CS3052: Practical 2 Report
Polynomial-Time Reductions

Matric Number: 170002815

05th May 2020

1 Submission Code Information

1.1 Structure

The submission directory contains this report as a PDF, and also contains the `src` directory. Within the `src` directory, the three reduction programs can be found, and four subdirectories can be found. The `example_problems` subdirectory contains all example problems used for testing the reductions, the `marshallers_unmarshallers` subdirectory contains Python scripts to marshall and unmarshall probelms, the `output` subdirectory contains the ouput files generated during the testing of my reductions, and the `sat_solver` subdirectory contains the Python script used to check if CNF problems can be satisfied.

1.2 Usage

To use my programs, please first ensure the latest version of Python3 is installed - you can find instructions for your operating system here: https://www.python.org/downloads/

You will also require PySAT if you would like to run the SAT solver, which you can find instructions for installation here: https://pysathq.github.io/installation.html

To run each reduction program, you will need to execute the Python script using your system's Python command (`python` or `python3`), and input a problem file into `stdin`. The simplest way of doing so is using the `cat` command along with a UNIX pipe, as shown below.

- $cat \langle in\_file \rangle \mid python \langle reduction\_program \rangle$

The graph colour to SAT reduction must include the number of colours used in the graph colouring problem as a command line argument, and should therefore be executed in the following way.

- $cat \langle in\_file \rangle \mid python \langle number\_of\_colours \rangle \langle reduction\_program \rangle$

2 Overview

The objective of this practical was to implement three polynomial reductions as separate programs, runnable from the command-line. Each reduction required a proof of correctness, evidence for polynomial execution time, and a program implementation. The required reductions are as follows:

1. SAT to 3SAT
2. 3SAT to graph colouring
3. Graph colouring to SAT

To accompany the reductions, the specification required an input/output language to be sourced or created for the communication of graphs and Boolean algebra equations. Section 4 of this report discusses the interface languages I decided to use for my reduction programs.

The testing of the my system and how this testing shows my work satisfies the specification is discussed in Section 6 of this report.

The report is completed by an evaluation and conclusion, which reflect back on the learning objectives of the practical, and also evaluate how successful I have been in relation to the specification.

3 Reduction Proofs

3.1 SAT to 3SAT

3.1.1 Overview

SAT and 3SAT are both Boolean satisfiability problems, where the Boolean formula in question is in Conjunctive Normal Form (CNF), however 3SAT has the added constraint that each clause in the formula has at most *three* literals.

The basis for my reduction was to substitute each clause in the provided formula with a 3SAT compliant expression, which is equisatisfiable. By definition, if each substituted expression is 3SAT compliant and equisatisfiable, then the overall formula is also 3SAT compliant and equisatisfiable to the original.

3.1.2 Three or Fewer Literal Clauses

Clauses with three or fewer literals are already 3SAT compliant and therefore do not need to be converted. In some definitions of 3SAT, *all* clauses must have three literals; in this case, one literal from each provided clause can be duplicated and applied in conjunction with the rest of the clause until this said clause has three literals.

3.1.3 Four or More Literal Clauses

To convert clauses with four or more literals into 3SAT compliant expressions, new atoms and their negations are introduced as "fresh" literals to create a new set of "entangled" clauses in CNF, each with at most 3 literals. The literals are described as *fresh*, as they are not represented in the provided formula or any previously-manipulated clauses, and the new clauses are described as *entangled*, as the truth value of one clause affects the truth value of at least one other in such a way that all clauses are interlinked.

The workings of this conversion are influenced by the Indian Institute of Technology Kharagpur's notes on the SAT to 3SAT reduction [1], and the general case is shown below.

- Provided Clause: $(l_1 \vee l_2 \vee \ldots \vee l_n)$
- Substitute Clauses: $(l_1 \vee l_2 \vee z_1) \wedge (\neg z_1 \vee l_3 \vee z_2) \wedge (\neg z_2 \vee l_4 \vee z_3) \wedge \ldots \wedge (\neg z_{n-4} \vee l_{n-2} \vee z_{n-3}) \wedge (\neg z_{n-3} \vee l_{n-1} \vee l_n)$, where z_1 through z_{n-3} and their negations are fresh literals.

To prove that the substitution is equisatisfiable to the original clause, all cases must be considered.

The first case is where no literals from the original clause are true (l_1 through l_n). In this case, each clause can only be satisfied by the fresh literals introduced in the substitution, however as the negation of each literal is contained in one of the adjacent clauses to itself, it is impossible to satisfy all clauses with any valid assignment to the fresh literals.

The reason for this is that for the first clause to be satisfied, z_1 must be true, meaning the second clause can only be satisfied by z_2 being true. This domino effect applies for all but the last clause, meaning all fresh variables are required to be true, however the final clause can only be satisfied if the last fresh variable is false. This is a contradiction as the last literal is a negation of a variable required to be true earlier in the assignment, and therefore shows that it is impossible to satisfy all clauses with any valid assignment to the fresh literals when all literals from the provided clause are false.

In the case that one or more of l_1 through l_n are true, at least one assignment to the new atoms in the fresh literals will satisfy the substitute clauses. If l_n or l_{n-1} are true, then all fresh variables can assigned to true, satisfying all clauses. If l_1 or l_2 are true, then all fresh variables can be assigned to false, which also satisfies all clauses. If any of l_3 through l_{n-2} are true, all atoms to the left of the clause containing the true literal can be assigned to true, and all atoms to the right can be assigned to false - this satisfies all clauses to the left and right of the clause containing the true literal, and the clause itself is satisfied by the true literal. If there is more than one true literal in the provided clause, the assignment for left-is-true, right-is-false still holds, as the literal(s) now considered to be true do(es) not dissatisfy any clause(s)

As the substitute clauses have at most 3 literals in disjunction in each clause, and the clauses are in conjunction with each other, the substitute clauses are a 3SAT problem. Repeating this substitution for every clause in the provided formula will return a 3SAT problem, as conjugating a 3SAT problem with another 3SAT problem will return a 3SAT problem by definition.

As the substituted clauses are each equisatisfiable to the provided clause, the formula as a whole after all substitutions are complete is equisatisfiable to the original formula.

3.2 3SAT to Graph Colouring

3.2.1 Inspiration

I believe the reduction from 3SAT to graph colouring is less intuitive and straight forward than the other two reductions required by the practical specification. The University of Liverpool's Computer Science lecture material [2] and Udacity's YouTube videos on algorithms [3] were used to aid the development of my answer, and their influence may be seen throughout.

3.2.2 Reduction Alteration

Before beginning the reduction, I decided to alter the reduction I was completing from "3SAT to graph colouring" to "3SAT to 3-colour graph colouring". This means that all 3SAT cases will be represented in terms of a graph colouring problem which has a palette of only three colours (i.e. only three colours can be used). As 3-colour graph colouring is still a graph colouring problem, my solution still satisfies the requirements of the specification.

There are two main reasons for altering the reduction I am aiming to complete: firstly, standardising the output of the reduction is very useful for the program-implementation, as the palette size does not need to be calculated, and secondly, there is a wider variety of online resources for this reduction, allowing me to verify my solution to a greater extent.

3.2.3 Setup

The palette used to demonstrate the reduction can be seen in Figure 1. The colours of green, red and blue are used, to represent the concepts True, False and Ground respectively, however as the colour choices are arbitrary, the reduction will hold true for any palette of three different colours. The assignment of True, False and Ground are also arbitrary, and therefore the reduction will also work for any assignment chosen.

True is chosen to represent the value of true for a node, False for the value of false, and Ground for the value of neither false nor true. The concept of Ground may be unintuitive to introduce, however the reduction is set up in such a way that nodes relating to Boolean values are connected to the Ground palette node, meaning the true-false nature of the values must be preserved in order for the graph to be properly coloured. This means Ground is purely a concept for within the reduction and does not influence the true-false nature of the problem.

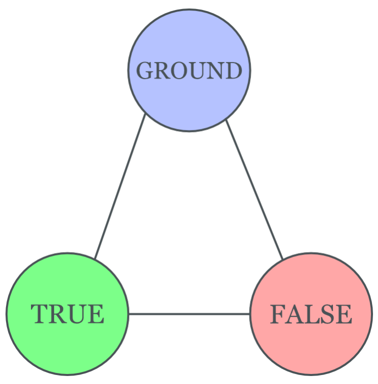


Figure 1: Reduction Colour Palette

To begin the reduction, each clause in the 3SAT formula is converted to a clause-gadget. A clause-gadget in this reduction is a sub-section of the graph which is equisatisfiable to a clause in the provided 3SAT formula.

3.2.4 Single-Literal Clause-Gadget

To build a clause-gadget equisatisfiable to a single-literal clause, the reduction colour palette graph can be used. By substituting the green True node to the value of the literal, the graph can only be coloured when this value is true, and cannot be coloured when this value is false. This means the 3-colour graph colouring problem associated with this clause-gadget is equisatisfiable to a single literal clause in a 3SAT problem. Figure 2 shows the visual representation of this clause-gadget, with l_1 representing the value of the single literal.

