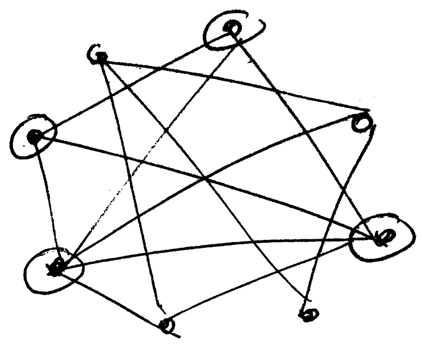
Tom Waszkowycz

# Solving NP-Complete Problems via Boolean Satisfiability



Computer Science Tripos Part II

King's College
University of Cambridge

May 12, 2015

# Proforma

| | |
|---|---|
| **Name:** | Tom Waszkowycz |
| **College:** | King's College |
| **Project Title:** | Solving NP-Complete Problems via Boolean Satisfiability |
| **Examination:** | Computer Science Tripos Part II, June 2015 |
| **Word Count:** | 11,925[1] |
| **Project Originator:** | Prof Anuj Dawar |
| **Supervisors:** | Pengming Wang<br>Prof Anuj Dawar |

## Original Aims of the Project

To implement a system which can solve arbitrary NP-complete problems. This will be done by translating these problems into instances of the Boolean Satisfiability problem, and then running a 3rd party 'Sat-solver' on these to produce a solution.

## Work Completed

All original goals of my project have been met successfully. A range of common NP-complete problems, given in Appendix C, can be solved with my system at a performance competitive with or greater than native algorithms. An extension to the core project has also been made which allows more efficient description and solving of more complex problem constraints.

[1] This word count was computed by `detex diss.tex | tr -cd '0-9A-Za-z \n' | wc -w`

## Special Difficulties

None.

# Declaration

I, Tom Waszkowycz of King's College, being a candidate for Part II of the Computer Science Tripos, hereby declare that this dissertation and the work described in it are my own work, unaided except as may be specified below, and that the dissertation does not contain material that has already been used to any substantial extent for a comparable purpose.

Signed

Date

# Contents

# List of Figures

# List of Tables

## Acknowledgements

I would like to thank Penging Wang and Prof Anuj Dawar for supervising this project. I also owe many thanks to Dr Timothy Griffin for his guidance and feedback.

# Chapter 1

# Introduction

## 1.1 Motivation

Many problems across a wide range of fields in computer science are NP-complete, including planning, scheduling, circuit design, data recovery, bioinformatics and compiler optimisation to name but a few. We want to be able to find optimal solutions to these problems efficiently, but no polynomial time algorithm has yet been discovered for any NP-complete problem, leading many to believe they are too intractable to be able to be solved in a reasonable amount of time.

Recently however, there has been much research into efficient methods of solving one particular NP-complete problem: Boolean Satisfiability (Sat), defined in Section 2.1.1. Over the past decade dramatic advances have been made into this field, outlined in Section 2.3. As a result extremely powerful Sat-solvers now exist, which are able to solve very large problem instances in a matter of seconds.

These developments are promising, but they do not directly help us to tackle the many other NP-complete problems which we would like to solve with similar speed. It would be unrealistic to hope for the necessary depth of research to be put into each of these many individual problems to result in similar performance improvements in each.

In this dissertation I hope to harness these developments in Sat to solve other problems more efficiently by making use of Fagin's theorem [15], described in Section 2.1.4. This theorem proves that the class of problems NP is exactly the set of all formulae expressible in existential second-order logic ($\exists$SO), described in Section 2.1.3. As $\mathsf{NP}$-complete $\subseteq$ $\mathsf{NP}$, this means we can express any $\mathsf{NP}$-complete problem $\mathcal{P}$ that we wish to solve in this logic.

From the definition of $\mathsf{NP}$-complete (Section 2.1.2) we know we can reduce $\mathcal{P}$ into an equivalent instance of any other $\mathsf{NP}$-complete problem in polynomial time. If we reduce $\mathcal{P}$ into an instance of BOOLEAN SATISFIABILITY $\mathcal{Q}$ we are then able to use a modern SAT-solver to solve $\mathcal{Q}$ with the efficiency that the recent developments into SAT-solving have given us. As $\mathcal{Q}$ is equivalent to $\mathcal{P}$, we can then translate the solution found by the SAT-solver for $\mathcal{Q}$ back into the original domain to quickly find an answer to $\mathcal{P}$.

This is the approach taken in this dissertation with my system, *Twist*[1].

## 1.2 Overview of *Twist*

The conversion from an $\mathsf{NP}$-complete problem described in $\exists$SO (see Section 2.1.3) through to a SAT problem and back again involves a number of stages. These are illustrated and described in Figure 1.1.

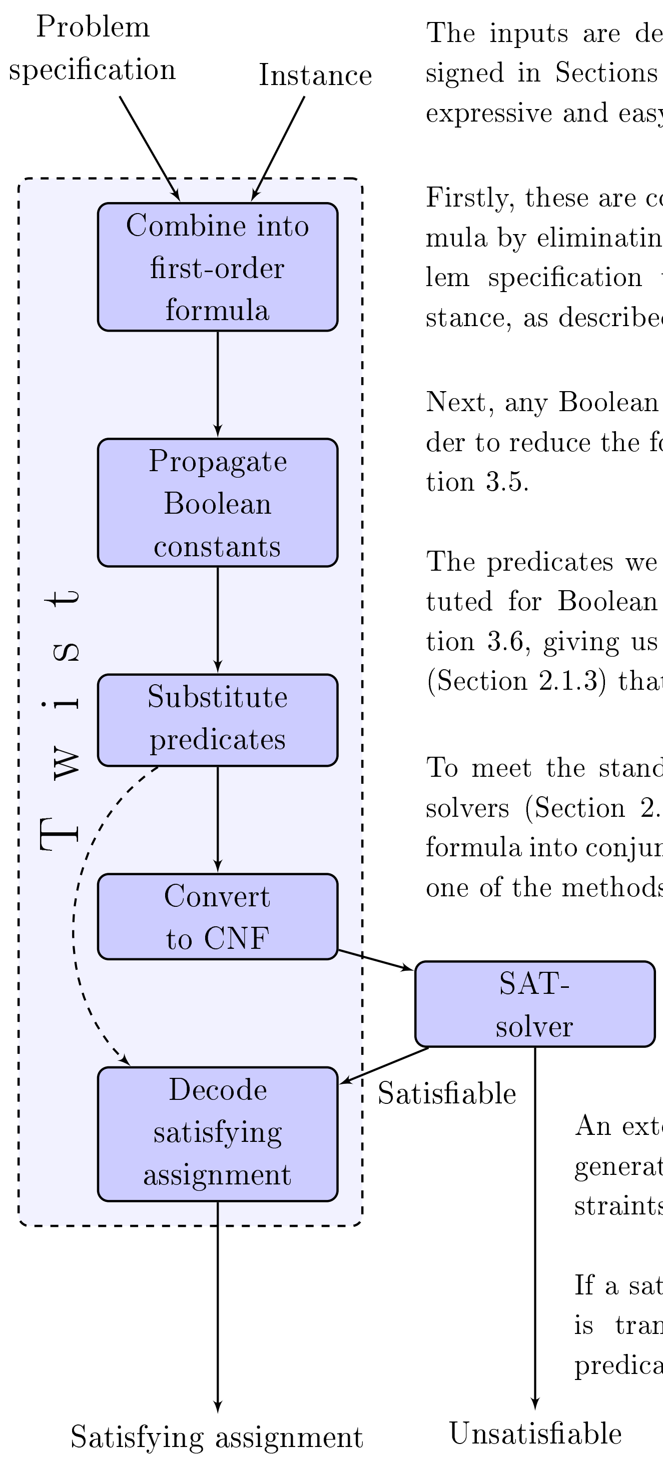
The input consists of two parts:

- The *problem specification*: a generalised description of the $\mathsf{NP}$-complete problem we wish to solve.
- Some *instance*: a finite set of relations on which we will solve our problem.

These two are input separately to allow *Twist* to easily solve multiple different instances of the same problem.

As all $\mathsf{NP}$-complete problems are decision problems (see Section 2.1.1), the output of *Twist* will be 'yes' or 'no', corresponding to whether or not the problem is satisfiable. However, in the case that a problem *is* satisfiable it is much more useful to present the user with the discovered assignment of the input predicates to the Boolean values $\{\texttt{true}, \texttt{false}\}$ which satisfies the problem's constraints. This is called the *satisfying assignment*.

[1] The name '*Twist*' comes from both the nature of translating a problem into BOOLEAN SATISFIABILITY and back again, and also the use of Fagin's theorem, the name Fagin being more commonly known as that of the antagonist from the Charles Dickens novel 'Oliver Twist'.



The inputs are described in versions of ∃SO designed in Sections 3.1 and 3.2 to be both highly expressive and easy to read and write.

Firstly, these are combined into one first-order formula by eliminating the quantifiers from the problem specification using the values from the instance, as described in Section 3.4.

Next, any Boolean constants are propagated in order to reduce the formula size, as described in Section 3.5.

The predicates we are solving for are then substituted for Boolean variables as described in Section 3.6, giving us a formula of propositonal logic (Section 2.1.3) that can be solved by a SAT-solver.

To meet the standards required by modern SAT-solvers (Section 2.3.2) we must then convert the formula into conjunctive normal form (CNF), using one of the methods described in Section 3.7.

This is then solved using a 3rd party SAT solver as described in Section 3.8.

An extension to instead allow *Twist* to generate and solve pseudo-Boolean constraints is described in Section 3.10.

If a satisfying assignment is found this is translated back into the original predicates, as described in Section 3.9.

Figure 1.1: Overview schematic of *Twist*.

## 1.3 3-Col demonstration

Here we give an example of how *Twist* is used to solve an instance of Graph 3-Colourability (3-Col), described in Section 2.1.2. The problem specification is our generalised description of 3-Col, and the instance is the graph that we wish to 3-colour.

### 1.3.1 Input

#### Problem specification

```
Given:
  V E   (* sets of vertices V and edges E given in the instance *)
Find:
  R G B (* 3 colours to colour our graph by the following rules *)
Satisfying:
        (* each vertex is exactly one colour *)
  (forall x in V (
       ( R(x) & ~G(x) & ~B(x))
     | (~R(x) &  G(x) & ~B(x))
     | (~R(x) & ~G(x) &  B(x)) ))
  &
        (* all connected vertices are different colours *)
  (forall x y in E (
       (R(x) & ~R(y))
     | (G(x) & ~G(y))
     | (B(x) & ~B(y)) ))
```

#### Instance

The instance graph $G$ that we wish to 3-colour is given diagrammatically in Figure 1.2, and described here in the plain text form that is input into *Twist*:

```
V 1      E 1 3
V 2      E 1 4
V 3      E 2 4
V 4      E 2 5
V 5      E 3 5
```

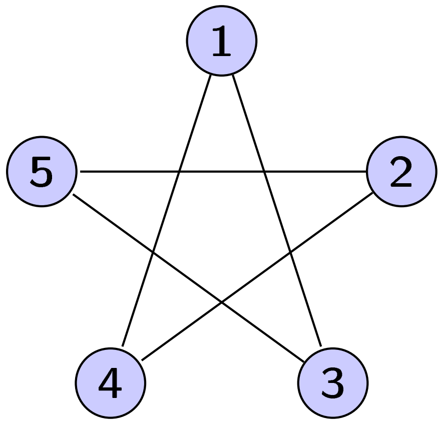


Figure 1.2: Example graph $G$ that we wish to 3-colour.

### 1.3.2 Output

In this example the solution is 'yes' the graph $G$ is 3-colourable, and the following satisfying assignment is returned by *Twist*:

```
SAT
R(1) B(2) B(3) G(4) G(5)
```

This assignment is shown diagrammatically to be a valid 3-colouring of $G$ in Figure 1.3.

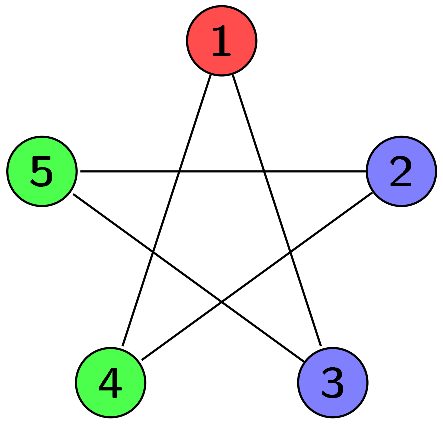


Figure 1.3: Our discovered solution to 3-Col on $G$.

## 1.4 Project goals

It is important to set out a clear and concise list of criteria which the project should meet. These should be kept in mind during all stages of planning, implementation and evaluation to ensure that the project achieves its goals successfully.

From my project proposal (reproduced in Appendix D) and the initial research into previous work on solving NP-complete problems and SAT-solving described in Chapter 2, I have derived the following goals for *Twist*:

1. **Completeness:** It should be possible to input *any* NP-complete problem into *Twist* to be solved.
2. **Correctness:** The solution returned by *Twist* should always be exact. A valid satisfying assignment should be returned in the original domain, unless no such assignment exists.
3. **Performance:** *Twist* should be able to solve problems at a speed competitive with current techniques.
4. **Extensibility:** The use of *any* 3rd-party SAT-solver should be allowed to solve the SAT instances. This is important as it means our optimisations are not bounded by current technology, but are open to harness future developments in SAT-solving techniques.

## 1.5 Beyond project goals

In addition to the criteria mentioned above, I was able to achieve several further goals which went beyond my project proposal:

- Analysis was made into how the difficulty of NP-complete problem instances can be identified using phase transitions (see Section 2.2.2). Significant differences were discovered in the performance at these transitions of *Twist* compared with existing methods, as discussed in Section 4.3.
- *Twist* was extended to allow the use of pseudo-Boolean constraints in Section 3.10. These allow constraints that are difficult to express in pure $\exists$SO to be more easily encoded, and solved with much greater performance as demonstrated in Section 4.3.5.

# Chapter 2

# Preparation

In this chapter the relevant theoretical background into complexity theory and logic is introduced. Preparatory research into current work on NP-complete problems is described, along with the developments made into Sat-solving which facilitate the approach taken by *Twist* to solve them. Finally the various development choices taken are justified.

## 2.1 Theoretical background

### 2.1.1 Decision problems

We begin with a description of some of the problems that are referred to throughout this dissertation, and will be used to describe various complexity classes. These are all *decision problems*: problems that have either a 'yes' or 'no' answer.

#### Boolean Satisfiability (Sat)

A formula of propositional logic (Section 2.1.3) is *satisfiable* if there exists some assignment of the variables in the formula to the Boolean values $\{$`true`, `false`$\}$ that makes the whole formula evaluate to `true`. The Boolean Satisfiability problem (Sat) is the problem of deciding whether or not a given propositional logic formula is satisfiable.

A simple example of an instance of the SAT problem is the formula:

$$a \vee (b \wedge \neg c)$$

This instance is satisfiable, as the assignment $a = \texttt{false}, b = \texttt{true}, c = \texttt{false}$ evaluates it to the value `true`.

However, the following formula is not satisfiable:

$$(a \vee b) \wedge \neg a \wedge \neg b$$

No assignments of $a$ and $b$ exist that can evaluate it to `true`.

Cook's theorem [9] proves BOOLEAN SATISFIABILITY to be NP-complete, meaning no polynomial time algorithm is known for solving it. One method would be simply to test every possible combination of `true` and `false` for each variable, and check if any of them satisfy the formula. However, for a formula containing $n$ variables there are $2^n$ such combinations to check. Even if we can evaluate $1,000,000$ attempts per second, to test all the combinations for a formula containing just 80 variables would take more than twice the current age of the universe.

However, over recent years dramatic advances have been made into finding more efficient algorithms to solve this problem, described in Section 2.3.1. These 'SAT-solvers' can now perform extremely well on the types of instances found in practice, and can solve problems with millions of variables in a matter of seconds.

#### GRAPH $k$-COLOURABILITY ($k$-COL)

We are given a graph $G = (V, E)$, where $V$ is a set of vertices, and $E \subseteq V \times V$ is a set of undirected edges between these. The problem of GRAPH $k$-COLOURABILITY asks whether or not it is possible to assign each vertex of the graph one of $k \in \mathbb{N}$ colours such that no two vertices connected by an edge share the same colour. An example of a 3-colourable graph with a valid 3-colouring is shown in Figure 1.3.

GRAPH $k$-COLOURABILITY will be referred back to throughout this dissertation as it is a simple example of the type of problem that we would like *Twist* to be able to solve, and also an important problem in computer science itself with many practical applications. These include scheduling jobs that rely on shared resources, and register allocation performed by compilers.

#### Optimisation problems

The problems we wish to solve are often not decision problems, but instead *optimisation problems*. Instead of having just a 'yes' or 'no' answer, these have a number of feasible solutions and we wish to seek the best (optimum) of these. For example, given some graph we may wish to find the smallest number of colours which can be used to colour it. However, it is possible to transform optimisation problems into decision problems and vice versa without significantly changing their computational difficulty. Therefore, it suffices to focus on decision problems from now on.

