Knowledge Representation and Reasoning 2022

Homework Assignment #1

Due: Feb 28, 2022, 23:59.
Please submit as a PDF file, typesetting with LaTeX is preferred.

For more detailed instructions, see:
`https://canvas.uva.nl/courses/28686/assignments/307963`

Goals of Exercise 1:

- demonstrate your understanding of how to express various types of knowledge into the language of propositional formulas in CNF.

Exercise 1 (3pts)**.** Consider the following puzzle called ***k-n*-Queens**. You are given positive integers n and k. The puzzle is played on an $n \times n$ chess board. You are to place *at least* k queens on this chess board in such a way that no two queens attack each other. Remember that in chess, a queen may take any number of steps, either horizontally, vertically, or diagonally.

In this assignment, you will provide an algorithm that for any given n and k, translates the puzzle to a propositional logic CNF formula $\varphi_{n,k}$ such that the satisfying truth assignments of $\varphi_{n,k}$ correspond one-to-one to the solutions of the puzzle (for this particular value of n and k). You will do this in two parts—by constructing two CNF formulas $\psi_{n,k}$ and $\chi_{n,k}$, such that $\varphi_{n,k} = \psi_{n,k} \wedge \chi_{n,k}$ and:

- $\psi_{n,k}$ expresses that no two placed queens may attack each other, and
- $\chi_{n,k}$ expresses that at least k queens are placed on the $n \times n$ board.

As a starting point: the formula that you construct contains propositional variables $p_{i,j}$ for $i, j \in \{1, \dots, n\}$, each of which indicates whether or not a queen is place on cell (i, j) of the chess board. Note that you may introduce additional propositional variables, besides these variables $p_{i,j}$.

(a) Explain how, given values for n and k, one can construct a CNF formula $\psi_{n,k}$ whose satisfying truth assignments correspond one-to-one to placements of (any number of) queens on an $n \times n$ chess board, in a way that no two queens attack each other.

Be sure to explain how the possible placements of queens where no two queens attack each other correspond one-to-one to the satisfying truth assignments of $\psi_{n,k}$.

(b) Show how one can construct a CNF formula $\chi_{n,k}$ that is satisfiable if and only if at least k variables $p_{i,j}$ are set to true. To be more precise, $\chi_{n,k}$ should have the property that for all truth assignments α to the variables $p_{i,j}$ it must hold that α can be extended to a satisfying assignment for $\chi_{n,k}$ if and only if α sets at least k variables $p_{i,j}$ to true.

The formula $\chi_{n,k}$ may contain additional propositional variables, besides the variables $p_{i,j}$.

Do this in a way that ensures that $\chi_{n,k}$ consists of at most, say, $10 \cdot n^4$ clauses, regardless of the value of k.[1]

Be sure to explain *why* your construction of of $\chi_{n,k}$ works correctly.

[1]So in particular, you may not use the solution where $\chi_{n,k}$ simply consists of all the $\binom{n^2}{k}$ clauses corresponding to the size-k subsets of $\{ p_{i,j} \mid i, j \in \{1, \dots, n\} \}$.

Goals of Exercise 2:

- demonstrate your understanding of how a black-box algorithm for (the decision variant of) SAT can be called multiple times to perform more complicated forms of reasoning.

Exercise 2 (3pts). In this assignment, you will show how one can use a black-box algorithm for a restricted form of reasoning to construct algorithms that can find the answer to more complicated reasoning tasks.

In particular, you will construct an algorithm C that can do the following. When given as input a propositional logic CNF formula φ and a positive integer n, it enumerates n satisfying assignments of φ—or all satisfying assignments of φ, if φ has less than n satisfying assignments. (Note: we only consider truth assignments over the propositional variables appearing in φ.)

The black-box algorithm A that you are given, and that you may call in your algorithm C, does the following. Given a propositional logic CNF formula φ, it outputs a 0 if φ is not satisfiable, and it outputs a 1 if φ is satisfiable. (So in case φ is satisfiable, it does not output a satisfying truth assignment for φ.)

(a) Begin by describing an algorithm B, that may call A multiple times, and that does the following. When given as input a propositional logic CNF formula φ, it either outputs "unsat", if φ is not satisfiable, or if φ is satisfiable, it outputs a satisfying truth assignment for φ.

