Houses - Constraint Programming

Afonso Jorge Moreira Maia Ramos[1,2] and João Dias Conde Azevedo[1,2]

[1] Faculty of Engineering of the University of Porto
[2] IEEE University of Porto

U.PORTO
FEUP FACULDADE DE ENGENHARIA
UNIVERSIDADE DO PORTO

Logic Programming
Programação em Lógica
3MIEIC06 - House_2

Abstract. This project was developed entirely using the SICStus Prolog developing environment for a Logic Programming course which proposed its students to solve either puzzles or optimisation problems using constraints programming within the Logic Programming realm. The problem chosen is a puzzle problem and has the objective of finding the pairs of houses among a list of house coordinates so that there are at max two different distances among these pairs. We will explain our implementation throughout this article.

Keywords: prolog · constraint · clpfd · logic-programming.

1 Introduction

For the Logic Programming Course, of the Master in Informatics and Computing Engineering at the Faculty of Engineering of the University of Porto, we were proposed with a problem in which we were obliged to only use constraint programming with SICStus Prolog and the CLP(FD) library.

Among the various problems that we could choose, we chose the House puzzle, an optimisation problem, because of it's simplicity when implemented without limiting the logic programming to just constraints, while when implementing without backtracking poses as a great challenge, enabling us to truly face the differences of the different ways of attaining the solution and to understand the paradigm shift.

This article is structured with the following layout:

- **Problem Description** - Detailed descriptions of the problem that is being analysed
- **Approach** - Description of the approach for the development of the solution
  - **Decision Variables** - Description of the decision variables and their domains
  - **Constraints** - Description of the constraints applied to the respective problem and their implementation
  - **Evaluation Function** - Description of the evaluation methods of the obtained solution and its implementation
  - **Search Strategy** - Description of the labelling strategy used, namely it's variables and values
- **Solution Presentation** - Explanation of the visualisation of the solution within SICStus
- **Results** - Demonstration of some application examples with different settings, complexity and instances
- **Conclusions and Future Work** - Conclusions of the project, analysis of the results and advantages of the implemented solution

2 Problem Description

The Houses Puzzle's is a puzzle created by Erich Friedman, a puzzles enthusiast, with the objective of connecting pairs of houses so that there are at max two different distances. These pairs of houses, with only one connection each, are created with the elements of a list with even number of house coordinates, which is the puzzle's input. Below is an example of a solution of a puzzle.

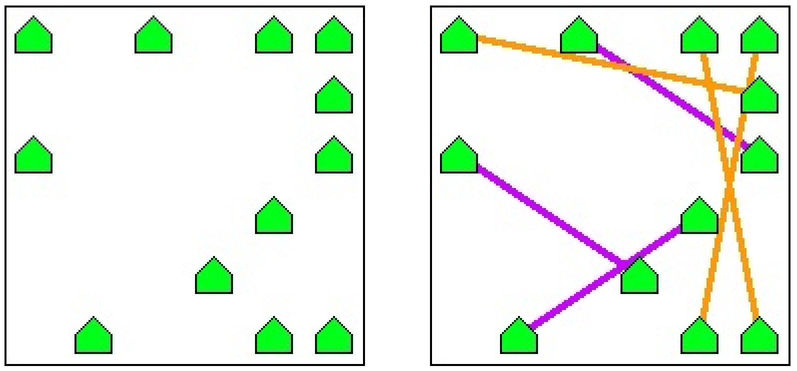
(a) House Puzzle Example (b) House Puzzle Example Solution

Fig. 1: An example of a representation of the pairs of houses with the two distinct distances among them.

3 Approach

Throughout the development of this project our implementation changed a considerable amount of times, one where we paired the coordinates themselves, so one house was seen as a list of coordinates (ex. [3, 4]), however, the latter implementation showed to be flawed due to later implementation difficulties related to verifying the existence of said house in the list of connected houses due to limitations set by the predicate *element(?N, +Vs, ?V)*. Of course other predicates like *table(+Tuples,+Extension)* could be used but the group decided to work with flat lists of integers, making it simpler, avoiding nested lists of lists and making it possible to use *element/3*.