### 2.1.2 Complexity classes

The various complexity classes required to define NP-complete problems will now be briefly described.

#### P: polynomial time

The class P contains all decision problems that can be solved in polynomial time ($O(n^k)$ for some $k > 0$, where $n$ is the length of our input). More precisely we specify this bound by the number of steps the algorithm takes to complete on a deterministic Turing machine.

#### Example: Graph 2-Colourability (2-Col)

When $k = 2$, the Graph $k$-Colourability problem is in the class P. To prove this we present a polynomial time algorithm for solving it: colour the first node red, then each of its neighbours blue, then each of these vertices' neighbours red, and so on. If we can continue this until all vertices are coloured then we have found a satisfying assignment, and the answer to the decision problem is 'yes'. If not, and we are forced to colour two connected vertices with the same colour, then the answer is 'no'. An example valid 2-colouring can be seen in Figure 2.1.

#### NP: non-deterministic polynomial time

NP contains decision problems whose 'yes' solutions can be verified in polynomial time. These problems can be solved in polynomial time by a non-deterministic Turing machine, by the machine first non-deterministically guessing a solution, and then verifying or rejecting this in polynomial time. This also means that $\mathsf{P} \subseteq \mathsf{NP}$.

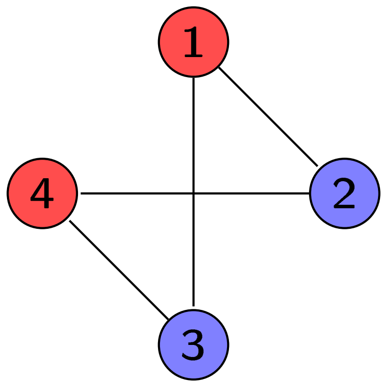


Figure 2.1: A valid 2-colouring of a graph.

### Example: Boolean Satisfiability

We can see that Sat $\in$ NP as the validity of an assignment can easily be checked in polynomial time by substituting the values into the formula and evaluating each of the logical connectives ($\wedge, \vee$ and $\neg$) until the Boolean value of the formula is found. This only takes a number of steps polynomial in the length of the input formula, and so Sat $\in$ NP.

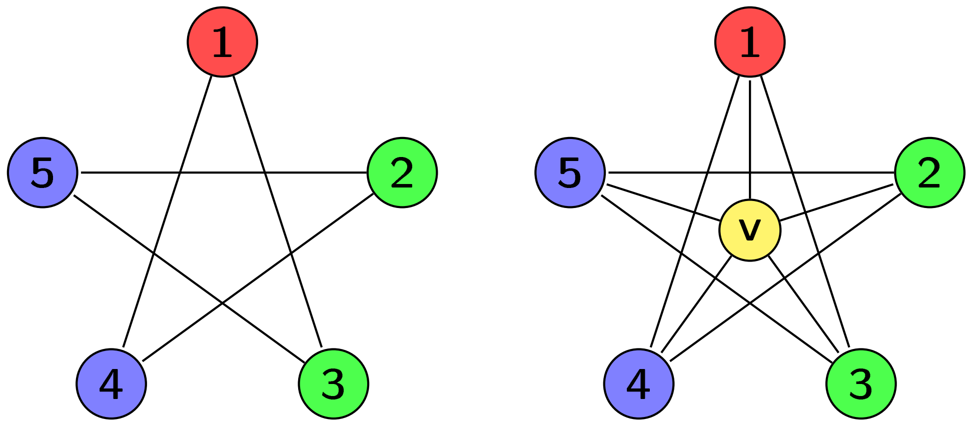
### NP-complete

NP-complete contains the most difficult NP problems: those that are also NP-hard. A problem $\mathcal{P}$ is NP-hard if it is possible to reduce any other NP problem $\mathcal{Q}$ to $\mathcal{P}$ in polynomial time. By reduce, we mean there exists some polynomial time algorithm $f$ that can transform instances $q$ of problem $\mathcal{Q}$ into instances $p$ of problem $\mathcal{P}$ which are equivalent: the answer to $p$ is the same as the answer to $q$.

### Example: Graph 4-Colourability (4-Col)

All $k$-Colourability problems can be seen to be in NP as colourings can be verified in $O(|E|)$ time by checking that each edge is monochromatic. We already know that the problem of Graph 3-Colourability (3-Col) is NP-complete [17]. As this means *all* other NP-complete problems can be reduced to 3-Col, for us to show that Graph 4-Colourability is NP-complete it suffices to show that we can reduce 3-Col to 4-Col in polynomial time.

Our function $f$ will take a graph, and add an additional vertex $v$, and edges from $v$ to all other vertices, as in Figure 2.2. This new graph is then 4-colourable if and only if the original graph was 3-colourable. As this reduction takes only



This graph is 3-colourable... ...if and only if this graph is 4-colourable.

Figure 2.2: Reduction from a 3-COLOURABILITY problem to an equivalent 4-COLOURABILTY problem.

$O(|V|)$ time, and we know we can reduce any NP-complete problem to 3-COL in polynomial time, we have found a way to reduce any NP-complete problem to 4-COL in polynomial time. This proves 4-COL to be NP-hard, and hence NP-complete.

#### P vs NP

No polynomial time algorithm has yet been discovered for any NP-complete problem, leading many to believe that none exist. Despite this, no super-polynomial lower bounds on the time complexity for any NP-complete problem have been proven either. As all NP-complete problems are reducible to each other in polynomial time, finding either of these would respectively prove that either all NP-complete problems can be solved in polynomial time (i.e $\mathsf{P} = \mathsf{NP}$), or none can ($\mathsf{P} \neq \mathsf{NP}$). This question of whether or not $\mathsf{P} = \mathsf{NP}$ is one of the most important outstanding problems in mathematics.

### 2.1.3 Existential second-order logic (∃SO) and other logics

Fagin's theorem says the class NP is exactly equivalent to a certain form of predicate logic named existential second-order logic, which is defined here. We begin with a description of less expressive logics, which will form a basis for ∃SO.

### Propositional logic

A sentence of propositional logic is made up of a number of Boolean variables (the propositions), which can either have the value `true` or `false`. These variables are then related by the logical connectives: conjunction ($\wedge$), disjunction ($\vee$), implication ($\rightarrow$), bi-implication ($\leftrightarrow$) and negation ($\neg$). Any formula of propositional logic is a valid instance of the BOOLEAN SATISFIABILITY problem.

### First-order logic

First-order logic extends propositional logic with relations and predicates, and by allowing us to quantify over the elements of a relation. A sentence of first-order logic is inductively defined by the following rules:

- A *term* is any constant, variable, or $n$-argument function $f(t_1, ..., t_n)$, where each $t_i$ is a term.
- An *atomic sentence* is an $n$-ary predicate symbol $P(t_1, ..., t_n)$, or $t_1 = t_2$, where each $t_i$ is a term.
- A *sentence* is an atomic sentence, two sentences $\varphi$ and $\varphi'$ connected by some logical connective $\in \{\wedge, \vee, \rightarrow, \leftrightarrow\}$, $\neg\varphi$, or $\forall x.\varphi$ or $\exists x.\varphi$ where $x$ is a variable free in $\varphi$.

### Second-order logic

Second-order logic is an extension to first-order logic which allows us to quantify over relations themselves, not just their elements. For example, the second-order sentence $\exists P.(P(x) \wedge P(y))$ says that $x$ and $y$ share some property $P$. This cannot be expressed in first-order logic.

### Existential second-order logic ($\exists$SO)

Existential second-order logic is a fragment of second-order logic with only *existential* second-order quantifiers (i.e. no universal quantifiers that quantify over second-order variables), and these must be at the start of the formula. This gives it the form $\exists R_1 ... \exists R_n.\varphi$, where each $R_i$ is a second-order variable and $\varphi$ is a first-order formula.

### 2.1.4 Fagin's theorem

In 1974 Ronald Fagin proved that the class of problems $\mathsf{NP}$ is exactly equivalent to the set of formulae expressible in existential second order logic ($\exists$SO) [15]. This result is important as it was the first machine-independent characterisation of a complexity class. As $\mathsf{NP}$-complete $\subseteq$ $\mathsf{NP}$, it means that all problems that *Twist* should accept can be expressed as a formula of this type, and so it suffices for my specification language for $\mathsf{NP}$-complete problems designed in Section 3.1 to be able to describe sentences of $\exists$SO.

#### Proof sketch

We prove the equivalence between $\exists$SO formulae $\Phi$ and $\mathsf{NP}$ problems $\mathcal{P}$ by proving both directions of the bi-implication separately:

- Sentence $\Phi$ is expressible in $\exists$SO $\Rightarrow$ $\Phi$ can be evaluated in $\mathsf{NP}$:
  Suppose $\Phi = \exists R_1, ..., R_n.\varphi$, where $\varphi$ is a first-order sentence.
  Our non-deterministic Turing machine can simply non-deterministically guess $R_1, ..., R_n$, and then check if $\varphi(R_1, ..., R_n)$ holds. This check can be done in polynomial time, hence $\Phi$ can be evaluated in $\mathsf{NP}$.

- Problem $\mathcal{P}$ can be evaluated in $\mathsf{NP}$ $\Rightarrow$ $\mathcal{P}$ is expressible as $\exists$SO sentence $\Phi$:
  We first define predicates $R_1, ..., R_n$ to encode the machine's tape contents, state, and head position at all possible times during the computation. These can be seen to describe the execution tableau of our evaluation, which can then be arbitrarily chosen (as they will be existentially quantified over).
  We must then express that this encoded information must describe a valid execution trace of a non-deterministic Turing machine. This can all be done with a first-order formula, that we call $\varphi$.
  Our final formula $\Phi$ is then given as the $\exists$SO sentence $\exists R_1, ..., R_n.\varphi$, which exactly describes $\mathcal{P}$.

#### Example $\exists$SO problem description: 3-Col

As an example $\exists$SO description of an $\mathsf{NP}$-complete problem I will use Graph 3-Colourability (3-Col), defined in section 2.1.2. 3-Col can be written in $\exists$SO as follows, with a brief explanation of the purpose of each part of the formula on the right hand side:

$$\begin{array}{ll}
\exists R\ \exists G\ \exists B\ ( & \}\ \text{There are 3 colours: R, G and B.} \\
\left.\begin{array}{l}
\quad \forall v \in V(\ \ (R(v) \wedge \neg G(v) \wedge \neg B(v)) \\
\qquad\qquad \vee(\neg R(v) \wedge G(v) \wedge \neg B(v)) \\
\qquad\qquad \vee(\neg R(v) \wedge \neg G(v) \wedge B(v))\ )
\end{array}\right\} & \text{Each vertex } v \in V \text{ is coloured with exactly one colour.} \\
\quad \wedge & \\
\left.\begin{array}{l}
\quad \forall (x,y) \in E(\ \ (R(x) \wedge \neg R(y)) \\
\qquad\qquad \vee(G(x) \wedge \neg G(y)) \\
\qquad\qquad \vee(B(x) \wedge \neg B(y))\ )
\end{array}\right\} & \text{The endpoints of each edge } (x,y) \in E \text{ cannot be the same colour.} \\
) &
\end{array}$$

## 2.2 Research into solving NP-complete problems

### 2.2.1 Current techniques for solving NP-complete problems

NP-complete problems are typically solved using a special purpose algorithm for one particular problem. Due to their intractability, this algorithm cannot hope to exhaustively search all possible solutions. Instead, the approach taken depends on whether or not we require an *exact* solution: one which is guaranteed to be optimal. Depending on the answer to this question, we can either use an approximate or heuristic algorithm to reduce the time it takes to find a solution to our problem.

#### Approximation

If it is not important that the solution returned is always optimal, then we can instead choose to tackle the much simpler task of searching for an approximate solution, which is merely close to optimal. Approximate solutions are much easier to find, and often algorithms that run in polynomial time can be found which give approximate solutions guaranteed to be within some bound of the optimum solution [22]. For example, we can find an approximate solution for the Vertex Cover problem (see Appendix C.8) that is at most twice the size of the optimal one by simply repeatedly adding *both* endpoints of any uncovered edge to the vertex cover, until none remain.

However not all problems can be approximated either efficiently or well, and in fact many lower bounds on the optimality of efficient approximations have been found [18].

#### Heuristics

If we do require the optimal solution to our problem, then we must use an *exact* algorithm that can guarantee this. In this case, to overcome the exponential running time we can use heuristics to search the most likely possible solutions first, in the hope of finding a solution quickly on average.

These heuristics are designed to work well in most cases, but in the worst case their performance is still exponential. An example heuristic for graph colouring is to simply colour each vertex with the first available colour, and backtrack if we get stuck. This can work reasonably well in practice, however on certain shapes of graphs the order it chooses colourings will result in extremely poor performance.

### 2.2.2 Distribution of difficult NP-complete problems

The classification of problems as NP-complete is based entirely on asymptotic worst-case analysis. However not all instances of these problems are difficult in practice. Empirical results have shown that in fact the majority of randomly generated instances of many NP-complete problems can often be solved at well below the worst case running time, and that the exponential blow-up characteristic of NP-complete problems only occurs for a very small fraction of problem instances [31, 35].

#### Underconstrained and overconstrained problems

This unusual distribution is due to most problems being either *underconstrained* or *overconstrained*. A problem is underconstrained if it contains few constraints compared with the number of unknowns in the problem. This means there is a high density of solutions, making them very easy to find. A problem is overconstrained if it has many constraints compared with the amount of unknowns. This means solutions are very unlikely to exist, meaning the seach tree is small.

Search-based algorithms are very good at solving problems in one of these two categories. Possible solutions can either very quickly be found to be a correct solution (as there are few constraints restricting it), or very quickly be eliminated (as there are many constraints restricting it). It has been empirically found that almost all instances do in fact fall in to one of these two 'easy' categories [35], meaning they can be solved very efficiently.

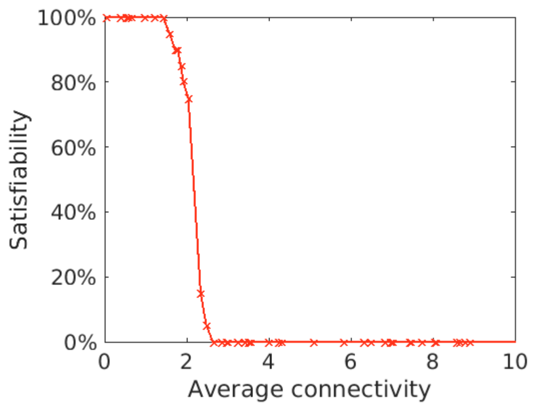


Figure 2.3: Satisfiability distribution of 3-COLOURABILITY of uniformly randomly generated graphs of 50 vertices.

### Phase transitions in NP-complete problems

How constrained some problem instance is can often be well described by just one property of the instance [6]. When this property is plotted against satisfiability (i.e. the percentage of instances that have a 'yes' solution), a step-like *phase transition* can often be seen, in the shape of Figure 2.3. Such phenomena have been discovered in several NP-complete problems, including GRAPH COLOURABILITY, HAMILTONIAN CYCLE, and BOOLEAN SATISFIABILITY [6].

One such property for the problem of GRAPH COLOURABILITY is the average connectivity of the graph instances [6]. For a graph $G = (V, E)$ this is the mean number of edges each node has, equal to $|E|/|V|$. When this is plotted against colourability of uniformly randomly generated graphs as in Figure 2.3 we see a clear phase transition between colourable and non-colourable graphs. There exists a certain threshold of average connectivity above which almost all graphs are non-colourable due to being overconstrained, and below which almost all graphs are colourable due to being underconstrained.

For the BOOLEAN SATISFIABILITY problem (with fixed clause length), a critical property is the ratio of variables to clauses. Below a certain value of this almost all instances are satisfiable, and above it almost none are.

### Difficult NP-complete problems

The 'difficult' NP-complete problems - those which take the most time to solve - are often not those with the largest instances, but instead the problems that occur at this boundary between underconstrained and overconstrained instances seen in phase transition diagrams. The DPLL algorithm can easily solve a 3-SAT instance with 1000 variables and 3000 clauses (underconstrained), or an instance with 1000 variables and 50,000 clauses (overconstrained) in seconds. However even a highly optimised DPLL solver cannot solve random 3-SAT instances with just 250 variables and 1075 clauses in a short amount of time [31].

This is because at these boundaries the probability of a solution is low but non-negligible [6]. There are typically many well-separated almost-solutions (or local minima) to be evaluated. This traps search algorithms such as DPLL and causes them to trash, as they backtrack across large search trees. Evidence of this can be seen as the number of DPLL calls required to solve 3-SAT instances rises sharply close to this threshold [31].

### Implications for *Twist*

The distribution of difficult instances of NP-complete problems according to phase transitions has a number of implications for *Twist*. It indicates that the approach of using advanced SAT-solvers is likely to be successful: SAT-solvers are good at solving difficult instances efficiently, and it is these difficult instances that take the most time with naive solving algorithms. The distribution of difficult instances also means that although the SAT instances generated by *Twist* are likely to be very large, this does not necessarily mean they will take a long time to solve. We have seen that solving time is often more dependent on how constrained the instances are than their size.

