Explainable SAT Proofs for Social Choice and Argumentation Theory

Henrijs Princis

May 2022

Abstract

The method of using SAT solvers to find human readable proofs for theorems in social choice and argumentation theory is evaluated. Four key limitations are identified and possible future work is suggested remedying these. The method was applied to four theorems - three in social choice and one in argumentation theory. A novel approach of generating human readable text description of proofs is introduced and evaluated.

Acknowledgements

I would like to give a special thanks to Dr. Richard Booth for his continuous support, constant encouragement and invaluable insight to this project. His idea sparked my curiosity and made this project possible.

Table of Contents

Contents

List of Figures

1 Introduction

1.1 Motivation

Computer generated proofs have a lot of potential. As early as 1970s an unsolved problem in mathematics was proven by a computer [2]. Since then, tremendous progress has been made to improving existing proof techniques and the computing power has grown exponentially. This might make one think that human mathematicians only have to come up with theorems and that these can be then verified by computers. However, this is clearly not the case. Hardest solved problems of today such as Fermat's Last Theorem [44] or Pioncre Conjecture [38] were not proven by computers. They were proven by human mathematicians.

This project aims to explore the limitations of current state of the art (SoTA) systems as well as *how* to adapt existing proof techniques such that they become more accessible to everyone. In particular, in the context of social choice and argumentation theory. While a lot of work has gone into creating systems of mathematics which lend themselves to easy automation, it is only recently that the research community has taken a closer look at how to adequately explain the results these systems produce. For example usage of the word XAI (Explainable AI) has exponentially increased in popularity over the last decade 1.

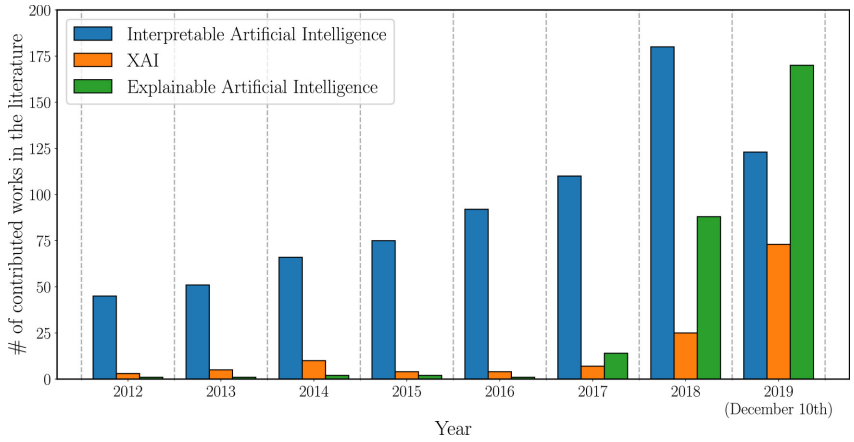


Figure 1: This illustration was taken from [7]. It shows the "Evolution of the number of total publications whose title, abstract and/or keywords refer to the field of XAI during the last years."

1.2 Project Outline

At first glance, it may seem that this project is trying to tackle four independent fields, namely, **social choice** which concerns itself with how to aggregate preferences of a group, **argumentation theory** which tries to answer which

argument are good or bad, **computer assisted proofs** and **proof explainability**.

The main aim of the project is to study ways computer generated proofs can be explained. However, there exist numerous techniques for using computers to assist humans in proving theorems. Thus the scope was narrowed to theorems in social choice since the field lends itself nicely to computer assistance. This is because it uses axiomatic method, combinatorial structures and can be defined using elementary mathematical notions [21]. A more specific description of the problems concerning social choice is given in Section 2.1. Famous impossibility results in social choice will be tackled using SAT solvers. A method eloquently described in [21].

It turns out that argumentation theory shares many commonalities with social choice. Furthermore, a sub-field of argumentation theory is judgement aggregation (JA), which concerns itself with how to aggregate different agent arguments so an overall verdict can be passed. This overlaps with social choice. For a more elaborate description see Section 2.3 on bridging the gap between the two fields. This means that JA is a logical next step for evaluating the versatility of using SAT solvers to aid in finding proofs. In other words, we would like to explain proof techniques which are versatile and applicable to many fields.

Finally, computer generated proofs and their explainability will be intimately connected because the way that the proof is generated will also be the *reason* why it is correct. One of the major challenges that this project will tackle is how to resolve the communication barrier between computers which can work through thousands of very well specified cases simultaneously and humans who think of theorems in a very abstract way. Removing the communication barrier by automatically finding the most compact proof and then translating it into natural language should allow for greater accessibility and more trust in the output of the algorithms.

1.2.1 Overview of Sections

In background, motivation behind some of the problems in social choice and argumentation theory is provided. Properties encoding the motivation are defined. Notation is introduced which will be used throughout the report.

In approach, I will describe the high-level process of translating the properties defined in background into SAT. I will also discuss the two approaches to explain the results produced by SAT solvers.

In implementation, I go over how this high-level description of the problems translates into pseudo-code. I also briefly discuss how the two explanations of SAT results were implemented.

In Results and Evaluation section I report and discuss the performance of SAT solvers, visualisation of the approach and evaluation of the method overall. I discuss some of the strengths and limitations which this method faced.

In future work, I discuss some potential remedies for the issues which became apparent in results and evaluation section.

The conclusion will provide a short summary of the main takeaways of this project and summarise my findings in a larger context.

In reflection on learning section, I provide an evaluation of decisions made throughout the report.

2 Background

2.1 Social Choice

Social choice is a sub-field of computer science which studies how to combine individual preferences into preferences of a group. For example, it studies how to decide a winner of an election based on voter's ballots.

At first glance, it might seem that representing a group's opinion is as simple as taking the majority vote. Indeed, this intuition holds for elections with only two candidates (or alternatives); however, as soon as a third candidate is introduced, problems begin to arise.

Motivating Example In the 2000 American presidential election, George W. Bush, Al Gore and Ralph Nader were running for president. Ralph Nader was a third party candidate and was not expected to win. The predictions came true and he received a small, but not insignificant, 2.7% of all votes. George W. Bush ended up winning the election by a much slimmer margin.

The poll on Nader's website indicates that majority of people who voted for Nader would have voted for Bush instead if Nader was not running. This means that Gore would have won had Nader not ran. This means that Nader "spoiled" the election.

Thought Exercise Imagine that you could decide the format of how the election is ran to *prevent* situations where removing a candidate *spoils* an election. What is the best way to do that? Is it even possible?

Before we answer that question, we need to define what the format of the ballot is. In other words, how do voters represent their preferences and what it means to aggregate them.

2.1.1 Ballots or Preferences

A voter's ballot is represented as a *weak* linear order over candidates. A possible ballot for candidates A, B, and C might be presented as $A \prec B \simeq C$.

This ballot states that candidate A is the most preferred and B and C are tied for second.

Sometimes ties might be undesirable in which case the ballot is a *strict* linear order over the candidates. For example, $A \prec B \prec C$. In this report, I will mostly be using *strict* linear orders to represent ballots.

A *profile* is a *tuple* of ballots or preferences. For example, if voter 1 has preference of $A \prec B$ and voter 2 has preferences of $B \prec A$, then a profile which represents both of their preferences is ($A \prec B$, $B \prec A$).

2.1.2 Voting Rule or Social Welfare Function

A voting rule otherwise known as social welfare function will map a *profile* to a winning candidate(s) (or alternative(s)). Intuitively, a voting rule is just an aggregation procedure.

Given a profile, a voting rule may output a single candidate as the winner. For example candidate A. It may also output multiple candidates as winners. For example "A and B". Finally, it could output a *ranking* of the candidates. For example $A \prec B \prec C$.

An example of a voting rule is "majority rule" which states that if a candidate has received a majority of the votes, then that candidate should win. Majority rule, is not defined for all profiles. What if no candidate reaches a Majority? Majority rule also *cannot* produce a ranking of participants. An intuitive generalisation of majority rule is Borda count. Voters assign n points to their top choice, n-1 points to their second choice and so on. The winner of the election is then the candidate which scores the most points.

There are certain properties which are desirable for all voting rules. I provide brief descriptions below, but cover their specific meaning more in Approach 3.4. For more elaborate descriptions and importance of the properties visit The Stanford Encyclopedia of Philosophy for Arrow's theorem [34].

- Non-Dictatorship

  This property states that there should not exist a voter whose ballot *always* decides the winner of the election. It could be that other voter preferences are considered, but this is only in cases when the dictator is indifferent between the alternatives (I.E. there is a tie between them).

- Pareto Efficiency

  If every voter prefers candidate A over B, then A should rank higher than B in the election.

  In the case of single winner elections, Pareto Efficiency is obeyed if there does not exist candidate A which is preferred by everyone to candidate B, but the voting rule has assigned candidate B as the winner.

- Unrestricted Domain

  Every voter should be able to submit any set of preferences. In other words, they may rank the candidates in any order they choose with the ability to indicate ties.

- Social Ordering

  Is a property stating that there can be no cycles for the winner of the election. This means that the voting rule should only produce linear orders

with each candidate only being listed once. For example, the aggregation procedure cannot produce this linear order $A \prec B \prec A$ since it contains a cycle.

- Independence of Irrelevant Alternatives (*IoIA*).

  This property states that the voting function should only consider the relative position between any two candidates to decide their relative position in the election.

  For example, given 3 candidates A, B, and C and a single voter who considers two strong linear orders $o_1 = A \prec B \prec C$ and $o_2 = B \prec A \prec C$.

  Notice, that in both linear orders candidate A is preferred to candidate C. *IoIA* tells us that for winning ranking, the relative position of A and C should stay the same. I.E. either $A \prec C$ or $C \prec A$ for both elections. It rules out the possibility of o_1 having a different relative ranking of candidates A and C to o_2 in the election.

  For example, if the voter would use o_1 and the winning ranking of that election would be $A \prec C$ and then use o_2 to and obtain a ranking where $B \prec C$ then *IoIIA* would not hold.

  Similarly, if there are many voters in the election and some voter changes his ballot from o_1 to o_2, then the relative ranking between A and C should stay the same.

- Manipulable and Strategy-Proof

  A voting rule is said to be manipulable if some voter can misrepresent her preferences and change the outcome of the election in such a way that the new winner of the election is more preferred by her than what it was under her original set of preferences. A voting rule is strategy-proof if it is not manipulable.

  This property was not obeyed by the American election system, since candidates who voted for Ralph Nader would have been better off voting for Al Gore.

Ideally, a voting rule such as "Borda count" will obey all axioms presented. However, in 1950 Arrow Published his famous PhD thesis [3], in which he presents Arrow's impossibility theorem. It states that any voting rule cannot simultaneously obey Non-dictatorship, Pareto Efficiency, Unrestricted Domain, Social Ordering and Independence of Irrelevant Alternatives.

Indeed, Borda count obeys all properties except *Independence of Irrelevant Alternatives*.

Later in 1973, Gibbard and Satterhwaite proved that all voting rules in which a single winner is produced, must be either a dictatorship, limit the possible outcomes to only two alternatives or be manipulable [23]. This is known as the Gibbard and Satterhwaite Theorem (GST).

Finally, a weaker version of GST (referred to as simplified GST in this report) was used as an introductory example in [21]. It states that no voting rules which assign a single winner may obey strategy-proofness and majority rule.

In summary, we have identified three theorems, namely, simplified GST, full GST, and Arrow's impossibility theorem for which we will verify the base-cases using a SAT solver.

2.2 Abstract Argumentation

2.2.1 Monotonic and Non-Monotonic Logic

Abstract Argumentation is mainly built on Dung's model of argumentation [20]. Dung's paper describes how to model non-monotonic logic. In non-monotonic logic, a conclusion may be retracted in light of further evidence [37].

For example, if Anna is a person we can reasonably infer that she can walk. However, if we learn that Anna is only a 6 months old baby, then we can retract the conclusion that she can walk.

Traditional logic does not capture such relationships very well. I.e.

p1 All human can walk.

p2 Anna is a human.

c1 Anna can walk.

p3 All babies cannot walk.

p4 Anna is a baby.

c2 Anna cannot walk.

It is easy to see that the set of premises produce a contradiction. Since anything may be derived from a contradiction we have a problem.

2.2.2 Argument Framework

Dung's main contribution is that of an Argument Framework (AF). The idea is to represent arguments as nodes in a graph. These are statements which are typically true. Directed edges are added denoting an attack relation between arguments.

In our example, one argument would consist of p1, p2 and c1 and would be represented as a single node. The other node would correspond to p3, p4 and c2. We may denote the set of arguments as *Args*. In the example *Args* = {A, B}.

Argument A rules out the possibility of argument B being true and vice versa. Therefore, A attacks B and B attacks A. More formally, the set of all attack relations may be denoted as a set consisting of tuples *Args* x *Args*. Let $\rightharpoonup$ denote the set of tuples. In our example $\rightharpoonup$ = {(A,B), (B,A)}

An argument framework or (AF) is then a set {*Args*, $\rightharpoonup$}

Each argument or node may have three labels.

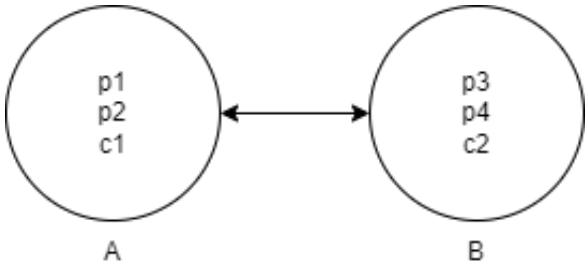


Figure 2: This image shows the AF consisting of nodes A and B attacking each other. Any AF where there are only two nodes attacking each other is known as two loop AF.

- In - corresponds to accepted.
- Out - corresponds to rejected.
- Undecided - means that there is not enough information to decide.

It is possible to label the two arguments in 9 ways; however, we would also like them to respect the attack relation. In our example, it doesn't make sense to accept both c1 and c2.

Dung's proposed method uses the *complete* and *admissible* semantics to enforce the constraints. Multiple formulations exist which are logically equivalent, but I will use Caminada's et. al. formulation in this report [6].

1. If an argument is labelled *in*, then all arguments which are attacked by it are labelled *out*.
2. If an argument is labelled *out*, then there exists at least one argument that attacks it which is *in*.
3. If an argument is *undecided*, then at least one of its attackers is also *undecided* and none of the attackers are *in*.

A complete semantic is one which obeys all three of the axioms. An admissible semantic is one which obeys only the first two, but not necessarily the third. All complete labellings are admissible, but not vice versa.

In this case the there exist three *complete* and three *admissible* labellings of the aforementioned scenario: A is *in* and B is *out*. A is *out* and B is *in*. And finally A is *undec* and B is *undec*.

Each complete labelling of the arguments corresponds to a coherent interpretation of the facts. Either we reject the argument which states that Anna is a baby, and conclude that Anna can walk. Or we reject the argument which states that all humans can walk and conclude that Anna cannot walk. Finally, we have the option of saying that there is not enough information to tell, and we can label both arguments as being undecided.

We can divide the complete labellings even further into grounded and preferred semantics. Grounded semantics only accepts arguments which we are required to accept. In this case, the grounded semantic is that both nodes are labelled as *undec*. Preferred semantics on the other hand is a complete labelling which accepts as many arguments as possible. In this case, there are two preferred semantic labellings A is *in* and B is *out* and A is *out* and B is *in*.

In summary, we have introduced the notion of an argument framework (AF) which is widely used in argumentation theory and defined the *complete* semantic. We are now ready to see how argumentation theory overlaps with social choice.

2.3 Bridging the Gap Between Social Choice, Argumentation Theory, and Judgment Aggregation

Social Choice comes up in Abstract Argumentation as soon as we have multiple *agents*. For example, imagine that in a court of law there are multiple arguments presented and that each juror (or *agent*) will form a coherent view of what happened. In other words, they will assign a complete labelling to all arguments presented so far.

At the end of a trial, the jurors need to come to a decision as a group of whether the suspect is guilty. In other words, they need a procedure to aggregate their reasoning so a verdict can be reached. Judgement Aggregation is a sub-field of Social Choice which tries to solve the issue of how to aggregate reasoning.

In judgement aggregation, agents submit their labellings of an AF and then an aggregation procedure is used to decide the group's labelling of the AF. A helpful analogy connecting social choice to argumentation theory is to think of the agents as voters, the labellings of AFs provided by the agents as ballots and the aggregation procedure as a voting rule.

2.4 Judgement Aggregation

Similarly to social choice, we have certain properties which we would like our aggregation procedure to obey. The ones which will be used in this report are Isomorphism, Anonymity, Unanimity, AF-Independence and Collective Rationality. Only very basic, intuitive descriptions are provided here. For more elaborate descriptions see [11].

- **Anonymity** is a property which states that the aggregated result should not consider *who* submitted each preferences. For example, if agent 1 submits labelling A and agent 2 submits labelling B then the aggregated labelling C should be the same as if agent 1 submitted labelling B and agent 2 submitted labelling A.

- **Unanimity** is a property which states that if an argument in the AF is labelled the same by everyone then that argument's aggregated label should be what everyone assigned it. For example, if everyone votes that

the argument a in an AF is *in*, then the aggregated label of argument a should be *in*. Using the analogy to social choice, this loosely corresponds to *Pareto Efficiency*.

- **Isomorphism** is a property which states that two argument frameworks which are isomorphic to one another should be labelled the same way. In other words, if the "structure" of two argument frameworks is the same, then they should be aggregated in the same way.