Thus, our solver predicate takes a list of houses. Each house is a list of coordinates [x, y]. Coordinates X and Y go from [0, Range] where Range can be any given one. Given this representation and as mentioned before, we focused on working with flat lists of integers.

With the input list of houses we create a flat coordinate list. Thus we end up with a list of integers where each 2 consecutive integers are coordinates of an house.

Connections are represented as a flat list of integers. Each 2 integers represent a connection between the house located at the position of the first connection index and the house located at the position of the second. This means connections are a list of integers where each 2 consecutive integers are indexes to the Houses list and represent a connection.

Distances are kept squared so they are always integers i.e. they are the sum of 2 squared integers (difference of coordinates).

3.1 Decision Variables

As mentioned, for decision variables we used a list of length two for the Distances and for the Connections we used a list with the size of the input size, where each integer points to the house list. Each 2 consecutive ones are a connection.

3.2 Constraints

As for constraints, we made sure the Connections list was the size of the houses list and it's integers were all different. This because each one of the integers is an index to the houses list. We restricted it's domain to be between the first index, one, and the last, size of the houses list.

We restricted the Distances list to size two and forced them to be different from each other. We also restricted their domain between 1 (minimal distance) and the length of a *diagonal in our domain*, which would be equivalent to having one house in the (0,0) and the furthest away house in the (UB, UB), where UB is the upper bound of our coordinates domain. With basic mathematics it is clear the upper bound of the distance which we keep squared will be UB * UB + UB * UB, equivalent to 2 * UB * UB.

We restricted each connection in the Connections list to have a distance of one or another in the Distances list.

It is also worth mentioning the restrictions made to avoid symmetries. In fact, by avoiding permutations of connections such as declaring A to B is the same as B to A and by declaring a list with connections A to B and C to D is the same solution as a list of connections C to D and A to B, made the program MUCH faster. Several orders of magnitude faster.

And that was it. That simple.

3.3 Search Strategy

In early development we used the default labelling options, which are the *leftmost* option for variable selection and the *step* option for variable value selection.

Only after completing the program we tried new options. In order to find out the best combination of options we tried **every single combination of variable selection and value attribution labelling options possible**.

A summary of those tests can be found in the Results section. Looking at the time results we determined that there was a clear distinction between some combinations which were awful and some which were good.

Among the good ones, the combination of the options *first-fail or first-fail-constrained* for variable selection and *down* for value attribution was optimal, achieving better results than the other ones up to four orders of magnitude. Thus we used the **ffc and down labelling options for our solver**.

For our generator, we did not run multiple tests, so we ended up using the *first-fail-constraint* labelling option and *our custom value selection option.* This was implemented in order to achieve random values for the variables, specifically because we wanted to randomise the distance value.

4 Solution Presentation

After entering the puzzle solving menu, we present the user with two distinct options, solving a pre-loaded puzzle, or generating a new one. When presenting the solution itself we list the pairs of houses (connections), as well as a distance indicator for each pair to show which ones have one distance and which ones have the other.

```
=====================================
=            Puzzles Menu           =
=====================================
=                                   =
=    1. Solve saved puzzles         =
=    2. Generate new puzzle         =
=    3. Back                        =
=                                   =
=====================================
Choose an option:        1
Enter puzzle name:       puzzle1


This might take a few minutes... or not :D
---Make the following connections---
Connection: [3,0] <-> [4,4] by a distance of 17
Connection: [4,0] <-> [0,1] by a distance of 17
Connection: [4,1] <-> [0,2] by a distance of 17
Connection: [1,2] <-> [2,3] by a distance of 2
Connection: [4,3] <-> [0,4] by a distance of 17
---End---

Press <Enter> to continue.
|:
```

