Formal Methods in Computer Science

UE 185.A93 WS 2019/20

Exercise Sheet 1

Wolfgang Dvořák

Technische Universität Wien

WS 2019

Reductions

Reductions are a powerful tool in Complexity Theory to show

- a problem to be undecidable
- a problem to be NP-hard.
- a problem to be at least as hard as some other problem

In practice reductions can be used to actually solve (hard) problems:

- good performing systems
- with relatively low effort

Reductions

Popular approach to tackle NP-hard problems

- encode them in (reduce them to) a known NP hard formalisms,
- use sophisticated systems for that formalisms;
- and use the solution for the constructed problem to compute a solution for the original problem.

Prominent target formalisms:

- propositional logic (SAT), quantified Boolean formulas (QBFs)
- logic programming (LP)
- integer linear programming (ILP)
- constraint satisfaction problems (CSP)

Exercise Sheet 1

In this exercise we aim to solve

- graph coloring problems

with

- a reduction to proposition logic and
- exploiting SAT-Solver technology.

Additional information on the exercise sheet (in tuwel)!

Graph-Coloring - 3-COLORABILITY

We already know the 3-COLORABILITY problem

- One is given an undirected graph and
- three colors: red, green, blue.
- We want to find a coloring of the vertices of the graph
- such that no to adjacent vertices have to same color,

An easy way to set up a system for 3-COLORABILITY is

- use the reduction from the lecture to propositional logic,
  $$\varphi_G = \varphi_1 \wedge \varphi_2 \wedge \varphi_3$$
- use a SAT-solver to compute a model
- extracting a coloring from the model of the formula.
  e.g. color vertex i red if a_i^r is in the model.

Graph-Coloring - Our notion of 4-COLORABILITY

We consider a graph coloring problems on directed graphs $G = (V, E)$.

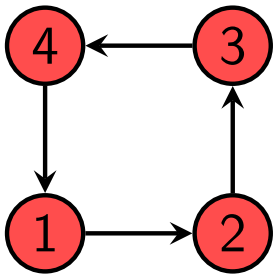
- A a set of agents,
- $R \subseteq A \times A$ a relation of directed conflicts between vertices

A 4-coloring of a graph $G = (V, E)$ is a function $C : V \mapsto \{r, g, b, o\}$, where r,g,b,o stand for the colors red, green, blue, and orange.

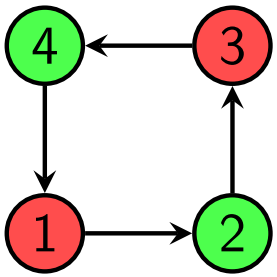
A 4-coloring is called a valid coloring iff it satisfies all of the following:

- 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.

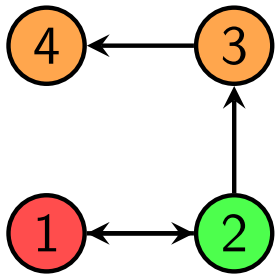
Graph-Coloring - Our notion of 4-COLORABILITY



