White And Tan

Nuno Filipe Amaral Oliveira[up201806525] and Luís Miguel Afonso Pinto[up201806206]
FEUP-PLOG, Class 3MIEIC02, Grupo White and Tan_4

[1] Faculty of Engineering of the University of Porto - FEUP

Abstract. This project was done for the subject of PLOG (Logic Programming) in the Masters in Informatics and Computing Engineering. The project consists of developing an application with SICStus Prolog that could find a solution to a decision/optimization problem. To do so we used the clpfd library that contains powerful tools to solve restriction-based problems.

Keywords: White and Tan, Prolog, clpfd.

1 Introduction

Given the option to work with an optimization problem or a decision one, our group chose the latter and the game we ended up with was the Tan and White.
The basic White and Tan problem consists of 4x4 board with uncolored arrows pointing to any direction (diagonal, vertical or horizontal). The point of the game is to color some of the arrows so, in the end, every uncolored arrow should point to exactly one uncolored arrow and every colored arrow should point to exactly to two colored arrows.
The application should also allow the creation of a new, random problem of custom size and solve it.

The structure of the rest of this report is as follows:

- **Problem description** – detailed description of the problem/theme including all the involved restrictions.
- **Approach** – Describe our approach to the problem as decision problem with restrictions. Will contain the following subsections:
  - **Decision Variables** – Description of the decision variables, domains and meaning within this project.
  - **Constraints** – Description of all the constraints used and their implementation using SICStus Prolog.
- **Solution Presentation** – Explanation of the predicates that allow the visualization of the solution in text mode.
- **Experiments and Results**:
  - **Dimensional analyses** – examples of some instantiations of the problem with different sizes and respective solution.
  - **Search strategies** – results of different search strategies.

- **Conclusions and Future Work** – What we take from this project and short final thoughts about the results, the pros and cons of our solution and aspects that could be improved.
- **References** – Articles that helped in the development of this project.
- **Annex** – Other, less relevant results/elements that are not essential to the report.

2 Problem Description

As explained before, given a valid puzzle (puzzle is solvable) with every arrow uncolored our application should be able to find a solution to that specific problem and show it to the user. We must then paint specific arrows so that every uncolored arrow points to only one other uncolored arrow and every colored arrow points to two other colored arrows. The board must have the same number of lines and columns, in other words, a square matrix, its minimum size is 4x4 and it can be increased but as it does the harder it is to create one that is valid.

3 Approach

To solve this problem, we decided to represent the board with a square matrix with numbers from 0 to 7. Each number represents a direction starting at 0 corresponding to north, 1 to northeast, etc.

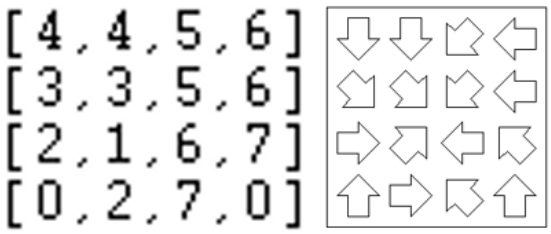

Fig. 1. - Internal representation of a White and Tan puzzle

3.1 Decision Variables

The solution to this problem is a square matrix with the same size as the one initially given to solve. Each cell will have a 0 or a 1 depending on if the arrow of the corresponding cell in the first matrix should be colored or not. If it is 1, the arrow must be colored and if it is 0, it should not be touched. With this information, the domain of every list in the matrix is from 0 to 1.

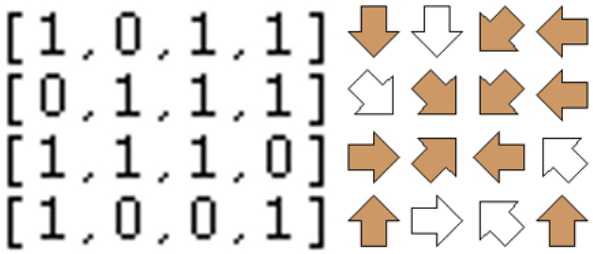

Fig. 2. - Internal representation of the solution to the puzzle shown on Fig.1.

3.2 Restrictions

Every cell in the output matrix must be a 0 or a 1. This restriction is used so we know what arrows we need to paint in the original matrix.