Listing 1: Output example for a puzzle solution

For this purpose, we created the predicate *printSolution(-Houses,-Connections)*.

```
/*  printSolution(+Houses, +Connections)

    Prints the solution computed by the solver in an
    human-friendly way.
*/
printSolution(_, []).
printSolution(Houses, [I1, I2|Connections]):-
    nth1(I1, Houses, House1),
    nth1(I2, Houses, House2),
    computeDistance(House1, House2, Distance),
    write('Connection: '), write(House1), write(' <-> '),
    write(House2), write(' by a distance of '),
    write(Distance), nl,
    printSolution(Houses, Connections).
```

Listing 2: Solution printing predicate

5 Results

Value	leftmost (default)	min	max	ff	ffc	anti-ff	occurrence	max-regret
step (default)	3743	9400	6715	4468	4544	26433	4064	4914
enum	3659	6139	6322	12907	13961	11546	3671	4410
bisect	3713	8063	2207	3703	3539	7589	3823	5884
median	10246	20995	20915	1365	1393	81431	11946	12234
middle	10199	22694	21816	1388	1411	77432	11788	11770
up	3932	9705	6714	4455	4513	25587	4104	4725
down	12040	23936	105239	263	262	133886	13572	13471

Table 1: Labelling combinations and time results

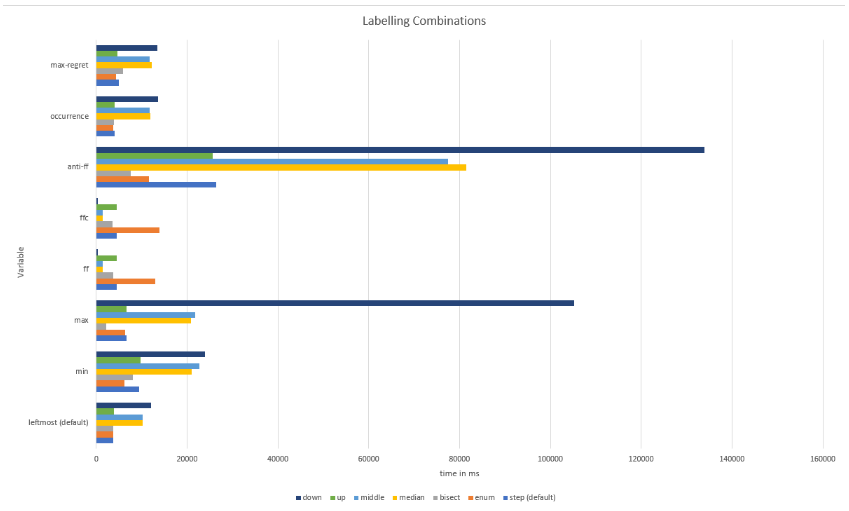

Fig. 2: Table 1 content visually displayed by means of a graph.

As mentioned, the group did several tests. In the previous table we can see each and every combination of labelling options for variable selection and value attribution and the time results for the solver.
In the top row we have the different labelling options for variable selection and on the left column the different labelling options for the value attribution. All the times are in milliseconds (ms).
This times were obtained when passing the same puzzle as input to our solver. That puzzle was the one below:

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

As concluded before, the group opted for the clearer shorter run times of the *ff and ffc* labelling options together with the *down* one.
The group considers it makes sense, as the *first-fail-constrained* option will first select variables with small domains i.e. they are very constrained and we will in theory most likely find out their actual value faster, helping with the remaining constraints due to the constraint propagation mechanism of the CLPFD library. As for the *down* option, the group considers that since in the puzzle input used and in most of the puzzles, the distances are usually high (squared distances), and so it is faster to search from the maximum value down than from 1 (minimal distance) and from there up.

V	**3**	**6**	**12**	**24**
2	2	0	0	0
4	3	0	2	6
6	3	0	21	57
8	9	130	1011	1731
10	79	5740	23383	765025

Table 2: Scalability tests

The table above expresses the scalibility tests done by the group. We tried with increasing number of houses, represented in the left column, and with many domain ranges, the top row. We used the optimla labelling options for us, *ffc + down*.
As expected, as the number of houses increases and the number of possible points in space increases aswell with the domain range, the solver takes more and more time.

