Polynomial Reductions Report

Contents

1. Task 1

Initially I read through the code and identified which methods had been provided to us. I then broke down task 1 and identified the steps needed to complete it. These made up 3 phases in my mind, the transformation from SAT to 3SAT, the transformation from 3SAT to a graph, and finally colouring the graph and identifying the solving values if they are present. To run the project please make and run prog using the provided makefile.

1.1 SAT to 3SAT

I started this by writing pseudo code for the transformation, to ensure I fully understood the algorithm and what steps I would need to take, this can be found in figure 1.1.1 I used the same approach as described in lectures, splitting large clauses and adding the additional literals, initially I tried a different approach as described in the pseudo code but this was further refined later on.

Figure 1.1.1 Pseudo Code for SAT to 3SAT Transformation
//get number of clauses in sat and store //for each clause

```
//if clause length == 3
//move on to next clause

//if clause length < 3
        //find n = number of literals(clause)
        //for 3-n
                //copy literal 1 and add to clause
        //move on to next clause

//if clause length > 3
        //get number_lits(clause)
        //n <- clause length - 2
        //create n new clauses

        //for clause 1 add first two existing literals
        //add new literal
        //add clause 1 to sat instance
        //add new literals negation to clause 2

//startloop:

        //if curr literal is the  (clause length-1)th
                //add last two literals to current clause
                //add clause to sat instance
                //remove starting clause from sat instance
                //## end loop ##

        //else
                //add curr literal to curr clause
                //add new literal to curr clause
                //add clause to sat instance
                //add its negation to the next clause

//repeat loop

//##loop end##:

//move on to next clause
```

Detailed in this figure is the pseudo code I created to fully understand the steps needed within the transformation from a SAT instance to a 3SAT compliant instance. This essentially meant ensuring each clause contains 3 literals exactly, so any clauses with three literals are left alone. Any clauses with less than 3 literals are added to by repeating existing literals, and clauses with more than 3 literals are broken down into multiple smaller clauses.

I found that on the last step of my implementation that it is fairly difficult to remove a clause from a SAT instance, there is no method provided and I would have had to write my own, I also realised keeping the original sat instance may be useful later on, so instead I reworked my code to instead create and return a new SAT instance.
Following this at a later stage I realised that a lot of the code when reducing a large clause to smaller ones was repeated, I worked through and identified a method that would work to split the clause using one loop, the pseudo code for this is shown in Figure 1.1.2. The only problem I had was deleting a literal, I found that the best way to do this was simply increment the pointer to the list of literals, but had to make sure to return it to the start after the method so it could be destroyed correctly.

Figure 1.1.2 Pseudo code for a cleaner reduction method

```
//if clause length > 3
   //long_clause = copy the clause

   //while long_clause.length >3
      //make a new clause
      //j = find the next joining literal to use
      //add the first two literals from long_clause
      //add a positive  j
      //add clause to sat instance
      //remove first two literals from long_clause
      //insert a negative j

//add a copy of  the original long clause to sat instance
//move back the pointer to literals in the long clause
//free the long clause
```

This pseudo code describes a cleaner way to reduce a long clause to a series of smaller ones. I had to copy the clause as it is split within the method and I wanted to leave the origina sat instance in tact. I also had to add a copy of the clause as the last step, so I could free all the literals that had been skipped over in the last step.

I then continued to construct the code for phase 1 before moving on to tackle the transformation in phase 2 with a similar approach.

1.2 3SAT to Graph

I again started this phase by writing pseudo code for the transformation (shown in figure 1.2.1). I used the lecture slides and worked through example conversions on paper to ensure I properly understood the algorithm prior to implementation, it was useful as it showed a good visual breakdown of the transformation.

Figure 1.2.1 Pseudo code for 3SAT to Graph Transformation

```
//take your 3 sat form
//get total number of variables
//get total number of clauses

//add a node to represent every variable (Y's)
//add a node to represent every +ve literal
//add a node to represent every -ve literal
//add a node to represent every clause

//join every positive literal to its negative one
//join every Y to every other Y
//join every Y representing a variable to all other variables than the one it represents
//join every Clause node to every variable not appearing in its clause

//return the new graph
```

Detailed in this figure is the pseudo code I created to fully understand the steps needed to transform a 3SAT instance to a colourable graph.