Every arrow that is not colored (0 in the output matrix) must point to exactly one other arrow that is uncolored and the rest must be colored. This means that in the specified direction there are only 2 cells with uncolored arrows and all others are colored.

```
apply_restriction(Tan,Dir,Blank,X,Y,Cols,Lines):-
    Tan #= 0,
    get_list(Dir,Blank,X,Y,Cols,Lines,Lista),
    length(Lista,LL),
    S #= LL - 2,
    global_cardinality(Lista,[0-2,1-S]).
(The predicate get_list/7 returns in Lista all cells in
the direction specified on the original puzzle matrix
while apply_restriction/7 defines the restriction itself
for that cell)
```

Every arrow that is colored (1 in the output matrix) must point to exactly two other colored arrow and the rest are uncolored. With this restriction we assure that in that row/column/diagonal there are no more no less than 3 cells with colored arrows and the rest remains uncolored.

```
apply_restriction(Tan,Dir,Blank,X,Y,Cols,Lines):-
    Tan #= 1,
    get_list(Dir,Blank,X,Y,Cols,Lines,Lista),
    length(Lista,LL),
    S #= LL - 3,
    global_cardinality(Lista,[0-S,1-3]).
```

The predicate apply_restriction/7 defines one of these last two restrictions to every cell in the output matrix.

4 Solution Presentation

To view the solution to a certain puzzle, use the solver/1 predicate. This predicate must receive a valid matrix representation of the puzzle and it can be obtained with the problem/2 predicate that has a few predefined problems, it can also be input manually or even created randomly with create/2. Create/2 receives the order of the square matrix, creates one filled with completely random values and tries to solve it. If the created matrix is solvable it returns it in the other argument, otherwise it creates another random matrix and tries again.
Upon receiving the matrix, solver/1 uses tan_and_white/2 to obtain the solution and print it to the terminal with print_matrix/1 (separates very individual lists with a newline character).

```
solver(Dir):-
    tan_and_white(Dir,Solution),
    print_matrix(Dir),nl,
    print_matrix(Solution),nl.
```

The predicate tan_and_white/2 starts by creating the output matrix and define the domain of all elements.

```
tan_and_white(Dir,Solution):-
    length(Dir,SizeD),
    matrix_generator(SizeD,SizeD,Blank),
    domain_to_list(Blank,0,1),
    length(Blank,L),
    nth0(0,Blank,Linha),
    length(Linha,C),
    tan_and_white(Blank,Dir,0,0,SizeD,C,L,Solution).
```

Then it uses tan_and_white/8 to apply the restrictions mentioned on section 3.2 to every cell in that matrix.

```
tan_and_white(Blank,Dir,X,Y,Size,Col,Line,O):-
    nth0(Y,Blank,BlankR),
    nth0(X,BlankR,B),
    nth0(Y,Dir,DirR),
    nth0(X,DirR,D),
    apply_restriction(B,D,Blank,X,Y,Col,Line),
    P is Y * Col,
    P1 is P + X,
    P2 is P1 + 1,
    X1 is P2 mod Col,
    Y1 is div(P2,Col),
    tan_and_white(Blank,Dir,X1,Y1,Size,Col,Line,O).
```

To run the application just choose one of the following ways:

- Predefined Puzzle (1-8):
  - problem(X,A),solver(A).
- Random Puzzle, X is the size of the square Matrix (4, 5 or 6 only):
  - create(X,A),solver(A).
- Manual Puzzle:
  - solver([[4,4,5,6],[3,3,5,6],[2,1,6,7],[0,2,7,0]]).
- No printing, just solving and confirmation (input the matrix any way you desire):
  - tan_and_white([[4,4,5,6],[3,3,5,6],[2,1,6,7],[0,2,7,0]],Sol).

5 Experiments and Results

5.1 Dimensional analyses

We tested our application's performance with 3 different sized puzzles, all valid.

Table 1. – Statistics for puzzle solving

Size	Time (s)	Pruning	Backtracks	Constraints
4x4	0.01	364	116	109
5x5	0.02	1070	325	366
6x6	0.01	154	21	57

As it is clear to see the application solves a 4x4 puzzle almost instantly, generating a reasonable number of constraints and a small number of backtracks. As for the 5x5 puzzle, the solving time double as did the number of backtracks which leads to conclude that the solving time is somewhat proportional to the backtracks. The numbers of constraints created triples which makes sense since there are more cells on the puzzle.
Looking over to the 6x6 puzzle, the solving time barely changes, but the rest of the data goes against what would be expected. Instead of increasing backtracks and constraints like the previous size did, those elements diminished a lot. Because it is very hard to make a valid 6x6 White and Tan puzzle we believe finding a solution is also simpler, especially for a logic programming language, hence the low numbers.
We also tested the creation of a random puzzle. Testing if the puzzle is valid and has a solution is part of the creation (and creating another until it checks both the conditions) so the following statistics also include that.