6 Conclusions and Future Work

In summary, with this puzzle the group learned to work with a constraint programming paradigm. In this case we used SICStus PROLOG, but constraint logic programming is widely used in optimisation's and scheduling problems and in more languages than PROLOG.
During development it was clear to us the importance of the modelling phase of constraint programming where the way the data is represented is of huge importance. It was also made clearer the importance of defining the correct domains to each domain variable.
Last but not least, with the experiences performed, the group saw in first hand the huge impact labelling options can have, not in the actual outcome but in the time it takes to compute it.
In the future, the group would like to learn more about constraint programming as it it's power and utility was way above expected. Thinking in a more declarative way and restricting the domain of a seemingly hard or impossible problem, can quickly turn it into something a computer easily finds a solution for.

7 Anexes

7.1 houses.pl

```prolog
/* PUZZLE PROBLEM:
↪  https://www2.stetson.edu/~efriedma/puzzle/house/ */

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

:- include('menus.pl').
:- include('cli.pl').
:- include('restrictions.pl').
:- include('puzzles.pl').
:- include('utils.pl').

/* Application entry point */
houses:- main_menu.
```

7.2 restrictions.pl

```prolog
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%            CLPFD           %
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
```

```
/*  connect(+Houses, -Connections, -Runtime)

    Solver predicate. Takes a list of houses. Each house is a list
↪   of coordinates [x, y]. Coordinates X and Y go from [0,
↪   Range].
    Computes the solution and returns the connections made in
↪   Connections and the total runtime in Runtime.

    ---Implementation---

    We focused on working with FLAT lists of integers. It makes it
↪   easier to use some CLPFD predicates without
    constantly flattening the lists. As such below we describe our
↪   representation.

    1 - The argument passed onto the predicate is a list of lists,
↪   each representing an house as a pair [x, y]
    2 - That list is flattenned thus we have a FlatHouses list
↪   where each 2 consecutive integers are coordinates of an
↪   house.
    3 - Connections is also a flat list of integers. Each 2
↪   integers represent a connection between the house located at
↪   the
    position of the first connection index and the house located
↪   at the position of the second.
    This means connections are a list of integers where each 2
↪   consecutive integers are indexes to the Houses list and
↪   represent a connection.
    4 - Distances are kept squared so they are always integers
↪   (sum of 2 squared integers).
*/
connect(Houses, Connections, Runtime):-
    statistics(runtime,[Start|_]), %statistics - runtime
     ↪  calculation (ms)

    append(Houses, FlatHouses),
    max_member(Max, FlatHouses), %max coordinate will represent
     ↪  our domain range

    length(Houses, NHouses),
    length(Connections, NHouses), %each connection integer will
     ↪  point to an house
    domain(Connections, 1, NHouses),
    all_distinct(Connections), %all the houses must be connected
     ↪  so all are different
    Distances = [D1, D2], %there are 2 different distances
    all_distinct(Distances),
    MaxDis is 2 * Max * Max,
    domain(Distances, 1, MaxDis), %can't be 0 (2 houses same
    ↪  coords) and range from 1 to the maximum representing a
    ↪  diagonal on our "squared matrix" domain range

    restrictConnectDistances(Connections, FlatHouses, Distances),
    ↪  %connections will have one of the two existing distances

    %eliminate symmetries
    D1 #< D2,
    removeCoordsPermutation(Connections), %avoids permutations of
    ↪  type A <-> B and B <-> A
    removeConnectionsPermutation(Connections), %avoids
    ↪  permutations of type [A<->B, C<->D] and [C<->D, A<->B]
    labeling([ffc, down], [D1, D2|Connections]),

    %statistics - runtime calculation (ms)
    statistics(runtime,[Stop|_]),
    Runtime is Stop - Start.


/*  generate(-Houses, +NHouses, +Domain)

    Generator predicate. Returns generated puzzle in Houses. Takes
↪   the even number of houses to generate and the X and Y upper
↪   bound limit.

    ---Implementation---

    We followed a similiar line of thought as previously. The
↪   representation is the same. The restrictions end up being the
