The ghosts of forgotten things: A study on size after forgetting

Paolo Liberatore[*]

Abstract

Forgetting is removing variables from a logical formula while preserving the constraints on the other variables. In spite of being a form of reduction, it does not always decrease the size of the formula and may sometimes increase it. This article discusses the implications of such an increase and analyzes the computational properties of the phenomenon. Given a propositional Horn formula, a set of variables and a maximum allowed size, deciding whether forgetting the variables from the formula can be expressed in that size is D^p-hard in Σ_2^p. The same problem for unrestricted propositional formulae is D_2^p-hard in Σ_3^p. The hardness results employ superredundancy: a superirredundant clause is in all formulae of minimal size equivalent to a given one. This concept may be useful outside forgetting.

1 Introduction

Several articles mention simplification as an advantage of forgetting, if not its motivation. After all, forgetting means deleting some piece of knowledge, and less is more. Less knowledge is easier to remember, easier to work with, easier to interpret. To cite a few:

- "With an ever growing stream of information, bounded memory and short response time suggest that not all information can be kept and treated in the same way. [...] forgetting [...] helps us to deal with information overload and to put a focus of attention" [12].
- "For example, in query answering, if one can determine what is relevant with respect to a query, then forgetting the irrelevant part of a knowledge base may yield more efficient query-answering" [9].
- "Moreover, forgetting may be applicable in summarising a knowledge base by suppressing lesser details, or for reusing part of a knowledge base by removing an unneeded part of a larger knowledge base, or in clarifying relations between predicates" [8].
- "For performing reasoning tasks (planning, prediction, query answering, etc.) in an action domain, not all actions of that domain might be necessary. By instructing the reasoning system to forget about these unnecessary/irrelevant actions, without changing the causal relations among fluents, we might obtain solutions using less computational time/space" [14].

[*]DIIAG, Sapienza University of Rome. `liberato@diag.uniroma1.it`

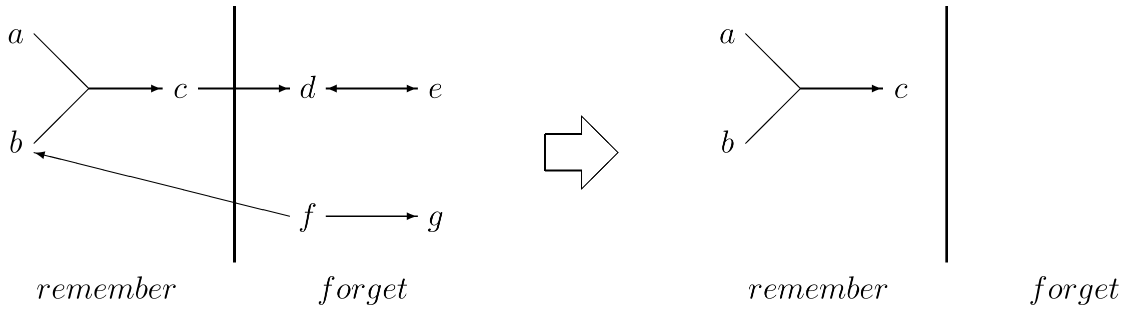
- "There are often scenarios of interest where we want to model the fact that certain information is discarded. In practice, for example, an agent may simply not have enough memory capacity to remember everything he has learned" [15].
- "The most immediate application of forgetting is to model agents with limited resources (e.g., robots), or agents that need to deal with vast knowledge bases (e.g., cloud computing), or more ambitiously, dealing with the problem of lifelong learning. In all such cases it is no longer reasonable to assume that all knowledge acquired over the operation of an agent can be retained indefinitely" [34].
- "For example, we have a knowledge base K and a query Q. It may be hard to determine if Q is true or false directly from K. However, if we discard or forget some part of K that is independent of Q, the querying task may become much easier" [43].
- "To some extent, all of these can be reduced to the problem of extracting relevant segments out of large ontologies for the purpose of effective management of ontologies so that the tractability for both humans and computers is enhanced. Such segments are not mere fragments of ontologies, but stand alone as ontologies in their own right. The intuition here is similar to views in databases: an existing ontology is tailored to a smaller ontology so that an optimal ontology is produced for specific applications" [11].

These authors are right: if forgetting simplifies the body of knowledge then it is good for reducing the amount of information to store, for increasing the efficiency of querying it, for clarifying the relationships between facts, for obtaining solutions more easily, for retaining by agents of limited memory, for tailoring knowledge to a specific application. If forgetting simplifies the body of knowledge, all these motivations are valid.

