CSC410

Assignment 3

Due on October 19, 2020 at 11:59pm

Assignment Format and Guidelines on Submission

This assignment is worth 10% of the total course mark. Submit on Markus and follow these rules:

- This is a group assignment, designed for groups of 4 students. The volume of work is designed so that it would be reasonable for a group of 4. You may choose to work in a group of 3. Groups of fewer than 3 students will be not be accepted. You cannot submit individually. You can use Piazza, Quercus, or the old-fashioned face-to-face to form groups.
- Each group should submit three files named `q1a.py`, `q1b.py` and `q2.py`. Note that Markus has been setup to accept exactly those 3 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.
- In this assignment, your problem encodings can only use boolean variables. You should not solve the problems algorithmically, which means that the solution of the problem can always be deduced only from the solution given by the solver.

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. All functions necessary to complete the assignment have been presented in Tutorial 2.
- You are not allowed to import any other package than the ones already imported in your python files.
- 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.

Word of Advice

The goal of this assignment is to design encodings for some puzzles. 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 majority of your effort goes into devising a correct encoding of the problem in SAT.

Problem 1(a) (40 points)

The goal is to learn how to encode a simple reachability problem. You are given an input grid of size $n \times m$ where cells are either empty (zeroes) or blocked (ones), and we want to query if there is a path from one cell to another avoiding the blocked cells. On this grid, you can move from one empty cell to an adjacent empty cell either vertically or horizontally, but not diagonally. You cannot enter a blocked cell. Your task is to encode the problem into SAT.

Given your encoding, a solver should produce an answer to the following: is there a path from cell (i_b, j_b) to cell (i_e, j_e)? That is, if there is a path, the solver returns `sat` and if not, the solver returns `unsat`.

```
0 1 3 4
0 0 1 0 0
0 0 1 0 0
0 0 1 0 0
0 0 0 1 0
```

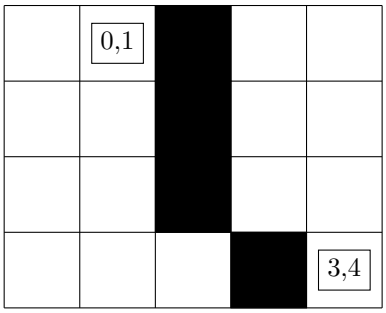

Figure 1: Example text representation and visual representation of a problem. Remark, there is no path from (0,1) to (3,4).

Input format In the input text file, the first line of the input contains 4 integers separated by spaces. These are in order i_b, j_b, i_e, and j_e respectively. The following n lines are the lines of the grid. Each line contains exactly m integers separated by spaces, each integer is `0` or `1`. A `0` represents an empty cell and and a `1` represents a blocked cell.

Figure 1 gives an example text input on the left, and the corresponding grid on the right, with the starting and ending cells marked. Note that the python code for parsing the input (a text file) is already present in `q1a.py` and *you do not need to modify it.* For testing pruposes, two examples inputs are provided in the `test_inputs` folder: `q1a_sat.txt` is satisfiable and `q1a_unsat.txt` is not.

Tasks Implement the function `encoding(pt_from, pt_to, grid)` in `q1a.py`. The function should return `True` if there is a path from cell `pt_from` to cell `pt_to` in the grid `grid`, and `False` otherwise. You are not allowed to modify the signature of the function, but you can add as many helper functions as you want.

Problem 1(b) (30 points)

You will build on your efforts for solving the previous problem for this one. You will solve the puzzle of the *psycho killer* by encoding it into a SAT problem. Here is the puzzle for the case of a 4×4 grid:

> *A building has 16 rooms, arranged in a 4×4 grid. There is a door between every pair of adjacent rooms ("adjacent" meaning north, south, west and east, but no diagonals). Only the room in the southeast corner has a door that leads out of the building.*
>
> *In the initial configuration there is one person in each room. The person in the northwest is a psycho killer. The psycho killer has the following traits: if he enters a room where there is another person, he immediately kills that person. But he also cannot stand the sight of blood, so he will not enter any room where there is a dead person.*
>
> *As it happened, from that initial configuration, the psycho killer managed to get out of the building after killing all the other 15 people. What path did he take?*

Tasks Your task is to encode this problem into a SAT problem. The puzzle above is defined for a 4×4 grid, but you will produce an encoding that generalizes to any $n \times n$ grid. The psycho killer starts at the northwest corner (room $(0, 0)$) and the exit is at the southeast corner (room $(n-1, n-1)$).

Implement the function `encoding(n)` in `q1b.py`. It takes as input the size of the grid n, and should return the list of moves taken by the psycho killer. If the killer can take a path killing all the $n*n-1$ people, the list of moves should take the killer from room $(0, 0)$ to room $(n-1, n-1)$. If there is no such path, the list returned should be empty.

Problem 2 (30 points)

In this problem, you will experiment with giving two different encodings to the same problem, and comparing their performance. 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 [1] 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 $\{R_{i,j} \,|1 \leq i < n, 1 \leq j \leq k\}$. 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 `q2.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 `q2.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 `q2.py`. You can execute the script `q2.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 ($0 <$ `K` $<$ `N`).

   ```
   python q2.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.

References

[1] Carsten Sinz. Towards an optimal cnf encoding of boolean cardinality constraints. In *International conference on principles and practice of constraint programming*, pages 827–831. Springer, 2005.