CSC322 Spring 2023

Project – SMT-based KenKen Solving

In this project, you will write a simple program to translate partially solved KenKen puzzles into SMT-LIB format, which can be passed to a SMT solver such at `mathsat` to generate solutions. As an additional task, you can write a prettyprinter for puzzles (unsolved and solved).
For this project, we will only work with 7x7 puzzles.

You should work in groups, preferably of 3 members (but groups of 2 or 4 are OK.) Only one submission per group is required. You may use any language you want for your implementation, but it should be possible for the grader to run and test your code on `linux.csc.uvic.ca`, *without installing any new software.* The `mathsat` system is available on `linux.csc.uvic.ca`. Documentation for `mathsat` may be found at

https://mathsat.fbk.eu/documentation.html

All the KenKen puzzles we are using are from

https://www.kenkenpuzzle.com

Basic Task

To complete the basic task, you must write code to implement *two programs (commands)*

- `kenken2smt` reads a **single** KenKen puzzle puzzle (in a format specified it below) and converts it to SMT-LIB format, as described in the article *Solving Kenken using an SMT (integer) solver.*
- `smt2kenken` converts the output of `mathsat` to a KenKen solution (string of digits.)

Optionally, you may also implement

- `pp` which takes a KenKen puzzle (and possibly a solution) and prints the puzzle (solved puzzle) in a nice format. There is quite a bit of flexibility on how to do this part.

You may use any language to implement your translator as long as we can test it as described below.

Note that the above programs are meant to solve a **single** KenKen instance. You may want to write additional code in order to do testing and performance evaluation. You should at least document your approach to doing this, but we will not evaluate or run any code you have created for testing and evaluation. The grader will only look at your code for `kenken2smt` and `smt2kenken`.

Interfacing with `mathsat`

Your commands should read their input from `STDIN` and write to `STDOUT`. The following session shows how they should work.

```
$ cat 73491-puzzle.txt
#kenken www.kenkenpuzzle.com Puzzle 73491 9x9 Medium
r1.16+,r2.1-,r2,r3.5-,r3,r4.3/,r4,r5.13+,r5
r1,r6.4/,r6,r7.3-,r8.45*,r8,r9.22+,r9,r5
r1,r10.3+,r10,r7,r11.2-,r11,r9,r12.2-,r12
r13.3/,r14.1-,r15.8-,r16.120*,r16,r16,r17.20*,r18.2-,r19.2
r13,r14,r15,r20.2-,r20,r17,r17,r18,r21.17+
r22.9+,r23.63*,r24.1-,r25.5-,r24,r26.1-,r27.48*,r21,r21
r22,r23,r24,r28.5-,r28,r26,r27,r29.4/,r29
r30.3,r31.432*,r32.20+,r32,r33.18+,r34.3+,r34,r35.13+,r36.2-
```

```
r31,r31,r32,r32,r33,r33,r33,r34,r3
$ ./kenken2smt <puzzle.txt >puzzle.smt
$ mathsat <puzzle.smt >model.smt
$ ./smt2kenken <model.stm >solution.smt
$ cat solution.txt
7546193281284957638127469356495381722318645974753826195962738413679512849831274 56
```

Note that after we execute the command

```
$ mathsat <puzzle.smt >model.smt
```

if the file puzzle is solvable, the first line of `model.smt` will just be `sat`. Also, if your input file includes the command

```
(set-option :produce-models true)
```

The file will contain a setting of variables giving a solution to the puzzle, e.g.,

```
( (v0 1)
  (v1 5)
  (v2 7)
  ...
)
```

If `mathsat` is called with the `-stats` flag then following the model there will be a collection of statistics, e.g.,

```
...
;; statistics
(
...
 :time-seconds 0.007
 :memory-mb 16.285
)
```

Note The encoding format we will be using for puzzles differs slightly from that given in the article. You can assume that regions are listed in row-major order and are numbered consecutively `r1,r2,...`. As in the article, the initial occurrence of a cell for a region will also contain an arithmetic constraint, e.g., `r3.15+` means that the entries in region `r3` must add up to `15`. For a single-cell region, there is a value but not operation, e.g., `r4.5` means region `r4` contains the value `5`.

The encoding makes it particularly simple to do the translation – you can keep two counters, a region counter and a cell counter. The region counter can be used to index into an array. Each array entry should contain a list with the operation, and the cell numbers of the cells in that region. There should be one first-order variable for each cell.

