Complexity Practical 2 Report

160020220

April 2019

Contents

1 Introduction

In lectures we studied the importance of polynomial reductions between NP problems. A polynomial-time reduction is a method for solving one problem using another problem. if a hypothetical subroutine that solves a given problem X, then the input reduction or transformation of a second given problem Y for X allows Y to be solved by calling the algorithm on the reduced product. If both the time required to transform the first problem to the second, and the number of times the subroutine is called is polynomial, then the first problem is polynomial-time reducible to the second.

A polynomial-time reduction proves that the first problem is no more difficult than the second one, because whenever an efficient algorithm exists for the second problem, one exists for the first problem as well. The opposite statement stands as well.

The aim of this practical is to implement the reduction between Propositional Satisfiability (SAT) problems and graph coloring (COL) problems. Then to use the solutions and processes to perform a reduction analysis between problems.

2 Design

The project is split in 3 algorithm classes : Sattothreesat.java ; Coltosat.java ; Threesattocol.java.

and 2 parsing classes : COL.java ; CNF.java.

The makefile produces the executables required to run the stacks check and the main is using a switch statement to select which algorithm to run. This is mostly for testing and complexity analysis.

2.1 Algorithm Implementation

When thinking about how I am Designing my code and implementing the algorithms, I first think about the objects and data structures a=that would be easy to use and debug. I chose to represent the clauses and edge using the same data structure, a List¡String¿. I could have used a List¡Integer¿ to represent the literals but I preferred using string for an easier manipulation and concatenation. A List¡String¿ can also be converted easily to a List¡Integer¿ if needed.

Each clause or edge is stored in a List¡List¡String¿¿. An important precision here is that all my lists are ArrayLists¡¿ and their elements can be accessed using their respective index. The option I had here was to use a HashSet to prevent duplicates. However, according to the specs, duplicate clauses are allowed. Hashmaps could also be used but I preferred using listing rather than mapping.

All we need to perform the reductions is this list of clauses or edges. furthermore, The parsing objects will provide getters to allow an easier variable processing (e.g number of variables).

2.2 Parsing

2.2.1 CNF

The CNF parser reads DIMACS cnf format and write in said format, we have 2 methods to do so.

First, the read method reads from the input stream using a buffered reader and runs the process line by line. Here, the process is split in 2 parts again. The while reading loop starts by making sure everything in the preamble is correct. The comments and new lines are ignored, then we read the header ONCE and set the variables we need (e.g number of clauses and variables). Error checking is then performed after this first part before reading the clauses one by one. An important precision to give is that I use Booleans and continue statements to make sure the preamble is processed correctly. Now, the while loop processes the clauses by following simple steps and performing error checking at each of these. It add variables to the Hashmap of variables if it is not yet in it. We use a Hashmap data structure because the complexity to perform a look-up that has a O(1) complexity while for lists and set, the contain method has a O(n) complexity. These variables are added to a clause and when we have a ”0”, the clause gets added to the list of clauses. Finally, we check that we have the correct number of clauses in the clause list and we return said list.

Second, the write method is a very simple method. It takes a list of clauses, checks the number of variables then prints the hearder ”p cnf numVar numCla”. After this, the list of clauses is simply outputed following the cnf format.

2.2.2 COL

The COL parser also reads and writes DIMACS col format. However, we will not discuss the writing process as it is very similar to the one discussed above. The read method however is very different in this case while sharing some similarities. We still read the header to get the number of edges, the number of nodes and the number of colors. However, To produce the clauses for our list of clauses, we start the reduction algorithm here. This part will be discussed later on. The reader produces and returns a list of clauses.

3 Implementation

The ideas behind all these reduction algorithms are heavily inspired by the ones given in the practical specification.

3.1 SAT to 3 SAT

We describe a polynomial time reduction from SAT to 3SAT. The reduction takes an arbitrary SAT instance A as input, and transforms it to a 3SAT instance B , such that satisfiability is preserved, i.e., B is satisfiable if and only if A is satisfiable. Recall that a SAT instance is an AND of some clauses, and each clause is OR of some literals. A 3SAT instance is a special type of SAT instance in which each clause has exactly 3 literals.

