Ben-Gurion University of the Negev
Department of Computer Science
Logic programming

Assignment 2, Semester B, 2018

Deadline: April 13 at 23:00

Part 1: Nonograms

Nonograms are picture logic puzzles in which cells in a grid have to be filled or left blank according to numbers given at the side of each row and column of the grid to reveal a hidden picture. The list of numbers next to each row and column shows how many blocks (contiguous filled-in cells) there are in the row or column and measure their sizes.

A nonogram puzzle is a term of the form `nonogram(N,M,ColData,RowData)` where `N` is the number of rows, `M` is the number of columns, `ColData` is a list of `M` lists of numbers (some could be empty) describing the constraints on the columns and `RowData` is a list of `N` lists of numbers (some could be empty) describing the constraints on the rows.

A solution to the nonogram puzzle is defined as a matrix, representing the rows of the solution. The values of each row are 0 (blank cell) or 1 (filled cell). The matrix itself is represented as a length `N` list of length `M` lists (see example below, and in Tasks 1 and 2).

For example, one solution of the nonogram puzzle `nonogram(4,4,ColData,RowData)` in which `ColData = [[3],[1,1],[1,1],[2]]` and `RowData = [[4],[1,1],[2],[1]]` is the following 4×4 matrix:

```
[[1, 1, 1, 1],
 [1, 0, 0, 1],
 [1, 1, 0, 0],
 [0, 0, 1, 0]]
```

Task 1: A Nonogram Verifier (10%)

Define a Prolog predicate `is_nonogram(Puzzle,Solution)` with mode `is_nonogram(+,+)`, which given a nonogram puzzle `Puzzle` and a Prolog term `Solution`, verifies that `Solution` is a solution to the nonogram puzzle described by `Puzzle`.

For example:

```
?- Puzzle   = nonogram(4, 4,
                       [[3],[1,1],[1,1],[2]],
                       [[4],[1,1],[2],[1]]),
   Solution = [[1, 1, 1, 1],
               [1, 0, 0, 1],
```

```
               [1, 1, 0, 0],
               [0, 0, 1, 0]],
   is_nonogram(Puzzle, Solution).
true.

?- Puzzle   = nonogram(4, 4,
                       [[3],[1,1],[1,1],[2]],
                       [[4],[1,1],[2],[1]]),
   Solution = [[1, 1, 1, 1],
               [1, 0, 0, 1],
               [1, 1, 0, 0],
               [1, 0, 1, 0]],
   is_nonogram(Puzzle, Solution).
false.
```

Task 2: A Nonogram Solver (25%)

Define a Prolog predicate `nonogram_solve(Puzzle, Solution)` with mode `nonogram_solve(+,-)` to backtrack over all solutions for a nonogram puzzle.

Examples:

```
?- nonogram_solve(nonogram(4,4,
                  [[3],[1,1],[1,1],[2]],
                  [[4],[1,1],[2],[1]]),
         BoardByRows).
BoardByRows = [[1, 1, 1, 1],
               [1, 0, 0, 1],
               [1, 1, 0, 0],
               [0, 0, 1, 0]] ;
false.

?- nonogram_solve(nonogram(8,7,
                  [[2],[1,1],[2],[2,4],[1,1,2],[1,1,1,1],[2,2]],
                  [[2],[1,1],[1,1],[2],[2,1],[1,2,2],[4,1],[3]]),
         BoardByRows).
BoardByRows = [[0, 0, 0, 1, 1, 0, 0],
               [0, 0, 0, 1, 0, 1, 0],
               [0, 0, 0, 0, 1, 0, 1],
               [0, 0, 0, 0, 0, 1, 1],
               [1, 1, 0, 1, 0, 0, 0],
               [1, 0, 1, 1, 0, 1, 1],
               [0, 1, 1, 1, 1, 0, 1],
               [0, 0, 0, 1, 1, 1, 0]] ;
BoardByRows = [[0, 0, 0, 0, 1, 1, 0],
```

```
                [0, 0, 0, 1, 0, 0, 1],
                [0, 0, 0, 1, 0, 0, 1],
                [0, 0, 0, 0, 1, 1, 0],
                [1, 1, 0, 1, 0, 0, 0],
                [1, 0, 1, 1, 0, 1, 1],
                [0, 1, 1, 1, 1, 0, 1],
                [0, 0, 0, 1, 1, 1, 0]] ;
false.
```

