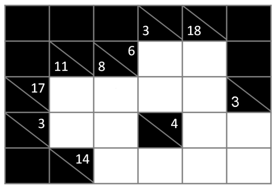
Ben-Gurion University of the Negev
Department of Computer Science
Logic programming

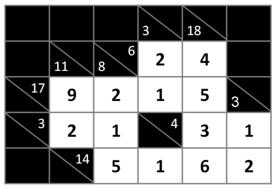
Assignment 3, Semester B, 2022

Deadline: June 28 at 23:00

Kakuro with BEE

In the game of Kakuro you are required to fill horizontal and vertical blocks with distinct integers between 1 and 9 which sum to a given clue. (you can read more about the game of Kakuro on Wikipedia) and here is an example Kakuro board (left) with its solution (right):





We represent Kakuro instances as a list of blocks. Each element of the list is of the form `ClueSum = Block`, where `ClueSum` is an integer and `Block` is a list of the block's variables. Below is our representation for the Kakuro board in the previous example:

$$\left[\begin{array}{l} \mathtt{14} = [\mathtt{I}_{11}, \mathtt{I}_{12}, \mathtt{I}_{13}, \mathtt{I}_{14}],\ \mathtt{17} = [\mathtt{I}_3, \mathtt{I}_4, \mathtt{I}_5, \mathtt{I}_6],\ \mathtt{3} = [\mathtt{I}_7, \mathtt{I}_8],\ \mathtt{4} = [\mathtt{I}_9, \mathtt{I}_{10}],\ \mathtt{11} = [\mathtt{I}_3, \mathtt{I}_7], \\ \mathtt{6} = [\mathtt{I}_1, \mathtt{I}_2],\ \mathtt{8} = [\mathtt{I}_4, \mathtt{I}_8, \mathtt{I}_{11}],\ \mathtt{3} = [\mathtt{I}_1, \mathtt{I}_5],\ \mathtt{18} = [\mathtt{I}_2, \mathtt{I}_6, \mathtt{I}_9, \mathtt{I}_{13}],\ \mathtt{3} = [\mathtt{I}_{10}, \mathtt{I}_{14}] \end{array}\right]$$

A legal Kakuro solution is a list of terms as described above, which contains distinct integers between 1 and 9 instead of variables, such that each block's sum is equal to the clue. Below is our representation for the solution of the Kakuro board in the previous example:

$$\left[\begin{array}{l} \mathtt{14} = [5,1,6,2],\ \mathtt{17} = [9,2,1,5],\ \mathtt{3} = [2,1],\ \mathtt{4} = [3,1],\ \mathtt{11} = [9,2], \\ \mathtt{6} = [2,4],\ \mathtt{8} = [2,1,5],\ \mathtt{3} = [\mathtt{I}_1, \mathtt{I}_5],\ \mathtt{18} = [4,5,3,6],\ \mathtt{3} = [1,2] \end{array}\right]$$

Task 1: kakuroVerify (10 %)

Write a Prolog predicate `kakuroVerify(Instance, Solution)` with mode `kakuroVerify(+,+)`, which succeeds if and only if its first argument represents a legal Kakuro instance, and its second argument represents a legal solution to that instance.

Task 2: kakuroEncode (25 %)

Write a Prolog predicate `kakuroEncode(Instance,Map,Constraints)`. The predicate has mode `kakuroEncode(+,-,-)`. The predicate takes an instance of Kakuro, and encodes it to a set of BEE constraints `Constraints` which is satisfiable if and only if the mapping of BEE integer variables specified in `Map` is a solution to the Kakuro instance.

Task 3: kakuroDecode (5 %)

Write a Prolog predicate `kakuroDecode(Map,Solution)` with mode `kakuroDecode(+,-)`, which decodes the `Map` of variables generated by `kakuroEncode/3` to a Kakuro solution.

Task 4: kakuroSolve (10 %)

Write a Prolog predicate `kakuroSolve(Instance,Solution)`, with mode `kakuroSolve(+,-)` that takes a Kakuro instance, solves it using `BEE`. When solving you must encode the instance to `BEE` constraints, decode the solution and verify it. We recommend you use the `BEE` framework to implement `kakuroSolve`

Killer Sudoku with `BEE`

A *Killer Sudoku* is a 9×9 Sudoku puzzle. In this type of puzzle one must fill the values of cells according to the following constraints

1. Each row, column and box consist of all-different values (between 1 and 9).
2. Any two cells separated by a Knight's move (the chess-piece) must be different.
3. Any two cells separated by a King's move (the chess-piece) must be different.
4. The values of any two cells which appear next to each other in a row, or column must have absolute difference of at least 2.

