CSC410

Assignment 3

Due on November 16, 2023 at 11:59pm

Assignment Format and Guidelines on Submission

This assignment is worth 15% of the total course mark. Submit on Markus and follow these rules:

- This is a group assignment, designed for groups of up to 4 students.
- Each group should submit three files named `q1.py`, `q2.py` and `q3.py`. Graduate students will additionally submit `g.zip`. Note that Markus has been setup to accept exactly those 4 files. All the necessary helper functions should be included in each file for each problem. You should not change any of the names or signatures of the functions already provided in the files, and you are not allowed to import any other package than the ones already imported.
- Your problems (1) and (3) encodings can only use boolean variables. You should not solve the problems algorithmically, which means that the solution of the problem is the one given by the solver.
- Note that this assignment already has a due date that is one week later than the previous two, so that it does not interfere with your break week. But, as a result, there will be **no extensions under any circumstances** for the due date of this one. Consider this a blanket one-week extension for the entire class.

Note that your assignment will be automatically graded. Your functions will be called from another script. If you mess with the signature, the call will fail and the autograder will give you a 0 mark.

The submission system will remain open for 12 hours after the deadline, but there is a penalty deduction formula set in Markus that deducts 4% for every hour of late submission up to 12 hours.

Instructions about Z3 and Python

- You have to use the Python API of Z3 to produce the encoding and call the solver.
- You are not allowed to import any other package than the ones already imported in your python files. The idea is to solve the task using the solver and not through programming in Python.
- For problems (1) and (3), you should only use `Bool` literals in the encoding. More precisely, you can use any variable type in your Python program, but you should only pass variables created via `Bool()` to the solver. This means that you will be using Z3 as a SAT-solver and not in its more expressive capacity as an SMT solver. For problem (2), on the other hand, you should provide a different SMT encoding.

Word of Advice

The goal of this assignment is to use constraint solvers to solve problems. The interface to the solver is minimal and you will need to use only a few functions. The difficulty is not in using the solver, in the sense that you need to know how to do fancy things with it. The main effort goes into devising a correct encoding of each problem in SAT or SMT.

Problem 1 (30 points)

In this problem, you will experiment with giving two different encodings to the same problem, and comparing their performance. The learning objective is to gain an understanding that there are many ways to encode a problem as a constraint solving problem, and some are better than others.

The At-most-k constraint on n ($n > k$) variables $(X_1, \ldots, X_n)$ is noted $\leq_k (X_1, \ldots, X_n)$ and satisfied iff at most k of the $X_1, \ldots, X_n$ variables are true.

Naive encoding Devise a simple encoding for the At-most-k constraint, and implement it in the `naive(literals, k)` function.

- The function should return all the clauses of the encoding that ensures at most `k` literals in the list `literals` can be true.
- You are not allowed to create any new variables for the encoding in this function.

Encoding using a sequential counter Sinz [**?**] introduced an encoding for the At-most-k constraint by encoding a sequential counter that counts the number of X_i that are true. For more details on why this encoding works, you can read the original paper, but this is not required for the assignment.

To encode $\leq_k (X_1, \ldots, X_n)$, this encoding introduces $(n-1)*k$ new variables $\{R_{i,j} \mid 1 \leq i < n, 1 \leq j \leq k\}$. Below is the conjunctive normal form of the encoding:

$$\begin{gathered}
\neg X_1 \vee R_{11} \\
\textstyle\bigwedge_{j=2}^{k} \neg R_{1,j} \\
\textstyle\bigwedge_{i=2}^{n-1} \left((\neg X_i \vee R_{i,1}) \wedge (\neg R_{i-1,1} \vee R_{i,1})\right) \\
\textstyle\bigwedge_{i=2}^{n-1} \bigwedge_{j=2}^{k} (\neg X_i \vee \neg R_{i-1,j-1} \vee R_{i,j}) \wedge (\neg R_{i-1,j} \vee R_{i,j}) \\
\textstyle\bigwedge_{i=2}^{n-1} (\neg X_i \vee \neg R_{i-1,k}) \\
\wedge (\neg X_n \vee \neg R_{n-1,k})
\end{gathered}$$

1. Implement this encoding in the `sequential_counter(literals,k)` function in `q1.py`. The function should return all the clauses of the encoding that ensure at most `k` literals in the list `literals` are true.

2. Compare the performance of the two encodings and write a short paragraph in the comments in `q1.py` to explain which encoding performs better depending on `n` and `k`, if there is one.

   To help you, a small test function has been implemented in `q1.py`. You can execute the script `q1.py` with three arguments: `E` is 0 to use your encoding, `1` to use the sequential encoding. `N` is the number of variables and `K` is the number of variables that have to be set to true ($0 < \texttt{K} < \texttt{N}$).

   ```
   python q1.py E N K
   ```

   If your encoding is correct, the response should be `PASSED in (-)s`, where the "`-` " will be replaced with the running time (in seconds) of the solver to solve the encoded constraints.

   Note that the test function encodes the constraint ensuring exactly k variables are true, not the weaker constraint at-most-k. To do so, it calls your implementation of at-most-k twice, with different arguments.