↪   same except now
    we don't know the house list and we need to randomize the
↪   distances.

    1 - While most is the same, there is now the extra need of
↪   ensuring all pairs [x,y] are different. That means between 2
↪   houses
    of the generated list, either their X is different or their Y
↪   is different (or both are different).
```

```
    2 - We also needed to not only restrict the distances to
↪   possible values but also to have different values for each
↪   generation. For that
    we followed:
↪   https://sicstus.sics.se/sicstus/docs/4.0.4/html/sicstus/Enumeration-Predicates.html
    Basically, we created our custom value selection method for
↪   the distance variables. We made it random.
*/
generate(Houses, NHouses, Domain):-
    NFlatHouses is NHouses * 2,
    length(FlatHouses, NFlatHouses), %for each house we have 2
     ↪  coordinates, so double-sized list
    domain(FlatHouses, 0, Domain), %the coordinates can go from 0
     ↪  to a defined upper bound
    ensureDifferentHouses(FlatHouses), %all house coordinates are
     ↪  different

    length(Connections, NHouses), %same as before, connections are
     ↪  indexes to houses
    domain(Connections, 1, NHouses),
    all_distinct(Connections), %all indexes to all houses appear
     ↪  because all connected

    MaxDis is 2 * Domain * Domain,
    Distances = [D1, D2],
    domain(Distances, 1, MaxDis), %distances go from [1, MaxDis]
     ↪  where MaxDis represents the distance of a "diagonal"
    all_distinct(Distances),  %they're both different

    restrictConnectDistances(Connections, FlatHouses, Distances),
     ↪  %same as before connections have one of the two distances

    %symmetries removal
    removeCoordsPermutation(Connections),
    removeConnectionsPermutation(Connections),
    D1 #< D2,

    append(FlatHouses, Connections, Vars), %unique list of domain
     ↪  variables
    append(Vars, Distances, Vars2),
    labeling([value(randomLabeling), ffc], Vars2),
    buildHouseList(FlatHouses, Houses, []). %turn into puzzle
     ↪  input format
```

```
/*  restrictConnectDistances(+Connections, +FlatHouses,
 ↪  +Distances)

    Ensures all connections have one of two distances.
    Takes the connections list, a list of integers where for each
 ↪  2 consecutive integers there is a connection between
    the house pointed by the first integer and the house pointed
 ↪  by the second integer.
    Takes the FlatHouses list to be able to compute the distance
 ↪  of the connection and the Distances list to check it is
 ↪  contained there.
*/
restrictConnectDistances([], _, _).
restrictConnectDistances([I1, I2|Connections], FlatHouses,
 ↪  Distances):-
    HX1 #= I1 * 2 - 1,
    HY1 #= I1 * 2,
    HX2 #= I2 * 2 - 1,
    HY2 #= I2 * 2,
    element(HX1, FlatHouses, X1),
    element(HY1, FlatHouses, Y1),
    element(HX2, FlatHouses, X2),
    element(HY2, FlatHouses, Y2),
    computeDistance([X1, Y1], [X2, Y2], Dis),
    element(_, Distances, Dis),
    restrictConnectDistances(Connections, FlatHouses, Distances).

/*  ensureUnique(+[X1, Y1], +Houses)

    Ensures the pair [X1, Y1] is unique in the list of Houses.
    A coordinate is unique if there is no other with the same X
 ↪  and Y.
*/
ensureUnique(_, []).
ensureUnique([X1, Y1], [X2, Y2|Houses]):-
    X1 #\= X2 #\/ Y1 #\= Y2,
    ensureUnique([X1, Y1], Houses).


/*  ensureDifferentHouses(+FlatHouses)

    Ensures all the houses are different from each other.
*/
ensureDifferentHouses([]).
ensureDifferentHouses([X1, Y1|FlatHouses]):-
```

```
    ensureUnique([X1, Y1], FlatHouses),
    ensureDifferentHouses(FlatHouses).


/*  buildHouseList(-FlatHouses, +Houses, -Acc)

    Builds a puzzle formatted list i.e. a list of houses (lists
 ↪  [x,y]) from the FlatHouses list.
*/
buildHouseList([], Houses, Houses).
buildHouseList([X, Y|FlatHouses], Houses, Acc):-
    buildHouseList(FlatHouses, Houses, [[X, Y]|Acc]).


/*  removeCoordsPermutation(+Connections)

    Ensures that connection A<->B appears once and not also
 ↪  B<->A.
*/
removeCoordsPermutation([]).
removeCoordsPermutation([C1, C2|Connections]):-
        C1 #=< C2,
        removeCoordsPermutation(Connections).


/*  removeConnectionsPermutation(+Connections)

    Ensures that connection list [A<->B, C<->D] appears once and
 ↪  not also permutations like [C<->D, A<->B].
*/
removeConnectionsPermutation([_, _]).
removeConnectionsPermutation([C1, _, C3|Connections]):-
        C1 #=< C3,
        removeConnectionsPermutation([C3|Connections]).
```

