Exact machine learning

Improving space and speed of MaxSAT solvers for correlation clustering

M. Marchal

Exact machine learning

Improving space and speed of MaxSAT solvers for correlation clustering

by

M. Marchal

to obtain the degree of Master of Science
at the Delft University of Technology,
to be defended publicly on Friday 27th of August, 2021 at 9:30 AM.

Student number:	4360842	
Project duration:	November, 2020 – August, 2021	
Thesis committee:	Dr. Ir. Leo J. J. van Iersel	Associate Professor
	Dr. E. Demirović,	Assistant Professor

TUDelft

Acknowledgements

I would like to thank Emir Demirović for tirelessly assisting me with this thesis. He has provided me with useful feedback, comments and suggestions during every stage of this project. Due to the pandemic, we have not been able to meet in person but thanks to our weekly zoom meetings, this has not posed a problem.
Finally, I would like to thank Ingela and René, my parents, who have supported me throughout my academic career and beyond.

Preface

If you want to solve problems, you don't just solve the ones that are there. You find and make more, and go after the impossible ones. Fostering a love and obsession with problems is how you solve problems.

by Michael Stevens a.k.a. Vsauce

Abstract

Clustering is an important unsupervised learning task, with many applications in machine learning, computer vision, formal program verification and finance. Heuristic approaches such as local search are an excellent strategy for estimating the optimal solution, but they run the danger of getting stuck in local optima. Furthermore, it is non-trivial to add user-specific constraints or other objectives to these approaches without compromising their functionality, which makes them inflexible. Generic optimisation techniques such as MaxSAT and Integer Programming are more versatile, but they scale poorly on bigger datasets. Additionally, the fact that the search space of correlation clustering is inherently symmetric only exacerbates the scalability issue, especially for exhaustive search strategies.

In this work, a novel customised search algorithm for correlation clustering is proposed. It aims to unite the flexibility of exhaustive optimisation techniques with an improved scalability and the possibility to declaratively specify domain specific knowledge in the form of a propagator interface. The algorithm uses a specialised hybrid MaxSAT solver with a symmetry-free encoding. Traditionally, this encoding would restrict scalability due to the prohibitive space complexity. However, this problem is avoided by using lazy data structures to represent key components of the encoding, which reduces the space requirement by several orders of magnitude. In addition, a novel encoding is presented that drastically reduces the runtime for sparse graphs. The algorithm employs incremental bounding techniques that under certain conditions reduce the number of decisions required by the solver to prove optimality.

Contents

1 Introduction

Clustering is one of the core problems in the field of data analysis. The objective of clustering is to divide data points into groups in a way that unveils an underlying structure in the data. Clustering is a versatile tool and it can be applied to many different contexts due to the fact that it does not require the dataset to contain explicit labels. Rather, it requires a notion of similarity; similar data points are mostly placed in the same cluster, while dissimilar points are not. Examples of fields using clustering are pattern recognition, machine learning, image analysis, bioinformatics and finance. For example, in the field of finance, the set of stocks can be clustered in order to obtain a classification of stocks into different *sectors*. In machine learning, clustering can be used to preprocess the data and remove outliers. Figure 1.1 shows an example of a graph which we might want to analyse using a clustering technique. Given the similarities between nodes (indicated by colours of the edges), the question is: which nodes belong to which group?

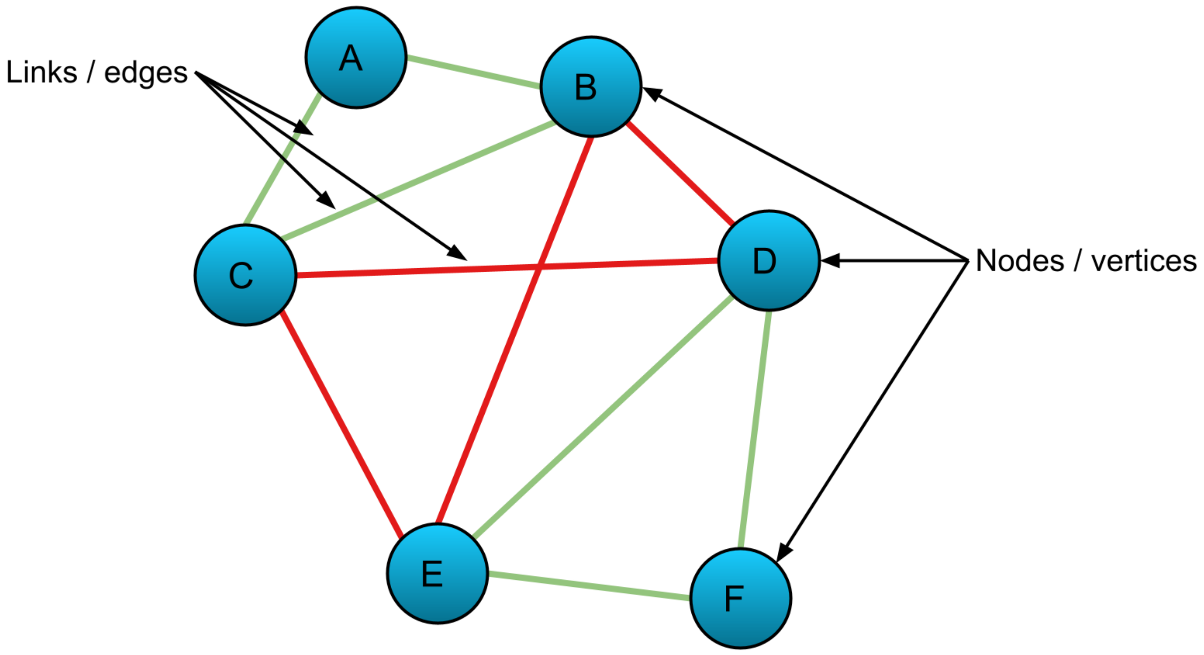


Figure 1.1: An example of a clustering problem: which nodes (labelled A through F) belong to which cluster? Based on the similarities between nodes, indicated by colour, we might discover an underlying structure.

There are several types of clustering definitions. For example, in some definitions, clusters are ‘fuzzy’ and data points can belong to multiple clusters with differing degrees of membership. Other types of clustering include hierarchical clustering, where clusters are subdivided into a hierarchical

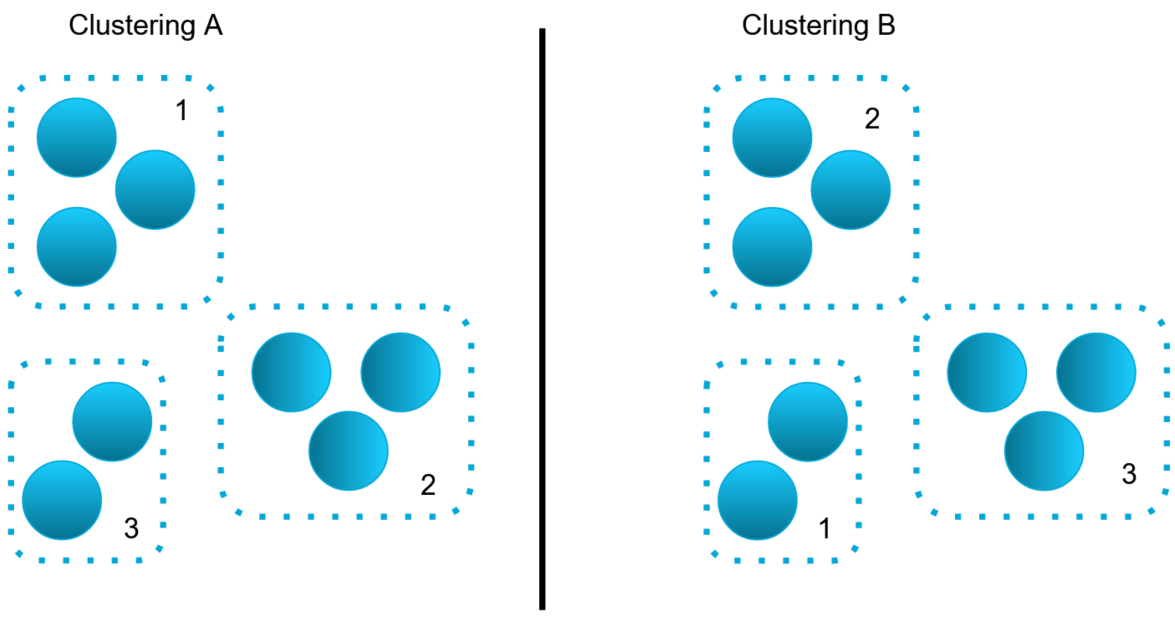


Figure 1.2: An example to illustrate the symmetry that is inherent to clustering: dashed blue boxes indicate a cluster, and the number inside represents the cluster index. Clustering A and B are clearly different, but they are simply different appearances of what is essentially the same thing.

structure. Interestingly, one thing that all clustering paradigms have in common is that the set of all possible clusterings is highly symmetric, meaning that many different clusterings in the space of all clusterings are essentially the same thing. This is because relabeling the indices of clusters has no real effect on the semantics. Figure 1.2 illustrates this by showing two distinct clusterings A and B. We can argue that A and B differ in the same way that your right and left hands differ: they are mirror images of what is essentially the same thing. Generally speaking, a symmetric search space is troublesome for search algorithms.

Correlation clustering In this thesis, we investigate the weighted variant of the *correlation clustering* paradigm by Bansal et al., 2004. Given a network of data points with links between points either labelled 'positive', 'negative' or 'neutral', divide these points into non-overlapping groups in such a way that the number of negative links *within* groups and the positive links *between* groups is minimised. Contrary to popular clustering algorithms like k-means, k-sum and k-center, correlation clustering does not require specifying a number of clusters a priori; rather, the number of clusters is part of the optimal solution (Becker, 2005). This definition is useful for tasks where it is difficult to determine an upper bound on the number of clusters, such as clustering web pages (Becker, 2005) or detecting communities online.

Solving correlation clustering Correlation clustering can be solved in many different ways. Broadly speaking, we can distinguish two categories: heuristic and exact approaches. The former approach attempts to find a sufficiently good solution without worrying about whether there are better solutions, whereas the latter is concerned with finding (one of) the best solutions. Correlation clustering is NP-complete (proven by Bansal et al., 2004), which means the runtime of any exact approach is expected to be exponential in the input size (assuming $P \neq NP$). Nevertheless, with current technology and advances in the field of optimisation, it is possible to solve moderately sized problems within a reasonable time. Additionally, there are several strong benefits to solving a problem to optimality, instead of relying on heuristics. These facts together warrant further investigation into the efficiency and scalability of exact approaches for correlation clustering.

Optimisation The intersection of data analysis and constraint programming has been identified as a promising and high-potential research field. Berg and Järvisalo, 2017 has demonstrated that a Boolean optimisation strategy called MaxSAT is competitive in the space of exact solvers for the problem of correlation clustering. However, there are several drawbacks associated with the model proposed by Berg and Järvisalo, 2017. In this thesis, we discuss these limitations and suggest novel techniques to mitigate their impact.

1.1. Exact optimisation

Using a heuristic strategy is a natural response when presented with a problem that is NP-complete. Understandably so, since the exponentially scaling runtime of exact algorithms and our limited time on this planet are not a great combination. However, there are several serious drawbacks associated with heuristic methods, both from a strategic as well as an ethical standpoint. We introduce these shortcomings here, and we motivate how and why exact approaches are effective at countering them. We subsequently motivate the reason for choosing a MaxSAT based approach instead of other exact methods.

Shortcomings of non-exact methods

Suboptimality Approximative methods (or: approximation algorithms) typically have a very short runtime, but this comes at a cost of solution quality. It is possible for such an algorithm to have some type of quality guarantee. For example, Bansal et al., 2004 proposes an $\mathcal{O}(\log n)$ approximation algorithm, meaning that the optimal solution is at most $\mathcal{O}(\log n)$ times better than the solution produced by the algorithm. However, not every approximative method has such a guarantee; it is often a non-trivial task to derive and prove such a property. In the case where there is no quality guarantee, this puts us at a strategic disadvantage: a possible competitor, who is also attempting to solve the problem, may have a solution that is significantly better than the one produced by our algorithm. Hebrard and Katsirelos, 2020 give a compelling argument in favor of having a provably optimal solution, namely that when there is a very large sum of money (or, more generally: resources) involved, the difference between an almost-optimal and a truly optimal solution is significant. The problem of suboptimality is not an issue when using exact algorithms since we know for an absolute fact that the solution produced by an exact algorithm cannot be improved any further.

Imposing constraints Another disadvantage of non-exact methods, heuristics in particular, is their inflexibility with regards to imposing constraints on the solutions that the algorithm produces. A heuristic is a procedure that builds a solution by making local, greedy decisions. In general, these greedy decisions do not respect any additional constraints imposed on the problem, which means the heuristic algorithm will violate them. Of course, it might be possible to modify the heuristic so that it does respect the constraints, but this is not straightforward and essentially comes down to developing a whole new algorithm. Moreover, if the original heuristic algorithm had any performance guarantees, the new heuristic will not possess these.

Why would one want to impose additional constraints on a problem? The reason for this is the mismatch between an abstract model of a problem and its real-world counterpart. For correlation clustering, there are several examples of constraints that one might want to impose on the problem:

1. Disallowing any clusters from having more (or fewer) than x points in them.
2. Disallowing certain specific combinations of points to be co-clustered. For example, due to external factors, it is required to not have points a, b and c in the same cluster.
3. Demanding that the number of clusters is exactly equal to some number y.
4. Demanding that the number of negative links within a cluster is no more than z.

With the increasing prevalence of decision-making algorithms in modern society, it is of utmost importance to ensure that these algorithms do not put individuals at a disadvantage by discriminating against them based on socially sensitive features. Imposing constraints on the problem can be an effective tool in reducing this type of harmful effects. Another example of an undesirable solution is the following example:

Suppose we are trying to partition a team of people into groups in which they will work together on a project. We know that some pairs of people enjoy working together, and some don't. In order to solve the problem and create desirable groups of people, we can cast it as a correlation clustering problem. However, in this setting, we would like to add an additional constraint to the problem, namely: groups of people should be at least of size two. Otherwise, we would have people working alone on the project and this is undesired.

Shortcomings of exact methods

The arguments presented above give reasons for using exact methods over approximate ones. Unfortunately, exact methods have several problems, primarily regarding the scalability of such approaches. For example, the Integer Linear Programming (ILP) formulation for correlation clustering is proposed in Ailon et al., 2005 and Gael and Zhu, n.d. This formulation eliminates a large portion of the symmetries that are inherent to clustering, but Berg and Järvisalo, 2017 remark that it scales poorly due to the fact that the number of constraints is cubic in the number of points. Another approach is the Quadratic Integer Programming (QIP) formulation by Bonizzoni et al., 2008. The number of constraints of this formulation scales better than the ILP formulation under certain conditions, but Berg and Järvisalo, 2017's findings suggest that it scales poorly on larger datasets in practice. In their experiments, the QIP formulation was unable to solve any of the instances with more than 50 points. The authors conjecture that this is due to the non-convexity of the objective function. Another possible explanation is that this QIP formulation does not eliminate symmetries in the search space.

MaxSAT approaches Berg and Järvisalo, 2017 propose three MaxSAT encodings of correlation clustering which resemble the ILP and QIP formulations. They find that all three encodings are superior to the QIP and ILP formulations in their runtime experiments.
The first encoding is based on the ILP formulation, and we will refer to it as the transitive encoding. Similar to its ILP-cousin, this encoding suffers greatly from scalability due to the cubic number of clauses required to represent an instance of correlation clustering. The second encoding resembles the QIP formulation, and will be referred to as the unary encoding. This encoding allows us to specify an upper bound on the number of clusters. It requires slightly fewer variables and clauses under certain conditions, and it becomes slightly more compact as the graph's density decreases. The third and final encoding represents the cluster index of data points using boolean variables. This encoding is more compact than unary, requiring significantly fewer variables and clauses.

1.2. Contributions

In this work, we propose a MaxSAT solver that is specifically engineered for correlation clustering. Features of the solver include a highly compact way of lazily storing clauses, a novel encoding that (under certain conditions) sharply reduces the number of variables required and a technique that can incrementally compute bounds on the problem.
The research questions are the following:

- How can we improve scalability of representational complexity for correlation clustering instances in MaxSAT?
- How can we leverage the domain knowledge of a problem-specific solver?

Scalability This work proposes novel techniques that aim to improve the runtime and space requirements for larger datasets. The runtime is reduced by using a novel encoding and by dynamically computing bounds. Furthermore, the scalability is addressed by using a lazy internal representation of the problem.

Hybrid problem-specific solver The algorithm that is used here maintains a representation of the graph while solving. This makes the algorithm a hybrid solver, because pure MaxSAT solvers can only reason about their basic elements: variables, literals and clauses. The graph representation is able to derive lower bounds (see chapter 5) and it is used to create clausal explanations for the sparse encoding (see chapter 4).

1.3. Organisation

This thesis is organised as follows. In chapter 2, we state the problem formally and define any technical terms from the fields of graph theory, (Max)SAT and clustering. In chapter 3, we discuss relevant scientific work and relate it to the research questions. Chapters 4 and 5 explain the technical aspect of the contributions of this thesis; the former explains the novel contributions to the encoding by Berg and Järvisalo, 2017 and the latter presents a novel bounding algorithm for correlation clustering. Chapter 6 features an experimental evaluation of the performance of our work on real-world datasets, comparing it to the baseline by Berg and Järvisalo, 2017. Furthermore, the efficacy of the bounding techniques presented in chapter 5 is evaluated using appropriate metrics. Chapter 8 summarises the most important findings of the experimental evaluation and draws conclusions. Finally, chapter 7 speculates on promising directions for future work and reflects on the findings of the experimental evaluation.

2 Problem setting

Preliminaries

This thesis makes contributions to MaxSAT solving for correlation clustering. In this chapter, we formally define concepts related to (Max)SAT and clustering; in the second part we define the problem formally with examples.

2.0.1. Graphs

A graph G is a tuple (V, E). V is the set of *nodes* (or vertices) and can be any set of distinct identifiers. Sometimes, integers are used to identify nodes, but any symbol or sequence of symbols can be used to identify a node in a graph. A common convention is to let $N = |V|$. E is the set of *edges* of the graph. An edge represents a relation between two vertices. In general, an edge can have any number of properties (for example: capacity, weight, color, etc.). In correlation clustering, the only relevant property of an edge is the weight (or similarity). We assume that the similarity is symmetric which implies that an edge from $u \in V$ to $v \in V$ is the exact same thing as an edge from v to u. In this work, it is assumed that not necessarily every pair of nodes $u, v \in V$ have an edge between them. Finally, a common convention is to let $|E| = M$.
Graphs are commonly used to represent relations between objects. A well-known alternative name for graphs is a network. Graphs can be directed, meaning that edges are a 'one way street', or undirected; in the latter case, the edge going from i to j is considered equivalent to the edge going from j to i. In our setting of correlation clustering, edges represent similarities between objects. Since similarity is a symmetric property, we assume in this work that graphs are undirected. Bansal et al., 2004's definition of correlation clustering does allow for asymmetric similarities, but Berg and Järvisalo, 2017 shows that the assumption of symmetric similarities can be done without loss of generality. We refer the interested reader to Berg and Järvisalo, 2017 section 2.2.

Types of graphs Graphs can be classified based on different properties, such as edge density, connectedness or maximum path length. For this thesis, it is important to discuss one particular type of graph: the tree. A tree graph $G_T = (V_T, E_T)$ has several properties:

1. If $|V_T| = N$, then $|E_T| = N - 1$.
2. For every $v_i, v_j \in V_T$, there is exactly one path from v_i to v_j and vice-versa.
3. G_T is connected, and removing any edge from E_T would disconnect the graph.

A special type of tree is the spanning tree. Suppose we have a graph $G_0 = (V_0, E_0)$, which is not a tree. Then we can define a spanning tree over G_0. Call it $G_{spanning} = (V_{spanning}, E_{spanning})$; then $G_{spanning}$ has the following properties:

1. $G_{spanning}$ is a tree.

2. $V_{spanning} = V_0$.
3. $E_{spanning} \subseteq E_0$.

In chapter 4, we make use of this concept by introducing an algorithm that computes a spanning tree over a cluster using a partial solution.

2.0.2. Clustering

Clustering is a broad term used to describe a large number of problems that are all related by their general objective: dividing points up in a group in some optimal way. Clustering models can vary wildly and clustering definitions are sometimes the origin of confusion. Estivill-Castro, 2002 argues that this is the case because the notion 'cluster ' cannot be well-defined, and that the problem of clustering is highly context-dependent. Estivill-Castro, 2002 gives further recommendations for classifying clustering models, and we refer the interested reader to their work. In this thesis however, we adopt one specific clustering model coined by Bansal et al., 2004.

Correlation clustering Bansal et al., 2004 define the problem of correlation clustering. It is a variant with qualitative similarity information, hard clusters and a specific objective function. We define them here.

The *similarity matrix* W is a matrix that carries the information about edge similarities. W is square and symmetric, and its elements are members of $\{-1, 1, 0\}$. A similarity of -1 indicates that the two endpoints of the edge are dissimilar, while a value of 1 indicates they are similar. If, for $u, v \in V$, there is no edge between u and v, the similarity value for these two points is 0.

A *solution* for correlation clustering is any function $f : V \longrightarrow \mathcal{N}$ that partitions V. That is, f should assign every $v \in V$ to exactly one cluster.

The *cost* of a clustering f is defined in terms of 'mistakes ' made by f. A good clustering will assign points with high similarity to the same cluster, while assigning points with high dissimilarity to different clusters. The cost of a solution is the total number of mistakes made by f. There are two types of mistakes:

- A positive mistake: the case where the edge is labelled positive, yet the endpoints of the edge are assigned to different clusters. The right image in figure 2.1 shows an example of a positive mistake.
- A negative mistake: the case where the edge is labelled negative, yet the endpoints of the edge are assigned to the same cluster. The left image in figure 2.1 shows an example of a negative mistake.

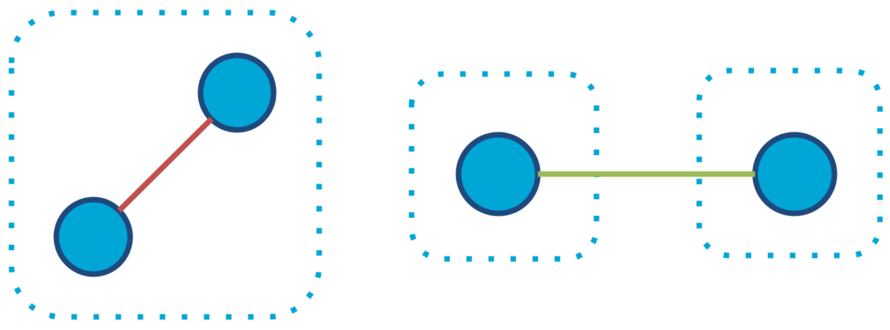


Figure 2.1: Visual representation of the objective function shown in equation 2.7. Dashed blue boxes indicate a cluster. The left hand side shows a negative mistake (a negative edge *within* a cluster) and the right hand side shows a positive mistake (a positive edge *between* clusters).

2.0.3. Mathematical optimisation

Mathematical optimisation is an umbrella term for any technology that selects a best element from a set of alternatives. The topic of this thesis is concerned with finding an optimal clustering from the large set of all possible clusterings, and can therefore be considered a mathematical optimisation problem. We briefly describe the most commonly used techniques in this section. The optimisation technique that is used in this thesis is discussed later in more detail.