A few notes:

- Consider the following description for rows (or columns): `[[2,3],[],[2,2,1]]`. It means that the first row has two blocks (with lengths 2 and 3), the second row has no blocks (all squares are white), and the third row has 3 blocks (with lengths 2,2, and 1).
- If you want to read more about Nonograms and solving techniques, a good place to start is wikipedia: `http://en.wikipedia.org/wiki/Nonogram`.
- You can find a file with a collection of puzzles linked here: `http://www.cs.bgu.ac.il/~lp182/Assignments`
- Your solver will be graded as follows: 60% for a simple but correct solver, 20% additional points for a solver which can solve (most) 10x10 puzzles in under 10 seconds, and 10% points extra for an advanced solver that can solve some of the harder puzzles. The remaining 10% points will be allocated for the design, documentation and elegance of your submission.

Part 2: Ramsey Graphs (Brute Force)

A Ramsey $(s,t;n)$-graph is an edge coloring of the complete graph with n vertices using two colors such that there is no clique of size s in the first color and no clique of size t in the second color. The Ramsey number $R(s,t)$ is the smallest number n for which there exists no Ramsey $(s,t;n)$-graph. In this assignment we will look for Ramsey $(s,t;n)$-graphs. Several values of $R(s,t)$ are known, for example: $R(3,3)=5$, $R(3,4)=9$, and $R(4,4)=18$ so we can hope to find a Ramsey $(3,3;5)$-graph, a Ramsey $(3,4;8)$-graph, and a Ramsey $(4,4;17)$-graph. A complete list of known bounds of Ramsey $(s,t;n)$ graphs can be found here: `https://en.wikipedia.org/wiki/Ramsey's_theorem#Ramsey_numbers`

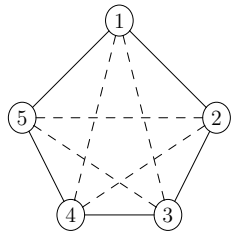

Figure 1: A Ramsey $(3,3;5)$ coloring in two colors: solid lines and dashed.

Your task is to compute the largest Ramsey $(s,t;n)$-graphs (largest values of n) you can. You will represent the graph as an adjacency matrix where the value 0 indicates the first color and the value 1 indicates the second color.

Task 3: A Ramsey Verifier (10%)

Write a prolog predicate `verify_ramsey(Instance, Solution, CounterExample)`, with mode `verify_ramsey(+,+,-)` which unifies `CounterExample` with a list of vertices in `Solution` that form a monochromatic clique (if one exists) or with the constant `ramsey` if `Solution` is a legal Ramsey coloring (corresponding to the Instance=r(S,T,N)).

```
?- Instance = r(3,3,5),
   Solution = [[0,0,1,1,0],
               [0,0,0,1,1],
               [1,0,0,0,0],
               [1,1,0,0,0],
               [0,1,0,0,0]],
   verify_ramsey(Instance, Solution, Counter).

Counter = [3,4,5]


?- Instance = r(3,3,5),
   Solution = [[0,0,1,0,1],
               [0,0,1,1,0],
               [1,1,0,0,0],
               [0,1,0,0,1],
               [1,0,0,1,0]],
   verify_ramsey(Instance, Solution, Counter).

Counter = ramsey
```

Task 4: A Ramsey (Brute Force) Solver (25%)