## 2.3 Research into SAT-solving

SAT-solvers have made tremendous progress in the past 15 years. These developments mean that translation of other NP-complete problems to SAT has become a viable technique for solving them efficiently. This is the approach taken by *Twist*.

### 2.3.1 Developments in Sat-solving

Cook's proof that Sat is NP-complete [9] tells us that there are no known algorithms for solving it in less than exponential worst-case time complexity. Nevertheless, many effective heuristic-based algorithms have been discovered. The most famous and successful of these is the backtracking search-based DPLL algorithm invented by Davis, Putnam, Logemann and Loveland in 1962 [10], which for certain natural distributions of problems has polynomial average-case time complexity.

Given ten minutes of computing time, the original DPLL algorithm could only solve problems with up to around 15 variables. Since then there have been (and continue to be) dramatic advances in Sat-solver performance. The introduction of conflict-directed clause learning (CDCL) [30] has enabled solvers to trim large parts of the search tree of possible solutions. Studies have shown that restarting the search from time to time also helps to decrease the average runtime significantly [20]. The next variable to be assigned can be chosen more effectively by selecting ones which seem to be more important [32]. The invention of improved data structures has facilitated new methods such as the two watched literal scheme, which avoids unnessary work during unit propagation [32]. Attempts have also been made to adjust implementations to make more intelligent utilisation of the provided hardware resources, such as the cache and the prefetching unit [19]. As these developments increase the size of formulas that can be solved, Sat solvers also become useful for a more diverse range of applications. These new applications push Sat-solvers' development even further, leading to additional novel algorithms continually being introduced [33]. As a result of such developments, Sat-solvers can now handle problems with millions of variables in a matter of seconds [2].

Since 2002 regular Sat competitions [1, 2] have been held in which state of the art solvers compete against one another to solve the greatest number of problems, or solve them in the least amount of time. This encourages progress both through the competitive environment, and by promoting awareness of new solvers and solving techniques.

By allowing *Twist* to feed its Boolean Satisfiability output into any user-specified Sat-solver, I hope to be able to harness the power of these advances. This extensibility also means that *Twist* will be able to utilise any future advances made in Sat-solving, thus increasing the efficiency by which it can solve problems.

### 2.3.2 SAT-solver interface requirements

It is important that the interface between *Twist* and an external SAT-solver is well specified, as this will allow the specific solver used to be interchanged easily and reliably.

We begin by specifying the format of the SAT problem output from *Twist* to the SAT-solver as shown in Figure 1.1. So as to be operable with as wide a range of SAT-solvers as possible, I have chosen to give my output in the DIMACS CNF format, as it is the standard used by most modern SAT-solvers and all major SAT-solving competitions.

#### Conjunctive normal form (CNF)

Conjunctive normal form is a standard structure of expressions of propositional logic. To define CNF we use the following recursive definitions:

- A *literal* is a Boolean variable $x$ or its negation $\neg x$
- A *clause* is a disjunction of literals, e.g. $a \vee b \vee \neg c$
- A *CNF* formula is a conjunction of clauses, e.g. $(a \vee b) \wedge (\neg b \vee c) \wedge (a \vee \neg d)$

Any formula of propositional logic can be converted into a logically equivalent CNF formula, as described in Section 3.7.

#### DIMACS CNF file format

DIMACS CNF [5] is a standard file format for CNF expressions, used by all major SAT-solvers. It contains some meta-data describing the file type and the number of variables and clauses in the formula. Then the clauses follow one per line, with each variable given by a unique integer, and its negation given by the negative of the integer. Line endings are marked by `0`s, and comment lines begin with `c`. For example take the following CNF formula:

$$(x_1 \vee \neg x_5 \vee x_4) \wedge (\neg x_1 \vee x_5 \vee x_3 \vee x_4) \wedge (\neg x_3 \vee \neg x_4)$$

In DIMACS CNF format this is:

```
c example formula
p cnf 5 3
1 -5 4 0
-1 5 3 4 0
-3 -4 0
```

#### Sat-solver output

A Sat-solver will find the 'yes'/'no' answer to the supplied Sat instance, and if the answer is 'yes' then it will return a satisfying assignment of the variables in the Sat instance. However we would like the assignment in terms of the domain of the original problem, so we must translate this back as shown in Figure 1.1. This requires us to be able to read in the standard Sat-solver outputs used in Sat-solver competitons [2].

The output of a Sat-solver given an unsatisfiable instance such as $x_1 \wedge \neg x_1$ is simply:

```
UNSAT
```

The output of a Sat-solver given a satisfiable instance however will contain the satisfying assignment of the variables found. Here we give the output to the problem $x_1 \vee (x_2 \wedge \neg x_3)$. Assignments of a variable $x_n$ to `true` are given by `n`, an assignment to `false` is given as `-n`, and `0` marks the end of the file:

```
SAT
-1 2 -3 0
```

### 2.3.3 Selection of a Sat-solver

The Sat-solver I have decided to use during development and testing of *Twist* is MiniSat version 2.2.0, available online under the MIT licence [12]. It is a small yet efficient conflict-driven clause learning algorithm, designed to be easily modifiable and extensible [13]. It is also very popular: 17 of the 44 Sat-solvers presented at the 2014 Sat competition were based on MiniSat [3]. All these factors make it a good first choice of Sat-solver to use with *Twist*. A comparison of the performance of this and several other leading Sat-solvers is given in Section 4.4.3.

## 2.4 Development practices

As with any project of this size, software engineering techniques such as a structured development model and well informed choice of tools are vital in ensuring my project successfully meets its goals in the given time frame.

### 2.4.1 Development model

As the translation process from ∃SO problem specification to CNF Sat instance is highly modular (see Figure 1.1), I employed an iterative development model. Each stage of this process was built up incrementally and unit tested separately before development of the next stage began. This worked to ensure that each individual part was functioning correctly, and in turn that the system worked reliably as a whole.

The first problem I worked on solving was Graph 3-Colourability as it is relatively simple and easy to express in ∃SO. Once the full pipeline of stages was working correctly for this and I had met my basic requirements, I then moved on to testing it with more complex problems, and making any changes required to solve these. Timings of each stage were made to identify bottlenecks, which were then optimised. A suite of regression tests was used to ensure that none of these changes introduced bugs into *Twist*.

### 2.4.2 Choice of tools

#### Programming language

The programming language I chose for this project was OCaml version 4.02.1 [21]. Its pattern matching capabilities, performance, portability, and development tools (such as `ocamllex` and `oocamlyacc`) mean it is very well suited to the efficient manipulation of abstract syntax trees required required by *Twist*, as described in Section 3.3. This also meant that the source code written was cleaner and clearer than it would have been had I used a less well suited language, as illustrated by the source code sample in Appendix A. This has meant it has been much easier to debug and hopefully will be more approachable for future developers.

#### Revision control

Git version 1.9.1 was used for revision control to save and compare different versions of *Twist* during the incremental development process. Git's branching and merging features were well suited for the non-linear workflow enforced by this development model, and allowed more structured development of new features.

Daily backups were made to the university Managed Cluster Service (MCS) to ensure that there was no single point of failure.

# Chapter 3

# Implementation

This chapter describes in turn the implementation of each of the stages of *Twist* shown in Figure 1.1. We begin with a description of how the languages for describing problem specifications and instances were designed, and the data structure used to express the various types of logical formula throughout the reduction process. Finally we conclude with an extension to the core project that allows for more efficient solving of more complex problems.

## 3.1 Designing a specification language for NP-complete problems

I wish to construct a specification language that can accept descriptions of *all* NP-complete problems. We have already seen from Fagin's theorem [15] that to do this it suffices to use a language that accepts formulae of existential second-order logic ($\exists$SO) (see Section 2.1.4). Hence my specification language should adhere to $\exists$SO as strictly as possible in order to maintain equivalence to the class NP.

However, $\exists$SO descriptions of NP-complete problems are often very different from their high-level English descriptions, containing many different predicate and logical symbols and nested quantifiers. In order to increase both readability and writeability I have included in my specification language a number of deviations from and extensions to pure $\exists$SO, outlined in the following subsections.

### 3.1.1 Preamble

A preamble is added to aid clarity of the inputs and outputs expected of a particular problem. Pure ∃SO is of the form $\exists R_1 ... \exists R_n . \varphi$ (see Section 2.1.3), where the first-order formula $\varphi$ often contains many different predicate symbols, some of which are the predicates $R_1 ... R_n$ we are solving for, and some of which are input to the problem in the supplied instance. To increase the readability of my problem specifications I have extended the ∃SO description with a preamble which explicitly lists which predicates are given as input and which the problem is solving for. This then allows us to discard the second-order quantifiers $\exists R_1 ... \exists R_n$ from our ∃SO description as these will be listed in the `Find` section. We then only need to give the remaining first-order formula $\varphi$ to describe the problem's constraints. Our problem is now described as follows:

`Given:`
< List of predicate symbols given in instance >
`Find:`
< List of predicate symbols we are solving for >
`Satisfying:`
< First-order problem description over the above predicate symbols >

### 3.1.2 Logical symbols

Predicate logic uses a number of logical symbols: $\forall, \exists, \in, \wedge, \vee, \neg, \rightarrow, \leftrightarrow$. While Unicode versions of all these symbols do exist [36] they are not easily typed, so instead the characters given in Table 3.1 are used. These follow common substitutions used in programming languages for these logical operators, and for those operators that are not commonly used more explicit substitutions are used. This means that these substitutions should be both easy to type and easy to read their logical meaning.

### 3.1.3 Comments

To bridge the gap between the high-level English description of a formula and its ∃SO formula comments can be inserted at any point, delimited with ($*$ and $*$). These allow the use of plain text annotations to aid in the understanding of the often complex problem specifications. Again the syntax of these follows the syntax of many programming languages.

| Logical symbol | Substitute symbol |
|---|---|
| $\forall$ | `forall` |
| $\exists$ | `exists` |
| $\in$ | `in` |
| $\wedge$ | `&` |
| $\vee$ | `\|` |
| $\neg$ | `~` |
| $\rightarrow$ | `->` |
| $\leftrightarrow$ | `<->` |

Table 3.1: Substitutions for logical symbols used in my specification language.

### 3.1.4 3-Col specification example

Here we give the description of the Graph 3-Colourability problem that will be input into *Twist*, along with some instance graph. The original ∃SO description from Section 2.1.4 is also given for comparison:

```
Given:
  V E
Find:
  R G B
Satisfying:
  (* each vertex is only 1 colour *)
  (forall x in V (
       ( R(x) & ~G(x) & ~B(x))
     | (~R(x) &  G(x) & ~B(x))
     | (~R(x) & ~G(x) &  B(x)) ))
  &
  (* all connected vertices are
     different colours *)
  (forall x y in E (
       (R(x) & ~R(y))
     | (G(x) & ~G(y))
     | (B(x) & ~B(y)) ))
```

$$
\begin{array}{l}
\exists R\ \exists G\ \exists B\ ( \\
\quad \forall v \in V( \quad (R(v) \wedge \neg G(v) \wedge \neg B(v)) \\
\qquad\qquad \vee (\neg R(v) \wedge G(v) \wedge \neg B(v)) \\
\qquad\qquad \vee (\neg R(v) \wedge \neg G(v) \wedge B(v))\ ) \\
\quad \wedge \\
\quad \forall (x,y) \in E( \quad (R(x) \wedge \neg R(y)) \\
\qquad\qquad \vee (G(x) \wedge \neg G(y)) \\
\qquad\qquad \vee (G(x) \wedge \neg G(y))\ ) \\
)
\end{array}
$$

Various other common NP-complete problems are listed in both their usual English descriptions and my specification language in Appendix C.

## 3.2 Describing problem instances

So that we can solve many different instances of an $\mathsf{NP}$-complete problem $\mathcal{P}$, the $\exists$SO specification $\Phi$ for this problem and an instance $\mathfrak{A}$ are input separately into *Twist*. This then allows us to easily substitute in new instances $\mathfrak{A}'$ to form a new $\mathcal{P}'$ without having to completely rephrase the decision problem $\mathcal{P}$ to solve each new instance.

Thus far most of the problem instances we have encountered have been graphs consisting of a set of vertices $V$ and a set of undirected, non-weighted edges $E \subseteq V \times V$. However, different problems can take instances of many different forms. For example the Exact Cover by 3-sets problem described in Appendix C.3 requires a set $X$ and a collection $C$ of 3-element subsets of $X$. The exact relations required for an instance of a problem will be given in the `Find:` section of the problem preamble (see Section 3.1.1).

### 3.2.1 Instance requirements

To allow for such a wide range of instances to be used, the format required by *Twist* should be as general as possible. All instances consist of one or more sets, or relations. It must be possible to differentiate these sets from one another, and to refer to them from the problem specification. Elements of these sets may be single values such as a set of vertices $\{x, y, z\}$, or multiple values such as a set of edges $\{\{x_1, x_2\}, \{y_1, y_2\}, \{z_1, z_2\}\}$.

### 3.2.2 Instance file format

To meet this specification I have designed the following plain text file format for specifications. Each set is identified by a single upper case character, equal to the predicate symbol used in the problem specification to refer to it. Elements of a set are separated by a new line, to allow individual elements to be edited easily. Each line begins with the set ID character, followed by a number of values. A value can be any string of characters, but will often be an integer as to be easily distinguishable.

### 3.2.3 Graph instance example

An example of the graph instance from Section 1.3.1 described in this file format is given in Table 3.2.

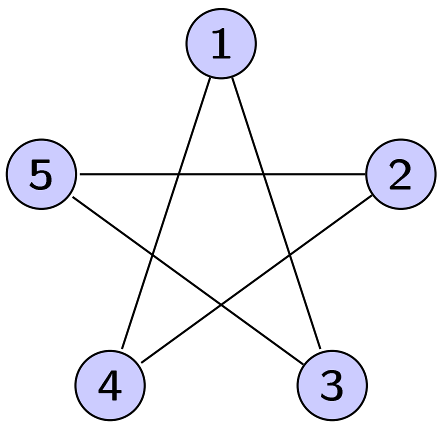
| Pictorial representation | Set representation | *Twist* file format |
|---|---|---|
|  | G = (V,E)<br><br>V = {1,2,3,4,5}<br><br>E = {{1,3},<br>{1,4},<br>{2,4},<br>{2,5},<br>{3,5}} | `V 1`<br>`V 2`<br>`V 3`<br>`V 4`<br>`V 5`<br><br>`E 1 3`<br>`E 1 4`<br>`E 2 4`<br>`E 2 5`<br>`E 3 5` |

Table 3.2: Example graph instance represented in pictorial and set formats, and the format required by *Twist*.

## 3.3 Logical expressions as ASTs

The reduction of an NP-complete problem into a CNF SAT instance requires several stages of conversions and transformations, as shown in Figure 1.1. Each stage will take a logical formula of the form $X$ and transform it into an equivalent formula of the form $Y$, which is one step closer to the required form of our output. The data structure used to represent each of these types of formula will be an abstract syntax tree (AST). An example AST for a propositional formula is shown in Table 3.3.

This data structure has a number of advantages over the usual sequential representation of a formula. Firstly it allows us to represent only the information about our expressions which is necessary for the required transformations. Brackets can be discarded as they are implicit in the tree structure, along with other syntax necessary for disambiguating the problem description. Secondly, its hierarchical structure is very well suited to the kinds of recursive operations described throughout this chapter that we will be performing on it. Here OCaml's pattern matching functionalities can be exploited to allow us to elegantly and efficiently perform these operations on the ASTs. An example of such code is given in Appendix A. Abstracting the results of each stage into a different AST allows us to distinguish between them more definitely and aids code modularity, the benefits of which are discussed in Section 2.4.1.

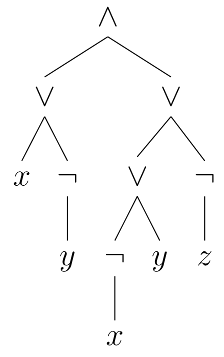
| Sequential representation | AST representation |
|---|---|
| $(x \vee \neg y) \wedge (\neg x \vee y \vee \neg z)$ |  |

Table 3.3: Comparison of equivalent sequential and AST representations of a formula of propositional logic.

## 3.4 Combining the problem specification and instance

The system is input with a problem specification $\Phi$ and an instance $\mathfrak{A}$ as in Figure 1.1, which together represent the problem $\mathcal{P}$ that we wish to solve. But to convert this into a CNF propositional formula appropriate for a Sat-solver, we must first combine $\Phi$ and $\mathfrak{A}$ into *one* first-order formula describing $\mathcal{P}$. This formula would then, for example, represent the statement "Graph $G$ is 3-colourable". This is done by "expanding out" the problem specification $\Phi$ with values from the instance $\mathfrak{A}$.

### 3.4.1 Expanding problems out with instance values

To perform this combination of problem specification and instance we will insert the values of the instance into the problem specification, which is describing the conditions these values must meet for the problem to be satisfiable. We do this by performing the following transformations[1]:

- Replace each $\forall x \in X.\ \varphi(x)$ with $\bigwedge_{x_i \in X} \varphi(x_i)$
- Replace each $\exists x \in X.\ \varphi(x)$ with $\bigvee_{x_i \in X} \varphi(x_i)$
- Replace each $R(\vec{a})$, for $R \in \mathfrak{A}$, with its truth value in $\mathfrak{A}$

[1] Where set $X = \{x_1, x_2, ..., x_n\} \in \mathfrak{A}$.