Constraint programming Constraint programming is a generic, declarative technique that is used to model real-world problems. It can be seen as a cousin of the algorithm; an algorithm specifies how to find a solution to a problem, whereas a constraint program only specifies *properties* of a valid and desirable solution.
A constraint program typically consists of a set of core elements:

1. Variables
2. A domain for every variable. A domain is a finite set of values that a variable can take.
3. A set of constraints. A constraint can be seen as a 'joint domain' of a set of variables, forbidding certain combinations of value assignments to these variables. An example of a constraint is $x_1 + x_2 \leq 5$.
4. An objective function. This is a function mapping the set of variables to a real number. The objective can be to either minimise or maximise this function.

Integer Linear Programming (ILP) & Quadratic Integer Programming (QIP) ILP and QIP are techniques used to solve optimisation problems. The techniques are characterised by the fact that variables are discrete, meaning they can only take on integer values, and the fact that constraints are a linear combination of the variables. QIP allows for an objective function with quadratic terms, whereas ILP restricts the objective function to be a linear combination of the variables. For example, the expression $4x_1x_2 + 2x_0$ is a valid QIP objective, but not a valid ILP objective.

2.0.4. SAT

The satisfiability problem is arguably the most fundamental problem in theoretical computer science. It was used in the first NP-completeness proof by Cook, 1971 and consequently spawned a new area of research: complexity theory. Practical applications of SAT are plentiful, ranging from formal program verification to scheduling.

Conventions and nomenclature The Satisfiability problem is concerned with whether there exists an assignment of values to variables $\{x_1, x_2, ...x_M\}$ such that $\phi(x_1, x_2, ...x_M)$ evaluates to true. A short overview of commonly used terms in SAT is the following:

- Variable: the fundamental atom of a SAT formula. It can be assigned true or false. In the context of SAT-solving, a variable can also be unassigned.
- Literal: either a variable, or its negation. If x_1 is a variable, then x_1 and $\neg x_1$ are literals. If x_1 is assigned false, then the literal $\neg x_1$ is considered to be true.
- Clause: a disjunction of literals.
- Unit clause: a clause containing only one literal.
- (Boolean) formula / proposition: a conjunction of clauses.
- SAT/Satisfiable: When a boolean formula has an assignment that causes it to evaluate to true.
- UNSAT/Unsatisfiable: When every possible assignment to variables leads to the boolean formula evaluating to false.

SAT problems are logical propositions that are specified using clauses. It is a common convention to assume the clauses are represented in conjunctive normal form (CNF), i.e. an AND of ORs. A clause consists of a finite number of literals; a literal is either a variable, or its negation.

Example of a SAT problem Equation 2.1 shows an example of a logical proposition in CNF. The boolean formula has seven variables and four clauses.

$$\phi = (\neg x_1 \vee x_2 \vee x_5) \wedge (\neg x_4 \vee \neg x_7) \wedge (x_3 \vee x_6 \vee \neg x_5) \wedge (\neg x_7) \tag{2.1}$$

The right-most clause, $\neg x_7$, only contains one literal and it is a common convention to call this a unit clause. ϕ is satisfiable because we can find values for $\{x1, ...x_7\}$ such that ϕ evaluates to true. For example, if we set x_2, x_3, x_5, x_6 to true while setting $x1, x_4, x_7$ to false, the formula will evaluate to true. Note that this solution is not unique; there are others.

2.0.5. SAT solving

One of the reasons for the widespread use of of SAT in many applications is that *Conflict-Driven Clause Learning*-SAT solvers are so effective in practice (Biere, Heule, et al., 2009). In this section, we discuss several core concepts that are needed in order to grasp the concept of CDCL. Biere, Biere, et al., 2009 gives an excellent high-level description of the CDCL algorithm; we repeat the pseudocode here and refer to it throughout this section.

DPLL DPLL, named after its inventors Davis et al., 1962, is a search strategy specifically engineered for SAT problems. It works by selecting any 'unconsidered' literal x and assigning it to true. It then simplifies the formula given the chosen literal and its value, then recursively solves the new boolean formula. If the new formula is unsatisfiable (specifically: not all clauses are satisfied, and the domain of one or more variables is empty), the algorithm assigns x to false, simplifies the formula and tries to solve it again. This way, the search space is exhaustively searched. If both the 'true' and 'false' branch report unsatisfiability, the program returns UNSAT. Simplification of a formula is done by removing any satisfied clauses from ϕ and eliminating false literals from clauses.

Propagation An important concept in SAT solving is propagation. Propagation lies at the core of modern SAT algorithms, since it cuts down the search space by inferring information from a partial solution, as well as detecting conflicts when one occurs. Propagation is done at two places in the CDCL algorithm: firstly at the beginning, in order to check whether the formula is UNSAT without making any assumptions, and secondly deeper in the loop in order to infer unit information and detect conflicts. Propagation is done by the *UnitPropagation(ϕ, A)* procedure in algorithm 1.

Suppose there is a unit clause, $\omega = x_1$. We know that if we assign x_1 to false, ω will evaluate to false, and so the formula becomes unsatisfiable. Therefore, we are certain that *if* the formula has a satisfying solution, it will assign x_1 to true. Therefore, we can *infer* that x_1 must be true. Propagation is the event where we can infer the value of certain literals *as a result* of assigning a value to another literal. Consider the following example, where we start with the following boolean formula:

$$\phi = (x_1 \vee x_2) \wedge (\neg x_1 \vee \neg x_2) \tag{2.2}$$

Now suppose we assign x_1 to true; simplification of the formula yields ϕ':

$$\begin{aligned} \phi' &= (\top \vee x_2) \wedge (\bot \vee \neg x_2) \\ &= (\neg x_2) \end{aligned} \tag{2.3}$$

Now, ϕ' has a clause that is unit, so we can say that as a result of assigning x_1 to true, we have inferred that x_2 must be false. Moreover, the *explanation* for x_2 becoming false is given by the conjunction of literals that led to x_2's falsehood. In our example here, the explanation for $\neg x_2$ would be x_1. Note that an explanation can consist of more than one literal.

Propagation can sometimes lead to conflicts. Suppose that we have two clauses $\omega_1 = (x_1 \vee x_2)$ and $\omega_2 = (x_1 \vee \neg x_2)$. Now suppose the solver decides to set x_1 to false; this causes both ω_1 and ω_2 to become unit, leading to the propagation of unit information. However, the unit information is contradictory: ω_1 will infer that x_2 is true, but ω_2 will cause x_2 to be false. This is what is called a conflict, and the solver will have to backtrack to resolve the conflict.

CDCL Conflict-driven clause learning, or CDCL, is a technique proposed by Marques and Silva, 1996 and Bayardo Jr and Schrag, 1997. It greatly improves the speed of SAT solvers by learning from conflicts that happen during search. Suppose that unit propagation of the current partial assignment leads to a conflict. Naturally, this is the result of the current partial solution A being unsatisfiable. Not every assigned literal $x \in A$ is responsible for the conflict, but a subset $A' \subseteq A$ is. Since A' caused a conflict, we know that the following is true:

$$\left(\bigwedge_{x \in A'} x\right) \longrightarrow \text{UNSAT} \tag{2.4}$$

Now, we consider the contrapositive of the proposition, which is logically equivalent:

$$\begin{aligned} \neg\text{UNSAT} &\longrightarrow \neg\left(\bigwedge_{x \in A'} x\right) \\ = \text{SAT} &\longrightarrow \bigvee_{x \in A'} \neg x \end{aligned} \tag{2.5}$$

The bottom equation reads: if there is a satisfying assignment, then the disjunction $\bigvee_{x \in A'} \neg x$ must be true. A disjunction of literals is a clause, and that is exactly what CDCL 'learns' as a result of the conflict. Algorithm 1 gives the pseudocode for CDCL SAT-solvers, which is taken from Biere, Heule, et al., 2009. The procedure of extracting a clause from a conflict is performed by the subprocedure *ConflictAnalysis(ϕ, A)* in algorithm 1. A refers to the solver's current partial solution, B refers to the decision level to which the solver should backtrack after a conflict and finally dl stands for the current decision level.

Algorithm 1: Pseudocode for a typical CDCL algorithm. This pseudocode was adopted from Biere, Biere, et al., 2009.

```
Result: True if $\phi$ is SAT, and false otherwise.
if UnitPropagation($\phi$, A) yields a conflict then
    return UNSAT
end
dl ←0;
while NotAllVariablesAssigned($\phi$, A) do
    (x, v) ←SelectBranchingVariable($\phi$, A);
    dl ←dl + 1;
    A ←A $\cup (x, v)$;
    if UnitPropagation($\phi$, A) yields a conflict then
        B ←ConflictAnalysis($\phi$, A);
        if B < 0 then
            return UNSAT
        end
        else
            BackTrack($\phi$, A, B);
            dl ←B
        end
    end
end
```

For completeness, we provide a short summary of the subprocedures in algorithm 1. The short summaries are also by Biere, Heule, et al., 2009 and we paraphrase them here.

- *UnitPropagation(ϕ, A)*: iteratively applies the unit clause rule to clauses. When a conflict is detected, this procedure returns false.

- *PickBranchingVariable(ϕ, A)*: selects a variable from the set of unassigned variables, and assigns it a value.
- *ConflictAnalysis(ϕ, A)*: analyzes the most recent conflict and learns a clause from it, which it adds to the clause database. Returns the decisions level B to which the solver should backtrack after processing the conflict.
- *Backtrack(ϕ, A, B)*: instructs the solver to backtrack to the decision level given by B, undoing all variable assignments in A that belong to a higher decision level that B.
- *AllVariablesAssigned(ϕ, A)*: tests whether all variables have been assigned. If this is the case, then that means the solver has found a solution and it should return SAT.

CDCL is extremely effective at solving SAT problems. Open source SAT solvers are able to solve real-world boolean formulas consisting of millions of variables. However, since SAT is NP-complete, it is guaranteed that there are instance classes which are hard to solve (unless P = NP). Ganesh and Vard, n.d. investigate the reasons for why SAT solvers perform so well on real-world instances, but they do not give a definitive explanation.

Building your own solver Eén and Sörensson, 2004 Implementing a SAT solver is by no means a trivial task. Fortunately, Eén and Sörensson, 2004 gives a detailed step-by-step explanation for implementing one. For this thesis, the SAT solver proposed by Eén and Sörensson, 2004 was implemented in C++ as a preparatory task.

2.0.6. MaxSAT

MaxSAT is the optimisation variant of SAT: the goal is to find an assignment that maximises the number of satisfied clauses. MaxSAT is an excellent tool to solve mathematical optimisation due to its ability to both translate constraints and objectives as clauses.

Variations There are different variants of MaxSAT. For example, in weighted MaxSAT, every clause has a weight and the goal is to maximise the weight of the satisfied clauses. Partial weighted MaxSAT is a variant where the set of clauses consists of hard and soft clauses; the goal is to find an assignment satisfying *all* hard clauses while minimising the number of falsified soft clauses. This variant is useful for modelling problems, because it allows us to translate constraints as hard clauses, while encoding the objective function using the soft clauses. As such, we make use of the partial weighted MaxSAT variant in this thesis.

2.0.7. MaxSAT solvers

Most MaxSAT solvers are developed for general purposes and aim to optimise performance on a large set of benchmarks from a variety of domains. According to Cai and Zhang, 2020, the most popular and effective approach for MaxSAT is the SAT-based approach, which transforms the optimisation problem into a sequence of SAT decision problems.
SAT-based approaches for MaxSAT can be divided into two categories: iterative search and core-guided search. Iterative search uses a SAT-solver as a subroutine, and iteratively attempts to find an assignment satisfying at least k soft clauses. Depending on the direction of this linear search, k is either increased or decreased in every iteration. In the case of a bottom-up linear search, optimality is reached at $k = k' - 1$ whenever the the formula becomes unsatisfiable at $k = k'$. In the top-down approach, optimality is reached when the formula with $k = k' + 1$ clauses is UNSAT, but *is* SAT with $k = k'$.

2.1. Problem statement

Given a graph $G = (V, E)$, we are looking for a function $f : V \longrightarrow \mathcal{N}$ that partitions V in a way that minimises the number of mistakes. Recall that a partition over a set of objects, in this case V, assigns exactly one number to each $v \in V$. In predicate logic, this is defined as follows:

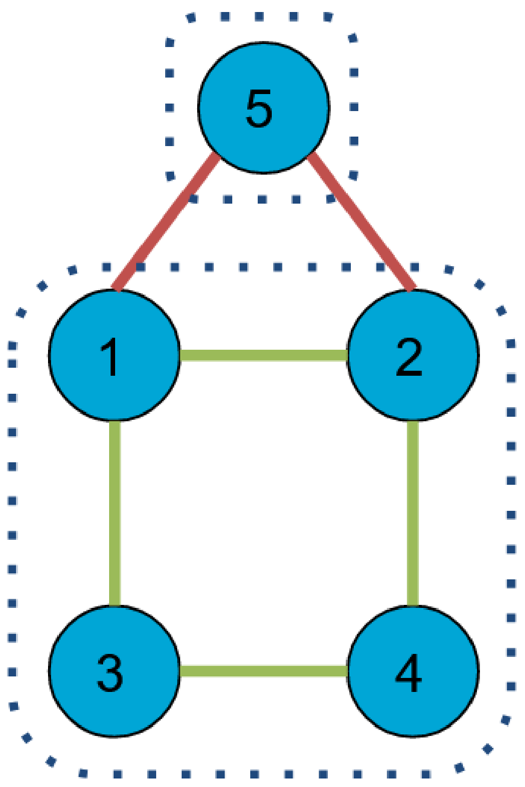
$$f \text{ is a partition of V } \Leftrightarrow \forall v \in V, \exists n. f(v) = n \tag{2.6}$$

The number of mistakes, or: the cost of f is defined as follows:

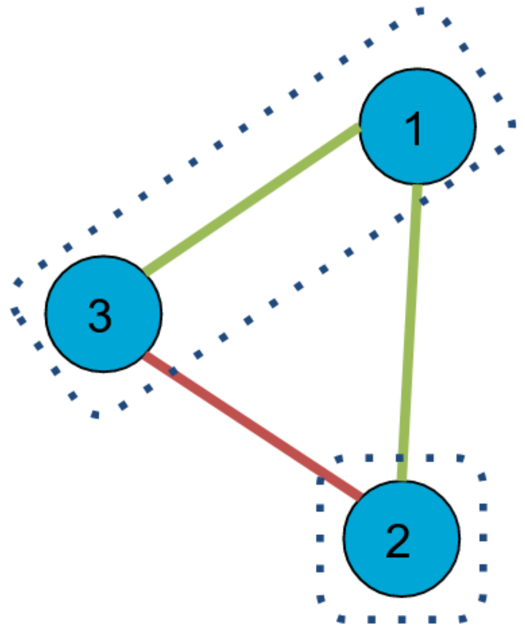
$$c(f) = \sum_{i,j \in V s.t. W_{i,j} < 0} \mathbb{1}[c(i) = c(j)] + \sum_{i,j \in V s.t. W_{i,j} > 0} \mathbb{1}[c(i) \neq c(j)] \tag{2.7}$$

Here, the indicator function $\mathbb{1}[.]$ is used. It is a function that returns 1 if its argument evaluates to true, and 0 otherwise.
Equation 2.7 is composed of two parts. The left hand term penalises putting two dissimilar points in a cluster, whereas the right hand term penalises putting similar points in different clusters. Figure 2.1 shows these two types of mistakes. Figure 2.2 shows four different clusterings of some arbitrary graphs; the reader is encouraged to verify that the number of mistakes is the same as the number written in the caption.

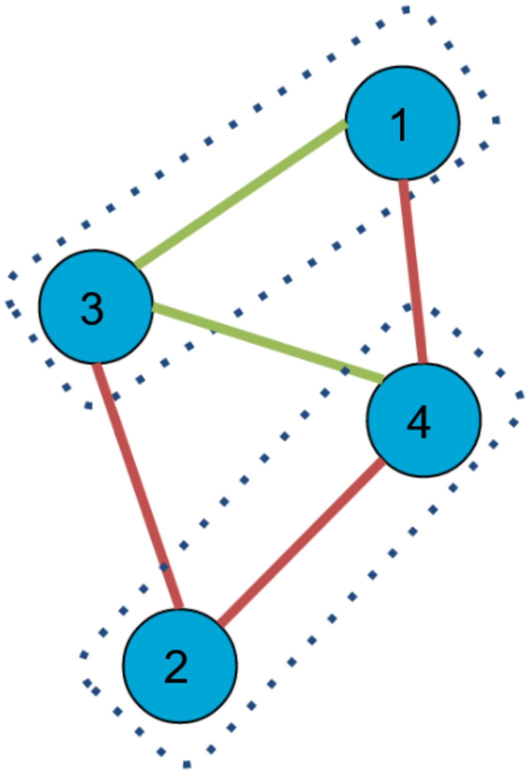
From data to similarity matrix Correlation clustering requires a qualitative notion of similarity between any two points. An N-by-N matrix with entries $W_{i,j} \in \{0, +1, -1\}$ is often used for this. One can assume that this matrix is given, without giving much thought from whence it came. However, this work features an empirical analysis of real-world *datasets*, and similarity matrices don't appear from thin air: one has to construct them. A *dataset* is a collection of datapoints, where each datapoint is a vector in $\mathcal{R}^d$. d is the dimensionality of the datapoint. Section 6 describes the procedure for converting a dataset into a similarity matrix.



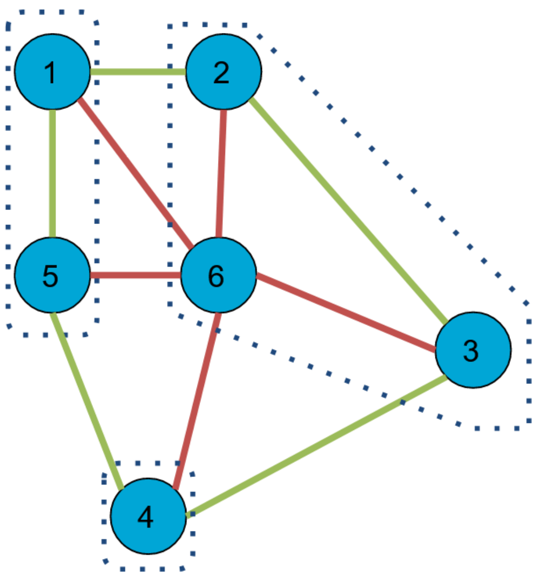
(a) A clustering with zero mistakes. All green edges are contained within a cluster, and every red edge crosses a cluster boundary.



(b) A clustering with one mistake: the green edge between node 1 and 2.



(c) A clustering with two mistakes: the red edge between node 2 and 4, and the green edge between node 3 and 4.



(d) A clustering with five mistakes: edge $\{1, 2\}$, $\{2, 6\}$, $\{3, 6\}$, $\{3, 4\}$ and $\{4, 5\}$.

Figure 2.2: Different examples of clustering different graphs. Blue circles indicate nodes, red lines indicate a negative edge and green lines indicate a positive edge. The number of mistakes is written in the subcaption, along with the edges that are responsible for the mistake.

3 Related work

In this section, we treat recent scientific works that are relevant to our research questions.

3.1. SAT

The Satisfiability problem, one of the most fundamental problems in computer science, is introduced in chapter 2 as a theoretical problem. However, SAT-related techniques have gotten increasing interest due to their applicability in developing generic approaches to solve common data analysis problems. Here, we describe relevant scientific work that has harnessed the power of SAT solvers.

Work using SAT Hebrard and Katsirelos, 2020 proposes a method to solve the graph colouring problem using (Max)SAT techniques combined with CDCL. Among others, Hebrard and Katsirelos, 2020 introduce a datastructure which is responsible for propagating transitivity constraints. This idea is used as a foundation for our lazy encoding of correlation clustering into MaxSAT, which is further explained in chapter 4.

An older but relevant work is Goldberg et al., 2001, who use SAT for combinational equivalence checking, a key problem in verification methods for digital systems. In short, the problem addresses the question whether two logic circuits are logically equivalent. This question can be answered by formulating it as a SAT formula; let f_1 and f_2 be the logical functions implemented by the two circuits. We then want to know whether there exists an input to these functions such that $f_1(input) \neq f_2(input)$. This is the same as computing whether $\neg(f_1 \Leftrightarrow f_2)$ is SAT. In Goldberg et al., 2001's work, a speedup of two orders of magnitude is achieved by using a SAT-based approach.

Propagation rules A minor but very relevant contribution by Hebrard and Katsirelos, 2020 is the transitivity propagation rules that they propose. Their work aims at solving the graph colouring problem using a hybrid CP/SAT method. Since the relation 'x has the same colour as y' and 'x is co-clustered with y' are both equivalence relations, they are bound by transitivity constraints. Hebrard and Katsirelos, 2020 explain how to propagate transitivity whenever a variable is assigned a value. Their propagation rules are implemented by maintaining a graph H that is updated as the partial assignment of values to variables grows. H contains a mapping $b : V \longrightarrow \mathcal{N}$ which assigns an index to every node. Whenever a literal becomes true (call this literal p), the propagator is invoked. When $p = x_{u,v}$, which corresponds to the nodes u and v getting the same colour, then for all $v' \in b(v)$ and $u' \in b(u)$, $x_{u',v'}$ becomes true. Similarly, when $p = \neg x_{u,v}$, for all $u' \in b(u), v' \in b(v)$, $x_{u',v'}$ becomes false. We use these propagation rules in our lazy encoding in chapter 4.

In addition to inferring *which* literals can be inferred, Hebrard and Katsirelos, 2020 also show what the explanations for these inferences are. It is important to distinguish between a reason and an explanation: the reason for a propagations is the entire clause which caused the propagation to trigger, whereas an explanation is the conjunction of literals that implies the inferred literal. The difference between reasons and explanation is minor. Hebrard and Katsirelos, 2020 use the transitivity clauses as the reasons for inferences; since it is a rather straightforward process, we do not repeat it here.

3.1.1. Solvers

A surprising property of the SAT problem is that it is intractable, yet very large formulas with millions of variables (taken from industrial benchmarks) have been solved using SAT solvers. Buss and Nordström, 2021 provide insight into this remarkable result by analyzing SAT using proof complexity. A talk by Ganesh and Vard, n.d. attempts at providing answers through a more empirial approach, but the unreasonable effectiveness of SAT solvers remains an open question to this day. Here, we describe some of the scientific work that has led to modern SAT solvers.

Davis et al., 1962 proposed an algorithm for solving SAT, and it has seen widespread use since its inception. Another important step in SAT solvers was the work of J. M. Silva and Sakallah, 1997 who proposed GRASP, a search algorithm for satisfiability that used Conflict Driven Clause Learning (CDCL), which catapulted the efficiency of SAT solvers.

CDCL CDCL is an important component of modern SAT solvers. During search, the solver will make assumptions about the solution ('decisions') in an attempt to find a satisfying assignment. These assumptions frequently lead to cases where the assumptions will cause the formula to become unsatisfiable, and the solver has to undo some assumptions. These cases are called conflicts, and CDCL learns from these by extracting the 'root cause' for the conflict. It then forces the solver to not make this mistake again by disallowing the specific circumstances that led to the conflict. In chapter 2, CDCL is explained in more detail.

3.2. Correlation clustering

Encodings

Here we describe Berg and Järvisalo, 2017's encodings in more detail. These are relevant since they form the basis for the lazy encoding and the sparse encoding in chapter 4.