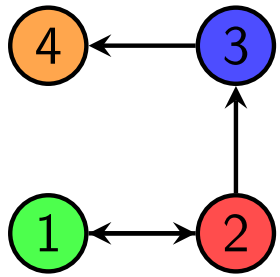
invalid



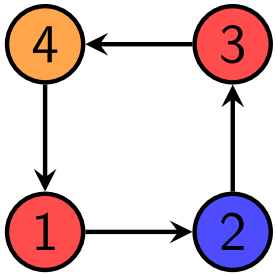
valid



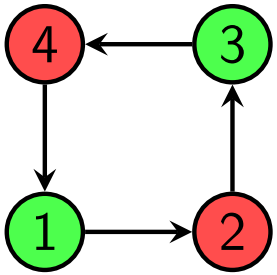
invalid



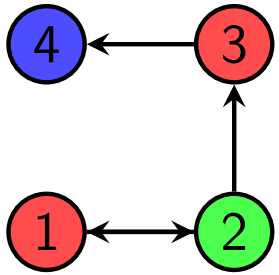
valid



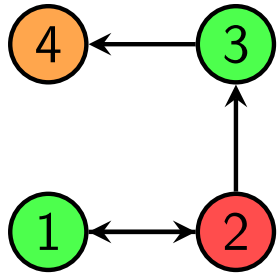
invalid



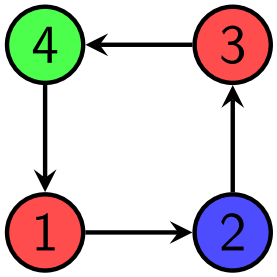
valid



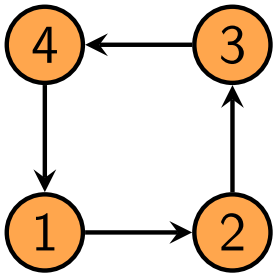
valid



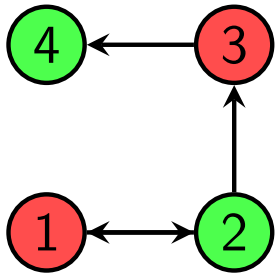
invalid



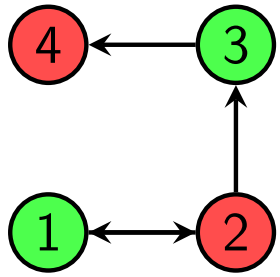
invalid



valid



valid



valid

Exercise 1: Valid Colorings

Problem: Identifying valid colorings of a graph

- Given: Graph $G = (V, E)$
- Goal: Find one (several/all resp.) valid coloring of G.

Tasks:

1. Provide a reduction that takes a graph G as input and provides a propositional formula such that
   - the models of the formula are in one-to-one correspondence with the valid colorings of G and
   - one can easily obtain a coloring from a model of the formula.
2. Use the reduction in a SAT-solving framework (provided in tuwel) to
   - decide whether a given graph has a valid coloring
   - if so compute such a coloring.

Exercise 2: Valid Colorings with a given set of red vertices

Problem: Identifying valid colorings that color certain vertices red.

- Given: Graph $G = (V, E)$ and a set $S_r \subseteq V$.
- Goal: Find one (several/all resp.) valid coloring C of G such that $C(v) = r$ for all $v \in S_r$.

Tasks:

1. Provide a reduction that takes a graph G and a set of S_r vertices as input and provides a propositional formula such that
   - the models of the formula are in one-to-one correspondence with the valid colorings of G that color all vertices in S_r red, and
   - one can easily obtain such a coloring from a model of the formula.
2. Use the reduction in a SAT-solving framework (provided in tuwel) to
   - decide whether a given graph has a valid coloring that color all vertices in S_r red
   - if so compute such a coloring.

Exercise 3:

Problem: Identifying valid colorings that color certain vertices red and the set of red colored vertices is maximal among all valid colorings.

- Given: Graph $G = (V, E)$ and a set $S_r \subseteq V$.
- Goal: Find one (several/all resp.) valid coloring C of G such that $C(v) = r$ for all $v \in S_r$ and there is no coloring C' with $\{v \mid C(v) = r\} \subset \{v \mid C'(v) = r\}$.

Complexity

- Testing maximality is an co-NP problem: $\hookrightarrow$ can be efficiently reduced to a SAT-problem
- while computing a maximal valid coloring is beyond NP/co-NP $\hookrightarrow$ cannot be efficiently reduced to a single SAT-problem

Exercise 3:

We can compute a maximal valid coloring as follows:

- Compute a valid coloring C (cf. Exercise 1 & 2)
- While C changes do
  - Test whether C is maximal
  - If not update C to the larger set computed in the previous step.
- return C

Notice:
We need a procedure that returns a counter example if C is not maximal.

Exercise 3: Identifying Maximal Valid Colorings

Tasks:

1. Provide a reduction from testing maximality of a valid coloring to SAT in propositional logic such that,
   - the formula is satisfiable iff the graph has a "larger" valid coloring, and
   - one can easily obtain a "larger" valid coloring from a model of the formula.
2. Use the reduction to complete the SAT-solving framework (provided in tuwel) that deals with maximal valid colorings.
   - decide whether a given valid coloring is maximal in a given graph.

Discuss but don't copy

Your are strongly encouraged to

- discuss the problems 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.

Please obey the following rules:

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

Deliveries

- 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.