This is simpler than the problem of encoding Sudoku puzzles in cnf. The basic constraints you need are just range constraints for the variables (i.e., every variable must be between 1 and 7) as well as uniqueness constraints for each row and colum. The arithmetic constraints can converted almost directly from the entries of the array described above.

You should test your commands on the files provided in the `.zip` file provided with the project. This file contains a directory structure which should be self-explanatory. At the top level, puzzles are organized by the operations allowed in the puzzle. You produce a report which summarizes the results of your test, based on the statistics provided by If `mathsat` called with the `-stats` flag. Give average and worst-case statistics. A good report will organize the statistics according to allowed operations and puzzle difficulty. *It is up to you* *to do some reading to understand the statistics provided. Of course you don't have to understand everything, but include anything that you understand an think is interesting.* Note that you will need to write a testing harness that interfaces with your commands and with `mathsat` to do the testing. You do not need to include the code for this testing harness in your submission.

Prettyprinter

There is only one extended task, the prettyprinter. This is worth 10% of the grade. You are pretty much on your own here, but you can use the provided script `fetch.sh` to get a JSON file which gives information that can be used to print a puzzle. The format of the file is roughly as follows:

The puzzle data (i.e., not the difficulty/id/etc metadata used by the website) is stored in the `.data` field of the top-level JSON object, which is a string consisting of five parts: `A`, `T`, `S`,`V`, and `H` (in that order), which are abbreviations for "answer", "target", "symbol", "vertical", and "horizontal", respectively. Each of these parts are matrices with rows separated by vertical whitespace and columns separated by horizontal whitespace. The parts themselves are separated from their labels and from each other by vertical whitespace. The amount of whitespace used is not consistent. The `A`, `T` and `S` parts are size*size matrices consisting of decimal numbers and, in the case of `S`, the symbols `+-*/`. In `T` and `S`, `0` is used as a dummy element for cells which are not the top-left cell of their region. For "freebie" cells (i.e., cells which are the only cell in their region), `S` contains a `1`. On the other hand, `V` and `H` are size*(size-1) `0,1` -matrices. Note that `H` is transposed (i.e., the values along the ith row of `H` tell you where to put lines in the ith column of the grid).

Deliverables and Detailed Grade Breakdown

Your submission should include

1. Your code, with documentation on how to use it. Your code should produce two *Linux executables*: `kenken2smt` reads a *single Sudoku* description from `STDIN` and writes a CNF description to `STDOUT`, and `smt2kenken` reads the output produced by `mathsat` for a single puzzle instance from `STDIN` and writes puzzle solution (string of 49 digits) `STDOUT`. If you include a prettyprinter, give detailed instructions on how to use it

2. A `README.md` file describing the entire contents of the submission as well as any details you feel are relevant. *Make sure you put the name and Student ID of all group members here.*

3. A report giving background, anything to know about your implementation, and any test results obtained as described above. If you implement the prettyprinter, include some examples of unsolved puzzles and their solved versions.

Submit everything as a single `.tar.gz` file. The name of the file should be the Brightspace ID of the group member who submits the file (only one submission per group is required.) The submitted file should extract to a single directory with the same name as the `.tar.gz` file. More specifically, to create the submission, you should be in a directory which contains the directory `subid`, and execute the following:

```
tar cvzf subid.tar.gz subid
```

where `subid` is the Brightspace ID of the submitter, as described above.)

The grader will test your submission as follows. After executing `tar xvzf subid.tar.gz; cd subid`

- If your executables are non-binary (e.g., shell scripts) the grader should find them at the top level of the directory; otherwise

- The grader should be able to execute `make clean`, followed by `make target` to create the executables. If you are not able to use `make` you should give clear instructions on how to build your commands in the `README`

The grader will then try your code on a number of examples.

DO NOT submit any executable of `mathsat`. **DO NOT** submit *binary* executables for your solutions. If the source files need to be compiled, include a `MAKEFILE` as described above.

You only need to provide one submission for your group. The breakdown for the code grade: (3) for

Basic Task	
Code	8
Report (general)	2.5
Report (performance evaluation)	2
`README`	1
Prettyprinter	1.5

Table 1: Grade Breakdown

following the specification for two executable files with the required input/output behaviour as described in the **Interfacing** section above. (3) for correctness – this will be determined by testing your commands on some subset of the sample puzzles from the test directory. (2) for clarity, style and economy of code.