CS3052: Computational Complexity
Practical 2: Reductions

Introduction

In Practical 2, we were asked to algorithmically perform polynomial reductions on a number of NP-Complete problems. These problems were focused on Graph Colouring and Satisfiability problems. We were then asked to analyse the complexity of these algorithms based on running them circularly. Meaning we take a colouring problem, convert it to satisfiability problems, and then back to a colouring problem and measure the increase in *nodes.*

Running Code

As per the spec a Makefile and executables are included and function properly. Running *make* builds the executables and running *make clean* gets rid of the class files so they can be built again. The programs are built in Java.

Task 1

Part A:

Design

The first part of Task 1 asked a user to implement SAT to 3-SAT reduction algorithm. A guideline for this reduction was given in the appendix however I found the lectures to be of most help in breaking down and understanding Satisfiability problems. From trying to follow the guidelines in the appendix I kept making inaccurate algorithms that would not give the correct new literals or the correct number of them. However after looking at many "befores and afters" of the reductions I came to understand a good way to approach building it. Basically my design isolates every literal that are not the first or last two literals of the clause being broken down. Then it inserts a new literal in the empty spaces.

Implementation

As an example of the design described above, the clause `1 2 3 4 5 6 7` will be broken down to:

```
1 2 8 0
-8 3 9 0
-9 4 10 0
-10 5 11 0
```

```
-11 6 7 0
```

To implement this I first use a BufferedReader to interpret each line that is added to standard input. As each line is interpreted and if it follows correct syntax it is added to a global ArrayList containing all of the clauses of the original SAT instance. Once all clauses of the SAT instance is added to the ArrayList I then convert the instance to 3-SAT. First I traverse through each clause in the SAT instance and if any of them have more than three literals I then break them down into different clauses. First, the first two literals are put into one clause with a new literal. Then I move through creating new literals based on the previous literal. If the previous new literal was a positive number the next new literal is negative. On the other hand if the previous new literal was negative, the next new literal is a negation of that number + 1. Between the first and last clauses which contain the first and last two original literals in the broken-down clause, I put the rest of the original literals isolated between two new literals.

Testing and Evaluation

The reduction passes all SAT-3SAT stacscheck tests as well as original tests that were used in task four. I also used some sample .cnf files to test from this site: http://people.sc.fsu.edu/~jburkardt/data/cnf/cnf.html

Part B:

Design

This second part of task one asked us to create a reduction from kCol to SAT. This reduction definitely took me the longest amount of time to implement because I had trouble understanding how to represent the new SAT literals. I also felt that the instructions in the appendix did not make it immediately clear what each variable would represent. However after watching this video: https://www.youtube.com/watch?v=HhFSgXbWiWY&feature=youtu.be

And getting some help from my peers I got a better understanding of how to design the algorithm. I followed the parts laid out in the spec appendix making sure I had ALO clauses, AMO clauses, and Edge clauses.

Implementation

In order to implement ColToSat I first followed the same structure as I did in the previous reduction to read each line from BufferedReader and interpret them in a method called *interpretLine* to check for syntax and add all the edge clauses into an ArrayList. Then I implemented the SAT clauses and stored those in an ArrayList. Firstly, to define the scope of the literals I made them equal to the number of nodes multiplied

by the number of colours. This may be confusing but each group of the number of colours of numbers represents a single node combination with each colour. For example for nodes A, B, C and colours red, green they are numerically represented like this:
1=A,red. 2=A,green. 3=B,red. 4=B,green. 5=C,red. 6=C,green.
So because there are 2 colours, nodes with the same colour will always incrementally be two apart.

To create the "At Least One" clauses I added all the different colour combination of each single node into one clause. Using the example above, each ALO clause would look like this:
1 2 0
3 4 0
5 6 0

To create "At Most One" clauses I created a negative clause for each colour for each node in which all colours associated with that node except for one was represented. For example:
-1 0
-2 0
-3 0
-4 0
-5 0
-6 0

This is not incredibly clear given that the example only has two colours but it does follow the structure of the program.