You are to write a Prolog predicate `find_ramsey(Instance, Solution)` with mode `find_ramsey(+, -)`, which takes an instance of the Ramsey $(s,t;n)$ problem `Instance = r(S,T,N)` and unifies `Solution` with a graph represented as an adjacency matrix such that `Solution` is a Ramsey $(S,T;N)$-graph. If no solution exists the predicate should fail. You may assume that $N > 0$ and that $S, T > 0$.

```
?- Instance = r(3,3,5),
   find_ramsey(Instance, Solution).

Solution = [[0,0,1,0,1],
```

```
            [0,0,1,1,0],
            [1,1,0,0,0],
            [0,1,0,0,1],
            [1,0,0,1,0]]

?- Instance = r(3,3,6),
   find_ramsey(Instance, Solution).

false.
```

Part 3: Ramsey Graphs (SAT)

Task 5: CNF Encoding (15%)

Write a Prolog predicate `encode_ramsey(Instance, Map, CNF)` with mode `encode_ramsey(+,-,-)`. The predicate takes an `Instance` = `r(S,T,N)` and encodes it to a `CNF` formula which is satisfiable if and only if the $N \times N$ adjacency matrix (of Boolean variables) specified in `Map` has a Ramsey $(S, T; N)$ coloring.

Task 6: Decode (10%)

Write a Prolog predicate `decode_ramsey(Map,Solution)` with mode `decode_ramsey(+,-)`, which decodes the `Map` of Boolean variables generated by `encode/3` to a more readable format (i.e., using 0,1).

Task 7: Solve Ramsey with SAT Solver (5%)

Write a Prolog predicate `solve_ramsey(Instance,Solution)` with mode `solve_ramsey(+,-)` which takes a Ramsey instance, encodes it, and calls a SAT solver. If the CNF is satisfiable, your predicate should decode the solution, and verify its correctness. If the CNF is unsatisfiable you must unify `Solution` with the atom `unsat`. For example:

```
?- solve_ramsey(r(3,3,5), Solution).
Solution = [[...], ...]

?- solve_ramsey(r(3,3,6), Solution).
Solution = unsat
```

SAT Solvers

Starting with this assignment we will use a SAT Solver. The course website contains a section dedicated to the installation and configuration of SAT Solvers.

Initially, you may use the naive SAT Solver which is provided at the SAT Solvers page of the course website. This SAT Solver is written in Prolog, and is relatively weak. It will not handle the more complex cases you are required to solve. Therefore, once you feel more confident in your work, you should switch to an industrial SAT Solver. We will test your work assuming you used an industrial SAT Solver!

Grading & Procedures

After Solving :

When grading your work, an emphasis will be given on code efficiency and readability. We appreciate effective code writing. The easier it is to read your code — the more we appreciate it! Even if you submit the wrong answer. So please indent your code, add good comments.

Procedure

Submit a single file called `ex2.zip`, which must contain the file `ex2.pl` with the assignment's solution. You may also include in `ex2.zip` a PDF file detailing anything you wish to expand on regarding your solution: state briefly any cool ideas you used, provide references, and say a few words on your approach. Make sure that your code is well documented and clear. Pay attention to style (we will when we check your code!). Credit will be given to elegant code writing. Do not use constructs that we did not yet cover in class (no cuts, no findall, no negation, no if-then-else, no external libraries that were not seen in class). You may work in pairs (on condition that the work division is more or less 50:50). Submission is via the submission system. If you take any parts of your solution from an external source you must acknowledge this source in the comments (at file's start). The file `bonus1.pdf` should also include the following statement, filled to reflect your details:

> We, Name1 (ID number1) and Name2 (ID number2), assert that the work we submitted is entirely or own. We have not received any part from any other student in the class (or other source), nor did we give parts of it for use to others. We have clearly marked in the comments of our program any code taken from an external source. We assert that the division of work between us is: Name1 X% and Name2 (100-X)%.

Please note that we test your work using a Linux installed SWI-Prolog (as in the CS Labs) – so please make sure your assignment runs on such a configuration.