  For example, suppose you have a two-loop AF and one of the arguments (or nodes in graphical representation) is named *p1* and the other is *p2*. You also have two agents ag_1 and ag_2 both of which label *p1* as *in* and *p2* as *out*. Finally, suppose the aggregation procedure labels *p1* to be *in* and *p2* to be *out*. What isomorphic property says, is that had the agents submitted a labelling in which they both label *p2* as *in* and *p1* as *out*, the aggregation procedure *must* produce the *same* labelling except with the new node names. I.E. *p2* as *in* and *p1* as *out*.

- **AF-Independence** is a property that is related to Independence of Irrelevant Alternatives in social choice. It states that the aggregation procedure should only "decide" what to label each argument solely based on the labels given by participants to that argument. For example, suppose you have a two-loop AF with two nodes *p1* and *p2*. The voting rule should be able to decide what label *p1* is without knowing how people labelled *p2*.

- **Collective Rationality** is a property that states that the produced labelling after aggregation should be a complete labelling. For example, suppose you have a two-loop AF with two nodes *p1* and *p2*, then it should never be the case that the aggregation procedure labels both nodes as *out*.

Theorem in Argumentation Theory Whilst there are many impossibility results in JA, in this report I chose to focus on Theorem 3 in [11]. This is because it uses many properties that typically a voting rule does and it was easy to adapt the SAT solving approach developed for Social choice. The theorem states that: There is no aggregation method satisfying all of Isomorphism, Anonymity, Unanimity, AF-Independence and Collective Rationality where there are two or more agents.

Summary So far, we have covered the relationship between social choice and argumentation theory through the view of judgement aggregation. We have also defined three theorems in social choice as well as a theorem in argumentation theory. Our goal will be to use a computer to verify the base-cases for all of these theorems. Exactly how to go about using them will be introduced in the next section.

2.5 Computer Aided Proofs

Computer aided proofs have been used since the invention of computers. Perhaps the most famous such proof is that of the 4-colour theorem found by K. Appel and W. Haken. in 1976 [2].

Many automatic theorem provers and proof assistants exist nowadays. They are typically based on a zeroth (or *propositional*, first or higher order logic. Propositional logic is the least expressive, but is decidable, complete and sound. Whereas higher order logic is undecidable, and incomplete or unsound [43].

The book chapter which served as inspiration for this project uses propositional logic to encode base cases of theorems in social choice thus it will be the main focus of my work.

2.5.1 Propositional Logic

Propositional Logic uses propositions which are made out of *atoms* and logical connectives. Each atom may be either true or false. Logical connectives such as "and" and "or" may be used to join up individual atoms into a function. Atoms are sometimes referred to as literals in the literature.

A function in porpositional logic takes boolean variables as input and returns a boolean output.

For example: f(A, B, C) = (A or B) and not C.

In this case, A, B, and C are atoms. f(A, B, C) is a formula which takes in a truth assignment of A, B, and C and returns either true or false.

I.E. f(false, false, true) = false.

2.5.2 SAT and CNF

SAT is short for SATisfiability. In propositional logic, SAT refers to checking whether there exists an assignment of atoms which makes a propositional function true. If no such assignment exists, the formula is a contradiction and we call it unsatisfiable.

There are multiple ways of checking for SAT. The simplest way would be to try every assignment of variable and see if any of them satisfy the formula. It is easy to see that this would take $\mathcal{O}(2^n * m)$ time. Where n corresponds to the number of atoms and m corresponds to time spent computing the output of the function.

Checking for SATisfiability is an NP-complete problem [17]. This means that we can check solutions in polynomial time, but there are no known algorithms which can find a solution in polynomial time. This means, that the worst case time complexity for *any* algorithm so far will be x^n where n is some real number > 1 and x is a positive integer denoting the number of atoms in a formula.

CNF Most modern algorithms work on functions which are in CNF. CNF is short for Conjunction Normal Form. CNF is conjunction of clauses where **each**

clause is a disjunction of atoms. Any propositional logic formula can be converted into CNF. CNF of a formula is equivalent to the original formula in terms of satisfiability. Intuitively, one can think of conjunction as denoting "and" and disjunction as denoting "or".

For example, instead of defining f(A, B, C) = "(if A then B) and C". One could turn it into CNF f(A, B, C) = "(not A or B) and C" and preserve the meaning of the original formula. I.E. when the original f(A, B, C) is satisfied so is the CNF version and vice versa.

The procedure for converting into CNF is not widely important, but is used throughout the report. See lecture slides by Dillig [19] for how to convert into CNF.

DIMACS Finally, a CNF formula can be turned into a *DIMACS* format. This is the standard way to store CNF formulas and is the format expected by most SAT solvers. Each atom is assigned a number beginning at one. Then, all clauses all disjunctive clauses are put on the same line and zero is concatenated at the end. The zero indicates the end of a disjunctive clause. All conjunctive clauses appear on a new line.

For example, suppose your CNF formula is "(A or B) and (C or A)". To convert into DIMACS, all atoms (A, B, C) are replaced with (1, 2, 3) respectively. Resulting in a formula "(1 or 2) and (3 or 1)". We can replace all "or"s with spaces and all "and"s with newlines. Finally, a clause at the top that indicates the number of atoms (3) and number of clauses (2) is inserted resulting in the DIMACS file:

1. p cnf 3 2
2. 1 2 0
3. 3 1 0

Better SAT Solving Procedures Many approaches to solving SAT have been developed. While most do not provide significantly better worst-case time compolexity, they are often very powerful. "modern SAT solvers can often handle problems with millions of constraints and hundreds of thousands of variables" [36].

Among the most popular are Davis–Putnam–Logemann–Loveland algorithm (DPLL) [18] and conflict-driven clause learning (CDCL) [33]. Only DPLL will be explained because CDCL is almost entirely based on DPLL and is not needed to understand the report.

DPLL recursively assigns a truth value to an atom and splits the formula into 2 (one where the atom is true and another where the atom is false). It then makes trivial assignments. For example, if a clause consists of a single literal then we may assign this literal and eliminate the clauses which use it. This is known as *unit propagation*. **Pure literal elimination** assigns an atom if it only appears in one polarity I.E. it is always positive or always negated. If a

contradiction is encountered, the algorithm backtracks to last truth assignment it made. The process is then repeated until all possible assignments have been exhaustively checked or a valid truth assignment is found.

In the end, one ends up with a tree graph whose leaves correspond to an assignment of all variables. The non-leaf nodes represent decisions and the edges represent the implications of the previously made decisions I.e. unit propagations and pure literal eliminations. An amazing interactive resource to use is [1] with the help of which figure 3 was generated.

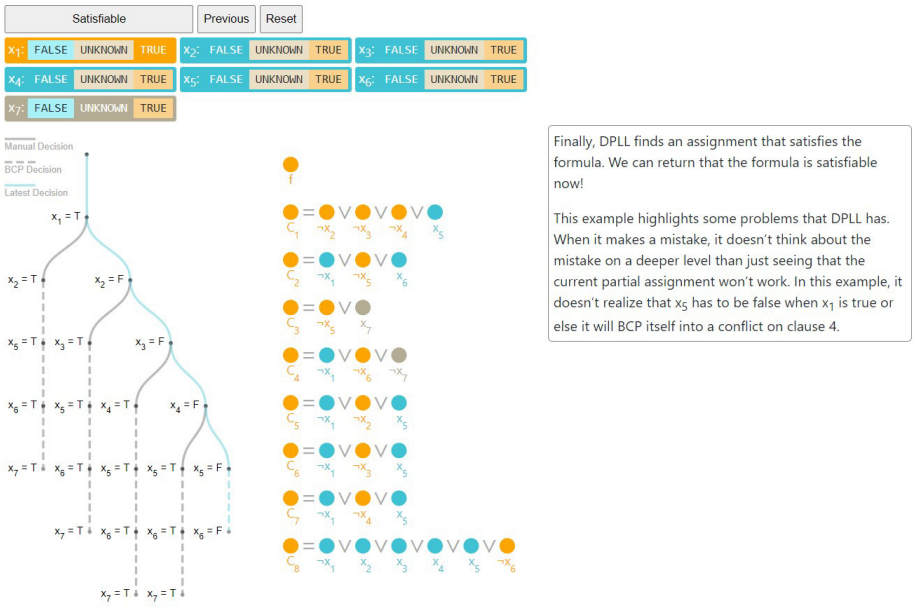


Figure 3: This image was taken from [1]. It illustrates the decision tree produced by the DPLL algorithm. You can see that a valid assignment was eventually found by the process.

2.6 Proof Extraction

While the diagram produced in figure 3 is very visually clear, it is not so straight forward to extract the logical proof in a rigorous manner. The general idea is that you can represent the assumptions you have made which lead to a contradiction as an *additional clause*. For example, suppose that assigning $x_1 = true$, $x_2 = false$ leads you to a contradiction. This means that not(x_1 and not(x_2)) is true.

This can be re-written in CNF as (not x_1) or x_2 and then added as a new clause. In the literature, this is known as a *conflict clause* because its negation creates a conflict. If required, we can always provide a justification for *why* it

creates a contradiction since DPLL only uses unit propagation or pure literal elimination. Thus the derived clause will result from applying one of these operations. Eventually, if the provided formula is unsatisfiable, the empty clause will be derived and added.

The main challenge with this approach to make it human readable is that the derivation can be hundreads of lines long. While it may be too tedious to check for a human, a simple small proof-checker program whose source code is easier to verify by a human is used. The checking procedure is known as Reverse Unit Propagation (RUP) [24].

It works backwards, starting at the final derived clause. It takes the negation of the whole clause and then substitutes negative atoms into all of the formulas above it. This should make one of the formulas false. The process is then attractively repeated. It is straight forward to see that this has a time complexity $\mathcal{O}(n^2)$ where n is the number of original clauses + number of conflict clauses which means it is reasonably fast.

2.6.1 Proof Formats: RUP, DRUP and FRAT

RUP proofs were first introduced in 2003 [24]. One issue with this approach is that it is slower to check, since for each new clause one must check all previous clauses.

A DRUP proof first introduced in 2013 [26] remedies the issue by adding deletion clauses no longer used by the solver. This offers a significant speedup to checking the proof using a small proof-checker. However, one of the goals is of this project is to allow for easy verification of proofs to *humans*. As such, we would like to know the *specific* reason why a clause was derived not just that it can be derived from *all* previous clauses.

A recent proof format called FRAT [5] offers *exactly* this functionality without compromising on the performance. Whenever a conflict clause is added, a justification is provided in terms of which *specific* clauses were used to derive it. See figure 4 for a FRAT style proof.

2.6.2 Minimum Unsatisfiable Subset (MUS)

Why Use MUS? After a formula has been converted into CNF and a SAT Solver has deemed it unsatsfiable, one might like to know *why* it is unsatisfiable. In 2.6, we saw an approach which is guaranteed to lead us to a *proof* for unsatisfiability. However, the proofs produced by the approach outlined in 2.6 are usually too long to verify by humans. Indeed, this was the case for this project.

Another way to answer this is by looking at the minimum set of disjunctive clauses which create a contradiction. Since our formula has been converted into *conjunction* of disjunctions, if a subset of disjunctions is unsatsfiable, then the whole formula is unsatsfiable. Since “false and c1 and c2 and ...” will evaluate to false. Where, c1 and c2 are disjunctive clauses.

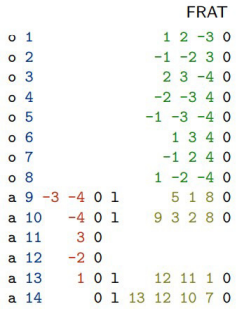


Figure 4: This image was taken from [5]. It illustrates the FRAT proof format. Lines beginning with o means that the clause was in the original CNF file. Lines beginning with a means that the clause was added by the sat solver. *Importantly,* "l" provides the clauses which were used to derive it. I.E. Line 9 adds "-3 or -4" and is derived by conjoining clauses with indexes 5,1,8 I.E.: "-1 or -3 or -4", "1 or 2 or -3", "1 or -2 or -4". combining 1 & 8 we get "1 or -4", combining (1 & 8) with 5, we get "-3 or -4". As required. This can It also has deletion and finalising clauses, but these are not useful for this project's use-case.

For example, suppose your CNF formula has the following form:

1. A or B
2. not A
3. not B
4. C
5. A or C

Note: the actual formula would be "(A or B) and (not A) and (not B) and (C) and (A or C)". The MUS for the formula would be clauses 1-3. I.E. (A or B) and (not A) and (not B) since no matter what you assign to A or B, at least one clause from clauses 1-3 will evaluate to false.

Calculating MUS A naive approach to find the MUS would be to iteratively add clauses until they create a contradiction. The time complexity of this approach is astonishingly high: $\mathcal{O}((c * k)!)$ where c denotes the number of disjunctive clauses, and k denotes the time taken to check each subset for SAT. We know from before that k is exponential with respect to number of atoms.

This approach can therefore only be used on MUSes which contain very few clauses. Modern MUS solvers include this as an option [8], but they mostly rely on heuristics to find an "approximate" MUS. I.E. they find a *small* set of clauses which is unsatsfiable, but they do *not* guarantee that it is the smallest set of unsatisfiable clauses.

Summary In summary, we have introduced two key techniques for showing that a SAT formula is unsatisfiable: extracting the MUS and using a proof-logger to verify the impossibility result produced.

2.7 Current Approaches

Now that we have gathered the required background knowledge on what are some of the problems in social choice, argumentation theory, and in explaining computer aided proofs we are now ready to discuss some of the more recent techniques in these fields.

2.7.1 SAT Solving in Social Choice

The method of using SAT solvers to verify the base case for famous theorems in social choice was first applied by Tang and Lin (2008) [32]. They discovered new proofs for Arrow's Impossibility and Sen's and Muller-Satterthwaite's Theorems. Using SAT solvers in social choice is quite widely adopted. Several works using them have been proposed such as [12] where SAT solvers were used to study the no-show paradox (when voters are better off not voting at all), or [14] to prove that "There is no majoritarian and Pareto optimal set-valued voting rule that satisfies Fishburn-strategyproofness when m $\geq$ 5, and n $\geq$ 7." Quote taken from [21].

2.7.2 Explainability of SAT proofs

The method of using SAT solvers was later adapted by Felix Brandt and Christian Geist [13] to use MUS as the key way to explain the unsatisfiability result. This allowed for proofs to become more human readable.

The MUS may still be difficult to read thus a visual aid of a graph diagram of the impossibility result was introduced by Brandt et. al.[4]. The diagram's edges represent inter-profile axiom's. I.E. axioms which connect the outcomes of more than one profile for example, strategy-proofness. The nodes in the diagram represent a profile. To interpret the diagram, one starts at the bottom and works their way up the diagram iteratively eliminating possible winners of the elections until none are left. See 5 for what a proof diagram of a MUS might look like.

Intuitively, providing a MUS diagram only gives a brief overview of the proof. Much of the reasoning behind why the proof works is left to the reader. This approach works well for relatively small muses, but as we will see in section 5.2.3 it quickly becomes impractical when considering complicated theorems whose MUS contains hundreds of nodes.

Finally, in a 2018 paper Brandt et. al. [15] introduce the idea of extracting the stack trace to provide further explainability for each step of the proof. In terms of SAT, each step of the proof is a new clause that consists of a combinations of previous clauses. For example, combining the two clauses (not

	voter 1	voter 2	voter 3
R_α	$a \succ b \succ c$	$b \succ a \succ c$	$c \succ a \succ b$
R_β	$a \succ b \succ c$	$b \succ a \succ c$	$a \succ b \succ c$
R_γ	$a \succ b \succ c$	$b \succ c \succ a$	$c \succ a \succ b$
R_δ	$a \succ b \succ c$	$c \succ a \succ b$	$c \succ a \succ b$
R_ϵ	$b \succ a \succ c$	$b \succ c \succ a$	$c \succ a \succ b$

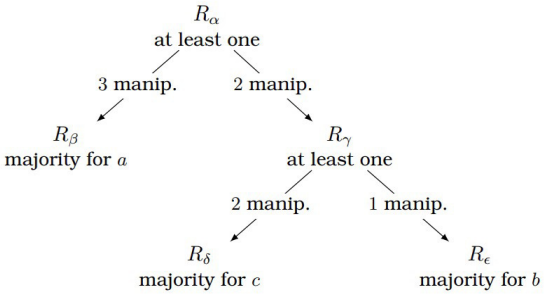


Figure 5: This image was taken from [21]. It illustrates Brandt's proposed approach of translating MUS into a proof diagram. On the left, we see voter profiles. On the right we see a a directed graph. For example, the graph tells us that profile γ can be manipulated by voter 1. to produce profile ϵ. Therefore, we can conclude that γ cannot be won by candidate b because this would mean that the election does not obey strategy-proofness. For a more elaborate description see [21]. This shows as illustration of what we are seeking to generate automatically.

A or not C) and (C) will result in a new clause of (not A). The procedure for this has been discussed in more detail in 2.6.

2.7.3 SAT Solving in Argumentation Theory

SAT Solving in Argumentation Theory mostly focuses on finding various semantics. Since 2015, the International Competition on Computational Models of Argumentation (ICCMA) has been running a bi-yearly competition which seeks to find the fastest way of calculating or counting preferred, complete, grounded and stable semantics.

Various approaches have been developed over the years many of which incorporate SAT solving. Either directly, by translating the problem into SAT [29], or by using a dynamic programming SAT solving technique [22]. In fact, the last approach won the 2021 competition.

3 Approach

In this section I will outline and justify the approach that was taken to encode three social choice theorems into SAT as well as *why* they were encoded into SAT.

This section will serve as a pre-requisite to explaining how a similar encoding strategy can be applied to a theorem in argumentation theory. This will show that the proposed method is versatile; however, we still need a way to evaluate its performance which is what will be covered next. After performance is evaluated,

the **key** finding of this report will be discussed. Namely, the approach taken to turn SAT proofs into human readable format.

