Logic and Computability SS24, Programming Assignment

Due: 05. 06. 2024, 23:59

In order to get started with the programming exercises, install the `z3-solver` package via pip:

- `pip install z3-solver`

If you have issues with the installation please, use the discord channel to ask for help.

We will provide you with the skeletons for the programming exercises so that you only have to do the SMT encoding and testing. Please do not change major parts within those skeletons without consulting your tutor first.

1. Start by cloning the following repository:
   `git clone https://git.pranger.xyz/sp/LAC-Practical-Assignments-2024 --origin upstream`

2. The submission will be handled via your personal submission repository. Add the remote repository with the following command:

   `git remote add origin https://git.teaching.iaik.tugraz.at/lc24/lc24gXX.git`

3. After you have finished solving the exercises, commit your changes and push them to your submission repository:

   `git push origin main`

   You do not need to tag your final commits. We will check out your last commit before the deadline.

4. Additionally to uploading your solutions via git **you need to tick** which exercises you have solved in the TeachCenter course.

You may work in groups of 2 students for this part of the practicals. If you work in pairs, we still ask you to **upload the solution to both of your personal repositories**. Additionally, please enter your details in the README of both repositories

---

1. [3 points] **Burglars**

   We have given the statements of three suspects:

   (a) Ed: "Fred did it, and Ted is innocent."

   (b) Fred: "If Ed is guilty, then so is Ted."

   (c) Ted: "I'm innocent, but at least one of the others is guilty."

   Your task is to use z3 and the provided skeleton file `burglars.py` to find out who is guilty and who is not.

2. [6 points] **Branchless Minimum.** We can use a bithack to compute the minimum of two integers by directly applying the result of the comparison, i.e. `slt tmp,rx,ry`.

   In order to verify this approach, we need to prove that the following statements compute the minimum of x and y.

   ```
   tmp = x < y ? 0x1 : 0x0      # slt tmp,rx,ry
   min = y ^ ((x ^ y) & -tmp)
   ```

   Note: This is a unary minus, not a bitwise toggle!

   Your task is to declare necessary z3 variables and add the needed constraints to the solver such that it checks for equivalence.

   The source code skeleton for this exercise can be found in `branchless_min.py`.

3. [8 points] **Magic Square.** For this exercise, we will have a look at the classic pen and paper puzzle called magic square. The premise of the game is very intuitive. You are given a grid of numbers, where some cells are not filled out. Your goal is to determine the missing numbers so that the sum in each *row*, *column*, and *diagonal* is the same.

   The magic square is read from a file, where the numbers are separated by spaces and unknown cells are marked with "_". For the input file '`test0.txt`' the resulting magic square is shown in Figure 1.

   Note that the size of the grid is not fixed but may vary, whilst always going to be square.

		28
	35	21
	7	

Figure 1: Example of a magic square.

   Our goal is then to find the missing cells and fill out the square. This is where z3 can help us! An example solution is shown in Figure 2.

14	63	28
49	35	21
42	7	56

Figure 2: Solution of the example magic square.

   The template handles the parsing of the input file, and the printing of the solved square. The source code skeleton for this exercise can be found in `square.py`. Your task is to add the needed constraints, such that Z3 provides a solution via the found satisfying model.

   To run your solver for an input you can simply pass it as an argument: `python3 square.py square0.txt`

4. [8 points] *Tents and Trees.* In this task you have to implement the contraints present in the Tents and Trees Puzzle[1]. In this puzzle one has to put a tent next to each tree on a $(n \times m)$ grid such that each tree has a tent either horizontally or vertically adjacent (not diagonally). Furthermore, no tent should be adjacent to another tent, not even diagonally. Numbers on the side of the grid indicate the required number of tents in a given row or column.

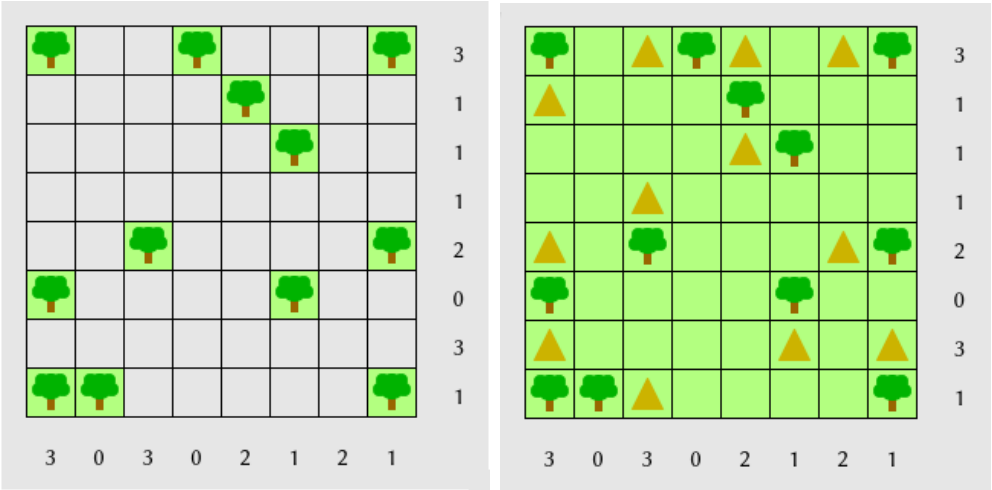

Figure 3: Example of a Tents and Trees Puzzle and its solution.