If.

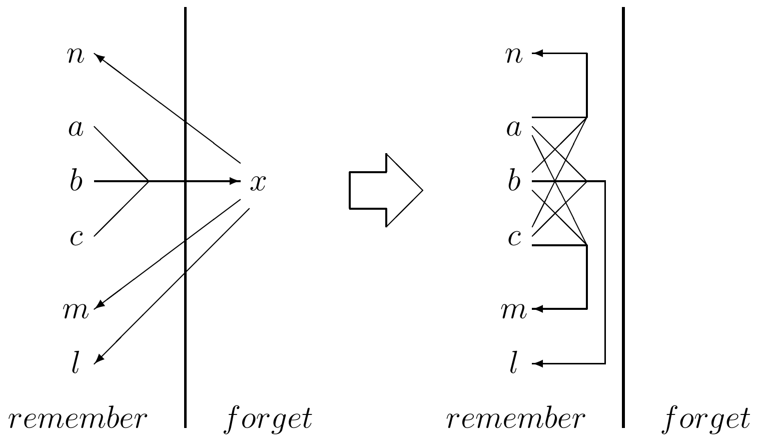
What if not? What if forgetting does not simplify the body of knowledge? What if it complicates it? What if it enlarges it instead of making it smaller?

This looks impossible. Forgetting is removing. Removing information, but still removing. Removing something leaves less, not more. What remains is less than what before, not more. Forgetting about d, e, f and g only leaves information about a, b and c.



The only information that remains is that a and b imply c. All the rest, like c implying d or f implying f is forgotten. What is left is smaller because it is only a part of what before.

This is the prototypical scenario of forgetting, the first that comes to mind when thinking about removing information: some information goes away, the rest remains. The rest is a part of the original. Smaller. Simpler. Easier to store, to query, to interpret. But prototypical does not mean exclusive.



Forgetting x complicates things instead of simplifying them. Whenever a, b and c are the case so is x. And x implies n, m and l. Like the neck of an hourglass, x funnels the first three variables in the upper bulb to the last three in the lower. Without it, these links need to be spelled out one by one: a, b and c imply n; a, b and c imply m; a, b and c imply l. The variable x acts like a shorthand for the first three variables together. Removing it forces repeating them.

Forgetting x deletes x but not its connections with the other variables. The lines that go from a, b and c to n, l and m survive. Like a ghost, x is no longer there in its body, but in its spirit: its bonds. These remain, weaved where x was.

The formula resulting from forgetting is still quite short, but this is only because the example is designed to be simple for the sake of clarity. Cases with larger size increase due to forgetting are easy to find.

The size of the formula resulting from forgetting matters for all reasons cited by the authors above. To summarize, it is important for:

1. sheer memory needed;
2. the cost of reasoning; formulae that are difficult for modern solvers are typically large; while efficiency is not directly related to size, small formulae are usually easy to solve
3. interpreting the information; the size of a formula tells something about how much the forgotten variables are related to the others.

The figures visualize forgetting as a cut between what is remembered and what is forgotten. This cut may divide parts that are easy to separate like in the first figure or parts that are not natural to separate like in the second figure. The first cut glides following the direction of the fabric of the knowledge base. The second is resisted by the connections it cuts.

The number of these connections does not tell the difference. How closely they hold together the parts across the cut does. Even if the links in the first example were $c \rightarrow d_1, \ldots, c \rightarrow d_{100}$ instead of $c \rightarrow d$, the result would be the same. What matters is not how many links connect the parts across the cut, but how they do.

An increase of size gauges the complexity of these connections. The first example is easy to cut because its implications are easy to ignore: c may imply d, but if d is forgotten this implication is removed and nothing else changes. Not the same in the second example:

forgetting x does not just remove its implication from a, b and c; it shifts its burden to the remaining variables.

A size increase suggests that the forgotten variables are closely connected to the remaining one. If forgetting is aimed at subdividing knowledge, it would be like the chapter on Spain next to that of Samoa and far away from France in an atlas. The natural division is by continents, not initials of the name. In general, the natural divisions are by topics, so that things closely connected stay close to each other. Forgetting about Samoa when describing Spain is easier than forgetting about France. Neglecting some obscure diplomatic relations is more natural than neglecting a bordering country.