As all sets $X$ or relations $R$ in the instance must be finite, the resulting single expression produced by these transformations will be logically equivalent to the interpretation of the problem specification over the given instance [26]. The rules are applied recursively to our problem AST, matching tree nodes against the left hand side of a transformation, and expanding them out into a large conjunction or disjunction or truth value given by the right hand side. Once all possible substitutions have been made we are left with a formula in our second form AST, representing a first-order formula describing $\mathcal{P}$.

**Example problem expansion**

Applying these rules to the 3-Col specification from Section 3.1.4 and the graph instance from Section 3.2.3 we get the expression:

$$
\begin{aligned}
&((R(5) \wedge \neg G(5) \wedge \neg B(5)) \vee (\neg R(5) \wedge G(5) \wedge \neg B(5)) \vee (\neg R(5) \wedge \neg G(5) \wedge B(5)))\\
\wedge\ &((R(4) \wedge \neg G(4) \wedge \neg B(4)) \vee (\neg R(4) \wedge G(4) \wedge \neg B(4)) \vee (\neg R(4) \wedge \neg G(4) \wedge B(4)))\\
\wedge\ &((R(3) \wedge \neg G(3) \wedge \neg B(3)) \vee (\neg R(3) \wedge G(3) \wedge \neg B(3)) \vee (\neg R(3) \wedge \neg G(3) \wedge B(3)))\\
\wedge\ &((R(2) \wedge \neg G(2) \wedge \neg B(2)) \vee (\neg R(2) \wedge G(2) \wedge \neg B(2)) \vee (\neg R(2) \wedge \neg G(2) \wedge B(2)))\\
\wedge\ &((R(1) \wedge \neg G(1) \wedge \neg B(1)) \vee (\neg R(1) \wedge G(1) \wedge \neg B(1)) \vee (\neg R(1) \wedge \neg G(1) \wedge B(1)))\\
\wedge\ &((R(4) \wedge \neg R(1)) \vee (G(4) \wedge \neg G(1)) \vee (B(4) \wedge \neg B(1)))\\
\wedge\ &((R(2) \wedge \neg R(4)) \vee (G(2) \wedge \neg G(4)) \vee (B(2) \wedge \neg B(4)))\\
\wedge\ &((R(5) \wedge \neg R(2)) \vee (G(5) \wedge \neg G(2)) \vee (B(5) \wedge \neg B(2)))\\
\wedge\ &((R(3) \wedge \neg R(5)) \vee (G(3) \wedge \neg G(5)) \vee (B(3) \wedge \neg B(5)))\\
\wedge\ &((R(1) \wedge \neg R(3)) \vee (G(1) \wedge \neg G(3)) \vee (B(1) \wedge \neg B(3)))
\end{aligned}
$$

This one sentence now completely describes the problem of 3-Colourability on this specific graph.

## 3.5 Propagation of Boolean constants

The formula generated by the expansion step described in Section 3.4 can be extremely long for complex problem specifications or large instances. In order to reduce the size of this formula before it is passed on to later intermediate stages we use Boolean propagation and short-circuit evaluation.

| Law | Conjunctive form | Disjunctive form |
|---|---|---|
| Identity | $x \land \texttt{true} = x$ | $x \lor \texttt{false} = x$ |
| Annihilator | $x \land \texttt{false} = \texttt{false}$ | $x \lor \texttt{true} = \texttt{true}$ |

Table 3.4: Identity and annihilator laws of Boolean algebra.

There exist two types of sub-formulae that can be evaluated to the Boolean values `true` or `false`:

- Predicates over sets given in the input instance are evaluated to `true` if and only if the specific predicate exists in the input set, as described in Section 3.4.
- Equalities e.g. $term_1 = term_2$ are evaluated to `true` if and only if the two terms are equivalent.

### 3.5.1 Boolean propagation

In order to remove unnecessary sub-formulae, these Boolean values can then be propagated according to the laws of Boolean algebra given in Table 3.4. For example, $((1 = 2) \land \varphi)$ can be reduced to `false`, as $1 = 2$ does not hold. This Boolean value can then be propagated further up the AST to further reduce the size of the formula. While there are no Boolean constants in the example formula in Section 3.4.1 to be propagated, the effects of Boolean propagation can be seen in Appendix B.

### 3.5.2 Short-circuit evaluation

Short-circuit evaluation can be used to increase the speed of Boolean propagation by skipping the evaluation of a sub-formula when the result is already determined. The formula AST is recursed top-down: each node evaluating each of its children in turn. However in the case of conjunction or disjunction nodes, if the first operand is found to be equivalent to `true` or `false` then we can skip the evaluation of the second operand as the value of this node is already determined by the laws of Boolean algebra (given in Table 3.4).

With the combination of propagation of Boolean values and short-circuit evaluation we can now efficiently reduce the size of our expanded formulas involving Boolean values. This greatly reduces the time needed for the following conversions necessary to produce a formula in the output required by the SAT-solver.

## 3.6 Predicate substitution

Our formula now contains all the atomic predicates we wish to solve for instantiated over the instance values. We must now convert this formula into prepositional logic by replacing these atomic predicates with uniquely identifiable Boolean variables, which the SAT-solver will search for a satisfying assignment of. Integer identifiers are chosen for these so as to meet the requirements for representing variables in the DIMACS CNF format (see Section 2.3.2) that we will be outputting to the SAT-solver.

### 3.6.1 Reverse substitution

The predicates these variables represent cannot simply be discarded however. If we wish to return a satisfying assignment to the user then this should be given in terms of these original predicates: the assignment of the arbitrarily numbered Boolean variables returned by the SAT-solver is meaningless. To allow for this we must ensure that the mapping of predicates to Boolean variables is bijective (a one-to-one correspondence), so that they can be successfully substituted back. This mapping is then passed to the 'decode satisfying assignment' stage shown in Figure 1.1 for reverse substitution.

### 3.6.2 Location of predicate substitution stage

These substitutions could feasibly be made at any point in the reduction process between the problem specification and instance combination (Section 3.4) and SAT-solving (Section 3.8). This position is chosen as it is when the first-order formula is smallest: after it has been shortened by Boolean propagation, but before it is expanded into a (usually much larger) CNF expression. The whole AST can be traversed to make all substitutions necessary in the least amount of time.

## 3.7 Conversion to CNF

As most modern SAT-solvers take their input in conjunctive normal form (CNF) (described in Section 2.3.2), I need to convert my propositional logic expression into this form.

### 3.7.1 Conversion to NNF

We begin by first converting our expression to negation normal form (NNF), which satisfies the requirements of CNF that negations can only appear in front of variables and the only connectives are conjunctions ($\wedge$) and disjunctions ($\vee$). To do this we perform the following transformations[2]:

1. Eliminate bi-implications:
   $P \leftrightarrow Q \implies (P \to Q) \wedge (Q \to P)$
2. Eliminate implications:
   $P \to Q \implies \neg P \vee Q$
3. Move negation inwards using De Morgan's Laws:
   $\neg(P \wedge Q) \implies \neg P \vee \neg Q$
   $\neg(P \vee Q) \implies \neg P \wedge \neg Q$
4. Remove double negations:
   $\neg\neg P \implies P$

These rules are again applied recursively, converting nodes of the form of the left hand side to that of the right hand side, and eventually outputting an AST which is in NNF.

### 3.7.2 Distribution rule

To convert our NNF formula into CNF, we now require that all disjunctions are over literals only (to form clauses), and that all conjunctions are over clauses only. One method of doing this is to repeatedly apply the distribution rule:

$$P \wedge (Q \vee R) \implies (P \vee Q) \wedge (P \vee R)$$

However, it is possible that this will give rise to an exponential increase in the size of our formula. If an initial formula of $n$ clauses is in a format very unlike CNF, then this rule will need to be applied many times and will eventually result in an output formula containing $2^n$ clauses.

[2] Here $A \implies B$ means we replace each occurrence of $A$ with $B$, an operation easily expressible using pattern matching.

For example, using this distribution rule on the formula :

$$(a \wedge b) \vee (c \wedge d) \vee (e \wedge f) \vee (g \wedge h)$$

Results in the CNF formula:

$$\begin{gathered}(a \vee c \vee e \vee g) \wedge (a \vee c \vee e \vee h) \wedge (a \vee c \vee f \vee g) \wedge (a \vee c \vee f \vee h) \wedge \\ (a \vee d \vee e \vee g) \wedge (a \vee d \vee e \vee h) \wedge (a \vee d \vee f \vee g) \wedge (a \vee d \vee f \vee h) \wedge \\ (b \vee c \vee e \vee g) \wedge (b \vee c \vee e \vee h) \wedge (b \vee c \vee f \vee g) \wedge (b \vee c \vee f \vee h) \wedge \\ (b \vee d \vee e \vee g) \wedge (b \vee d \vee e \vee h) \wedge (b \vee d \vee f \vee g) \wedge (b \vee d \vee f \vee h)\end{gathered}$$

For larger $n$, the results of this method can quickly become unmanageable. Extremely large CNF expressions may be produced which will take very long for a SAT-solver to solve.

### 3.7.3 Tseitin transformation

This exponential increase when converting to CNF can be avoided if we allow the conversion to only preserve satisfiability, rather than equivalence. The Tseitin transformation [34] is an alternative method to the distribution rule which produces a CNF formula whose length is *linear* in the size of the input.

It does this by introducing a new variable for each logical operator in the formula. A small CNF sub-expression that constrains this new variable to be equivalent to the operator's result is then produced. These sub-formulae are given in Table 3.5. When these expressions are generated for each logical operator in the input formula, they will together describe our original formula. As they are each in CNF, they can all be conjuncted together to give our final CNF formula. Due to the fixed length of each sub-expression produced, the length of this output is bounded by the length of the input multiplied by a constant factor.

| Logical operator | Introduced variable | CNF sub-expression produced |
|---|---|---|
| $A \wedge B = C$ | $C$ | $(\neg A \vee \neg B \vee C) \wedge (A \vee \neg C) \wedge (B \vee \neg C)$ |
| $A \vee B = C$ | $C$ | $(A \vee B \vee \neg C) \wedge (\neg A \vee C) \wedge (\neg B \vee C)$ |
| $\neg A = B$ | $B$ | $(\neg A \vee \neg B) \wedge (A \vee B)$ |

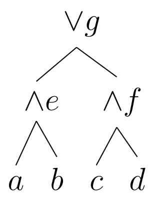
Table 3.5: CNF sub-expressions produced for various logical operators by the Tseitin transformation.

### Equivalence and satisfiability of Tseitin transformation

The introduction of new variables for each logical operator in the Tseitsin transformation means that the resulting CNF formula is no longer propositionally equivalent to the input formula. However the two are *equisatisfiable*, meaning that the output CNF formula is satisfiable if and only if the input formula is as well. If a satisfying assignment of the variables in this output formula is found, the assignments to all newly introduced variables can simply be discarded. This is because they are completely determined by the values of the input variables due to the constraints from Table 3.5 that we introduced. The remaining assignment will then be the required satisfying assignment to the original formula.

### Example of translating a formula to CNF using the Tseitin Transformation

An example of a formula that gives an exponential increase using the distribution conversion is $(a \wedge b) \vee (c \wedge d) =$



Where the variables $\{e, f, g\}$ are the new variables we have introduced to represent the result of each logical operator. For each operator we also generate certain clauses as described by Table 3.5, giving Table 3.6.

| Logical operator | Introduced variable | CNF sub-expression produced |
|---|---|---|
| $a \wedge b = e$ | $e$ | $(\neg a \vee \neg b \vee e) \wedge (a \vee \neg e) \wedge (b \vee \neg e)$ |
| $c \wedge d = f$ | $f$ | $(\neg c \vee \neg d \vee f) \wedge (c \vee \neg f) \wedge (d \vee \neg f)$ |
| $e \vee f = g$ | $g$ | $(e \vee f \vee \neg g) \wedge (\neg e \vee g) \wedge (\neg f \vee g)$ |

Table 3.6: CNF sub-expressions produced for our example formula by the Tseitin transformation.

Our final CNF expression is then a conjunction of the clauses in the right hand column of Table 3.6, along with the output operator $g$ (as this must be `true` to satisfy the expression):

$$\begin{aligned}&(\neg a \vee \neg b \vee e) \wedge (a \vee \neg e) \wedge (b \vee \neg e)\ \wedge \\ &(\neg c \vee \neg d \vee f) \wedge (c \vee \neg f) \wedge (d \vee \neg f)\ \wedge \\ &(e \vee f \vee \neg g) \wedge (\neg e \vee g) \wedge (\neg f \vee g) \wedge g\end{aligned}$$

One satisfying assignment of this is $a, \neg b, c, d, \neg e, f, g$. If we discard the introduced variables we are left with $a, \neg b, c, d$, which is a satisfying assignment to our original formula.

### 3.7.4 Comparison of distribution and Tseitin methods for CNF conversion

Whilst the Tseitin transformation does limit the *worst case* size increase to be linear in the size of the original expression, there is a reasonably high constant factor involved. The Tseitin transformation always produces three CNF clauses and introduces a new variable for every logical connective in the input formula, of which there may be very many. Compare this to the results of the distribution method: although this method can produce an exponential size increase, if it is given a formula already in CNF it will simply output the original expression, with zero increase in size. Which of these method gives the shortest result is a trade-off between the size of the input formula and the similarity of the input formula to CNF.

Different SAT-solvers may also be more or less efficient at solving the possibly larger formulae with fewer variables produced by the distribution method, compared with the shorter formulae with many more variables produced by the Tseitsin transformation. We have already seen in Section 2.2.2 how the time taken to solve a given SAT formula is more dependent on how constrained an instance is rather than the size of the formula. The implications of these different formats of SAT instances are discussed in Section 4.4.3.

## 3.8 SAT-solving

Once we have translated our formula into CNF, it is in the format required to be solved by a SAT-solver. More precisely the output file we produce should be in the DIMACS CNF standard described in Section 2.3.2. This is commonly used by all major SAT-solvers, so we are free to use any of the many off-the-shelf SAT-solvers available.

A 3rd party SAT-solver is used rather than my own implementation due to the wealth of research that has been put into developing these, as outlined in Section 2.3.1. Current SAT-solvers are now extremely sophisticated and incorporate many complex techniques. Implementing a SAT-solver with performance comparable to these systems would be infeasible. Fortunately this is not necessary as most of these are freely available as open-source software.

## 3.9 Returning a satisfying assignment

Once the SAT-solver has found an answer to our SAT problem it will output it in the format described in Section 2.3.2. We then wish to convert this back to the original domain to provide the answer to the original problem. This makes up the 'decode' stage of Figure 1.1.

As DIMACS CNF requires that all variables are described as an integer, the output from the SAT solver is described as such too. For example, the output to the 3-COL problem from Section 3.4.1 is the following satisfying assignment:

```
SAT
1 -2 -3 -4 -5 6 -7 8 -9 10 -11 -12 -13 14 -15 0
```

To get a meaningful solution to our problem, i.e. an assignment of which nodes should be coloured with which colour, we need to translate these variables back into the predicates they replaced. This is done using the bijective mapping of these made during the substitution, as described in Section 3.6.1. Any new variables introduced by the Tseitin transformation do not correspond to predicates, and so can simply be discarded[3].

[3] In this example the Tseitin transformation has not been used, to simplify the output.

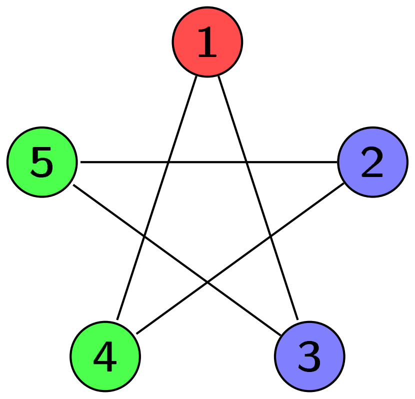


Figure 3.1: Our discovered 3-Col solution.

Doing this for our example gives:

```
SAT
B(3) -B(1) -G(3) -G(1) -R(3) R(1) -B(5) G(5) -R(5) B(2) -G(2)
-R(2) -B(4) G(4) -R(4)
```

If we then discard the predicates which are assigned to `false` and sort the results, we get the desired solution to our example problem:

```
R(1) B(2) B(3) G(4) G(5)
```

This solution is shown in Figure 3.1, and can clearly be seen to be a correct solution to 3-Col on the graph from Section 3.2.3.

## 3.10 Expressing set cardinality constraints

Not all problems can be described in ∃SO as easily as 3-Col. Complex constraints not only make problems more difficult to read and write, but they can also greatly increase the time it takes for *Twist* to solve them. One example of a constraint that frequently occurs in many common NP-complete problems, but is difficult to express in ∃SO, is set cardinality. In this section I describe an extension to *Twist*'s core functionality that allows such constraints to be easily specified and efficiently solved.