3.1 Translating Theorems in Social Choice to SAT

3.1.1 Aim

This section closely follows [21] where an approach of using SAT solvers to verify the base case of theorems in social choice was used in combination with an inductive argument to complete the proofs of the theorem.

The aim is to verify whether a possible voting rule exists which obeys some subset of the properties outlined in 2.1.2. It is important to note that we are only interested to show whether such voting rule exists for a *specific case*. For example, for three voter elections. This means we *cannot* prove theorems entirely using this method. We can only disprove them by finding a counter-example. This limitation is further discussed in 5.4.1.

3.2 Why use SAT?

A naive approach would be to generate all possible voting rules and check one-by-one whether each voting rule satisfies all of the properties. This approach would not work because of the very large search-space. To give an estimate of just have large the search space is, consider an election with *only* three voter v_1, v_2, v_3 and three alternatives - A, B, C. Furthermore, assume that only strict linear orders are valid ballots and that only a single alternative is declared the winner of any election.

There are $3! = 6$ possible ballots for every voter. Since there are three voters each of which can choose from 6 possible ballots, there are $6^3 = 216$ *profiles* that need to be assigned a winning alternative. One such assignment would be assigning all profile to be won by candidate A. Another, would be to assign profile 1 be won by candidate B, and rest by A. It is then easy to see that we would need to check 3^{216} voting rules! This number has **104** digits.

SAT allows us to encode the constraints and uses clever algorithms such as DPLL and CDCL (covered in section 2.5.2) to *efficiently* check the search space.

3.3 Encoding Into SAT

This subsection will outline the general approach taken to turn checking for an existence of a voting rule into a SAT formula. Each of the subsections will outline how to encode the properties required for each of the three social choice theorems.

To choose the base case Before we the theories, we need to make a design decision regarding the base-case. We want the base case to have enough "freedom" to allow for a contradiction, but not too much freedom because of the exploding search space even when using SAT solvers. A reasonable base-case

is to choose 3 voters $V = v_1, v_2, v_3$ and 3 alternatives $A = \{\text{A, B, C}\}$. This implies 216 possible profiles $P = \{(A \prec B \prec C,\ A \prec B \prec C,\ A \prec B \prec C),\ (B \prec A \prec C,\ A \prec B \prec C,\ A \prec B \prec C)\ ...\}$. This base case will be used throughout this project unless specified otherwise.

For single winner elections, a voting rule F will take a profile $p \in P$ and assign it an alternative $a \in A$. For multi-winner elections, the voting rule would assign a subset of A or a linear order from A depending on what the theorem states.

In this work, I will use the single-winner encoding for simplified GST and full GST. For Arrow's impossibility theorem the voting rule will assign a *strict* linear order.

For simplified and full GST, we can represent the voting rule F in SAT by creating three atoms per profile. Each atom corresponds to a profile being won by an alternative. So if we had four alternatives, then we would create four atoms per profile.

This can be written as $F_{p,a} = l$ where p is some linear profile, a is some winning alternative and l denotes the atomic variable corresponding to assigning profile p the winning candidate a. It is easy to see that there will be 216*3 = 648 atomic variables in total. When l is true, it means that profile p was won by candidate a.

Next, we need to introduce constraints which correspond to voting rule properties on those variables. This process is explained in the following sections. Once we have encoded all of the properties for our theorem we can use conjunction to join them all together into the final theorem. Seeing as we want our voting rule to obey *all* properties.

3.4 Simplified GST

Simplified GST states that there is no *resolute* voting rule F which satisfies *strategy-proofness* and *majority* criterion. Full GST directly implies simplified GST, but simplified GST serves as an illustrative example of the technique. See section 2.1.2 for the definition of strategy-proofness and other properties of voting rules.

AIM: We seek a set of rules which can be written in CNF which *forbid* assignments of F which do not satisfy the Simplified GST properties. This will allow us to run a SAT solver which will either return a voting rule which obeys all these properties or inform us that no such rule exists by returning "Unsat".

Encoding Resoluteness A voting rule is said to be resolute if there exists at most one winner given a profile. Remember, each profile is encoded as three atoms - one for each alternative. For example, if $p = (A \prec B \prec C,\ A \prec B \prec C,\ A \prec B \prec C)$ it is encoded as atoms: $l_{p,A}$, $l_{p,B}$, $l_{p,C}$. One way to think about it is that we never want any pair of atoms to both be true. To forbid a pair of atoms from being true in CNF, a disjunctive clause such as: (not($l_{p,A}$) or not($l_{p,B}$)) can be introduced. The clause will be *false* only if both $l_{p,A}$ and $l_{p,B}$ are true. We now need two more clauses forbidding $l_{p,A}$ & $l_{p,C}$ from both

being true and $l_{p,B}$ & $l_{p,C}$ from both being true. We can connect all three of the disnjunctive clauses using a conjunction. I.E. (not($l_{p,A}$) or not($l_{p,B}$)) *and* (not($l_{p,A}$) or not($l_{p,C}$)) *and* (not($l_{p,B}$) or not($l_{p,C}$)).

Finally, we need to repeat this procedure for all profiles p. In total, we will generate 216*3 = 648 clauses.

Encoding At Least One While this rule is not mentioned in the definition of Simplified GST, it is implied. We want our elections to have *at least one* winner given any profile p. This is much easier to encode: a clause like $l_{p,A}$ or $l_{p,B}$ or $l_{p,C}$ is sufficient to encode it for profile p. It now needs to be repeated for all other profiles generating a total of 216 disjunctive clauses.

Encoding Majority Majority criterion is satisfied if when at least two voters place some candidate A as their first choice in their ballots. Further suppose that S_A, S_B, S_C are sets of profiles where a majority of people voted for alternative A, B, and C respectively. For example when $p = (A \prec B \prec C,\ A \prec B \prec C,\ B \prec A \prec C)$, $p \in S_A$. The majority criterion can be encoded into logic by looping through all profiles which are in S and setting each one to the winning candidate. I.e.

- $\forall p \in S_A$, $l_{p,A} = ture$ and
- $\forall p \in S_B$, $l_{p,B} = ture$ and
- $\forall p \in S_C$, $l_{p,C} = ture$

Encoding Strategy-Proofness Remember strategy-proofness refers to the inability of some voter to manipulate the election by misrepresenting their true preferences. It is useful to think in terms of original election where the voter *does not* misrepresent their views and a *manipulated* election where they do.

We want to select all pairs of profiles, which only differ by a single voter's ballot. For example, (p_1,p_2) would be a valid pair, where $p_1 = (A \prec B \prec C,\ A \prec B \prec C,\ A \prec B \prec C)$, $p_2 = (A \prec B \prec C,\ A \prec B \prec C,\ B \prec A \prec C)$. In this case it is v_3 who attempts to manipulate the election by misrepresenting his true preference of $A \prec B \prec C$ as $B \prec A \prec C$. In general, we can denote the profile with i^{th} voter's true preferences as p_{org_i} and the profile with i^{th} voter's manipulated preferences as p_{manip_i}

If the original election was won by candidate C ($F_{p1,C} = true$), and the manipulated election was won by candidate A ($F_{p1,A} = true$), then the manipulation would be successful since v_3's original ballot ranks A higher than C. In general, we want "if $p_{org_i,N}$ then not($p_{manip_i,M}$) " where N and M are both alternatives and N is more preferred by voter i's original ballot than M. To convert the if rule into CNF, we can use law of implication. I.E. "not($p_{org_i,N}$) or not($p_{manip_i,M}$) ". Once we have generated all clauses of the form not($p_{org_i,N}$) or not($p_{manip_i,M}$) we need to use conjuntion to join them all together.

Summary In summary, we have seen how to encode Resoluteness, At least one, Majority, and Strategy-proofness into SAT for the case of 3 voters and 3 candidates. All that's left to do is to apply a SAT solver and see whether the formula is satisfiable.

3.5 Full GST

Full GST will borrow strategy-proofness, resolutness and at least one properties from simplified GST, but will do away with majority criterion. It will introduce two new criterion, namely, non-imposition and non-dictatorship.

Non-imposition or citizen sovereignty simply states that all candidates should have a chance of winning the election. By chance I mean that there should be some profile (set of ballots) that allows that candidate to win.

Luckily, this is very easy to encode in SAT. If the property holds, then at least one of the atoms corresponding to candidate A winning must be true. I.E. $F_{p1,a}$ or $F_{p2,a}$ or $F_{p3,a}...\forall p, p \in P$. Where P denotes the set of profiles and a is alternative A. We can add additional two clauses for candidate B and C. Finally, we can join the three clauses together using conjunction because we want *all* candidates to win at least one profile.

Non-dictatorship This property states that some special voter *should not* be a dictator. In other words, their most preferred candidate should *not* always win.

To encode this we need to collect all atoms corresponding to v_1's most preferred candidate winning and ensure that at least one of the atoms is false. Let K_1 denote the set of atoms for which the first voter's most preferred candidate wins. Then to encode this property into SAT, we can add a clause "not(k_1) or not(k_2) or not(k_3) ..."$\forall k, k \in K_1$. We then repeat the procedure for voter 2 and voter 3 and join the three disjunctive clauses using conjunction.

Summary This section has outline how to encode the full GST theorem's base case into SAT. Full GST consists of 5 properties which are non-dictatorship, non-imposition, strategy-proofness, at least one and resoluteness.

3.6 Arrow's Impossiblity Theorem Encoding

This is arguably the most famous theorem in social choice. It states that no voting rule can obey *all* five properties. These are non-dictatorship, Pareto efficiency, Independence of Irrelevant Alternatives (IoIA), unrestricted domain and social ordering.

The good news is that unrestricted domain (ability for all participants to submit any preference) and social ordering (ensuring that the output of the voting rule is acyclic) are encoded already by the way we have set up the SAT approach.

Note: for Arrow's Impossibility theorem the encoding is slightly changed. Instead of the voting rule providing a single winner, it provides a *strict* linear order of winners. I.E. It tries to aggregate the voter ballots in such a way that there is a first place, second place and third place candidates. For example, the voting rule could map the profile $p = (A \prec B \prec C,\ A \prec B \prec C,\ B \prec A \prec C)$, to $A \prec B \prec C$. Meaning candidate A came first, B second and C third. This means, every profile now has 6 atoms each corresponding to a permutation of {A, B, C}.

Pareto efficiency states that if everyone prefers candidate A over B, then B should rank higher than A. We can check whether each atom obeys this property by checking whether there exists a candidate who is more preferred than some other candidate in the corresponding atom's profile. If such a pair of candidates exists, *and* the atom has assigned the *unanimously* preferred candidate a lower rank than the less preferred candidate, we want to forbid this atom from becoming true. I.E. we add the negation of the atom as one of the CNF clauses. This procedure needs to be repeated for all atoms.

IoIA states that the ranking produced by a voting rule for any two alternatives A and B should *only* depend on their relative ranking given by the voters. This is somewhat challenging to encode. We can collect all profiles p for which all voters rank alternative A higher than alternative B. If the voting rule decides that A wins in any of the profiles, then A must win all of the profiles. I.e. "if a_1 then (a_2 and a_3 ...)" where a_n corresponds to all atoms which relate profiles where all voters preferred candidate A over B and candidate A won the election.

Note, this does *not* encode that candidate A *must* win the election if all candidates prefer A over B.

Similarly, all profiles where v_1 prefers B over A, but v_2 and v_3 both prefer A over B *must* also have the same winner. We can use the same procedure for all 8 combinations of the 3 voter relative preferences between A and B.

To finish encoding, we need to ensure that we have encoded it for all pairs of candidates, not just A and B. (We need to encode this for (A and C), and (B and C) and then join the clauses together using conjunction.

3.6.1 Extracting and Visualising the MUS

At this point, we have seen how to encode theorems into social choice into CNF which can be understood by modern SAT solvers. The SAT solver will either produce a valid assignment of atoms or return "UNSAT" meaning there is no assignment of atoms which can satisfy all the constraints. This means we have verified a base-case to one of the theorems from the above section.

The aim of this section is to *visualise* the contradiction. To do that we will take the output of the MUS extractor and put it into a graph that looks something like figure 5.

The key "ingredients" of the graph are a *labelled* set of **profiles** and labelled set of **arrows** connecting the profiles.

Each profile is labelled according to the *property* the clause is a part of encoding. For example, the clause "$l_{p,A}$ or $l_{p,B}$ or $l_{p,C}$" is encoding the property of majority rule.

Each arrow is labelled by voter who can *manipulate* the outcome of the election. The starting point of the arrow is a profile which contains the person's true or original ballot, and the end point of the arrow is the profile containing the manipulated ballot.

Extracting the MUS Modern MUS extractors such as MUSER2 [8] will return a set of indexes of disjunctive clauses which form the MUS. We can then use those indexes to look up the clauses. Finally, we can check what type of clause it was and which profiles were referred to by the clause by looking at the *atoms* in the clause.

For example, the MUS output might be "2 5 8 0". 0 indicates end of output, so we only need to look at clauses on lines 2, 5 and 8. Suppose clause on line 2 is "4 5 6 0". This clause states that atom "4 or 5 or 6" must be true. You might recognise this clause from the "at least one" property. The atoms 4, 5, 6 all refer to the same profile which needs to be assigned *at least one* winner.

This gives us enough information to make the required plot.

3.7 Translating Theorem 3 in Argumentation Theory to SAT

We have seen how to encode theorems in social choice into SAT. As a part of evaluating just how widely applicable using SAT solvers to verify base-cases of theorems is, it was decided to apply it to Theorem 3 in argumentation theory. It may help the reader to refer to Section 2.3 for a reminder of what each property in Theorem 3 means.

To verify the theorem's base case, I will be using 3 agents ag_1, ag_2 and ag_3. I will also be *fixing* the AF to the one in the figure 6:

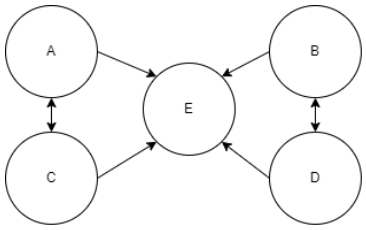


Figure 6: AF that serves as base-case for Theorem 3 in argumentation theory.

We will be using a similar general strategy to look for a contradiction in the outlined base-case as we did for argumentation theory. First, we need to generate all labellings agents can give. Remember, agents may only provide *complete* labellings. Secondly, we need to generate all possible aggregated labellings. I chose to only generate *complete* aggregated labelligns because it is

one of the properties of Theorem 3. The alternative, less efficient approach would have been to generate all aggregated labellings and then add in CNF clauses forbidding the atoms corresponding to incomplete labellings from being true.

Finally, we are ready to represent an aggregation procedure (equivalent to a voting rule in social choice) explicitly. The aggregation procedure will take in a 3-tuple of agent labellings (a *profile*) and assign it a labelling. Each pair (profile, aggregated preferences) is assigned an *atom*. When an atom is true, it means that our "voting rule" will aggregate the agent preferences to aggregated preferences.

Note, since we are fixing the AF, there is no need to encode Isomorphism because AF-independence already implies it. Collective rationality was implicitly when generating the voting rule by not generating any atoms which correspond to the aggregation rule assigning an incomplete labelling.

We are now ready to encode the properties into CNF.

At Least One & Resolute Similarly to Social choice, although not stated explicitly we would like our aggregation procedure to output exactly one labelling. Encoding at most one and at least one is done in the same way that it was done in social choice.

Anonymity To encode anonymity, I find all atoms corresponding to a permutation of a profile which assign the same winner. I then add a rule saying that if any one of the atoms is true, then all atoms are true. For example, suppose you have a profile p_1=(A, B, C) where A, B, and C are complete labellings given by agents 1, 2 and 3 respectively. Suppose you have an atom which corresponds to p_1 being aggregated to have the labelling D. We then find all atoms (a_1,a_2,a_3...) corresponding to a permutation of p_1 (for example (B, A, C)) being assigned label D. Finally, we can encode the anonymity constraint by encoding the constraint if a_n then a_b where a_n, $a_b \in$ atoms. In CNF, this would be translated as "not a_n or a_b".

Unanimity loosely translates to Pareto efficiency in social choice. To encode it, we go through all atoms and check if the profile referred to by the atom contains any argument that is unanimously labelled. If so, we check if the aggregated winner referred to by the atom has the same label for that argument. If it does not, we can exclude this atom by adding a CNF clause "not atom" for all such atoms.

AF-Independence This states that each argument can be aggregated individually. Similarly, to IoIA in social choice, we may split all atoms into groups. More specifically, each node in the AF will have several sub-groups. Each subgroup corresponds to one of 27 different ways of labelling that node by each agent. For example, one group will correspond to node A. Node A can be labelled by each agent as being *in*, *out*, or *undec*. Since we have 3 agents, there are

27 ways of labelling the argument. Any atom will fit into one of the subgroups and thus we can associate a subgroup with a set of atoms.

What AF-independence tells us is that if any one of the atoms belonging to the subgroup is true, then the entire subgroup must also be true. To encode this into CNF, suppose S is the set of atoms for subgroup one. For all pairs (s_1,s_2), where s_1,$s_2 \in S$ we can write a CNF clause stating "not s_1 or not s_2".

3.8 Solving the SAT and Evaluating the Performance

In previous sections, we have seen how to translate numerous theorems into SAT. However, a practical consideration one needs to make is to choose a SAT solver that is fast for their use case. Modern SAT solvers often have variable performance on different tasks.

Most standard tasks such as performance on the pigeonhole principle are covered in the proceedings papers of SAT competitions [39]; however, they do not include the performance for finding impossibility theorems in social choice. This will be the main *motivation* for this section. Although, interestingly, one of their benchmarks is calculating preferred extensions in argumentation theory.

I have chosen three SAT solvers which I will evaluate on the three theorems in social choice. All tests will be repeated 10 times and mean time to solve as well as the standard deviation will be recorded.

