IN1002 Introduction to Algorithms Courswork

February 24, 2023

This coursework tests your ability to create algorithms and to analyse their complexity. These are key skills that every computer scientist needs.

Because this is a second semester unit, we will *assume* that you have the following skills and knowledge from your first semester:

- You know what *Boolean arithmetic* is and what $\wedge$ (conjunction), $\vee$ (disjunction) and $\neg$ (negation) mean.
- You know what *Boolean satisfiability* is and how to work out if a Boolean expression is satisfiable or unsatisfiable.
- You can write small Java programs.
- You can write Java programs that work with 1 and 2 dimensional arrays.

You will not get marks for these but you need to know them to do this coursework. If you do not feel confident about these please revise your notes from last semester and talk to the staff running your practical sessions or your personal tutor.

1 Representation of Boolean Expressions

In this coursework we will work with Boolean expressions. To do this we will represent them using integers and arrays.

Boolean Variables will be represented by integers starting from 1. So rather than having variables called p and q we will have variables called v_1 and v_2 which we will represent using `1` and `2`.

Negations of variables will be represented by negative numbers, so `-1` represents $\neg v_1$ and `-483` represents $\neg v_{483}$.

Literal refers to a variable or its negation. For example v_3 is a literal which is represented by `3`. $\neg v_{11}$ is also a literal and is represented by `-11`.

Clauses are disjunctions of literals. We will represent these using arrays of integers. For example the clause $v_4 \vee (\neg v_7) \vee v_5$ is represented by an array `| 4 | -7 | 5 |`. An array that represents a clause must never contain zero. Clauses may contain the same variable several times or a variable and its negation.

Clause Database is an array of clauses. This represents the conjunction of all of the clauses. For example the array which contains: `| 1 | -2 |`, `| -1 | -2 | 3 |` and `| 2 |` represents the Boolean expression:
$(v_1 \vee (\neg v_2)) \wedge ((\neg v_1) \vee (\neg v_2) \vee v_3) \wedge v_2$

Assignments are maps that assign each variable to true or false. They are the part of a row in a truth table that gives the values for each variable. We will represent them using arrays. The first element of the array must be `0` and is ignored. All other elements must either be `1` or `-1`. If `a[i]` is `1` then v_i is set to true. If `a[i]` is `-1` then v_i is set to false[1].

For example if we have four variables and we want to represent that case when v_1 and v_4 are true and the others are false, we would represent this with an array `| 0 | 1 | -1 | -1 | 1 |`.

Partial Assignments are maps that assign each variable to true, false or unknown. They can describe sets of rows of a truth table. We will represent them using arrays. The first element of the array must be `0` and is ignored. All other elements must either be `1`, `-1` or `0`. If `a[i]` is `1` then v_i is set to true. If `a[i]` is `-1` then v_i is set to false. If `a[i]` is `0` then v_i is unknown.

For example if we have four variables and we want to represent that case when v_1 is true, v_3 is false and the others are unknown, we would represent this with an array `| 0 | 1 | 0 | -1 | 0 |`.

[1] This is a little unusual, normally false is represented using `0`. For this kind of algorithm it is easier to represent false using `-1` because then you can perform a logical negation by using arithmetic negation (i.e. `-`).

2 Tasks

Please read the instructions carefully. You will loose marks if you do not follow them.

Download the file `Solver.java` from Moodle. Put your name and your e-mail address in the comments at the top. This class gives you a number of methods which will simplify the task. Put your code in the marked locations. When you hand your work in the section marked `DO NOT CHANGE!` must not be changed from the original.

You must submit your modified version of `Solver.java` along with your answers as comments. You do not need to submit anything else. Do not submit `.class` files.

2.1 Part A

For each of the tasks below you must devise an algorithm, implement it in Java, makes sure that it compiles and runs and test it. You must also give the complexity class for the best case and worst case.

1. Fill in the `checkClause` method. The input is an array representing an assignment and an array representing a clause. It must output `true` if the clause is *satisfiable*. Otherwise it must give `false`. Give the best and worst case complexity in terms of the number of literals in the clause (v).