Transitive The transitive encoding proposed by Berg and Järvisalo, 2017 uses variables $x_{u,v} \forall\, u, v \in V$ with the semantics that $x_{u,v}$ is true if and only if u and v are co-clustered. The objective function is given by the following expression:

$$\max \sum_{W_{u,v} > 0} x_{u,v} + \sum_{W_{u,v} < 0} (1 - x_{u,v}) \tag{3.1}$$

Finally, the only constraint is the transitivity constraint. This constraint forces the solution to be a legal clustering. It is defined for every triple u, v, w of vertices:

$$\text{Transitivity} : \forall \text{ distinct } u, v, w \in V \qquad (\neg x_{u,v} \lor \neg x_{v,w} \lor x_{u,w}) \tag{3.2}$$

This constraint may seem foreign when it is presented in conjunctive normal form; rewriting it as an implication makes it easier to identify the semantics:

$$\text{Transitivity} : \forall \text{ distinct } u, v, w \in V \qquad x_{u,v} \land x_{v,w} \longrightarrow x_{u,w} \tag{3.3}$$

Two notable pieces of scientific work that improve upon the transitive encoding are by Miyauchi and Sukegawa, 2015 and subsequently Miyauchi et al., 2018, who propose improvements to the ILP formulation. The first paper identifies a subset of transitivity constraints that is redundant and by doing so, they reduce the prohibitive number of constraints required to represent the problem. Specifically, they show that the number of constraints is of order $\mathcal{O}(nm_{\geq 0})$, where n is the number of nodes and $m_{\geq 0}$ is the number of edges with non-negative weight. Miyauchi et al., 2018 then take advantage of their previous work by perturbing all zero-weighted edges in the graph in such a way that the optimal solution remains the same. By perturbing zero weight edges, they reduce $m_{\geq 0}$, which in turn reduces the number of transitivity constraints.

Unary The unary encoding proposed by Berg and Järvisalo, 2017 uses variables $y_i^k, \quad i \in V, 1 \leq k \leq K$. The interpretation is that y_i^k is true if and only if node i is assigned to cluster k. Berg and Järvisalo, 2017 define several derived variables in order to define the objective function. These details are omitted and we only give the constraints for this encoding: the ExactlyOne constraint. This constraint ensures that every node is assigned to exactly one cluster, and it is given by the following expression:

$$\text{ExactlyOne} : \forall i \in V \quad \sum_{1 \le k \le K} y_i^k = 1 \tag{3.4}$$

The constraint is not yet in clausal form. Berg and Järvisalo, 2017 omit the details for converting this expression into clausal form, but it can be done using two constraints: MoreThanZero and LessThanTwo. Together, they enforce ExactlyOne.

$$\text{MoreThanZero} : \forall i \in V \quad \bigvee_{1 \le k \le K} y_i^k \tag{3.5}$$

The LessThanTwo constraint is given by Sinz, 2005.

Binary The binary encoding proposed by Berg and Järvisalo, 2017 uses boolean variables to represent the cluster index of a vertex as a binary number. As such, the variables are $y_i^k, i \in V, 1 \le k \le a$ and y_i^k represents the k^{th} bit of the binary representation of i's cluster index. There are several derived boolean variables which are used to represent the objective function, but these are omitted here.

Bounds and approximations

Bansal et al., 2004 shows that there is a trivial 0.5-approximation by either putting all nodes in one cluster, or each node in a separate cluster. They also propose an $\mathcal{O}(\log n)$ approximation algorithm which uses linear programming combined with region-growing techniques. Bansal et al., 2004 proposes a Polynomial Time Approximation Scheme (PTAS) as well, making it possible to get close to the optimal solution at the cost of increased runtime.

Bansal et al., 2004 also shows that a certain substructure in a similarity graph can give a lower bound on the number of mistakes made by the optimal solution. This bound is static, meaning it can be computed before commencing the solving process. In chapter 5, we propose a method that goes beyond this idea by identifying two additional substructures with the same property.

The work by Hebrard and Katsirelos, 2020 is concerned with graph colouring, a different graph problem where the objective is to assign partition a graph into 'colours' such that no two adjacent vertices have the same colour. They compute dynamic bounds for the problem in two different ways. The first one is a greedy algorithm, which finds a 'good ' clique in the graph, which is used to give a lower bound on the minimum number of colours required. The other method is similar to the one proposed by Bansal et al., 2004, since it identifies a substructure in the graph which gives a lower bound on the number of colours. The substructure here is a Mycielskian subgraph, which was proposed by Mycielski, 1955.

4 Encoding

The encodings described here are a hybrid between a constraint program and a MaxSAT problem. In a regular MaxSAT problem, clauses are responsible for propagating unit information, but we instead let a propagator fulfil the role of clauses instead.
The first encoding is an adaptation of Hebrard and Katsirelos, 2020's work on graph colouring. The second encoding is inspired by Hebrard and Katsirelos, 2020, who state that "the main drawback of this model [the transitive encoding] is that we need a large number of variables. This is especially problematic for large sparse graphs where the number of non-edges is quadratic in the number of vertices and significantly larger than the number of edges".

4.1. Lazy encoding

Motivation The need for a lazy encoding stems from the fact that a purely clausal encoding requires $\mathcal{O}(|V|^3)$ clauses. Specifically, the number of transitivity clauses required is given by $(|V|^3 - 3|V|^2 + 2|V|)/2$ [1]. Transitivity clauses contain exactly three literals. If we assume that a literal requires 4 bytes to store (one 32-bit integer), then a graph with 500 nodes already requires 750 MB just to store the transitivity clauses. The lazy encoding only invokes these clauses on demand, meaning the clauses do not have to be generated a priori. Instead, we store the edge list of the graph which requires $\mathcal{O}(|E|)$ space, which is almost negligible compared to $\mathcal{O}(V^3)$.

4.1.1. Propagator interface

The representation of the graph structure fully replaces the transitivity clauses and this means that the responsibilities of these clauses must be fulfilled by the lazy encoding. Recall from chapter 2 that the *job* of a clause is to propagate unit information. As soon as all but one of the literals of a clause are false, we know that the last, unassigned literal must be set to true. If not, the clause would evaluate to false, making the solution unsatisfiable. The rules for deciding which literals become true as a result of the propagated literal are already given by Hebrard and Katsirelos, 2020 in chapter 3 so we do not repeat them here. Instead, we describe the propagator interface that is used in the solver's code. In the next subsection, we give implementation details for the underlying graph structure, which is integral to implementing the lazy encoding.

Propagator interface The propagator interface defines the responsibilities of a propagator. This interface is part of a MaxSAT solver by one of the authors of this thesis (Emir Demirović); it is not yet published.

- PropagateLiteral($x_{u,v}$): accepts a literal and returns false iff assigning this literal to true would lead to a conflict. Modifies the internal state of the propagator in order to reflect the fact that $x_{u,v}$ has been assigned true by the solver.

[1] This expression was determined empirically by fitting a third order polynomial through the number of transitivity clauses for different instances.

- Synchronise: synchronises the propagator's internal state with the solver's partial assignment. This function is usually called after a conflict has occurred.
- ExplainLiteralPropagation($x_{u,v}$): this function is only called if the propagator has propagated $x_{u,v}$ in the past. It returns a conjunction of literals which have caused $x_{u,v}$ to become true. That is, if $\bigwedge_{x \in X} x \longrightarrow x_{u,v}$, then this function returns X.
- ExplainFailure: returns a conjunction of literals which together cause a conflict. That is, if $\bigwedge_{x \in X} \longrightarrow \bot$, then ExplainFailure returns X.

4.1.2. Implementation

The implementation of the internal data structure that keeps track of the clusters under the current partial assignment closely resembles a union-find data structure. However, since the solver frequently backtracks throughout the solving process, it is required to have an undo mechanism in place.

Graph functionality The propagator uses a graph representation with the following interface methods:

1. Contract(u, v): merges the clusters of node u and v into a new cluster.
2. UndoContraction: undoes the most recent contraction, and returns the two vertices that caused the contraction.
3. AreVerticesCoClustered(u, v): returns true iff u and v are co-clustered.
4. Separate(u, v): separates the clusters of node u and v.
5. UndoSeparation: undoes the most recent separation.
6. AreVerticesSeparated(u, v): returns true iff u and v are separated.
7. Find(u): returns the cluster index of node u. Note that initially, every node is assigned to a separate cluster.
8. FindNeighbours(u): returns all nodes that are in the same cluster as u.

Whenever the top-level propagator gets invoked to propagate a literal $x_{u,v}$, the state of the graph should change in order to reflect the fact that $x_{u,v}$ has been assigned true. Depending on the polarity of the incoming literal, either $Contract(u, v)$ (for a positive literal) or $Separate(u, v)$ (for a negative literal) is called. These methods ensure that the graph's state is synchronised with the solver's partial assignment. Similarly, when the solver decides to backtrack, the state of the graph is kept synchronised by invoking $UndoContraction$ and $UndoSeparation$; the details for this are explained in this section as well.

Contraction Figure 4.1 displays how the internal representation changes after Contract is called. The middle image shows how the data structure changes after a contraction of two nodes. It can be seen that a constant number of operations has to be performed in order to execute the contraction; first, a new cluster object is created, which points to itself. Next, the clusters of node 1 and 2 are pointing to this new cluster. Finally, the new cluster has references to its mergees. In conclusion, Contract is an $\mathcal{O}(1)$ operation.

Whenever a contraction between u and v occurs, the graph records this transaction as a tuple (u, v) and pushes it on an undo stack $S_{contraction}$. This allows us to undo the merging in a later stage, when the solver backtracks.

Separation We track which pairs of nodes are separated by using an associative container $: V \longrightarrow \mathcal{P}(V)$ where the key is a node's index v and the value is a set containing all nodes u for which u and v are separated. The implementation for this is simply a C++ `unordered_map<int, unordered_set<int> >`, and therefore we do not include pseudocode that describes it.

Whenever a separation between u and v occurs, the graph records the transaction by pushing a tuple (u, v) on an undo stack $S_{separation}$. Similar to the contraction undo stack, this allows us to undo the required separations whenever the solver backtracks.

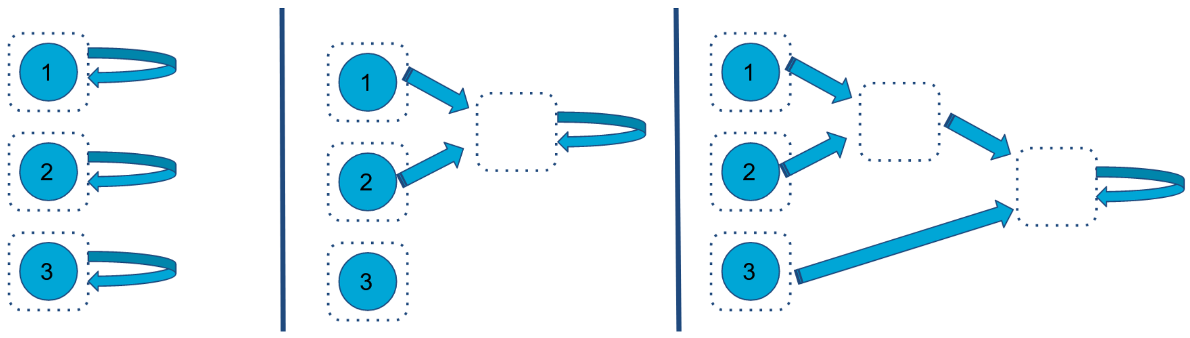


Figure 4.1: Internal representation of the graph. Dashed blue boxes indicate a cluster. In the implementation, a cluster either contains a vertex, or is empty and simply points to another cluster. Left figure: a graph without any contractions. All the nodes have their own cluster, which points to itself. Middle: the graph after Contract(1, 2) has been called. A new cluster is allocated, and the mergees' clusters are pointing towards it. The new cluster points towards itself, meaning the cluster is a terminating one. Right: the graph after Contract(1,2) and Contract(1,3) have been called.

Find The interface method Find is implemented by traveling through the chain of contractions in the graph. The Cluster datastructure used has two types of chain links: the *next* link, and two *prev* links. By convention, if a cluster has a *next* link that points to itself, then that cluster is 'terminal'. Algorithm 2 shows the pseudo implementation of Find.

Algorithm 2: Finding the cluster index of u.

```
Result: The cluster index of node u
curr = InitialClusterOf[u];
next = curr;
while curr.next != curr do
    curr = next;
    next = curr.next;
end
yield next
```

FindNeighbours Findneighbours(u) finds the 'cluster mates' of u. That is, it returns the set $\{v \in V \mid c(v) = c(u)\}$. The implementation uses a recursive relation which travels backwards through the contraction history of u's cluster.

$$Contents(B) = \begin{cases} B.content, & \text{if B.prev = null} \\ Contents(B.prev.left) \cup Contents(B.prev.right), & \text{otherwise.} \end{cases}$$

Equation 4.1.2 shows the recursion for finding the contents of a cluster B. FindNeighbours(u) is implemented by invoking Contents on Find(u). Figure 4.2 shows the recursion trace when calling Find-Neighbours(3) on the graph.

Side note: optimisation Hebrard and Katsirelos, 2020 remark that their propagator can safely do nothing when the incoming literal $x_{u,v}$ has two endpoints for which $b(u) = b(v)$. They call this a small but important optimisation. However, during implementation, we have found it *necessary* to do nothing when the endpoints are already in the same cluster. Failure to do so would sometimes lead to a cycle in the implication graph, causing an infinite loop during processing of learnt clauses.

4.1.3. Synchronisation

The challenge of keeping the internal graph structure synchronised with the solver's partial assignment lies in the backtracking component of the MaxSAT solver. As soon as the solver orders the propagator to synchronise, it looks at the history of assignments that were done by the solver (the trail). It searches for two edge literals: a positive literal $x_{u,v}$ and a negative literal $\neg x_{g,h}$. It chooses those literals that

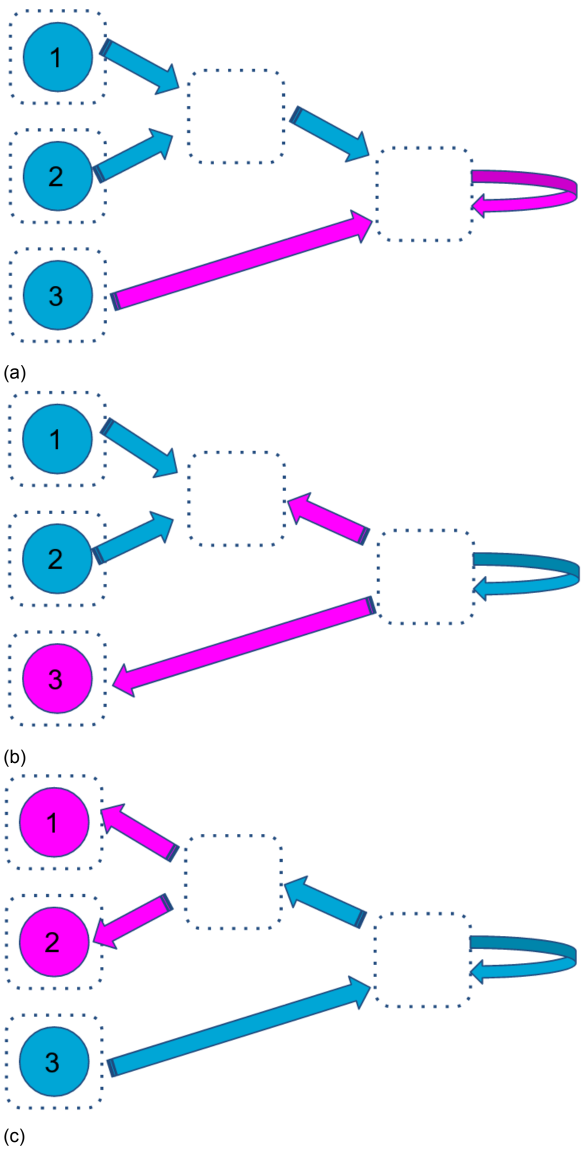


Figure 4.2: An example of what happens internally when calling FindNeighbours(3). (a) The algorithm first finds the cluster of node 3. (b) Contents is called on node 3's cluster. The bottom link finds a base case, whereas the top link recurses on a cluster. (c) Contents is called on a cluster, which travels backward and finds nodes 1 and 2.

have been unassigned most recently. These are called the 'undo destinations'. Then, it instructs the underlying graph to perform two tasks:

1. Call UndoContraction() until this function returns (u, v), called the contraction destination. When we've hit this literal, it means the graph's contraction structure is synchronised in accordance with the positive literals that are still assigned on the trail.

2. Call UndoSeparation() until this function returns (g, h), called the separation destination. Similarly, when we've hit this literal, it means the graph's separation structure is synchronised in accordance with the negative literals that are still assigned on the trail.

Figure 4.3 depicts an overview of how these undo destinations are selected by the propagator.

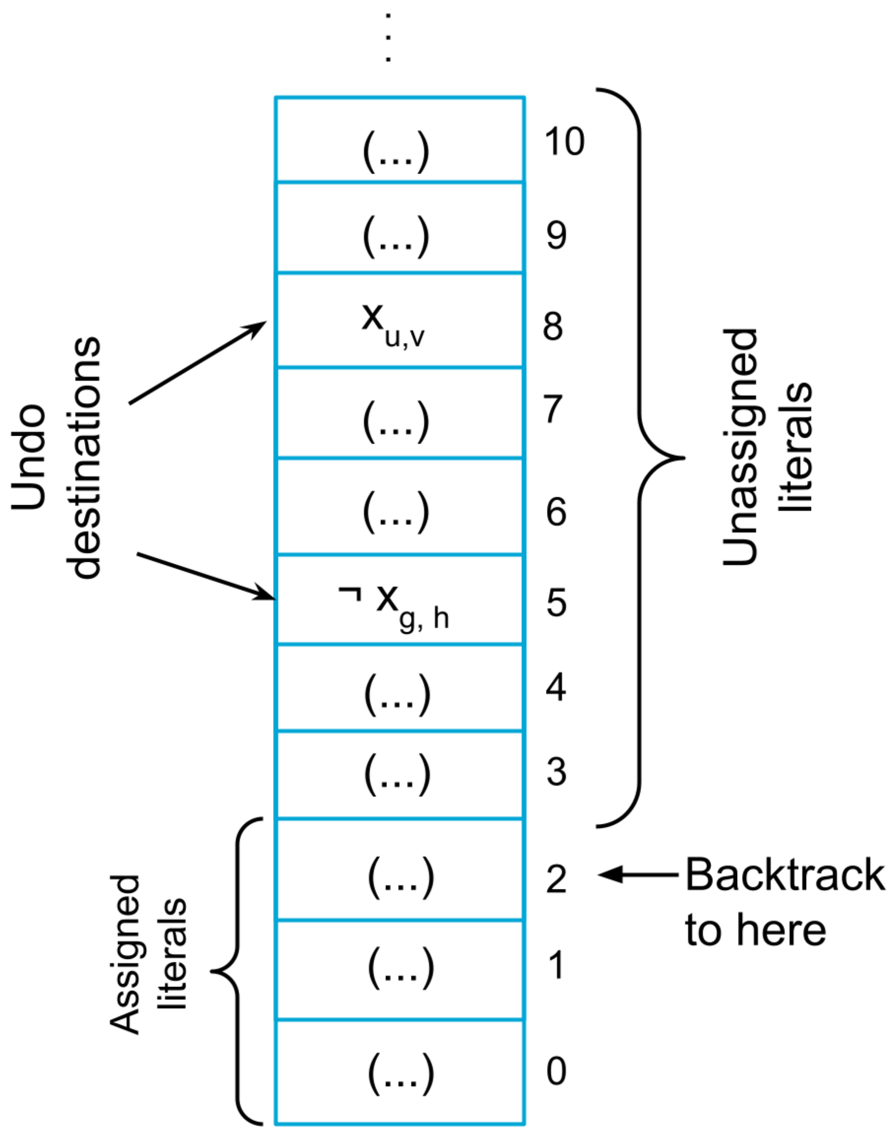


Figure 4.3: Depiction of the solver's history of literal assignments, which is also referred to as the trail. Blue boxes represent literal assignments, numbers represent position ordinals of the trail. We see that the solver will backtrack to position 2, unassigning any literal with ordinal ≥ 3. We see that $x_{u,v}$ and $\neg x_{g,h}$ are the literals that are closest to the backtrack position and are both edge literals. These become the undo destinations for the graph.

4.2. Sparse encoding

The transitive encoding proposed by Berg and Järvisalo, 2017 uses a boolean variable for every pair of vertices. We repeat the set of variables that they use here:

$$x_{i,j} \forall\ 1 \leq i < j \leq N \tag{4.1}$$

We argue that this set of variables is redundant since it is possible to correctly solve the problem using fewer variables. Namely, the set of variables that is used in the sparse encoding is the following:

$$x_{i,j} \forall\ 1 \leq i \leq j \leq N \text{ s.t. } \{v_i, v_j\} \in E \tag{4.2}$$

Modifications The encoding requires several modifications in order to work correctly. The primary modification is concerned with the way clausal explanations are generated for literal propagations. The solver must make sure the explanations contain only literals $x_{u,v}$ for which there exists an edge $\{u, v\}$ in the graph. In the following subsection, we describe the procedure in more detail.

4.2.1. Explanations

Recall from chapter 2 that in a SAT solver, a propagator must provide an explanation for anything it infers as a result of a propagation. An explanation for an inferred literal y consists of a set of literals X such that $\bigwedge_{x \in X} x \longrightarrow y$.

Example The reason why the 'normal' way of creating explanations is not sufficient for the sparse encoding is best explained using an example accompanied by a figure. Consider a cycle graph of 5 nodes. Now suppose nodes 1, 2, 3 and 4 are co-clustered, and now the solver now decides to co-cluster node 4 and node 5. We look at what happens next:

1. The solver sets the variable $x_{4,5}$ to true.
2. The solver asks the propagator to process this literal assignment, enqueue any new inferences and return true if propagation was a success (i.e. no conflicts were detected).
3. The propagator processes $x_{4,5}$ and infers one new literal assignment: $x_{1,5}$ will become true. The reader can verify that this is the case because nodes 1 and 4 are co-clustered, as well as nodes 1 and 5; as a result, nodes 4 and 5 must be co-clustered as well, and the inference of $x_{4,5}$ is thus valid.
4. The propagator will attempt to construct an explanation for the inference of $x_{4,5}$, but it fails: the variable $x_{1,4}$ is not defined because the edge $\{1, 4\}$ does not exist, and therefore $x_{1,4}$ does not exist either.

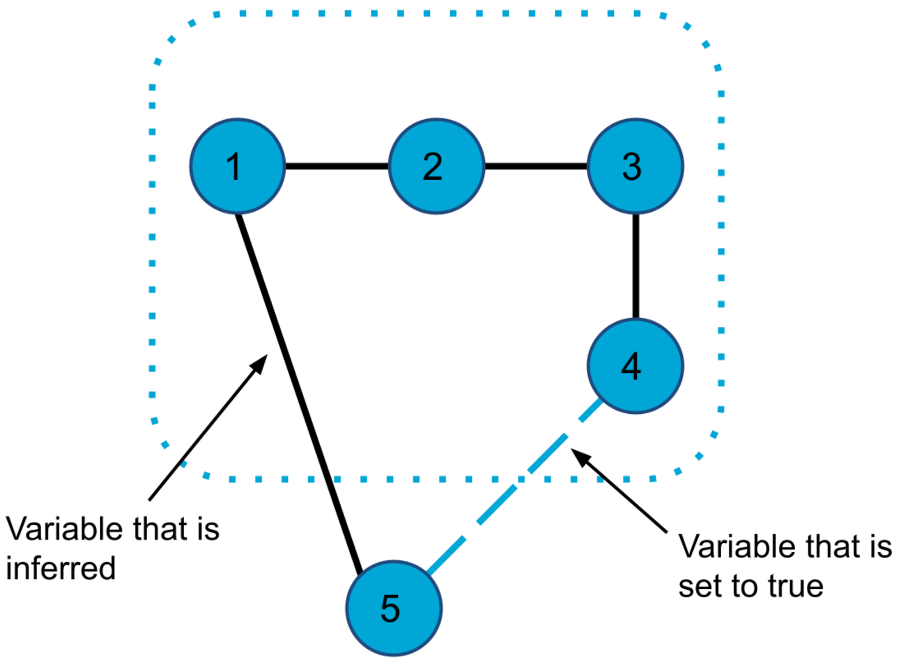


Figure 4.4: A cycle graph with a partial assignment: nodes 1, 2, 3 and 4 are co-clustered, which is indicated by the dashed box around these nodes. Lines between nodes indicate edges; the absence of lines means there is no edge. This example illustrates why a regular explanation scheme is not sufficient for the sparse encoding; the missing edges between vertices will cause issues when creating clausal explanations for inferences.