Due to strict time limitations, more challenging base-cases which have more voters and alternatives will not be checked.

3.9 Translating SAT Proofs to Natural Language

3.9.1 Motivation and Limitations of Previous Approaches

In the previous sections we have seen different ways of making SAT proof human readable. Either by proof extraction in section 2.6 or via using MUS 2.6.2 or by turning that MUS into a diagram 3.6.1.

It can be very difficult to interpret the extracted MUS because it is using clauses which refer to *atoms*. Each atom corresponds to a voting rule, but this information is not embedded into the MUS. To give *meaning* to *atoms* from clauses in the MUS one has to look them up. This is not very practical. *Note: it is also possible to use grouped CNF (GCNF clauses) where each group refers to a different property. However, this still does not solve the problem of knowing which specific profile the clause is referring to.*

This issue is remedied by the tree graph representation since it looks up the meaning of the clauses and displays them in a visual way. However, it is very difficult to make sense of large graphs. Verifying the large graphs *is not* trivial because it is hard to verify a small part of the graph independently.

3.9.2 Solution

Ideally, we are looking for a description similar to what is outlined in most papers as *explanation* of their proof. See figure 7 to get a sense of what we

would like our proof to look like.

Theorem 13.2 (Base Case). *For $m = n = 3$, there is no voting rule that satisfies strategyproofness and the majority criterion.*

Proof. Suppose f is a voting rule satisfying both axioms. We proceed in a bottom-up fashion, establishing which value f must take for each profile, and find that there is no possible value $f(R_\alpha)$ for the root node, contradiction.

Consider the profile R_γ, where $f(R_\gamma)$ needs to take some value. If $f(R_\gamma) = a$, then voter 2 can manipulate to obtain profile R_δ. By the majority criterion, $f(R_\delta) = c$, and so we have $f(R_\delta) \succ_2 f(R_\gamma)$, and this was a successful manipulation, contradicting strategyproofness of f. Hence $f(R_\gamma) \neq a$.

If $f(R_\gamma) = c$, then voter 1 can manipulate to obtain profile R_ϵ. By the majority criterion, $f(R_\epsilon) = b$. Thus $f(R_\epsilon) \succ_1 f(R_\gamma)$, and this was a successful manipulation, contradicting strategyproofness of f. Hence $f(R_\gamma) \neq c$. Hence $f(R_\gamma) = b$.

By the majority criterion, $f(R_\beta) = a$. Now consider profile R_α. In case $f(R_\alpha) = b$, then voter 3 can manipulate to obtain profile R_β. Since $f(R_\beta) = a$, this would be a successful manipulation for voter 3. Hence $f(R_\alpha) \neq b$.

Figure 7: This proof was taken from [21]. It is a natural language description of the proof in figure 5. This is something we would like to generate alongside our proof diagram.

To do this one can use a SAT solver with proof logging on the MUS. While the proof produced could be quite lengthy, it will be shorter than the one produced by using *all* clauses.

Secondly, it has a very nice property. Each part of the proof can be verified "independently". Suppose the proof is 1000 clauses long. We can ask a person to verify clauses 1 to 100, another to verify 101 to 200 ... etc.

Verifying "raw" clauses is not intuitive and requires that the user has a good understanding of logic. Furthermore, each atom of the clause needs to be looked up for it to have meaning (which profile it refers to and what property is the clause encoding).

Each step of the SAT proof will be a derived clause followed by clauses which were used to derive it. The proof will stop when the empty clause is derived indicating a contradiction. By translating all of the clauses into natural language, the proof step should become much more intuitive.

See figure 8 for a high level overview of the approach.

Suppose your SAT solver derives a clause "4 5 0" by combining clauses "4 5 6 0" and "-6 0". It may seem difficult to understand at first especially for those unfamiliar with SAT or logic.

However, we can mechanically translate this into natural language. A possible tranlation might look something like this: "Let $p_1 = (A \prec B \prec C$, $A \prec B \prec C$, $B \prec A \prec C)$. Using at least one property, we know that p_1 must be won by either candidate A or candidate B or candidate C. We have previously derived on step 131 of the proof that p_1 cannot be won by candidate C. Therefore, we can conclude that p_1 must be won by either candidate A or candidate B". *Note: step 131 is just used as an example here.*

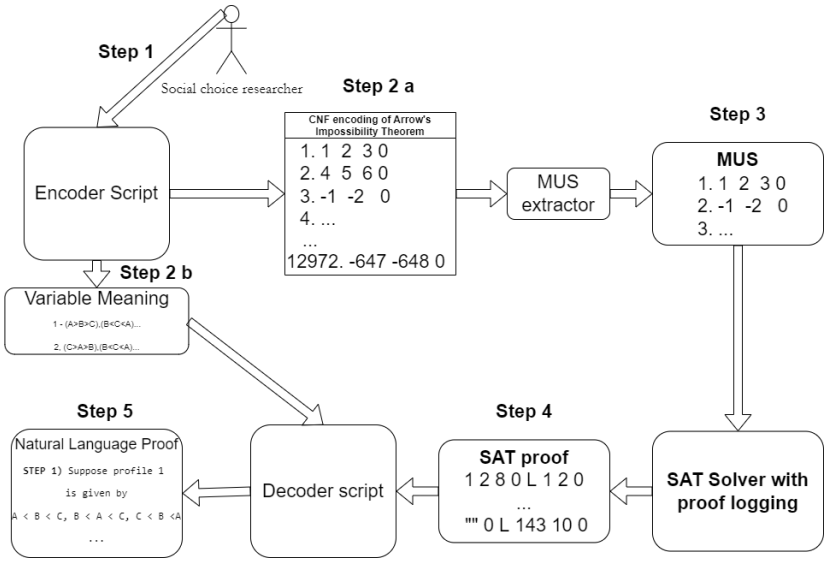


Figure 8: This figure shows the proposed solution. At step 1 a social choice researcher encodes the properties for which he would like to check the existence of a voting rule. The script translates the high-level specification of the properties into CNF formula (Step 2 a) It also saves what the meaning of each atom is and what property each clause is trying to encode in Variable Meaning file (step 2 b). After the theorem is turned into CNF, it is given to a MUS extractor which extracts the MUS (Step 3). Note in practice, often the MUS extractor returns the line numbers of clauses and not the clauses themselves so an additional step between 3 and 4 is sometimes needed. We now need to prove that the MUS is unsatisfiable using a SAT solver with proof logging (Step 4). This proof will be de-coded by the a proof-decoder script which takes variable meaning and SAT proof-logs as inputs and returns a natural language proof of the theorem (Step 5).

This is done by looking up all of the profiles which are used in the justifying clauses. Followed by looking up all of the candidates used in each clause. Then we can wrap it in natural language.

Summary A simple mechanical way of translating SAT proofs into natural language to verify impossibility result base-cases for theorems in social choice has been proposed. Special attention has been paid to ease of checking and allowing people unfamiliar with propositional logic to still be able to understand the impossibility result produced.

4 Implementation

So far, an approach to finding proofs for impossibility theorems in social choice and argumentation theory has been outlined. The approach glosses over specific implementation details which are covered in this section. It may help to refer to figure 8 for a high-level outline of what we are trying to do.

Prerequisites subsection will cover the "workspace" needed to get started using SAT solving and extracting MUSes. The workspace loosely translates to Steps (3 and 4) in figure 8. **Encoding the Theorems** section will cover the specific implementation of each axiom used in social choice. This corresponds to Step 2 a and Step 2 b in figure 8. Finally, the **Human Readable Proofs** section will cover the implementation for making proofs human readable either by creating a graph from the MUS or by generating justification of a proof in English. This corresponds to Step 5 in figure 8.

4.1 Prerequisites (OS)

Cutting edge SAT solvers are usually developed to run on Linux. Indeed, all SAT solvers which were evaluated, namely, MiniSAT [35], lingeling [10], sat4j [9] *only* contain build instructions for *Linux* systems. Finally, the MUS extraction tool MUSER2 [8] also only provided instructions to be built on Linux. This means the first step is to install **Linux OS**.

Installing Ubuntu I chose to install Ubuntu which is a beginner friendly Linux distribution. Since I only have access to a Windows machine, I took advantage of the new subsystem for Linux feature built into Windows. Finally, I installed Ubuntu application from Microsoft Windows Store [31]. This gave me access to a Linux terminal while allowing me to continue to use Windows.

Optional - Using a GUI While installing Ubuntu is sufficient for installing the SAT solvers, I took advantage of *LXDE* [30] which is a desktop environment. To actually see the desktop, you need to connect to it like you would to a remote PC. XLaunch is a Windows application which allows you to do this easily. By the end of this step, I had access to a desktop environment so I could issue commands such as copy using the mouse instead of using the terminal. This step adds convenience and saves time.

Note: during initial stages of implementation I was not aware of a Python library called PySAT [28] which integrates MiniSAT and a couple of other solvers directly into Python. This allows them to run directly on Windows, Mac or Linux. It is missing a MUS extractor so this section is still required.

4.1.1 SAT solvers & MUS extractor

Once we have setup a working Linux environment, installing the SAT and MUS solvers was straight forward.

- To install MiniSAT, it was sufficient to run "sudo apt install minisat".

  After this, you can use the minisat solver by typing "minisat" inside your terminal.

- To install lingeling, I used "git pull https://github.com/arminbiere/lingeling" followed by "./configure.sh && make" command.

  This builds the lingeling solver. To use it, simply navigate to where it was installed and type "./lingeling".

- To install sat4j solver, source code was downloaded from `https://gitlab.ow2.org/sat4j/sat4j/-/releases/`.

  Then, "cd ../dist/CUSTOM" where .. is the install directory. Finally, to run the solver one can issue the command "java -jar sat4j-sat.jar"

- To install MUSER2, I used "git pull https://github.com/meelgroup/muser" followed by "cd ./src/tools/muser2 && make".

  Now MUSER2 is available to run inside the install directory via the command "./muser2"

4.1.2 Python & Libraries

Now that we have access to SAT solvers, we need a way to encode the theorems into CNF. We do not want to manually write out all properties for each theorem by hand. Python will be used to *visualise* output of the MUS and to generate natural language proof descriptions.

I chose to use Python because it the programming language I was the most familiar with and it is one of the most popular high-level programming languages. While Python is an interpreted language which means that it will be slower than something like C++, it is not important for our use-case because encoding the axioms takes polynomial time, but solving the SAT takes exponential. This means performance for encoding does not play a large role and thus we can afford the conveniences of a higher-level language.

Python 3.9.6 [42] was used with two external libraries. Namely, matplotlib [27] which is a standard library to generate plots and networkx [25] which is a library specifically made to display *graphs*. Both were used to generate and plot graphs from the MUS.

To install Matplotlib, the command "pip install matplotlib" was used. To install networkx, the command "pip install networkx" was used.

Other libraries such as pickle [41] (used for storing the variables) and itertools [40] (used for generating permutations of preferences) which come pre-installed with Python were also used.

Summary This section has provided us with relevant tools to translate a high-level description of properties into CNF, make plots from MUS and interpret the proofs produced by SAT solvers.

4.2 Encoding the Theorems

Now that we have set up all of the necessary tools, we can begin work on creating a script which encodes relevant properties for theorems in social choice or argumentation theory.

An important consideration is to keep in mind our target audience, namely, social choice and argumentation theory researchers. One of the more common use-cases is to check whether some *subset* of properties allows for an existence of a voting rule or whether there exists an aggregation procedure for an AF. Therefore, instead of encoding a specific theorem, it might be more reasonable to provide the user with the ability to specify the axioms which they wish to use.

4.2.1 Social Choice

In my current implementation, I assume very little programming knowledge required to use the script. All the social choice researcher has to do, is to set variables to true or false for the properties they want to use to check an existence of a voting rule. See figure 9.

```
##arrow uses: rules_one, rules_resolute, non-dictatorship, pareto efficiency, BUT NOT imposition! Source: https://www.investopedia.com/ter

use_rules_one           = True
use_rules_resolute      = True
use_pareto_efficiency   = True
use_rules_dictatorship  = False
use_IoIA                = True  #IoIA - independence of irrelevant alternatives
use_rules_majority      = False
use_rules_strategy      = False
use_rules_imposition    = False
```

Figure 9: This figure shows all of the properties which can be encoded by the Encoder Script. To change which properties to use, simply set the corresponding variables to true/false. The properties which encode the base case for Arrow's impossibility theorem are selected.

The second thing which social choice researchers may wish to adjust is the number of alternatives and number of voters. Due to strict time constraints, this was not implemented, but is relatively simple to do.

I have shown how properties can be specified by social choice researchers. The next thing to do, is to show how they are implemented. It may help the reader to refer to Sections 3.4, 3.5 and 3.6 for a reminder of what each property encodes.

Before we do that, we need to define how to represent the voting rule which will be used to encode all properties. Finally, I will be using the DIMACS encoding covered Section 2.5.2 in my implementation.

- **possible_voting_rules**: this is a list which contains 648 2-tuples. The first element of the tuple is a profile and the second is the alternative who won. For example, running possible_voting_rules[0] will return (($A \prec B \prec C$, $A \prec B \prec C$, $A \prec B \prec C$), A) where ($A \prec B \prec C$, $A \prec B \prec C$,

$A \prec B \prec C$) is the profile alternative and A is the winning alternative. For multi-winner elections possible_voting_rules[0][1] would return $A \prec B \prec C$ instead of just A.

Each possible_voting_rule will correspond directly to an *atom* when encoded into SAT. Note that unique profiles are next to each other, so voting rules possible_voting_rules[0], possible_voting_rules[1] and possible_voting_rules[2] would return the same profile as their first element, but would return alternatives A, B, and C respectively as their second element.

Lastly, to get voter 2's ballot for atom 237 we can run:

possible_voting_rules[237-1][0][2-1] and this would return $A \prec B \prec C$.

237-1 accesses the 237^{th} atom. 0 indicates we want to look at the voter ballots, and finally 2-1 indicates we want to look at the ballot submitted by voter 2 (we subtract one because 0 indexing is used). To get the top choice of voter 2 for atom 237, we would issue:

possible_voting_rules[237-1][0][2-1][0]

and this would return: A

Note: I use pseudo-code instead of providing the actual code because I think that pseudo-code will be more informative than the specific implementation which will probably be confusing for the reader. If the reader so chooses, they can check the source files for the specific implementation.

At Least One Since there are three alternatives for every profile, we would like to generate a conjunctive clause for every profile by using disjunction on all alternatives. The algorithm below loops through all possible voting rules and uses the fact that the same profiles are located next to each other in possible_voting_rules to ensure that there is at least one winner per each profile. It returns a list of "all_clauses" where each element of the list is clause is a disjunction of atoms. It is assumed that later all_clauses is concatenated using conjunction and encoded and saved using DIMCAS encoding.

Algorithm 1 At Least One - Social Choice

Require: possible_voting_rules
  $Current\_profile \leftarrow possible\_voting\_rules[0][0]$
  $Current\_clause \leftarrow$ ""
  $all\_clauses \leftarrow [\,]$
  for i $\leq$ length(possible_voting_rules) **do**
    if $Current\_profile = possible\_voting\_rules[i][0]$ **then**
      $Current\_clause \leftarrow Current\_clause$ + i + " "
    else
      $all\_clauses$.append($Current\_clause$)
      $Current\_clause \leftarrow$ i
      $Current\_profile \leftarrow possible\_voting\_rules[i][0]$
    end if
  end for
  return $all\_clauses$

Resolute This states that two alternatives cannot both win the same profile. This is a bit more tricky to encode since we cannot rely on the order like we did to encode At Least One property. The simplest implementation is to use two nested for loops and check if both profiles are the same and have a different winner. If this is the case, then we want to ensure that both of these cannot be true simultaneously.

Algorithm 2 Resolute - Social Choice

Require: possible_voting_rules
  all_clauses ← [ ]
  for i ≤ length(possible_voting_rules) **do**
    for j ≤ length(possible_voting_rules) **do**
      if *possible_voting_rules*[*i*][0] = *possible_voting_rules*[*j*][0] & *possible_voting_rules*[*i*][1] ≠ *possible_voting_rules*[*j*][1] **then**
        all_clauses.append("-i -j")
      end if
    end for
  end for
  return *all_clauses*

Majority To check for a majority we simply need to verify that at least two voters have put the same alternative in the top position. If this is the case we, can check if the atom corresponds to the most preferred alternative. If it does, we need to add this as a clause to the set of clauses. I assume that we have a function called "most_liked(profile)" which takes in a profile as an argument and returns the candidate who has the majority vote. If no candidate has the majority vote, the function returns "".

Algorithm 3 Majority - Social Choice

Require: possible_voting_rules
  $all\_clauses \leftarrow [\,]$
  for i $\leq$ length(possible_voting_rules) **do**
    $Current\_profile \leftarrow possible\_voting\_rules[i][0]$
    $Majority\_Alternative \leftarrow most\_liked(Current\_profile)$
    if $Majority\_Alternative = possible\_voting\_rules[i][1]$ **then**
      $all\_clauses$.append("i")
    end if
  end for
  return $all\_clauses$

Imposition Imposition states that it is possible that every alternative can win. This is encoded as a conjunction of all atoms corresponding to alternative A, B and C respectively. We can simply encode this as three lists of numbers separated by 3. I.E. [[1,4,7..646],[2,5,8..647],[3,6,9..648]] and then turn each one into a clause by replacing the commas with spaces. I.E. list referring to alternative A winning would become "1 4 7 ... 646". This is possible because every 3^{rd} atom corresponds to every 3^{rd} *possible_voting_rule* which were generated such that every 3^{rd} index corresponds to a unique alternative.