   Note that the goal of the test function is to encode the constraint that exactly k variables are true. It achieves this by reducing the goal to a combination of results from two calls to your implementation of at-most-k.

Problem 2 (30 points)

Hidato is a famous puzzle game. You may have already played this game on your phone. If not, the time has finally come for you to know about it:

`https://en.wikipedia.org/wiki/Hidato`

Input format The partially completed grid is provided to you with `-` standing for each blank cell to be filled with a number, and `*` for blocked cells (that are not to be filled with a number). Each symbol two symbols is separated by a single space. Each line of the puzzle is written in a separate line of the input file. For example, the input file:

```
* * 7 6 - * *
* * 5 - - * *
31 - - 4 - - 18
- 33 2 12 15 19 -
29 28 1 14 - - -
* * - 24 22 * *
* * 25 - - * *
```

Has the following solution, represented in the same format:

```
* * 7 6 9 * *
* * 5 8 10 * *
31 32 3 4 11 16 18
30 33 2 12 15 19 17
29 28 1 14 13 21 20
* * 27 24 22 * *
* * 25 26 23 * *
```

The function parsing the input grid is provided in `q2.py`.

Task

The goal of this problem is to use the SMT solver to produce a solution for any given instance of the puzzle. You may use the theory of your choice to formulate a Hidato puzzle solver using Z3's Python interface. You will complete the function `solve(grid)` in `q2.py` such that the function answers:

Given a partially completed grid, is it possible to complete the grid obeying the rules of Hidato?

If the answer is no, then simply output `None`. If the answer is yes, then output the completed grid. For your solution, you are **not allowed** to use more variables for z3 (using Int() or Bool()) than there are cells on the grid.

Problem 3 (40 points)

The goal of this problem is to solve Hidato again, but this time without the power of SMT. For this instance, we want to limit ourselves to a SAT solver only. The encoding will naturally be less high level and less intuitive in this case.

The specification of the output is precisely the same as stated in Problem 2, but you will complete the function `solve(grid)` in `q3.py` this time which only permits you to use Boolean variables.

Problem 2 vs Problem 3

You are asked to redo the same problem in two different encodings, precisely so that you get a sense how different the solutions (encodings) can be. The constraints put in place in each case are there to make sure that you do not recycle an idea from one solution to the other, but rather rethink and resolve the problem given the context.

In both instances of Hidato, your wrapper code should have polynomial time complexity on the input size to produce the encoding (which itself should also be polynomial on the input size). Considering that Hidato is known to be NP-Complete, this effectively forces you to use the solver properly since any algorithmic solution to Hidato without the SAT/SMT oracle must have super-polynomial complexity.

The Graduate Problem

In this problem, you will design and implement a simple concolic execution engine for Python, powered by an SMT solver. You will use the concolic execution engine to explore Python functions and compute their functional summaries.

Assume f is a function with input variables $\vec{v_i}$ and output variables $\vec{v_o}$. A *functional summary* for f, denoted by ϕ_f, is a first-order formula over free variables $\vec{v_i}$ and $\vec{v_o}$ such that for every input $\vec{i}$, if $f(\vec{i}) = \vec{o}$ then $\phi_f[\vec{v_i} \leftarrow \vec{i}, \vec{v_o} \leftarrow \vec{o}]$ is satisfiable. The summary ϕ_f is *complete* if the implication holds for the other direction as well.

For example, a complete functional summary ϕ_{max} for the binary function $max(a, b)$ with output r is $(a \geq b \Rightarrow r = a) \wedge (a < b \Rightarrow r = b)$.

Symbolic execution can be used to compute a complete functional summary for a given function by exploring every program path and recording the path condition and the return expression on the path. The starter code in the directory `q4` contains a simple concolic execution engine that supports integer and boolean inputs. As your first step, familiarize yourself with the starter code. To compute a functional summary for a function via concolic execution, call `compute_functional_summary` in `symbolic.py`[1].

Task 1: Equivalence Checker (15 points)

Implement an equivalence checking algorithm (see TODO in `symbolic.py`) that takes two functions f_1 and f_2 as inputs and returns true if and only if f_1 and f_2 are functionally equivalent (i.e., they return the same output when given the same input). You can assume both functions terminate on all inputs, and both have a finite number of program paths.

You are allowed to modify any part of the starter code to complete this task. Once done, uncomment `test_eq()` in `symbolic_test.py` to test.

Hint: Use concolic execution to compute the functional summaries for f_1 and f_2.

Task 2: Extend the engine by supporting lists as inputs (25 points)

Extend the concolic execution engine to support functions with list inputs. Specifically, finish the implementation of the class `SymbolicList` in `symbolic_types.py`. For simplicity, we assume that the length of every input list is bounded by 5[2]. After the extension, the functional summary computed by `compute_functional_summary` should be complete with respect to this bound.

You are allowed to modify any part of the starter code to complete this task. Once done, uncomment `test_list()` in `symbolic_test.py` to test.

[1] An example of its usage is also illustrated in `symbolic_test.py`

[2] The bound is defined in `SymbolicList.List_MAX_Length`. Feel free to experiment with different bounds.