The last set of clauses called Edge clauses added a clause for each edge and colour so that no two nodes connected in one edge were of the same colour. This was implemented by going through the stored edge clauses in the ArrayList from the col instance. For each of those edges it provides negated clauses for each node combination of the same colour.

Testing and Evaluation

The program passes all of the latest versions of stacscheck as of 23/04 however I had difficulty with it passing previous versions and therefore had to refine it multiple times. It also successfully outputs for some of my own tests and tests that I have gotten from other sites such as https://mat.tepper.cmu.edu/COLOR/instances.html and

https://reference.wolfram.com/language/ref/format/DIMACS.html?view=all. These also came in handy for Task 3.

Task 2

Design

Task 2 involved designing and implementing a reduction from 3-SAT to kCOL. The provided appendix and some help from my peers greatly helped me create a design. The design roughly involves creating a variable called numColNodes which is equal to three times the number of variables plus the number of clauses. Then I conceptually group numbers 1-numColNodes into four categories similar to those laid out in the appendix. The first three groups are the length of the number of variables and represent in order x, the negation of x, and y. Then for however many clauses, the number of ints after that each represent a clause in order they are in the ArrayList.

Implementation

I used a BufferedReader like I did in the previous two implementations to read each line in standard input and interpret them. Every 3-SAT clause is added to an ArrayList.

To reduce a 3-SAT instance to kCOL I first make the total number of variables equal to four if it is less than four. Then I set the total number of nodes equal to three times the total variable number and add the number of clauses in the 3-SAT instance to this. Using a single for loop I first connect the x nodes to the negated x nodes and add each of these edges to an ArrayList. Then using nested for loops I connect the y nodes to each other. In doing this I also need to check if the ArrayList already contains this edge and add it if it does not. Next I connect y nodes to x and negated x nodes if they do not have the same colour. Again I only add these edges if they are not currently in the ArrayList. Lastly I connect the clause nodes only to literals that are not currently in those clauses.

Testing and Evaluation

On the lab machines the program passes all stacscheck tests for this reduction except for two. However on my downloaded stacscheck on my laptop it passes all tests so I am really not sure what is accurate. I also used tests of my own and reran through SAT to 3-SAT I used for previous Test 1 tests. There did not seem to be any issue and I made sure to check the Sat and Col solvers to make sure only ones with solutions created reductions with solutions.

Task 3

In this section I will analyse the theoretical and experimental complexity of my implemented algorithms in Tasks 1&2.

ColToSat

When converting a .col colouring graph to a satisfiability problem, the first thing to note is that the number of literals in the satisfiability problem will be equal to the number of nodes in the colouring problem multiplied by the number of colours. Then when analysing the structure in creating the clauses, this can be broken down into three parts.

In the first part, each node must have one clause representing all the colour combinations with that node. This would create the same amount of clauses as nodes. Then for the number of colours that can be associated with each node, there is a clause to represent every permutation that the node can not be. More specifically, each clause in this part is a combination of "not" literals associated with the node and all colours except for one.

In the last part of ColToSat clauses need to be created for each of the edges. An edge clause is created for each combination of the same colour represented by the nodes in that edge.

In mathematical terms, ColToSat can be represented as:
N = number of nodes in col
E = number of edges in col
C = number of colours in col
Clauses = number of clauses in sat
LitALO = number of literals in an ALO clause
LitAMO = number of literals in an AMO clause
LitEdge = number of literals in an edge clause

Clauses = (N) + (N*C) + (E*C)
LitALO = C
LitAMO = C-1
LitEdge = 2
TotalNumLiterals = N * C

SatToThreeSat

SatToThreeSat converts a SAT problem to a 3-SAT problem (a SAT problem with only 3 or less literals in each clause). Coming from a ColToSat conversion, a SAT problem would only have 4 or more literals in one clause if the number of colours in the colouring problem was greater than or equal to four. Otherwise it would not change.