Creating the correct explanation A correct explanation for the inference of $x_{1,5}$ has two components:

1. An explanation for why nodes 1 and 4 are in the same cluster
2. The literal that triggered the inference, call it the pivot literal. In this example, the pivot literal is $x_{1,4}$.

It is clear that explaining why nodes 1 and 4 are co-clustered involves finding a path of edges going from node 1 to node 4, using only nodes that are in the same cluster. We can see that this is the case by repeatedly applying the transitivity property. Let xCy be the relation "x is co-clustered with y". Equation 4.3 shows this by repeatedly applying the transitivity property.

$$\frac{\begin{array}{r} 1C2 \wedge 2C3 \longrightarrow 1C3 \\ 1C3 \wedge 3C4 \longrightarrow 1C4 \end{array}}{1C2 \wedge 2C3 \wedge 3C4 \longrightarrow 1C4} \tag{4.3}$$

A natural way of proceeding would be to devise a path-finding algorithm that is only allowed to use edges within a cluster. Initially, we had implemented this before realising that it is very expensive to do so, since every inferred literal requires running a path finding procedure. As a result, we have opted for an approach that is theoretically sub-optimal, but which hugely outperforms the initial approach in practice.

Example: spanning tree We can create a path through a cluster of nodes by leveraging the contraction history of that cluster. By recursively traveling through the cluster's history of mergees, we can construct a spanning tree over the entire cluster. Recall that a property of a spanning tree is that its edges form a path between every single pair of nodes in the tree. Here we show how to create such a spanning tree.
Figure 4.5 shows the contraction history of a graph of four nodes.

1. Initially, all nodes have their own cluster.
2. Then, nodes 1 and 3 are co-clustered ('contracted'). Nodes 2 and 4 are also contracted. There are now two clusters, containing nodes $\{1, 3\}$ and $\{2, 4\}$.
3. Finally, the two clusters are merged through the edge variable $x_{1,4}$. The final result is a cluster containing $\{1, 2, 3, 4\}$.

The spanning tree is obtained by collection all the pink edges from figure 4.5. This can be done in a way that is very similar to the FindNeighbours algorithm. Instead of collecting nodes, the algorithm travels back through the contraction history and collects 'pivot' edges: edges through which the cluster was merged with other clusters. The final result is guaranteed to be a spanning tree over the cluster! The conjunction of edge literals from the spanning tree are used as an explanation for the co-clusteredness of $u, v \in C$. Algorithm 3 shows the recursive procedure that is used to compute the spanning tree.

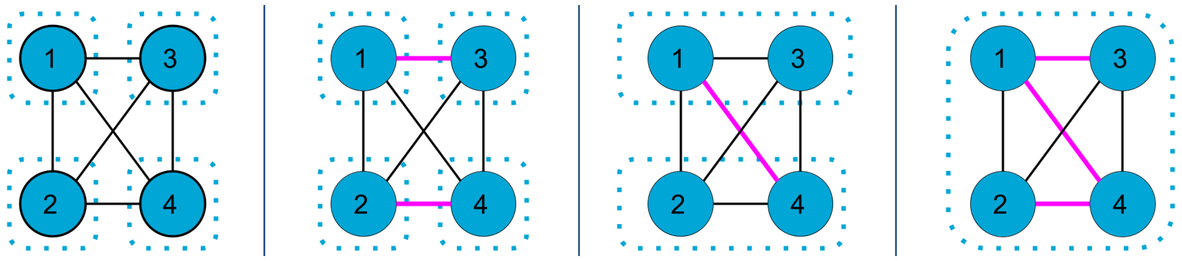


Figure 4.5: An example of the contraction history of a graph of four nodes. Edge colours are omitted for clarity, and because they are unimportant here. Initially, all nodes are in their own cluster, indicated by the dashed blue box around them. In the second panel, nodes 1 and 3 are merged, as well as 2 and 4. This is indicated by the bright pink line. In the third panel, we merge the two clusters of size two by co-clustering nodes 1 and 4. This causes their clusters to be merged. The fourth panel shows the final result: all nodes are in one cluster. We superimpose the spanning tree over the fourth panel (as pink lines); this is the set of edges that algorithm 3 would return.

Algorithm 3: MakeTree(C): a recursive procedure for generating a spanning tree over a cluster.

```
input  : A cluster $C$.
output: A collection of edges within $C$ that span $C$.
if $C$ has children then
    left ← $C.left$
    right ← $C.right$
    return $C$.pivot ∪ MakeTree(left) ∪ MakeTree(right)
else
    return ∅
end
```

5 Lower bounding

Similar to the ideas described in chapter 4, the methods we propose here are implemented through a propagator interface in the solver's code. Later on in this chapter, we show how bounds can be used in order to infer variables as a result of other variables becoming true.

5.1. Why bounds?

Finding a lower bound of a problem is a well-established way of increasing performance and reducing the amount of work required in order to solve a problem. A bound can either be static, meaning it can be computed at any time, independent of a partial solution. It can also be dynamic: a dynamic bounding algorithm will generate a bound *given* an incomplete / partial solution.

A bound can save time. If an algorithm is building up a solution, it can invoke a bounding algorithm which computes the *minimum* cost C that the solution is going to contain when it is completed. If the algorithm has already found a solution with cost $C' \leq C$, then the algorithm knows it can terminate its current branch in the search space: it won't find a better solution anyways. Consider the following fictional example:
A student is preparing for a final exam. They've already had a midterm exam, for which they've scored a disappointing 1 out of 10. In order to pass the course, they need an average grade of 6.0 out of 10. The student reasons that in order to pass the course, they would need to score 11 out of 10 points for the final exam. This is impossible, therefore it is impossible to pass the course this year. A clever student would consequently spend time on other things instead of studying for the final. This is a simple scenario where reasoning about bounds can save time.

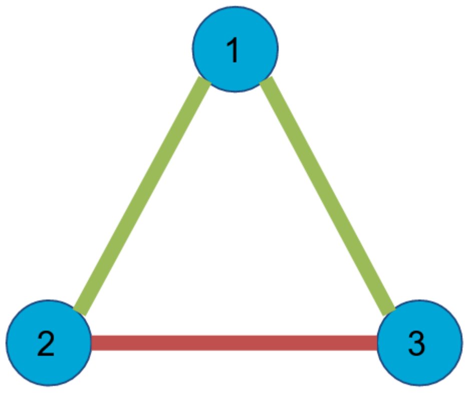
What makes a good bound? A bound can save time by pruning the search space, but it also costs time to compute. A good bound is therefore inexpensive to compute and is 'tight', meaning it closely approximates the true bound.

Hybrid solver Contrary to regular MaxSAT solvers, our algorithm is hybrid, meaning it uses techniques from MaxSAT while not strictly qualifying as a MaxSAT solver. The difference lies in the fact that our solver maintains a representation of the graph throughout the solving process whereas normally, the graph representation of the problem would get lost during the encoding step. This allows us to design lower bounding techniques that reason about the problem on a higher abstraction level.

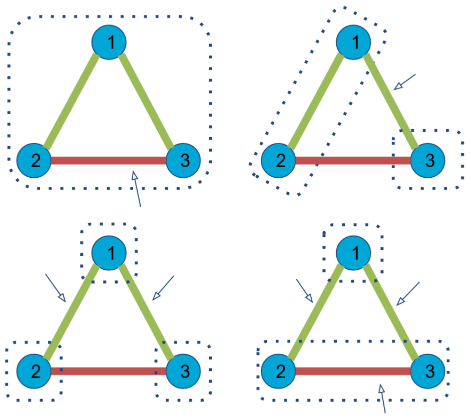
Explaining a bound In our algorithm, the lower bound has one purpose: whenever it sees that the current partial solution is *guaranteed* to be worse than our best known solution, it should raise a conflict. Recall from chapter 2 that (Max)SAT solvers require an explanation for a conflict, which is a conjunction that caused the conflict to happen. In the following sections, we explain how this clausal explanation is generated.

5.2. Triangle counting

Previous work The ideas presented here are inspired by Bansal et al., 2004's work. We briefly repeat their idea here, and then introduce our proposed extension.
Bansal et al., 2004 remark that a certain type of triangle in the similarity graph can predict a lower bound on the cost of any solution. Figure 5.1a shows this substructure. We challenge the reader to find a clustering of this graph that respects every single edge weight, i.e. find a clustering that does not make any mistakes. This is impossible: if we look at every possible clustering of this graph, there will always be at least one mistake. Therefore, Bansal et al., 2004 reason, we can say that a lower bound on the number of mistakes is 1.



(a) Bansal et al., 2004's substructure which allows us to reason about a lower bound of the optimal solution. Red edges indicate a negative edge, whereas green edges represent a positive edge.



(b) All possible clusterings of three points. Edges that incur a penalty are indicated by a small arrow. Note that every possible clustering of these three nodes results in a penalty of at least one.

Partial assignment Could there be more of these triangles; ones that will guarantee we make a mistake no matter how we cluster them? Unfortunately, there aren't. However, if we already have a partial solution where some pairs of vertices are already assigned a value (either co-clustered or not co-clustered), then there are exactly two more partially assigned triangles that admit a lower bound. We present them in the following sections.

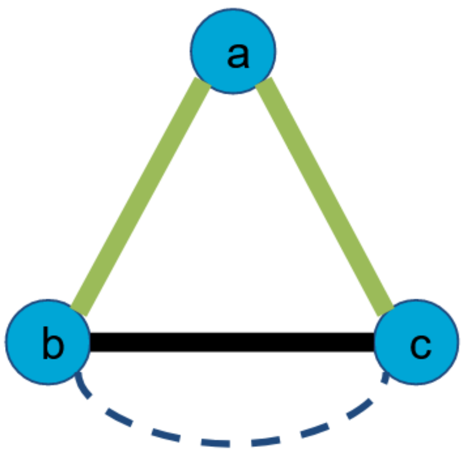
5.2.1. Lambda-triangles

The first type of triangle is called the Lambda triangle, because of its shape (it looks like a Λ). Figure 5.2a shows this triangle. The nodes b and c are separated under the current partial solution (i.e. the variable $x_{b,c}$ is assigned false). Note that the weight $w_{b,c}$ is irrelevant. The other two edges are not yet assigned, but they must have a positive weight. We leave it to the reader to verify that completing the clustering will lead to at least one mistake in every case. This property allows us to infer that when the similarity graph has a triangle as depicted in figure 5.2a, then the penalty will be at least one.

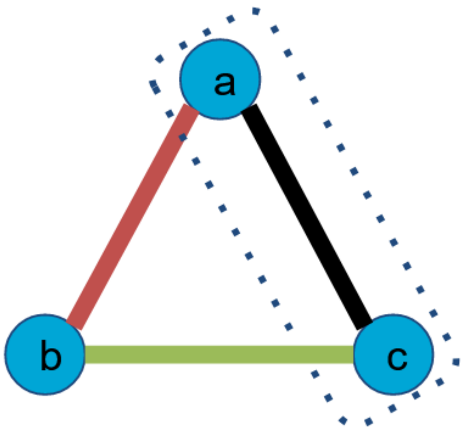
5.2.2. Delta-triangles

The second type of triangle is called the Delta triangle, once again because of its shape which looks like a Δ. This triangle has a positive and a negative edge, and a third edge whose endpoints are already co-clustered.

Mutual exclusivity There is a caveat associated with counting these types of triangles: you cannot use an edge in more than one triangle. This is best explained using an example. Figure 5.3 shows an example graph with a partial assignment: b and c are separated, as well as c and d. One could argue that this graph contains two Λ-triangles: triangle ABC and ACD. This would be incorrect, because these triangles share an edge: AC. As a result, the graph in figure 5.3 contains only one valid Λ-triangle. In general, we must demand that the Λ- and Δ-triangles have mutually exclusive edges. Section 5.2.3 gives a more detailed explanation of how to implement this.



(a) A Λ-triangle. The dashed black line between nodes b and c indicates b and c are not co-clustered under the current partial assignment. The black solid line between b and c indicates that the weight of the edge between b and c can be anything (0, +1, -1). Finally, it is a requirement that the variables $x_{a,c}$ and $x_{a,b}$ are not yet assigned a value.



(b) A Δ-triangle. The dashed box indicates that a and c are co-clustered under the current partial solution. The solid red and green lines between a, b and b, c respectively indicate the edge weight. The solid black line between a and c indicates that the weight between a and c can be anything. Finally, it is required that edge variables $x_{a,b}$ and $x_{b,c}$ are not yet assigned a value.

Figure 5.2: The two types of triangles that admit a lower bound. Colours indicate edge weight, dotted lines indicate co-clusterings and dashed lines indicate the fact that two points are not co-clustered.

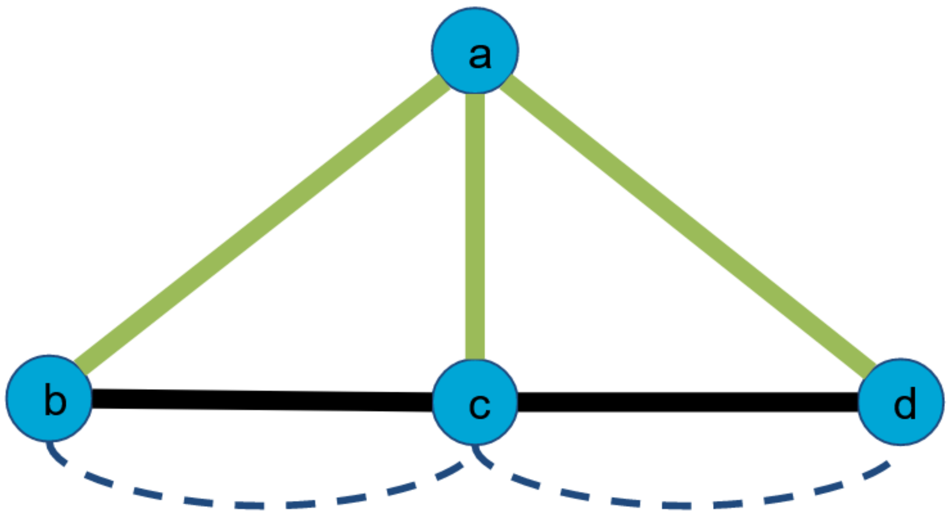


Figure 5.3: A case where we can identify two Λ-triangles, but only one counts. This is because the triangles ABC and ACD share an edge: AC.

5.2.3. Explanation

The purpose of these bounding techniques is to detect when we are entering a branch that is ‘destined to fail’. In other words, a branch whose partial solution indicates that the minimum penalty in this branch is going to be greater than the best known solution so far. In this case, we raise a conflict in the solver. Recall from chapter 2 that a conflict requires an explanation in the form of a conjunction of literals. In this section, we explain how to create such an explanation for a bound that is computed using Λ- and Δ-triangles.

Future and direct cost Suppose we know that the current partial assignment has cost C. C is composed of two components:

- Direct cost: this is the sum of mistakes made by the partial solution. It consists of all edge assignments that contradict the weight of the edge. Formally, this is the set of literals $\{x_{u,v} \in$

Assigned $|w_{u,v} < 0\} \cup \{\neg x_{u,v} \in$ Assigned $|w_{u,v} > 0\}$. The explanation for this type of cost is simple: the literal that violates the edge.

- Future cost: this is the number of triangles that are counted in the partially assigned graph. The explanation for these triangles is the edge of the triangle that is already assigned. In the Λ-triangle of figure 5.2a, this would be the edge BC. In the Δ-triangle of figure 5.2b, this would be the edge AC.

The full explanation for a bound computed using these triangles is the conjunction of the direct and the future cost:

$$\text{Explanation}(X) = \left(\bigwedge_{x_{u,v} \in \text{Future}} x_{u,v} \right) \wedge \left(\bigwedge_{x_{u,v} \in \text{Direct}} x_{u,v} \right) \tag{5.1}$$

For the interested reader The interested reader may wonder why the explanation for the Λ- and Δ-triangles is valid. Here we show why that is the case for the Λ-triangle; the proof for the Δ-triangle is very similar.
Recall that the transitivity constraint holds for every triple of nodes, and so also for a, b and c which make up the Λ-triangle.:

$$\neg x_{a,b} \vee x_{b,c} \vee \neg x_{a,c} \tag{5.2}$$

We rewrite this as an implication, with $\neg x_{b,c}$ on the left hand side:

$$\begin{aligned} \neg x_{b,c} &\longrightarrow \neg \left(x_{a,c} \wedge x_{a,b} \right) \\ \neg x_{b,c} &\longrightarrow \neg x_{a,c} \vee \neg x_{a,b} \end{aligned} \tag{5.3}$$

Thus, we see that $\neg x_{b,c}$ causes either $\neg x_{a,c}$ or $\neg x_{a,b}$. Recall that in a Λ-triangle, $w_{a,c} = w_{a,b} = 1$. Combining this with the definition of the objective function (see chapter 2 equation 2.7, or chapter 3 equation 3.1), we see that the edges will contribute at least 1 point of penalty to the cost.

Implementation details

The ideas presented here are implemented in the solver in the form of a propagator. First, we give the high-level overview of the propagator implementation. Then, we give the details of how to detect new Λ- and Δ-triangles. Finally, we show how to detect *invalidated* triangles.

Propagator implementation

The propagator interface requires us to consume a literal and return false if the literal causes a conflict, and true otherwise. Algorithm 4 shows how the bounds propagator consumes literals.

Incremental detection

Instead of looking at the partially assigned graph and counting the triangles in it, we would like to *incrementally* keep a tally of encountered triangles. Fortunately, this can be done in $O(d)$ time per propagated literal, where d is the maximum degree of any node in the graph. We give detection algorithms for both Λ- and Δ-triangles here.

Detecting Λ-triangles Whenever the propagator is asked to propagate a negative literal $\neg x_{u,v}$, the Λ-detection algorithm is invoked. If it finds any triangles, it 'locks' the edges participating in the triangle. This way, edges are not used twice in a triangle (which would give incorrect behaviour). Algorithm 5 shows the pseudocode for the detection algorithm. The most expensive operation is computing N_u^+ and N_v^+, which costs $O(d)$, where d is the maximum degree of the graph.

Detecting Δ-triangles Whenever the propagator is asked to propagate a positive literal $x_{u,v}$, the Δ-detection algorithm is invoked. Algorithm 6 shows the pseudocode for this algorithm. Note that it is longer than algorithm 5; this is due to the fact that Δ-triangles are not symmetric, so we have to check for both the original triangle as well as the mirror image.

Algorithm 4: Propagating a literal in the bounds propagator. This algorithm mutates the state; it increments the LB variable, adds and removes literals from the conflict explanation and (un)locks edges.

```
Data: Incoming literal $x_{u,v}$.
Similarity matrix $W$.
Cost of best solution so far $UB$
Result: False if the literal caused a conflict, true otherwise. Updates the $LB$ variable.
changeInBound ← 0;
directPenalty ← 0;
numLambda ← 0;
numDelta ← 0;
numInvalidatedLambda ← 0;
numInvalidatedDelta ← 0;
if $x_{u,v}$ is a positive literal and $W(u,v) < 0$ then
    directPenalty ← $|W(u,v)|$;
end
if $x_{u,v}$ is a negative literal and $W(u,v) > 0$ then
    directPenalty ← $W(u,v)$;
end
if $x_{u,v}$ is a positive literal then
    numLambda ← CountLambda($x_{u,v}$);
end
else
    numDelta ← CountDelta($x_{u,v}$);
end
numInvalidatedDelta ← CountInvalidatedDelta($x_{u,v}$);
numInvalidatedLambda ← CountInvalidatedLambda($x_{u,v}$);
LB ← LB + (numLambda + numDelta + directPenalty) ;
LB ← LB - (numInvalidatedLambda + numInvalidatedDelta);
if (numDelta > 0) or (numLambda > 0) or (directPenalty > 0) then
    Add $x_{u,v}$ to the conflict explanation
end
if $LB \geq UB$ then
    return False                                  ▷ This is a bounds conflict
end
else
    return True
end
```

Detecting invalidated triangles Another caveat of the triangle-counting scheme is that triangles can be 'invalidated' as a result of other literals becoming true. The reason for this is quite involved: a triangle (Δ or Λ) **predicts** a penalty in the future. If that penalty eventually happens as a result of a literal assignment, the prediction has come true, which means the prediction is no longer valid. Recall from previous sections that the bound that is computed by the propagator consists of *direct* and *future* cost; a triangle being invalidated comes down to cost *flowing* from the future component to the direct component.

We can reason about which literal assignments invalidate which triangles. Together with the locking scheme, we can easily determine whether a literal assignment invalidates any triangles. In figure 5.2a, $\neg x_{a,b}$ or $\neg x_{a,c}$ would invalidate the triangle. In figure 5.2b, $\neg x_{b,c}$ or $x_{a,b}$ would invalidate the triangle. Algorithms 7 and 8 show the pseudocode for detecting invalidated triangles.

5.2.4. Inferences through bounds

A bounds propagator consumes propagated literals and raises a conflict if it sees that the lower bound of the number of mistakes exceeds the number of mistakes of the best solution so far. However, a bounds propagator can also infer the value of literals if the lower bound is slightly *below* the upper

Algorithm 5: CountLambda: incrementally detecting new Λ-triangles. This algorithm has side effects: it mutates the $Locked$ and $Pairs$ variable.

```
Data: Incoming literal $\neg x_{u,v}$.
Similarity matrix $W$.
Set of edges $E$.
Set of vertices $V$.
Set of Locked edges $Locked$.
Result: The number of Λ-triangles that are created by $\neg x_{u,v}$.
$N_u^+ \leftarrow \{w \in V | \{w,u\} \in E \wedge W(w,u) > 0 \wedge \{w,u\} \notin Locked \wedge x_{w,u} \notin \text{Assigned}\}$;
$N_v^+ \leftarrow \{w \in V | \{w,v\} \in E \wedge W(w,v) > 0 \wedge \{w,v\} \notin Locked \wedge x_{w,v} \notin \text{Assigned}\}$;
$T \leftarrow N_u^+ \cap N_v^+$;
for $t \in T$ do
    Lock edge $\{u,t\}$;
    Lock edge $\{v,t\}$;
    Register $(\{u,t\},\{v,t\})$ as a dual pair in $Pairs$.
end
return |T|
```

bound. In this section, we explain how this is done.