Example of a SAT instance:

x1 AND (x1 v x2) AND (x1 v x4 v x6 v x7) AND (x1 v x2 v x3 v x5 v x7).

Example of a 3SAT instance:

(x1 v x2 v x4) AND (x2 v x3 v x5) AND (x1 v x4 v x6).

Using the specification sheet: ”Any clauses with 3 or fewer literals can be left untouched. We split clauses with 4 or more literals as follows:

(l1, l2, l3, l4, . . .) becomes two clauses

(l1, l2, y1), (-y1, l3, l4, . . .)

where y1 is a new variable.”

To implement this algorithm, I use a new list called ”nlist” that will store the new 3SAT clause list. First, we iterate through the original list of clauses. If the clause size is greater than 3, we add the first 2 element to a new clause, create a new variable that gets added to said clause and it is added to nlist. Then, a ListIterator is used to remove these first 2 element for the original list and add the not new variable. While the clause we are processing has a size greater than 3, we repeat this process.

Finally, we pass nlist to the cnf writer.

3.2 COL to SAT

This reduction follows three simple steps to produce a list of clauses. These steps are very well explained in the specification sheet and are summarized like so:

We have n SAT variables such as : n = number of nodes * number of colors. This is because each node can take 1 color. If we have 3 colors red, blue and green, each node can be each color and that means we need 1 variable for each node state.

At least one color is true for each node. So, we produce a clause that state that one of the node’s state must be true.

However, a node can’t be more than one color. So, if a node state is true, the other states must be false. This is described by the following: ” The clause -yi,j , -yi,j’ says that node i can’t be both colour j and colour j’.” (spec sheet)

These steps can be done Right after getting the information needed from the header. They have been comment ALO and AMO respectively and do the process described above. In both cases, a new ArrayList gets created to store the new clause. This list then gets added to the list of clauses.

Finally, 2 connected nodes can’t be the same color for the problem to be satisfiable. Thus, we have to process the edges line by line and add an edge for each respective state of a node. Indeed, if we have 3 colors again and node a and b are connected, a-red and b-red can’t be true; a-blue and b-blue can’t be true; a-green and b-green can’t be true. This is translated to (-yi,j , -yi,j’) with i nodes and j colors.

This algorithm takes place for the most part in the reading method of COL.java.

3.3 3SAT to COL

This reduction was the fastest to do since the parsers were already coded and implemented. All the logic here was about connecting nodes to represent clauses, This was done in 4 parts. These parts were discribed in the practical sheets as follows:

Given an instance of 3-SAT, with k clauses and n 4 variables, we construct a graph with 3n + k vertices, which we call:

(x1, . . . , xn)

(-x1, . . . , -xn)

(y1, . . . , yn)

(C1, . . . , Ck)

The edges are given by the following rules and we these constitute the 4 parts of the algorithm inside of the Threesattocol.java class.

– Each vertex xi is joined to -xi. Knowing that negative literals are not allowed in DIMACS col, we represent those by just adding the variable number to the variable.

If we have 4 variables, -1 = numVar + 1 ; -2 = numVar + 2 ; etc...

Then we create an edge to join xi to -xi according to the representation above.

– Each vertex yi is joined to every other yj.

To do this operation, we create 2 integers that will serve as indexes (in this case indexes serve as values as well). Y variables are created after the X and -X variables, thus take the values of 'numVar*2 + index'.

we iterate through both indexes and add the generated clauses to the list at each step. Depending on the first index and if it has reached the numVar limit. We increment either the first or second index and keep looping through. The clauses get added to nlist that way.

– Each vertex yi is joined to xj and -xj , provided j != i

The logic above is used again. The main difference here is that a conditional is added to make sure j!=i.