If C (number of colours in the col problem) is greater than or equal to 4, the number of clauses added from SAT will be C-3. New literals are also created to connect the new split clauses.

New3SATClauses = C-3
New3SatVars = (C-4) + 1 *Note: This does not count the new negative literals.*
3SATTotalVars = (N*C) + (C-4) +1
3SATTotalClauses = N + (N*C) + (E*C) + (C-3)

ThreeSatToCol

ThreeSatToCol is the most complex out of the three reductions. The overall number of nodes is going to equal the number of clauses from the Three-SAT plus three times the number of variables in the Three-SAT instance.

Therefore:

NodesEndOfCycle = 3*(3SATTotalVars) + 3SATTotalClauses
NodesEndOfCycle = [3*((N*C) + (C-4) + 1)] + [N + (N*C) + (E*C) + (C-3)]
NodesEndOfCycle = 3NC + 3C - 9 + N + NC + EC + C -3
NodesEndOfCycle = N + 4NC + EC + 4C -12

This may not be entirely accurate however this is the best I can reduce theoretical mathematical breakdown of output node count given an input node count.

Below is an experimental chart showing the growth of node counts before and after a cycle of of ColToSat, SatToThreeSat, and ThreeSatToCol.

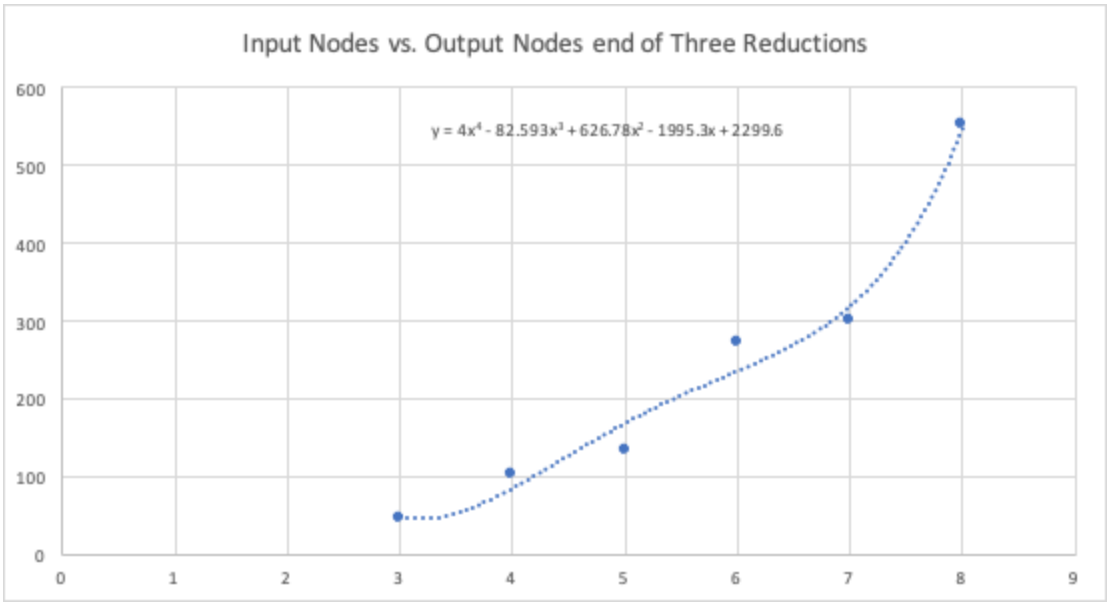

Figure 1: Number of input nodes plotted against the number of output nodes. Trendline shows an order four polynomial growth.