For example, the following are a Killer Sudoku puzzle and its solution.

```
[[_, _, _, _, _, _, _, _, _],        [[4, 8, 3, 7, 2, 6, 1, 5, 9],
 [_, _, _, _, _, _, _, _, _],         [7, 2, 6, 1, 5, 9, 4, 8, 3],
 [_, _, _, _, _, _, _, _, _],         [1, 5, 9, 4, 8, 3, 7, 2, 6],
 [_, _, _, _, 6, _, _, _, _],         [8, 3, 7, 2, 6, 1, 5, 9, 4],
 [_, _, 1, _, _, _, _, _, _],         [2, 6, 1, 5, 9, 4, 8, 3, 7],
 [_, _, _, _, _, _, 2, _, _],         [5, 9, 4, 8, 3, 7, 2, 6, 1],
 [_, _, _, _, _, _, _, _, 8],         [3, 7, 2, 6, 1, 5, 9, 4, 8],
 [_, _, _, _, _, _, _, _, _],         [6, 1, 5, 9, 4, 8, 3, 7, 2],
 [_, _, _, _, _, _, _, _, _]]         [9, 4, 8, 3, 7, 2, 6, 1, 5]]
```

A Killer Sudoku puzzle is represented by a list of assignment constraints (hints). The above puzzle is represented by the term:

```
Instance = killer([cell(5,3) = 1, cell(6,7) = 2, cell(4,5) = 6, cell(7,9) = 8])
```

A solution is represented as a list of assignment constraints:

```
Solution = [cell(1,1) = 4, cell(1,2) = 8, cell(1,3) = 3, ..., cell(9,9) = 5]
```

Notice that (like a normal Sudoku puzzle) a Killer Sudoku puzzle is *legal* only in case it has exactly one solution.

Task 5: Killer Sudoku Verifier (10%)

Write a Prolog predicate `verify_killer(Instance,Solution)` with mode `verify_killer(+,+)` which succeeds if and only if its second argument `Solution` represents a legal Killer Sudoku solution for the instance `Instance` in its first argument. Otherwise, `verify_killer` should fail. The instance is represented as a term `Instance=killer(Hints)` where `Hints` is a list of assignment constraints (hints) and the solution is represented as a list with 81 assignment constraints, one for each cell of the 9×9 board. This predicate does not verify the uniqueness of the `Solution`, only that it assigns valid values to the `Instance` cells. This predicate must also be deterministic. In this task you should assume that the given `Instance` is a legal Killer Sudoku instance. For example,

```
?- Instance = killer([cell(5,3) = 1, cell(6,7) = 2, cell(4,5) = 6, cell(7,9) = 8]),
   Solution = [cell(1,1) = 4,cell(1,2) = 8,...,cell(9,9) = 5], % full solution above
   verify_killer(Instance, Solution).
true.

?- Instance = killer([cell(5,3) = 1, cell(6,7) = 2, cell(4,5) = 6, cell(7,9) = 8]),
   Solution = [cell(1,1) = 8, ..., cell(2,3) = 8, ...],
   verify_killer(Instance, Solution).
false.

?- Instance = killer([cell(5,3) = 1, cell(6,7) = 2, cell(4,5) = 6, cell(7,9) = 8]),
   Solution = [cell(1,1) = 8, ..., cell(2,1) = 9, ...],
   verify_killer(Instance, Solution).
false.
```

Task 6: Killer Sudoku Encoder (20%)

Write a Prolog predicate `encode_killer(Instance, Map, Constraints)` with mode `encode_killer(+, -, -)` which given a Killer Sudoku instance which is represented by a list of hints (assignment constraints) `Instance = killer(Hints)` unifies `Map` with a representation of the instance as a list of 81 terms of the form `cell(I,J) = Value`, where `I` and `J` are Prolog integers representing the index of the cell, and `Value` represents a `BEE` integer between 1 and 9 encoded in `Constraints`. A call to the predicate instantiates `Constraints` with a set of `BEE` constraints such that the satisfying assignments for `Constraints` correspond to solutions of the Killer Sudoku instance `Instance` and bind the variables in `Map` to represent the corresponding integer values. In this task you should assume that the given `Instance` is a legal Killer Sudoku instance.

Task 7: Killer Sudoku Decoder (10%)

Consider the following predicate to solve Killer Sudoku puzzles using a sat solver.

```
solve_killer(Instance, Solution) :-
    runExpr(Instance, Map, encode_killer, decode_killer, verify_killer).
```

The call to `runExpr/5` should succeed and bind the variables in `Map` to a representation of integer values (using the representation of `BEE`).
Write the predicate `decode_killer(Map,Solution)` with mode `decode_killer(+, -)` which creates from the given `Map` a corresponding solution in the format which is a list of (81) assignment constraints.

Task 8: Legal Killer Sudoku Instance (10%)

Write a Prolog predicate `all_killer(Instance, Solutions)` with mode `all_killer(+, -)` which given an instance of a Killer Sudoku puzzle unifies `Solutions` with a list of all possible solutions to `Instance`. This predicate must be deterministic. If `Instance` is a legal Killer Sudoku instance then `Solutions` will contain exactly one solution.

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 a partial answer. So please indent your code, add good comments.

Procedure

Submit a single file called `ex3.pl` with the assignment's solution. Please include a header with following statement:

> /**** I, Name (ID number) assert that the work I submitted is entirely my own. I have not received any part from any other student in the class (or other source), nor did I give parts of it for use to others. I have clearly marked in the comments of my program any code taken from an external source. *****/

Submission is solo, i.e., you may *not* work in pairs. If you take any parts of your solution from an external source you must acknowledge this source in the comments. 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.