7.3 cli.pl

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%    COMMAND-LINE INTERFACE MODULE     %
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

/*  clear_console

    Simulates a console clear by printing out 100 newline
 ↪  characters ('\n').
*/
```

```
clear_console:-
    clear_console(100), !.


/*  clear_console(+N)

    Prints N newline characters ('\n') on console.
*/
clear_console(0).
clear_console(N):-
        nl,
        N1 is N-1,
        clear_console(N1).


/*  request_enter

    Requests user to press Enter key to advance.
    Reads the remaining newline character ('\n') in the console.
*/
request_enter:-
        write('Press <Enter> to continue.\n'),
        get_char(_), !.


/*  get_int(-Input)

    Reads a number up to 2-digits and unifies it with Input.
*/
get_int(Input):-
    get_code(FirstChar),
    get_code(SecondChar),
    aux_get_int(FirstChar, SecondChar, Input).

/* SecondChar code is 10 i.e. '\n' character code. 1-digit number,
↪  Input unified with it. */
aux_get_int(FirstChar, 10, Input):-
    Input is FirstChar - 48.

/*  SecondChar code is not 10 i.e. not '\n' character code.
↪  2-digit number, Input unified with it */
aux_get_int(FirstChar, SecondChar, Input):-
    FirstDigit is FirstChar - 48,
    SecondDigit is SecondChar - 48,
    Temp is FirstDigit * 10,
    Input is Temp + SecondDigit,
    get_char(_).


/*  get_word(-Word, +Acc)

    Reads characters and joins them in an atom untill \n
*/
get_word(Word, Acc):-
    get_char(Char),
    Char \= '\n',
    atom_concat(Acc, Char, Acc2),
    get_word(Word, Acc2).

get_word(Word, Word).
```

7.4 puzzles.pl

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%                      PRE-LOADED PUZZLES                       %
%     https://www2.stetson.edu/~efriedma/puzzle/house/          %
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

puzzle1([
    [3,0], [4,0],
    [0,1], [4,1],
    [0,2], [1,2],
    [2,3], [4,3],
    [0,4], [4,4]
]).

puzzle2([
    [0,0], [1,0], [2,0], [4,0],
    [2,1], [4,1],
    [3,2],
    [1,3],
    [0,4], [2,4]
]).

puzzle3([
    [2,0],
    [0,1], [2,1], [3,1],
    [1,2],
    [0,3], [4,3],
    [0,4], [3,4], [4,4]
]).
```

```
puzzle4([
    [4,0],
    [0,1], [3,1],
    [0,2], [3,2],
    [3,3], [4,3],
    [0,4], [2,4], [4,4]
]).

puzzle5([
        [0,0],[2,0],[3,0],
        [2,1],[3,1],
        [2,2],
        [0,3],[3,3]
]).

puzzle15([
    [0,0], [3,0], [5,0], [6,0],
    [3,1], [6,1],
    [0,2], [3,2],
    [0,3], [5,3],
    [6,4],
    [4,5],
    [3,6],[5,6]
]).

h8b24([[1,2],[24,0],[6,10],[5,16],[17,0],[24,6],[20,6],[17,6]]).

h10b12([[8,1],[7,3],[5,7],[4,9],[1,6],[3,5],[1,11],[6,10],[10,3],[7,8]]).

h2b3([[1,0],[2,3]]).

h2b6([[5,1],[4,1]]).

h2b6([[1,4],[4,0]]).

h2b12([[2,5],[11,3]]).

h2b24([[11,22],[24,15]]).

h4b3([[2,3],[3,0],[1,0],[0,2]]).

h4b6([[1,1],[6,2],[1,4],[2,3]]).

h4b12([[10,6],[12,11],[6,9],[9,4]]).
```

```
h4b24([[20,4],[9,16],[1,15],[8,19]]).

lllll([[4,13],[13,10],[3,5],[12,6]]).
```

