Formal Methods in Computer Science
Exercsie Sheet 1: Computability and Complexity

version 1

WS 2019/20

The deadline for submitting your solutions via tuwel is November 4, 2019.

1 Prologue

In the lecture we have seen reductions as a powerful tool to prove a problem to be at least as hard as some other problem. The different kinds of reductions are the standard technique to show a problem to be undecidable or NP-hard (or $\mathcal{C}$-hard for your favorite complexity class $\mathcal{C}$).

In practice reductions can be also used when designing a system for a combinatorial hard problem. This allows to obtain good performing systems with relatively low effort. For instance a popular approach to tackle problems that are known (or suspect) to be NP-complete is to encode them in (reduce them to) a known NP hard formalism for which sophisticated systems are already available. Prominent examples for such formalisms are propositional logic (SAT), logic programming (LP), integer linear programming (ILP), quantified Boolean formulas (QBFs), and constraint satisfaction (CSP).

Similar as in complexity theory one can use different kinds of reductions. In the simplest case, in analogy to many-one reductions, one processes the instance of the problem at hand to one instance of the target formalism, solves the newly generated instance with one of the sophisticated systems for target formalism and then uses the solution to compute a solution for the original problem instance. On the other hand, in analogy to Turing reductions, one can produce several problem instances in the target formalism, solve them with the existing system and then compute the solution of the original problem based on the solutions of all the constructed instances. Often the exact problem instances constructed already depend on the solutions of earlier instances.

A major difference to reductions in complexity theory is that we are not only interested in decision problems but in actually computing a solution for the problem at hand. This has to be taken into account in (a) the reduction step and (b) a post-processing step which maps the solution for the constructed problem to a solution of the original problem.

2 Exercises

In this exercise we consider graph coloring problems and aim to solve them via reductions to propositional logic and exploiting SAT-Solving systems (a.k.a. SAT-solvers).

Recall the NP-complete 3-COLORABILITY problem from the lecture where one is given an undirected graph and three colors (w.l.o.g. red, green, blue) and the task is to find a coloring of the vertices of the graph with the given colors such that no to adjacent vertices have to same color, if such a coloring exists. An easy way to set up a system for 3-COLORABILITY is to use the reduction from the lecture (with slight adjustments) to propositional logic, use a SAT-solver to compute a model, and finally extracting a coloring from the model of the formula.

While 3-COLORABILITY is probably the most prominent graph coloring problem, variants of COLORABILITY appear in many applications of graph techniques. In this exercise we consider a 4-COLORABILITY problem on directed graphs.

2.1 4-Colorings

We consider a set V of vertices and a relation $E \subseteq A \times A$ of directed conflict between these vertices, which together form a directed graph $G = (C, E)$. For an edge $(a, b) \in R$ we say that vertex a *disables* vertex b. We are interested in 4-colorings of the vertices (using the colors red, green, blue, orange) satisfying the following properties.

- No two adjacent vertices are both colored red.
- A vertex b is colored blue or green iff there is a vertex a that is colored red and disables b.
- If a vertex a disables a red colored vertex then a is colored green.
- If a vertex a is colored orange then there is another vertex b that disables a and is not colored green.

We call a 4-coloring satisfying the above conditions a valid coloring.

Clarifications:

- A graph might contain loops, i.e., edges (v,v) for some vertex v.
- A coloring assigns exactly one color to each vertex.

2.2 Tasks

Exercise 1

Provide a reduction that gets a graph as input and provides a propositional formula such that the formula is satisfiable if and only if the graph has a valid coloring. Moreover, this reduction should be such that the valid colorings of the graph are in one-to-one correspondence to the models of the propositional formula and one can easily compute the corresponding coloring when given one of the models.

Use the reduction to complete the parts of the SAT-solving framework provided in the tuwel course that deals with the computation of valid colorings.