The skeleton file for this problem prepares the parsing and some helper functions that determine neighbours of a given cell. The source code skeleton for this exercise can be found in `trees_and_tents.py`.

To run your solver for an input you can simply pass it as an argument: `python3 trees_and_tents.py forest0.txt`

[1] `https://www.chiark.greenend.org.uk/~sgtatham/puzzles/js/tents.html`

5. [10 points] [BONUS] **Rock-Paper-Scissors-Spock-Lizard** For this task you need to model the computation of a *linear-congruential generator*. If you want to read more about linear-congruential generators, please consult the wikipedia. Our opponent, the computer, computes its choice by first computing a pseudo-random number s_i that is based on the previous result s_{i-1}:

$$s_i = 11 \cdot s_{i-1} + 12345 \ \& \ 0\text{x}7\text{fff}. \tag{1}$$

Its choice c_i is then computed with the remainder modulo 5:

$$c_i = s_i \mod 5, \tag{2}$$

where

- $c_i = 0$ means Rock,
- $c_i = 1$ means Paper,
- $c_i = 2$ means Scissors,
- $c_i = 3$ means Spock, and
- $c_i = 4$ means Lizard.

Your task is to model the computation of each s_i, starting from s_1, and use your knowledge about the computer's choice to tell Z3 that $c_i == s_i \mod 5$. Note that you do not know the value of s_0 in this scenario. (The problem would become trivial if you would have that information!)

The skeleton code can be found in `rpssl.py`. The snippet contains a class `RPSSLComputer` that reads a seed value (`s_0`) and continuously returns the next computer's choice by calling `compute_choice()`.

You execute the script by passing it a value for the seed:

```
python3 rpssl.py <seed>.
```

After you have implemented the correct constraints, your script will win continuously after a few rounds, independently of the seed value.

Implementation

The snippet simulates the game against the computer. For all rounds $j \in \{1, \ldots, i\}$ that you have already played you can give Z3 the information that s_j has been computed via Eq. 1 and the computer's choice c_j via Eq. 2. Adding a constraint for Eq. 1 for the current round $i + 1$ will make Z3 compute a good guess for s_{i+1}. You can use this result to come up with a winning move against the computer.

The script starts with querying the computer for `preprocess_count` many choices and calls a function (`add_constraint(…)`) where you add the appropriate constraints to the solver. These constraints will describe the relationship between s_i and s_{i-1} and the relationship between s_i, c_i and the computer's choice.

After this preprocessing is done, you are going to use Z3 to compute potential good choices for the current round $i + 1$ in the game. In order to ask Z3 for a good choice you will model the computation that the computer will make for s_{i+1} and add this to the solver in the function `add_next_state_constraint(…)`.

You can then use the resulting value of s_{i+1} from the model and compute the remainder modulo 5 to beat the computer. Implement this as the return value of `get_players_choice(…)` using the `winning_mapping`.

The special variable `s_i_plus_1` will always hold the value for the current round. After you have extracted the value for s_{i+1} for the current round, you need to remove the constraint for this variable and add the constraints about the round that you have already played, using `add_constraint(…)`. Note that *this loop is already implemented for you* and uses `store_backtracking_point(…)` and `restore_backtracking_point(…)` to correctly add and remove the constraints for `s_i_plus_1`.

You are going to solve the task by filling in the function bodies of three different functions:

(a) `add_constraint(…)`
In this function we give Z3 our knowledge about the rounds we have played so far.
   i. Create a Z3 BitVector that stores 16 bits and append this to the `states` list.
   ii. Enforce that the newly added state evaluates to the LCG computation (1) of the previous state.
   iii. Enforce that the unsigned remainder modulo 5 of the newly added state is equal to the computer's choice.
   Hint: Use the built-in function URem(s_i, 5) to model the remainder.

(b) `add_next_state_constraint(…)`
In this function we tell Z3 that the computer will compute s_{i+1} via 1:
   i. Enforce that `s_i_plus_1` evaluates to the LCG computation (1) of the previous state.

(c) `get_players_choice(…)`

i. Use the model that the solver came up with and `winning_mapping` to decide the your choice for the next round of the game.