7.5 utils.pl

```
%%%%%%%%%%%%%%%%%%%%%
%   UTILS MODULE    %
%%%%%%%%%%%%%%%%%%%%%


/*  computeDistance(+[X1, Y1], +[X2, Y2], -Dis)

    Computes the squared Euclidean distance between [X1, Y1] and
 ↪  [X2, Y2].
*/
computeDistance([X1, Y1], [X2, Y2], Dis):-
    DiffX #= X2 - X1,
    DiffY #= Y2 - Y1,
    Dis #= DiffX * DiffX + DiffY * DiffY.


/*  printSolution(+Houses, +Connections)

    Prints the solution computed by the solver in an
 ↪  human-friendly way.
*/
printSolution(_, []).
printSolution(Houses, [I1, I2|Connections]):-
    nth1(I1, Houses, House1),
    nth1(I2, Houses, House2),
    computeDistance(House1, House2, Distance),
    write('Connection: '), write(House1), write(' <-> '),
     ↪  write(House2), write(' by a distance of '),
     ↪  write(Distance), nl,
    printSolution(Houses, Connections).


/*  randomLabeling(Var, _Rest, BB, BB1)

    Custom random labeling heuristic for value selection.
*/
randomLabeling(Var, _Rest, BB, BB1):-
    fd_set(Var, Set),
    randomSelector(Set, Value),
    (
      first_bound(BB, BB1), Var #= Value;
      later_bound(BB, BB1), Var #\= Value
    ).


/*  randomSelector(Set, BestValue)

    Randomly selects a variable from the Set.
*/
randomSelector(Set, BestValue):-
    fdset_to_list(Set, List),
    length(List, Len),
    random(0, Len, RandomIndex),
    nth0(RandomIndex, List, BestValue).


/*  puzzleFilePath(-FilePath)

    Gives the created puzzles file path.
*/
puzzleFilePath(FilePath):-
    current_directory(Dir),
    atom_concat(Dir, 'puzzles.pl', FilePath).


/*  savePuzzle(+Houses, +PuzzleName)

    Saves the puzzle Houses given with the name PuzzleName in the
 ↪  created puzzles file.
*/
savePuzzle(Houses, PuzzleName):-
    puzzleFilePath(PuzzlesPath),
    Term =.. [PuzzleName, Houses],
    open(PuzzlesPath, append, Stream),
    write(Stream, '\n'),
    write_term(Stream, Term, []),
    write(Stream, '.\n'),
    close(Stream),
    write(PuzzleName), write(' saved successfully at '),
     ↪  write(PuzzlesPath), nl, nl,
    reconsult(PuzzlesPath).
```

