortest trajectories from (x_D,y_D) to (x_n,y_n). Finally, we have to merge these two obtained bundles in a set of trajectories as combinations of every trajectory from the first bundle with every trajectory from the second bundle. Therefore, we define the recursive function which makes such

combinations and combines all trajectories into a single bundle:

```
Fixpoint Add_bundles (b1 b2 : list (list chessboard)) (i j n : nat) {struct n} : list (list
chessboard) := match n with S n' => match i with S i' => match j with S j' => (app (nth
(i-1) b1 (square 0 0 :: nil)) (nth (j-1) b2 (square 0 0 :: nil))) :: (Add_bundles b1 b2 i j' n')
| _ => Add_bundles b1 b2 i' (length b2) n' end | _ => nil end | _ => nil end.
```

After generating bundles we can proceed to solve the problem in a manner analogous to the methods proposed in previous sections.[6]

12 Conclusion

In this paper we propose some systematic ideas for the establishment of a formal system for solving SPG's as special type of retrograde chess problems by using Coq - a formal proof management system, while in [10], [11] and [12] we presented a formal system for reasoning about other types of retrograde chess problems, also using Coq. The formal bases of the system described in above publications are very similar to the one in this paper. The systems differ in detail in accordance with their purposes, which are described in the introduction to this article. In this way we get a good foundation for the integration of these two systems into one that will be able to find the SPGs much more complex than those presented in this paper.

There is a general deficit of scientific articles and developed computer systems covering this area. We also think that this approach can be extended to a wide range of complex practical problems. As can be seen from our work, built-in tactics provided with the standard distribution of Coq frequently results in long scripts. In addition to the general improvement of the system described in this article, it is possible to extend this work towards the development of new Coq's tactics.

References

[1] Bertot, Y., Castéran, P.: Interactive Theorem Proving and Program Development, Springer-Verlag, Berlin and Heidelberg, Germany, 2004.

[2] Delahaye, D.: A Tactic Language for the System Coq, Proceedings of Logic for Programming and Automated Reasoning, 2000, pp. 85-95.

[3] Dupuis, É.: Euclide, available at http://lestourtereaux.free.fr, 22th April 2010.

[4] Giménez, E.: A tutorial on recursive types in coq, Technical report, INRIA, 1998.

[5] Gimenez, E., Castéran, P.: A Tutorial on [Co]Inductive Types in Coq, available at http://www.labri.fr/perso/casteran/RecTutorial.pdf, January, 31st 2007.

[6] Huet, G., Kahn, G., Paulin-Mohring, C.: The Coq Proof Assistant - A Tutorial, available at http://coq.inria.fr/V8.2pl1/files/Tutorial.pdf, 27st February, 2009.

[7] Hwa, T., Whipkey, C.: Retractor, available at http://www-cs-students.stanford.edu/~hwatheod, 22th April 2010.

[8] Janko, O., de Heer, J.: The Retrograde Analysis Corner, available at http://www.janko.at/Retros, 22th April 2010.

[9] Leroy, X., Doligez, D., Garrigue, J., R´emy, D., Vouillon, J.: The Objective Caml system, available at http://caml.inria.fr/distrib/ocaml-3.11/ocaml-3.11-refman.pdf, April, 15st 2010.

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

[11] Maliković, M.: Developing heuristics for solving retrograde chess problems, Ph. D. Thesis, University of Zagreb, Faculty of organization and informatics, 2008.

[12] Maliković, M., Čubrilo, M.: What Were the Last Moves?, International Review on Computers and Software, Vol. 5, No. 1, 2010, pp. 59-70.

[13] Smullyan, R. M.: Chess Mysteries of Sherlock Holmes: Fifty Tantalizing Problems of Chess Detection, Random House Inc., 1994.

[14] Stilman, B.: A Formal Model for Heuristic Search, Proceedings of the 22nd annual ACM computer science conference on Scaling up: meeting the challenge of complexity in real-world computing applications, Phoenix, Arizona, United States, 1994, pp. 380-389.

[15] Stilman, B.: A Linguistic approach to geometric reasoning, An international Journal: Computers & Mathematics with Applications, Vol. 26, No. 7, 1993, pp. 29-58.

[16] Stilman, B.: A Linguistic Geometry of the Chess Model, Advances in Computer Chess 7, 1994, pp. 91-117.

[17] Stilman, B.: Linguistic geometry: from search to construction, Kluwer Academic Publishers, 2000.

[18] The Coq Development Team: The Coq Proof Assistant Reference Manual Version 8.2, available at http://coq.inria.fr/refman/, 27st February, 2009.

[19] Wassong, P.: The Natch home page, available at http://natch.free.fr/Natch.html, 22th April 2010.

[20] Wilts, G., Frolkin, A.: Shortest Proof Games, Privately published in Karlsruhe, 1991.

[6] Note that for solving problem showed at figure 4 we have to introduce one more rule and that is: "After some retrograde moves, the opponent's king may not be in check". Detailed analysis and formalization of this rule may be found in [11] and partly in [12].