There were a few more complicated features in implementing this algorithm, a few points of interest in implementation are listed below.

Mapping Nodes Back to Variables

This is an essential part of not only the graph creation but later solving the graph, as it allows you to figure out how many colours are allowed to be used. I did this by passing over the SAT and adding each unique variable to an array, then passing this array with the sat instance to the transform to graph method. This would allow me to know what variables are represented by what nodes later on in the program. I chose to order my nodes as firstly all Y, then all +X then all -X and finally clause nodes. This meant the number of node for the Y would produce the variable when used as an index for the array. It also meant that later on I could use a mod of the X nodes to get the index to the array, this was quite a neat mapping.

Linking Every Y Node to Every Other Node

Since the edges are not directional, you only need to link each node to every other node once. Now I could have done this by looping through each Y Node, and then looping through again and each time check the nodes are different, and a link doesn't already exist before linking but instead I found an implementation that avoids this testing. I only link nodes with those of a greater value, ensuring that no node ever tries to link with itself, and no link is ever repeated.

Linking Every Clause to Literals not Within The Clause

Linking X nodes to their opposing literal was easy enough as was linking Y nodes to everyone but the one they represented. I linked each clause to each literal by looping through the nodes representing literals, getting their literal equivalent. I then loop through every clause and if the literal does not appear within it, join the clause and literal node.

Following this logic was fairly simple, and the graphs produced are as expected, at this stage it is important to note the ordering used within your graph, will obviously affect the following methods used on it.

1.3 Graph Colouring and Results

Graph colouring was simple enough as I used the supplied subroutine, I also ensured to put a test to make sure there were enough variables for the colouring to work (>=4).

To establish the results, I knew that every Y node's colouring would be true, and since they are n Y nodes and n+1 colours there is one colour that is false. I found the false colour and then compared it to each positive literal to establish a value that would satisfy the clause. The only difference between 3SAT and SAT values, are that the variables added to conform to 3SAT would not apply to SAT as they do not appear. I ensured to note beside these variables that they only apply to the SAT clause.

I also noted that with this algorithm and colouring the Y nodes first that it is always (when I tested it) the n+1 colour that is false. However as this is simply the colouring algorithm used I will not rely on it. As using non clear dependencies is never good, and relying on a certain approach with a colouring algorithm would not be good at all.

The next thing I noticed was how fast my colouring algorithm is, this did initially make me suspicious that it was failing but I investigated and identified the reasons for this further in task 2.

2. Task2

The methods we have been given to read in SAT instances ensure that a correctly formed SAT instance is given. I did test a few case of empty expressions and number 0 variables and they were all rejected. Considering this I then tested the program on different forms of SAT clause and varying lengths, as advised I kept to small numbers to ensure a short time.

2.1 Overall Testing

I have kept all of the test results from my program in a file labelled testresults.txt in the submission. I followed a certain pattern of tests starting off with simple results and getting more complicated. I wanted to cover the following cases:

- Single Clause SAT

- Multiple Clause SAT
- Single literal clauses
- Ordered Numbers
- Two literal clauses
- Three literal clauses
- Bigger than 3 literal clauses
- Negative and Positive literals
- All negative literals
- All positive literals
- SAT with less than 4 variables
- SAT that cannot be solved
- SAT that can be solved
- Repeated Clauses

I did perform 18 different tests with a combination of these as results shown in figure 2.1.2 I have shown a couple of examples in further detail in figure 2.1.1. I verified all of the tests I performed on paper to ensure correct values were given to satisfy the initial expression and that the transformation did show both the correct 3SAT form and correct graph. The graphs especially for the longer expressions took a long time to work through.

Figure 2.1.1 Worked Through Examples

Note the italic text is the program output

I provided the SAT: {[2,-3,4][-3,-10]} this was correctly read in:

SAT form:
{[2, -3, 4], [-3, -10]}

This 3SAT is an equivalent representation of the given clause, adding a repeated -3 to the second clause.

Transformed to 3SAT form:
{[2, -3, 4], [-3, -10, -3]}

The 3SAT is then transformed to a graph, I checked the following points, using colours to demonstrate their appearance on the graph.