Off-by-one Consider the scenario where the bounds propagator consumes a literal which causes the lower bound to get bumped to $LB = UB - 1$. This is a crucial value, because if another literal comes along and also increases the lower bound, we will have to raise a conflict. A clever propagator will conclude that there can be no more mistakes in the future: every literal that is associated with a weighted edge *must* agree with the weight on its edge. If not, we will certainly have a conflict. Let $z_{u,v}$ be an incoming literal which we are about to consume. The set of literals that we can infer whenever $LB = UB - 1$ is the following:

$$\text{Inferred}(z_{u,v}) = \{x_{i,j} \mid W(i,j) > 0 \wedge x_{i,j} \notin \text{Assigned}\} \bigcup \{\neg x_{i,j} \mid W(i,j) < 0 \wedge x_{i,j} \notin \text{Assigned}\} \quad (5.4)$$

Caveats A subtle problem with this approach of inferring literals is an interaction with the triangle counting methods discussed in previous sections. It is not true in general that, for an edge $\{i,j\}$ s.t. $W(i,j) > 0$, assigning $z_{i,j}$ to false will lead to an increase in the lower bound. It is possible that $z_{i,j}$ partakes in a Λ- or Δ-triangle and therefore, its penalty is already accounted for (it was predicted by the triangle). Therefore, equation 5.4 is not correct whenever triangle counting is enabled. The correct way of determining the set of inferred literals is given by the following expression. Note that $CountInvalidatedTriangles(z_{u,v})$ is implicitly the sum of invalidated lambda and delta triangles.

$$\begin{cases} \emptyset & CountInvalidatedTriangles(z_{u,v}) \neq 0 \\ \text{Inferred}(z_{u,v}) & CountInvalidatedTriangles(z_{u,v}) = 0 \end{cases}$$

Algorithm 6: CountDelta: incrementally detecting new Δ-triangles. This algorithm has side effects: it mutates the $Locked$ and $Pairs$ variable.

```
Data: Incoming literal $\neg x_{u,v}$.
Similarity matrix $W$.
Set of edges $E$.
Set of vertices $V$.
Set of locked edges $Locked$.
Set of dual edge pairs $Pairs$.
Set of assigned variables $Assigned$.
Result: The number of Δ-triangles that are created by $\neg x_{u,v}$.
$N_u^- \leftarrow \{w \in V | \{w,u\} \in E \wedge W(w,u) < 0 \wedge \{w,u\} \notin Locked \wedge x_{w,u} \notin$ Assigned$\}$;
$N_v^+ \leftarrow \{w \in V | \{w,v\} \in E \wedge W(w,v) > 0 \wedge \{w,v\} \notin Locked \wedge x_{w,v} \notin$ Assigned$\}$;
$T_1 \leftarrow N_u^- \cap N_v^+$;
for $t \in T_1$ do
    Lock edge $\{u,t\}$;
    Lock edge $\{v,t\}$;
    Register $(\{u,t\},\{v,t\})$ as a dual pair in $Pairs$.
end
$N_u^+ \leftarrow \{w \in V | \{w,u\} \in E \wedge W(w,u) > 0 \wedge \{w,u\} \notin Locked \wedge x_{w,u} \notin$ Assigned$\}$;
$N_v^- \leftarrow \{w \in V | \{w,v\} \in E \wedge W(w,v) < 0 \wedge \{w,v\} \notin Locked \wedge x_{w,v} \notin$ Assigned$\}$;
$T_2 \leftarrow N_u^+ \cap N_v^-$;
for $t \in T_2$ do
    Lock edge $\{u,t\}$;
    Lock edge $\{v,t\}$;
    Register $(\{u,t\},\{v,t\})$ as a dual pair in $Pairs$.
end
return $|T_1| + |T_2|$
```

Algorithm 7: CountInvalidatedDelta: an algorithm for detecting invalidated Δ-triangles.

```
Data: Incoming literal $x_{u,v}$.
Set of locked edges $Locked$.
Set of dual-locked pairs $Pairs$. Similarity matrix $W$.
Result: 1 if $\neg x_{u,v}$ invalidates a Λ-triangle, 0 otherwise. This algorithm has side effects: it
        mutates $Locked$ and $Pairs$
Remove $\{u,v\}$ from $Locked$;
if $W(u, v) > 0$ then
    if $x_{u,v} \in Locked$ then
        $\{u',v'\} \leftarrow Pairs[u,v]$;
        Remove $\{u,v\}$ and $\{u',v'\}$ from $Pairs$;
        return 1
    end
end
```

Algorithm 8: CountInvalidatedLambda: an algorithm for detecting invalidated Λ-triangles.

```
Data: Incoming literal $x_{u,v}$.
Set of locked edges $Locked$.
Set of dual-locked pairs $Pairs$. Similarity matrix $W$.
Result: 1 if $\neg x_{u,v}$ invalidates a $\Lambda$-triangle, 0 otherwise. This algorithm has side effects: it
        mutates $Locked$ and $Pairs$
if W(u, v) > 0 and $x_{u,v}$ is a negative literal then
    if $\{u, v\} \in Locked$ then
        $\{u', v'\} \leftarrow Pairs[u, v]$;
        Remove $\{u, v\}$ and $\{u', v'\}$ from $Pairs$;
        Remove $\{u, v\}$ and $\{u', v'\}$ from $Locked$;
        return 1
    end
end
if W(u, v) < 0 and $x_{u,v}$ is a positive literal then
    if $\{u, v\} \in Locked$ then
        $\{u', v'\} \leftarrow Pairs[u, v]$;
        Remove $\{u, v\}$ and $\{u', v'\}$ from $Pairs$;
        Remove $\{u, v\}$ and $\{u', v'\}$ from $Locked$;
        return 1
    end
end
return 0
```

6 Experimental evaluation

In this experimental evaluation, we do an empirical analysis on the effectiveness of the techniques proposed in chapters 4 and 5. We use real-world problem instances to evaluate the efficacy of our methods, and we compare them to the baseline encodings by which the ideas in this thesis are inspired.

6.1. Set up

In this section, we describe the data, conditions and parameters which we used to evaluate the performance of our proposed techniques.

6.1.1. Metrics used for evaluation

Time In the domain of exhaustive search, a natural metric for performance is the time required for the solver to find the optimal solution and prove its optimality. However, we are not interested in the physical quantity of time itself, because this would depend on the hardware platform used: a researcher with a faster or slower computer would find different results than the ones presented here. Instead, we use the notion of normalised time. The normalised time is a quantity $t_{norm} \in [0, 1]$. It is computed as follows.
Let $Solvers$ be the set of solvers that are compared, let $Instances$ be the set of benchmarks. Let $s \in Solvers$ and $i \in Instances$.

$$t_{norm}(s, i) = \frac{t(s, i)}{\max_{s' \in Solvers} t(s', instance)} \tag{6.1}$$

Decisions Another notion of performance is the number of decisions that is required to solve the problem to optimality. The reason for using this metric is motivated by the fact that MaxSAT solvers are pieces of highly optimised software. Small details in the implementation can make a lot of difference in the time required to solve instances. This work does not focus on writing high performance code; rather it makes a conceptual contribution. Therefore, we would like a metric that abstracts away from not only the hardware, but also the *software*. The number of decisions is the appropriate metric for this, since it is independent of the implementation in code.

Number of instances solved A commonly used metric that is used for evaluating the speed of solvers is the number of instances that it can solve given a certain time budget. Among others, Berg and Järvisalo, 2017 uses this type of metric. Formally, this metric can be expressed as a function of the timeout. Let $Solvers$ be the set of solvers that are compared, let $Instances$ be the set of benchmarks. Let $T : S \times I \longrightarrow \mathcal{R}$ be the function mapping a solver-instance pair to the time it takes for that solver mode to solve the instance to optimality.

$$n_{solved}(t, s) = |\{i \in Instances | T(s, i) < t\}| \tag{6.2}$$

6.1.2. Benchmarks

The benchmarks that are used are similar to the ones used by Berg and Järvisalo, 2017 and Miyauchi et al., 2018. All of the datasets were taken from the UCI Machine Learning Repository Dua and Graff, 2017. For the purpose of exploring extreme cases of sparse graphs, we have created several synthetic benchmarks as well. A brief description of the used datasets can be found in table 6.1. For a more detailed description of the instances used in the dataset 'Original', see table 6.2.
We have deliberately opted for using real-world datasets for several reasons:

1. The applicability of the work in the real-world is better evaluated using data from said real-world. Moreover, (Max)SAT solvers have shown excellent performance on real-world benchmarks (see Ganesh and Vard, n.d. for a treatise on why this is the case)

2. The datasets are publicly available, making it easier to verify our findings.

We do realise that twenty datasets is on the low side. This is due to the fact that we had to select datasets that were 'within the realm of solvability' for our solvers, meaning that the runtime was somewhere between one and ten thousand seconds. Moreover, each dataset required some form of manual processing due to heterogeneous data formats. Lastly, the environment in which we ran our experiments featured a 5GB storage limit. This, combined with the fact that the baseline encodings (Transitive, Binary, Unary) were rather large (even in their compressed form), it would have been unviable to use a much larger number of instances.

Name	#Instances	Description
Original	20	A collection of real-world instances used to evaluate the performance of our proposed algorithm.
Inflated	189	An inflated version of the Original dataset which is used to investigate the relation between graph properties and our algorithm's runtime. The dataset is created by taking instances from Original and randomly removing edges.
Trees	20	A collection of trees used to investigate the performance of our algorithm on very sparse graphs. Tree sizes range from 50 to 350 nodes

Table 6.1: A short description of the instances used in the experimental evaluation.

Creating similarity matrices from data Correlation clustering requires a similarity matrix, while real-world data usually appears in the form of datasets: a collection of datapoints, each consisting of a feature vector. Berg and Järvisalo, 2017 proposes a method to convert a dataset into a similarity matrix. They first compute the normalised Euclidean distance between every pair of points. They then use a linear inverse mapping to map distances to the $[-0.5, 0.5]$ interval, which is interpreted as the similarity between the two points. In this work, Berg and Järvisalo, 2017's approach is adopted.

6.1.3. Hardware

We ran the experiments described here on the Starexec service developed and maintained by Stump et al., 2014. The Starexec compute node has the following hardware specs:

1. Operating System: CentOS Linux release 7.7.1908 (Core)

2. C++ compiler: gcc-4.8.5-39.el7.x86_64

3. CPU: Intel(R) Xeon(R) CPU E5-2609 0 @ 2.40GHz

4. RAM: 264 GB, of which 128GB can be used by a single program.

5. Maximum timeout: 5 hours.

ID	num_vertices	num_edges	max_degree	edge density	Reference
amphibians	188	12271	181	0.698	Blachnik et al., 2019
ceramic	87	2453	79	0.656	He et al., 2016
coimbra	115	4003	107	0.611	Patrı́cio et al., 2018
colposcopy_0	97	2566	96	0.551	Fernandes et al., 2017
colposcopy_1	96	1730	66	0.380	Fernandes et al., 2017
divorce	169	9034	159	0.636	Yöntem et al., 2019
ecoli	336	48927	335	0.869	**Dua:2019**
fertility	100	4241	95	0.857	Gil et al., 2012
forestfire	243	25148	238	0.855	Cortez and Morais, 2007
gastro	699	195354	673	0.801	Mesejo et al., 2016
heartfailure	298	42087	295	0.951	Chicco and Jurman, 2020
iris	150	9682	145	0.866	Fisher, 1936
leaf	340	48700	338	0.845	P. F. Silva et al., 2013
lungcancer	30	192	26	0.441	Hong and Yang, 1991
machine	209	15738	208	0.724	Bergadano et al., 1990
parkinsons	194	14042	193	0.750	Little et al., 2007
seeds	194	16032	192	0.856	Charytanowicz et al., 2010
slumptest	102	4016	98	0.780	Yeh, 2007
stoneflakes	78	2031	76	0.676	Weber, 2009
wine	178	10900	171	0.692	Aeberhard et al., 1994

Table 6.2: A more detailed description of the instances used in the ‘Original’ set of instances. All datasets were found through the website of Dua and Graff, 2017.

6.1.4. Software

This thesis expands on Pumpkin, a MaxSAT solver developed by Emir Demirović (one of the authors of this paper); it is yet to be published. Recall from chapter 2 that there are two flavours of MaxSAT solvers: linear search and core-guided search. Pumpkin supports both these flavours and allows the user to specify how much time should be spent in each mode. From initial testing, it became apparent that the core-guided mode is unequivocally faster at solving the instances than the linear-search strategy. For this reason, the runtime analysis is primarily focused on the performance of the core-guided solver.
The solver is written in C++14 and compiled on a virtual machine that mimics the Starexec environment. The following flags are passed to the compiler:

- NDEBUG: this flag disables any assertions in the code.
- O3: enables the highest level of optimisation.

Telemetry We collect data from the solver throughout the solving process. For example, the number of decisions that the solver has made is a performance indicator, and we collect this data by using counters in the code. We assume that the overhead imposed by these counters is negligible.

6.2. Results

6.2.1. Lazy Encoding

The lazy encoding, as described in chapter 4, is evaluated here.

Space Recall that the lazy encoding does not require the problem to be converted into conjunctive normal form (CNF); instead, it only stores the edges and their weights. Figure 6.1 shows the sizes of the different encodings. It's clear that the lazy encoder is several orders of magnitude more compact.

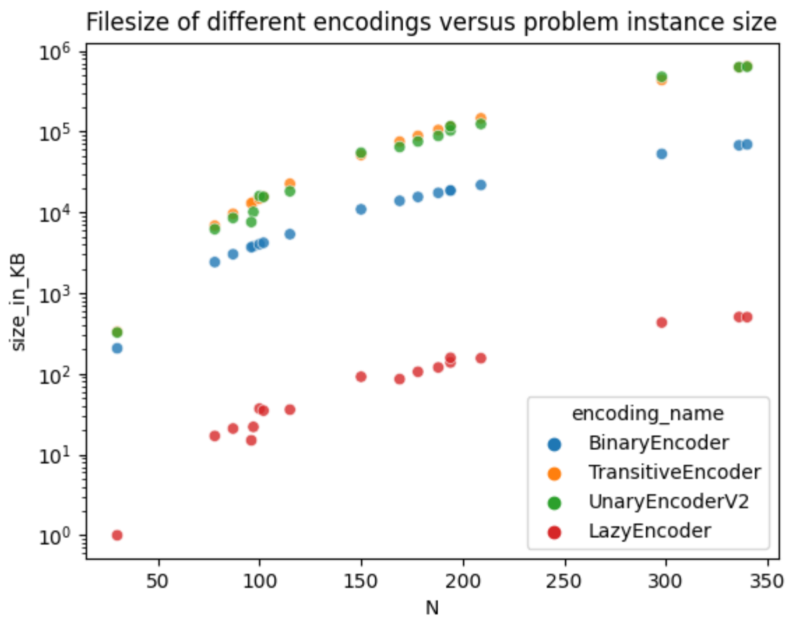


Figure 6.1: File sizes of different encodings. LazyEncoder is several orders of magnitude more compact than the others.

Runtime and decisions A downside of the dense encoding is that it is less optimised than a pure clausal representation of the problem and as a result, the runtime of the dense encoding is not as low as it could be. However, it is still competitive compared to the three encodings proposed by Berg and Järvisalo, 2017; in some cases, the lazy dense encoding outperforms all other encodings. Figure 6.2 shows the runtime and the number of decisions of our lazy encoding compared to the transitive and the unary encoding. Table **??**, which is delegated to the appendix, shows the raw data, including the number of decisions and the runtime in seconds. Also in the appendix is a table that aggregates the data, giving the mean number of normalised decisions and time as well as the standard deviation of these quantities. Table **??** shows these values.

6.2.2. Sparse encoding

We compared the sparse encoding to the dense encoding as well as the baseline WCNF encodings proposed by Berg and Järvisalo, 2017. Moreover, since the sparse encoding should theoretically perform better on sparse graphs, we generated new instances by 'trimming' some of the original instances in order to vary the density. Finally, we showcase the performance of the sparse encoding on the most sparse type of graph: the tree. Although it is unlikely to have a similarity graph that is a tree, it is valuable and interesting to see sparse's performance on these graphs.

Baseline comparison Figure 6.2 shows the performance of the sparse propagator on the set of instances. We see that the sparse propagator performs worse than its dense counterpart for almost every benchmark.

Learnt clauses The reason for sparse's worse performance is presumably not the extra work that is required to create explanations (see chapter 4 section 4.2.1), but the lower quality of the learnt clauses (see chapter 2 for an explanation on what constitutes a good clause). This hypothesis is fortified by figure 6.3. Here we see that the sparse solver mode requires far more decisions in order to solve the problem to optimality. Recall that the number of decisions is a deterministic metric that is not affected by 'bad code'. The fact that sparse requires more decisions can only be attributed to the difference between sparse and dense, which is the way they create their explanations. Indeed, figure 6.5 confirms this hypothesis. Figure 6.5 shows the average learnt clause size for every instance from the benchmark set; we see that sparse consistently has larger learnt clauses on average. Note that the relation between clause size and quality is not completely clear; it is often the case that a learnt clause of size 2 is much more powerful than one of size 3, so a small difference in clause size can have a big effect.

Influence of edge density The sparse encoding requires fewer variables than the dense encoding as the edge density of the instance decreases. We would expect that the sparse encoding becomes relatively more performant than the dense encoding; to test this hypothesis, we plot the relative performance of sparse and dense on the 'Inflated ' dataset (see table 6.1). Figure 6.6 shows the performance of the sparse encoding relative to the dense one. We can see that as the edge density decreases, the relative performance of sparse increases both in the required time to solve the instance as well as the number of decisions.

Trees An extreme example of a sparse graph is a tree. We investigate the performance of the sparse encoding when the similarity graph is a tree. The trees in this benchmark are generated synthetically and randomly. We were unable to generate WCNF encodings for trees larger than 350 nodes, because the program that encodes these instances ran out of memory. Moreover, the expected size of a tree with 2000 nodes in WCNF would be well over 50GB, making it hugely impractical to work with. However, it was no problem to encode very large trees in the custom format that is used for the dense and sparse encoding.
Figure 6.7 shows the performance of the sparse encoding on trees. We see that the sparse encoding scales much better than the dense encoding; at 5000 nodes, sparse performs over a thousand times better, both in terms of time as well as number of decisions. Due to aforementioned reasons, data is missing for Transitive, Unary and Binary at higher node counts. However, even the most conservative extrapolation of the data would conclude that Transitive and Unary scale much worse than sparse and dense on this dataset.

6.2.3. Bounds

Unfortunately, there seems to be a negative synergy between the triangle counting bounds introduced in chapter 5 and MaxSAT solvers using core-guided search. Figure 6.9 shows that triangle counting methods have virtually no effect on the program execution. For this reason, we include an analysis of the triangle counting methods with a solver that uses linear search.

Modes Recall from chapter 5 that we propose two different, independent (but similar) bounding techniques. It is possible to turn them on and off independently, which means there are four different solver modes:

1. Basic: does not use any of the techniques described in chapter 5.
2. Lambda: only uses the lambda technique.
3. Delta: only uses the delta technique.
4. LambdaDelta: uses both the lambda and delta technique.

Furthermore, each of these modes has the possibility to enable inferences (see section 5.2.4).

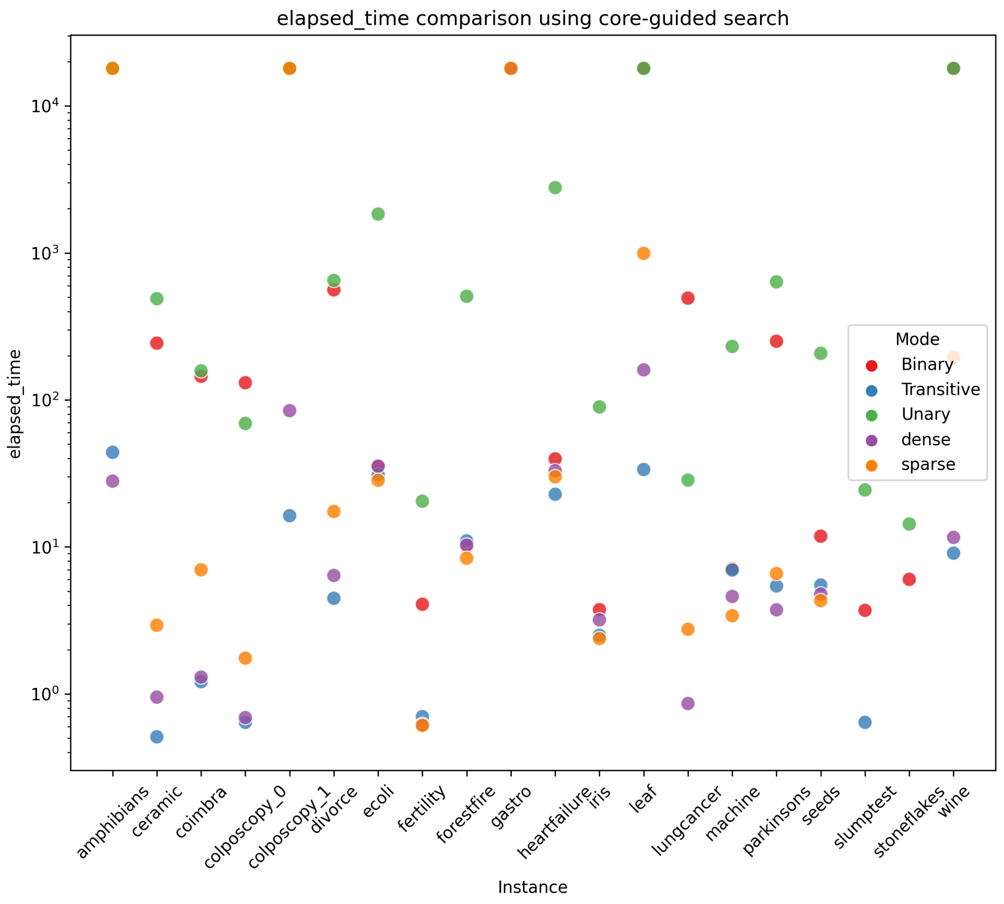
Evaluation of triangle counting effectiveness
We evaluate the performance of the different triangle counting methods on a modified set of instances. In figure 6.3, we use a collection of twenty instances; for the evaluation of the triangle counting methods, we inflate this dataset by randomly removing a percentage of the edges. The reason for this is that we would like to see the influence of the edge density and the maximum degree on the runtime.

Figure 6.8 shows three similar figures, each one comparing the effect of a triangle counting method to the baseline. The y-axis represents the normalised time (where the slowest mode for that instance gets the value 1). We can clearly see that the higher the maximum degree, the slower the solver is when triangle counting is enabled. Recall from chapter 5 that the complexity of the triangle counting method is a function of d_{max}. Figures 6.8b, 6.8c and 6.8d illustrate this. Furthermore, these figures show that the runtime is nearly always severely negatively impacted by using triangle counting methods; one would hope that the number of decisions is lower when counting triangles. Unfortunately, this is not the case: figure 6.9 shows the number of decisions required to solve the problem to optimality. There are very small differences for some instances, but they are negligible. Figure 6.10 shows the effect of the different modes of the bounds propagator on the normalised number of decisions required to prove optimality. We reiterate that figure 6.10 uses linear search instead of core-guided search. The figure shows that using the bounds propagator (in any of its modes) changes the number of decisions required, but not in a convincingly good or bad way. Looking at the aggregation of the data (represented as a distribution), we see that the differences are minor. Another notable feature is that the three graphs seem identical.