Do this in a way that avoids an exponential blow-up in the running time.[2] For example, this means that B may not just iterate over all possible truth assignments over the variables in φ. (You may count each call to A as a single step in the running time.) An informal explanation of why there is no exponential blow-up in your algorithm is enough—no need to give a fully detailed proof of this.

(b) Next, describe an algorithm C, that may call A and B multiple times and that does what is described above.

Again, do this in a way that avoids an exponential blow-up in the running time—and again, an informal argument why this is the case is enough.[2] (You may count each call to A or B as a single step in the running time.)

Goals of Exercise 3:

- demonstrate your understanding of how ASP can be used to solve other problems.

Exercise 3 (4pts). In this assignment, you will show how to compute satisfying truth assignments for propositional logic formulas φ—that are not necessarily in CNF—using answer set programming.

In other words, you should describe[3] an algorithm that does the following.

- It takes as input a propositional logic formula φ.
- If φ is unsatisfiable, it outputs "unsatisfiable."
- If φ is satisfiable, it outputs some truth assignment α that satisfies it.

Moreover, it may use as subroutine an algorithm that given an answer set program P produces an answer set of P (if such an answer set exists).

Again, do this in a way that avoids an exponential blow-up in the running time[2]—and again, an informal argument why there is no exponential blow-up is enough. You may assume that calling the answer set finding subroutine does not add to the time steps of your algorithm.

[2]The reason for this requirement is that one can give a simple exponential-time algorithm that just iterates over all possible truth assignments, for example.

[3]You do not have to give an entirely detailed description of your algorithm, e.g., in terms of pseudo-code. Instead, describe in high-level terms what steps the algorithm takes and what it outputs, on any given input. Make sure to be precise enough so that your description makes clear *how* the various steps work. If there are multiple ways to carry out a given step, and it matters how exactly this step is carried out, you should specify how the step is carried out.

Hints:

- To get a bit more concrete idea what kind of solution you should be thinking of, have a look at Example 1. (In this example, there are some claims that would have to be accompanied by an explanation of why they are true, to get full points in a solution—these are indicated with the symbol ‡.)
- The Tseytin transformation might come in handy.

Example 1. Consider the following problem. As input, you are given an undirected graph $G = (V, E)$. The task is to find a 3-coloring[4] of the graph G—that is, a function μ that maps each vertex $v \in V$ to, say, either red, green or blue—such that for each edge $\{u, v\} \in E$ it holds that u and v get assigned different colors. The problem is to find such a coloring μ if it exists, or to find out if no such coloring exists.

We describe an algorithm that solves this problem by using answer set programming. The algorithm works as follows. It takes as input an undirected graph $G = (V, E)$, where $V = \{v_1, \dots, v_n\}$, and it produces an answer set program P as follows. For each vertex v_i, it adds the following three rules to P:

```
red(i) :- not green(i), not blue(i).
green(i) :- not red(i), not blue(i).
blue(i) :- not red(i), not green(i).
```

Then, for each edge $e = \{v_i, v_j\} \in E$, it adds the following constraints to P:

```
:- red(i), red(j).
:- green(i), green(j).
:- blue(i), blue(j).
```

The answer set program P that is constructed in this way has the following properties.

For each answer set A of P it holds that for each v_i, there is exactly one of `red(i)`, `green(i)` and `blue(i)` in A.‡ In other words, we can construct a coloring μ of G from the answer set A as follows. For each v_i, if `red(i)` $\in A$, then let $\mu(v_i) =$ red; if `green(i)` $\in A$, then let $\mu(v_i) =$ green; and if `blue(i)` $\in A$, then let $\mu(v_i) =$ blue.

Moreover, for each answer set A of P, the coloring μ that is constructed from A has the property that it does not assign the same color to any two vertices that are connected with an edge in E.‡

In fact, whenever there is a 3-coloring of G that has the required property that no connected vertices are assigned the same color, the program P has an answer set A that corresponds to it.‡

So we can find such a 3-coloring of the graph G, if it exists, as follows.

1. Construct the program P, as described above.
2. Call the answer set solving algorithm to find some answer set A of P.
3. Transform A into the 3-coloring μ, as described above.

Both the construction of P and the transformation of A to μ do not involve any exponential blow-up: both only involve iterating over all parts of the input once, and for each part adding some elements to the output.

[4] See, e.g., `https://en.wikipedia.org/wiki/Graph_coloring`.