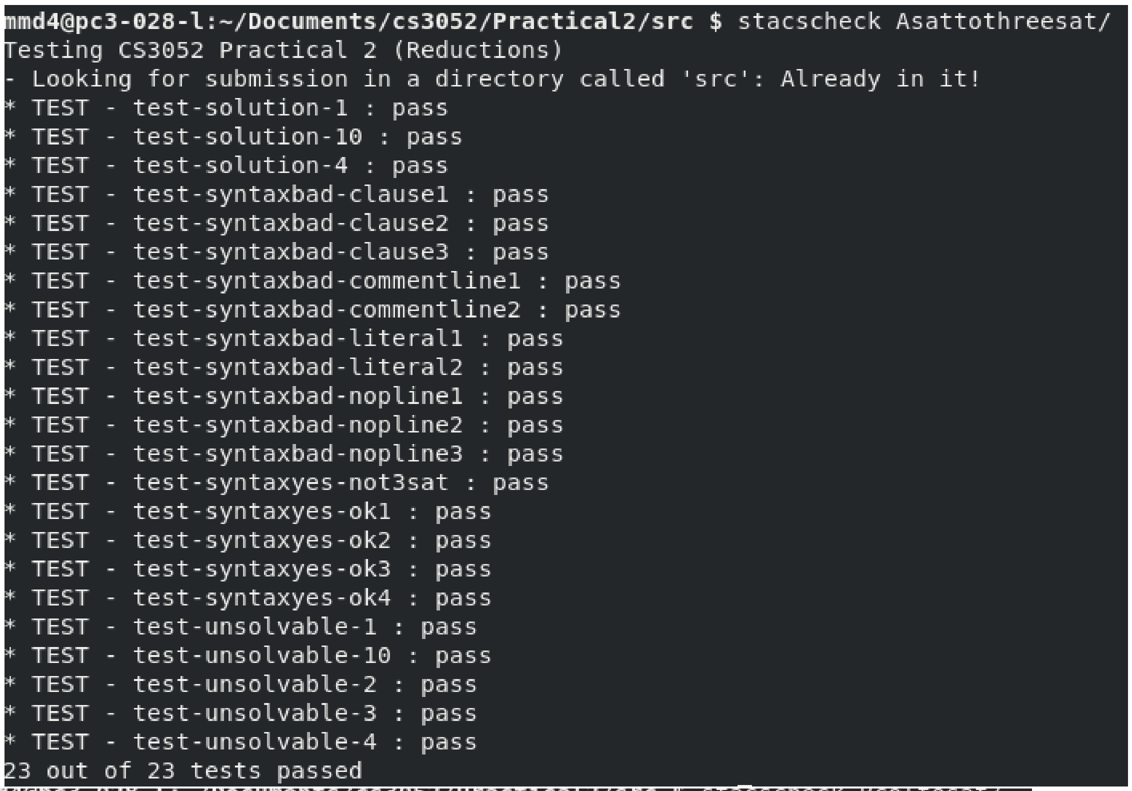
– Each vertex Ci is joined to each literal xj or -xj which is not in clause i.

Again, the iteration by 2 indexes logic is used. The main difference here is that we use a conditional with list.contains() to make sure we add only the literals that are not present in clause i. At this step, Error checking is also performed to make sure that we have a 3SAT format.

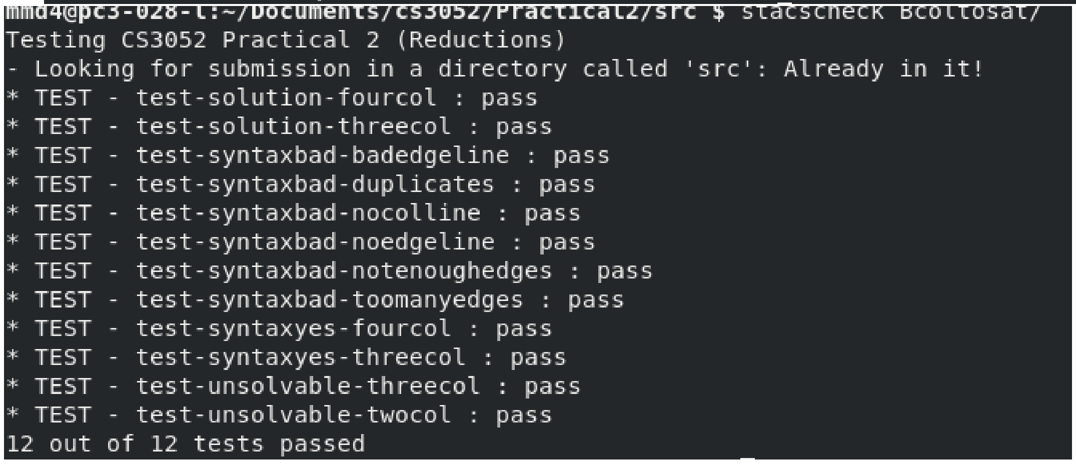
nlist is then given to the COL parser and written to the output stream.

