Program Verification 2022
Project 1

March 30, 2022

1 General Information

- This is the first graded project of the Program Verification course, which will count for 30% of the final grade.
- You can earn up to 50 points in total: 36 for the quality of the submitted solution and 14 for the final presentation and Q&A which will take place at the end of the semester for both projects.
- *The project is to be completed individually by each student participating in the course.*
- The submission deadline is 23:59 (CEST) on April 27, 2022.
- Skeleton files for this project are available on the course website[1].
- Submit your solution by sending an e-mail with the subject "[PV] Submission Project 1" containing a single `.zip` file with the completed skeleton files to `aurel.bily@inf.ethz.ch`.

2 Tool

Tasks 2–5 are to be completed using Viper. We strongly recommend using Viper via the VS Code Viper IDE extension[2]. Your submission must work with the Viper and Z3 versions distributed with the current version of Viper IDE. In general, your solutions should work with at least one Viper backend (Silicon or Carbon).

For detailed information about Viper's assertions and expressions, we recommend the Viper tutorial[3].

[1] `https://www.pm.inf.ethz.ch/education/courses/program-verification.html`
[2] `https://www.pm.inf.ethz.ch/research/viper/downloads.html`
[3] `https://viper.ethz.ch/tutorial/`

3 Tasks

Task 1: SMT/Akari (8 points)

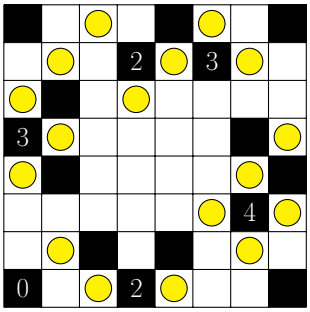
We consider a puzzle game called "Akari" or "Light Up". The rules are as follows:

- This game is played on a rectangular grid of walls and empty tiles.
- Lightbulbs must be placed on some of the empty tiles until all non-wall tiles are lit.
- Lightbulbs shine light in the 4 cardinal directions until the light is stopped by a wall tile.
- Two lightbulbs are not allowed to shine light on each other.
- Some wall tiles have a number which indicates how many lightbulbs must be placed in the adjacent empty tiles (between 0 and 4).

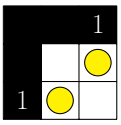
Your task is to implement a solver for Akari puzzles using Python/Z3. In the skeleton file `task1/akari.py`, you will find:

- An explanation of how grids are represented.
- A set of 7 puzzle grids, `GRID_A` through `GRID_G`, visualised on page 3.
- An explanation of how solutions are represented.
- Solutions to the first 3 puzzle grids, `SOLN_A` through `SOLN_C`.
- The function `validate_solution` which accepts a grid and a solution and indicates whether the solution is valid.
- An unimplemented `solve` function.
- A set of assertions checking that the `solve` function does indeed solve the given puzzle grids.

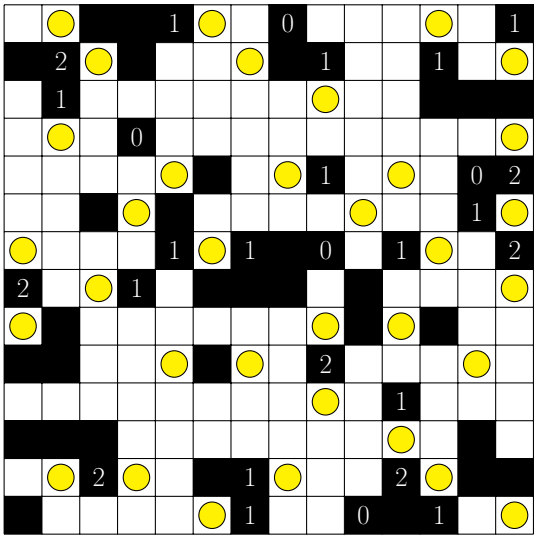
Modify the body of the `solve` function (line 264 in `task1/akari.py`). Your implementation should be capable of solving all of the given grids. The seven given grids each have a unique solution, but in general your implementation should find *a* solution, if there is one, and should return `None` otherwise. Make sure your code is documented well: in particular, explain what each constraint given to the SMT solver means, using inline comments.