Forgetting may be abstracting [29]. Cold weather increases virus survival, which facilitates virus transmission, which causes flu. Forgetting about viruses: cold weather causes flu. But forgetting is not always natural as an abstraction. A low battery level, a bad UPS unit and a black out cause a laptop not to start; which causes a report not to be completed, a movie not to be watched and a game not to be played. Forgetting about the laptop is a complication more than an abstraction: the three preconditions cause the first effect, they cause the second effect, and they cause the third effect. If x is the laptop not starting this is the example in the second figure, where forgetting increases size. That cold weather causes cold is short, simple, a basic fact of life for most people. Brevity is the soul of abstraction.

In summary, a short formula is preferred for storage and computational reasons. The size of the formula after forgetting is important for epistemological reason, to evaluate how natural a partition or abstraction of knowledge is. Either way, the question is: how large is a formula after forgetting variables?

The question is not obvious as it looks. Several formulae represent the same piece of knowledge. For example, $a \vee (\neg a \wedge b)$ is the same as the shorter $a \vee b$. The problem of formula size without forgetting eluded complexity researchers for twenty years: it was the prototypical problem for which the polynomial hierarchy was created in the seventies [37], but framing it exactly into one of these classes only succeeded at the end of the nineties [40]. This is the problem of whether a formula is equivalent to another of a given size.

The problem studied in this article is whether forgetting some variables from a formula is equivalent to a formula of given size.

Forgetting is not complicated. A simple recipe for forgetting x from F is: replace x with true in F, replace x with false in F, disjoin the two resulting formulae [23]. If F is in conjunctive normal form, another recipe is: replace all clauses containing x with their resolution [8, 35]. The first solution may not maintain the syntactic form of the formula. None of them is guaranteed to produce a minimal one.

Forgetting no variable from a formula results in the formula itself. Insisting on forgetting something does not change complexity: every formula F is the result of forgetting x from $F \wedge x$ if x is a variable not in F. The complexity of the size of F is a subcase of the size of forgetting x from F. It is however not an interesting subcase: the question is how much size decreases or increases due to forgetting. If F has size 100 before forgetting and 10 after, this looks like a decrease. But is not if F is equivalent to a formula of size 5 before forgetting and to none of size 9 or less afterwards. This is a size increase, not a decrease.

The next section introduces the concept of forgetting. The most significant part of the definition is that forgetting is not a one-in/one-out transformation. Because of equivalence, forgetting can be expressed in several ways. Rather than preferring one over the others, forgetting is defined as a relation between formulae: F' expresses forgetting x from F. As every other formula F'' equivalent to F' does. Some equivalent formulation of forgetting are given, and some ways to compute a formula that expresses forgetting.

Diving straight into the complexity analysis would present a problem: while membership of forgetting to certain complexity classes is easy to prove, hardness is not. The problem is that many formulae are equivalent to any given one, the same that made minimization without forgetting difficult to computationally classify. A mechanism that simplifies the task is to ensure that some parts of the formula remain after minimization. These parts act as guides that keep the formula in a certain shape. A similar idea is to ensure that some clauses survive forgetting and then minimization: that of superredundancy. It has its own section because of the number and length of the involved proofs. Another motivation is that it may be useful beyond hardness proof, in the more general class of existence proofs.

Complexity is studied first in the Horn restriction because of its slightly simpler proofs. It is then continued in the general case.

A number of examples and counterexamples rely on calculating the resolution closure of a formula, its redundant and superredundant clauses, its minimal equivalent formulae, the result of forgetting a variable from it and the minimal formulae equivalent to that. Calculating them step by step would be long and tedious. The program `minimize.py` is instead referred to in these cases. It is a small Python [42] program aimed at simplicity rather than efficiency.

2 Preliminaries

2.1 Formulae

The formulae in this article are all propositional in conjunctive normal form (CNF): they are set of clauses, a clause being the disjunction of some literals and a literal a propositional variable or its negation.

The variables a formula A contains are denoted $Var(A)$.

Definition 1 *The size of a formula is the number of variables occurrences it contains.*

This is not the same as the cardinality of $Var(A)$ because a variable may occur multiple times in a formula. For example, $A = \{a, \neg a \vee b, a \vee \neg b\}$ has size five because it contains five literal occurrences even if its variables are only two. The size is obtained by removing from the formula as it is written all propositional operators, commas and parenthesis and counting the number of symbols left.

Other definitions are possible but are not considered in this article. An alternative measure of size is the total number of symbols a formula contains (including conjunctions, disjunctions, negations and parenthesis). Another is the number of clauses (regardless of their length).