An example problem containing such a constraint is the Clique problem. This asks whether or not some graph $G(V, E)$ has a *clique* of size $k$: some set of nodes

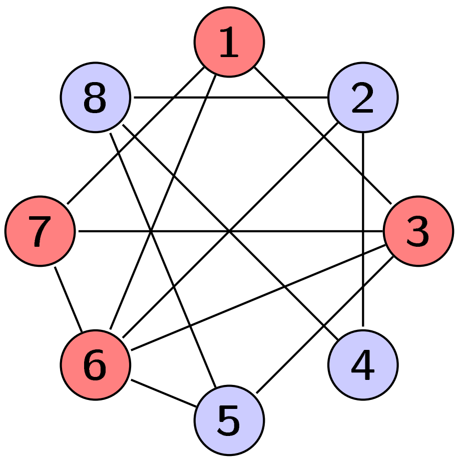


Figure 3.2: A clique of size 4.

$C \subseteq V$ of at least size $k$ ($|C| \geq K$), such that $\forall x, y.C(x) \wedge C(y) \Rightarrow E(x, y)$, i.e. every node of the clique is joined by an edge to every other node of the clique. An example of a graph with a clique of size 4 (shown in red) is given in Figure 3.2.

However, expressing the cardinality constraint $|C| \geq k$ cannot be done easily in second-order logic. To achieve this we must specify that the input to our problem is the graph $G$ plus some set $U$, where $|U| = k$. We then test that $|C| = |U|$ by asking if there exists a binary relation $F$ that is a one-to-one correspondence between $C$ and $U$. This can be done in first-order logic, giving our $\exists$SO description of the Clique problem as:

$$
\begin{array}{l}
\exists C\ \exists F\ (\\
\qquad \forall x, y \in V(C(x) \wedge C(y) \to E(xy)) \quad \}\text{Nodes in the clique are connected}\\
\quad \wedge \forall x \in V\ \forall y \in U(F(x, y) \to C(x)) \quad \}F \text{ relates clique nodes to } U\\
\quad \left.\begin{array}{l} \wedge \forall x \in V(C(x) \to \exists ! y \in U(F(x, y)))\text{[4]}\\ \wedge \forall y \in U(\exists ! x \in C(F(x, y) \wedge C(x))) \end{array}\right\} \begin{array}{l} F \text{ is a one-to-one correspondence}\\ \text{between } C \text{ and } U \end{array}\\
)
\end{array}
$$

From this we can see that our final description of the Clique problem will be quite large. Not only is it difficult for users to read and understand, but the multiple nested quantifiers will mean that expanding the formula out with the instance values as described in Section 3.4 will produce an extremely large expression. This severely limits the size of instances we can expect to solve in a

[4] $\exists ! x(\varphi(x))$ means "there exists *exactly one* $x$ such that $\varphi(x)$". This can be expressed in pure first-order logic as $\exists x(\varphi(x) \wedge \forall y(\varphi(y) \to (x = y)))$

reasonable amount of time.

### 3.10.1 Pseudo-Boolean constraints

As our formalisms of second-order and propositional logic are strict, this is the best we can do using the tools provided so far. However, this lengthy description of set cardinality can be eliminated by extending our use of propositional logic to *pseudo-Boolean constraints.*

If we define a variable $x$ of value `true` or `false` to instead have integer value 1 or 0 respectively, then a SAT clause $x \vee y \vee z$ can be written in the form $x+y+z \geq 1$. One or more of the variables must be `true` $(= 1)$ for this inequality to be satisfied, as is our requirement of a clause. A negated variable $\neg x$ is then equivalent to $(1 - x)$.

With pseudo-Boolean constraints, we write clauses in this way but no longer constrain the right-hand side of the inequality to be 1. This then allows us to specify how many variables in a clause must be `true`.

### 3.10.2 Expressing SAT instances containing cardinality constraints with pseudo-Boolean constraints

All ordinary CNF clauses can simply be expressed as described above, with a value of 1 on the right side of the inequality. The benefit of using pseudo-Boolean constraints is that we can use them to easily express a set cardinality constraint of the form $|C| \geq k$, as $\sum_{c \in C} c \geq k$. For example, with the set of vertices $\{1, 2, 3, 4, 5, 6, 7, 8\}$ from Figure 3.2 the constraint "the clique is of size $\geq 4$" becomes $C(1)+C(2)+C(3)+C(4)+C(5)+C(6)+C(7)+C(8) \geq 4$.

By extending *Twist* to allow such constraints to be encoded in this way we can now express cardinality constraints without the additional input set $U$ or the bijection $F$. Our modified ∃SO specification for CLIQUE is now as follows:

$$
\begin{array}{ll}
\exists C\ ( & \\
\quad \forall x, y \in V(C(x) \wedge C(y) \rightarrow E(xy)) & \}\text{Nodes in the clique are connected} \\
\quad \wedge\ |C| \geq k & \}C \text{ is of size } k \\
) &
\end{array}
$$

The description of this problem in *Twist*'s specification language is given in Appendix C.1.

### 3.10.3 Solving pseudo-Boolean constraints

As pseudo-Boolean constraints have greater expressive power than basic CNF clauses, we cannot use an ordinary SAT-solver to solve them. Instead, we can either use a special pseudo-Boolean solver, or the pseudo-Boolean constraints can be converted into propositional SAT-clauses and solved by a standard SAT-solver. Recent work has found conversion methods allowing standard SAT-solvers to perform on a par with even the best native pseudo-Boolean solvers, particularly in cases where most of the constraints are standard SAT clauses, as is ours [14]. This means the extension to use pseudo-Boolean constraints has not sacrificed *Twist*'s extensibility requirement, as discussed in Section 4.4.2.

As a result, our much more efficient descriptions of set cardinality constraints greatly improve the performance with which *Twist* can solve the many NP-complete problems containing them. Evidence of this is given in Section 4.3.5.

# Chapter 4

# Evaluation

In this chapter *Twist* is evaluated against each of the four project goals identified in Section 1.4, giving both theoretical and empirical evidence.

## 4.1 Completeness

> **It should be possible to input *any* NP-complete problem into *Twist* to be solved.**

### 4.1.1 Theoretical proof

The completeness of *Twist* follows from the use of existential second-order logic ($\exists$SO) for the specification language and Fagin's theorem [15]. Fagin's theorem proves that any NP-complete problem can be expressed by some $\exists$SO formula, and hence written in our specification language and input into *Twist*.

Whilst the specification language does involve various deviations from pure $\exists$SO (described in Section 3.1) these strictly do not subtract from the expressive power of the language, meaning this property is not invalidated. All further stages are completely generalised, allowing them to operate on any problem. Whether or not this means they can then be guaranteed to be solved correctly is discussed in Section 4.2: Correctness below.

### 4.1.2 Finding ∃SO descriptions of NP-complete problems

Evidence is now given for a more practical notion of completeness: That we are able to input descriptions of NP-complete problems that are not only theoretically correct, but also succinct enough to be easily written, read, and solved by *Twist*.

Unfortunately this does not follow directly from our theoretical proof of correctness. The conversion from arbitrary NP problem to ∃SO formula given in the proof of Fagin's theorem in Section 2.1.4 requires us to create a formula that completely describes a non-deterministic Turing machine and its entire execution trace. It is extremely impractical to give an ∃SO formula for some problem in this way.

Formulating an ∃SO description of an NP-complete problem such that is both understandable and correct is non-trivial. To combat this my specification language was designed with readability and writeability in mind, including features such as input and output declarations and comments (see Section 3.1). In Section 3.10 we also see how *Twist* can be extended to allow for much clearer descriptions of set cardinality constraints, which appear in many NP-complete problems but which are difficult to describe concisely in ∃SO.

As a result of these changes and the careful construction of problem descriptions, many common NP-complete problems can now be expressed succinctly in my specification language and solved by *Twist*. A selection of these are listed in Appendix C. The completeness of *Twist* to solve such a wide range of problems now means that the benefits outlined in Sections 4.3: Performance and 4.4: Extensibility can be applied to *any* NP-complete problem.

## 4.2 Correctness

**The solution returned by *Twist* should always be exact.**

### 4.2.1 Proof sketch

*Twist*'s correctness is given by Cook's theorem [9], which proves that any NP-complete problem can be reduced into an *equivalent* instance of BOOLEAN SATISFIABILITY. This preservation of equivalence is followed by each of *Twist*'s reduction stages:

As the sets being quantified over are strictly finite the expansions described in Section 3.4 result in a combined formula equivalent to our problem specification and instance inputs [26]. Boolean propagation and predicate substitution do not invalidate this. Similarly conversion to NNF and to CNF by the distribution rule both preserve equivalence of the formula. The Tseitin transformation does not preserve equivalence, however it does preserve satisfiability (see Section 3.7.3), meaning any solution generated from the resulting formula will still be correct. It cannot be proven that 3rd party SAT-solvers are correct, however we make use of exact SAT-solvers which areguaranteed to terminate. Conversion of the satisfying assignment back into the original domain is trivially correct due to the use of a bijective mapping, and as a result all solutions produced by *Twist* should be correct.

### 4.2.2 Testing for correctness

Whilst the proof sketch given above shows that the conversion and solving of problems should be correct, this relies on the implementation following this correctly, which is often not the case. As proof of program correctness for a system of this size is quite complex, empirical evidence is gathered instead.

A suite of regression tests are used to ensure that whenever any changes are made to my code base, correct results are still produced. This tests both standard and edge cases for a number of different problems, with the aim of detecting any possible errors introduced by changes to my code base. It was very useful during the development of *Twist* to both be able to discover errors quickly, and also be reassured that any changes made were correct.

Such a test suite obviously cannot be complete, and so correctness was also tested during all performance comparison tests in Section 4.3 by comparing the results returned by *Twist* and the algorithm being tested against. This provides further empirical evidence that no cases were missed by my regression test suite. Whilst it is not determined that the comparison algorithms are correct, the chance of multiple different systems simultaneously giving the wrong answer drops exponentially, meaning that we can say with a very high degree of certainty that *Twist* is correct.

## 4.3 Performance

***Twist* should be able to solve problems at a speed competitive with current techniques.**

In this section I show how *Twist* outperforms traditional approaches when solving the most difficult instances of NP-complete problems. By analysing performance we also see surprising evidence for instance difficulty being preserved through translation into SAT, indicating some unforeseen uses for *Twist*. Finally I demonstrate the performance benefits offered by the extension to my core system described in Section 3.10.

### 4.3.1 Experimental setup

To compare the performance of *Twist* against a native algorithm for an NP-complete problem I use OCamlGraph's graph colouring functor [8]. This is a good opponent for *Twist* as it is a standard heuristic based algorithm, commonly used by systems that require graph colouring to be done with reasonable efficiency, but which do not wish to devote the large amount of resources required to implement a more sophisticated solution. It uses a backtracking breadth-first traversal approach to assigning new colours, and uses Kempe's linear-time simplification [25] as an additional optimising heuristic.

These comparisons were made on 2,000 uniformly generated random graphs. On each graph both the OCamlGraph algorithm and *Twist* (using the specification for COLOURABILITY given in Section 2.1.2) were run on a four-core 2.50 GHz Intel Core i5-2450M CPU. For each size of graph each algorithm was repeated 20 times and the mean running time was taken. Performance is plotted against a graph's average connectivity[1].

### 4.3.2 Experimental results

The results of comparing the two systems on 3-COL on 50 vertex graphs are shown in Figure 4.1. It can be seen that in most cases, the native graph colouring algorithm is faster than *Twist*. This is due to the overheads from *Twist*'s several translation stages, necessary for completeness: SAT-solving typically accounts

[1] As discussed in Section 2.2.2, the average connectivity of graph $G = (V, E)$, equal to $|E|/|V|$, is known to be a good measure of how constrained a graph problem is [6], and hence its difficulty.

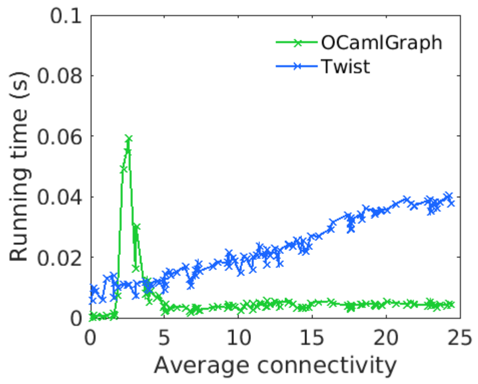


Figure 4.1: Performance comparison of *Twist* and OCamlGraph for 3-colouring graphs with 50 vertices.

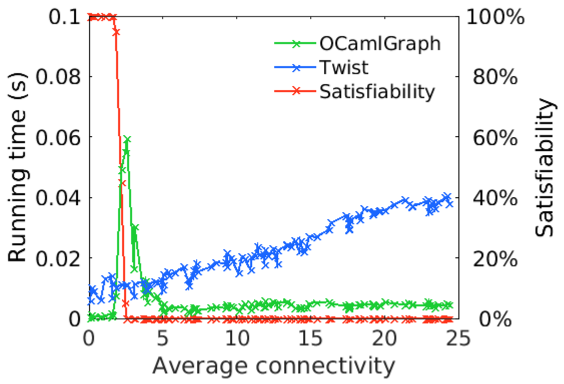
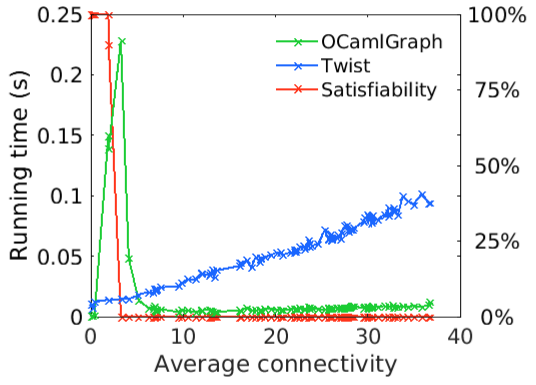


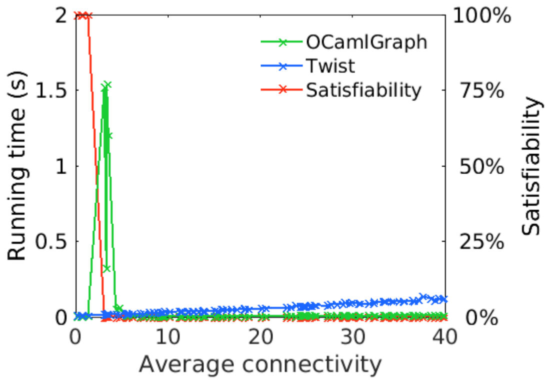
Figure 4.2: Satisfiability of 3-Col on graphs tested in Figure 4.1.

for less than half of *Twist*'s running time here. However for graphs with a certain structure, namely those with an average connectivity close to the number of colours being used, the native algorithm's performance drops considerably whilst the performance of *Twist* does not. By additionally analysing the average satisfiability (i.e. what proportion of graphs of a certain connectivity are 3-colourable) in Figure 4.2 we see a possible reason for this behaviour. The spike in running time occurs almost exactly at the phase transition between colourable and non-colourable graphs (see Section 2.2.2).

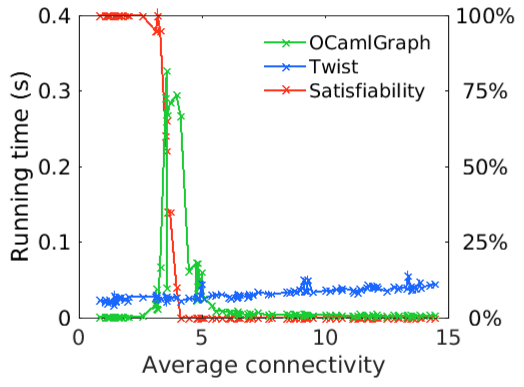
When we look at more ambitious problems in Figure 4.3 such 3-Col on larger instance graphs or 4-Col this effect becomes even stronger, with OCamlGraph giving an even larger spike in comparison to *Twist*'s performance occurring exactly at this phase transition threshold. These spikes in running time occurring at a problem's phase transition supports claims in the literature that it is the NP-complete problem instances that lie close to this threshold that are the most 'difficult', and are most accountable for their exponential worst-case running time [6]. Below the threshold connectivity value of $\sim k$ nearly all graphs are $k$-colourable, and so a solution is easy to find. Above the threshold nearly all graphs are not $k$-colourable, and so it is very easy to trim the search tree of possible solutions. But at the threshold connectivity there is a low but non-negligible probability of a solution, meaning that there are many well-separated almost-solutions (local minima). These cause heuristic search algorithms to backtrack a lot before a solution is found or all solutions can be refuted, resulting in the large spike in running time.



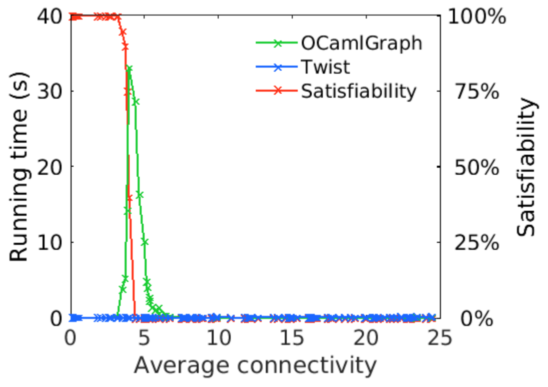
(a) 3-Col on graphs of 75 vertices.