4 Testing

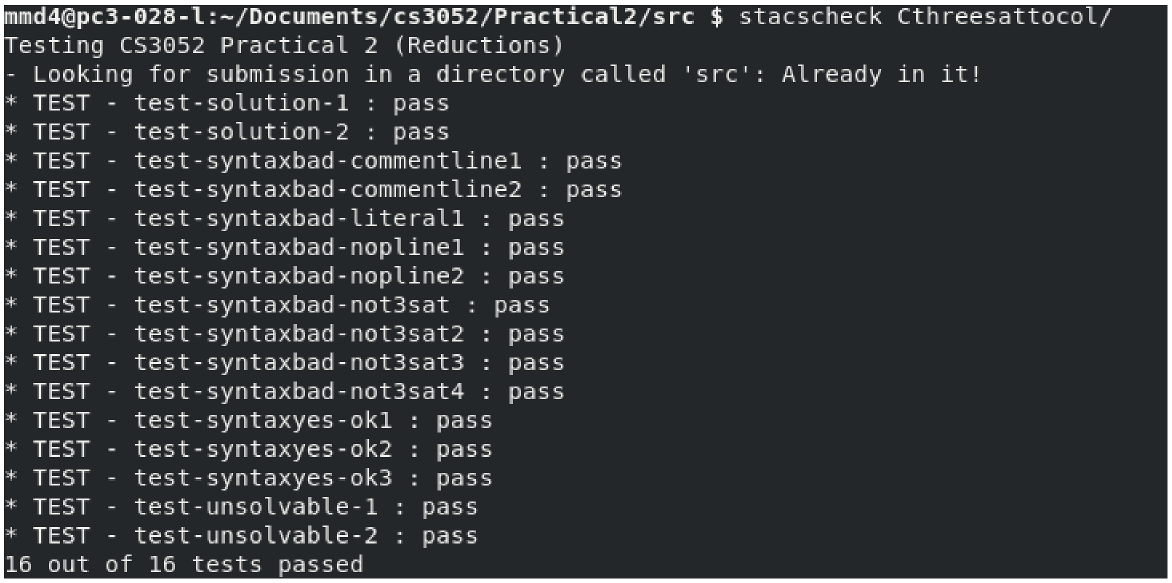
Since this practical is about complexity and not parsing, The syntax checkers of the stacscheck should be enough. Here are proof of running the tests:



```
mmd4@pc3-028-l:~/Documents/cs3052/Practical2/src $ stacscheck Asattothreesat/
Testing CS3052 Practical 2 (Reductions)
- Looking for submission in a directory called 'src': Already in it!
* TEST - test-solution-1 : pass
* TEST - test-solution-10 : pass
* TEST - test-solution-4 : pass
* TEST - test-syntaxbad-clause1 : pass
* TEST - test-syntaxbad-clause2 : pass
* TEST - test-syntaxbad-clause3 : pass
* TEST - test-syntaxbad-commentline1 : pass
* TEST - test-syntaxbad-commentline2 : pass
* TEST - test-syntaxbad-literal1 : pass
* TEST - test-syntaxbad-literal2 : pass
* TEST - test-syntaxbad-nopline1 : pass
* TEST - test-syntaxbad-nopline2 : pass
* TEST - test-syntaxbad-nopline3 : pass
* TEST - test-syntaxyes-not3sat : pass
* TEST - test-syntaxyes-ok1 : pass
* TEST - test-syntaxyes-ok2 : pass
* TEST - test-syntaxyes-ok3 : pass
* TEST - test-syntaxyes-ok4 : pass
* TEST - test-unsolvable-1 : pass
* TEST - test-unsolvable-10 : pass
* TEST - test-unsolvable-2 : pass
* TEST - test-unsolvable-3 : pass
* TEST - test-unsolvable-4 : pass
23 out of 23 tests passed
```



```
mmd4@pc3-028-l:~/Documents/cs3052/Practical2/src $ stacscheck Bcoltosat/
Testing CS3052 Practical 2 (Reductions)
- Looking for submission in a directory called 'src': Already in it!
* TEST - test-solution-fourcol : pass
* TEST - test-solution-threecol : pass
* TEST - test-syntaxbad-badedgeline : pass
* TEST - test-syntaxbad-duplicates : pass
* TEST - test-syntaxbad-nocolline : pass
* TEST - test-syntaxbad-noedgeline : pass
* TEST - test-syntaxbad-notenoughedges : pass
* TEST - test-syntaxbad-toomanyedges : pass
* TEST - test-syntaxyes-fourcol : pass
* TEST - test-syntaxyes-threecol : pass
* TEST - test-unsolvable-threecol : pass
* TEST - test-unsolvable-twocol : pass
12 out of 12 tests passed
```