The definition of size implies the definition of minimality: a formula is minimal if it is equivalent to no formula smaller than it. Given a formula, a minimal equivalent formula is a possibly different but equivalent formula that is minimal. As an example, $A = \{a, \neg a \vee b, a \vee \neg b\}$ has size five since it contains five literal occurrencies; yet, it is equivalent to $B = \{a, b\}$, which only contains two literal occurrencies. No formula equivalent to A or B is smaller than that: B is minimal. Minimizing a formula means obtaining a minimal equivalent formula. This problem has long been studied [22, 7].

Definition 2 *The clauses of a formula A that contain a literal l are denoted by $A \cap l = \{c \in A \mid l \in c\}$.*

This notation cannot cause confusion: when is between two sets, the symbol $\cap$ denotes their intersection; then is between a set and a literal, it denotes the clauses of the set that contain the literal. This is like seeing $A \cap l$ as the shortening of $A \cap \text{clauses}(l)$, where $\text{clauses}(l)$ is the set of all possible clauses that contain the literal l.

When a formula entails a clause but none of its strict subclauses, the clause is a prime implicate of the formula. Formally, $F \models c$ holds but $F \models c'$ does not for any clause c' comprising a strict subset of the literals of c. Prime implicates are a common tool in formula minimization [22, 7] but not in this article, where a concept based on resolution takes its place.

2.2 Resolution

Resolution is a syntactic derivation mechanism that produces a new clause that is a consequence of two clauses: $c_1 \vee l, c_2 \vee \neg l \vdash c_1 \vee c_2$. Sometimes $\vdash_R$ is used in place of $\vdash$ to emphasize the use of resolution as the syntactic derivation rule. This is unnecessary in this article since no other derivation rule is ever mentioned.

Unless noted otherwise, tautologies are excluded. Writing $c_1 \vee a, c_2 \vee \neg a \vdash c_1 \vee c_2$ implicitly assumes that none of the three clauses is a tautology unless explicitly stated. Two clauses that would resolve in a tautology are considered not to resolve; this is not a limitation since resolution derivations containing tautologies can be simplified by removing them along with the clauses generating them. Tautologies are forbidden in formulae, which is not a limitation either since tautologies are always satisfied. This assumption has normally little importance, but is crucial to superredundancy, a concept defined in the next section.

Lemma 1 *A clause that is the result of resolving two clauses does not contain the resolving variable and is different from both of them if none of them is a tautology.*

Proof. A clause c is the result of resolving two clauses only if they have the form $c_1 \vee a$ and $c_2 \vee \neg a$ for some variable a, and c is $c_1 \vee c_2$. If $c_1 \vee c_2$ is equal to $c_1 \vee a$ then it contains a. Since clauses are sets of literals, they do not contain repeated elements. As a result, $a \notin c_1$ as otherwise $c_1 \vee a$ would contain a twice. Together with $a \in c_1 \vee c_2$ this implies $a \in c_2$, which makes $c_2 \vee \neg a$ a tautology. The case $c = c_2 \vee \neg a$ is similar. □

In what follows tautologies are excluded from formulae and from resolution derivations. As a result, resolving two clauses always generate a clause different from them.

A resolution proof $F \vdash G$ is a binary forest where the roots are the clauses of G, the leaves the clauses of F and every parent is the result of resolving its two children. The alternative form of direct acyclic graphs can be converted to this by duplicating subgraphs. The increased number of nodes does not constitute a problem because resolution is not used for automated theorem proving in this article but as a theoretical tool.

Definition 3 *The resolution closure of a formula F is the set $ResCn(F) = \{c \mid F \vdash c\}$ of all clauses that result from applying resolution zero or more times to the clauses of F.*

The clauses of F are derivable by zero-step resolutions from F. Therefore, $F \vdash c$ and $c \in \text{ResCn}(F)$ hold for every $c \in F$.

The resolution closure is similar to the deductive closure but not identical. For example, $a \vee b \vee c$ is in the deductive closure of $F = \{a \vee b\}$ but not in the resolution closure. It is a consequence of F but is not obtained by resolving clauses of F.