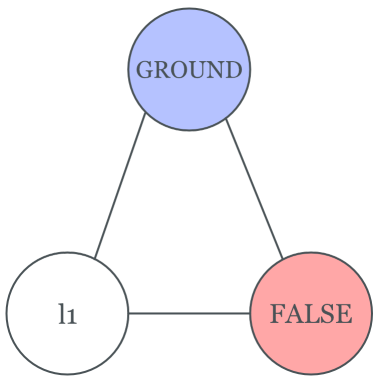


Figure 2: Single-Literal Clause-Gadget

3.2.5 Two-Literal Clause-Gadget

To build a clause-gadget for clauses with two literals, an OR-gadget can be used to re-create the disjunction of the two terms from the 3SAT formula in a graph-colouring scenario. The OR-gadget used in this reduction can be seen in Figure 3.

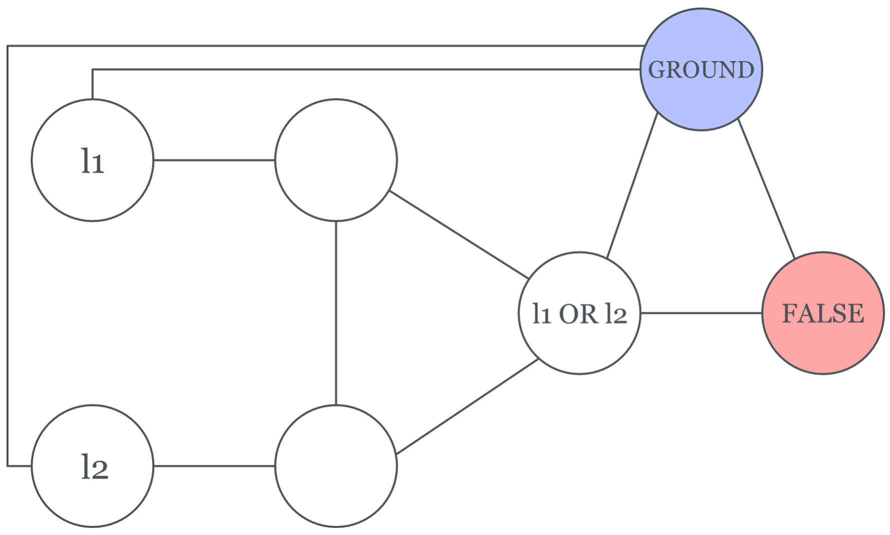


Figure 3: OR-Gadget (Two-Literal Clause-Gadget)

The OR-gadget pictured in Figure 3 is equisatisfiable to the SAT problem for the formula $(l_1 \vee l_2)$. To explain why, consider each case:

- **Case 1 - Both l_1 and l_2 are True**: The SAT problem is satisfied by the definition of OR, and the graph can be coloured by setting the node adjacent to l_1 as Ground and the node adjacent to l_2 as False. This forces the node labelled "l_1 OR l_2" to be True, which does not match the colour of the connected Ground and False nodes, therefore satisfying the OR-gadget graph. Figure 4 shows a visual representation of a satisfying assignment for this case.

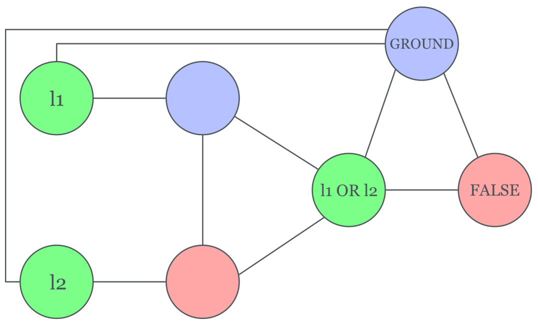


Figure 4: OR-Gadget Case 1

- **Case 2 - l_1 is False; l_2 is True**: The SAT problem is satisfied by the definition of OR, and the graph can be coloured by setting the node connected to l_1 to Ground and the node connected to l_2 as False. The node labelled "l_1 OR l_2" can then be coloured True to satisfy the colouring problem. Figure 5 shows a visual representation of a satisfying assignment for this case.

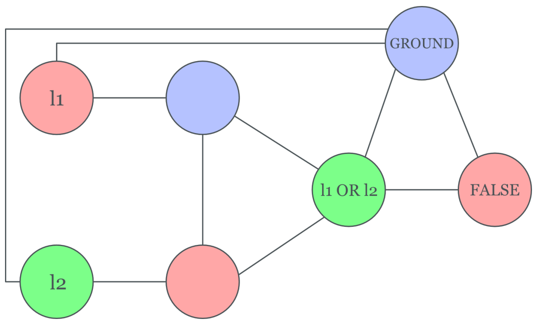


Figure 5: OR-Gadget Case 2

- **Case 3 - l_1 is True l_2 is False**: The SAT problem is satisfied by the definition of OR, and the graph can be coloured by setting the node connected to l_2 to Ground and the node connected to l_1 as False. The node labelled "l_1 OR l_2" can then be coloured True to satisfy the colouring problem. In fact, since the labelled nodes l_1 and l_2 are arbitrary, this can be considered to be the same as Case 2. Figure 6 shows a visual representation of a satisfying assignment for this case.

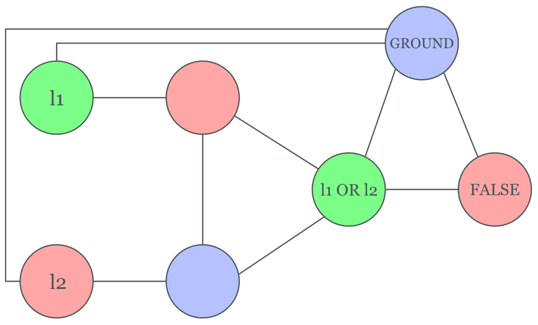


Figure 6: OR-Gadget Case 3

- **Case 4 - Both l_1 and l_2 are False**: The SAT problem is unsatisfiable by the definition of OR, and the graph cannot be coloured with a palette of three colours. The reason for this is that the node labelled "l_1 OR l_2" must be coloured True (green) to avoid matching the connected Ground or False nodes. This leads on to force the two unlabelled nodes to be either False or Ground, and since they are connected, one node must be False and one node must be Ground to satisfy the colouring problem up to this point. However, since l_1 and l_2 are both false, the graph cannot be coloured using three colours.

NOTE: The cases where l_1 and/or l_2 are coloured to represent Ground are not considered, as this would dissatisfy the colouring problem by having two nodes of the same colour involved in the same edge.

3.2.6 Three-Literal Clause-Gadget

Through the associativity of Boolean algebra, SAT clauses with three literals can be converted to a two-term disjunction where one term is a two-literal disjunction, i.e. $(l_1 \vee l_2 \vee l_3)$ becomes $((l_1 \vee l_2) \vee l_3)$. Using this feature of Boolean algebra, three-literal clauses can be expressed using two OR-gadgets, where the output of one is one of the inputs to the other. Figure 7 shows how two OR-gadgets can be arranged to build a three-literal clause-gadget.

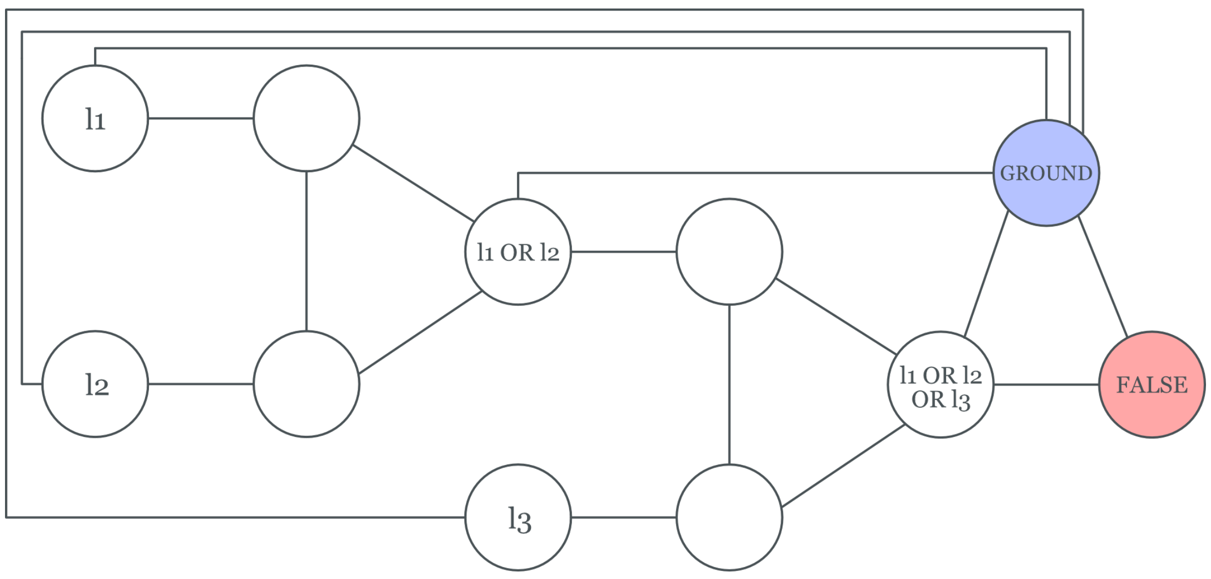


Figure 7: Three-Literal Clause Gadget

The first OR-gadget in Figure 7 creates a disjunction between l_1 and l_2, with the output being fed into another OR-gadget which creates a disjunction of this output with l_3.