Plotted with a trendline, the increase of nodes before and after the cycle seems to show polynomial growth. The trendline for this plot gives an equation of:

$$y = 4x^4 - 82.593x^3 + 626.78x^2 - 1995.3x + 2299.6$$

When plotted with a simpler polynomial n^2 trendline, the plot looks like this:

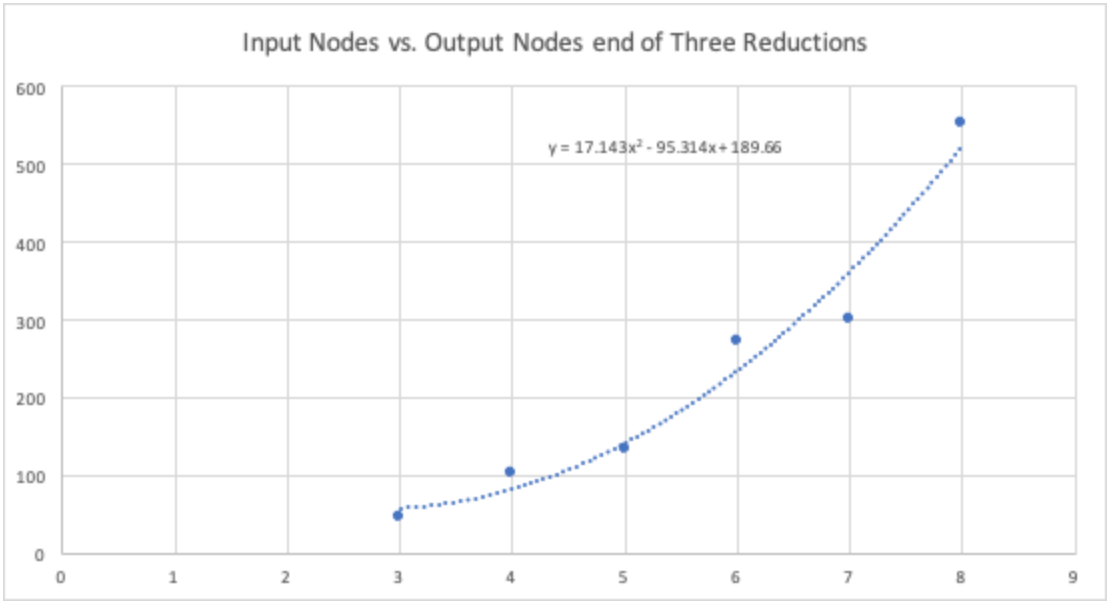

Figure 2: Order 2 polynomial trendline for data points.

The order-2 trendline also accurately depicts the output nodes vs. the input nodes. This is the equation for the given trendline:

$$y = 17.143x^2 - 95.314x + 189.66$$

Experimental analysis of the complexity of the three algorithms is difficult in the sense that my laptop which I have to compute these on does not have much processing power to experiment with Col or SAT instances with too many edges and clauses. However the data that I have been able to attain shows a clear polynomial growth of outputs given these amount of node inputs.

Code in Polynomial Time

ColToSat

The *ColToSat* reduction algorithm I wrote is based on the given algorithm in the appendix as well as a helpful video I found online. The most complex part of the algorithm occurs when creating the "At Most One" clauses. These clauses occur in the section of the algorithm that depicts that each node must have one colour. These clauses consist of a number of "negative" literals that show the node can not be more than one colour at once.

To do this, I go through each of the node-colour combinations three times within for-loops. This gives a theoretical O($(NC)^3$) complexity or rather O(N^3) complexity.

SatToThreeSat

In SatToThreeSat I pretty accurately followed the given algorithm from my lecture notes and the appendix. In the main part of the algorithm, the number of literals in a clause is parsed (if it's > 3) for the number of clauses in the original SAT instance. This would give a theoretical complexity of O(N^2) as the number of literals in a clause increases to be the same amount as the number of clauses itself.

ThreeSatToCol

ThreeSatToCol involved reducing a 3-SAT instance into a kCol instance. In the most complex part of this algorithm we had to connect all of the new vertices to each other (Ys, Xs, NegXs). This involved four for-loop traversals of the variables and the edge clauses in the newly built col problem. Therefore, this algorithm would have a maximum

complexity of O(N^4) especially as the number of edge clauses in the new kCol instance increases.