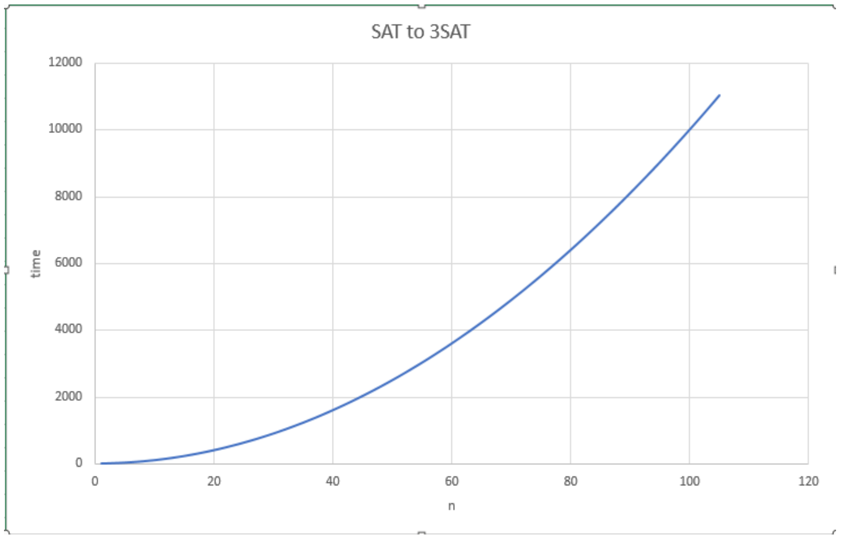


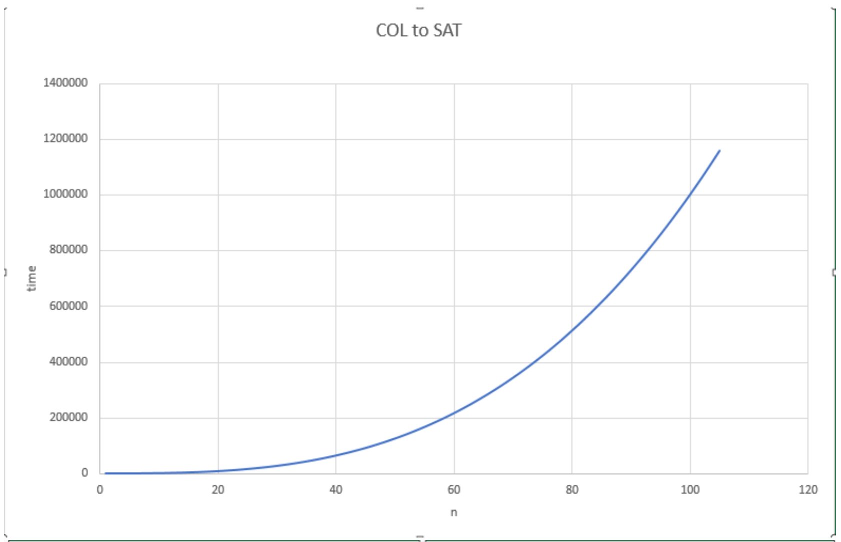
```
mmd4@pc3-028-l:~/Documents/cs3052/Practical2/src $ stacscheck Cthreesattocol/
Testing CS3052 Practical 2 (Reductions)
- Looking for submission in a directory called 'src': Already in it!
* TEST - test-solution-1 : pass
* TEST - test-solution-2 : pass
* TEST - test-syntaxbad-commentline1 : pass
* TEST - test-syntaxbad-commentline2 : pass
* TEST - test-syntaxbad-literal1 : pass
* TEST - test-syntaxbad-nopline1 : pass
* TEST - test-syntaxbad-nopline2 : pass
* TEST - test-syntaxbad-not3sat : pass
* TEST - test-syntaxbad-not3sat2 : pass
* TEST - test-syntaxbad-not3sat3 : pass
* TEST - test-syntaxbad-not3sat4 : pass
* TEST - test-syntaxyes-ok1 : pass
* TEST - test-syntaxyes-ok2 : pass
* TEST - test-syntaxyes-ok3 : pass
* TEST - test-unsolvable-1 : pass
* TEST - test-unsolvable-2 : pass
16 out of 16 tests passed
```