To ensure that this clause-gadget can be satisfied by any literal inside the clause, the output of the first OR-gadget (labelled "l_1 OR l_2") is not connected to the False palette node. If this connection was not removed, the case where l_3 equating to true and l_1 and l_2 equating to false would not be colourable, and would therefore not be satisfiable when the 3SAT problem would be.

As the three-literal clause-gadget shown in Figure 7 uses two OR-gadgets, which were validated to be equisatisfiable by considering all cases in Section 3.2.5, the three-literal clause-gadget is also equisatisfiable to a three-literal SAT clause.

3.2.7 Combining Clause-Gadgets

The one-, two- and three-literal clause-gadgets can be used to represent each and every clause in a three-colour graph colouring problem, however when combining clauses which contain literals that are the same, or are negations of each other, adjustments should be made.

Literals which are the same should be represented by the same node, meaning all nodes representing literal l_i must be combined into a single node for the final graph, including all edges. Figure 8 shows an example of two clause-gadgets both containing the same literal being combined into a graph.

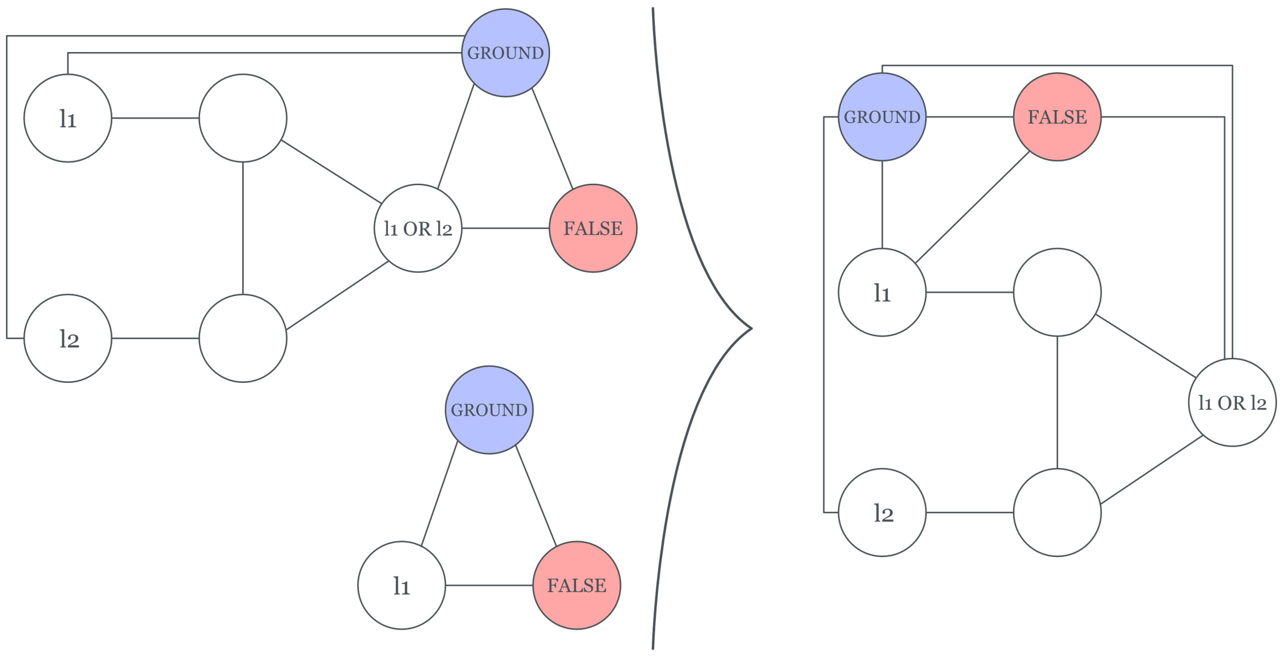


Figure 8: Example of two clause-gadgets being combined, both of which contain the same literal, l_1.

Literals which are negations of each other must also be considered when combining clause-gadgets, as the truth-value of one directly affects the truth-value of the other. To represent this in a three-colour graph-colouring scenario, each literal which has a negation should be connected to each other by an edge; as all literals are connected the Ground node, only one of the two nodes connected to each other can be true without dissatisfying the colouring problem - this reflects the definition of OR in Boolean algebra. Figure 9 shows an example of a literal and its negation being connected when they are used in different clause-gadgets.

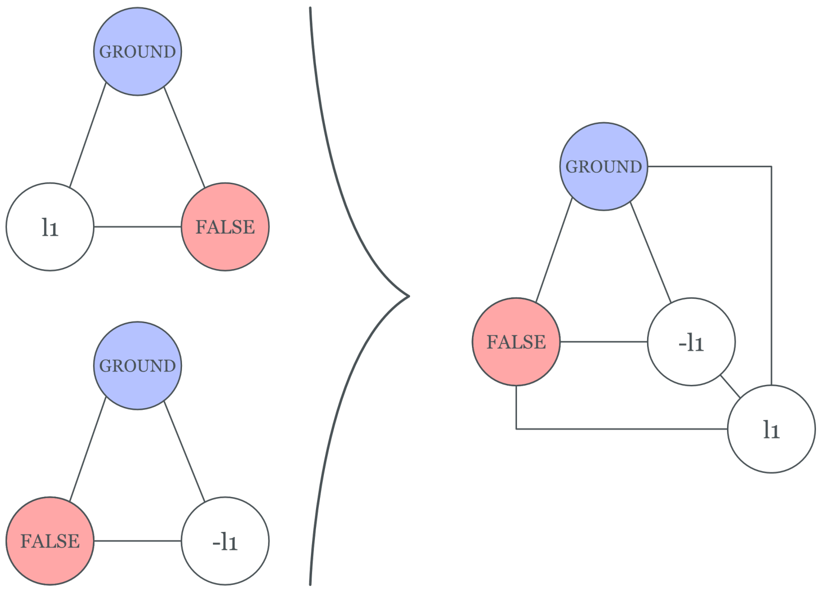


Figure 9: Example of two clause-gadgets being combined, each containing a negation of a literal in the other.

Since all clause-gadgets are equisatisfiable to the clauses they represent, and since the combining of identical nodes as well as the connection of negations preserves the true-false nature of Boolean algebra, completing this transformation is a successful reduction from 3SAT to graph-colouring.

3.2.8 Output Optimisation and Example

As the Ground and False nodes are used in every clause-gadget, they can be shared between the gadgets to reduce the number of nodes and edges required in the problem. For example, the output nodes of two two-clause gadgets can both be connected to the same Ground and False nodes - this is demonstrated in Figure 10, which represents $((l_1 \lor l_2) \land (l_3 \lor l_4))$ as a three-colour graph colouring problem.

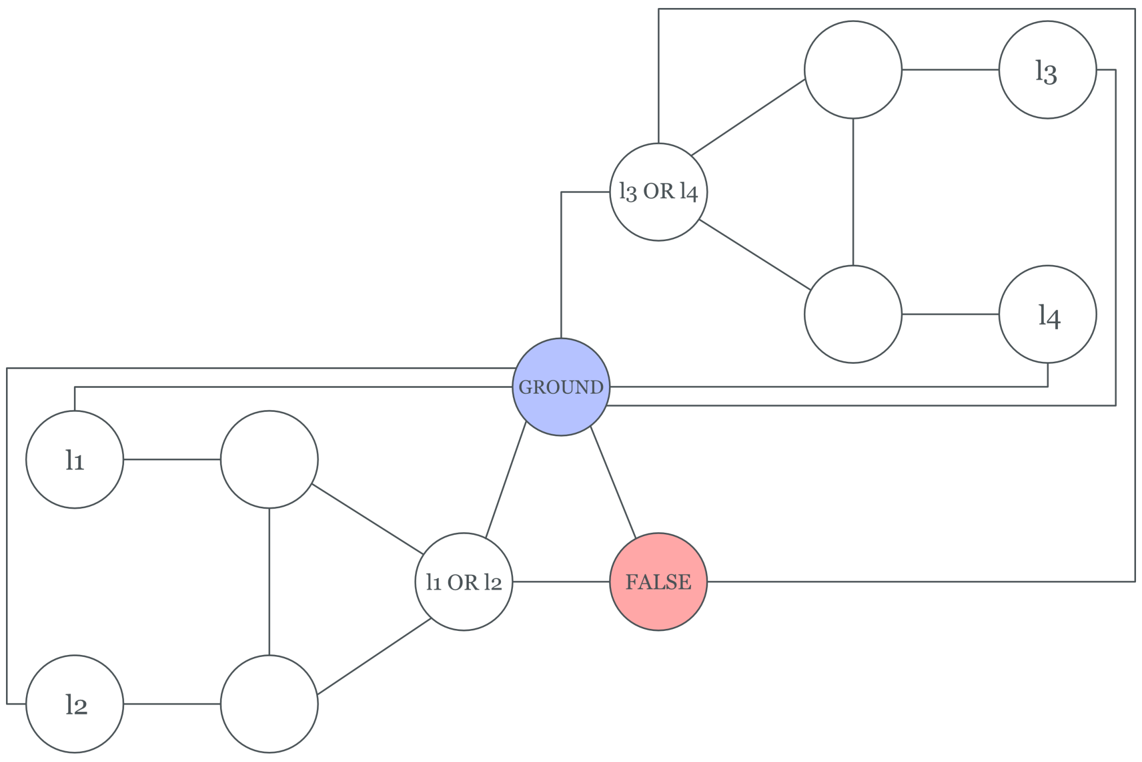


Figure 10: $((l_1 \lor l_2) \land (l_3 \lor l_4))$ as a three-colour graph colouring problem.