Table 2. - Statistics for puzzle creation

Size	Time (s)	Backtracks	Constraints
4x4	0.01	370	61
5x5	0.04	1339	920
6x6	0.62	17664	8540

Even though the creation applies restrictions, it also depends on random number generation, so the statistics are the average of several creations and we removed some outliers. The 4x4 and 5x5 are generated relatively fast with no visual delay most of the times. Contrary to that, creating a 6x6 puzzle may take a while due to the sheer difficulty of creating a valid one through random number generation and that is also why the statistics on this table may vary heavily in reality.

5.2 Search strategies

The restrictions were applied to each individual cell while also taking into consideration the direction the arrow was pointing at (the restriction applied depending on how many arrows where being pointed to). The use of this method along with the global_cardinality/2 predicate from the clpfd library was sufficient for the solution to be found.

6 Conclusions and Future Work

Our puzzle was not the best to test our code on different sized boards. We think a 7x7 puzzle is impossible to make without changing the rules and so our test subjects are very limited.
At the beginning we were uncertain how we should approach the development of the application but after a careful analysis of the clpfd library documentation and other materials given provided by the teachers we think we did what was asked of us on this project which helped us a lot understanding how to use restrictions with SICStus Prolog.
We also think that some aspects, like in every application could be improved, like the change the puzzle creation to not be so random decreasing processing time.
In conclusion, we believe the application was developed successfully and it contributed positively to our capabilities as students.

7 References

Along with several pdf's provided by the teachers, our research included:

- https://erich-friedman.github.io/puzzle/whitetan/
- https://sicstus.sics.se/sicstus/docs/latest4/pdf/sicstus.pdf
- https://www.springer.com/gp/computer-science/lncs/conference-proceedings-guidelines?countryChanged=true