7.6 menus.pl

```
%%%%%%%%%%%%%%%%%%%%%%%
%    MENUS MODULE     %
%%%%%%%%%%%%%%%%%%%%%%%

/* Main menu */
main_menu:-
        print_main_menu,
        get_int(Input),
        main_menu_option(Input).

main_menu_option(1):-
        puzzles_menu.

main_menu_option(2):-
        about.

/* Does nothing, terminates program */
main_menu_option(3).

main_menu_option(_):-
        write('\nError: invalid input.\n'),
        request_enter,
        main_menu.

print_main_menu:-
        clear_console,
        write('=================================\n'),
        write('=             Houses            =\n'),
        write('=================================\n'),
        write('=                               =\n'),
        write('=   1. Solve Puzzles            =\n'),
        write('=   2. About                    =\n'),
        write('=   3. Exit                     =\n'),
        write('=                               =\n'),
        write('=================================\n'),
        write('Choose an option:\t').


/* About */
about:-
        print_about,
        request_enter,
        main_menu.
```

```
print_about:-
        clear_console,
        write('=================================================================\n'),
        write('=                            ABOUT
        ↪  =\n'),
        write('=================================================================\n'),
        write('=   In Houses the objective is to connect pairs of
        ↪  houses so   =\n'),
        write('=   that there are at max two different distances.
        ↪  =\n'),
        write('=
        ↪  =\n'),
        write('=   All houses must be connected to a single
        ↪  different house.  =\n'),
        write('=
        ↪  =\n'),
        write('=   Each puzzle below has a unique solution.
        ↪  =\n'),
        write('=
        ↪  =\n'),
        write('=   Some pre-loaded puzzles were created by Erich
        ↪  =\n'),
        write('=   Friedman, 2008.
        ↪  =\n'),
        write('=
        ↪  =\n'),
        write('=   Houses Puzzles was developed for the course of
        ↪  =\n'),
        write('=   Logic Programming.
        ↪  =\n'),
        write('=
        ↪  =\n'),
        write('=   It is written entirely in SICStus PROLOG using
        ↪  the CLPFD   =\n'),
        write('=   library (Constraing Logic Programming with
        ↪  Finite Domains) =\n'),
        write('=
        ↪  =\n'),
        write('=   Developed by:
        ↪  =\n'),
        write('=    > Afonso Ramos
        ↪  =\n'),
        write('=    > Joao Conde
        ↪  =\n'),
        write('=
        ↪  =\n'),
        write('================================================================\n').


/* Puzzles menu */
puzzles_menu:-
        print_puzzles_menu,
        get_int(Input),
        puzzles_menu_option(Input).

puzzles_menu_option(1):-
        solve_puzzles_prompt.

puzzles_menu_option(2):-
        generate_puzzles_prompt.

puzzles_menu_option(3):-
        main_menu.

puzzles_menu_option(_):-
        write('\nError: invalid input.\n'),
        request_enter,
        puzzles_menu.

print_puzzles_menu:-
        clear_console,
        write('=================================\n'),
        write('=         Puzzles Menu          =\n'),
        write('=================================\n'),
        write('=                               =\n'),
        write('=   1. Solve saved puzzles      =\n'),
        write('=   2. Generate new puzzle      =\n'),
        write('=   3. Back                     =\n'),
        write('=                               =\n'),
        write('=================================\n'),
        write('Choose an option:\t').


/* Solver interface */
solve_puzzles_prompt:-
        print_puzzle_to_solve(PuzzleName), nl,
        Term =.. [PuzzleName, Houses],
        Term,
        nl, write('This might take a few minutes... or not :D'),
         ↪  nl, nl,
        connect(Houses, Connections, Runtime),
        %print of the solution in an human-friendly way
    write('---Make the following connections---'), nl,
    printSolution(Houses, Connections),
    write('---End---'), nl,
        write('---Total solver runtime of '), write(Runtime),
        ↪  write(' ms'), nl,
        request_enter,
        puzzles_menu.

print_puzzle_to_solve(PuzzleName):-
        write('Enter puzzle name:\t'),
        get_word(PuzzleName, '').


/* Generator interface */
generate_puzzles_prompt:-
        print_generate_puzzle(PuzzleName, NHouses, Domain), nl,
        nl, write('This might take a few minutes... or not :D'),
        ↪  nl, nl,
        generate(Houses, NHouses, Domain),
        savePuzzle(Houses, PuzzleName),
        request_enter,
        puzzles_menu.

print_generate_puzzle(PuzzleName, NHouses, Domain):-
        write('Enter new UNIQUE puzzle name:\t'),
        get_word(PuzzleName, ''),
        write('Enter EVEN number of houses:\t'),
        get_int(NHouses),
        write('Enter coordinates domain upper bound (> 1):\t'),
        get_int(Domain).
```