3.3 Graph Colouring to SAT

3.3.1 Overview

In my opinion, the most intuitive way to reduce the graph colouring problem to SAT is to express the exact problem by using Boolean algebra. The satisfiability of a graph colouring problem is described by the following three conditions:

1. Each graph node has at least one colour.
2. Each graph node has no more than one colour.
3. No two adjacent graph nodes have the same colour.

NOTE: My method to complete this reduction is influenced by the University of Texas' Computer Science notes on complexity reductions [4].

3.3.2 Graph Colouring Problem to Predicate Logic

In fact, describing these points as Boolean algebra formulae is fairly trivial, and these formulae are shown below. The case described is the general case for a graph with n nodes, m colours and p edges. N_{i,C_j} represents the truth value of graph node i having the colour j, where $i \in \{1 \ldots n\}$ and $j \in \{1 \ldots m\}$. $E_{a,1,C_b}$ represents the truth value of the first node involved in edge a having the colour b, where $a \in \{1 \ldots p\}$, and $E_{a,2,C_b}$ represents the same, except the second node involved in edge a is considered rather than the first.

1. $\forall N(N_{C_1} \vee N_{C_2} \vee \ldots \vee N_{C_m})$
2. $\forall N(\forall x(\forall y(\neg(N_{C_x} \wedge N_{C_y}) \vee (x = y))))$
3. $\forall E(\forall x(\neg(E_{1,C_x} \wedge E_{2,C_x})))$

NOTE: The first subscript number to determine which node (N) or edge (E) is considered is excluded, as the $\forall$ quantifier implies every node or edge should be considered.

3.3.3 Predicate Logic to Propositional Logic

The graph colouring problem is now represented in Boolean algebra, however SAT has the requirement of being expressed using propositional logic. Considering every case for each quantifier gives the following formulae.

1. $(N_{1,C_1} \vee \ldots \vee N_{1,C_m}) \wedge \ldots \wedge (N_{n,C_1} \vee \ldots \vee N_{n,C_m})$

2. $\Big[\Big(\neg(N_{1,C_1} \wedge N_{1,C_2}) \wedge \ldots \wedge \neg(N_{1,C_1} \wedge N_{1,C_m})\Big) \wedge \Big(\neg(N_{1,C_2} \wedge N_{1,C_3}) \wedge \ldots \wedge \neg(N_{1,C_2} \wedge N_{1,C_m})\Big) \wedge \ldots \wedge \Big(\neg(N_{1,C_{m-1}} \wedge N_{1,C_m})\Big)\Big]$

   $\wedge \ldots \wedge$

   $\Big[\Big(\neg(N_{n,C_1} \wedge N_{n,C_2}) \wedge \ldots \wedge \neg(N_{n,C_1} \wedge N_{n,C_m})\Big) \wedge \Big(\neg(N_{n,C_2} \wedge N_{n,C_3}) \wedge \ldots \wedge \neg(N_{n,C_2} \wedge N_{n,C_n})\Big) \wedge \ldots \wedge \Big(\neg(N_{n,C_{n-1}} \wedge N_{n,C_n})\Big)\Big]$

3. $\Big(\neg(E_{1,1,C_1} \wedge E_{1,2,C_1}) \wedge ... \wedge \neg(E_{1,1,C_m} \wedge E_{1,2,C_m})\Big) \wedge \Big(\neg(E_{p,1,C_1} \wedge E_{p,2,C_1}) \wedge ... \wedge \neg(E_{p,1,C_m} \wedge E_{p,2,C_m})\Big)$

NOTE: Formula 2 above may look confusing, however it can be interpreted as "for every combination of two different colours, no node can have both colours". Each set of square brackets contains all requirements for exactly one node.

3.3.4 Predicate Logic to CNF

Although the formulae are now expressed in terms of propositional logic, SAT also requires that formulae should be in CNF, where a formula is a conjunction of clauses, and clauses are disjunctions of literals.

Formula 1 above is already in CNF, however formulae 2 and 3 need manipulation to be represented in CNF. Through the use of De Morgan's Law, the clauses contained within formulae 2 and 3 can be converted to CNF by bringing each negation inside the set of parenthesis it is being applied to, therefore converting every clause into a disjunction of literals. The manipulated formulae are shown below.

1. [*Unchanged*] $(N_{1,C_1} \lor \ldots \lor N_{1,C_m}) \land \ldots \land (N_{n,C_1} \lor \ldots \lor N_{n,C_m})$

2. $\Big[\Big((\neg N_{1,C_1} \lor \neg N_{1,C_2}) \land \ldots \land (\neg N_{1,C_1} \lor \neg N_{1,C_m})\Big) \land \Big((\neg N_{1,C_2} \lor \neg N_{1,C_3}) \land \ldots \land (\neg N_{1,C_2} \lor \neg N_{1,C_n})\Big) \land$
   $\ldots \land \Big((\neg N_{1,C_{n-1}} \lor \neg N_{1,C_n})\Big)\Big]$
   $\land \ldots \land$
   $\Big[\Big((\neg N_{n,C_1} \lor \neg N_{n,C_2}) \land \ldots \land (\neg N_{n,C_1} \lor \neg N_{n,C_m})\Big) \land \Big((\neg N_{n,C_2} \lor \neg N_{n,C_3}) \land \ldots \land (\neg N_{n,C_2} \lor \neg N_{n,C_m})\Big) \land$
   $\ldots \land \Big((\neg N_{n,C_{m-1}} \lor \neg N_{n,C_m})\Big)\Big]$

3. $\Big((\neg E_{1,1,C_1} \lor \neg E_{1,2,C_1}) \land \ldots \land (\neg E_{1,1,C_m} \lor \neg E_{1,2,C_m})\Big) \land \Big((\neg E_{p,1,C_1} \lor \neg E_{p,2,C_1}) \land \ldots \land (\neg E_{p,1,C_m} \lor \neg E_{p,2,C_m})\Big)$

All clauses in formulae 1, 2 and 3 are now disjunctions of literals. The brackets and parenthesis can be removed through the application of the associativity of Boolean algebra, which means formulae 1, 2 and 3 are now in CNF.

The original three conditions for the satisfiability of a graph colouring problem must all be made true by at least one assignment for the problem to be satisfiable. This can be represented using the CNF formulae 1, 2 and 3 by conjugating them together; by definition this conjugation is also in CNF, meaning it is a SAT problem.

As the new formula created by combining formulae 1, 2 and 3 describes exactly the problem of graph colouring satisfiability, the two problems are equisatisfiable. As the new formula is a SAT problem, this is a successful reduction of graph colouring to SAT.

NOTE: The final formula is not displayed, as it would be difficult to read and reduce the concision of this report.

4 Interface Languages

As described in Section 2, the specification required an interface language to be create or source for both graphs and Boolean algebra. For these I decided to use the DIMACS formats, each of which are briefly discussed with examples in the next two subsections.

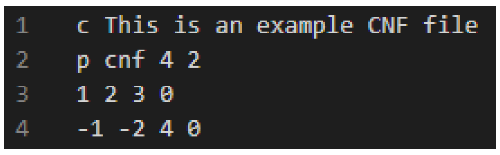
4.1 Boolean Algebra Interface Language

As all Boolean algebra problems involved in this practical are either SAT or 3SAT problems, I decided to use the DIMACS CNF format [5] as the interface language for Boolean algebra.

DIMACS CNF format is a popular notation for SAT problems. It uses integers to represent literals, with negative numbers equating to the negation of the absolute value of the number; for example -3 is the negation of 3 as $\neg l_1$ is the negation of l_1. Each clause is defined by a line ending in 0, and literals are listed with a singular white-space character between them.

The line beginning with the character `p` in DIMACS CNF files describes the problem encoded within the file. For CNF problems, the structure of the line is `p cnf <NUM_VARS> <NUM_CLAUSES>` , which implies the problem is in CNF format, has `<NUM_VARS>` variables used within the formula, and the formula is made of `<NUM_CLAUSES` clauses. Lines beginning with the character `c` are comments, and it is convention for one to be used in line 1 to describe the purpose of the file. It is also convention for DIMACS CNF files to have the file-extension `.cnf`.