- There are 4 variables and 2 clauses so there should be 14 nodes
- The first 4 nodes should all be connected to each other.
- Each Y node should be connected to all X nodes (6) not representing the same literal
- Each X node should be connected to all Y nodes (3) all not representing the same literal.
- Each X node should be connected to its opposite.
- Clause 1 node should be connected to all X nodes (5) not contained in it.

- Clause 2 should be connected to all X nodes (6) not contained in it.

Transformed to Graph:

Undirected graph 14 vertices, adjacencies:
0: 1 2 3 5 6 7 9 10 11
1: 0 2 3 4 6 7 8 10 11
2: 0 1 3 4 5 7 8 9 11
3: 0 1 2 4 5 6 8 9 10
4: 1 2 3 8 13
5: 0 2 3 9 12 13
6: 0 1 3 10 13
7: 0 1 2 11 12 13
8: 1 2 3 4 12 13
9: 0 2 3 5
10: 0 1 3 6 12 13
11: 0 1 2 7 12
12: 5 7 8 10 11
13: 4 5 6 7 8 10

As there are 4 variables I should be permitted to use n + 1 colours:

Permitted to use 5 colours

It has provided the following variables values as solutions:

Variable 2 should be 1.
Variable 3 should be 1.
Variable 4 should be 1.
Variable 10 should be 0.

Placed back into the sat instance you would get:

{[2, -3, 4], [-3, -10]} = {[1, -1, 1], [-1, -0]} = (1 ^ 0 ^ 1) V (0 ^ 1) = Evaluates true.

As the 3SAT uses the same variables and:

[-3, -10, -3] = [-1,-0,-1] = (0 ^ 1 ^ 0) = Evaluates true

Hence the solution is correct.

I followed the process repeatedly for the further tests show in the table below:

Figure 2.1.2 Table of Test Results

SAT Expression Input	3SAT Expression	Values (of positive literal)
{[-2]}	{[-2, -2, -2]}	-
{[3]}	{[3, 3, 3]}	-
{[2,3]}	{[2, 3, 2]}	-
{[-5,-11]}	{[-5, -11, -5]}	-
{[6,-30]}	{[6, -30, 6]}	-
{[-5][5]}	{[-5, -5, -5], [5, 5, 5]}	-
{[2,3,4]}	{[2, 3, 4]}	-
{[2,3,4,5]}	{[2, 3, 6], [-6, 4, 5]}	All 1
{[2,3,4][5]}	{[2, 3, 4], [5, 5, 5]}	All 1
{[-1][1][2,3,4]}	{[-1, -1, -1], [1, 1, 1], [2, 3, 4]}	Not able to be coloured
{[2,-4,-8][3,-4,8]}	{[2, -4, -8], [3, -4, 8]}	All 1
{[-2,-3,-4][2,3,4][5]}	{[-2, -3, -4], [2, 3, 4], [5, 5, 5]}	2 -> 1 3 -> 1 4 -> 0 5 -> 1
{[1,2,3][4,5,6][7,8,9]}	{[1, 2, 3], [4, 5, 6], [7, 8, 9]}	All 1
{[5,3,-2,-4,6,7,8,9,-10]}	{[5, 3, 11], [-11, -2, 12], [-12, -4, 13], [-13, 6, 14], [-14, 7, 15], [-15, 8, 16], [-16, 9, -10]}	All 1
{[1,2,3][4,5,6][7,8,9][-9,-8,-7][-6,-5,-4][-3,-2,-1]}	{[1, 2, 3], [4, 5, 6], [7, 8, 9], [-9, -8, -7], [-6, -5, -4], [-3, -2, -1]}	1 -> 1 2 -> 1 3 -> 0 4 -> 1 5 -> 1 6 -> 0 7 -> 1 8 -> 1 9 -> 0
{[1,2,5][-8,9,11][8,12,13][-8,-12,13]}	{[1, 2, 5], [-8, 9, 11], [8, 12, 13], [-8, -12, 13]}	All 1