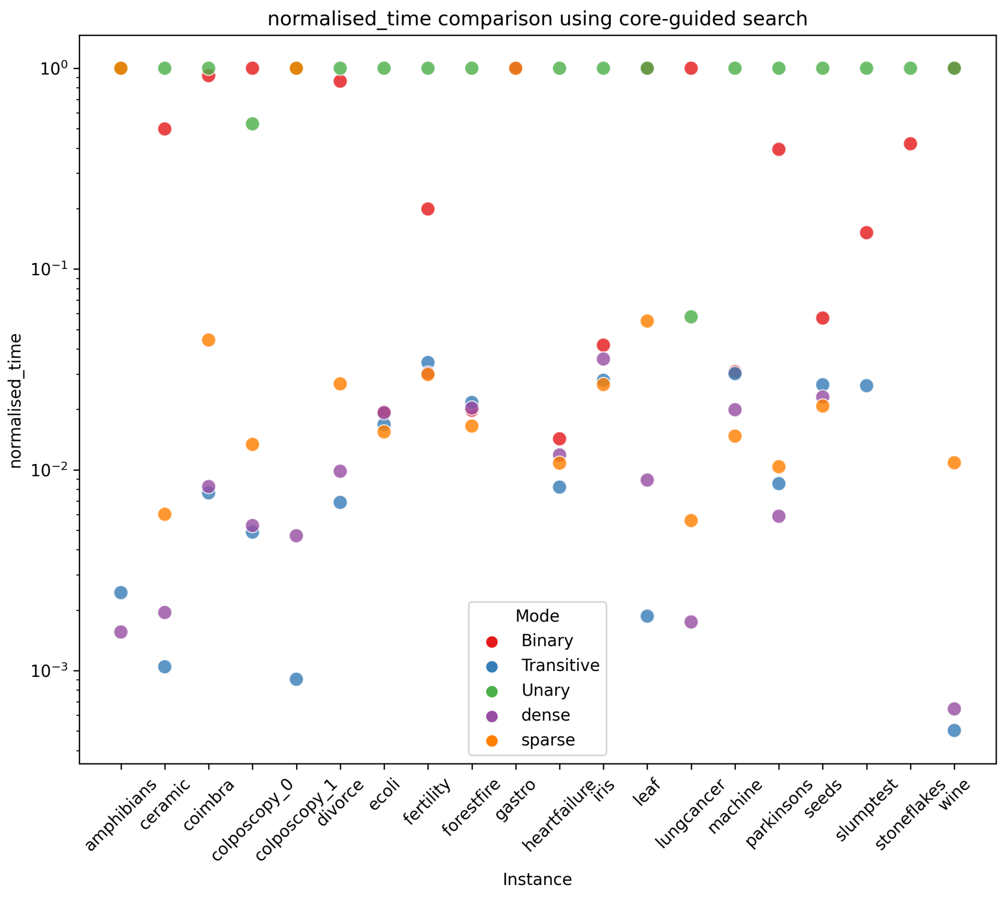
Evaluation of inferences
In this section, we investigate the effect of using inferences in combination with the bounds propagator. We benchmark four different modes of the solver:

1. Sparse: the sparse encoding without any bounds inferences.
2. Dense: the dense encoding without any bounds inferences.
3. Sparse-infer: the sparse encoding with bounds inferences enabled. Note that this mode does not make use of any triangle counting methods.
4. Dense-infer: the dense encoding with bounds inferences enabled. Note that this mode does not make use of any triangle counting methods.

For each of these modes, we investigate their effects when using core-guided search as opposed to using linear search.
Figure 6.11 shows the number of solved instances as a function of the maximum timeout. Figure 6.12 shows the number of decisions that are required to solve the problem to optimality. Combining the information from both figures, we see that when core-guided search is enabled, the bounds inferences do virtually nothing. However, when the solver uses linear search, there is a difference, which can be seen in figures 6.11b and 6.12b. However, it is unclear whether the effect is positive; in some cases, the number of decisions required to solve the problem is higher when bounds inferences are enabled.

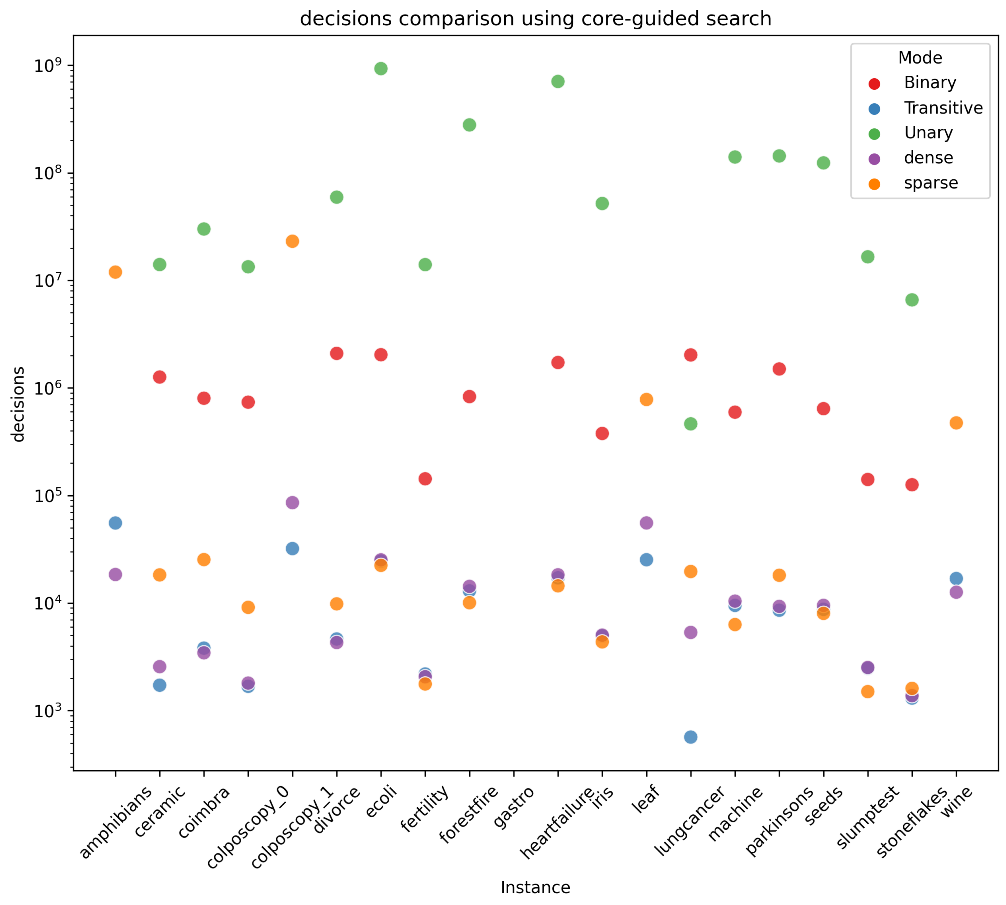


(a) Runtime in seconds.

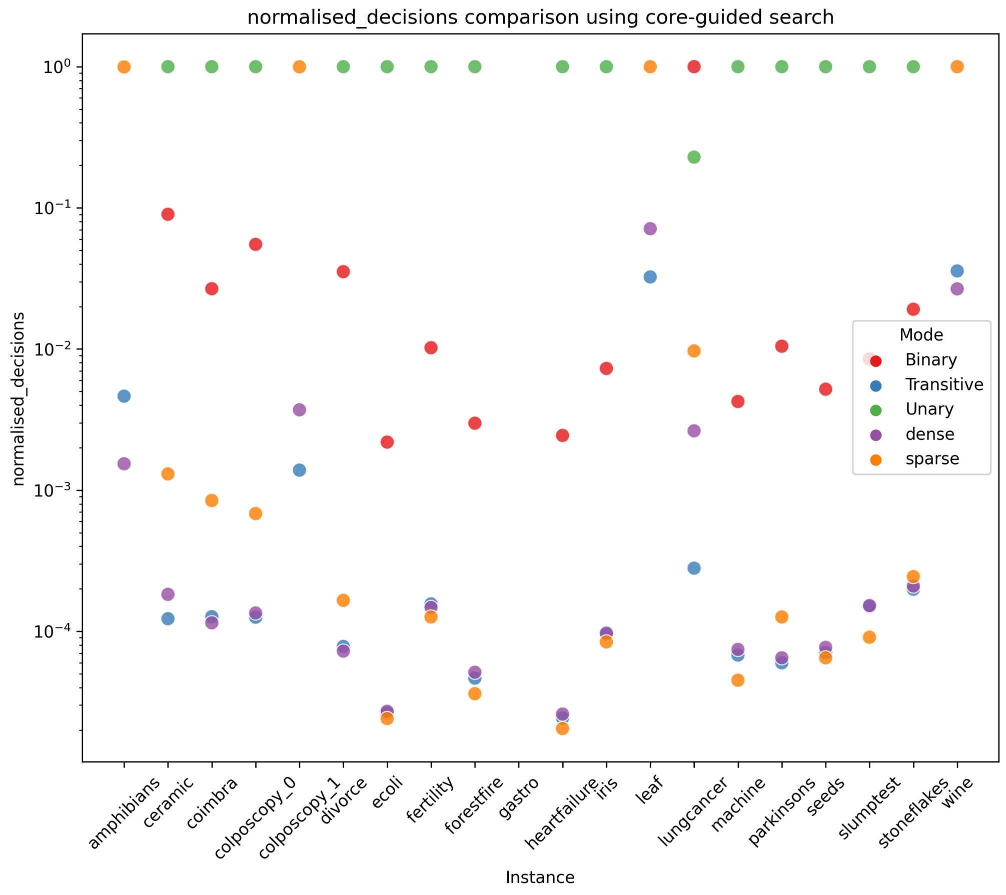


(b) Normalised runtime (unitless).

Figure 6.2: Runtime of Berg and Järvisalo, 2017's encodings versus our lazy and sparse encodings. Missing markers indicate a timeout. For example, Unary was unable to finish the instance Gastro within the time limit.

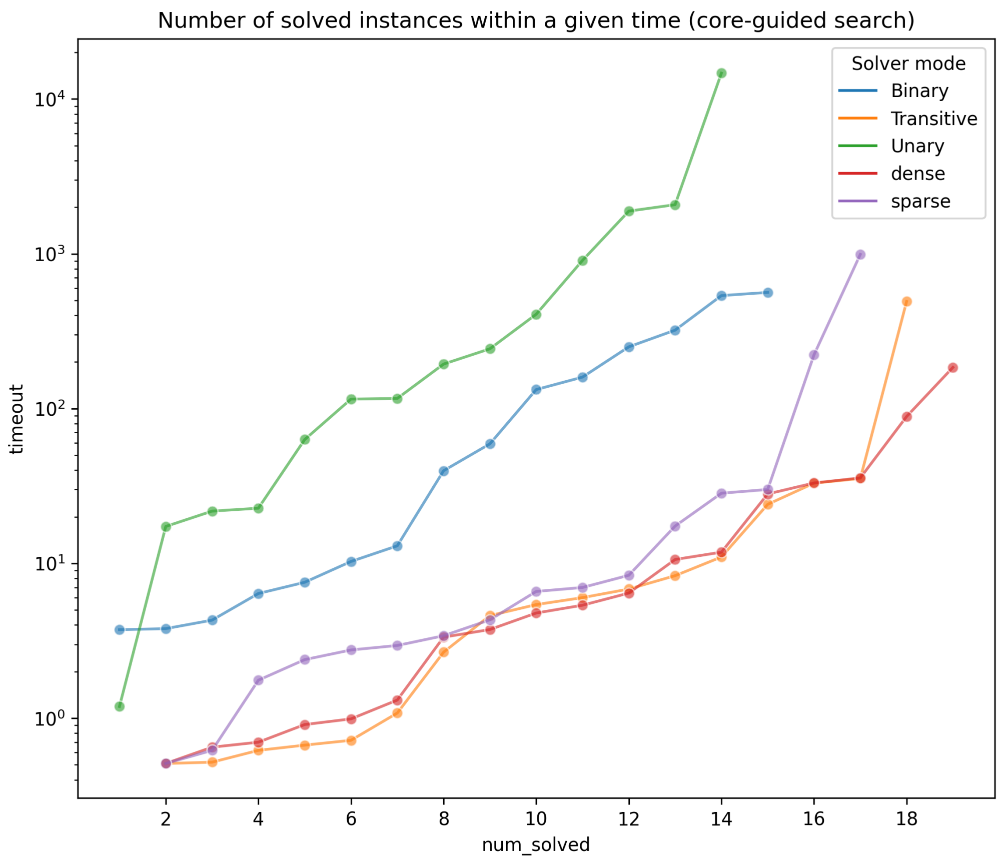


(a) Number of decisions required to prove optimality.

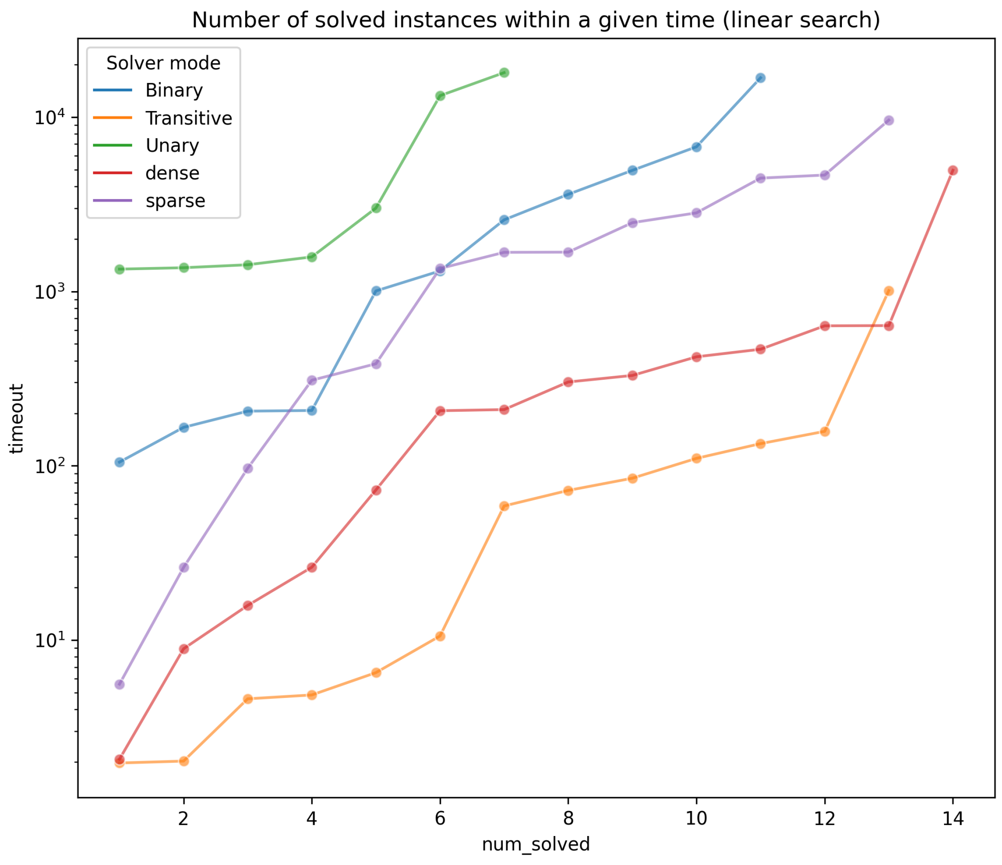


(b) Normalised number of decisions.

Figure 6.3: Runtime of Berg and Järvisalo, 2017's encodings versus our lazy and sparse encodings. The lazy encoding is called 'dense' here.



(a) Number of instances solved given a certain timeout. The solver uses core-guided search in this figure.



(b) Number of instances solved given a certain timeout. The solver uses linear search in this figure.

Figure 6.4: A plot showing the effect of turning on inference for both the sparse and dense encoding. See section 5.2.4 for an explanation of what inferences entails.

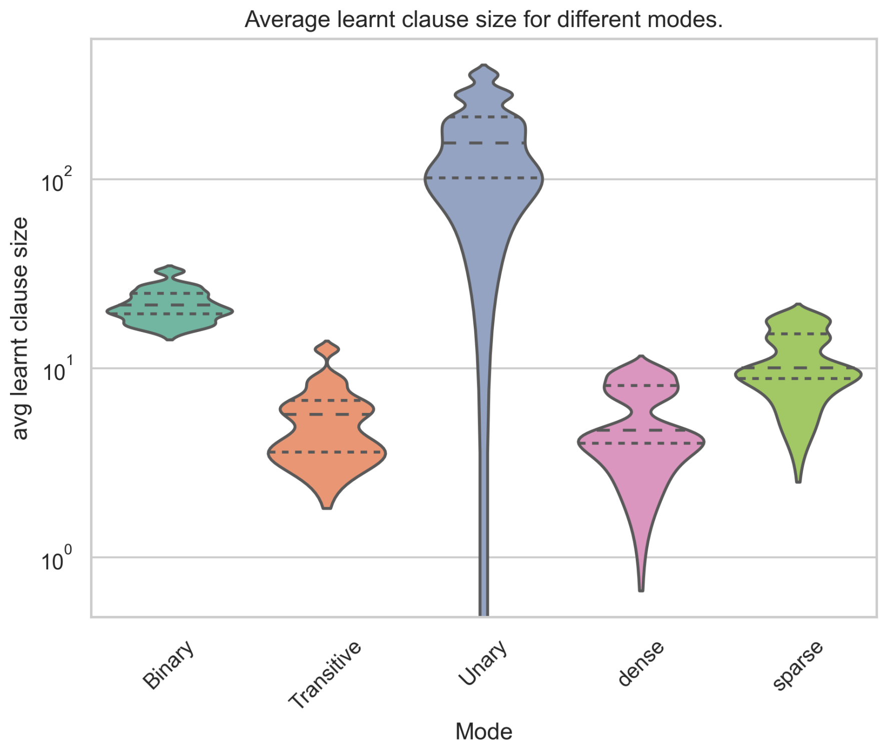
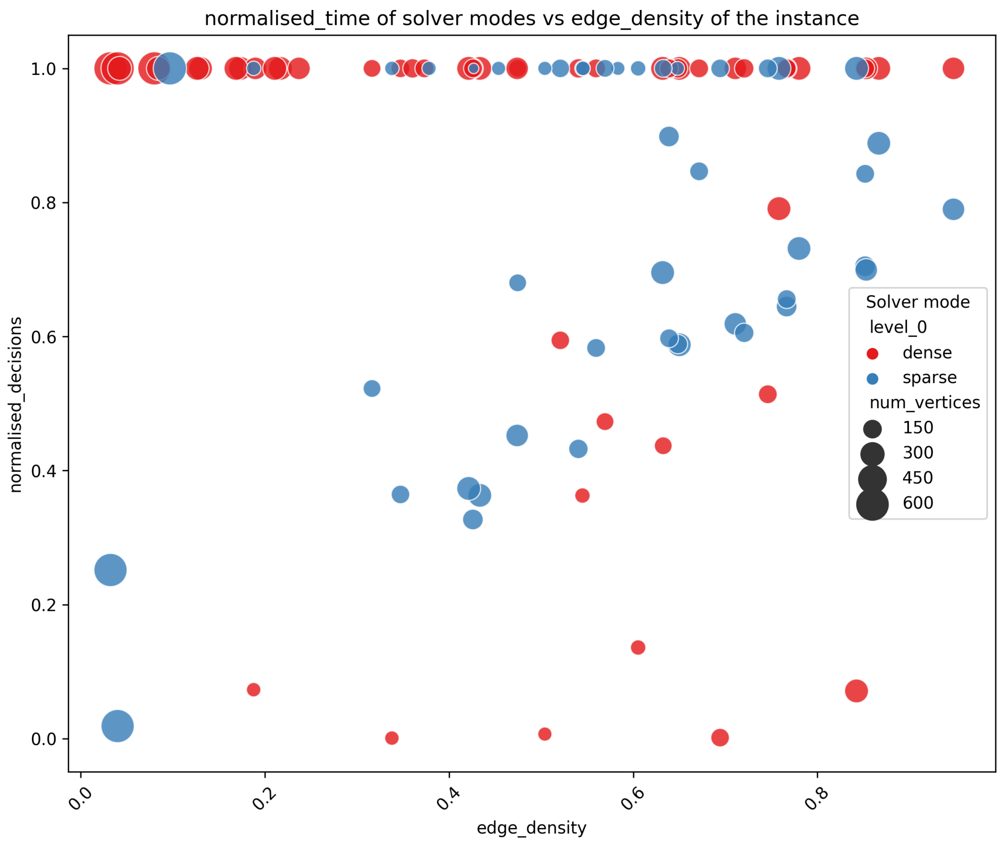
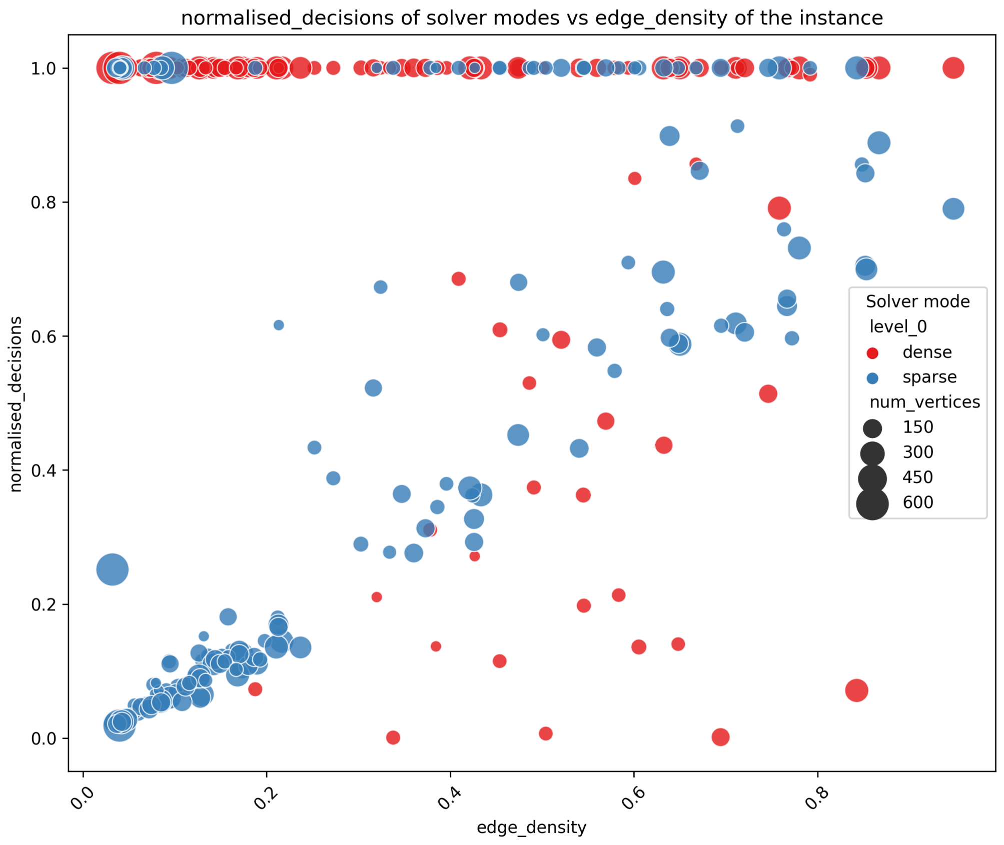


Figure 6.5: Violin plot for the average learnt clause size for different instances and different modes. As a general rule of thumb, smaller clauses are (much) better at pruning the search space than larger clauses. Dashed lines within violins indicate quartiles of the distribution. The dataset used for this plot is the ‘Original’ dataset (see table 6.1).

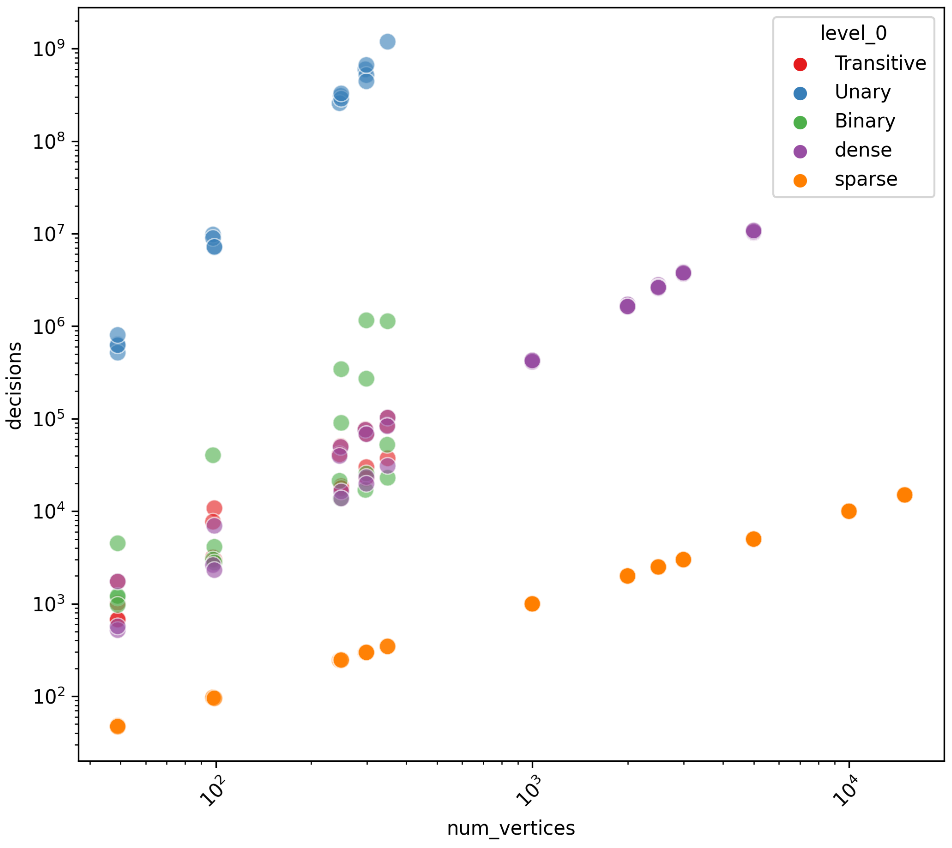


(a) The relative amount of time required to solve an instance for the modes dense and sparse. Each point represents a solver-instance pair; the size of the point indicates the instance's number of nodes. Note that we have chosen to omit any instance with a solving time less than 1 second, because these measurements could be inaccurate.