2. Fill in the `checkClauseDatabase` method. The input is an array representing an assignment and the clause database. It must output `true` if the assignment satisifies *all* of the clauses. If one or more of the clauses is unsatifiable it must return `false`. Give the best and worst case complexity in terms of the length of the longest clause (l) and the number of clauses (c).

3. Fill in the `checkClausePartial` method. The input is an array representing a partial assignment and an array representing a clause. It must output `1` if the clause is *satisfiable*. It must output `-1` if the clause is *unsatisfiable*. Otherwise it must output `0`. Give the best and worst case complexity in terms of the number of literals in the clause (v).

4. Fill in the `findUnit` method. The input is an array representing a partial assignment and an array representing a clause. If there is exactly one literal in the clause that is unknown and all other are false then it returns the literal that is unknown. Otherwise it returns `0`. Give the best and worst case complexity in terms of the number of literals in the clause (v).

2.2 Part B

Download the your personalised zip file. This contains 15 files. Each file contains a clause database. The `loadClauseDatabase` method can load one of these files into the program.

Come up with an algorithm for the `checkSat` method and implement it. The input is the clause database. The output is an assignment that satisfies the clause database or `null` if there are no assignments that satisfy the clause database. You do not need to give the complexity of this method. You may use extra methods if that allows you to clearly express you algorithm.

Your program must be able to solve as many of the clause databases you have been given as quickly as possible. You must say in the comment which clause databases you think you can solve.

If your program does not compile or does not run, you will get 0 for Part B!

3 Deadline

Sun 9^{th} April 2023 17:00:00

4 Mark Scheme

This is individual coursework. It is worth 50% of the unit mark and is marked out of 50, so each mark is worth 1% of your over-all course mark.

4.1 Part A : 16 marks

Each question is worth 4 marks each. 2 marks for the algorithm itself. 1 for the best case complexity. 1 for the worst case complexity.

4.2 Part B : 34 marks

For each of your 15 personalised filess, your program will be run for 15 seconds. If it has given a correct answer in that time, you will be awarded 1 mark.

For each of the 15 marking files, your program will be run for 15 seconds. If it has given a correct answer in that time, you will be awarded 1 mark. These marking files will not be released but are similar to your personalised files.

A final 4 marks will be awarded for algorithm and code quality and creativity.

5 Hints

This is not part of the coursework. These are some ideas, hints and reminders to help you complete the coursework.

5.1 Working

- If you are stuck on how to start, try doing a few example by hand and then try to write down how you did it.
- Save copies of your work as you go. Especially when you have got something working. It can be easy to make a small change to an algorithm and break things. It is useful to be able to go back to the previous working one. A version control system like `git` would be ideal for doing this.
- Part B is significantly more challenging than Part A. Don't leave it to the last minute!
- Write comments as you go. Don't explain *what* you are doing (unless it is really complex), explain *why*. This will help you understand your code later.

5.2 Part B

- There is not a *one right answer*. Different algorithms will work for different people. There is a reason for this which we will cover in a later lecture. This also means that if you look up algorithms for this problem[2], it may or may not work for your problems.
- You might think this favours people with a faster computer. It doesn't for reasons that we will cover in lectures. With some algorithms and some inputs it is possible to take a *very* long time. I would suggest that you don't leave the solver running for more than a few minutes unless you really want to.
- The data files are human readable. The early ones are simple enough to do by hand.
- The data files get progressively more challenging but this is not necessarily perfectly linear. Don't get stuck on one and not look at the others. Maybe you could write a script to run all of the tests?
- There are some surprisingly simple recursive solutions that work really quite well.
- Ordering is very important. Doing things in numerical order may not be the best or the fastest way.

[2]Be careful, there are some very sophisticated and complicated ones!

- You can change the clauses in the clause database if you want *but* make sure that you don't add or remove satisfying assignments.
- You can build extra arrays to make finding things faster but be aware that you could spend more time on building them than they save you. Maybe complexity theory helps avoid this?
- Clauses of length 1 can be treated differently. As can clauses of length 2. Maybe you can use this to your advantage?