Dictatorship For single winner elections, dictatorship states that a candidate's preferred choice must lose in some profile. For multi-winner elections, it states that any candidate's most preferred linear order must sometimes not be the linear order of the group.

In the below algorithm, I only implement multi-winner case because it borrows a lot of similarities from the single winner case. Covering both is redundant.

The general strategy is to split all atoms into 3 categories - Category where voter 1's, voter 2's and voter 3's linear orders match the aggregated linear order of the group. Most atoms will not belong to any of the categories. Out of these 3 categories, we want at least one atom to be false. Thus we can use negated disjunction on all atoms in each category.

Algorithm 4 Dictatorship - Social Choice

```
Require: possible_voting_rules
  all_clauses ← ["","",""]
  for i ≤ length(possible_voting_rules) do
      for v ≤ length(nr_of_voters) do
          if possible_voting_rules[i][v][0] = possible_voting_rules[i][1] then
              all_clauses[v] = all_clauses[v] + "-i "
          end if
      end for
  end for
  all_clauses.append(all_clauses[0],all_clauses[1],all_clauses[2])
  return all_clauses
```

Pareto Efficiency This property only makes sense when thinking about voting rules which produce a linear order instead of a single winner. It is trivial that no single winner voting rule can satisfy it the way it has been defined in this report because there could be two candidates more preferred by everyone but only a single winner can be assigned.

Therefore, for this property I assume that the second element of all possible voting rules is a strict linear order. I.E. *possible_voting_rules*[0][1] = (A $\prec$ B $\prec$ C).

The general strategy is to loop through every voting rule and check if that rule has some candidate that is more proffered than some other candidate. If so and if the less preferred candidate wins, then forbid this voting rule.

I introduce a function called "is_dominated(profile)" which returns a list of tuples of candidates who are dominated by some other candidate in the profile. For example, is_dominated((A $\prec$ B $\prec$ C,A $\prec$ B $\prec$ C,A $\prec$ B $\prec$ C)) will return a list [(A,B),(A,C),(B,C)] since every voter prefers A to B, A to C and B to C.

I also introduce a function called "ranks_higher(linear_order, candidate_tuple)" which takes in a linear order and a tuple of two alternatives and returns true if in the profile candidate in the first position of the tuple ranks higher than candidate in the second position of tuple. For example ranks_higher(A $\prec$ B $\prec$ C, (C,A)) returns false, because C was less preferred than A.

Algorithm 5 Pareto efficiency - Social Choice

```
Require: possible_voting_rules
  all_clauses ← [ ]
  for i ≤ length(possible_voting_rules) do
      dominated_candidates ← is_dominated(possible_voting_rules[i][0])
      for j ≤ length(dominated_candidates) do
          if  ranks_higher(possible_voting_rules[i][1],dominated_candidates[j])
  then
              all_clauses.append(-i)
          end if
      end for
  end for
  return all_clauses
```

Independence of Irrelevant Alternatives This is probably the hardest rule to encode. It states that if everyone's preferences between candidates A and B remain unchanged then the group's preference between candidates A and B should also remain unchanged. For similar reasons to Pareto efficiency, it only makes sense to discuss this property in context of voting rules which produce strict linear orders.

This property can be broken down into 3 categories - relative preference between alternatives A and B, A and C, B and C. Since we are dealing with strict voting rules, if voter 1 does not rank A higher than B, then they have ranked B higher than A.

We can break down each category further by splitting it into $(2^3) = 8$ subcategories. Where 3 comes from the fact we have 3 voters. We can fit every atom inside one of these 8 subcategories. To see which subcategory an atom belongs to, we start at 1 and add 4 if in the atom refers to a profile in which voter 1 prefers A over B. We add 2 if voter 2 prefers A over B. We add 1 if voter 3 prefers A over B. All voting rules only belong to one of the 8 subcategories.

For each subcategory, we can further break down the atoms into two *cases* ones where the voting rule assigned alternative A as the winner and others where the voting rule assigned B as the winner.

IoIA in this context then means that if any atom of a *case* is true, then all atoms of the other case must be *false*. The last thing to note before looking at the pseudo-code is that we can translate "if a then not b" into CNF by "not a or not b" and into DIMACS by "-1 -2".

I introduce two functions - get_case(profile, alternative_tuple) which returns the subcategory which the alternative tuple belongs to in the given profile. For example, get_case((B $\prec$ A $\prec$ C, A $\prec$ B $\prec$ C, A $\prec$ B $\prec$ C), (A,B)) will return 4 (1 + 0 + 2 + 1) since we start at 1, voter 1 doesn't prefer A over B so we add zero. Voter 2 does prefer A over B so we add 2. And voter 3 also prefers A over B so we add 1. The second function get_subcase(linear_order,alternative_tuple) will take in a linear order corresponding to the group's linear ranking and alternative tuple. It will return a true when alternative tuple is ranked the same in linear order. For example get_subcase(A $\prec$ B $\prec$ C, (B,A)) returns false because B is not ranked above A in the linear order.

Algorithm 6 Independence of Irrelevant Alternatives - Social Choice

```
Require: possible_voting_rules
  all_clauses ← [ ]
  alternative_tuples ← [(A,B),(A,C),(B,C)]
  for i ≤ length(possible_voting_rules) do
     for j ≤ length(alternative_tuples) do
        case = get_case(possible_voting_rules[i][0], alternative_tuples[j])
        subcase = get_subcase(possible_voting_rules[i][1],alternative_tuples[j])
        for ii ≤ length(possible_voting_rules) do
           for jj ≤ length(alternative_tuples) do
              case_other = get_case(possible_voting_rules[ii][0], alternative_tuples[jj])
              subcase_other = get_subcase(possible_voting_rules[ii][1],alternative_tuples[jj])
              if case = case_other & subcase ≠ subcase_other then
                 all_clauses.append("-i -ii")
              end if
           end for
        end for
     end for
  end for
  return all_clauses
```

Strategy-proofness I decided to only implement strategy-proofness for single winner elections. This is because of the difficulty defining whether a linear order is better or worse for a voter and thus defining whether a manipulation was successful. Multiple definitions could exist for multi-winner elections such as using Borda count of the voting rule's linear ranking or counting manipulations as successful only if *all* of voter's preferences align with the ranking produced by voting rule.

To encode strategy-proofness one needs to forbid all pairs of atoms which are incompatible with each other due to breaking strategy-proofness. Let p_1 be the profile referred to by atom 1 and contain the voter's true preferences. Let p_2 be the profile referred to by atom 2 and contain the voter's manipulated preferences. Two atoms are then incompatible with each other, if some voter can change their preferences in p_1 and thus turn p_1 into p_2 *and* atom 2's winner is better for the voter who manipulates than atom 1's winner. For example

atom 1 could be: ((A $\prec$ B $\prec$ C, A $\prec$ B $\prec$ C, B $\prec$ C $\prec$ A), A) and

atom 2 could be: ((A $\prec$ B $\prec$ C, A $\prec$ B $\prec$ C, C $\prec$ B $\prec$ A), B). In this case voter 3 has successfully manipulated the election because he prefers alternative B over A and by switching his preferences to C $\prec$ B $\prec$ A he has produced a desierable outcome and thus *both* atoms cannot be true.

Finally, I define a function is_better(linear_order,alternative_tuple) which returns true when the first alternative in alternative_tuple is ranked higher in the linear order than second alternative. This functions the same way that the get_subcase function does, but I thought the name of is_better was more

appropriate.

Algorithm 7 strategy-proofness - Social Choice

```
Require: possible_voting_rules
  all_clauses ← [ ]
  for i ≤ length(possible_voting_rules) do
      for j ≤ number_of_voters do
          original_profile ← possible_voting_rules[i][0]
          voter_true_prefs ← original_profile[j]
          winner_true_prefs ← possible_voting_rules[i][1]
          for ii ≤ length(possible_voting_rules) do
              manip_profile ← possible_voting_rules[ii][0]
              voter_manip_prefs ← manip_profile[j]
              winner_manip_prefs ← possible_voting_rules[ii][1]
              profile_differ_one ← true
              for l ≤ length(number_of_voters) do
                  if l ≠ j & original_profile[l] ≠ manip_profile[l] then
                      profile_differ_one ← false
                  end if
              end for
              if profile_differ_one & is_better(voter_true_prefs,(winner_manip_prefs,winner_true_prefs)
  ) then
                  all_clauses.append("-i -ii")
              end if
          end for
      end for
  end for
  return all_clauses
```

4.2.2 Summary

We have seen how to encode 8 properties used by the three theorems in social choice. It was noted that properties like independence of irrelevant alternatives only applies to linear orders and does *not* apply to single winner elections. Therefore, I decide to split the encoding into 2 scripts: GSTencoder.py and ArrowEncoder.py since simplified and full GST use a single winner election method and Arrow impossibility theorem uses strict linear orders. See figure 10 for which properties were used to generate the three impossibility results. A better implementation may have been to provide both encodings in a single script and letting the user choose between linear orders and single-winner elections. This is discussed further in Section 6 on future work.

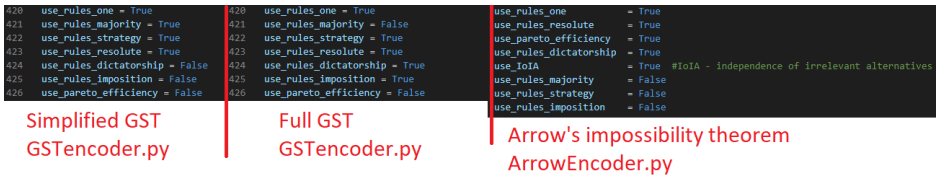


Figure 10: This figure shows the configuration used for the three theorem and from which file they are.

4.2.3 Preserving the Meaning (Step 2 b)

We have now seen how to encode all relevant properties for theorems in social choice. For the decoder script to work as intended, we do need to save the meaning of all literals and meaning of what property each clause is trying to encode.

Meaning of the atoms is stored by saving the possible_voting_rules variable. Atom with number i will correspond to i^{th} element in the list.

Finally, a dictionary is saved to preserve the meaning of all clauses. The dictionary's key is a CNF clause and the value is which property is encoded by the clause. See figure 11 for how this was implemented.

4.2.4 Argumentation Theory

Argumentation theory implementation details were not much different from social choice and thus were not covered as a part of this report. I believe that the approach section covers enough high-level detail for the reader to be able to understand the source-code.

4.3 Human Readable Proofs

There are two approaches covered in section 2.6 which have been discussed to make proofs human readable. Approach one was to generate a diagram from

```
#save the meaning behind the generated variables
all_rules_dict = {}
write_rules = True
if write_rules:
    for rule in rules_one:
        all_rules_dict[rule] = "rules one"
    for rule in rules_resolute:
        all_rules_dict[rule] = "rules resolute"
    for rule in rules_majority:
        all_rules_dict[rule] = "rules majority"
    for rule in rules_strategy:
        all_rules_dict[rule] = "rules strategy"
    for rule in rules_imposition:
        all_rules_dict[rule] = "rules imposition"
    for rule in rules_dictatorship:
        all_rules_dict[rule] = "rules dictatorship"
    for rule in rules_pareto:
        all_rules_dict[rule] = "rules pareto"


with open('ExplainableMUS\explainations.pickle', 'wb') as f:
    pickle.dump([possible_functions,all_rules_dict], f)
```

Figure 11: This figure shows how the dictionary for step 2 b is generated and saved as a pickle file.

MUS and approach two was to generate a natural language description of the proof found using a SAT solver with proof logging.

4.3.1 MUS diagram

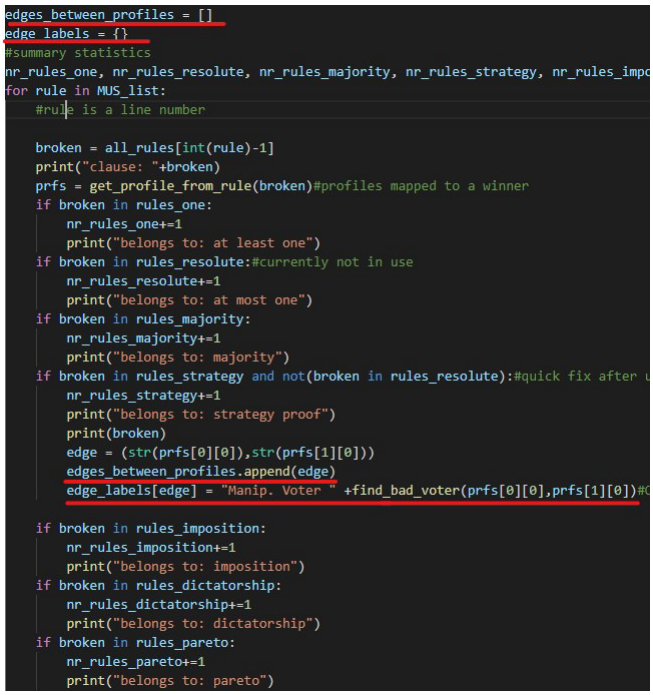
Generating a MUS diagram requires two things: the MUS produced by MUSER2 (Step 3) and meaning of clauses saved in (Step 2 b). Only a very basic implementation of visualising the MUS was done due to limitations of what could be displayed in the MUS diagram becoming apparent and the advantages of a solution integrating natural language directly becoming clear.

To create a MUS diagram we go through all of the clauses produced by the MUS extractor. We gather a list of profiles that the clauses in the MUS use by looking up every atom in the MUS. Each profile will be a node in our graph.

To encode strategy-proofness we get the two profiles associated with each strategy-proofness clause and add a directed edge going from voter's true preference to manipulated preference.

4.3.2 Natural Language (Step 5)

This section of implementation corresponds to the final step of my proposed solution. This means that all of the meanings of clauses and atoms have been saved and that we have now generated a FRAT proof. It may help the viewer to refer to figure 4 to remind them of the format that FRAT proofs produce

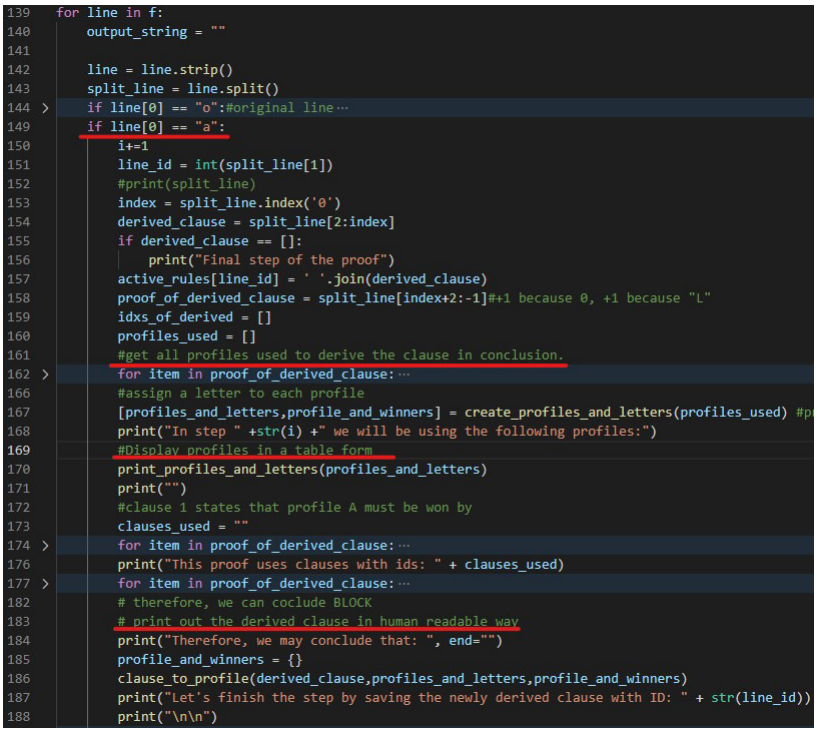


```
edges_between_profiles = []
edge_labels = {}
#summary statistics
nr_rules_one, nr_rules_resolute, nr_rules_majority, nr_rules_strategy, nr_rules_impo
for rule in MUS_list:
    #rule is a line number

    broken = all_rules[int(rule)-1]
    print("clause: "+broken)
    prfs = get_profile_from_rule(broken)#profiles mapped to a winner
    if broken in rules_one:
        nr_rules_one+=1
        print("belongs to: at least one")
    if broken in rules_resolute:#currently not in use
        nr_rules_resolute+=1
        print("belongs to: at most one")
    if broken in rules_majority:
        nr_rules_majority+=1
        print("belongs to: majority")
    if broken in rules_strategy and not(broken in rules_resolute):#quick fix after u
        nr_rules_strategy+=1
        print("belongs to: strategy proof")
        print(broken)
        edge = (str(prfs[0][0]),str(prfs[1][0]))
        edges_between_profiles.append(edge)
        edge_labels[edge] = "Manip. Voter " +find_bad_voter(prfs[0][0],prfs[1][0])#C

    if broken in rules_imposition:
        nr_rules_imposition+=1
        print("belongs to: imposition")
    if broken in rules_dictatorship:
        nr_rules_dictatorship+=1
        print("belongs to: dictatorship")
    if broken in rules_pareto:
        nr_rules_pareto+=1
        print("belongs to: pareto")
```

Figure 12: This figure shows how the labels and edges were generated for the MUS diagram. MUS_list is a list containing all clauses generated by the MUS extractor.

and to section 3.9 to see the approach taken to translate the proof into natural language.