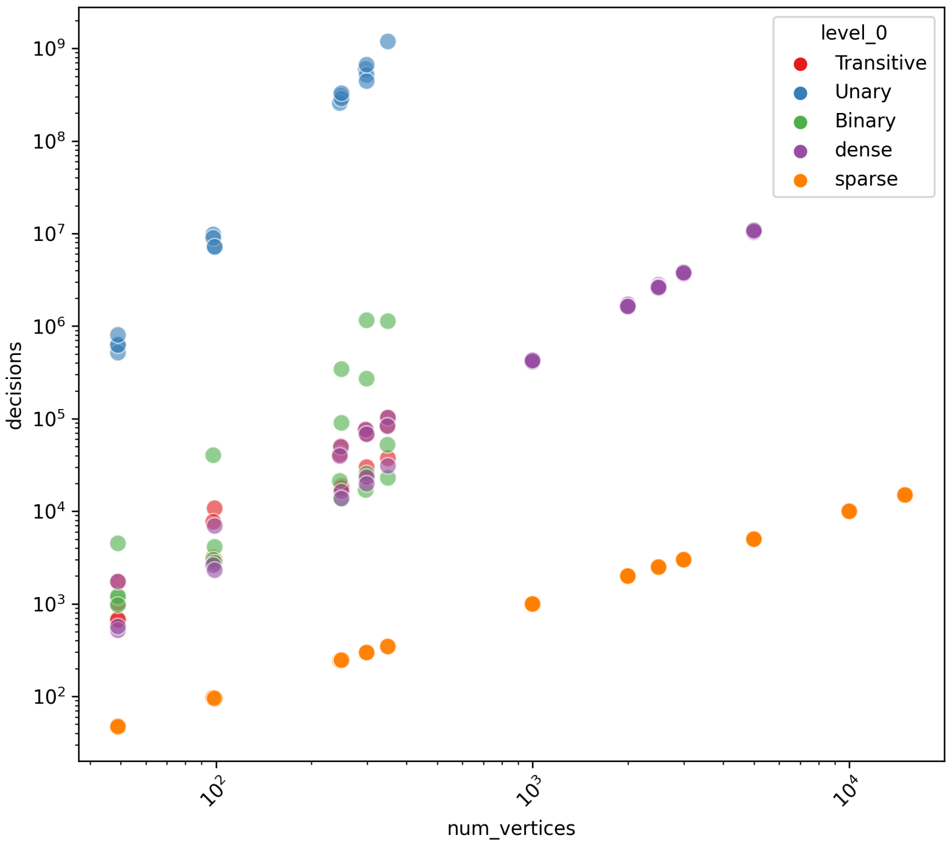


(b) The relative number of decisions required to solve the problem for the modes dense and sparse. Each point indicates a solver-instance pair; the size of the point indicates the instance's number of nodes. Contrary to figure 6.6a, we have not chosen to omit solver-instance pairs with a solving time less than 1 second. This is because the number of decisions is a deterministic metric, and therefore has no variance.

Figure 6.6: Two plots showing the relative performance of sparse and dense as a function of edge density. The dataset used for this plot is 'Inflated' (see table 6.1 for a short description on this dataset).

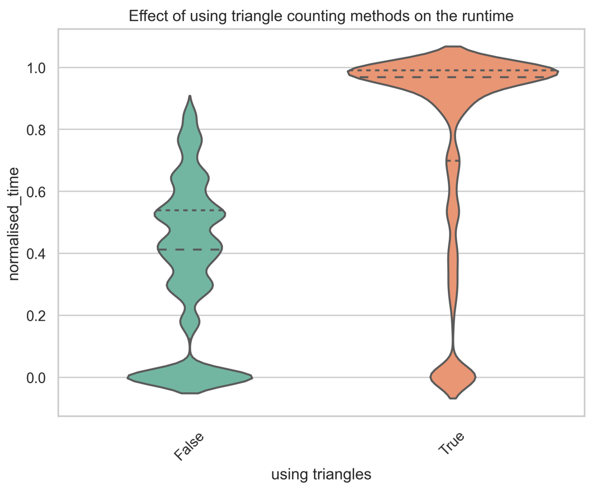


(a) Time required to solve the instance to optimality. The minimum resolution of the timer on Starexec was 1 millisecond;

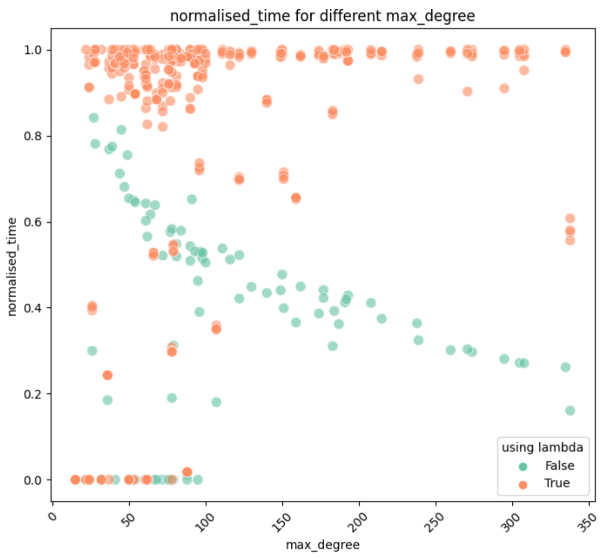


(b) Number of decisions required to reach optimality.

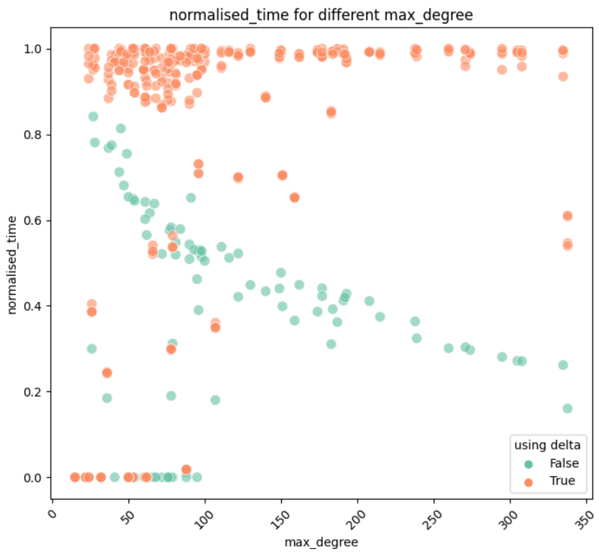
Figure 6.7: Comparing the performance of our encodings on tree-graphs. We see that the sparse encoding is almost indifferent to the problem size, whereas the baseline encodings by Berg and Järvisalo, 2017 break down very quickly. Missing datapoints represent either a timeout (for example, the dense encoding on the tree with 10k nodes), or an inability to encode the problem into the correct format (a Transitive encoding of a 1000-node tree would already be over 5 GB)



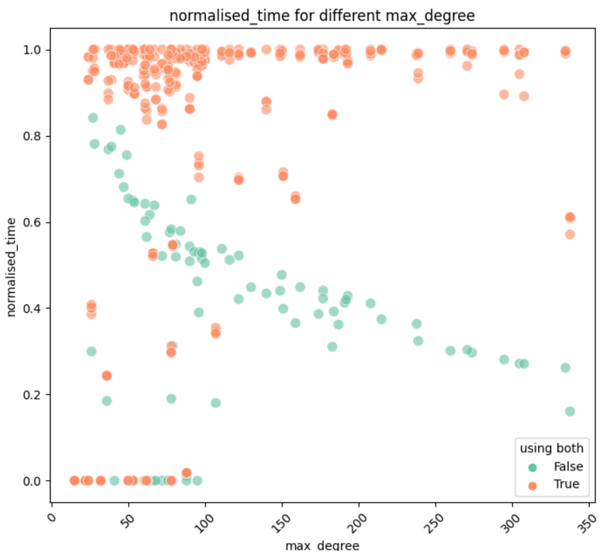
(a) Violin plot of the aggregated effect of using triangle counting methods on the runtime. Dashed lines within violins indicate quartiles of the distribution.



(b) Effect of counting Λ-triangles. Each point represents the normalised time that was required to solve the problem to optimality. Colours indicate whether the solver was using Λ-triangles.

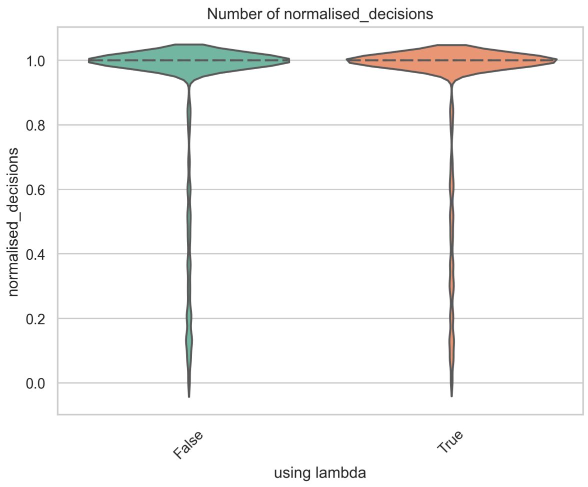


(c) Effect of counting Δ-triangles. Each point represents the normalised time that was required to solve the problem to optimality. Colours indicate whether the solver was using Δ-triangles.

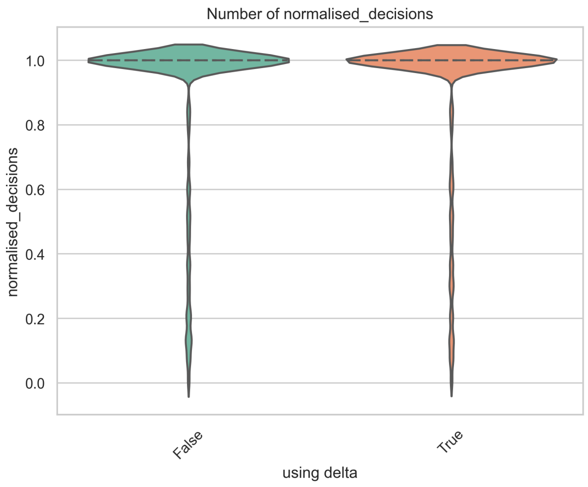


(d) Each point represents the normalised time that was required to solve the problem to optimality. Colours indicate whether the solver was using both Λ- and Δ-triangles.

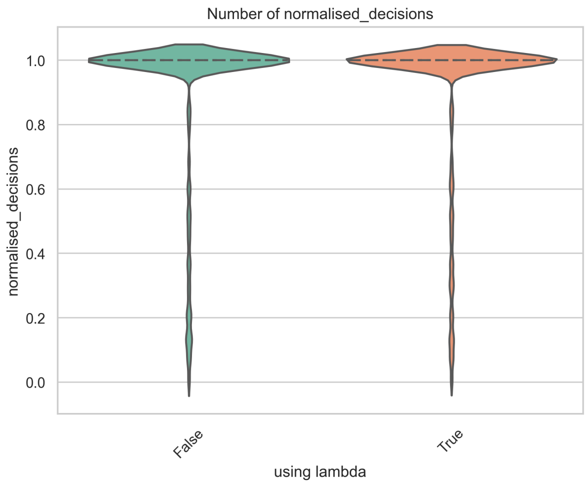
Figure 6.8: Comparing the runtime of several triangle modes: only Δ, only Λ, and both Δ and Λ. The set of instances used here is an inflated version of the ones used in figure 6.3. We create new instances by randomly removing a percentage of the edges of the original instances.



(a) Effect of counting Λ-triangles.

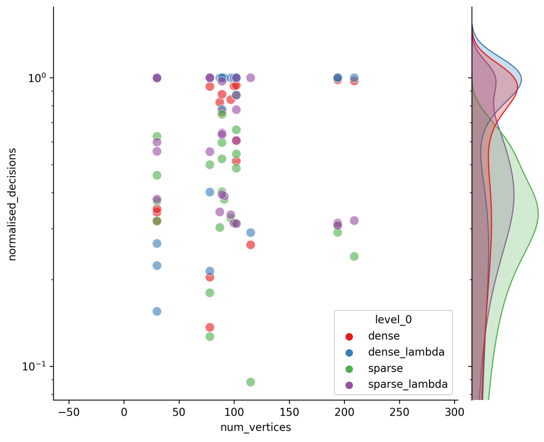


(b) Effect of counting Δ-triangles.

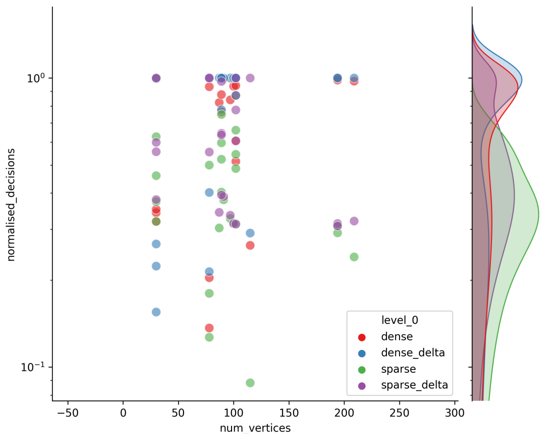


(c) Effect of counting Λ- and Δ-triangles at the same time.

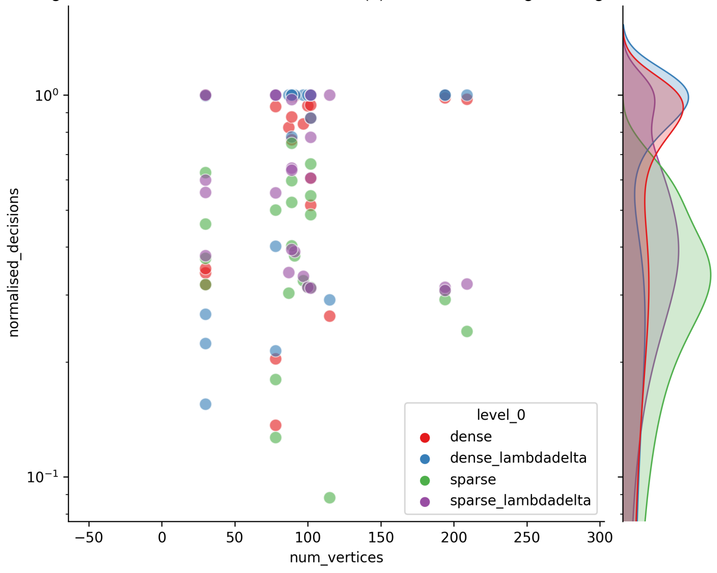
Figure 6.9: Comparing the number of decisions required to solve the problem to optimality where the MaxSAT solver uses core-guided search. We see that the effect of triangle counting methods is almost non-existent. For this figures, we have used the same set of instances as in figure 6.8.



(a) Effect of counting Λ-triangles.

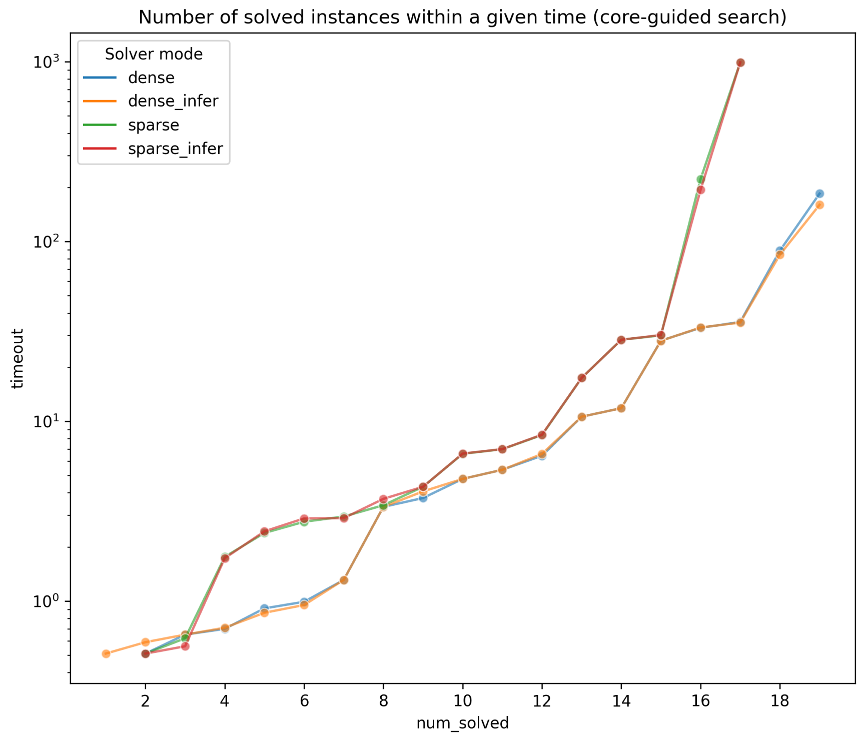


(b) Effect of counting Δ-triangles.

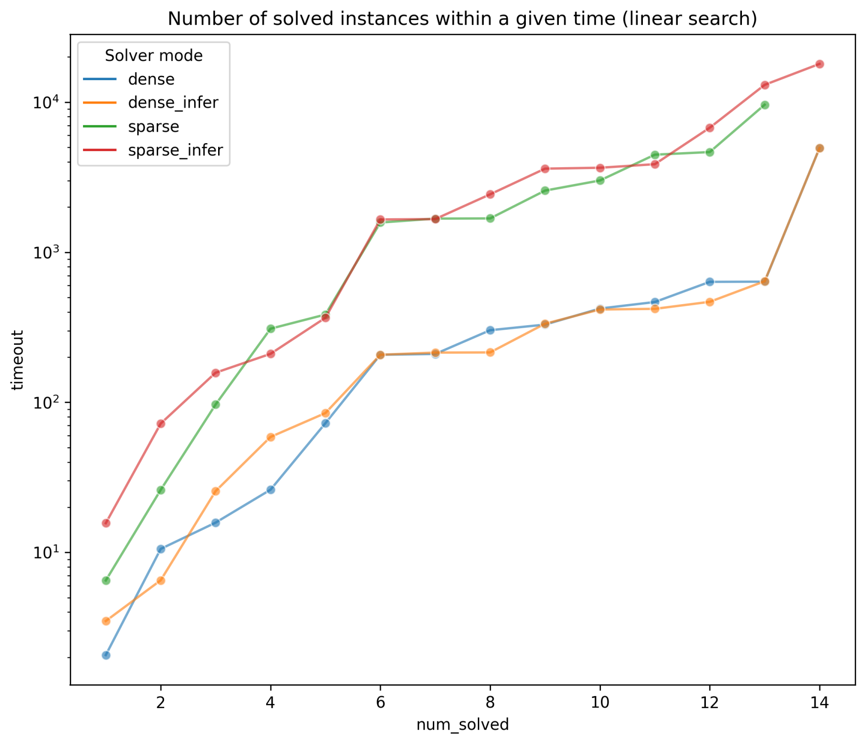


(c) Effect of counting Λ- and Δ-triangles at the same time.

Figure 6.10: Comparing the number of decisions required to solve the problem to optimality where the MaxSAT solver uses linear search. We can see that the bounds propagator does have an effect here, although it is not clear whether this is good.

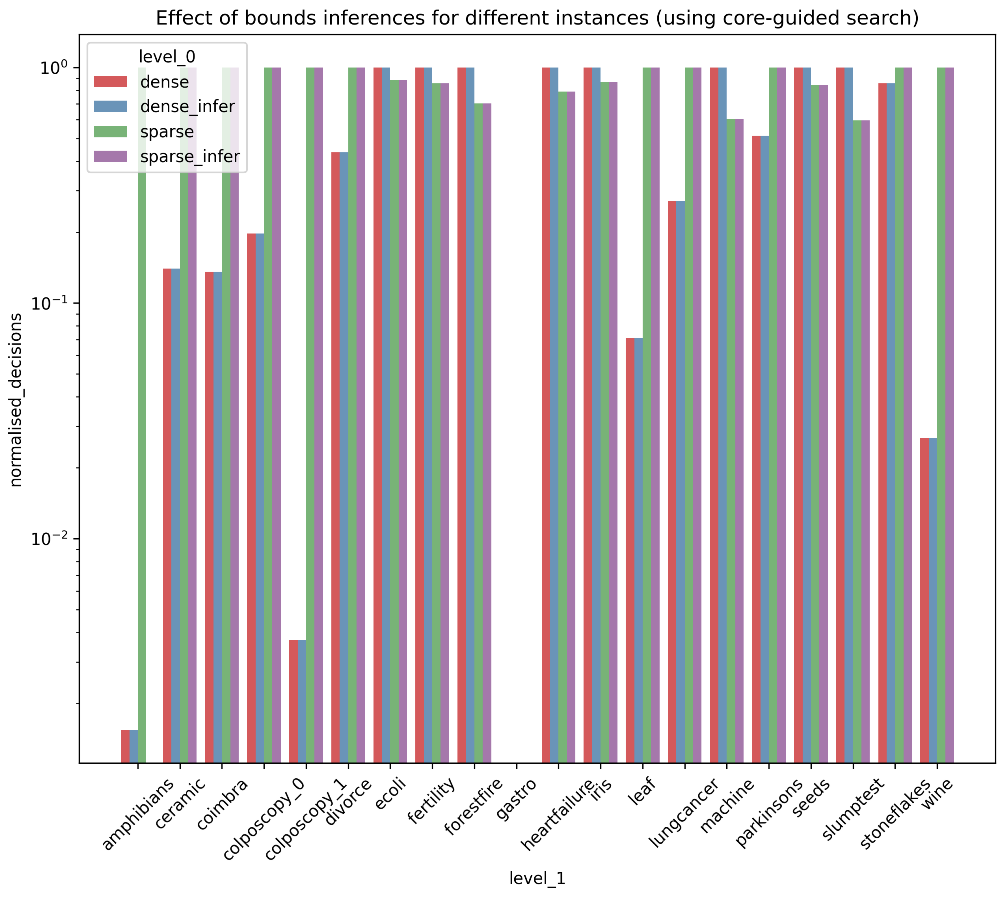


(a) Number of solved instances versus the maximum timeout. The solver uses core-guided search here.