All clauses in the resolution closure $\text{ResCn}(F)$ are in the deductive closure but not the other way around. The closures differ because resolution does not expand clauses: $a \vee b \vee c$ is not a resolution consequence of $a \vee b$. Adding expansion kills the difference [24, 36].

$$F \models c \text{ if and only if } c' \in \text{ResCn}(F) \text{ for some } c' \subseteq c$$

That resolution does not include expansion may suggest that it cannot generate any non-minimal clause. That would be too good to be true, since it would prove a clause minimal just because it is obtained by resolution. In facts, it is not the case. Expansion is only one of the reasons clauses may not be minimal, as seen in the formula $\{a \vee b \vee c, a \vee b \vee e, \neg e \vee c \vee d\}$: the second and third clauses resolve to $a \vee c \vee b \vee d$, which is however not minimal: it is contained in the first clause of the formula, $a \vee b \vee c$. This example is in the `resolutionnotminimal.py` file of `minimize.py`.

What is the case is that resolution generates all prime implicates [24, 36], the minimal entailed clauses. The relation between $\text{ResCn}(F)$ and the deductive closure of F tells that if a clause is entailed a subset of it is generated by resolution; since the only entailed subclause of a prime implicate is itself, it is the only one resolution may generate. Removing all clauses that contains others from $\text{ResCn}(F)$ results in the set of the prime implicates of F.

Minimal equivalent formulae are all made of minimal entailed clauses, as otherwise literals could be removed from them. Since resolution allows deriving all prime implicates of a formula [24, 36] it derives all clauses of all minimal equivalent formulae.

Property 1 *If B is a minimal CNF formula equivalent to A, then $B \subseteq ResCn(A)$.*

This property relies on size being defined as the total number of literal occurrences. It does not hold in the alternative definition where size is the number of clauses, since that allows a clause that is not minimal by itself to be in a minimal formula.

While $\text{ResCn}(F)$ contains all clauses generated by an arbitrary number of resolutions, some properties used in the following require the clauses obtained by a single resolution step.

Definition 4 *The resolution of two formulae is the set of clauses obtained by resolving each clause of the first formula with each clause of the second:*

$$resolve(A, B) = \{c \mid c', c'' \vdash c \textit{ where } c' \in A \textit{ and } c'' \in B\}$$

If either of the two formulae comprises a single clause, the abbreviations $resolve(A, c) = resolve(A, \{c\})$, $resolve(c, B) = resolve(\{c\}, B)$ and $resolve(c, c') = resolve(\{c\}, \{c'\})$ are used.

A single clause of A resolved with a single clause of B and nothing else: this is $\text{resolve}(A, B)$. Exactly one resolution of one clause with one clause. Not zero, not multiple ones. A clause of A is not by itself in $\text{resolve}(A, B)$ unless it is also the resolvent of another clause of A with a clause of B.

3 Forgetting

Forget is defined semantically: no specific formula is given as the unique result of forgetting variables from another. That would separate forgetting from size optimization: forget from A results in B, which may then be equivalent to a smaller formula B'. This double step is an unjustified complication: because of equivalence, B' still expresses forgetting from A. Rather than defining B as the result of forgetting and then searching among its equivalent formulae B', every equivalent formula such as B' is defined as expressing forgetting.

This definition is not only simpler, it also better captures the meaning of forgetting: same information, just limited to some variables [8]. The information a knowledge base carries is its set of models, the situations it tells possible. Or the formulae it entails. If B and B' entail the same formulae, they are both valid ways of forgetting.

In this view, the whole point of A is query answering: does condition C hold according to the information A carries? Forgetting some variables means losing information; namely, all information related to the forgotten variables. And vice versa: all information not related to the forgotten variables survives. In propositional logic, $A \models C$ coincides with $B \models C$ for every formula C not involving the forgotten variables, otherwise B does not express forgetting these variables from A. Adding that B only contains the variables not to be forgotten completes the definition.

Definition 5 *A formula B expresses forgetting all variables from A but Y if and only if $Var(B) \subseteq Y$ and $B \models C$ is the same as $A \models C$ for all formulae C such that $Var(C) \subseteq Y$.*

The definition sets a constraint over B rather than uniquely defining a specific formula. Every formula B fits it as long as it is built over the right variables and has the right consequences.

Syntax is irrelevant to this definition. As it should: every B' that is syntactically different but equivalent to B carries the same information. There is no reason to confer $A[\mathsf{true}/x] \vee A[\mathsf{false}/x]$ a special status among all formulae holding the same information. Every formula equivalent to it, every formula that entails the same formulae is an equally valid result of forgetting.

The definition captures this parity among formulae by not defining forgetting as a single specific formula and then delegating the definition of its alternatives to equivalence. If B expresses forgetting some variables from A and B' is equivalent to B and contains the same variables, then B' also expresses forgetting the same variables from A. This is because equivalence implies equality of consequences.