- In the class *SatEncodings*: Use your reduction to complete the method *fourColoring(graph)* that generates an CNF formula, and the method *getColoringFromModel(coloring, model)* that generates a coloring given a model of the constructed formula.
- Additionally consider the class *Coloring*: Complete the method *isValidColoringOf(graph)* by implementing a polynomial time test whether the coloring is valid for a given graph.

Exercise 2

Now assume your are given a set of vertices S which have to be colored red. Extend your reduction from Exercise 1 such that it only returns valid colorings which color all vertices in S red. Use this reduction to complete the *fourColoring(graph, redVertices)* method in the *SatEncodings* class.

Exercise 3

Given a coloring C that colors certain vertices red we are interested whether we could color further vertices red as well. More formally, we want to know whether there is a coloring C' such that the set of vertices C_R colored red by C is a strict subset of the set of vertices C'_R colored red by C'. If there is no such S' we call C a maximal valid coloring. Testing maximality is an coNP problem while computing the maximal valid coloring is beyond NP/coNP. That is, the former can be efficiently reduced to a SAT-problem while the latter can not. However, one can solve the latter by iteratively solving the former problem, assuming that the former provides a counter-example if the coloring is not minimal, as follows. First, one computes a coloring as in problem 1 or 2. Then one iteratively tests whether the current set is minimal. If so we are done; if not the current coloring is replaced by the counter example and we proceed with testing for minimality.

Provide a reduction that takes an instance of the above problem, i.e., a graph $G = (A, R)$ and a valid coloring C for G, as input and provides a propositional formula such that the formula is unsatisfiable if and only if C maximal among the valid colorings. Again, this reduction should be such that one can easily obtain a coloring from a model of the formula, i.e., a counter-example C' for the minimality of C.

Use the reduction to complete the parts of the SAT-solving framework provided in the tuwel course that deal with the computation of maximal colorings.

- In the class *SatEncodings*: Use your reduction to complete the method *maxColoring(graph, coloring)* that generates an CNF formula, and (if necessary) adapt the method *getColoringFromModel(coloring, model)* that generates a coloring given a model of the constructed formula.

3 The Template for the Implementation

For the implementation part of the exercises we provide a java template (see tuwel course) for implementing SAT-reductions together with a class implementing four colorings. You only have to code the parts specific to the above problems. Use the given template and complete the *SatEncodings.java* and *Coloring.java* files.

The template exploits the Sat4j [1] SAT-solver and the JGraphT[2] package to process graphs. The corresponding APIs might be useful:

- Sat4j API: `www.sat4j.org/maven234/org.ow2.sat4j.core/apidocs/`
- JGraphT API: `http://jgrapht.org/javadoc/`

3.1 SAT-Solver

Modern SAT solver typically support formulas in conjunctive normal form (CNF) only. That is, your reduction must provide a CNF formula. If not so in first place add an other step transforming your formula into a CNF formula.

One of standard formats for SAT-solvers is the so called DIMACS format. For a formula with n variables the variables are represented by integers between 1 and n. DIMACS files are text files with three types of lines

- Lines starting with a "c" are comments
- The problem line: there is exactly one line starting with "p" which is called the problem line. It specifies the number of variables and clauses, and must appear before any clause is defined. For a CNF formula with n variables and m clause the problem line is"p cnf n m"
- For each clause there is one line with a sequence of integers corresponding to the literals of the clause. Negative literals are represented by negative integers. The line is terminated by a final value of 0.

[1] `http://www.sat4j.org/`
[2] `http://jgrapht.org/`

An example is provided in Figure 1. See `http://www.satcompetition.org/2009/format-benchmarks2009.html` for further reference.