(b) 3-Col on graphs of 85 vertices.



(c) 4-Col on graphs of 30 vertices.



(d) 4-Col on graphs of 50 vertices.

Figure 4.3: Performance comparison of *Twist* and OCamlGraph for various Graph Colourability problems.

A reasoning for why *Twist* performs better close to these thresholds when compared with the native algorithm is that the advanced heuristics used by the SAT-solvers *Twist* employs are much better at dealing with these difficult problems. The well researched SAT-solving techniques outlined in Section 2.3.1 are much better equipped to efficiently solve problems that are neither under nor over constrained. It is at this threshold where the benefits of this can be seen, as the advantages of being able to use well a developed SAT-solver outweigh the overheads needed to convert a problem into a SAT instance.

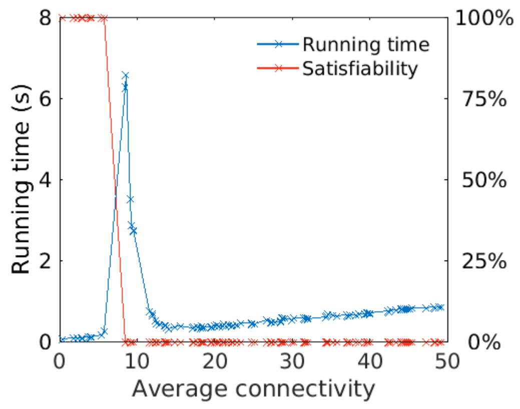
The benefit of *Twist* reliably (with low variance) having a low running time is that it can be used to solve much larger instances with little risk of the algorithm not finding an exact solution before some given time limit is reached. With native algorithms we do not have this guarantee, as they are stumped by the difficult instances occurring at a phase transition where their less sophisticated heuristics are not sufficient to find an answer quickly.

### 4.3.3 A closer look at phase transitions

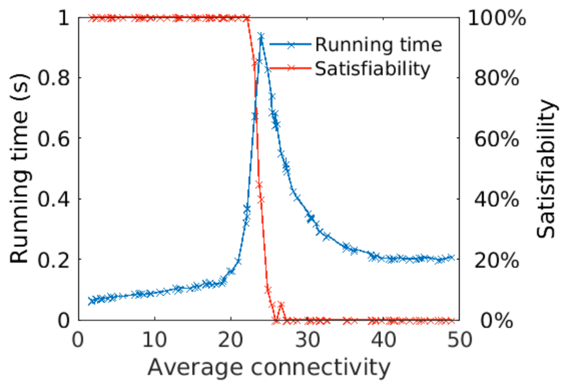
One immediate question this gives rise to is whether or not our approach of reducing problems into BOOLEAN SATISFIABILITY has eliminated the occurrence of difficult instances at the phase transition. However, as we increase the difficulty of problems analysed further we can see in Figure 4.4 that *Twist* also produces performance spikes around the phase transition thresholds[2]. While the conversion costs remain roughly proportional to instance size, at these spikes SAT-solving now accounts for the vast majority of the running time. The use of advanced SAT-solvers has not eliminated the occurrence of such a spike, but simply managed to suppress it in the instances shown in Figures 4.2 and 4.3 due to the SAT-solvers' superior efficiency for difficult problems. This illustrates why the extensibility criteria of my project is important as the introduction of new, improved SAT-solvers may suppress the emergence of these performance spikes even further.

The prevalence of these spikes has some important implications for complexity theory: it implies that the difficulty of an NP-complete problem persists through its reduction to SAT. The difficulty of, for example, a CLIQUE problem is the same as the difficulty of the SAT instance generated by reducing this problem

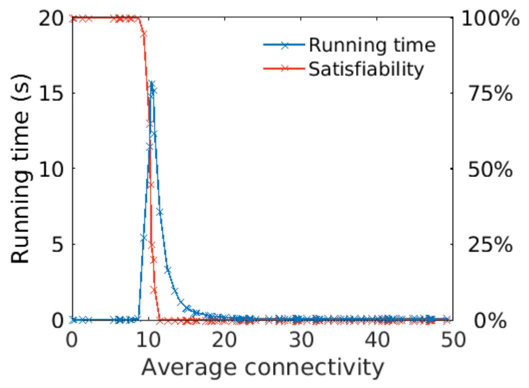
[2] Here we are no longer comparing against native algorithms as they take much too long to solve, due to their much larger performance spikes.



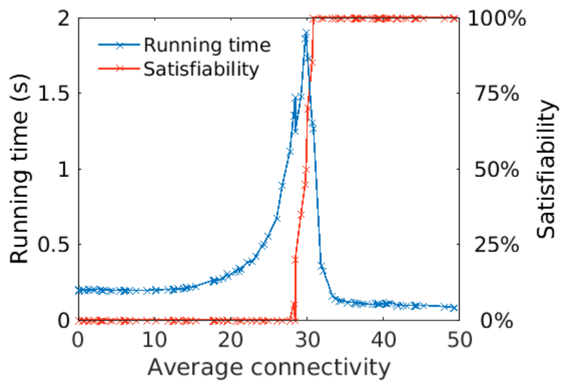
(a) 5-Col on graphs of 100 vertices.



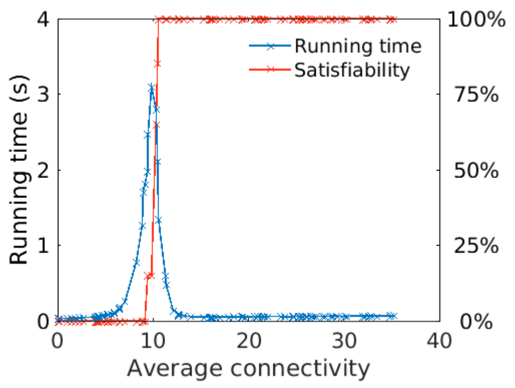
(b) Independent Set with $K = 10$ on graphs of 100 vertices.



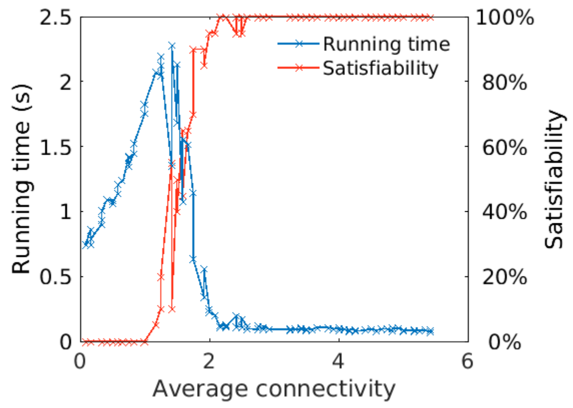
(c) Vertex Cover with $K = 80$ on graphs of 100 vertices.



(d) Clique with $K = 12$ on graphs of 100 vertices.



(e) Dominating Set with $K = 5$ on graphs of 75 vertices.



(f) Hamiltonian Path on graphs of 12 vertices.

Figure 4.4: Running time of *Twist* on various NP-complete problems (given in Appendix C), showing performance spikes near their phase transitions.

into a CNF formula. This means it will not be possible to effectively make all instances of some NP-complete problem $\mathcal{P}$ easy to solve simply by reducing difficult instances of $\mathcal{P}$ into easier equivalent instances of some other problem $\mathcal{Q}$. However it also means that systems such as *Twist* can provide a *reliable* performance improvement to solving NP-complete problems. If the initial problem was easy to solve (under or over constrained), then this preservation of difficulty means the SAT instance generated will be guaranteed to also be easy to solve, leaving only the limited conversion overhead costs. If the initial problem was difficult to solve (it exists at a phase transition threshold) then so will the produced SAT instance, however here we can then exploit the advanced SAT-solving techniques to solve this as efficiently as currently technology allows. This demonstrates that *Twist* consistently provides an economical alternative to native solving implementations.

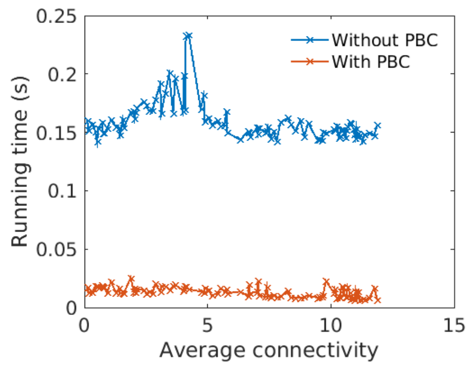
### 4.3.4 Additional uses of *Twist*

This connection between phase transitions and solving time shown by performance analysis of *Twist* suggests some additional, unforeseen uses for my project. One is that it provides a way to generate difficult instances of SAT problems. This is a non trivial problem. Difficult SAT instances are very valuable for use in SAT-solver competitions, so that solvers' performance can be compared on these crucial difficult problems, rather than trivial instances.

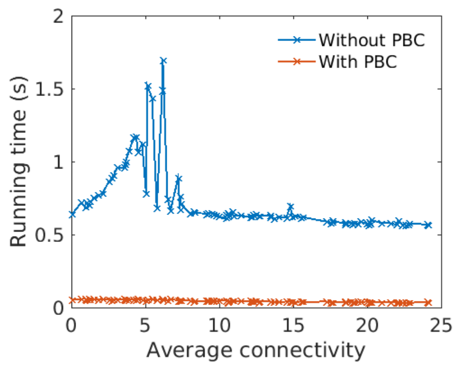
A second new use, facilitated also by the completeness criteria (see Section 4.1), is that *Twist* can be used to find the location of these phase transition thresholds in a range of different problems, as in Figure 4.4. These phase transition diagrams allow the behaviour of complex systems to be characterised at a macroscopic level. They allow us to predict whether or not a given instance will be difficult to solve, and this can be used to estimate its running time, or perhaps to determine the best approach to solve an instance based on its difficulty.

### 4.3.5 Performance benefits of using pseudo-Boolean constraints

Bottlenecks were identified in attempting to solve problems which involve constraints not easily expressible in $\exists$SO, such as the set cardinality constraints described in Section 3.10. To remedy these, *Twist* was extended to accommodate pseudo-Boolean constraints.



(a) Graphs of 25 vertices.



(b) Graphs of 50 vertices.

Figure 4.5: Comparison of *Twist* solving the CLIQUE problem with $K = 5$, both and without the pseudo-Boolean constraint (PBC) extension.

Whilst these solvers are more complex and hence slower than ordinary SAT-solvers, their increased expressive power allows much more efficient solving of these complex constraints. This results in a net gain in performance, seen in Figure 4.5. The pseudo-Boolean solver used here is MiniSat+ [11], an extension to the MiniSat SAT-solver. It is likely that further similar extensions to allow more efficient expression and solving of other common problem constraints are possible, but this is left as future work.

## 4.4 Extensibility

**The use of *any* 3rd-party SAT-solver should be allowed to solve the SAT instances.**

### 4.4.1 File format standards

*Twist*'s extensibility is given by its use of the commonly used DIMACS CNF standards for SAT-solver input and output. This describes both the form of the input formulae (CNF), and the syntax of this required by the file format formats (see Section 2.3.2). These standards are used by the annual SAT-solver competitions, and hence all modern SAT-solvers developed already conform to these standards and so can be used with *Twist*. All that is required to use *Twist* with a certain SAT-solver is for its execution path to be supplied as a command line argument.

### 4.4.2 Extensibility implications of using pseudo-Boolean solvers

Pseudo-Boolean solvers, introduced in Section 3.10.1, also follow input and output standards used in pseudo-Boolean competitions [28]. However as these are a more special case there is less research interest into developing pseudo-Boolean tools, and as a result currently available solvers do not always provide use of the cutting edge advances from current SAT-solver development.

However, the use of pseudo-Boolean constraints does not harm the extensibility of *Twist* due to the existence of tools that efficiently convert pseudo-Boolean constraints into ordinary CNF SAT clauses [14]. Any regular SAT-solver can then be used to solve these, meaning *Twist* maintains its extensibility. Evidence that the performance costs in this additional translation between constraint types is outweighed by the benefits of the pseudo-Boolean's greater expressibility for set cardinality constraints is given in Figure 4.5.

### 4.4.3 Comparison of different SAT-solvers

The need for extensibility is now illustrated by a performance comparison of a selection of different SAT-solvers running on a number of difficult 3-COLOURABILITY problems. The results are given in Table 4.1. The problems are difficult benchmark instances for the ASSAT answer-set programming system [27]. The solvers selected are:

- MiniSat version 2.2.0 [12].
- Lingeling version avy-86bf266-140429 [4], winner of the 2014 gold medal[3] in the Application SAT+UNSAT category.
- glueSplit_clasp version 1.0 [7], winner of the 2014 gold medal in the Hard combinatorial SAT+UNSAT category.
- dimetheus version 2.100.994 [16], winner of the 2014 gold medal in the Random category.

Table 4.1 lists properties of the input graph instance, the SAT instance produced, the solution to the decision problem, and mean timings for each of the above solvers. The solver with the best time for each instance is written in bold, and any times exceeding the predefined limit of 60 seconds are marked with '-'.

[3] Medals listed are from the 2014 international SAT Competition [2].

| Graph vertices | Graph edges | # Variables generated | # Clauses generated | SAT-solver running time (s) | | | | Soln. |
|---|---|---|---|---|---|---|---|---|
| | | | | MiniSat | Lingeling | glueSplit | Dimetheus | |
| 100 | 570 | 300 | 7,260 | **0.028** | 0.075 | 0.039 | 0.045 | no |
| 300 | 1,760 | 900 | 22,180 | **0.082** | 0.172 | 0.090 | 0.143 | no |
| 600 | 3,554 | 1,800 | 44,632 | **0.178** | 0.346 | 0.203 | 0.310 | no |
| 1,000 | 5,950 | 3,000 | 74,600 | **0.276** | 0.551 | 0.310 | 0.519 | no |
| 3,000 | 17,956 | 9,000 | 224,648 | **0.840** | 1.647 | 1.075 | 1.723 | no |
| 6,000 | 35,946 | 18,000 | 449,568 | **2.015** | 3.050 | 2.255 | 3.611 | no |
| 10,000 | 10,000 | 30,000 | 350,000 | 3.000 | **1.767** | 2.276 | 3.261 | yes |
| 10,000 | 11,000 | 30,000 | 357,992 | 3.277 | **2.023** | 2.321 | 3.418 | yes |
| 10,000 | 12,000 | 30,000 | 366,000 | 3.141 | **2.150** | 2.361 | 3.994 | yes |
| 10,000 | 13,000 | 30,000 | 374,000 | 3.228 | **2.445** | 2.465 | 5.673 | yes |
| 10,000 | 14,000 | 30,000 | 381,992 | 3.352 | 2.754 | **2.604** | 3.362 | yes |
| 10,000 | 15,000 | 30,000 | 389,992 | 3.491 | 3.081 | **2.729** | 3.462 | yes |
| 10,000 | 16,000 | 30,000 | 397,992 | 3.528 | 3.433 | **2.905** | 3.559 | yes |
| 10,000 | 17,000 | 30,000 | 405,984 | 3.699 | 3.224 | **3.054** | 3.691 | yes |
| 10,000 | 18,000 | 30,000 | 413,992 | 3.932 | **3.167** | 4.106 | 5.112 | yes |
| 10,000 | 19,000 | 30,000 | 421,984 | 4.907 | **3.350** | 9.638 | 6.355 | yes |
| 10,000 | 20,000 | 30,000 | 430,000 | 12.410 | **3.377** | 19.398 | - | yes |
| 10,000 | 21,000 | 30,000 | 437,992 | **1.838** | 2.667 | 4.153 | 2.884 | no |
| 10,000 | 22,000 | 30,000 | 445,984 | - | - | - | - | - |
| 10,000 | 23,000 | 30,000 | 453,976 | - | - | - | - | - |
| 10,000 | 24,000 | 30,000 | 461,968 | - | - | - | - | - |
| 10,000 | 25,000 | 30,000 | 469,992 | - | - | - | - | - |

Table 4.1: Performance comparison of different SAT-solvers on ASSAT benchmark 3-COL problems.

As can be seen from Table 4.1, no particular current SAT-solver outperforms all other solvers in all instances, even for one fixed problem. Which solver performs best is predicted to be even more diverse were we to compare different problem types, as they each generate very different SAT instances. Further evidence for this is given in SAT competition results [2]. In these competitions there are several different categories of instances to solve, each with different CNF characteristics (such as size, clause to variable ratio), origins, or goals (SAT / UNSAT) [3]. No one SAT-solver achieved the gold award in more than one of these categories in 2014. This illustrates why extensibility is required, and shows how the best SAT solver to use depends greatly on the format of SAT instances it will be solving.