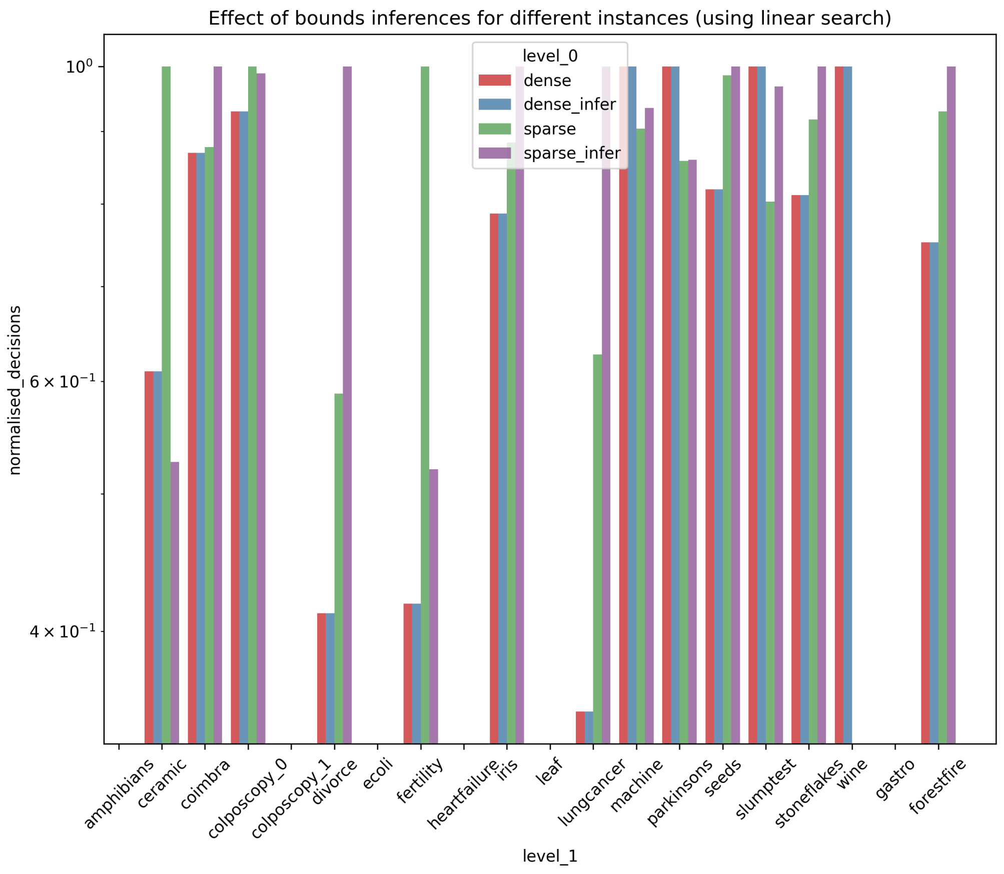


(b) Number of solved instances versus the maximum timeout. The solver uses linear search here.

Figure 6.11



(a) Effect of bounds inferences while the solver is using core-guided search.



(b) Effect of bounds inferences while the solver is using linear search.

Figure 6.12: Investigating the effect of using bounds inferences on the number of decisions required to prove optimality. The x-axis shows the name of the instance, and coloured dots in that column indicate the normalised number of decisions that was required by that solver in order to prove optimality. The dataset used for this image is the 'Original' dataset.

7 Discussion

In this chapter, we reflect on some of the findings of this thesis and speculate about possible explanations for them.

7.1. Theorising about noteworthy results

The experimental evaluation has yielded several noteworthy results which are not fully understood by us yet. Therefore, we speculate on their explanation in this section.

The first notable result is the fact that the effects of our bound propagator are not always positive. When using linear search, the bounds propagator will occasionally reduce the number of decisions required in order to prove optimality. However, oftentimes it will increase this number as well. This indicates that conflicts generated by the bounds propagator are harmful, in a sense that they do not efficiently contribute to pruning the search space. A possible explanation is that the bounds propagator's conflict prevents the solver from learning powerful clauses. Another possibility is that due to the conflicts generated by the propagator, a different but equally optimal solution is obtained.

Another notable result is the large number of decisions that are required by the sparse propagator. Even though sparse uses fewer variables than dense, the mean number of decisions is over forty times that of the dense propagator. The fact that sparse's explanations for variable propagations are by definition larger than the dense propagator (technically: not smaller) suggests that these explanations negatively impact the size of learnt clauses, which in turn increases the number of decisions required. On top of this, the explanations for propagations are sub-optimal, since a spanning tree is given, whereas a simple path would suffice. Nevertheless, we stand behind the current explanation generation scheme for sparse, since empirical tests have shown that it is much faster than having to find a path everytime a variable is propagated.

Finally, the experimental evaluation has revealed an inconsistency of our results with those of Berg and Järvisalo, 2017, who perform an evaluation of their MaxSAT encodings: Binary, Unary and Transitive. Their findings suggest that Binary is consistently faster than the other two encodings, but our results contradict this (see figure 6.4). A possible explanation for this is a mismatch between our implementation of Binary and that of Berg and Järvisalo, 2017. Of course, the implementation has been checked for correctness by ensuring that the optimal value of every instance is consistent across encodings. The difference could lie in the set of instances that has been chosen as benchmarks, or an incorrect interpretation on our part. Another possibility is the fact that Berg and Järvisalo, 2017 use the MaxHS solver, which is a different solver from the one we have used.

7.2. Reflection

In this section, we reflect on some of the results and either provide an explanation for these results or place them into context.

Performance: triangle counting The efficacy of the triangle counting methods proposed in chapter 5 is evaluated by investigating whether the method is capable of generating strong enough bounds,

and by seeing whether those bounds are actually worth the time that is required to compute them.
Based on the experimental evaluation, we can make two simple statements: when the solver is using core-guided search, none of the triangle counting methods do anything since the number of decisions is the same regardless of which mode is enabled. Next, we see that when the solver is using linear search instead, the triangle counting methods do affect the number of decisions required to solve the problem. However, this effect is neither significantly positive nor negative.
Figure 6.9 reveals a peculiar pattern: all three triangle counting modes have the exact same effect on the number of decisions. This strongly suggests that the observed effect comes solely from tracking the direct penalty of the partial solution, a feature which all three modes have in common. This would mean the concept of counting triangles is inferior and instead just keeping track of the lower bound by counting penalising assignments is sufficient. Furthermore, this hypothesis would explain the reason why there is no effect in the core-guided search mode of the solver, since the core-guided mode already keeps track of penalising assignments by itself.

Linear search vs. core-guided search In the initial stages of the project, the solver was only capable of doing linear search. Core-guided search was added in a later update (along with several other features), and it appeared that this method was several orders of magnitude more effective at solving instances. However, core-guided search diminished the effect of the lower bounding techniques presented in 5. Had we known that core-guided search would diminish the effects of the bounds propagator, we would not have spent time developing the triangle counting methods any further.

Sparse similarity graphs The sparse encoding becomes more powerful as the density of the similarity graph decreases. Recall that our method of creating similarity graphs follows the method proposed by Berg and Järvisalo, 2017. First, all similarities between every pair of points are computed, yielding a number in $[-0.5, 0.5]$. Then, all edges with a similarity that is not ‘extreme’ enough are removed. For example, we might remove all edges with similarities in $[-0.1, 0.1]$. For very high dimensional data, the similarity might suffer from the curse of dimensionality. This in turn would lead to many edges having a similarity that is not extreme, leading to the removal of these edges. Therefore, high dimensional data may lead to sparse similarity graphs which could be beneficial for the sparse encoding.

8 Conclusion and future work

In this thesis, we have proposed and evaluated a hybrid MaxSAT solver specifically engineered for solving correlation clustering. This work aims to advance the capabilities of exact solvers for machine learning problems because exhaustive search provides strong benefits regarding fair AI. Key features of the solver include a lazy representation of clauses that reduces problem sizes by several orders of magnitude, a novel encoding that reduces solving times on sparse graphs, and a novel bounding technique whose effectiveness is very minor when using linear search and nonexistent when using core-guided search. The lazy dense encoding performs slightly worse than the baseline transitive encoding on a set of real-world benchmarks but this difference is not significant. Additionally, the fact that the lazy encodings can be extended further through the propagator interface justifies a preference for the lazy encodings over the baselines.

8.1. Recommendations for further research

An answer is only as good as the number of questions it spawns. Here, we describe some promising directions for future research on a conceptual and technical level.

8.1.1. Sparse: unique paths are relatively rare

The sparse encoding introduced in chapter 4 proposes a formulation with fewer variables, but this comes at the cost of having to do work to compute paths within a cluster. We give an algorithm to compute such paths efficiently, but computing paths might not be necessary altogether; instead, we may choose to introduce new variables during the search process only as they are needed. Whenever our propagator is asked to explain one of its inferences, we add an extra variable to the solver which exists only to reduce the size of the explanation.

Wouldn't this approach introduce many new variables, defeating the whole purpose of the sparse encoding? It does not: relatively few variables would be introduced using such a scheme. In fact, the larger the graph, the fewer variables would have to be introduced. Figure 8.1 shows the relative number of variables that would have to be introduced as a function of the graph's number of nodes. The y-axis is the number of new edges relative to $n(n-1)/2$. For a graph with 300 nodes, approximately 10% of $300 \cdot 299/2 = 89700$ new variables would have to be introduced. Depending on the number of edges of the original graph, this number might be negligible and the approach of introducing new variables may improve runtimes.

8.1.2. Tree-width

In chapter 6, we've seen that the sparse encoding convincingly outperforms the other approaches when used on trees. Trees are easily identified by their structure, but the odds of encountering a similarity graph that is exactly a tree are very slim. Instead, we might use a notion of 'tree-ness' instead. The term treewidth is a measure for how similar a graph is to a tree; a graph with high treewidth looks nothing like a tree, whereas a graph with treewidth equal to one *is* a tree. Unfortunately, computing the treewidth of a graph is NP-complete (shown by Arnborg et al., 1987), but plentiful approximation

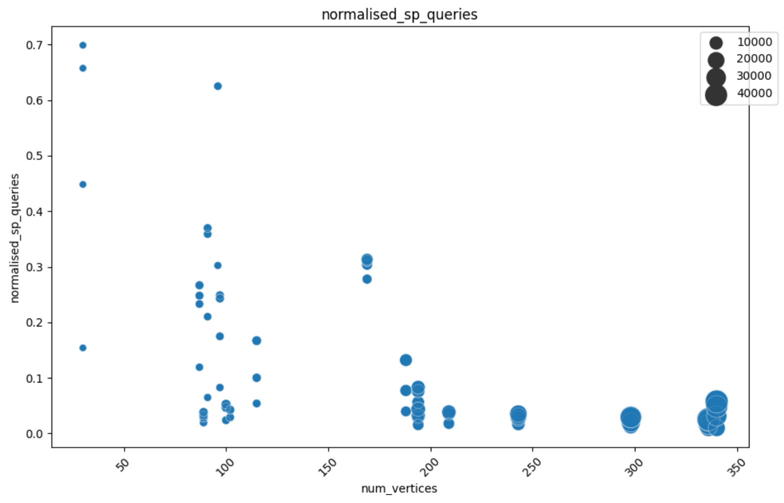


Figure 8.1: The number of unique shortest path queries that are done throughout the solving process. Shortest path calculations are done whenever we create an explanation for an inferred literal; if we choose to introduce a new variable instead, we would need equally many variables as there are shortest paths. The y-axis is normalised, so that a value of 1 indicates that there were $n(n-1)/2$ unique shortest path queries.

algorithms exist as shown by Bodlaender, 1997.
It would be interesting to see whether the performance of the sparse encoding increases not only inversely with the edge density, but also with the treewidth of a graph.

8.1.3. Applications to other problems

The ideas presented in this thesis are primarily concerned with solving the problem of correlation clustering but several components are theoretically usable in other domains. Specifically, the dense and sparse encodings are reusable in problems where the encoding represents an equivalence relation. Any combinatorial optimisation problem where the search space is encoded using an equivalence relation is a potential candidate. An additional use case for our work is a clustering or partitioning problem where the similarities between points are sparse or expensive to compute.

Concluding remarks

The integration of combinatorial optimisation into machine learning tasks is an exciting and promising research direction. This work proposes algorithms that improve the performance of one technique (hybrid MaxSAT solvers) on one particular problem: correlation clustering. We are curious about future scientific work that explores different combinations of exhaustive search with fundamental machine learning problems.

A Appendix

Table A shows the number of decisions and the runtime of our work, dense and sparse, versus these same metrics for the baseline encodings by Berg and Järvisalo, 2017. The runtime is in seconds and the timeout is 18 000 seconds.
Table A.1 shows the mean and standard deviations of the normalised number of decisions and elapsed time for our modes on the Original dataset.

level_1	Binary decisions	Transitive decisions	Unary decisions	dense decisions	sparse decisions	Binary time	Transitive time	Unary time	dense time	sparse time
amphibians	-	**55521**	-	18438	11937757	-	44.01	-	**28.02**	-
ceramic	1262934	**1725**	14031527	2566	18288	243.3	0.51	488.04	**0.95**	2.93
coimbra	802502	3815	30026078	**3451**	25372	144.66	**1.21**	157.56	1.3	6.98
colposcopy_0	737579	**1684**	13380187	1805	9131	130.78	**0.64**	69.1	0.69	1.75
colposcopy_1	-	**32110**	-	85871	23104435	-	**16.3**	-	84.5	-
divorce	2095352	4640	59350829	**4300**	9840	560.07	**4.47**	649.64	6.39	17.41
ecoli	2036847	24589	929972166	25244	**22423**	35.68	30.9	1837.2	35.37	**28.39**
fertility	143023	2193	14008886	2068	**1770**	4.07	0.7	20.45	0.62	**0.61**
forestfire	831293	12989	278807280	14300	**10079**	10.04	10.97	506.84	10.29	**8.37**
gastro	-	-	**706595803**			-	-	**2780.3**	-	-
heartfailure	1725008	17177	51834603	18321	**14469**	39.68	**22.8**	89.55	32.98	30
iris	377466	4946	-	5037	**4359**	3.74	2.5	-	3.19	**2.38**
leaf	-	**25302**	463506	55600	781000	-	33.6	**28.46**	159.98	991.11
lungcancer	2027283	**568**	140186860	5337	19656	492.35	**0**	230.98	0.86	2.75
machine	595446	9520	143476854	10432	**6316**	7.1	6.96	635.17	4.6	**3.4**
parkinsons	1501404	**8583**	123634789	9316	18131	250.57	5.42	207.38	**3.73**	6.58
seeds	641789	8747	16558215	9522	**8024**	11.82	5.5	24.39	4.78	**4.31**
slumptest	140962	2478	6587940	2521	**1504**	3.7	0.64	14.29	**0**	**0**
stoneflakes	125948	**1308**	-	1379	1610	6.01	**0**	-	**0**	**0**
wine	-	16922	-	**12625**	473403	-	**9.06**	-	11.6	195.19

Mode	Mean normalised decisions	Std normalised decisions	Mean normalised time	Std normalised time
Binary	0.0853	0.254	0.531	0.435
Transitive	**0.00399**	0.0107	0.0613	0.221
Unary	0.949	0.199	0.926	0.236
dense	0.00564	0.0170	**0.0604**	0.221
sparse	0.211	0.418	0.165	0.360

Table A.1: Aggregated runtimes and number of decisions for the Original dataset. The mean and standard deviations are taken over the normalised quantities.

Bibliography

Aeberhard, S., Coomans, D., & De Vel, O. (1994). Comparative analysis of statistical pattern recognition methods in high dimensional settings. *Pattern Recognition*, *27*(8), 1065–1077.

Ailon, N., Charikar, M., & Newman, A. (2005). Aggregating inconsistent information. *Proceedings of the thirty-seventh annual ACM symposium on Theory of computing - STOC 05*. https://doi.org/10.1145/1060590.1060692

Arnborg, S., Corneil, D. G., & Proskurowski, A. (1987). Complexity of finding embeddings in ak-tree. *SIAM Journal on Algebraic Discrete Methods*, *8*(2), 277–284.

Bansal, N., Blum, A., & Chawla, S. (2004). Correlation clustering. *Machine Learning*, *56*(1-3), 89–113. https://doi.org/10.1023/b:mach.0000033116.57574.95

Bayardo Jr, R. J., & Schrag, R. (1997). Using csp look-back techniques to solve real-world sat instances. *Aaai/iaai*, 203–208.

Becker, H. (2005). A survey of correlation clustering.

Berg, J., & Järvisalo, M. (2017). Cost-optimal constrained correlation clustering via weighted partial maximum satisfiability. *Artificial Intelligence*, *244*, 110–142. https://doi.org/10.1016/j.artint.2015.07.001

Bergadano, F., Giordana, A., Saitta, L., De Marchi, D., & Brancadori, F. (1990). Integrated learning in a real domain. *Machine learning proceedings 1990* (pp. 322–329). Elsevier.

Biere, A., Biere, A., Heule, M., van Maaren, H., & Walsh, T. (2009). *Handbook of satisfiability: Volume 185 frontiers in artificial intelligence and applications*. IOS Press.

Biere, A., Heule, M., & van Maaren, H. (2009). *Handbook of satisfiability* (Vol. 185). IOS press.

Blachnik, M., Sołtysiak, M., & Dąbrowska, D. (2019). Predicting presence of amphibian species using features obtained from gis and satellite images. *ISPRS International Journal of Geo-Information*, *8*(3), 123.

Bodlaender, H. L. (1997). Treewidth: Algorithmic techniques and results. *International Symposium on Mathematical Foundations of Computer Science*, 19–36.

Bonizzoni, P., Vedova, G. D., Dondi, R., & Jiang, T. (2008). On the approximation of correlation clustering and consensus clustering. *Journal of Computer and System Sciences*, *74*(5), 671–696. https://doi.org/10.1016/j.jcss.2007.06.024

Buss, S., & Nordström, J. (2021). Proof complexity and sat solving. *Handbook of Satisfiability*, *336*, 233–350.

Cai, S., & Zhang, X. (2020). Pure maxsat and its applications to combinatorial optimization via linear local search. *International Conference on Principles and Practice of Constraint Programming*, 90–106.

Charytanowicz, M., Niewczas, J., Kulczycki, P., Kowalski, P. A., Łukasik, S., & Żak, S. (2010). Complete gradient clustering algorithm for features analysis of x-ray images. *Information technologies in biomedicine* (pp. 15–24). Springer.

Chicco, D., & Jurman, G. (2020). Machine learning can predict survival of patients with heart failure from serum creatinine and ejection fraction alone. *BMC medical informatics and decision making*, *20*(1), 1–16.

Cook, S. A. (1971). The complexity of theorem-proving procedures. *Proceedings of the third annual ACM symposium on Theory of computing - STOC 71*. https://doi.org/10.1145/800157.805047

Cortez, P., & Morais, A. d. J. R. (2007). A data mining approach to predict forest fires using meteorological data.

Davis, M., Logemann, G., & Loveland, D. (1962). A machine program for theorem-proving. *Communications of the ACM*, *5*(7), 394–397. https://doi.org/10.1145/368273.368557

Dua, D., & Graff, C. (2017). UCI machine learning repository. http://archive.ics.uci.edu/ml

Eén, N., & Sörensson, N. (2004). An extensible sat-solver. *Theory and Applications of Satisfiability Testing Lecture Notes in Computer Science*, 502–518. https://doi.org/10.1007/978-3-540-24605-3_37

Estivill-Castro, V. (2002). Why so many clustering algorithms: A position paper. *ACM SIGKDD explorations newsletter*, *4*(1), 65–75.

Fernandes, K., Cardoso, J. S., & Fernandes, J. (2017). Transfer learning with partial observability applied to cervical cancer screening. *Iberian conference on pattern recognition and image analysis*, 243–250.

Fisher, R. A. (1936). The use of multiple measurements in taxonomic problems. *Annals of eugenics*, *7*(2), 179–188.

Gael, J., & Zhu, X. (n.d.). Correlation clustering for crosslingual link detection. https://www.ijcai.org/Proceedings/07/Papers/282.pdf

Ganesh, V., & Vard, M. Y. (n.d.). On the unreasonable e ectiveness of sat solvers. https://www.cs.rice.edu/~vardi/papers/SATSolvers21.pdf

Gil, D., Girela, J. L., De Juan, J., Gomez-Torres, M. J., & Johnsson, M. (2012). Predicting seminal quality with artificial intelligence methods. *Expert Systems with Applications*, *39*(16), 12564–12573.

Goldberg, E. I., Prasad, M. R., & Brayton, R. K. (2001). Using sat for combinational equivalence checking. *Proceedings Design, Automation and Test in Europe. Conference and Exhibition 2001*, 114–121.

He, Z., Zhang, M., & Zhang, H. (2016). Data-driven research on chemical features of jingdezhen and longquan celadon by energy dispersive x-ray fluorescence. *Ceramics International*, *42*(4), 5123–5129.

Hebrard, E., & Katsirelos, G. (2020). Constraint and satisfiability reasoning for graph coloring. *Journal of Artificial Intelligence Research*, *69*, 33–65. https://doi.org/10.1613/jair.1.11313

Hong, Z.-Q., & Yang, J.-Y. (1991). Optimal discriminant plane for a small number of samples and design method of classifier on the plane. *pattern recognition*, *24*(4), 317–324.

Little, M., McSharry, P., Roberts, S., Costello, D., & Moroz, I. (2007). Exploiting nonlinear recurrence and fractal scaling properties for voice disorder detection. *Nature Precedings*, 1–1.

Marques, J., & Silva, J. P. M. (1996). Grasp—a new search algorithm for satisfiability. *in Proceedings of the International Conference on Computer-Aided Design*, 220–227.

Mesejo, P., Pizarro, D., Abergel, A., Rouquette, O., Beorchia, S., Poincloux, L., & Bartoli, A. (2016). Computer-aided classification of gastrointestinal lesions in regular colonoscopy. *IEEE transactions on medical imaging*, *35*(9), 2051–2063.

Miyauchi, A., Sonobe, T., & Sukegawa, N. (2018). Exact clustering via integer programming and maximum satisfiability. *32nd AAAI Conference on Artificial Intelligence, AAAI 2018*, 1387–1394.

Miyauchi, A., & Sukegawa, N. (2015). Redundant constraints in the standard formulation for the clique partitioning problem. *Optimization Letters*, *9*(1), 199–207.

Mycielski, J. (1955). Sur le coloriage des graphes. *Colloq. Math*, *3*(161-162), 9.

Patrı́cio, M., Pereira, J., Crisóstomo, J., Matafome, P., Gomes, M., Seiça, R., & Caramelo, F. (2018). Using resistin, glucose, age and bmi to predict the presence of breast cancer. *BMC cancer*, *18*(1), 1–8.

Silva, J. M., & Sakallah, K. (1997). Grasp-a new search algorithm for satisfiability. *Proceedings of International Conference on Computer Aided Design*. https://doi.org/10.1109/iccad.1996.569607

Silva, P. F., Marcal, A. R., & da Silva, R. M. A. (2013). Evaluation of features for leaf discrimination. *International Conference Image Analysis and Recognition*, 197–204.

Sinz, C. (2005). Towards an optimal cnf encoding of boolean cardinality constraints. *Principles and Practice of Constraint Programming - CP 2005 Lecture Notes in Computer Science*, 827–831. https://doi.org/10.1007/11564751_73

Stump, A., Sutcliffe, G., & Tinelli, C. (2014). Starexec: A cross-community infrastructure for logic solving. *Automated Reasoning Lecture Notes in Computer Science*, 367–373. https://doi.org/10.1007/978-3-319-08587-6_28

Weber, T. (2009). The lower/middle palaeolithic transition. is there a lower/middle palaeolithic transition? *Preistoria Alpina*, *44*, 17–24.

Yeh, I.-C. (2007). Modeling slump flow of concrete using second-order regressions and artificial neural networks. *Cement and concrete composites*, *29*(6), 474–480.

Yöntem, M. K., Kemal, A., Ilhan, T., & KILIÇARSLAN, S. (2019). Divorce prediction using correlation based feature selection and artificial neural networks. *Nevşehir Hacı Bektaş Veli ”Universitesi SBE Dergisi*, *9*(1), 259–273.