An example of a file using DIMACS CNF format can be seen in Figure 11. The file describes a CNF problem with 4 variables and 2 clauses, which can be inferred from the line beginning with the character `p`. The two clauses are equivalent to those in the formula $(l_1 \lor l_2 \lor l_3) \land (\neg l_1 \lor \neg l_2 \lor l_4)$, and this is the formula described by the file.



```
c This is an example CNF file
p cnf 4 2
1 2 3 0
-1 -2 4 0
```

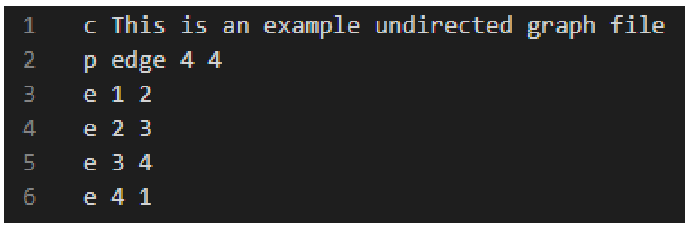
Figure 11: An example DIMACS CNF file.

4.2 Graph Interface Language

For similar reasons as the Boolean algebra interface language choice, and to keep consistency within this practical, I decided to use the DIMACS undirected graph format [6]. As with DIMACS CNF format, lines beginning with the character `c` are comments, and the line beginning with the character `p` is the problem line. The problem line format for DIMACS undirected graph files is `p edge <NUM_NODES> <NUM_EDGES>`, which implies the file describes the edges of an undirected graph with `NUM_NODES` nodes and `NUM_EDGES` edges.

Each line following the problem line should start with the character `e`, and describes one edge in the graph. These lines are in the format `e u v`, where `u` and `v` are nodes involved in an edge on the graph, which is not described elsewhere in the file (either as `e u v` or `e v u`). DIMACS undirected graph format files are typically created with the `.col` file extension.

An example of a file using DIMACS undirected graph format can be seen in Figure 12. The file describes an undirected graph problem which can be imagined as a square - 4 nodes and 4 edges, with each node being involved with exactly two edges.



```
c This is an example undirected graph file
p edge 4 4
e 1 2
e 2 3
e 3 4
e 4 1
```

Figure 12: An example DIMACS undirected graph file.

5 Reduction Implementations

5.1 Setup & Auxiliary Code

To implement the reductions required, I decided to use Python 3 as my programming language. The main reasons for this choice is that Python is fast for developing small projects as it has a light-weight syntax. Python is an interpreted language, meaning it is likely slower than the equivalent code written in C or C++, however performance at language level was not required by the specification.

To aid in development on my reduction programs, I created functions to unmarshall DIMACS CNF and graph format files into Python objects, and also created functions to marshall these objects back to their respective file formats, printing them to `stdout`. These functions can be found in the `dimacs_cnf.py` and `dimacs_undirected_graph.py` files inside the `src/marshallers_unmarshallers` directory. These files also contain definitions for the `CNF` and `graph` classes, which are the classes of the objects used for the memory representations of problems.

5.1.1 Polynomial Time?

The DIMACS problem formats used in my implementation list graph edges or CNF clauses line by line, suggesting a linear time complexity is required to parse all information from these files ($\mathcal{O}(n)$). This is true for graph files, as the lines have a fixed number of characters to represent the maximum of two nodes in an edge, however CNF clauses can have an infinite number of literals, making the time complexity for parsing these files $\mathcal{O}(n \cdot m)$ where n is the number of lines and m is the average number of literals in each line.

The functions have a constant ($\mathcal{O}(1)$) start-up and clean-up time, as there is only one problem line, meaning by the nature of time complexity, the marshalling and unmarshalling functions for CNF and graph files have a polynomial and linear execution time respectively.

As the marshalling and unmarshalling functions are executed a constant number of times in each reduction, they are not discussed explicitly in determining whether each reduction has polynomial time complexity, as their linear/polynomial time complexity combined with a fix execution count will never cause a reduction to be greater than polynomial time complexity.

5.2 SAT to 3SAT

To implement the SAT to 3SAT reduction, the SAT problem is first read into the program using the auxiliary functions described in Section 5.1. The clauses of the SAT problem are then iterated through, following the procedure described in Section 3.1 to convert them into 3SAT compliant clauses which are then added to an output problem object.

As mentioned in Section 3.1 of this report, clauses with three or fewer literals in the provided SAT problem are already 3SAT compliant, and are therefore copied directly from the input problem to the output problem.

Clauses with four or more literals need substituting with a set of clauses which contain fresh literals. The fresh literals are generated by first finding the highest value from the input problem, and then only using values greater than this for fresh literals and their negations - the use of integers as variables in DIMACS CNF format facilitates this process greatly. The substitute clauses are created to match those described in Section 3.1 of this report, and are added to the output problem upon generation.

Fresh variables are added to the CNF object's variable set when first used, and the clause and variable counters are updated throughout the conversion and substitution process.

To finish the reduction, the output problem is written to `stdout` in DIMACS CNF format.

5.2.1 Polynomial Time?

The sections of this reduction program which do not have constant time complexity ($\mathcal{O}(1)$) are finding the highest value used in the input problem, converting each clause to 3SAT, and adding fresh variables to the output problem object.

Finding the highest value used in the input problem has $\mathcal{O}(a \cdot b)$ execution time, where a is the number of clauses in the input problem and b is the average length of each clause. The reason for this is that every literal in every clause must be considered.

Adding fresh variables to the output problem object has linear time complexity ($\mathcal{O}(c)$, where c is the number of fresh literals used), as each literal used needs to be added exactly once.

Converting all clauses to 3SAT clauses is of quadratic time complexity. The reason for this is that converting a clause with three or fewer literals is done in constant time, as the clause does not need manipulating, and converting a clause with four or more literals is done in linear time, as each literal in the clause has a constant time operation applied to it. Each clause needs to be converted once, leading to the quadratic time complexity.

The greatest time complexity of all operations in the SAT to 3SAT reduction is quadratic, and therefore this reduction is of polynomial time complexity.

5.2.2 Correctness

Although my implementation of the reduction from SAT to 3SAT cannot be proven to be correct for every possible case of SAT (as there are infinitely many), the implementation follows the reduction proof in Section 3.1, and can therefore be assumed to be correct.

The reduction program is tested for generic short and long cases, which is discussed in the Testing section of this report.

5.3 3SAT to Graph Colouring

To begin this reduction, the 3SAT problem is imported into a Python object using the CNF unmarshaller described in Section 5.1 of this report. Before converting any clauses from the input problem, a graph object is made to represent the output problem, and the Ground and False palette nodes used in all clause-gadgets are created and connected (see Section 3.2).

The created Ground and False nodes must be fresh nodes, which are not equivalent to variables found in the 3SAT formula or their negations. Fresh nodes are generated in a similar fashion to how fresh literals were generated in the SAT to 3SAT reduction - find the greatest value used for a literal in the input problem, and use the values greater than this for the fresh nodes.

After creating the palette nodes, each clause is converted into a clause-gadget to match the number of literals involved in the clause, as described in Section 3.2 of this report. The clause-gadget is then added to the output graph object, connecting to the palette nodes as required.

Literals are converted into nodes by using the same integer value. This is beneficial, as it is easy to generate fresh literals for palette nodes or those involved in OR-gadgets, and it also means that duplicate literals are handled automatically as the literals in the 3SAT formula have equivalent names in the graph representation. Adding repeated nodes to the output graph object has no effect as the nodes are stored as a set, and adding edges involving the repeated node will automatically connect to the existing node in the graph, as the same node number is used.

One-literal clauses are added to the output graph by adding the literal as a node along with the edges to connect it to the Ground and False palette nodes.

Two-literal clauses are added to the output graph by first adding the literals to the graph and connecting them to the Ground palette node. Next, auxiliary nodes are added to create the body of the OR-gadget - these nodes are referenced as "adjacent-to-literal nodes" or "ATL nodes" in the Python code, and are labelled in Figure 13 to provided understanding of which nodes these names reference. The ATL nodes are connected as required, and then the output node is added and connected to all required nodes to complete the OR-gadget (i.e. two-literal clause-gadget) - Figure 13 shows which node is described by the name "output node".

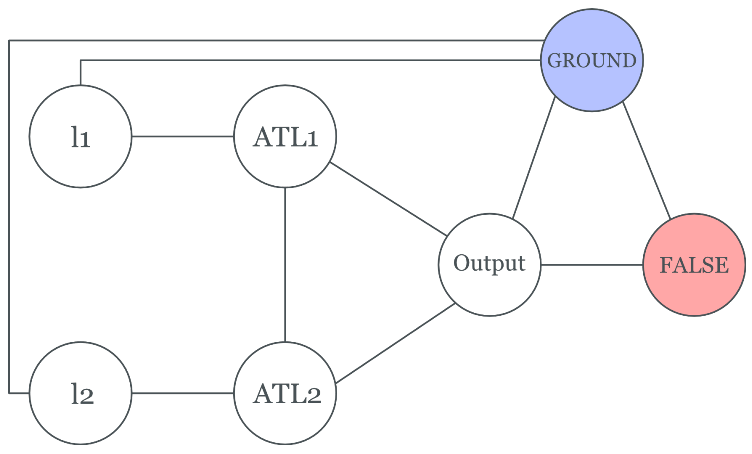


Figure 13: A two-literal clause-gadget labelled to match Python naming.

Three-literal clauses are added to the output graph in a similar fashion to two-literal clauses, except the the four extra nodes are included and connected as required. The name "half-output" or "HO" is used in the Python code, and this represents the node which acts as the output for the nested OR-gadget. The node adjacent to the half-output node is labelled "adjacent-to-half-output" or "ATHO". Figure 14 show the HO and ATHO nodes labelled in a visual scenario.