3.1 Equivalent conditions

Forgetting could be based on consistency rather than inference. Or it could be based on models. It would still be the same, at least in propositional logic because of the way inference, consistency and models are related.

The following lemma formally proves that forgetting defined in terms of mutual consistency is the same as that defined in terms of inference.

Lemma 2 *A formula B over the variables Y expresses forgetting all variables from A but Y if and only if $A \wedge D$ is equisatisfiable with $A \wedge D$ for all formulae D over variables Y.*

Proof. The definition of B expressing forgetting is that it is built over the variables Y and that $A \models C$ is the same as $B \models C$ for every formula C on the alphabet Y. The two entailments are respectively the same as the inconsistency of $A \wedge \neg C$ and $B \wedge \neg C$. They coincide if and only if $A \wedge D$ and $B \wedge D$ are equisatisfiable, where $D = \neg C$. In the other way around, $A \wedge D$ and $B \wedge D$ are equisatisfiable if and only if $A \wedge \neg C$ and $B \wedge \neg C$ are, where $C = \neg D$. $\square$

The condition mostly used in this article is the restriction of mutual consistency to sets of literals. Every formula is equivalent to a DNF over the same variables, and also to a DNF where every term contains all variables of the formula. The consistency of $A \wedge D$ is the same as the consistency of A with every term, and the same for B. The following two corollaries follow.

Corollary 1 *A formula B over the variables Y expresses forgetting all variables but Y from A if and only if $S \cup A$ is equisatisfiable with $S \cup B$ for all sets of literals S over variables Y.*

Corollary 2 *A formula B over the variables Y expresses forgetting all variables but Y from A if and only if $S \cup A$ is equisatisfiable with $S \cup B$ for all sets of literals S over variables Y that contain all variables in Y.*

3.2 How to forget

Three properties related to computing forgetting are proved: it can be performed one variable at time, it can be performed by resolution, and it may be performed on the independent parts of the formula, if any.

The first property is an almost direct consequence of the definition.

Lemma 3 *If B expresses forgetting the variables Y from A and C expresses forgetting the variables Z from B, then C expresses forgetting $Y \cup Z$ from A.*

Proof. Since B expresses forgetting Y from A it is build over the variables $Var(A)\backslash Y$. Since C expresses forgetting Z from B it is build over the variables $Var(B)\backslash Z$, which is a subset of $Var(A)\backslash Y\backslash Z = Var(A)\backslash(Y \cup Z)$. This is the first condition for C expressing forgetting $Y \cup Z$ from A.

Let D be a formula over the variables $Var(A)\backslash(Y \cup Z)$. This alphabet is also $Var(A)\backslash Y\backslash Z$, which emphasizes that the variables of D are a subset of $Var(A)\backslash Y$. The assumption that B expresses forgetting Y from A implies that $A \models D$ is equivalent to $B \models D$ since the variables of D are all in $Var(A)\backslash Y$. The assumption that C expresses forgetting Z from B implies that $B \models D$ is the same as $C \models D$ since $Var(B) = Var(A)\backslash Y$ and the alphabet of D is $Var(A)\backslash Y\backslash Z = Var(B)\backslash Z$. The equivalence of $A \models D$ with $B \models D$ and the equivalence of $B \models D$ with $C \models D$ implies that $A \models D$ is the same as $C \models D$. This holds for every formula D over the variables of $A\backslash(Y \cup Z)$, and defines C expressing forgetting all variables of $Y \cup Z$ from A. $\square$

Forget is expressed by $A[\mathsf{true}/x] \vee A[\mathsf{false}/x]$, but this formula does not maintain the syntactic form of A: a CNF like $a \wedge (x \vee b \vee c)$ becomes the non-CNF $((a) \vee (a \wedge (b \vee c))$. While this formula can be turned into CNF, directly combining clauses is more convenient when working on CNFs.

Forgetting can be performed by resolution, as proved by Delgrande and Wassermann [10] in the Horn case and extended to the general case by Delgrande [8]. The function resolve(A, B) provided by Definition 4 gives the clauses obtained by resolving each clause of A with each clause of B, if they resolve. The notation $A \cap l$ introduced in definition 2 gives the clauses of A that contain the literal l.