Our framework does not produce DIMACS files, but uses an internal representation as `IVec<IVecInt>` object which is close to the above format. That is, `IVecInt` is a vector of integers representing a clause as in the DIMACS format, but does not require the terminating 0. `IVec<IVecInt>` is a collection of clauses, i.e., a representation of a CNF formula (details in the API `http://www.sat4j.org/maven234/org.ow2.sat4j.core/apidocs/org/sat4j/core/`).

$$\begin{aligned}\varphi =&(x_1 \vee x_2 \vee \neg x_4 \vee x_7) \wedge \\ &(\neg x_1 \vee x_3 \vee \neg x_5 \vee x_4 \vee x_7) \wedge \\ &(x_1 \vee \neg x_2) \wedge \\ &(x_4 \vee x_5 \vee \neg x_6)\end{aligned}$$

```
c Example
c
p cnf 7 4
1 2 -4 7 0
-1 3 -5 4 7 0
1 -2 0
4 5 -6 0
```

Figure 1: Example for a CNF formula and its encoding in the DIMACS format.

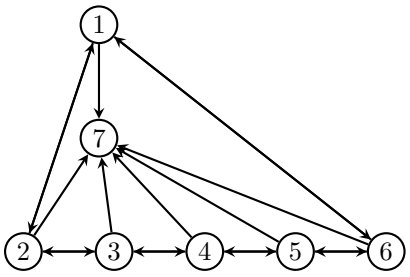
3.2 DIMACS Graph format

The input graphs are stored in a DIMACS format [3] which we describe in the following: The vertices of the graph with n vertices are the integers from 1 up to n. DIMACS files are text files with three types of lines

- Lines starting with a "c" are comments
- The problem line: there is exactly one line starting with "p" which is called the problem line. It specifies the number of vertices and edges and must appear before any edges is defined. For a graph with n vertices and m edges the problem line is"p edge n m"
- For each edge there is one edge descriptor line starting with e. The line corresponding to edge (v,w) is "e v w ".

An example is provided in Figure 2. For further reference on the DIMACS format see `http://prolland.free.fr/works/research/dsat/dimacs.html`. You can also generate graphs in DIMACS format using the JGraphT package (see `https://jgrapht.org/javadoc/org/jgrapht/io/DIMACSExporter.html`).

[3] Be aware that the Center for Discrete Mathematics and Theoretical Computer Science (DIMACS) proposed several slightly different formats for different purposes.




```
c
c A 3-colorable Graph
c
p edge 7 18
e 1 2
e 2 3
e 3 4
e 4 5
e 5 6
e 6 1
e 1 6
e 6 5
e 5 4
e 4 3
e 3 2
e 2 1
e 1 7
e 2 7
e 3 7
e 4 7
e 5 7
e 6 7
```

Figure 2: Example for a graph and its encoding in the DIMACS format.

Clarifications:

- For a graph with 10 vertices use the integers in $[1, 10]$ as names for the vertices. More generally, for a graph with n vertices use the integers in $[1, n]$ as names for the vertices.

4 Some Rules

Your are encouraged to discuss the exercises with other participants of the course (e.g. in the tuwel forum of the course) and share benchmark graphs and other resources you find useful with them. However, please obey the following rules.

- Write your own code.
- Write the report in your own words.
- Submit your solution in time.

4.1 Deliveries

Please upload a single zip archive in tuwel which contains:

- A short report (2-5 pages) on your solution (as pdf). In particular such a report should contain:
  - Your name
  - A short description of the studied problems
  - Your reductions to solve the exercises
  - A correctness argument for each of your reductions. In particular, explain how the models of the constructed SAT instance correspond to valid colorings.
  - A description of how the propositional atoms used in your reduction are mapped to the integer representation (DIMACS format) used in the implementation.
- The java files you had to complete:
  - *SatEncodings.java*
  - *Coloring.java*

  Test your implementations! (preferable under Java SE 11)
- At least five graphs in the described DIMACS format (that you used to test your system). Make sure your implementation works correctly on those graphs.

4.2 Assessment

The maximum score for this exercise is 15 points. The solution of each exercise must contain both the report and the completed java files. Solutions where either the report or the java files are missing are not considered for grading. However, we accept solutions where exercise 2 and/or 3 are missing.