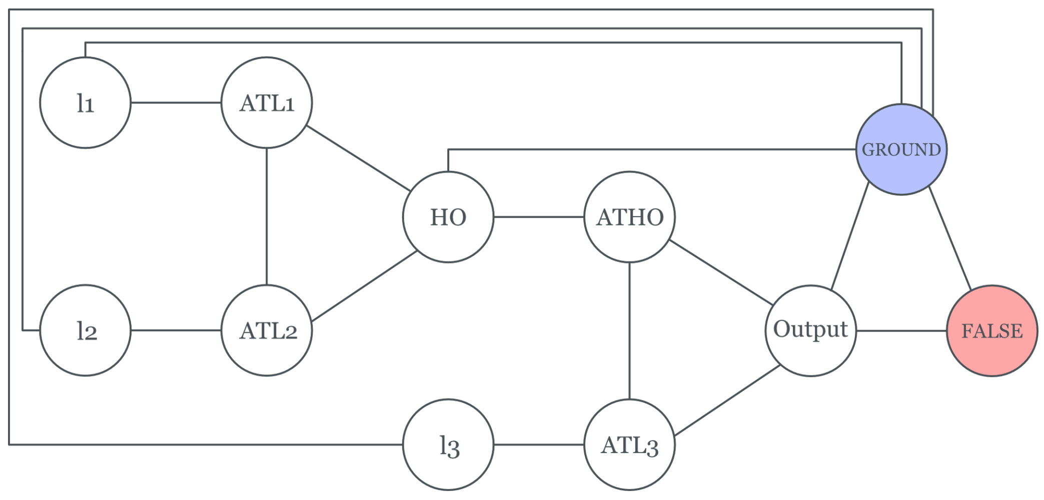


Figure 14: A three-literal clause-gadget labelled to match Python naming.

To incorporate the negations discussed in Section 3.2.7 into the graph, all nodes are iterated over after all clauses are added, checking for negations. Any nodes which have a negation involved in the graph are connected to each other as well as the Ground palette node.

To comply with DIMACS undirected graph format, the edges set in the graph object is checked for back-to-front duplicates (i.e. an entry describing the edge between u and v and an entry describing the edge between v and u). If any back-to-front duplicates are found, one of the pair will be added to a "to remove" set, which will be iterated through after all pairs are found - this avoids the data structure being mutated while being iterated through. Exact duplicates will not be included, as the edges are stored as a tuple set.

5.3.1 Polynomial Time?

The operations which take longer than constant time in the 3SAT to graph colour reduction are finding the highest value variable in the 3SAT problem, converting each clause into a clause-gadget in the graph, connecting negations in the graph, and removing back-to-front duplicates.

Finding the highest value is identical in time complexity to that discussed in Section 5.2.1 - polynomial.

Converting each clause into a clause-gadget has a fixed set of operations for each clause, making this entire procedure linear in time complexity ($\mathcal{O}(n)$), as every clause needs to be converted.

Connecting negations in the graph has $\mathcal{O}(a^2)$ time complexity, where a is the number of nodes in the output graph. The reason for this is that every node needs to be checked against every other node, which is done using two for loops, making this a procedure of quadratic time complexity.

Removing back-to-front duplicate edges from the graph is not the most efficient operation, however it has only quadratic time complexity ($\mathcal{O}(b^2)$, where b is the number of edges in the output graph's edge set). The reason for this is that every edge needs to be checked against every other edge. Removing the edges is a linear operation, as the `remove()` operation for a set has constant time complexity [7] and the remove set needs to be iterated through.

As no operation has greater time complexity than quadratic, the 3SAT to graph colour reduction also has quadratic time complexity ($\mathcal{O}(n^2)$), meaning it is of polynomial time complexity.

5.3.2 Correctness

Although my implementation of the reduction from 3SAT to graph colouring cannot be proven to be correct for every possible case of 3SAT (as there are infinitely many), the implementation follows the reduction proof in Section 3.2, and can therefore be assumed to be correct.

The reduction program is tested for generic short and long cases, which is discussed in the Testing section of this report.

5.4 Graph Colouring to SAT

The reduction of a graph colouring problem to a SAT problem is the only reduction which required a variable which was not defined inside the problem file. This variable is number of colours to be used in the graph colouring problem. Therefore I implemented this reduction to take an integer command line argument for how many colours are to be used.

To begin the reduction of graph colouring to SAT, the colour command line variable is checked and parsed into a positive integer, printing user-friendly statements to the command line in the case of misuse.

Following this, the graph for the colouring problem is unmarshalled using one of the functions described in Section 5.1 of this report. Before reducing the problem, a CNF output object is created to store the output in the Python interpreter's memory.

To convert the problem from the input graph and colour count to a CNF problem, the procedure described in Section 3.3 of this report is used. A brief reminder of the conditions discussed in this section is shown below.

1. Each graph node has at least one colour.
2. Each graph node has no more than one colour.
3. No two adjacent graph nodes have the same colour.

To transfer the concept of nodes having a truth value associated with each colour into DIMACS integer variable format, I created a concise function to generate a unique value for each colour-node combination. This function is named `variable_from_node_and_colour` and uses the colours represented as numbers in combination with the node's value to produce the unique value required.

To transfer condition 1 into code, all nodes are iterated over. Within each iteration all colours are iterated over and the colour-node truth variable is added to a clause for each node. This recreates the CNF formula described in Section 3.3 of this report.

To transfer condition 2 into code, the same node iteration loop as condition 1 is used, however within this loop, each two-colour combination is generated by applying two for loops. The node-colour1-colour2 combination is converted into a clause to match the format described in Section 3.3 of this report.

To transfer condition 3 into code, the input problem's edges are iterated over, with each colour being iterated over for each edge by using a nested for loop. Each edge-colour combination is converted into into a CNF clause by calculating the colour-node truth value variable for both nodes in the edge and negating them before adding them to a Python list. As the edges are stored as tuples, it is a simple operation to access each node in a given edge.

5.4.1 Polynomial Time?

The operations which are greater than constant time complexity for the graph colouring to SAT reduction are transferring conditions 1, 2 and 3 into code.

Each of these operations use the `variable_from_node_and_colour` function, which makes use of multiplication and addition in such a way that the function has quadratic time complexity. This is because multiplication has quadratic time complexity and addition has linear time complexity [8], and each are used only once in the function. The quadratic time complexity of this function is referenced as $\mathcal{O}(v^2)$ for the rest of this section.

Transferring condition 1 has polynomial time complexity equivalent to $\mathcal{O}(n \cdot c \cdot v^2)$, where n is the number of nodes in the graph and c is the number of colours in the problem. The reason for this is that all nodes are iterated over in a linear fashion, with each colour being iterated over for each node; each colour-node combination calls the `variable_from_node_and_colour` function, leading to the polynomial time complexity.

Transferring condition 2 has polynomial time complexity equivalent to $\mathcal{O}(n \cdot c^2 \cdot v^2)$, where n is the number of nodes in the graph and c is the number of colours in the problem. The reason for this is that all nodes are iterated over in a linear fashion, and within each node, every two-colour combination is found using two nested for loops inside the node loop. Each two-colour-node combination calls the `variable_from_node_and_colour` function, leading to the polynomial time complexity.

Transferring condition 3 has polynomial time complexity equivalent to $\mathcal{O}(e \cdot c \cdot v^2)$, where e is the number of edges in the graph and c is the number of colours in the problem. the reason for this is that every edge needs to be iterated over, with every colour being iterated over on each loop. Each edge-colour combination calls the `variable_from_node_and_colour` function twice, leading to the polynomial time complexity.

Since all operations in the graph colour to SAT reduction have at most polynomial time complexity, the reduction as a whole has polynomial time complexity.

5.4.2 Correctness

Although my implementation of the reduction from graph colouring to SAT cannot be proven to be correct for every possible case of graph-colouring problem (as there are infinitely many), the implementation follows the reduction proof in Section 3.3, and can therefore be assumed to be correct.

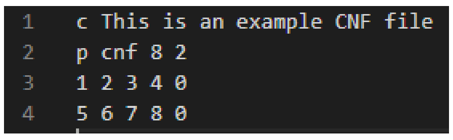
The reduction program is tested for generic short and long cases, which is discussed in the Testing section of this report.

6 Testing

6.1 Simple Input & Output Comparisons

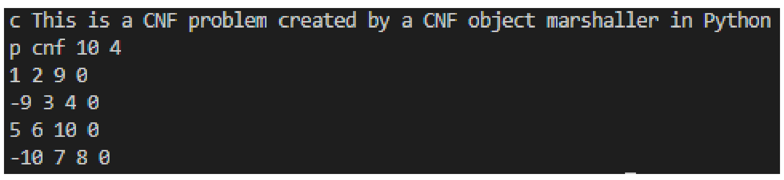
To begin testing my reduction programs, I provided simple input files and compared the output files to what was expected.