{[2,-14,-13,-12,12,-1,14,16,2]}	{[2, -14, 17], [-17, -13, 18], [-18, -12, 19], [-19, 12, 20], [-20, -1, 21], [-21, 14, 22], [-22, 16, 2]}	All 1
{[3,4,5][-5,6,9][1][-2,3][-1][2,7, 6][3,4,5]}	{[3, 4, 5], [-5, 6, 9], [1, 1, 1], [-2, 3, -2], [-1, -1, -1], [2, 7, 6], [3, 4, 5]}	Not Able to be Coloured
{[2,2,2]}	{[2, 2, 2]}	-
{[2,3,3][-3,-3,4][-4,5,6]}	{[2, 3, 3], [-3, -3, 4], [-4, 5, 6]}	All 1
{[1,2,3,4,4,4,4]}	{[1, 2, 5], [-5, 3, 6], [-6, 4, 7], [-7, 4, 8], [-8, 4, 4]}	All 1

2.2 Ordering and Point of Interest

When running my tests I noticed that the colouring algorithm was very fast, as we are told it is worst case exponential time complexity I did expect it to be much slower, especially when peers in the lab running tests were visibly waiting for ones with any more than 9 variables, and I had tested mine on cases of 100 variables plus and it is almost instant to the human eye. I have included these tests in the test result file.

Considering that every transformation is polynomial apart from the colouring I considered this part first. On closer inspection I realised that the colouring algorithm will run colouring the Y nodes first as per my order, this means that already each is given a different value and hence the remaining graph to try and colour will be greatly reduced, as for a satisfiable graph each Y node will need to be a different colour. Peers who had a different node order were getting a much slower result I confirmed this by briefly switching the node ordering and the result was indeed much slower, with visible delays on any clauses with a number of variables in the low teens or higher. I think this is a really interesting example on how both a small change in approach can make a massive difference to your algorithm and also on how one can heuristically make the average case for even worst case exponential algorithms very much quicker.

3. Task3

A polynomial function is one where:

$$T(n) = O(n^{k})$$

Following I have considered the steps within each transformation and why both transformations are polynomial time complexity.

3.1 SAT to 3SAT

In the best case for the transformation the function is still linear, as the SAT expression must be iterated over once. This means there will be n constant operations and hence overall linear complexity, which is a polynomial complexity as:

$$O(n) = O(n^1)$$

In the worst case all variables will be in one clause and the clause will be larger than 3 and have to be split up into smaller clauses. For n variables you will need to split the clause n-3 times.

$$O(n-3) \;=\; O(n) \;=\; O(n^1)$$

Again this is upper bounded by a polynomial function, and hence the time complexity of the function in respect to n in the best and worst case is polynomial:

$$T(n) \;=\; O\,(n^1)$$

3.2 3SAT to Graph

For this function I will consider the worst case complexity to find the upper bound of complexity. There are really 2 variables affecting the complexity, n (the number of variables) and c the number of clauses. I am considering complexity in relation to n so will assume a fixed size c for this consideration

Initially we iterate through the sat instance for every literal and note every existing variable, this will be:

$$O\,(n)$$

We then calculate the number of nodes needed for the graph and create it, both constant time operations.

The next step is to join every Y vertices to every other Y vertices. As the graph is undirected every Y node only needs to be joined to the ones after it and the last node does not need to be joined as it will already have been joined by all the other nodes, so:

$$O\,(\tfrac{n-1}{2}) = \; O\,(n)$$

Every Y node will need to be joined to every X node that is not representing the same variable value as that Y node, to establish which X nodes will need to be joined you will have to iterate over every Y node which will be:

$$O(n)$$

and then for each Y node iterate through every x node, because even the ones you don't have to join will be tested for to see if you would have to join them anyway, this will be:

$$O((n)(2n)) = O(2n^2) = O(n^2)$$

Every X node is then joined to its opposite node this will be:

$$O(n)$$

Every clause node is joined to every node apart from the ones it contains this is logically:

$$O(2n - 3)$$

however as we need to test the nodes even if we don't join to them it will actually be:

$$O(2n)$$

Evaluating these individual complexities you can establish that the overall complexity is dominated by the:

$$O(n^2)$$

term. This means that the equation is:

$$T(n) = O(n^2)$$

Again the transformation is upper bounded by a polynomial expression, and hence can be consider to be polynomial complexity.