The general strategy is to go through the FRAT proof line by line, whenever a conflict clause is added, retrieve the associated profiles (both of the derived clause and the clauses used as justification), also retrieve their winners and wrap it all in natural language. Finally, I format all profiles used in the proof by adding a table which makes the proof generated easier to read and more visually appealing. See figure 13 for the outline of the code that was used to generate it.



```
for line in f:
    output_string = ""

    line = line.strip()
    split_line = line.split()
    if line[0] == "o":#original line ···
    if line[0] == "a":
        i+=1
        line_id = int(split_line[1])
        #print(split_line)
        index = split_line.index('0')
        derived_clause = split_line[2:index]
        if derived_clause == []:
            print("Final step of the proof")
        active_rules[line_id] = ' '.join(derived_clause)
        proof_of_derived_clause = split_line[index+2:-1]#+1 because 0, +1 because "L"
        idxs_of_derived = []
        profiles_used = []
        #get all profiles used to derive the clause in conclusion.
        for item in proof_of_derived_clause: ···
        #assign a letter to each profile
        [profiles_and_letters,profile_and_winners] = create_profiles_and_letters(profiles_used) #p
        print("In step " +str(i) +" we will be using the following profiles:")
        #Display profiles in a table form
        print_profiles_and_letters(profiles_and_letters)
        print("")
        #clause 1 states that profile A must be won by
        clauses_used = ""
        for item in proof_of_derived_clause: ···
        print("This proof uses clauses with ids: " + clauses_used)
        for item in proof_of_derived_clause: ···
        # therefore, we can coclude BLOCK
        # print out the derived clause in human readable way
        print("Therefore, we may conclude that: ", end="")
        profile_and_winners = {}
        clause_to_profile(derived_clause,profiles_and_letters,profile_and_winners)
        print("Let's finish the step by saving the newly derived clause with ID: " + str(line_id))
        print("\n\n")
```

Figure 13: This figure shows the outline of the function which turns the FRAT proof into English. You can see that the proof begins by confirming that the FRAT line is a newly derived clause then it lists the step number, followed by displaying a table of all profiles used to derive the clause. It concludes by saving the newly derived clause so that it can be referred to by other clauses later in the proof.

5 Results and Evaluation

We have now seen how to encode proofs into SAT and how to turn those proofs into natural language or MUS diagrams. All that's left to do, is to choose the best SAT solver by **comparing different SAT solver performance**. Then we will take a look at the two current approaches to **explain SAT proofs**, namely, using a visual MUS diagram and using a natural language description. Finally, we will take a look at the **Strengths and Weaknesses** of both approaches. This will serve as an introduction into future work.

5.1 Performance Comparison of SAT Solvers

The goal of this section is determine whether the type of SAT solver used when trying to find an impossibility results in social choice has a *big* impact on the

time taken to find it. This evaluation is by no means extensive since only three theorems and three different solvers were used, but it if there is a big difference then finding the best SAT solver for finding impossibility theorems in social choice could be a future topic to research.

Hypothesis I have chosen three SAT solvers to use to benchmark performance. MiniSat is a lightweight solver which was first released in 1997 and has received only slight modifications over the years. I expect it to perform the slowest and it should highlight all of the progress that has been made over the last 20 years in SAT solving. Sat4j is a state of the art SAT solving library implemented in *Java* and thus runs on a virtual machine. I expect it to perform better than MiniSat, but worse than lingeling. Lingeling was chosen as a relatively recent solver (first released in 2010) which has been continuously developed. It won the silver medal in 2016 and a modified version called tree-lingeling won the gold medal that year. I expect it to outperform all other solvers.

Setup All SAT solvers came with an in-built timing function which was used to record the time taken to solve each theorem. I ran every theorem through each SAT solver 10 times to examine the variability and get a more accurate picture of how long it takes to solve each SAT formula.

I tested the time taken to find impossibilities on 3 theorems in social choice, namely, simplified GST, GST, and Arrow's impossibility theorem. In figure 10 I provide the configuration of axioms which were used to generate the CNF file. Note, I do not run the SAT solvers on the MUS, I run the SAT solvers on the fully encoded theorems.

Whilst testing, I alternate between different SAT solvers to clear the cache. Not clearing the cache between runs offers a speedup, but is not realistic for our use-case.

Note, lingeling solver only offers to measure performance rounded to the nearest tenth of a second. Since all formulas were solved very quickly, this makes the comparison somewhat inaccurate, but it still should give a rough idea of the performance.

Results The timing of each run is available in the Appendix 24. The results only partially matched my hypothesis. Indeed, there has been lots of progress made on solving difficult SAT problems with many clauses. The most difficult theorem to solve turned out to be Arrow's impossibility theorem because it uses the most atoms (1296) and has the highest number of clauses (322,113).

For Arrow's impossibility theorem, MiniSat averaged 369 ms, Sat4j came it at respectable 182ms, but lingeling completed it in 100ms.

Both simplified GST and full GST produced the opposite results. MiniSat and Sat4j outperformed the newer lingeling solver with MiniSat being the fastest averaging 18ms over the two theorems, Sat4j being the second fastest averaging 62ms and lingeling averaging 150ms over the two theorems.

My guess is that java virtual machine takes time to set up; however, MiniSat takes no time thus for smaller formulas it is faster. However, as the number of clauses and variables increases, the modern advances in SAT solving become apparent as seen when looking at Arrow's impossibility theorem.

In summary, the results indicate that the choice of the SAT solver is not widely important for the base-cases discussed here. I leave the evaluation of performance for base-cases with more voters and alternatives as future work.

5.2 Explainable SAT Proofs for Social Choice

We have found that the formulas corresponding to base-cases for simplified GST, full GST, and Arrow's Impossibility theorem are all *unsatisfiable*. This is to be expected.

We now turn our attention to explaining *why* they are unsatisfiable. In so doing, we should be able to tell if the proof found is valid and we have not made any mistakes in the encoding. A helpful exercise is to imagine that we do not know whether a formula is unsatisfiable and whether our encoding is correct.

The two approaches considered are MUS diagrams and generating natural language description from the SAT proof logger. After reporting the results, both methods are evaluated.

Note: the full versions of all of the proofs discussed can be found inside "proofs" folder inside my submission.

5.2.1 MUS Diagrams

MUS diagrams were generated to visualise the constraints in the MUS. Each node is a profile and each arrow is a constraint concerning two profiles. For example, strategy-proofness. CNF clauses which contain more than two profiles (such as non-dictatorship) were not visualised at all because it was unclear how they *should* be visualised.

Simplified GST Everything for simplified GST went right in terms of finding and visualising the MUS. The MUS found was *only* 7 clauses long and is relatively easy to understand. It marks an improvement over the one reported in [21] which was 9 clauses long and was used as inspiration for this work.

This is the smallest possible MUS because the MUS extractor MUSER2 was ran in *insertion* mode without refinement meaning that clauses were iteratively added until an unsatisfiable subset was formed.

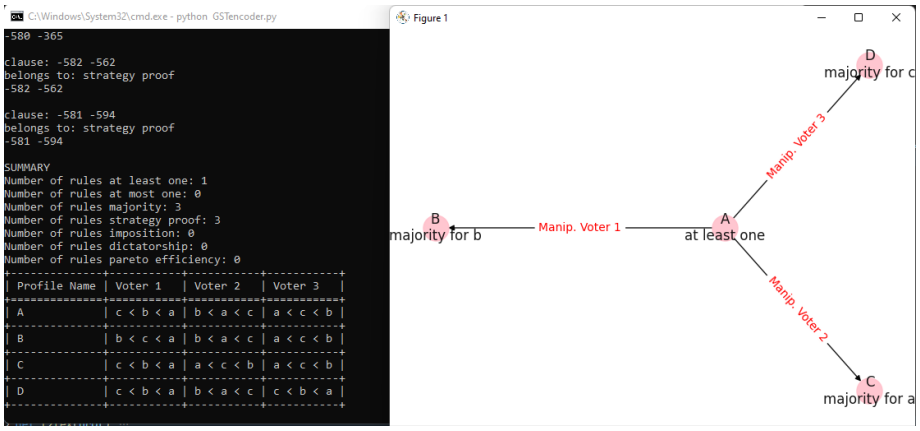


Figure 14: This figure shows a visualisation of the smallest possible MUS for simplified GST. It shows that profile A (the one in the middle) cannot be assigned any alternative as a winner. Suppose a voting rule assigns profile A to have the winner of alternative a. Voter 1's true preferences for profile A are $c \prec b \prec a$ which means that he really does not want alternative a to win. He can misrepresent his preferences as being $b \prec c \prec a$ and thus achieve a majority for alternative b. According to his true preferences b winning the election is better than a winning the election thus this has been a successful manipulation. Similar reasoning can be applied to exclude assigning alternatives b or c thus leaving profile A without a winner.

Full GST For Full GST, visualising the smallest MUS does not offer much insight into whether the proof is correct. This is because the MUS found used *all* 216 profiles. This meant that the MUS had 216 nodes which makes it very difficult to visualise. See figure 15 for "visualisation" of the MUS. To make matters worse, there are no "root" nodes. I.E. all nodes have outgoing edges meaning there is no clear place to start. Furthermore, I do not display the dictatorship rules because it states that one of 72 profiles *must* be false. This is not exactly easy to display in a visual diagram.

Whilst visualising the MUS seems hopeless we can still get a sense about whether the MUS found is correct by checking the *summary* statistics of what each clause in the MUS encodes. See figure 16 for the summary statistics of the full GST MUS as well as the decoding of some of the profiles.

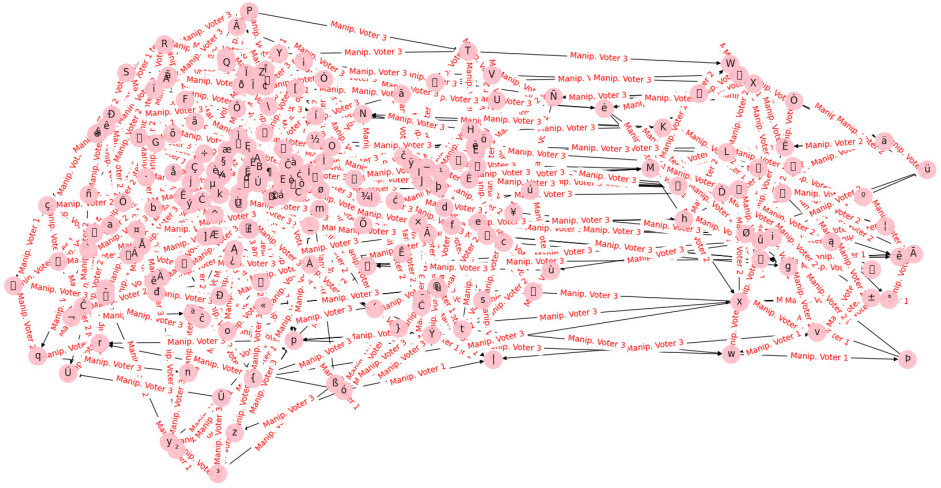


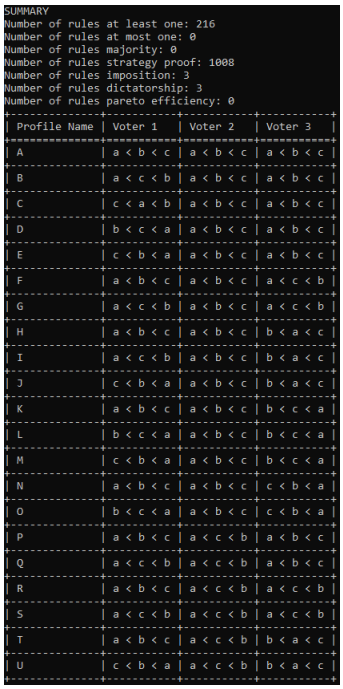
Figure 15: The smallest MUS found for full GST contains 216 profiles and uses 1230 clauses. I provide the meaning of some node label into figure 16. The full decoding of all profiles is provided in results folder of the submission.

The summary statistics tell us that the MUS uses four types of clauses - at least one, strategy proof, imposition and dictatorship. This sounds similar to the simplified GST proof in which it was found that it is impossible for some profile to be assigned a winner.

We should take anything we might imply from the summary statistics with a pinch of salt because the MUS extractor was ran in approximate mode due to time-complexity of finding the smallest MUS. (I.e. it finds a small set of unsatisfiable clauses, but there might exist a smaller set). Secondly, just because the *smallest* subset uses clauses from a certain property *does not* imply that the type of clause is required to make the CNF formula unsatisfiable. I.E. CNF formula might still contain a contradiction if some "at least one" clauses are excluded.

This might make one wonder if that really is the case. I.e. does excluding one of these properties would imply that CNF encoding becomes satisfiable. This intuition is confirmed when the encoder script is configured to not use one of the four properties which were used in the MUS. For all four properties the CNF formula became satisfiable.

The second thing to note about the MUS is that it uses *all* of the clauses in at least one, imposition and dictatorship. This suggests that all of those clauses are needed for a contradiction. Indeed, this hypothesis was also confirmed. For example, excluding the clause forbidding voter 3 from becoming a dictator and running this through a SAT solver generates a valid assignment of atoms where



```
SUMMARY
Number of rules at least one: 216
Number of rules at most one: 0
Number of rules majority: 0
Number of rules strategy proof: 1008
Number of rules imposition: 3
Number of rules dictatorship: 3
Number of rules pareto efficiency: 0
```

Profile Name	Voter 1	Voter 2	Voter 3
A	a < b < c	a < b < c	a < b < c
B	a < c < b	a < b < c	a < b < c
C	c < a < b	a < b < c	a < b < c
D	b < c < a	a < b < c	a < b < c
E	c < b < a	a < b < c	a < b < c
F	a < b < c	a < b < c	a < c < b
G	a < c < b	a < b < c	a < c < b
H	a < b < c	a < b < c	b < a < c
I	a < c < b	a < b < c	b < a < c
J	c < b < a	a < b < c	b < a < c
K	a < b < c	a < b < c	b < c < a
L	b < c < a	a < b < c	b < c < a
M	c < b < a	a < b < c	b < c < a
N	a < b < c	a < b < c	c < b < a
O	b < c < a	a < b < c	c < b < a
P	a < b < c	a < c < b	a < b < c
Q	a < c < b	a < c < b	a < b < c
R	a < b < c	a < c < b	a < c < b
S	a < c < b	a < c < b	a < c < b
T	a < b < c	a < c < b	b < a < c
U	c < b < a	a < c < b	b < a < c

Figure 16: This figure shows the profiles of first couple of nodes used in figure 15. More importantly, it also displays the *summary* statics of what each clause in the MUS encodes.

voter 3 is a dictator.

I am unable to see why specifically 1008 clauses of strategy-proofness were needed. it is only approximate and therefore slightly fewer strategy-proof clauses might have been needed to produce a contradiction.

Arrow's impossibility theorem Similarly to Full GST, the MUS contains far too many clauses (2,189 to be exact) to be understandable only using the MUS diagram. Nevertheless, I provide the diagram generated in figure 17. Similarly to full GST, a better insight might be found using the summary statistics on the next page.

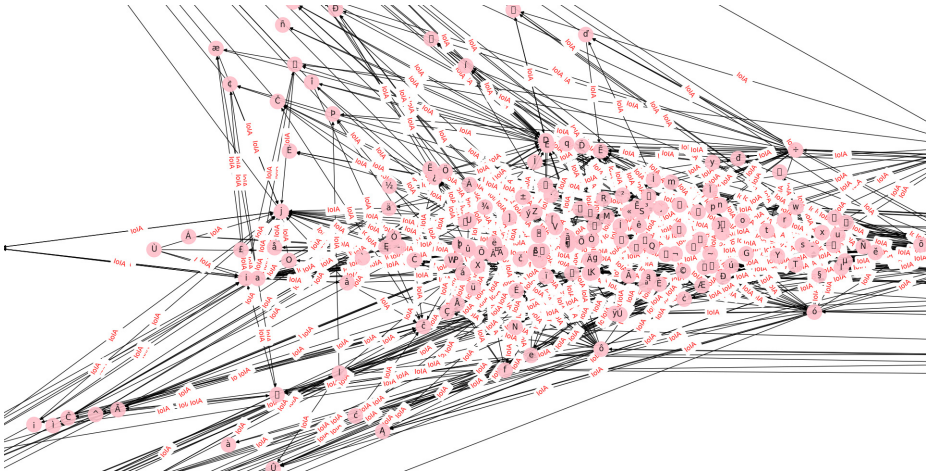
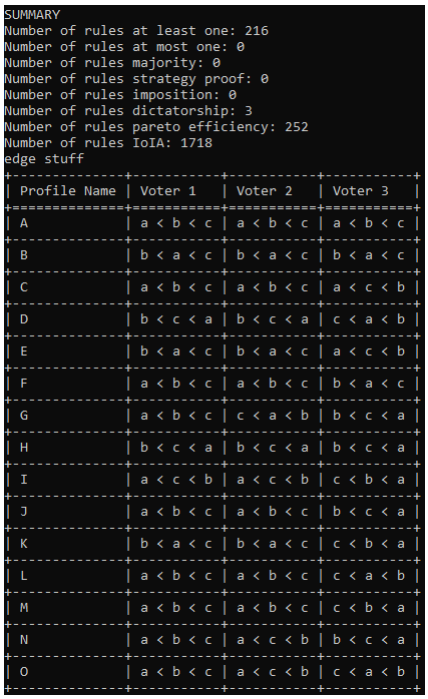


Figure 17: The smallest MUS found for Arrow's impossibility theorem contains 216 profiles and uses 2,189 clauses. I provide the meaning of some node label into figure 18. The full decoding of all profiles is provided in results folder of the submission.

The summary statistics are similar to full GST's summary statistics in that all clauses relating to dictatorship property and at least one property were required to make the formula unsatisfiable. Again, this was verified by removing one clause relating to either property and finding that the new CNF formula was satisfiable.