The simple input file for the SAT to 3SAT reduction is named `SAT-Example1.cnf` and can be found in the `src/example_problems` directory. The contents of this file is displayed in Figure 15, and the output of the reduction is displayed in Figure 16. It can be seen in Figure 15 that the input problem consists of two clauses, each 4 literals in length - the correct output of the program should therefore contain 4 clauses, each with three literals one of which is fresh literal in each clause. It can be seen in Figure 16 that this is the case.



```
c This is an example CNF file
p cnf 8 2
1 2 3 4 0
5 6 7 8 0
```

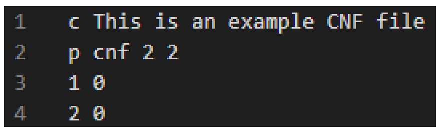
Figure 15: The contents of the `SAT-Example-1.cnf` file.



```
c This is a CNF problem created by a CNF object marshaller in Python
p cnf 10 4
1 2 9 0
-9 3 4 0
5 6 10 0
-10 7 8 0
```

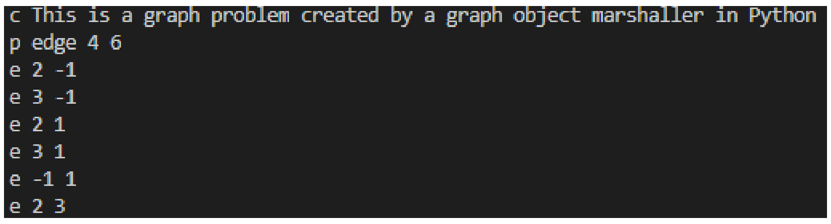
Figure 16: The output of the reduction of the `SAT-Example-1.cnf` file.

The simple input file for the 3SAT to graph colour reduction is named `3SAT-Example1.cnf` and can be found in the `src/example_problems` directory. The contents of this file is displayed in Figure 17, and the output of the reduction for a three-colour palette is displayed in Figure 18. It can be seen in Figure 19 that the input problem is simply two clauses each with a single literal, meaning the output should have 4 nodes connected with all possible edges, except the edge connecting the literals in the original 3SAT problem, as their true-false values are not linked in any way. By setting the literals from the input problem to 1 and 2, and the Ground and False palette nodes to 3 and 4 respectively, the output graph is as expected for a 3-colour colouring problem.



```
c This is an example CNF file
p cnf 2 2
1 0
2 0
```

Figure 17: The contents of the `3SAT-Example1.cnf` file.



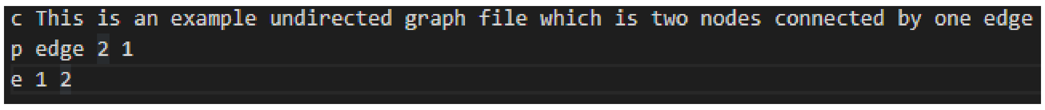
```
c This is a graph problem created by a graph object marshaller in Python
p edge 4 6
e 2 -1
e 3 -1
e 2 1
e 3 1
e -1 1
e 2 3
```

Figure 18: The output of the reduction of the `3SAT-Example1.cnf` file.

The simple input file for the graph colour to SAT reduction is named `GC-Example1.cnf` and can be found in the `src/example_problems` directory. The contents of this file is displayed in Figure 19, and the output of the reduction for a two-colour palette is displayed in Figure 20. It can be seen in Figure 19 that the input graph is simply two nodes connected by one edge, meaning the output should contain the clauses to enforce the following conditions.

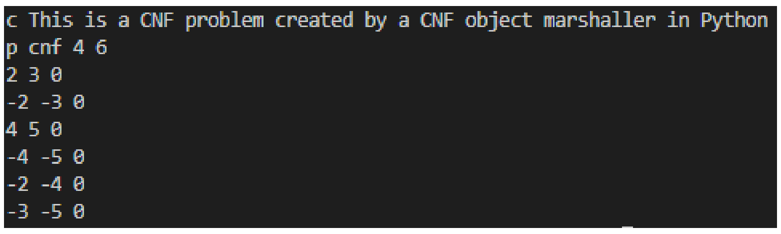
1. Node 1 should have at least colour 1 or colour 2. This is enforced by the first clause in the output problem.
2. Node 2 should have at least colour 1 or colour 2. This is enforced by the third clause in the output problem.
3. Node 1 should not have colour 1 and colour 2 at the same time. This is enforced by the second clause in the output problem.
4. Node 2 should not have colour 1 and colour 2 at the same time. This is enforced by the fourth clause in the output problem.
5. Node 1 and node 2 should not both have colour 1. This is enforced by the fifth clause in the output problem.
6. Node 1 and node 2 should not both have colour 2. This is enforced by the sixth clause in the output problem.

As described in the list above, the output displayed in Figure 20 is as expected for the input problem displayed in Figure 19.



```
c This is an example undirected graph file which is two nodes connected by one edge
p edge 2 1
e 1 2
```

Figure 19: The contents of the `GC-Example1.col` file.

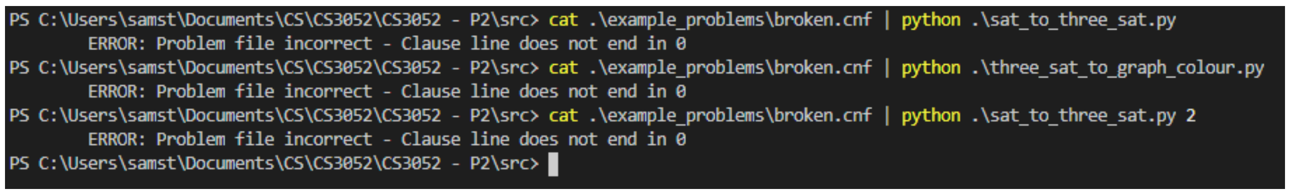


```
c This is a CNF problem created by a CNF object marshaller in Python
p cnf 4 6
2 3 0
-2 -3 0
4 5 0
-4 -5 0
-2 -4 0
-3 -5 0
```

Figure 20: The output of the reduction of the `GC-Example1.col` file with 2 colours.

6.2 Incorrect Input

Before testing my reduction programs on more complicated examples, I decided to test incorrectly formatted examples. The files `broken.cnf` and `broken.col` can be found in the `src/example_problems` directory, and contain an incorrectly formatted encoding of a DIMACS CNF problem and DIMACS undirected graph problem respectively. The output of the three reduction programs when provided with one of these files are shown in Figure 21, and it can be seen that all reductions ignore the input and provide a helpful message to the user.



```
PS C:\Users\samst\Documents\CS\CS3052\CS3052 - P2\src> cat .\example_problems\broken.cnf | python .\sat_to_three_sat.py
        ERROR: Problem file incorrect - Clause line does not end in 0
PS C:\Users\samst\Documents\CS\CS3052\CS3052 - P2\src> cat .\example_problems\broken.cnf | python .\three_sat_to_graph_colour.py
        ERROR: Problem file incorrect - Clause line does not end in 0
PS C:\Users\samst\Documents\CS\CS3052\CS3052 - P2\src> cat .\example_problems\broken.cnf | python .\sat_to_three_sat.py 2
        ERROR: Problem file incorrect - Clause line does not end in 0
PS C:\Users\samst\Documents\CS\CS3052\CS3052 - P2\src>
```

Figure 21: The output of all reduction programs when supplied with incorrectly formatted problems.

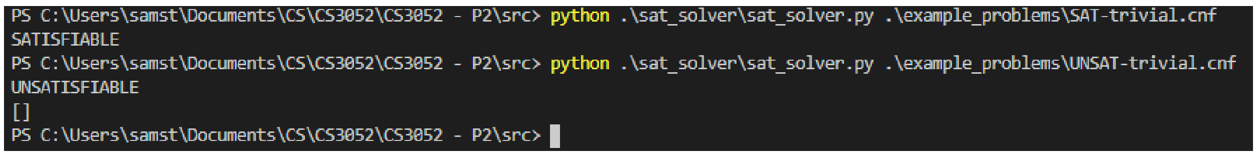
6.3 PySAT & SATLIB

To test my reductions further, I decided to increase the size and complexity of problem being reduced. In order to do so, I required a large problem file, and a SAT solver.

For the large problem file, I used the University of British Columbia's SATLIB collection [9], from which I took one satisfiable CNF problem (201) and one unsatisfiable CNF problem (202) from the JNH collection - these are store in the files `ubc_sat_201.cnf` and `ubc_unsat_202.cnf` inside the `src/example_problems` directory.

For the SAT solver, I decided to use PySAT [10]. To use this SAT solver, I first had to create a simple Python script, which can be found in the `src/sat_solver` directory. This script takes and checks a command line argument for the DIMACS CNF file to check the satisfiability for, and then prints "UNSATISFIABLE" plus a reason if the CNF problem is unsatisfiable or "SATISFIABLE" if the CNF problem is satisfiable.

The script was very simple to develop and involved code copied directly from PySAT's documentation, so required very little testing, however to ensure the script was outputting properly, I created two simple test files `SAT-trivial.cnf` and `UNSAT-trivial.cnf` which contain a trivial satisfiable and unsatisfiable CNF problem respectively. The output of the PySAT script running on these trivial problems can be seen in Figure 22, and it can be seen that the script produces the expected outputs.



```
PS C:\Users\samst\Documents\CS\CS3052\CS3052 - P2\src> python .\sat_solver\sat_solver.py .\example_problems\SAT-trivial.cnf
SATISFIABLE
PS C:\Users\samst\Documents\CS\CS3052\CS3052 - P2\src> python .\sat_solver\sat_solver.py .\example_problems\UNSAT-trivial.cnf
UNSATISFIABLE
[]
PS C:\Users\samst\Documents\CS\CS3052\CS3052 - P2\src>
```

Figure 22: PySAT script deciding trivial SAT and UNSAT problems correctly.