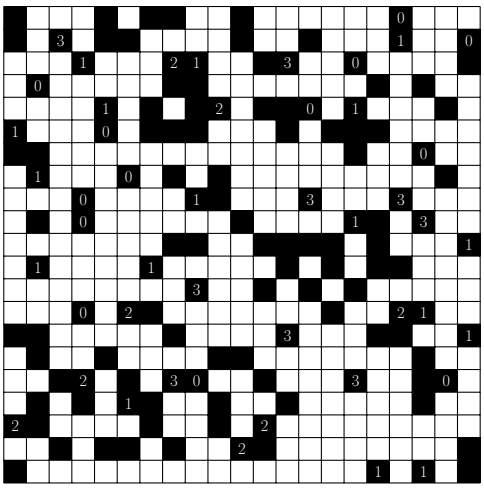
GRID_B, SOLN_B



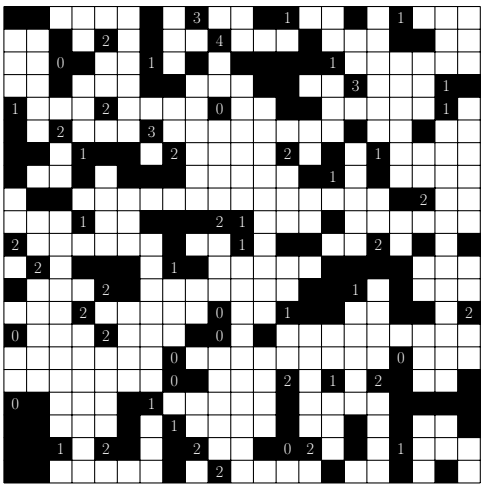
GRID_A, SOLN_A



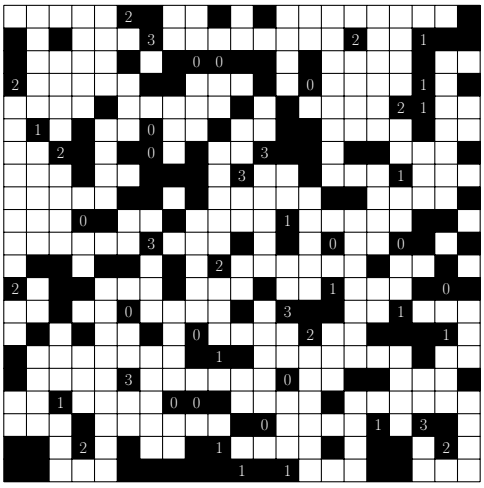
GRID_C, SOLN_C



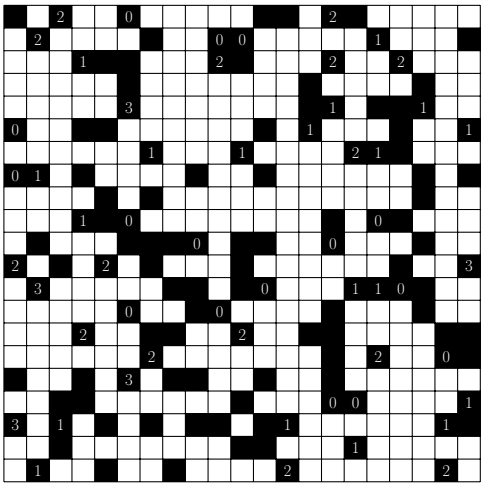
GRID_D



GRID_E



GRID_F



GRID_G

Task 2: Loop invariants (6 points)

In this task, you are given a reference implementation written in Python for two methods with a loop. For each method, translate it to Viper and choose a correct loop invariant such that the postcondition verifies. The skeleton files contain both the reference implementation (`task2/reference.py`) and a Viper template for each method (`task2/pow_fast.vpr`, `next_prime.vpr`).

Faster exponentiation

We consider an exponentiation method similar to the naïve one presented in the second homework assignment. This one is more optimised: it uses exponentiation by squaring and multiplying[4]. The proof of this method relies on the fact that $\texttt{b}^{\texttt{y}} = (\texttt{b}^2)^{\texttt{y}/2}$ for even `y`. The verifier cannot prove this property automatically, so we introduce a lemma using an abstract method:

```
method lemma_pow(b: Int, y: Int)
    requires y >= 0
    requires y % 2 == 0
    ensures math_pow(b, y) == math_pow(b * b, y / 2)
```

A "call" to this method lets the verifier know that the lemma holds for the given `b` and `y`. Translate the `pow_fast` method to Viper (line 20 in `task2/pow_fast.vpr`) and choose a loop invariant such that the postcondition verifies.