Assessment

- Maximum score: 15 points.
- The solution of each part must contain both the report and the completed java classes.
- Solutions where either the report or java classes are missing are not considered for grading.
- However, we accept solutions where exercise 2 and/or 3 are missing.

Used Libraries

The template for exercise exploits the following packages:

- Sat4j[1] SAT-solver
  API: `www.sat4j.org/maven234/org.ow2.sat4j.core/apidocs/`
- JGraphT[2] package to process graphs.
  API: `http://jgrapht.org/javadoc/`

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

DIMACS-format

Modern SAT solver typically support

- formulas in conjunctive normal form (CNF) only.

$\hookrightarrow$ The reduction has to provide a CNF formula.

DIMACS format: Standard formats for SAT-solvers is the so called .

- 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 line starting with "p" which is the problem line. 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.

DIMACS-format - Example

- Start with some formula:
  $\varphi = (a \rightarrow b) \wedge (c \vee d \vee \neg b)$
- We need a formula in CNF:
  $\varphi = (\neg a \vee b) \wedge (c \vee d \vee \neg b)$
- Map variables to integers:
  $\varphi = (\neg x_1 \vee x_2) \wedge (x_3 \vee x_4 \vee \neg x_2)$

DIMACS-Encoding:

```
c Example
c
p cnf 4 2
-1 2 0
3 4 -2 0
```

SAT4j

Our framework uses an internal representation:

- Formula is represented as Vector of Integer Vectors `IVec<IVecInt>`.
- `IVecInt` is a vector of integers representing a clause
- `IVec<IVecInt>` is a collection of clauses

Example:

```
IVec<IVecInt> cnf = new Vec<IVecInt>();
// First clause
IVecInt clause = new VecInt();
clause.push(-1);
clause.push(2);
cnf.push(clause);
// Second clause
clause = new VecInt();
clause.push(3);
clause.push(4);
clause.push(-2);
cnf.push(clause);
```

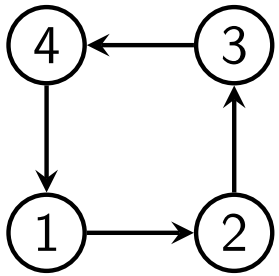
SAT4j

```
//  Initialize the SAT solver
ISolver solver = SolverFactory.newDefault();

try {
  // Feeds the formula to the solver
  solver.addAllClauses(cnf);

  IProblem instance = solver;

  solver.setTimeout(3600);

  // Computes a model
  if (instance.isSatisfiable()) {
      int model[] = instance.model();
      ... Use model to solve the original problem ...
      }
  } catch (ContradictionException e) {
      System.out.println("There is no model.");
  }
```

JGraphT - DIMACS Graph format

The input graphs are stored in a DIMACS format:

- The vertices of the graph are the integers from 1 up to n.
- Lines starting with a "c" are comments
- The line starting with "p" which is called the problem line. 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 ".



DIMACS-Encoding:

c Example
p edge 4 4
e 1 2
e 2 3
e 3 4
e 4 1

JGraphT - Code Snippets

Our template uses the *org.jgrapht.graph.DefaultDirectedGraph* class.

```
import org.jgrapht.graph.DefaultDirectedGraph;
import org.jgrapht.graph.DefaultEdge;

DefaultDirectedGraph<Integer, DefaultEdge> graph =
                                    readGraphFromFile(file.graph);

//iterate over all vertices
for(int vertex : graph.vertexSet()) {
      ...
}

//iterate over all edges
int source, target;
for (DefaultEdge edge : graph.edgeSet()) {
       source = graph.getEdgeSource(edge);
       target = graph.getEdgeTarget(edge);
       ...
}
```

JGraphT - Code Snippets

```
int source, target;

//iterate over the incoming edges of a vertex
for (DefaultEdge edge : graph.incomingEdgesOf(vertex)) {
      source = graph.getEdgeSource(edge);
      // graph.getEdgeTarget(edge);
      ...
}

//iterate over the outgoing edges of a vertex
for (DefaultEdge edge : graph.outgoingEdgesOf(vertex)) {
      //graph.getEdgeSource(edge);
      target = graph.getEdgeTarget(edge);
      ...
}
```

Questions?