4. Task4

4.1 Graph to SAT Method

My method for this transformation relies on the fact that the graph is constructed with Y nodes first, followed by X+ve nodes, X-ve nodes and finally the clause nodes. It is interesting to

consider whether is a method that can transform any graph back to 3SAT instance and that is something that I will think about more later on.

My method works by iterating over the nodes until the first node is found that is not connected to node 0. This will allow me to establish the number of variables for the graph, as the first non-connected node will be the first X+ve node. This means the index of this node is equivalent to the number of variables in the graph. Using this information you can then find the number of clauses to use for your sat instance, and the node which represents each one.
Using the node for the clause, you can then iterate through the X nodes and for every X node not connected to the clause add it to the clause. You can slightly improve the algorithm by breaking from checking literals when a clause has the maximum number of 3.
The method was fairly easy to implement I did consider whether it was possible to use a function that would transform any graph to a sat instance, using simply the number of nodes and edges between them, to get the corresponding sat instance when the rules described for creating the graph had been followed as specified. With further time it would be interesting to investigate this.

4.2 Comparing SAT sizes

When you construct the graph from the 3SAT clause there is no way to represent duplicate literals within a clause, it makes sense that any duplicate literals cannot be retrieved. This does not affect the result of the expression as each literal is joined by an or, and we know that:

(A V A) = (A)

This means that any duplicates within the original SAT instance will not be found so the reproduced SAT will be less any duplicates. It is also impossible to tell which variables were added at the 3SAT stage and which were present before, so it the reconstructed SAT instance will have the same number of variables as the 3SAT transformation had.
To identify this pattern in real time I ran the same tests as in Task 2 and compared the original SAT instance and then 3SAT instances to the reconstructed one, (shown in figure 4.2.1).

Figure 4.2.1 Table Comparing Reconstructed SAT Clauses

SAT Expression Input	3SAT Expression	Reconstructed SAT
{[-2]}	{[-2, -2, -2]}	{[-1]}
{[3]}	{[3, 3, 3]}	{[1]}
{[2,3]}	{[2, 3, 2]}	{[2, 3]}
{[-5,-11]}	{[-5, -11, -5]}	{[-2, -3]}
{[6,-30]}	{[6, -30, 6]}	{[2, -3]}

{[-5][5]}	{[-5, -5, -5], [5, 5, 5]}	{[-1], [1]}
{[2,3,4]}	{[2, 3, 4]}	{[3, 4, 5]}
{[2,3,4,5]}	{[2, 3, 6], [-6, 4, 5]}	{[5, 6, 7], [8, 9, -7]}
{[2,3,4][5]}	{[2, 3, 4], [5, 5, 5]}	{[4, 5, 6], [7]}
{[-1][1][2,3,4]}	{[-1, -1, -1], [1, 1, 1], [2, 3, 4]}	{[-4], [4], [5, 6, 7]}
{[2,-4,-8][3,-4,8]}	{[2, -4, -8], [3, -4, 8]}	{[4, -5, -6], [6, 7, -5]}
{[-2,-3,-4][2,3,4][5]}	{[-2, -3, -4], [2, 3, 4], [5, 5, 5]}	{[-4, -5, -6], [4, 5, 6], [7]}
{[1,2,3][4,5,6][7,8,9]}	{[1, 2, 3], [4, 5, 6], [7, 8, 9]}	{[9, 10, 11], [12, 13, 14], [15, 16, 17]}
{[5,3,-2,-4,6,7,8,9,-10]}	{[5, 3, 11], [-11, -2, 12], [-12, -4, 13], [-13, 6, 14], [-14, 7, 15], [-15, 8, 16], [-16, 9, -10]}	{[15, 16, 17], [19, -17, -18], [21, -19, -20], [22, 23, -21], [24, 25, -23], [26, 27, -25], [28, -27, -29]}
{[1,2,3][4,5,6][7,8,9][-9,-8,-7][-6,-5,-4][-3,-2,-1]}	{[1, 2, 3], [4, 5, 6], [7, 8, 9], [-9, -8, -7], [-6, -5, -4], [-3, -2, -1]}	{[9, 10, 11], [12, 13, 14], [15, 16, 17], [-15, -16, -17], [-12, -13, -14], [-9, -10, -11]}
{[1,2,5][-8,9,11][8,12,13][-8,-12,13]}	{[1, 2, 5], [-8, 9, 11], [8, 12, 13], [-8, -12, 13]}	{[8, 9, 10], [12, 13, -11], [11, 14, 15], [15, -11, -14]}
{[2,-14,-13,-12,12,-1,14,16,2]}	{[2, -14, 17], [-17, -13, 18], [-18, -12, 19], [-19, 12, 20], [-20, -1, 21], [-21, 14, 22], [-22, 16, 2]}	{[12, 14, -13], [16, -14, -15], [18, -16, -17], [17, 19, -18], [21, -19, -20], [13, 22, -21], [12, 23, -22]}
{[3,4,5][-5,6,9][1][-2,3][-1][2,7,6][3,4,5]}	{[3, 4, 5], [-5, 6, 9], [1, 1, 1], [-2, 3, -2], [-1, -1, -1], [2, 7, 6], [3, 4, 5]}	{[8, 9, 10], [11, 12, -10], [13], [8, -14], [-13], [11, 14, 15], [8, 9, 10]}
{[2,2,2]}	{[2, 2, 2]}	{[1]}
{[2,3,3][-3,-3,4][-4,5,6]}	{[2, 3, 3], [-3, -3, 4], [-4, 5, 6]}	{[5, 6], [7, -6], [8, 9, -7]}
{[1,2,3,4,4,4,4]}	{[1, 2, 5], [-5, 3, 6], [-6, 4, 7], [-7, 4, 8], [-8, 4, 4]}	{[8, 9, 10], [11, 12, -10], [13, 14, -12], [13, 15, -14], [13, -15]}