I am not sure why specifically 1,718 (out of 314,928) clauses were needed for IoIA or why exactly 252 (out of 486) were required for Pareto efficiency. This does imply that there is some redundancy and that not all IoIA or Pareto efficiency clauses are needed to produce a contradiction.



```
SUMMARY
Number of rules at least one: 216
Number of rules at most one: 0
Number of rules majority: 0
Number of rules strategy proof: 0
Number of rules imposition: 0
Number of rules dictatorship: 3
Number of rules pareto efficiency: 252
Number of rules IoIA: 1718
edge stuff
```

Profile Name	Voter 1	Voter 2	Voter 3
A	a ≺ b ≺ c	a ≺ b ≺ c	a ≺ b ≺ c
B	b ≺ a ≺ c	b ≺ a ≺ c	b ≺ a ≺ c
C	a ≺ b ≺ c	a ≺ b ≺ c	a ≺ c ≺ b
D	b ≺ c ≺ a	b ≺ c ≺ a	c ≺ a ≺ b
E	b ≺ a ≺ c	b ≺ a ≺ c	a ≺ c ≺ b
F	a ≺ b ≺ c	a ≺ b ≺ c	b ≺ a ≺ c
G	a ≺ b ≺ c	c ≺ a ≺ b	b ≺ c ≺ a
H	b ≺ c ≺ a	b ≺ c ≺ a	b ≺ c ≺ a
I	a ≺ c ≺ b	a ≺ c ≺ b	c ≺ b ≺ a
J	a ≺ b ≺ c	a ≺ b ≺ c	b ≺ c ≺ a
K	b ≺ a ≺ c	b ≺ a ≺ c	c ≺ b ≺ a
L	a ≺ b ≺ c	a ≺ b ≺ c	c ≺ a ≺ b
M	a ≺ b ≺ c	a ≺ b ≺ c	c ≺ b ≺ a
N	a ≺ b ≺ c	a ≺ c ≺ b	b ≺ c ≺ a
O	a ≺ b ≺ c	a ≺ c ≺ b	c ≺ a ≺ b

Figure 18: This figure shows the profiles of first couple of nodes used in figure 17. More importantly, it also displays the *summary* statics of what each clause in the MUS encodes.

5.2.2 Natural Language Proofs

We have seen how to visualise the smallest set of clauses using MUS diagrams. In this section, I show the natural language descriptions generated for simplified GST, full GST and Arrow's impossibility theorems and provide brief descriptions. For evaluation and further discussion of the natural language descriptions see Section 5.2.3 on evaluation of MUS and natural language.

Simplified GST The description generated for simplified GST was relatively straight forward to understand although it was a little bit lengthy. The proof starts by listing all original CNF clauses and providing what meaning is encoded by each.

Then, the proof proceeds to derive conflict clauses until finally, the empty clause is derived indicating an impossibility result. I display the output of the proof as two figures 19, 20. Please note that the profile names are re-assigned at every step. I.E. Profile A in step 1 is not the same as profile A in step 2! Also note that whenever a profile must be won by candidate "not c", where c is some candidate the step is saying that the profile *cannot* be won by candidate c. In other words, candidate c being assigned as the winner would lead to a

contradiction.

The second thing to note is that while it may be difficult to see how all of the steps of the proof fit together, so long as each step can be checked independently we can gain confidence in the derived proof. For further discussion, see evaluation section.

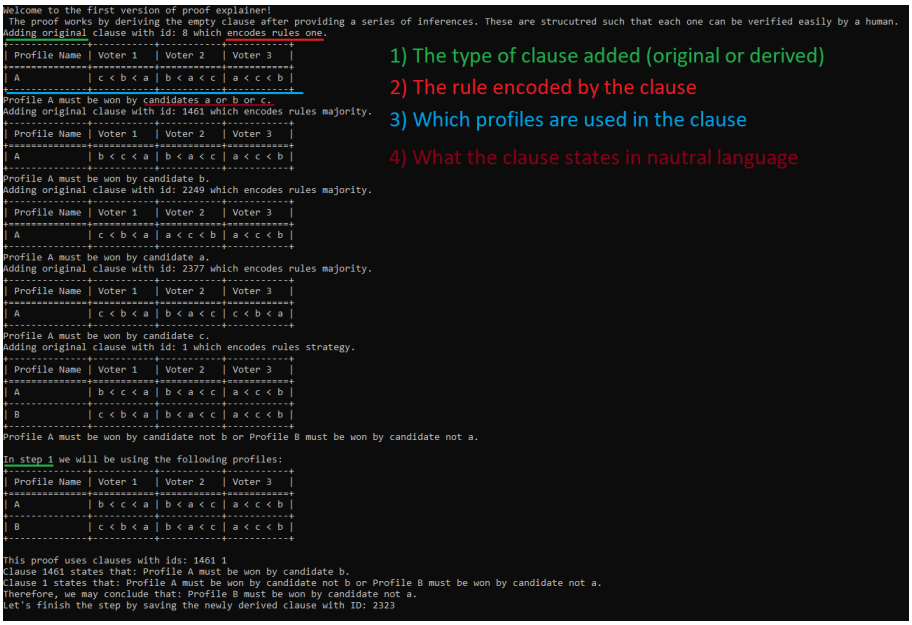


Figure 19: This figure shows the first part of the impossibility result derived for simplified GST. It shows how the FRAT style proof was converted into natural language. For each of the original clauses, the script states what axiom it is encoding and which profiles it is referring to. In step 1 of the proof we derive that profile B cannot be won by candidate a. You might remember profile B as being the centre node in the MUS diagram in figure 14.

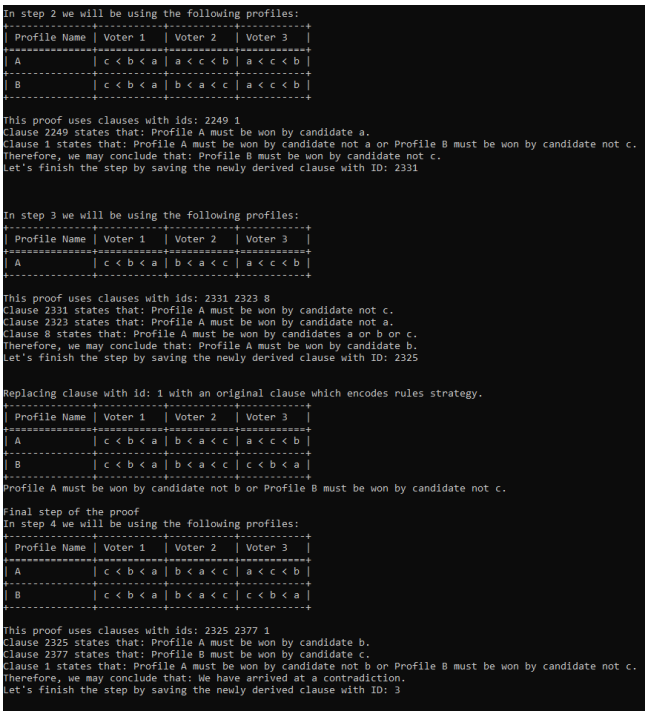


Figure 20: This figure shows the final part of the impossibility result derived for simplified GST. It shows the steps 2-4. In step 2, we derive that profile B cannot be won by candidate c. Step 3 shows that profile A must be won by candidate b. Step 4 concludes the proof by showing that profile A must be won by candidate b and candidate c, but also cannot be won by candidate b or c. Clearly a contradiction.

Full GST I do not provide the full proof for the full GST in this report (instead it is uploaded under results) because it contains 711 steps and is *45,691* lines long. The majority of it is printing out all of the profiles in table format.

What is interesting, is that the proof ends in a similar way to the simplified GST MUS in that there is a profile where one of the alternatives must win the election, but it has been shown that no individual candidate can win because this would imply a contradiction. See figure 21.

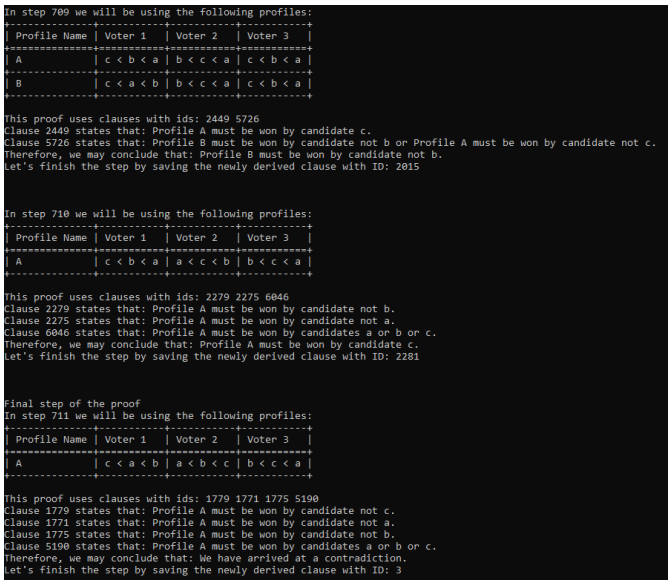


Figure 21: This figure shows the final part of the impossibility result derived for full GST. The interesting part is the final step 711 where an alternative must be assigned as a winner to profile A, but assigning any candidate has been shown to lead to a contradiction.

Arrow's Impossibility Theorem Unfortunately, there exists a bug in the official FRAT proof logging system developed for MiniSat. Thus a full proof of the impossibility of the base-case was not generated. The bug is that one of the conflict clauses uses a clause with an id that has not been specified.

The partial proof leading upon that point was recovered and saved. In figure 22, I provide a small part of the proof. Important to note is that every atom corresponds to a strict linear order instead of just a single candidate. This is automatically detected and changed accordingly by the script.

The partial proof is 21,211 lines long which means that it is too long to read and understand by a single person.

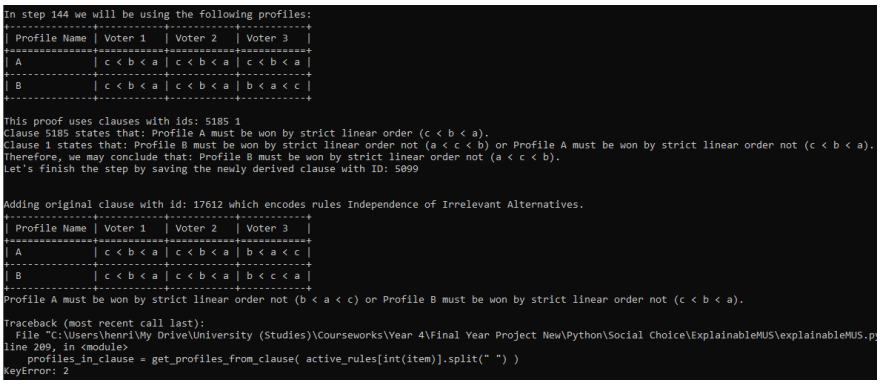


Figure 22: This figure shows the last steps of the impossibility result for Arrow's impossibility theorem before a clause derivation requires an index that has not been assigned. The clause with id 2 is not one of the original clauses, is not added in any step *and* there have been no index reallocation steps which means that the justification provided by the proof-logger is faulty.

5.2.3 Evaluation of MUS Diagrams and Natural Language Proofs

This section will discuss the benefits and limitations of both MUS diagrams and natural language proofs. Both of the approaches performed well on the simplified GST, but did poorly on the full GST and Arrow's theorems.

This is to be expected because they are both attempting to explain the same subset of clauses which are unsatisfiable. If the subset of clauses is too large, then either method will not be easily understandable.

MUS visualisation The MUS visualisation is easier to understand when the number of clauses is very small. It gives a nice, visual way of seeing the contradiction. For example, I think that figure 14 illustrates the simplified GSTs impossibility result very clearly.

Whilst MUS visualisation worked great on the simplified GST, it is somewhat of a toy example. The key challenges to visualising the more complicated MUSes are: readability, representation power, reducibility, and mental strain.

Readability suffers greatly when the MUS produced contains many profiles and many nodes. This was the case for full GST and Arrow's impossibility theorems. Secondly, having to look up what letter each profile corresponds to was inconvenient, but representing them on a diagram (which was initially done) made the diagram very overloaded.

Representation Power is somewhat limited. In the implementation section, I briefly discuss that I do not visualise clauses which refer to more than 2 profiles. Indeed this seems to be an open challenge. A potential solution could be to encode such clauses using a different colour.

Reducibility and Mental Strain. The MUS diagram only functions as

a whole unit. You cannot break it down into small pieces. Understand each piece individually and then move on to another part of the graph. Secondly, you need to keep in mind all of the deductions you have inferred since then. Even in figure 14, the user is asked to remember the possible winners of each profile which causes mental strain. For more complicated cases, this is quite challenging to do.

Natural Language Proofs The natural language proofs solve some of the problems faced by simply visualising the MUS. Firstly, it is **reducible** into discrete steps each of which can be checked individually. Secondly, the user does not need to use a lot of mental strain to verify each step. Although in the current implementation long clauses may require significant mental effort (a possible solution is provided to this in future work Section 6). Thirdly, It does not struggle with **representation power**. Clauses referring to more than two profiles are easily represented.

However, the proposed solution struggles with **readability**. Reducing the mental burden on the human means explicitly explaining every step. This produced proofs which are far longer than practically useful for most circumstances. Potential remedies to this are outlined in future work Section 6.

5.3 Explainable SAT Proof for Argumentation Theory

The impossibility result for argumentation theory was exactly the same which was found in the paper by Booth et al. [11]. Due to time constraints, I did not adapt the script which translates the proof into natural language; however, the MUS of the proof was very short and only referred to a single profile thus was easy to understand and visualise.

The profile for which no aggregate winner can be assigned is displayed in figure 23.

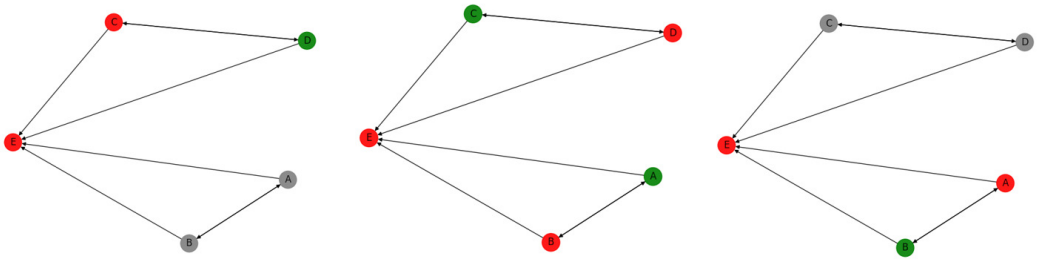


Figure 23: This figure shows the only profile referred to by the MUS which formed a contradiction for Theorem 3 in argumentation theory. Nodes which are *in* are coloured green. Nodes which are *out* are coloured red. Nodes which are *undec* are coloured grey. Node E is labelled *out* by all participants. All other nodes have 1 vote for *in*, 1 for *out* and 1 for *undec*.

The MUS found consisted of 10 clauses. 1 - at least one, 1 - unanimous and 8 AF-independence clauses. The at least one clause states that the profile shown in figure 23 must be assigned a winner. The unanimous clause states that node E must be aggregated to "out". Finally, the 8 AF-independence clauses state that nodes A, B, C, and D must have the same labels since they belong to the same subgroup (1 vote for *in*, 1 vote for *out* and 1 vote for *undec*). However, there is no *complete* labelling where node E is *out*, **and** nodes A, B, C and D have the same label. Thus we have verified a Theorem 3's base case for 3 voters.

5.4 Evaluation of Using SAT for Social Choice and Argumentation Theory

The technique of using SAT solving for theorems in social choice and argumentation theory has been very successful overall. We have derived some of the most famous results in social choice and shown that this technique is universal enough that it can be applied in other domains such as argumentation theory. We have found that this technique can sometimes even produce results which are explainable and easily understandable by humans.

However, this technique does have some serious limitations which prevents it from being universally applicable. I have split the limitations of this technique into two sections: **issues with SAT solving** where I discuss inherent problems with the technique and **user friendliness**, where I discuss issues related to how easy to use it is.

5.4.1 Issues with SAT Solving

Inefficient Encoding One of the hopes was to apply this technique to an *unsolved* conjecture in argumentation theory. Unlike all theorems tested in this paper, the conjecture only stated that a single property *always* holds. This property very loosely translated to saying that a certain voting rule is "strategy-proof".

The way to encode this property into SAT was to generate all winners for all profiles according to the voting rule and then check that all pairs of profiles obeyed the property. However, by generating the encoding, we will have already checked whether the property holds. Since if we find a pair of profiles which can be manipulated, we will have found a contradiction without needing to use the SAT solver. This means that the technique is only applicable to cases where you solve the problem by encoding it. Typically, the more properties you have the higher the chance that using a SAT solver is the right approach.

In practice, this means that the technique's applicability is limited in cases when generating the encoding is difficult.

Not Expressive After writing the script to check the base-cases and being unable to find a contradiction, I began to hypothesize that the conjecture was true.

Unfortunately, you cannot use SAT solving to *prove* theorems. This is not surprising because it is not using an expressive logic, but at the same time, it further limits its applicability.

Furthermore, even when a counter-example *is* found, we may want to prove the impossibility for *any* number of voters, not just the number used in the base-cases which were verified by the solver. The only silver lining is that the counter-example may provide us with intuition about how to prove the general theorem and serve as a base-case for an inductive argument.

5.4.2 Not User Friendly

Whilst criticisms about how user-friendly a technique is does not limit its power, it is a major hurdle preventing a wider adoption and more research. Overall, it is time-consuming to set-up and difficult to verify.

The setup requires familiarity with SAT and a strategy to express the problem one is trying to solve as a SAT problem. Once you have a strategy (similar to what was discussed in the approach section), generating the CNF encoding is usually straight forward.