Theorem 1 ([8, Theorem 6]) *The formula* $A \backslash (A \cap x) \backslash (A \cap \neg x) \cup \mathit{resolve}(A \cap x, A \cap \neg x)$ *expresses forgetting x from A.*

Forgetting a single variable is not a limitation because Lemma 3 tells that forgetting a set of variables can be performed one at time: forgetting x first and $Y \backslash \{x\}$ then is the same as forgetting Y.

The problem is that forgetting this way may produce non-minimal formulae even from minimal ones. For example, $A = \{a \vee b \vee x, \neg x \vee c, a \vee c\}$ is minimal, but resolving x out to forget it produces $\{a \vee b \vee c, a \vee c\}$, which is not minimal since the first clause is entailed by the second. That A is minimal is shown by the `minimize.py` program with the first formula in the `outresolve.py` file. The other formulae in that file show similar examples where the formula obtained by resolving out a variable contains a redundant literal or is irredundant although not minimal.

When a formula comprises two independent parts with no shared variable, forgetting from the formula is the same as forgetting from the two parts separately. This property is used in the following hardness proofs that merge two polynomial-time reductions.

Lemma 4 *A formula C expresses forgetting the variables Y from A and D expresses forgetting the variables Y from B if and only if $C \cup D$ expresses forgetting the variables Y from $A \cup B$, if A and B are built over disjoint alphabets.*

Proof. The claim is first proved when Y comprises a single variable x. By Theorem 1, forgetting x from $A \cup B$ is expressed by the following formula.

$$(A \cup B) \backslash ((A \cup B) \cap x) \backslash ((A \cup B) \cap \neg x) \cup \text{resolve}((A \cup B) \cap x) \cup ((A \cup B) \cap \neg x))$$

By definition $(A \cup B) \cap x$ is the set of clauses of $A \cup B$ that contain x. Therefore, it is the union of such clauses of A and of B. In formulae, $(A \cup B) \cap x = (A \cap x) \cup (B \cap x)$. Applying this equality to the formula expressing forgetting turns into the following.

$$(A \cup B) \backslash ((A \cap x) \cup (B \cap x)) \backslash ((A \cap \neg x) \cup (B \cap \neg x)) \cup \text{resolve}((A \cap x) \cup (B \cap x)) \cup ((A \cap \neg x) \cup (B \cap \neg x))$$

Since the variables of A and B are disjoint by assumption, the variable x either is in A or in B, but not in both. Only the first case is considered: x is in A and not in B; the other case is analogous. Since x does not occur in B, this formula contains no clause including x and none including $\neg x$. In formulae, $B \cap x = \emptyset$ and $B \cap \neg x = \emptyset$. These identities simplify the formula expressing forgetting to the following.

$$(A \cup B) \backslash (A \cap x) \backslash (A \cap \neg x) \cup \text{resolve}((A \cap x) \cup (A \cap \neg x))$$

Since $A \cap x$ and $A \cap \neg x$ only contain clauses of A and none of B, the set differences can be applied to A only.

$$B \cup A \backslash (A \cap x) \backslash (A \cap \neg x) \cup \text{resolve}((A \cap x) \cup (A \cap \neg x))$$

This is the union of B and $A \backslash (A \cap x) \backslash (A \cap \neg x) \cup \text{resolve}((A \cap x) \cup (A \cap \neg x))$. The second part expresses forgetting x from A by Theorem 1. Since x is not in B, this formula B it expresses forgetting x from B. This proves that forgetting a single variable x from $A \cup B$ is expressed by the union of formula expressing forgetting it from A and from B.

The claim is proved by iterating over the variables to forget. Let A and B be two formulae built over disjoint alphabets and Y the variables to forget. If Y comprises a single variable, the claim that forgetting can be done separately from A and from B is proved above. Otherwise, Lemma 3 proves that forgetting Y is the same as forgetting an arbitrary variable x of Y first and then $Y \backslash \{x\}$. What proved above is that forgetting x can be done by forgetting it separately from A and B. This is a proof by induction over the number of the variables to forget, because it follows on Y from being proven on $Y \backslash \{x\}$. □

3.3 Necessary literals

Finding a minimal version of a formula is difficult [30, 39, 17]. Finding a minimal formula expressing forgetting is further complicated by the addition of forgetting. While forgetting can be done by substitution ($A[\mathsf{true}/x] \vee A[\mathsf{false}/x]$) or by resolution ($A \backslash (A \cap x) \backslash (A \cap \neg x) \cup \text{resolve}(A \cap x, A \cap \neg x)$), the first produces a non-CNF formula and the second may increase size exponentially with the number of variables to be forgotten.