From considering these we can establish that the number of variables is equal to that of the 3SAT clause, the number of literals is equal (excluding duplicates in the same clause) to those in the 3SAT clause, it does not matter if the duplicates were in the original SAT either they are still

excluded from the reconstructed clause. The only time duplicates in the original sat will be represented is if they were contained within a clause larger than 3 in the original SAT expression - (see the last test) - they will then appear in separate clauses in the 3SAT and hence they are represented in the graph, and will be found within the reconstruction.
Ordering is not preserved and nor would we expect it to be. The actual numbers used to refer to the variables are not preserved either but again there is no way to preserve them. The reason for the varying numbers representing them is the fact that a previous iterator is used count the number of variables is used to create the new variables, this has no real effect as they are just used to represent another variable.

The patterns we see to have emerged relate back to what was discussed before performing the tests, and are as expected. Regardless of the difference between the SAT expressions, each of them still represents the same conditions to be satisfied, in the same way that if you can solve a 3SAT you can then solve the SAT. If you can solve the reconstructed SAT you should be able to solve either the 3SAT or a SAT.

4.3 Transformation Complexity

The transformation from a graph to 3SAT should also be of polynomial complexity.

Firstly we need to iterate over the nodes to find the first node that is not joined to the 0th node, thus establishing the number of variables, as we make the same number of comparisons as variables it will be:

$$O\,(n)$$

I will then calculate the number of clauses and for each clause node I need to iterate through every X node and add the ones not joined to the clause node to the new clause. If we consider the number of clauses to be a constant C.

$$O\,(C\ (2n))$$

However as we establish complexity with regards to n, and we assume C to be constant we can disregard it as we would the 2 and hence:

$$T(n)\ = O\,(n)$$

when c the number of clauses is constant. Again this is another transformation in polynomial time.

5.Critical Evaluation

I have genuinely found this practical incredibly interesting and on further investigation have realised how many problems a SAT solver can be used within, something I was never aware of before. With more time I would consider further trying to establish an algorithm for any graph to be converted to a SAT instance, regardless of node order. I would also look at the ordering of the graph further and with extensive testing I could perhaps find out more about its effects, on the best, average and worst case time complexities. Discovering this by accident almost, has highlighted to me the importance of considering every aspect of an approach when designing algorithms and I will try and bear this in mind in future.

I do feel that I did try to evolve my code from the initial methods built on pseudo code to more refined and less resource expensive techniques. There are areas I am sure I could have improved further and given more time perhaps refinement is another aspect I could consider.