Unfortunately I was unable to find a graph colouring problem solver or a large undirected graph in DIMACS format to test the graph colour to SAT and 3SAT to graph colour reductions. However, by piping the output from each reduction into other reductions, I was able to pass both the large satisfiable and unsatisfiable CNF problems into the reduction programs and end up with another CNF problem which should be equisatisfiable.

To use all reduction programs, I used command line piping to pipe the output of the SAT to 3SAT reduction into the 3SAT to graph colouring reduction, which is then piped into the graph colouring to SAT reduction. This output file is then checked using the PySAT script to determine whether it is equisatisfiable to the original CNF formula.

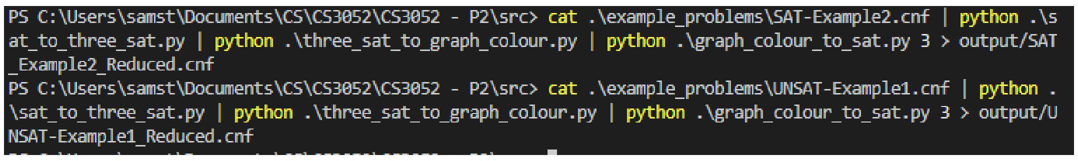
Unfortunately when completing this step, there were around 17,000 nodes being created by the first graph, meaning the reduction completed by the graph to SAT reducing program would have taken around 12 hours to complete.

In an attempt to reduce the computation time, I cut 700 of the 800 clauses from the input CNF problem file, however I encountered a new issue - the 3SAT to graph colouring problem produces A LOT of edges, leading to over 23,000 clauses after this problem is reduced to a SAT problem by the next program, which caused an overflow error for PySAT. The output of this reduction can be seen in the file named `ubc_sat_reduced.cnf` in the `src/output` directory.

Unfortunately with limited time for this practical, I was unable to consider alternative reductions for 3SAT to graph colouring, however I assume that limiting my reduction to only 3 colours lead to many more auxiliary nodes than necessary.

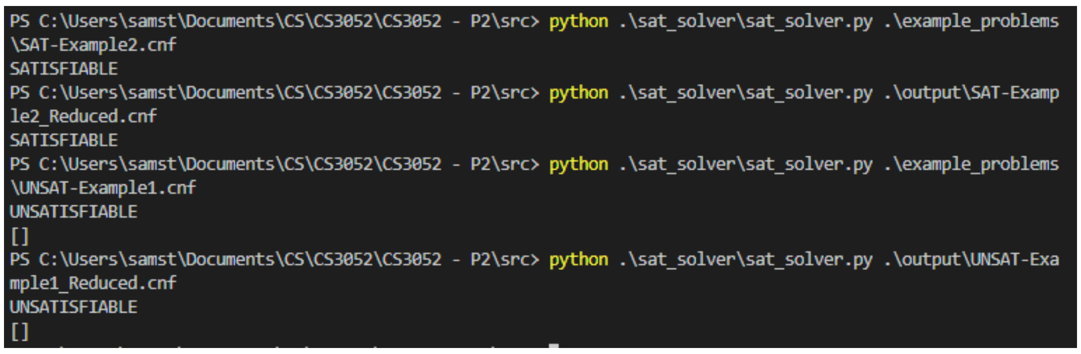
In an attempt to still test my program, I created a small CNF problem which is satisfiable and one which is unsatisfiable to test the piping of reductions and to check the equisatisfiability of the circularly reduced CNF problems. These files are named `SAT-Example2.cnf` and `UNSAT-Example1.cnf` respectively, and can be found in the `src/example_problems` directory.

Figure 23 shows the execution of the piping of the small satisfiable and unsatisfiable CNF problems through the different reductions, and Figure 24 shows the output of the PySAT script when executed with the original problems and the reduced problems. It can be seen in Figure 24 that the original and reduced problems have the same output, and we can therefore be confident that the reductions are all correct.



```
PS C:\Users\samst\Documents\CS\CS3052\CS3052 - P2\src> cat .\example_problems\SAT-Example2.cnf | python .\s
at_to_three_sat.py | python .\three_sat_to_graph_colour.py | python .\graph_colour_to_sat.py 3 > output/SAT
_Example2_Reduced.cnf
PS C:\Users\samst\Documents\CS\CS3052\CS3052 - P2\src> cat .\example_problems\UNSAT-Example1.cnf | python .
\sat_to_three_sat.py | python .\three_sat_to_graph_colour.py | python .\graph_colour_to_sat.py 3 > output/U
NSAT-Example1_Reduced.cnf
```

Figure 23: Piping reduction outputs between other reductions.



```
PS C:\Users\samst\Documents\CS\CS3052\CS3052 - P2\src> python .\sat_solver\sat_solver.py .\example_problems
\SAT-Example2.cnf
SATISFIABLE
PS C:\Users\samst\Documents\CS\CS3052\CS3052 - P2\src> python .\sat_solver\sat_solver.py .\output\SAT-Examp
le2_Reduced.cnf
SATISFIABLE
PS C:\Users\samst\Documents\CS\CS3052\CS3052 - P2\src> python .\sat_solver\sat_solver.py .\example_problems
\UNSAT-Example1.cnf
UNSATISFIABLE
[]
PS C:\Users\samst\Documents\CS\CS3052\CS3052 - P2\src> python .\sat_solver\sat_solver.py .\output\UNSAT-Exa
mple1_Reduced.cnf
UNSATISFIABLE
[]
```

Figure 24: PySAT script showing that circular reductions are equisatisfiable.

6.4 Testing Conclusion

As mentioned in Section 5, it is impossible to show that each reduction implementation is correct for every possible input, as there are infinitely many for each program. However, as the programs are based directly on the reduction proofs from Section 3, and the programs handle generic cases without any errors, and transform generic positive/negative cases of one problem into positive/negative cases of another problem correctly, it can be said with confidence that the implementations are correct.

7 Limitations

Besides the required time for reductions and the number of edges/clauses produced in each reduction, the main limitations of my programs are imposed by using DIMACS file formats and the specification of the computer running the programs.

Using DIMACS file formats imposes the limitations that only problems expressed using DIMACS file formats can be reduced by my programs, meaning other problems must be convert into DIMACS compliant files before reducing them using the program.

The choice of Python 3 allows for numbers used to represent literals and nodes in DIMACS files to be very large, as Python 3 has built-in integer overflow, however the system's memory limit still imposes a cap on the size of the numbers being used to represent literals/nodes [11].

8 Evaluation & Conclusion

Overall I believe I have been successful with this practical - I have implemented the basic specification to a high standard, and have completed thorough testing to accompany my work, which resulted in interesting discoveries.

Reflecting on the learning objectives for this practical listed in the specification, I have learned a lot about polynomial time reductions, and have gained knowledge on programming with mathematical objects, as well as analysing theoretical and implemented algorithms.

Given more time, I would enjoy fixing my implementation to produce fewer nodes/edges, and also attempting some more complicated reductions.

References

[1] Indian institute of technology kharagpur - reduction from sat to 3sat.
https://cse.iitkgp.ac.in/ palash/2018AlgoDesignAnalysis/SAT-3SAT.pdf [Accessed May 2020].

[2] University of liverpool computational np-completeness lecture slides.
https://cgi.csc.liv.ac.uk/ igor/COMP309/3CP.pdf [Accessed May 2020].

[3] Youtube — udacity - intro to algorithms playlist.
https://cgi.csc.liv.ac.uk/ igor/COMP309/3CP.pdf [Accessed May 2020].

[4] University of texas - computer science reduction lecture notes.
https://www.cs.utexas.edu/users/vl/teaching/lbai/coloring.pdf [Accessed May 2020].

[5] Florida state university, department of computer science - dimacs cnf format.
https://people.sc.fsu.edu/ jburkardt/data/cnf/cnf.html [Accessed May 2020].

[6] State key laboratory of computer science - dimacs graph format.
http://lcs.ios.ac.cn/ caisw/Resource/about_DIMACS_graph_format.txt [Accessed May 2020].

[7] Python wiki - time complexity.
https://wiki.python.org/moin/TimeComplexity [Accessed May 2020].

[8] University of toronto - complexity of arithmetic.
https://www.cs.toronto.edu/ guerzhoy/180/lectures/W11/lec1/ComplArithm.html
[Accessed May 2020].

[9] University of british columbia - satlib benchmarks.
https://www.cs.ubc.ca/ hoos/SATLIB/benchm.html [Accessed May 2020].

[10] Pysat: Sat technology in python.
https://www.cs.ubc.ca/ hoos/SATLIB/benchm.html [Accessed May 2020].

[11] Geeks for geeks - maximum integer value in python.
https://www.geeksforgeeks.org/what-is-maximum-possible-value-of-an-integer-in-python
[Accessed May 2020].