8 Annex

8.1 Source Code

```
:- use_module(library(clpfd)).
:- use_module(library(lists)).
:- use_module(library(random)).

%imprime a solução para dir formatada
%solver(+Dir)
solver(Dir):-
    tan_and_white(Dir,Solution),
    print_matrix(Dir),nl,
    print_matrix(Solution),nl.

%devolve em solution a solução para dir
%tan_and_white(+Dir,-Solution)
tan_and_white(Dir,Solution):-
    length(Dir,SizeD),
    matrix_generator(SizeD,SizeD,Blank),
    domain_to_list(Blank,0,1),
    length(Blank,L),
    nth0(0,Blank,Linha),
    length(Linha,C),
    statistics(total_runtime, _),
    tan_and_white(Blank,Dir,0,0,SizeD,C,L,Solution).

print_time(Msg):-statistics(total_runtime,[_,T]),TS is ((T//10)*10)/1000, nl,write(Msg), write(TS), write('s'), nl, nl.


%devolve em no ultimo parametro a solução de dir, verifica se a procura acabou
%tan_and_white(+Blank,+_Dir,+X,+Y,+_Size,+Col,+Line,-Blank)
tan_and_white(Blank,_Dir,X,Y,_Size,Col,Line,Blank):-
    (X>=Col; Y>=Line).
    %print_time("Time: "),
    %fd_statistics,statistics,
    %write(Blank).

%devolve em O a solução para Dir, pesquisa em torno da dir na posição (x,y)
%tan_and_white(+Blank,+Dir,+X,+Y,+_Size,+Col,+Line,-O)
tan_and_white(Blank,Dir,X,Y,Size,Col,Line,O):-
    nth0(Y,Blank,BlankR),
    nth0(X,BlankR,B),

    nth0(Y,Dir,DirR),
    nth0(X,DirR,D),
    apply_restriction(B,D,Blank,X,Y,Col,Line),

    P is Y * Col,
    P1 is P + X,
    P2 is P1 + 1,
    X1 is P2 mod Col,
    Y1 is div(P2,Col),

    tan_and_white(Blank,Dir,X1,Y1,Size,Col,Line,O).

%aplica restricoes a Blank, força a que na lista obtida, o numero de setas esteja correto
%apply_restriction(+Tan,+Dir,+Blank,+X,+Y,+Cols,-Lines)
apply_restriction(Tan,Dir,Blank,X,Y,Cols,Lines):-
    Tan #= 1,
    get_list(Dir,Blank,X,Y,Cols,Lines,Lista),
    length(Lista,LL),
    S #= LL - 3,
    global_cardinality(Lista,[0-S,1-3]).

%aplica restricoes a Blank, força a que na lista obtida, o numero de setas esteja correto
%apply_restriction(+Tan,+Dir,+Blank,+X,+Y,+Cols,-Lines)
apply_restriction(Tan,Dir,Blank,X,Y,Cols,Lines):-
    Tan #= 0,
    get_list(Dir,Blank,X,Y,Cols,Lines,Lista),
    length(Lista,LL),
    S #= LL - 2,
    global_cardinality(Lista,[0-2,1-S]).
```

```
%verifica os elementos na direção da seta dir, na posição (X,Y)
%get_list(+Dir,+Blank,+X,+Y,+Cols,+Lines,-Lista)
get_list(0,_Blank,_X,Y,_Cols,_Lines,Lista):-
    Y<0,
    Lista = [].
get_list(0,Blank,X,Y,Cols,Lines,Lista):-
    Y >= 0,
    Y1 is Y - 1,
    get_list(0,Blank,X,Y1,Cols,Lines,OutLista),
    nth0(Y,Blank,Row),
    nth0(X,Row,Elem),
    Lista = [Elem|OutLista].

get_list(1,_Blank,_X,Y,_Cols,_Lines,Lista):-
    Y<0,
    Lista = [].
get_list(1,_Blank,X,_Y,Cols,_Lines,Lista):-
    X>=Cols,
    Lista = [].
get_list(1,Blank,X,Y,Cols,Lines,Lista):-
    Y >= 0,
    X < Cols,
    Y1 is Y - 1,
    X1 is X + 1,
    get_list(1,Blank,X1,Y1,Cols,Lines,OutLista),
    nth0(Y,Blank,Row),
    nth0(X,Row,Elem),
    Lista = [Elem|OutLista].

get_list(2,_Blank,X,_Y,Cols,_Lines,Lista):-
    X>=Cols,
    Lista = [].
get_list(2,Blank,X,Y,Cols,Lines,Lista):-
    X < Cols,
    X1 is X + 1,
    get_list(2,Blank,X1,Y,Cols,Lines,OutLista),
    nth0(Y,Blank,Row),
    nth0(X,Row,Elem),
    Lista = [Elem|OutLista].

get_list(3,_Blank,_X,Y,_Cols,Lines,Lista):-
    Y>=Lines,
    Lista = [].
get_list(3,_Blank,X,_Y,Cols,_Lines,Lista):-
    X>=Cols,
    Lista = [].
get_list(3,Blank,X,Y,Cols,Lines,Lista):-
    Y < Lines,
    X < Cols,
    Y1 is Y + 1,
    X1 is X + 1,
    get_list(3,Blank,X1,Y1,Cols,Lines,OutLista),
    nth0(Y,Blank,Row),
    nth0(X,Row,Elem),
    Lista = [Elem|OutLista].

get_list(4,_Blank,_X,Y,_Cols,Lines,Lista):-
    Y >= Lines,
    Lista = [].
get_list(4,Blank,X,Y,Cols,Lines,Lista):-
    Y < Lines,
    Y1 is Y + 1,
    get_list(4,Blank,X,Y1,Cols,Lines,OutLista),
    nth0(Y,Blank,Row),
    nth0(X,Row,Elem),
    Lista = [Elem|OutLista].

get_list(5,_Blank,_X,Y,_Cols,Lines,Lista):-
    Y>=Lines,
    Lista = [].
get_list(5,_Blank,X,_Y,_Cols,_Lines,Lista):-
    X < 0,
    Lista = [].
get_list(5,Blank,X,Y,Cols,Lines,Lista):-
    Y < Lines,
    X >= 0,
    Y1 is Y + 1,
    X1 is X - 1,
    get_list(5,Blank,X1,Y1,Cols,Lines,OutLista),
    nth0(Y,Blank,Row),
    nth0(X,Row,Elem),
    Lista = [Elem|OutLista].
```

```
get_list(6,_Blank,X,_Y,_Cols,_Lines,Lista):-
    X < 0,
    Lista = [].
get_list(6,Blank,X,Y,Cols,Lines,Lista):-
    X >= 0,
    X1 is X - 1,
    get_list(6,Blank,X1,Y,Cols,Lines,OutLista),
    nth0(Y,Blank,Row),
    nth0(X,Row,Elem),
    Lista = [Elem|OutLista].

get_list(7,_Blank,X,_Y,_Cols,_Lines,Lista):-
    X < 0,
    Lista = [].
get_list(7,_Blank,_X,Y,_Cols,_Lines,Lista):-
    Y < 0,
    Lista = [].
get_list(7,Blank,X,Y,Cols,Lines,Lista):-
    Y >= 0,
    X >= 0,
    Y1 is Y - 1,
    X1 is X - 1,
    get_list(7,Blank,X1,Y1,Cols,Lines,OutLista),
    nth0(Y,Blank,Row),
    nth0(X,Row,Elem),
    Lista = [Elem|OutLista].


%aplica as restriçoes a uma lista de listas, cada valor deve estar entre Bot e Up
%domain_to_list(?ListOfLists,+Bot,+Up)
domain_to_list([],_,_).
domain_to_list([List|H],Bot,Up):-
    domain(List,Bot,Up),
    domain_to_list(H,Bot,Up).

%aplica as restriçoes à primeira fila de uma matriz,(elementos devem estar entre 2 e 6)
%domain_to_first_line(?Matrix)
domain_to_first_line([List|_H]):-
    domain(List,2,6).
```

```
%cria uma matriz de direções com o tamanho Size
%create(+Size,-Matrix)
create(Size,Matrix):-
    List_length #= Size,
    length(M,List_length),
    matrix_generator(Size,Size,M),
    create_random(Size,Matrix,M,_B).

create_random(Size,Matrix,M,B):-
    domain_to_list(M,0,7),
    domain_to_first_line(M),
    append(M,MM),
    nth0(1,M,M1),
    nth0(1,M1,P1),
    nth0(0,M,M2),
    nth0(0,M2,P2),
    random(0,7,R11),
    random(2,4,R00),
    P1 #= R11,
    P2 #= R00,

    tan_and_white(M,_),
    list2matrix(MM,Size,M),
    Matrix = M,
    ((B = N) ; (B\=N,fail)).

length_(Length, List) :- length(List, Length).
```

```
%transforma uma lista numa lista de listas
%list2matrix(+List, +RowSize, -Matrix)
list2matrix(List, RowSize, Matrix) :-
    length(List, L),
    HowManyRows is L div RowSize,
    length(Matrix, HowManyRows),
    maplist(length_(RowSize), Matrix),
    append(Matrix, List).

%gera uma maytriz de tamanho Size
%matrix_generator(+Size,+NLinha,-Matrix)
matrix_generator(Size,1,Matrix):-
    length(M2,Size),
    M3 = M2,
    Matrix = [M3].
matrix_generator(Size,Left,Matrix):-
    Left > 1,
    L is Left - 1,
    matrix_generator(Size,L,M1),
    length(M2,Size),
    append([M2],M1,M3),
    Matrix = M3.

print_matrix([]).
print_matrix([H|R]):-
    write(H),
    nl,
    print_matrix(R).
```