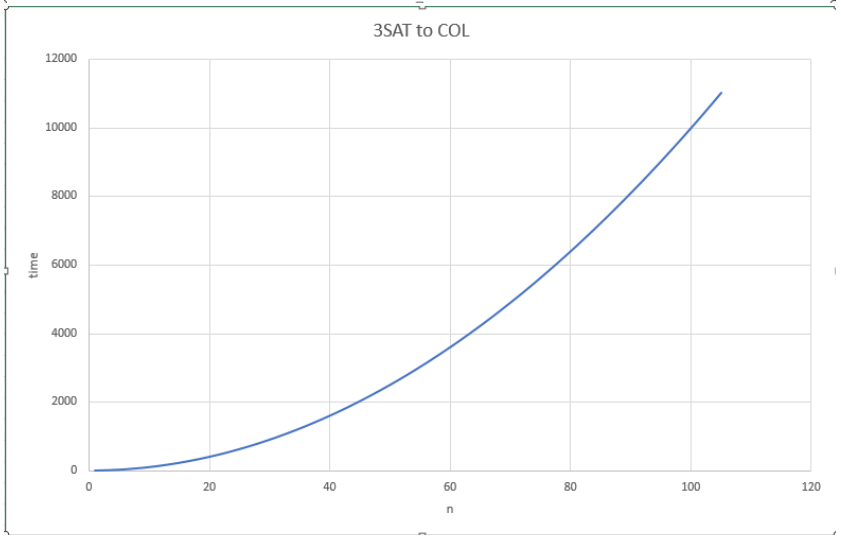
Further testing was done throughout the practical to check for duplicates and making sure parsing was correct however. The code is commented in that regard. Testing was mostly successful in that regard.

5 Task 3

Our algorithms are polynomial reduction algorithms. The best and easiest way to show that is to produce and analyze the respective complexity graphs of the algorithms:







As we can clearly see, We have a polynomial time complexity for all 3 algorithms because the graphs all present an upward parabola that translate an operation of the following form:

f(x)=n'c or POWER(n,c).

According to the commented code. We have many loops but also a few nested loops. knowing that a for loop or while loop treating n is O(n) complexity and thus, a nested loop explains the O(npow) complexity. Plus, The complexity analysis has been done using timestamps average on multiple repetitions to produce the graphs.

According to this, A circular reduction would mean that we generate more and more clauses throughout the process. Putting the graphs together and having :

Circular = COLtoSAT + SATto3SAT + 3SATtoCOL

the circular analysis should show that from COL to COL, the algorithms produce n(pow)c nodes with c being a constant.

6 Conclusion

In conclusion, The practical was dealing with parsing and complexity. Both parts were interesting, Although I don't' know if my complexity analysis should have been more extensive. All the data is provided in the practical folder and this report is 2450 words long.