Next prime

In the second homework assignment we also defined an `is_prime` method. We will reintroduce it here as an abstract method and define a new method that should find the next prime *after* the given prime input. Translate the `next_prime` method to Viper (line 19 in `task2/next_prime.vpr`) and choose a loop invariant such that the postcondition verifies.

[4] `https://en.wikipedia.org/wiki/Exponentiation_by_squaring`

Task 3: Flip-flop (8 points)

In this task, we will consider the encoding of a new control flow statement into Viper. This statement, the *flip-flop*, has the following general syntax:

```
flip {
    (flip statements)
} flop {
    (flop statements)
}
```

The first time a flip-flop statement is executed in a given method, the first branch is taken (`(flip statements)` above). The second time, the other branch (`(flop statements)` above). As the flip-flop statement is executed again and again, the branches keep alternating.

Note that a flip-flop statement is useless unless it is placed inside a loop, allowing the statement to be executed multiple times. It is also possible to nest flip-flop statements inside each other.

As an example, this program computes the `n`-th Fibonacci number:

```
a := 0
b := 1
res := a
i := 0
while (i < n) {
    flip {
        a := a + b
        res := b
    } flop {
        b := a + b
        res := a
    }
    i := i + 1
}
```

Design an encoding of flip-flop statements into Viper. Describe this encoding in words in the file `task3/encoding.txt`, with consideration for flip-flop states, nested flip-flops, and interaction with loop invariants.

Apply your encoding to the Viper programs provided. In each program, you should replace the flip-flop statement written in a comment with your chosen encoding, though you are allowed to add more Viper code to the function in general if the encoding relies on it. In each program, also choose a suitable loop invariant to make the postcondition verify.

- `task3/fibonacci.vpr` to compute Fibonacci numbers and
- `task3/mod8.vpr` to (inefficiently) compute a `mod 8` operation.

Task 4: Set operations (8 points)

In this task we will use Viper *sets*. Sets are a built-in type in Viper, just like integers or Booleans. They are equipped with a number of mathematical operators, such as[5]:

- `a union b` representing `a` $\cup$ `b`,
- `a intersection b` representing `a` $\cap$ `b`,
- `a setminus b` representing `a` $\setminus$ `b`, or
- `a in b` representing `a` $\in$ `b`.

Your task is to encode, specify, and verify *iterative* versions of set union, intersection, and difference. These methods should each contain a loop with an invariant summarising the progress thus far.

In the definition of your methods, you will need a way to choose an arbitrary element of a set. Such a method can be implemented using an abstract Viper method `choose` with an appropriate specification.

Note that while specifications (assumptions, assertions, invariants) may use any set operations, only the following may be used within the loop body and loop guard (we consider them *directly executable*):

- choosing an element using the choice operator: `a := choose(b)`,
- checking the set's cardinality: `|a|`,
- checking if an element is contained in a set: `a in b`,
- extending a set by *one* element: `a := a union Set(b)`, and
- removing *one* element from a set: `a := a setminus Set(b)`.

Specify the `choose` operator (line 3 in `task4/set_ops.vpr`), then implement and verify iterative set union (line 13), iterative set intersection (line 26), and iterative set difference (line 39).

[5] A complete reference can be found in the Viper online tutorial: `https://viper.ethz.ch/tutorial/#expressions`

Task 5: Sequence minimum and maximum (6 points)

In this task we will use Viper *sequences*. Like sets, sequences are a built-in type in Viper and are equipped with predefined operators, such as:

- `|a|` representing the length of the sequence.
- `a[b]` representing the `b`-th element of sequence `a` (where the first element is at index `0`),

Your task is to encode, specify, and verify a *recursive* version of a method to compute the minimum and maximum of a given sequence simultaneously (i.e. returning two outputs). Akin to array-based methods in C or Java, the method will always receive the full sequence in addition to two indices: the inclusive start of a range, and the inclusive end of a range. The method should return the minimal and maximal element found in the given range. There are two cases to consider:

- The range spans one element only: the minimum and maximum are that element.
- The range spans more than one element: the method should split the range into two and recurse, then merge the results of the two recursive calls.

Add a specification to the `seq_max` method (line 9 in `task5/seq_max.vpr`). Implement the method recursively (line 11).