8.2 Predefined Problems

```
problem(1,
   [[4,4,5,6],
    [3,3,5,6],
    [2,1,6,7],
    [0,2,7,0]]
       ).

problem(2,
    [[4,4,5,6,6],
     [2,2,3,4,6],
     [1,0,7,6,0],
     [1,1,0,0,6],
     [0,1,0,7,0]
    ]
).

problem(3,
    [[3,2,3,3,2,4],
    [2,5,3,1,4,0],
    [1,1,0,2,5,5],
    [0,0,1,0,0,4],
    [1,0,2,0,5,0],
    [0,1,1,1,0,7]
    ]
)
```

```
problem(7,
    [[4,4,4,6,4],
     [3,4,3,5,6],
     [1,2,0,4,5],
     [2,1,6,0,6],
     [0,0,2,6,6]
    ]
).
problem(8,[
    [2,3,3,3,2,4],
    [2,1,4,1,4,0],
    [0,1,2,0,5,5],
    [2,0,0,3,0,4],
    [2,0,1,0,0,0],
    [0,1,0,2,7,6]
]).
```

```
problem(4, [[3,2,2,3,4,4],[2,1,3,2,5,0],[0,1,2,4,1,5],[2,0,0,0,0,4],[1,1,0,0,6,0],[0,0,1,2,0,7]]).

problem(5,
   [[2,3,4,5],
    [2,3,6,5],
    [1,2,0,0],
    [1,2,7,7]]
       ).

problem(6,
    [[2,3,3,6,5],
     [4,4,2,4,4],
     [1,3,1,0,7],
     [1,2,1,7,0],
     [2,0,2,7,0]
    ]
).
```