As the CNF instances produced by *Twist* are fairly unique, SAT solving performance could perhaps be improved by more in-depth analysis of how best to configure the many parameters often provided by SAT-solvers to achieve the best performance with *Twist*. Lingeling alone has over 300 different parameters that control how the solver searches for solutions [4]. It has been found that significant performance improvements can be made just with more analysis into how features of state of the art solvers can be combined and their settings tweaked [29, 24]. It is likely that this kind of research could benefit *Twist* by developing a SAT-solver that uses variants of the advanced techniques described in Section 2.3.1 honed to solve SAT instances specifically of the type generated by *Twist* as well as possible. However this itself is an extremely wide area of research, and so is left as future work.

# Chapter 5

# Conclusion

This project has succeeded in demonstrating that reduction to BOOLEAN SATISFIABILITY and the use of SAT-solvers does indeed provide a viable way of solving NP-complete problems. Each of the original project goals identified in Section 1.4 has been achieved:

1. Fagin's theorem and the use of ∃SO in my problem specification language allow for any problem to be input into *Twist*.

2. The preservation of satisfiability during problem reduction along with extensive test evidence has shown that the produced solutions to these problems are correct.

3. Although the overheads involved in reducing problems to SAT mean that most trivial instances take longer to solve on *Twist* than on a native algorithm, performance improvements are found at the most difficult instances that account for the majority of a problem's running costs. Here the well-researched heuristic techniques employed by modern SAT-solvers provide a considerable advantage, and solve instances much more efficiently than a native algorithm.

4. The extensibility of *Twist* to use any supplied SAT-solver is provided by the use of standardised interfaces, and means that these benefits are able to increase further as developments in SAT-solving continue to advance.

An extension to the core project has been made, detailed in Section 3.10, which demonstrates the improvements to both performance and usability possible for solving more complex problems that are difficult to express in ∃SO. Surprising results have also been found, such as further evidence that the difficulty of NP-complete problems is preserved under reductions between problems. This has lead to the discovery of additional uses for *Twist*, including generating difficult SAT instances and phase transition diagrams that can demonstrate the distribution of difficult instances of a problem. Overall this project has been very enjoyable, and I have learned much about the connection between theoretical results in complexity theory and their practical uses.

## 5.1 Further work

A number of opportunities for further work were identified during the analysis of *Twist* in Chapter 4:

- In Section 4.3 empirical evidence for the benefits of *Twist*'s approach compared with traditional algorithms is given. This is consistent with observations made in the literature [6, 31], however further work is needed in comparing *Twist* against more algorithms across a wider range of NP-complete problems.

- The vast performance improvements gained by the introduction of pseudo-Boolean constraints demonstrated in Section 4.3.5 indicates that further research into more efficient methods of describing and solving other complex constraints commonly found in NP-complete problems may well lead to similar performance gains.

- The comparison of different solvers in Section 4.4.3 illustrated the possible benefits of further analysis into how SAT-solver techniques can be combined and their parameters configured. SAT-solvers optimised to solve the types of SAT instances generated by *Twist* could provide substantial performance improvements.

# Bibliography

[1] Adrian Balint, Anton Belov, Matti Järvisalo, and Carsten Sinz. SAT Challenge 2012. `baldur.iti.kit.edu/SAT-Challenge-2012`.

[2] A Belov, D Diepold, M J H Heule, and M Järvisalo. The international SAT Competitions web page. `www.satcompetition.org`.

[3] A Belov, D Diepold, M J H Heule, and M Järvisalo. Solver and benchmark descriptions. In *Proceedings of SAT Competition 2014: Solver and Benchmark Descriptions*, volume B-2014-2, 2014.

[4] Armin Biere. Yet another local search solver and Lingeling and friends entering the SAT Competition 2014. In *Proceedings of SAT Competition 2014:Solver and Benchmark Descriptions*, volume B-2014-2, pages 39–40, 2014.

[5] DIMACS Challenge Committee. Satisfiability: Suggested format. *DIMACS Challenge. DIMACS*, 1993.

[6] Peter Cheeseman, Bob Kanefsky, and William Taylor. Where the really hard problems are. In *Proceedings of IJCAI*, volume 91, pages 163–169, 1991.

[7] Jingchao Chen. Glue_lgl_split and Gluesplit_clasp with a split and merging strategy. In *Proceedings of SAT Competition 2014:Solver and Benchmark Descriptions*, volume B-2014-2, pages 37–38, 2014.

[8] Sylvain Conchon, Jean-Christophe Filliâtre, and Julien Signoles. Designing a generic graph library using ML functors. *Trends in functional programming*, 8:124–140, 2007.

[9] Stephen A. Cook. The complexity of theorem-proving procedures. In *Proceedings of the Third Annual ACM Symposium on Theory of Computing*, STOC '71, pages 151–158. ACM, 1971.

[10] Martin Davis, George Logemann, and Donald Loveland. A machine program for theorem-proving. *Commun. ACM*, 5(7):394–397, July 1962.

[11] Niklas Eén and Niklas Sörensson. MiniSat+. `minisat.se/MiniSat+.html`.

[12] Niklas Eén and Niklas Sörensson. The MiniSat page. `minisat.se`.

[13] Niklas Eén and Niklas Sörensson. An extensible SAT-solver. In *Theory and applications of satisfiability testing*, pages 502–518. Springer, 2004.

[14] Niklas Eén and Niklas Sörensson. Translating pseudo-Boolean constraints into SAT. *Journal on Satisfiability, Boolean Modeling and Computation*, 2:1–26, 2006.

[15] R. Fagin. Generalized first-order spectra and polynomial-time recognizable sets. In R. Karp, editor, *Complexity of Computation*, pages 43–73. Amer Mathematical Society, 1974.

[16] Oliver Gableske. Dimetheus. In *Proceedings of SAT Competition 2014: Solver and Benchmark Descriptions*, volume B-2014-2, pages 29–30, 2014.

[17] Michael R. Garey and David S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-completeness*. W. H. Freeman, 1979.

[18] Johan Håstad. Some optimal inapproximability results. In *STOC '97: Proceedings of the twenty-ninth annual ACM symposium on Theory of computing*, pages 1–10. ACM Press, 1997.

[19] Steffen Hölldobler, Norbert Manthey, and Ari Saptawijaya. Improving resource-unaware SAT solvers. In *Logic for Programming, Artificial Intelligence, and Reasoning*, pages 519–534. Springer, 2010.

[20] Jinbo Huang. The effect of restarts on the efficiency of clause learning. In *IJCAI*, volume 7, pages 2318–2323, 2007.

[21] INRIA. The OCaml language. `caml.inria.fr/ocaml/index.en.html`.

[22] V. Kann. On approximability of NP-complete optimization problems. *PhD Thesis, Royal Institute of Technology*, 1992.

[23] Richard M Karp. *Reducibility among combinatorial problems*. Springer, 1972.

[24] Hadi Katebi, Karem A Sakallah, and João P Marques-Silva. Empirical study of the anatomy of modern SAT solvers. In *Theory and Applications of Satisfiability Testing-SAT 2011*, pages 343–356. Springer, 2011.

[25] Alfred B Kempe. On the geographical problem of the four colours. *American journal of mathematics*, 2(3):193–200, 1879.

[26] Leonid Libkin. *Elements of Finite Model Theory*. Springer Science & Business Media, 2004.

[27] Fangzhen Lin and Yuting Zhao. ASSAT: Computing answer sets of a logic program by SAT solvers. *Artificial Intelligence*, 157(1):115–137, 2004.

[28] V Manquinho and O Roussel. Pseudo-Boolean Competition 2012. `www.cril.univ-artois.fr/PB12`.

[29] Norbert Manthey. *Improving SAT solvers using state-of-the-art techniques*. PhD thesis, Diploma thesis, Institut für Künstliche Intelligenz, Fakultät Informatik, Technische Universität Dresden, 2010.

[30] João P Marques-Silva and Karem A Sakallah. GRASP: A search algorithm for propositional satisfiability. *Computers, IEEE Transactions on*, 48(5):506–521, 1999.

[31] David Mitchell, Bart Selman, and Hector Levesque. Hard and easy distributions of SAT problems. In *AAAI*, volume 92, pages 459–465, 1992.

[32] Matthew W Moskewicz, Conor F Madigan, Ying Zhao, Lintao Zhang, and Sharad Malik. Chaff: Engineering an efficient SAT solver. In *Proceedings of the 38th annual Design Automation Conference*, pages 530–535. ACM, 2001.

[33] Mate Soos. Enhanced gaussian elimination in DPLL-based SAT solvers. In *POS@ SAT*, pages 2–14, 2010.

[34] Grigori S Tseitin. On the complexity of derivation in propositional calculus. In *Automation of Reasoning 2: Classical Papers on Computational Logic 1967-1970*, pages 466–483. Springer, 1983.

[35] Jonathan S Turner. Almost all k-colorable graphs are easy to color. *Journal of Algorithms*, 9(1):63–82, 1988.

[36] The Unicode Consortium. Unicode character code charts. `www.unicode.org/charts`.

[37] Mihalis Yannakakis and Fanica Gavril. Edge dominating sets in graphs. *SIAM Journal on Applied Mathematics*, 38(3):364–372, 1980.

# Appendix A

# Source code sample

Here OCaml source code is given for converting an expression into NNF and CNF by the distribution rule, as described in Section 3.7.

```
open Sub

(* converts e to NNF *)
let rec nnf_expr e =
    match e with
    | Not_s (And_s(e1,e2)) -> nnf_expr ( Or_s (Not_s e1,Not_s e2) )
    | Not_s ( Or_s(e1,e2)) -> nnf_expr ( And_s (Not_s e1,Not_s e2) )
    | Not_s ( Not_s e1 ) -> nnf_expr e1
    | And_s (e1,e2) -> And_s (nnf_expr e1, nnf_expr e2)
    | Or_s  (e1,e2) ->  Or_s (nnf_expr e1, nnf_expr e2)
    | Not_s e1 ->  Not_s (nnf_expr e1)
    | Sub_s s1 -> Sub_s s1
    | Card_s _ -> e

(* distributes ORs inwards over ANDs by distribution rule *)
let rec dist_expr e =
    match e with
    | Or_s (And_s(e1, e2), e3) ->
        dist_expr ( And_s(Or_s(e1,e3), Or_s(e2,e3)) )
    | Or_s (e1, And_s(e2, e3)) ->
        dist_expr ( And_s(Or_s(e1,e2), Or_s(e1,e3)) )
    | Or_s  (e1, e2) ->
         let e1 = dist_expr e1 in
         let e2 = dist_expr e2 in (
         match e1, e2 with
         | And_s (_, _), _ -> dist_expr ( Or_s (e1, e2) )
         | _, And_s (_, _) -> dist_expr ( Or_s (e1, e2) )
         | _, _ -> Or_s (e1, e2) )
    | And_s (e1,e2) -> And_s (dist_expr e1, dist_expr e2)
    | Not_s e1 ->  Not_s e1 (* e1 must be term so no need to recurse *)
    | Sub_s s1 -> Sub_s s1
    | Card_s _ -> e

(* converts expression e into CNF *)
let cnf_expr e = dist_expr ( nnf_expr e )
```

# Appendix B

# System output sample

This is the output of *Twist* run on the Clique problem specification given in Appendix C.1, and the instance from Figure 3.2. Verbose output is enabled, which shows the Boolean formula after each stage of the reduction.

```
Parsing problem...........
(forall x in V(forall y in V((x=y | (~(C(x) & C(y)) | (E(x y) | E(y x)))))) & (|C of V| >= K))
Time taken: 0.000326s

Parsing instance..........
Time taken: 0.000699s

Expanding problem.........
((((1 = 1 | (~(C(1) & C(1)) | (E(1 1) | E(1 1)))) & ((1 = 2 | (~(C(1) & C(2)) | (E(1 2) | E(2
1)))) & ((1 = 3 | (~(C(1) & C(3)) | (E(1 3) | E(3 1)))) & ((1 = 4 | (~(C(1) & C(4)) | (E(1 4) |
E(4 1)))) & ((1 = 5 | (~(C(1) & C(5)) | (E(1 5) | E(5 1)))) & ((1 = 6 | (~(C(1) & C(6)) | (E(1
6) | E(6 1)))) & ((1 = 7 | (~(C(1) & C(7)) | (E(1 7) | E(7 1)))) & (1 = 8 | (~(C(1) & C(8)) |
(E(1 8) | E(8 1))))))))))) & (((2 = 1 | (~(C(2) & C(1)) | (E(2 1) | E(1 2)))) & ((2 = 2 |
(~(C(2) & C(2)) | (E(2 2) | E(2 2)))) & ((2 = 3 | (~(C(2) & C(3)) | (E(2 3) | E(3 2)))) & ((2 =
4 | (~(C(2) & C(4)) | (E(2 4) | E(4 2)))) & ((2 = 5 | (~(C(2) & C(5)) | (E(2 5) | E(5 2)))) &
((2 = 6 | (~(C(2) & C(6)) | (E(2 6) | E(6 2)))) & ((2 = 7 | (~(C(2) & C(7)) | (E(2 7) | E(7
2)))) & (2 = 8 | (~(C(2) & C(8)) | (E(2 8) | E(8 2))))))))))) & (((3 = 1 | (~(C(3) & C(1)) |
(E(3 1) | E(1 3)))) & ((3 = 2 | (~(C(3) & C(2)) | (E(3 2) | E(2 3)))) & ((3 = 3 | (~(C(3) &
C(3)) | (E(3 3) | E(3 3)))) & ((3 = 4 | (~(C(3) & C(4)) | (E(3 4) | E(4 3)))) & ((3 = 5 |
(~(C(3) & C(5)) | (E(3 5) | E(5 3)))) & ((3 = 6 | (~(C(3) & C(6)) | (E(3 6) | E(6 3)))) & ((3 =
7 | (~(C(3) & C(7)) | (E(3 7) | E(7 3)))) & (3 = 8 | (~(C(3) & C(8)) | (E(3 8) | E(8
3))))))))))) & (((4 = 1 | (~(C(4) & C(1)) | (E(4 1) | E(1 4)))) & ((4 = 2 | (~(C(4) & C(2)) |
(E(4 2) | E(2 4)))) & ((4 = 3 | (~(C(4) & C(3)) | (E(4 3) | E(3 4)))) & ((4 = 4 | (~(C(4) &
C(4)) | (E(4 4) | E(4 4)))) & ((4 = 5 | (~(C(4) & C(5)) | (E(4 5) | E(5 4)))) & ((4 = 6 |
(~(C(4) & C(6)) | (E(4 6) | E(6 4)))) & ((4 = 7 | (~(C(4) & C(7)) | (E(4 7) | E(7 4)))) & (4 = 8
| (~(C(4) & C(8)) | (E(4 8) | E(8 4))))))))))) & (((5 = 1 | (~(C(5) & C(1)) | (E(5 1) | E(1
5)))) & ((5 = 2 | (~(C(5) & C(2)) | (E(5 2) | E(2 5)))) & ((5 = 3 | (~(C(5) & C(3)) | (E(5 3) |
E(3 5)))) & ((5 = 4 | (~(C(5) & C(4)) | (E(5 4) | E(4 5)))) & ((5 = 5 | (~(C(5) & C(5)) | (E(5
5) | E(5 5)))) & ((5 = 6 | (~(C(5) & C(6)) | (E(5 6) | E(6 5)))) & ((5 = 7 | (~(C(5) & C(7)) |
(E(5 7) | E(7 5)))) & (5 = 8 | (~(C(5) & C(8)) | (E(5 8) | E(8 5))))))))))) & (((6 = 1 | (~(C(6)
& C(1)) | (E(6 1) | E(1 6)))) & ((6 = 2 | (~(C(6) & C(2)) | (E(6 2) | E(2 6)))) & ((6 = 3 |
(~(C(6) & C(3)) | (E(6 3) | E(3 6)))) & ((6 = 4 | (~(C(6) & C(4)) | (E(6 4) | E(4 6)))) & ((6 =
5 | (~(C(6) & C(5)) | (E(6 5) | E(5 6)))) & ((6 = 6 | (~(C(6) & C(6)) | (E(6 6) | E(6 6)))) &
((6 = 7 | (~(C(6) & C(7)) | (E(6 7) | E(7 6)))) & (6 = 8 | (~(C(6) & C(8)) | (E(6 8) | E(8
6))))))))))) & (((7 = 1 | (~(C(7) & C(1)) | (E(7 1) | E(1 7)))) & ((7 = 2 | (~(C(7) & C(2)) |
(E(7 2) | E(2 7)))) & ((7 = 3 | (~(C(7) & C(3)) | (E(7 3) | E(3 7)))) & ((7 = 4 | (~(C(7) &
C(4)) | (E(7 4) | E(4 7)))) & ((7 = 5 | (~(C(7) & C(5)) | (E(7 5) | E(5 7)))) & ((7 = 6 |
(~(C(7) & C(6)) | (E(7 6) | E(6 7)))) & ((7 = 7 | (~(C(7) & C(7)) | (E(7 7) | E(7 7)))) & (7 = 8
| (~(C(7) & C(8)) | (E(7 8) | E(8 7))))))))))) & ((8 = 1 | (~(C(8) & C(1)) | (E(8 1) | E(1 8))))
& ((8 = 2 | (~(C(8) & C(2)) | (E(8 2) | E(2 8)))) & ((8 = 3 | (~(C(8) & C(3)) | (E(8 3) | E(3
8)))) & ((8 = 4 | (~(C(8) & C(4)) | (E(8 4) | E(4 8)))) & ((8 = 5 | (~(C(8) & C(5)) | (E(8 5) |
E(5 8)))) & ((8 = 6 | (~(C(8) & C(6)) | (E(8 6) | E(6 8)))) & ((8 = 7 | (~(C(8) & C(7)) | (E(8
7) | E(7 8)))) & (8 = 8 | (~(C(8) & C(8)) | (E(8 8) | E(8 8))))))))))))))))) & ([C(8) C(7) C(6)
C(5) C(4) C(3) C(2) C(1)] >= 4))
Time taken: 0.001921s

Propagating Booleans.......
(((~(C(1) & C(2)) & (~(C(1) & C(4)) & (~(C(1) & C(5)) & ~(C(1) & C(8))))) & ((~(C(2) & C(1)) &
(~(C(2) & C(3)) & (~(C(2) & C(5)) & (~(C(2) & C(6)) & ~(C(2) & C(7)))))) & ((~(C(3) & C(2)) &
(~(C(3) & C(4)) & ~(C(3) & C(8)))) & ((~(C(4) & C(1)) & (~(C(4) & C(3)) & (~(C(4) & C(5)) &
(~(C(4) & C(6)) & ~(C(4) & C(7)))))) & ((~(C(5) & C(1)) & (~(C(5) & C(2)) & (~(C(5) & C(4)) &
~(C(5) & C(7))))) & ((~(C(6) & C(2)) & (~(C(6) & C(4)) & ~(C(6) & C(8)))) & ((~(C(7) & C(2)) &
(~(C(7) & C(4)) & (~(C(7) & C(5)) & ~(C(7) & C(8))))) & (~(C(8) & C(1)) & (~(C(8) & C(3)) &
(~(C(8) & C(6)) & ~(C(8) & C(7))))))))))) & ([C(8) C(7) C(6) C(5) C(4) C(3) C(2) C(1)] >= 4))
Time taken: 0.001051s

Substituting predicates....
(((~(x8 & x7) & (~(x8 & x5) & (~(x8 & x4) & ~(x8 & x1)))) & ((~(x7 & x8) & (~(x7 & x6) & (~(x7 &
x4) & (~(x7 & x3) & ~(x7 & x2))))) & ((~(x6 & x7) & (~(x6 & x5) & ~(x6 & x1))) & ((~(x5 & x8) &
(~(x5 & x6) & (~(x5 & x4) & (~(x5 & x3) & ~(x5 & x2))))) & ((~(x4 & x8) & (~(x4 & x7) & (~(x4 &
x5) & ~(x4 & x2)))) & ((~(x3 & x7) & (~(x3 & x5) & ~(x3 & x1))) & ((~(x2 & x7) & (~(x2 & x5) &
(~(x2 & x4) & ~(x2 & x1)))) & (~(x1 & x8) & (~(x1 & x6) & (~(x1 & x3) & ~(x1 & x2)))))))))) &
([x8 x7 x6 x5 x4 x3 x2 x1] >= 4))
Time taken: 0.000424s

Converting to CNF..........
((((~x8 | ~x7) & ((~x8 | ~x5) & ((~x8 | ~x4) & (~x8 | ~x1)))) & (((~x7 | ~x8) & ((~x7 | ~x6) &
((~x7 | ~x4) & ((~x7 | ~x3) & (~x7 | ~x2))))) & (((~x6 | ~x7) & ((~x6 | ~x5) & (~x6 | ~x1))) &
(((~x5 | ~x8) & ((~x5 | ~x6) & ((~x5 | ~x4) & ((~x5 | ~x3) & (~x5 | ~x2))))) & (((~x4 | ~x8) &
((~x4 | ~x7) & ((~x4 | ~x5) & (~x4 | ~x2)))) & (((~x3 | ~x7) & ((~x3 | ~x5) & (~x3 | ~x1))) &
(((~x2 | ~x7) & ((~x2 | ~x5) & ((~x2 | ~x4) & (~x2 | ~x1)))) & ((~x1 | ~x8) & ((~x1 | ~x6) &
((~x1 | ~x3) & (~x1 | ~x2))))))))))) & ([x8 x7 x6 x5 x4 x3 x2 x1] >= 4))
Time taken: 0.000263s

Converting to PBC..........
Time taken: 0.000566s

Running PBC-solver.........
Time taken: 0.016087s

Replacing predicates.......

Satisfying assignment: C(1) C(3) C(6) C(7)

Total running time: 0.021337s
```