The second problem is that it is difficult to verify both solution and encoding. Most of this report focused on verifying and explaining the solution found. In practice, you also need to verify that the code used for encoding aligns with your mental model of what it should be encoding. Especially in cases where you *do* find a valid assignment.

6 Future work

I have split this section into two parts - **immediate future work** and **distant future work**. Immediate future work will focus on small things which could have been done had there been a little bit more time for the project (about a week's worth of time). Distant future work will focus on potential ways to solve shortcomings identified in evaluation of using SAT for Social Choice section 5.4, but these would take significantly more time (each could be a final year project in itself).

6.1 Immediate Future Work

6.1.1 Improvements to Encoder Script

There are several small improvements to the Encoder script for theorems in social choice I did not have time to make. These are allowing the user to specify the number of voters, the number of alternatives, what the voting rule should output (single winner, strict linear order or weak linear order). Finally, the encoder script could have a little-bit more documentation.

6.1.2 Improvements to Proof Explainer Script

I think that for long clauses with many justifying clauses it is not enough to just say that you can "combine these clauses" to derive the new clause. The step should be broken down even further.

A relatively simple solution would be to break large steps which use many clauses into multiple smaller steps using resolution to justify each sub-step. For example, instead of saying that clause c can be derived by combining clauses x, y, and z. Instead, it would combine x and y, then y and z to produce clause c.

6.2 Distant Future Work

This section aims to tackle some of the "deeper" issues with using and explaining SAT proofs.

6.2.1 Encoding Theorems Made Easy

One of the challenges is verifying that the encoding states what you think it does. It might be better to use First Order Logic which allows quantifiers and predicates for example "There exists a winner for all profiles". In fact, with some restrictions you can use First Order Logic as syntactic sugar for SAT thus getting best of both worlds in terms of readability and the good performance offered by SAT.

Another promising approach might be to train an AI like GitHub Copilot [16] which is able to generate a function implementation given a natural language description. You could provide a natural language description of a property and train it to generate a CNF encoding or you could train it to generate a function which generates the CNF encoding.

6.2.2 Improvements to Showing Impossibility for SAT

The biggest improvement needed for explainability of sat proofs is *reducing* their length. It is very impractical to have to spend days going through them *and* the natural language description proofs are often so specific that *insights* are buried into the details.

Shortest Derivation of Clause To remedy the first issue, I propose a search procedure which uses the least number of steps of resolution to derive every clause. The current implementation retrieves what the SAT solver used; however, it is not necessarily the case that some other set of clauses wouldn't be able to derive a contradiction in fewer steps.

Abstract Pattern Matching This is a more general idea and I am not exactly sure how one would specifically approach this. The goal is to look for patterns in the proof-log. For example, it may be the case that you can eliminate alternative a for all profiles. Using the current setup, this would take 216 steps each corresponding to combining two clauses related to a profile.

Instead, it would be really useful if we could detect such a pattern (or a similar patterns) and automatically generate an explanation of all 216 steps. The explaination would look something like this "For all profiles we may combine a strategy-proofness clause with a majority clause and arrive at a successful manipulation where alternative a wins. Therefore, we may exclude alternative a as being the winner of any profile."

The problem one would have is that they would not really know what patterns to look for. I guess one possible idea could be to use compression or encode some patterns which appear in multiple theorems.

Generating or finding such patterns may be very useful when trying to generalise the impossibility result to an arbitrary number of alternatives and voters.

Animation The best possible explaination of an impossibility result, in my opinion, would be to have a visual diagram *and* a natural language description. In each step of the proof, the part of the MUS diagram which encodes the profiles in question would be highlighted and along with the natural language description, would assist the user in verifying the validity of the step.

One of the challenges is how to display long clauses where some of the atoms are negated. For example, a clause saying "profile 1 is won by alternative b or profile 2 is won by alternative not a" is not easy to convey in the MUS diagram format.

6.2.3 Argumentation Theory SAT Proofs Natural Language Explanation

Due to time-constraints I was unable to adapt the natural language description code to argumentation theory. It is possible to use the same approach I used for social choice except to apply it to argumentation theory.

One of the challenges that needs resolving is how to display the labelled AF for all voters. (For social choice an ascii-table display each voter's preference was sufficient, but AFs are typically displayed as graphs).

MUS of a proof in argumentation theory would typically contain more than one profile. This means that to visualise the MUS you would have two graphs - one displaying the constraints *between* profiles and one displaying the actual profiles. This may be challenging to understand.

7 Conclusions

Using SAT solvers to find impossibility results in social choice and argumentation theory works well when the hardest part about the impossibility result is verifying the base-case due to the very large search space.

This approach provides little help in cases where the main difficulty is proving the inductive step of the theory or in cases where a satisfying assignment of atoms exists for all cases you check.

Providing a concise, human-readable explanation of why a formula is unsatisfiable is sometimes easier than others. For example, both natural language and MUS explanations provided insight for Simplified GST and Theorem 3 from argumentation theory, but did not provide a lot of insight for full GST and Arrow's theorems. I think that this is partly because full GST and Arrow's theorems are more *difficult* to prove.

In summary, we have explored a relatively new approach for generating proofs and in the process, learnt some of the limitations keeping them from wider applicability such as limited use-cases and difficulty of explaining the results generated.

8 Reflection on Learning

8.1 Good Decisions

Overall, I think many things went right for this project. I had set up an elaborate initial plan where I had assigned specific goals and specific deadlines which I followed consistently throughout the report.

The regular supervisory meetings provided me with a sense of direction and did not allow me to indefinitely explore a rabbit-hole no matter how interesting I thought it was.

I chose a topic which was something I wanted to learn more about and explore which kept me motivated throughout the final year project. I could have chosen a topic where I would have had an easy time because I knew a lot about it already, but I am glad I did not.

8.2 Bad Decisions

One of the miscalculations I made is how much content I had to write about. My intuition was guided by the length of the book-chapter which I was basing my work on. As it turns out, the report goes into much more detail and thus takes longer to write and appear as more content than it really is.

The second miscalculation was starting to write the report too late. I was unable to start at the planned time (3 weeks before the deadline) due to unforeseen circumstances. This lead to significant stress and a lower quality report. Perhaps, it would have been better to focus on a smaller field and write a higher quality report.

8.3 Unsure Decisions

Finally, there are some decisions which I am not sure whether to classify as good or bad. For example, I spent a significant amount of time (about 3 weeks) trying to prove a conjecture in argumentation theory. This was some of the most difficult, satisfying, and insightful experiences for the whole project. It was something I was not sure I could do and thus it provided me with a fantastic opportunity to learn. However, not much came out of it in terms of results. In

fact, it is not talked about at all in the report. I would probably have gotten a higher mark had I only focused on the parts I knew I could do.

In a similar vain, I was asked to help write a conference proceedings paper by my work supervisor during the final weeks before submission. I could have declined, but with my help the paper had a higher chance of being accepted and thus I complied. Again, I probably could have gotten a higher mark by declining. Only time will tell whether I did the right thing.

References

[1] Tony Quach Derek Wenger Add Gritman, Anthony Ha. Conflict driven clause learning - github pages. url: https://cse442-17f.github.io/conflict-driven-clause-learning/.

[2] K. Appel and W. Haken. Every planar map is four colorable. *Bulletin of the American Mathematical Society*, 82(5):711 – 712, 1976.

[3] Kenneth J. Arrow. A difficulty in the concept of social welfare. *Journal of Political Economy*, 58(4):328–346, August 1950.

[4] Georg Bachmeier, Felix Brandt, Christian Geist, Paul Harrenstein, Keyvan Kardel, Dominik Peters, and Hans Georg Seedig. k-majority digraphs and the hardness of voting with a constant number of voters. *CoRR*, abs/1704.06304, 2017.

[5] Seulkee Baek, Mario Carneiro, and Marijn J. H. Heule. A flexible proof format for sat solver-elaborator communication. In Jan Friso Groote and Kim Guldstrand Larsen, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, pages 59–75, Cham, 2021. Springer International Publishing.

[6] Pietro Baroni, Martin Caminada, and Massimiliano Giacomin. An introduction to argumentation semantics. *The Knowledge Engineering Review*, 26(4):365–410, December 2011.

[7] Alejandro Barredo Arrieta, Natalia Díaz-Rodríguez, Javier Del Ser, Adrien Bennetot, Siham Tabik, Alberto Barbado, Salvador Garcia, Sergio Gil-Lopez, Daniel Molina, Richard Benjamins, Raja Chatila, and Francisco Herrera. Explainable artificial intelligence (xai): Concepts, taxonomies, opportunities and challenges toward responsible ai. *Information Fusion*, 58:82–115, 2020.

[8] Anton Belov and Joao Marques-Silva. MUSer2: An efficient MUS extractor1. *J. Satisf. Boolean Model. Comput.*, 8(3-4):123–128, December 2012.

[9] Daniel Le Berre. sat4j. `https://gitlab.ow2.org/sat4j/sat4j/-/releases/`, 14 Dec 2020.

[10] Armin Biere. linegling. `https://github.com/arminbiere/lingeling`, 5 Jan 2022.

[11] Richard Booth, Edmond Awad, and Iyad Rahwan. Interval methods for judgment aggregation in argumentation. In *Fourteenth International Conference on the Principles of Knowledge Representation and Reasoning*, 2014.

[12] Florian Brandl, Felix Brandt, Christian Geist, and Johannes Hofbauer. Strategic abstention based on preference extensions: Positive results and computer-generated impossibilities. *Journal of Artificial Intelligence Research*, 66, December 2019.

[13] Felix Brandt and Christian Geist. Finding strategyproof social choice functions via SAT solving. *Journal of Artificial Intelligence Research*, 55:565–602, March 2016.

[14] Felix Brandt, Christian Saile, and Christian Stricker. Voting with ties: Strong impossibilities via sat solving. In *Proceedings of the 17th International Conference on Autonomous Agents and MultiAgent Systems*, pages 1285–1293, 2018.

[15] Felix Brandt, Christian Saile, and Christian Stricker. Voting with ties: Strong impossibilities via sat solving. In *Proceedings of the 17th International Conference on Autonomous Agents and MultiAgent Systems*, AAMAS '18, page 1285–1293, Richland, SC, 2018. International Foundation for Autonomous Agents and Multiagent Systems.

[16] Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde de Oliveira Pinto, Jared Kaplan, Harrison Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, Alex Ray, Raul Puri, Gretchen Krueger, Michael Petrov, Heidy Khlaaf, Girish Sastry, Pamela Mishkin, Brooke Chan, Scott Gray, Nick Ryder, Mikhail Pavlov, Alethea Power, Lukasz Kaiser, Mohammad Bavarian, Clemens Winter, Philippe Tillet, Felipe Petroski Such, Dave Cummings, Matthias Plappert, Fotios Chantzis, Elizabeth Barnes, Ariel Herbert-Voss, William Hebgen Guss, Alex Nichol, Alex Paino, Nikolas Tezak, Jie Tang, Igor Babuschkin, Suchir Balaji, Shantanu Jain, William Saunders, Christopher Hesse, Andrew N. Carr, Jan Leike, Joshua Achiam, Vedant Misra, Evan Morikawa, Alec Radford, Matthew Knight, Miles Brundage, Mira Murati, Katie Mayer, Peter Welinder, Bob McGrew, Dario Amodei, Sam McCandlish, Ilya Sutskever, and Wojciech Zaremba. Evaluating large language models trained on code. *CoRR*, abs/2107.03374, 2021.

[17] Stephen A. Cook. The complexity of theorem-proving procedures. In *Proceedings of the Third Annual ACM Symposium on Theory of Computing*, STOC '71, page 151–158, New York, NY, USA, 1971. Association for Computing Machinery.

[18] Martin Davis and Hilary Putnam. A computing procedure for quantification theory. *J. ACM*, 7(3):201–215, jul 1960.

[19] I˛sıl Dillig. Overview cs389l: Automated logical reasoning lecture 2 ...

[20] Phan Minh Dung. On the acceptability of arguments and its fundamental role in nonmonotonic reasoning, logic programming and n-person games. *Artificial Intelligence*, 77(2):321–357, 1995.

[21] Ulle Endriss. *Trends in Computational Social Choice*. Lulu.com, 2017.

[22] Johannes K Fichte, Markus Hecher, Piotr Gorczyca, and Ridhwan Dewoprabowo. Afolio dpdb–system description for iccma 2021 `http://argumentationcompetition.org/2021/downloads/a-folio-dpdb.pdf`.

[23] Allan Gibbard. Manipulation of voting schemes: A general result. *Econometrica*, 41(4):587, July 1973.

[24] E. Goldberg and Y. Novikov. Verification of proofs of unsatisfiability for cnf formulas. In *2003 Design, Automation and Test in Europe Conference and Exhibition*, pages 886–891, 2003.

[25] Aric A. Hagberg, Daniel A. Schult, and Pieter J. Swart. Exploring network structure, dynamics, and function using networkx. In Gaël Varoquaux, Travis Vaught, and Jarrod Millman, editors, *Proceedings of the 7th Python in Science Conference*, pages 11 – 15, Pasadena, CA USA, 2008.

[26] Marijn J.H. Heule, Warren A. Hunt, and Nathan Wetzler. Trimming while checking clausal proofs. In *2013 Formal Methods in Computer-Aided Design*, pages 181–188, 2013.

[27] J. D. Hunter. Matplotlib: A 2d graphics environment. *Computing in Science & Engineering*, 9(3):90–95, 2007.

[28] Alexey Ignatiev, Antonio Morgado, and Joao Marques-Silva. PySAT: A Python toolkit for prototyping with SAT oracles. In *SAT*, pages 428–437, 2018.

[29] Jean-Marie Lagniez, Emmanuel Lonca, Jean-Guy Mailly, and Julien Rossit. Design and results of ICCMA 2021. *CoRR*, abs/2109.08884, 2021.

[30] Canonical Group Limited. Lxde. `https://www.lxde.org/`, 2022.

[31] Canonical Group Limited. Ubuntu. `https://apps.microsoft.com/store/detail/ubuntu/9PDXGNCFSCZV?hl=en-us&gl=US`, 2022.

[32] Fangzhen Lin and Pingzhong Tang. Computer-aided proofs of arrow's and other impossibility theorems. In *AAAI*, 2008.

[33] J.P. Marques Silva and K.A. Sakallah. Grasp-a new search algorithm for satisfiability. In *Proceedings of International Conference on Computer Aided Design*, pages 220–227, 1996.

[34] Michael Morreau. Arrow's Theorem. In Edward N. Zalta, editor, *The Stanford Encyclopedia of Philosophy*. Metaphysics Research Lab, Stanford University, Winter 2019 edition, 2019.

[35] Nikalasso. Minisat. `https://github.com/niklasso/minisat`, 25 Sep 2013.

[36] Olga Ohrimenko, Peter J. Stuckey, and Michael Codish. Propagation = lazy clause generation. In Christian Bessière, editor, *Principles and Practice of Constraint Programming – CP 2007*, pages 544–558, Berlin, Heidelberg, 2007. Springer Berlin Heidelberg.

[37] Christian Strasser and G. Aldo Antonelli. Non-monotonic Logic. In Edward N. Zalta, editor, *The Stanford Encyclopedia of Philosophy*. Metaphysics Research Lab, Stanford University, Summer 2019 edition, 2019.

[38] Terence Tao. Perelman's proof of the poincaré conjecture: a nonlinear pde perspective, 2006.

[39] Balyo Tomas, Nils Froleyks, Marijn Heule, Markus Iser, Matti Järvisalo, Martin Suda, Helsinki Institute for Information Technology, Constraint Reasoning, Optimization research group / Matti Järvisalo, and Department of Computer Science. Proceedings of sat competition 2021 : Solver and benchmark descriptions, Jan 1970.

[40] Guido Van Rossum. *The Python Library Reference, release 3.8.2, Itertools*. Python Software Foundation, 2020.

[41] Guido Van Rossum. *The Python Library Reference, release 3.8.2, Pickle*. Python Software Foundation, 2020.

[42] Guido Van Rossum and Fred L. Drake. *Python 3 Reference Manual*. CreateSpace, Scotts Valley, CA, 2009.

[43] Jouko Väänänen. Second-order and Higher-order Logic. In Edward N. Zalta, editor, *The Stanford Encyclopedia of Philosophy*. Metaphysics Research Lab, Stanford University, Fall 2021 edition, 2021.

[44] Andrew Wiles. Modular elliptic curves and fermat's last theorem. *Annals of Mathematics*, 141(3):443–551, 1995.

Appendix

	MINISAT			SAT4j			lingeling		
	simplified	full	arrow	simplified	full	arrow	simplified	full	arrow
Run 1	0.004	0.019	0.377	0.024	0.04	0.182	0	0.3	0.1
Run 2	0.004	0.019	0.386	0.024	0.039	0.204	0	0.3	0.1
Run 3	0.004	0.018	0.348	0.022	0.04	0.199	0	0.3	0.1
Run 4	0.001	0.017	0.372	0.022	0.038	0.153	0	0.3	0.1
Run 5	0.004	0.009	0.344	0.022	0.038	0.167	0	0.3	0.1
Run 6	0.004	0.017	0.371	0.022	0.038	0.213	0	0.3	0.1
Run 7	0.004	0.017	0.388	0.022	0.038	0.168	0	0.3	0.1
Run 8	0.004	0.019	0.370	0.023	0.037	0.153	0	0.3	0.1
Run 9	0.001	0.009	0.365	0.023	0.039	0.199	0	0.3	0.1
Run 10	0.004	0.009	0.365	0.022	0.04	0.181	0	0.3	0.1
Mean	**0.003**	**0.015**	**0.369**	**0.023**	**0.039**	**0.182**	**0**	**0.3**	**0.1**

Figure 24: This figure shows all SAT solvers performance on the 10 runs on each of the 3 theorems.