The satisfying assignment found can be seen to be that shown in Figure 3.2.

# Appendix C

# List of NP-complete problems

Here I present a number of popular NP-complete problems. First the standard English description is given, along with a reference to a proof of its NP-completeness. The same problem is then described using my designed specification language.

## C.1 Clique

### Problem description

Instance: Graph $G = (V, E)$, positive integer $K \leq |V|$.
Question: Does $G$ contain a clique of size $K$ or more, i.e. a subset $C \leq V$ with $|C| \geq K$ such that every two vertices in $C$ are joined by an edge in $E$?
Proof: Transformation from Vertex Cover [23].

### Problem specification

```
Given:
  V E K
Find:
  C
Satisfying:
  (* all vertices in the clique must be connected by an edge *)
  (forall x in V ( forall y in V (
      ~(x=y) ->  ( (C(x) & C(y)) -> (E(x y) | E(y x)) )
  ) ) )
  &
  (* the size of the clique is at least equal to the input K *)
  ( |C of V| >= K )
```

## C.2 Dominating Set

### Problem description

Instance: Graph $G = (V, E)$, positive integer $K \leq |V|$.
Question: Is there a dominating set of size $K$ or less for $G$, i.e. a subset $D \subseteq V$ with $|D| \leq K$ such that for all $u \in V - D$ there is a $v \in D$ for which $\{e, v\} \in E$?
Proof: Transformation from Vertex Cover [17].

### Problem specification

```
Given:
  V E K
Find:
  D
Satisfying:
  (* all vertices x not in the set... *)
  (forall x in V ( ~D(x) ->
     (* ...are connected to some vertex y in the set *)
     exists y in V ( D(y) & (E(x y) | E(y x)) )
  ) )
  &
  (* the size of the set is at most K *)
  (|D of V| <= K)
```

## C.3 Exact Cover by 3-sets

### Problem description

Instance: Set $X$ with $|X| = 3q$ and a collection $C$ of 3-element subsets of $X$.
Question: Does $C$ contain an exact cover for $X$, i.e. a subcollection $E \subseteq C$ such that every element of $X$ occurs in exactly one member of $E$?
Proof: Transformation from 3-Dimensional Matching [23].

### Problem specification

```
Given:
  X C
Find:
  E
```

```
Satisfying:
  (* all elements of X are in one member of E... *)
  (forall x in X ( exists a b c in C (
    (E(a b c) & (a=x|b=x|c=x) )
    (* ...and only one member of E *)
    & (forall d e f in C (
       ( (E(d e f) & (d=x|e=x|f=x)) -> (a=d & b=e & c=f) )
    ) )
) ) )
```

## C.4 Graph 3-Colourability

### Problem description

Instance: Graph $G = (V, E)$.
Question: Is $G$ 3-colourable, i.e. does there exist a function $f \to \{R, G, B\}$ such that $f(u) \neq f(v)$ whenever $\{u, v\} \in E$?
Proof: Transformation from 3-Sat [23].

### Problem specification

```
Given:
  V E
Find:
  R G B
Satisfying:
  (* every vertex is exactly one colour *)
  (forall x in V (
        ( R(x) & ~G(x) & ~B(x))
      | (~R(x) &  G(x) & ~B(x))
      | (~R(x) & ~G(x) &  B(x))
  ))
  &
  (* all pairs of connected vertices are different colours *)
  (forall x y in E (
            (R(x) & ~R(y))
          | (G(x) & ~G(y))
          | (B(x) & ~B(y))
  ))
```

## C.5 Hamiltonian Path

### Problem description

Instance: Graph $G = (V, E)$.
Question: Does $G$ contain a Hamiltonian path, i.e. a path along edges $e \in E$ which visits each vertex $v \in V$ exactly once?
Proof: Transformation from Vertex Cover [17].

### Problem specification

```
Given:
  V E
Find:
  T (* linear order on V which will form our path *)
Satisfying:
  (* T is reflexive *)
  (forall x in V ( T(x x) ) )
  &
  (* T is transitive *)
  (forall x in V ( forall y in V ( forall z in V (
    (T(x y) & T(y z)) -> T(x z) ) ) ) )
  &
  (* T is antisymmetric *)
  (forall x in V ( forall y in V (
    (T(x y) & T(y x)) -> x=y ) ) )
  &
  (* T is a linear order on V if is RTA and: *)
  (forall x in V ( forall y in V (
    T(x y) | T(y x) ) ) )
  &
  (* y is successor of x in T implies there is an edge from x to y *)
  (forall x in V ( forall y in V (
    ( ~(x=y) & T(x y) &
      ~(exists z in V ( ~(z=x) & ~(z=y) & T(x z) & T(z y) )) ) ->
    (E(x y) | E(y x) ))
  ))
```

## C.6 Independent Set

### Problem description

Instance: Graph $G = (V, E)$, positive integer $K \leq |V|$.
Question: Does $G$ contain an independent set of size $K$ or more, i.e. a subset $I \subseteq V$ such that $|I| \geq K$ and such that no two vertices in $I$ are joined by an edge in $E$?
Proof: Transformation from Vertex Cover [17].

### Problem specification

```
Given:
  V E K
Find:
  I
Satisfying:
  (* if nodes a and b are both in the independent set I,
     then they must not be connected by any edge *)
  (forall a in V ( forall b in V (
    ( I(a) & I(b) ) -> ~( E(a b) | E(b a) )
  )))
  &
  (* the size of the independent set is at least K *)
  (|I of V| >= K)
```

## C.7 Minimum Maximal Matching

### Problem description

Instance: Graph $G = (V, E)$, positive integer $K \leq |E|$.
Question: Is there a subset $M \subseteq E$ with $|M| \leq K$ such that $M$ is a maximal matching, i.e. no two edges in $M$ share a common endpoint and every edge in $E - M$ shares a common endpoint with some edge in $M$?
Proof: Transformation from Vertex Cover for cubic graphs [37].

### Problem specification

```
Given:
  V E K
```

```
Find:
  M
Satisfying:
  (forall x y in E (
     (* no two edges in M share a common end point *)
     (M(x y) -> (forall a b in E ( ~(a=x & b=y) ->
        ( M(a b) -> (~a=x & ~a=y & ~b=x & ~b=y) ) ) ) )
     &
     (* edges not in M share an endpoint with some edge in M *)
     (~M(x y) -> (exists a b in E ( M(a b) & (x=a|x=b|y=a|y=b) )))
  ) )
  &
  (* the size of the set is at most K *)
  (|M of E| <= K)
```

## C.8 Vertex Cover

### Problem description

Instance: Graph $G = (V, E)$, positive integer $K \leq |V|$.
Question: Is there a vertex cover of size $K$ or less for $G$, i.e. a subset $C \subseteq V$ with $|C| \leq K$ such that for each edge $\{u, v\} \in E$ at least one of $u$ and $v$ belongs to $C$?
Proof: Transformation from 3-Sat [23].

### Problem specification

```
Given:
  V E K
Find:
  C
Satisfying:
  (* at least one endpoint of every edge is in the cover *)
  (forall x y in E ( C(x) | C(y) ) )
  &
  (* the size of the cover is at most K *)
  (|C of V| <= K)
```

# Appendix D

# Project Proposal

*Tom Waszkowycz*
*King's College*
*tw397*

Part II Project Proposal

## Solving NP-Complete Problems via Satisfiability

*23rd October 2014*

## Introduction and Description of the Work

Many problems in fields such as AI, circuit design and automatic theorem proving are NP-Complete. We want to solve them efficiently, but no polynomial time algorithm has been discovered for any NP-Complete problem, despite the fact that no super-polynomial lower bounds have been found either.
One problem in NP for which many of the instances that occur in practice can be solved relatively efficiently is the Boolean satisfiability problem (SAT). This is the problem of determining if there exists an assignment of the variables of a formula in Boolean logic that satisfies the formula - i.e. can evaluate it to true. Over the past decade there have been dramatic advances in heuristic based SAT-solvers, which now perform very well for many practical applications, and many of which are readily available as free or open source software.
The Cook-Levin theorem states that SAT is NP-Complete. This means that any problem in NP can be reduced in polynomial time to an instance of the Boolean satisfiability problem. I plan to use this result to take instances of various NP-Complete problems, reduce them into equivalent formulae of Boolean logic, and then use an 'efficient' SAT solver to compute a solution to the original problem. This can be done with any problem in NP, and hence any NP-Complete problem.

## Starting Point

I have knowledge of various systems of logic from attending courses in Discrete Mathematics in Part IA of the Computer Science Tripos and Logic and Proof in Part IB, and knowledge of computational complexity theory from attending the Complexity Theory course in Part IB.

## Substance and Structure of the Project

I aim to implement a system that can take an instance of a problem in the complexity class NP, reduce it to an equivalent Boolean satisfiability problem, and then attempt to solve it using a SAT-solver.
This will be done by first constructing a specification language to describe problems in NP, on which work has previously been done. Ronald Fagin proved in 1974 the class of problems NP is exactly the set of all formulae expressible in existential second-order logic (∃SO) (Fagin's Theorem). This means all problems that I would wish to input into my system can be expressed as such, and so the front end of the system would consist of a parser which accepts such languages.
As the input problem will be in NP, by the Cook-Levin theorem we can reduce this in polynomial time to an instance of SAT. This is done by substituting values and expanding all existential and universal quantifiers over the (finite) values of the domain given by the supplied instance. This will then produce a Boolean logic formula, which can be considered as an instance of SAT equivalent to the original problem instance. It would be useful to produce this in a standardised format (such as DIMACS-CNF).
This formula will then be fed into the back end of the system, consisting of a 3rd party SAT solver. This will then attempt to solve the formula, and hence the original problem instance that was input, outputting the result (if any). It is worth noting that all current SAT-solvers still have exponential worst case time complexity, and so it is not uncommon for them to fail to find a solution within their given time limit.
The first problem I plan to focus on first is graph three colourability (3COL), as it is relatively easy to specify in ∃SO, and instances of the problem can be described simply by a graph consisting of a set of vertexes and a set of the edges between them. To evaluate it I will test it against a number of 3COL instances, which can be found in online benchmark collections. I will evaluate its performance by comparing the size of problems that it is able to solve against the size of problems required in practical applications and against other similar systems.

Once I have this working for 3COL I plan to expand it to work with other NP-Complete problems, and test them similarly. A possible extension, time permitting, would be to incorporate problems that take both an integer parameter as well as a graph (e.g. Clique, Independent Set, Vertex Cover), which are more difficult to specify in ∃SO. I would consider SAT solvers which incorporate ways of expressing integer constraints, and extend my specification language to allow for these.
Possible further evaluation methods could involve contrasting performance of my system on different NP problems, or comparing performance with a range of different SAT-solvers. To improve my project I could look into extending the specification language to allow for easier descriptions of more complex NP languages, and any improvements that I could make to my system to improve performance.

## Special Resources Required

I will require use of my own laptop for development, one or more SAT-solvers (many are free or open source), and testing benchmarks of instances of NP-Complete problems (also freely available).

## Success Criterion

My system should be able to take a description of an NP problem (e.g. 3COL) in my specification language, along with an instance of this problem (i.e. a graph), convert this into an equivalent SAT instance, and feed this into a SAT-solver to determine the solution to the original problem instance.

## Timetable and Milestones

1. **27th Oct - 9th Nov** Initial research into similar work done
   **Deliverable:** More in depth knowledge and notes on existing research, and how this relates to my project

2. **10th Nov - 23rd Nov** Identification of appropriate SAT solver/s, testing benchmarks and specification language
   **Deliverable:** Be able to specify the above with reasons

3. **24th Nov - 7th Dec** Planning and initial coding of core project

4. **8th Dec - 21st Dec** Continue coding of core project
5. **22nd Dec - 4th Jan** Complete and test core project
   **Deliverable:** System tested and working for 3COL instances
6. **5th Jan - 18th Jan** Begin evaluation of system, and extend core project
7. **19th Jan - 1st Feb** Preparation of progress report
   **Deliverable:** Fully prepared document and presentation
8. **2nd Feb - 15th Feb** Complete evaluation and any further extensions
   **Deliverable:** System evaluated as described above
9. **16th Feb - 1st Mar** Begin writing dissertation
   **Deliverable:** Dissertation plan
10. **2nd Mar - 15th Mar** Continue dissertation writing
11. **16th Mar - 29th Mar** Complete draft of dissertation and submit to supervisor
    **Deliverable:** All parts of dissertation complete
12. **30th Mar - 12th Apr** Revise dissertation based on supervisor feedback
13. **13th Apr - 26th Apr** Any last minute revisions and preparation for submission
    **Deliverable:** Dissertation ready for submission