Finding the exact complexity of this problem proved difficult; not so much for membership to classes in the polynomial hierarchy but for hardness. Fortunately, proving hardness does not require finding the minimal size of arbitrary formulae, just for the formulae that are targets of the reduction. NP-hardness is for example proved by a reduction from SAT to a formula to be forgotten some variables; only for such formulae, the ones generated by the reduction, the minimal size after forgetting is necessary.

This is good news, because reductions do not generate all possible formulae. Rather the opposite: they usually produce formulae of a very specific form. Still better, a reduction can be altered to simplify computing the minimal size of the formulae it produces. If the minimal size is difficult to assess for the formulae produced by a reduction, the reduction itself can be changed to simplify them.

The reductions used in this article rely on two tricks to allow for simple proofs. The first is that some clauses of the formulae they generate are in all minimal-size equivalent formulae; this part of the minimal size is therefore always the same. The second is that the rest of the minimal size is due to the presence or absence of certain literals in all formulae expressing forgetting; where these literals occur if present does not matter, only whether they are present or not.

The first trick is based on superredundancy, defined in the next section.

The second requires proving that a literal is contained in all formulae that express forgetting. This is preliminarily proved when no forgetting is involved. When the next lemmas say "A contains l" they mean: A contains a clause that contains the literal l.

Lemma 5 *If S is a set of literals such that $S \cup A$ is consistent but $S \backslash \{l\} \cup \{\neg l\} \cup A$ is not, the CNF formula A contains l.*

Proof. Since $S \cup A$ is consistent, it has a model M.

The claim is that A contains l. This is proved by contradiction, assuming that no clause of A contains l. By construction $S \backslash \{l\}$ does not contain l either. As a result, $A' = S \backslash \{l\} \cup A$ does not contain l. It is still satisfied by M because M satisfies its superset $S \cup A$. Let M' be the model that sets l to false and all other variables the same as M. Let $l_1 \vee \cdots \vee l_m$ be an arbitrary clause of A'. Since M satisfies A, it satisfies at least one of these literals l_i. Since A does not contain l, this literal l_i is either $\neg l$ or a literal over a different variable. In the first case M' satisfies $l_i = \neg l$ because it sets l to false; in the second because it sets l_i the same as M, which satisfies l_i. This happens for all clauses of A', proving that M' satisfies A'.

Since M' also satisfies $\neg l$ because it sets l to false, it satisfies $A' \cup \{\neg l\} = S \backslash \{l\} \cup \{\neg l\} \cup A$, contrary to its assumed unsatisfiability. □

Since consistency with S and with $S \backslash \{l\} \cup \{\neg l\}$ are unaffected by syntactic changes, they are the same for all formulae equivalent to A. In other words, if the conditions of the lemma hold for A they also hold for every formula equivalent to A.

This property carries to formulae expressing forgetting by constraining S to be only contain variables not to be forgotten.

Lemma 6 *If S is a set of literals over the variables Y such that $S \cup A$ is consistent but $S \backslash \{l\} \cup \{\neg l\} \cup A$ is not, every CNF formula that expresses forgetting all variables but Y from A contains l.*

Proof. Let B be a formula expressing forgetting all variables from A but Y. By Corollary 1, since S is a set of literals over Y the consistency of $S \cup A$ equates that of $S \cup B$. The same holds for $S \backslash \{l\} \cup \{\neg l\}$ since its variables are the same.

The lemma assumes the consistency of $S \cup A$ and the inconsistency of $S \backslash \{l\} \cup \{\neg l\}$. They imply the consistency of $S \cup B$ and the inconsistency of $S \backslash \{l\} \cup \{\neg l\} \cup B$. These two conditions imply that B contains l by Lemma 5. □

How is this lemma used? To prove that reductions from a problem to the problem of minimal size of forgetting work. Not all reductions can be proved correct this way. The ones used in this article are built to allow that. They generate a formula that contains a certain literal l that may or may not meet the condition of the lemma. Depending on this, l may or may not be necessary after forgetting. This is a $+0$ or a $+1$ in the size of the minimal formulae expressing forgetting. If the other literals occurrences are k, the minimal size is $k + 0$ or $k + 1$ depending on whether the conditions of Lemma 6 are met.

In order for this to work, the $+0/+1$ separation is not enough. Equally important to the $k + 0$ vs. $k + 1$ size is that the other addend k stays the same. This is the number of the other literal occurrences. The formulae produced by the reduction may or may not contain a literal l, but this is useless if the rest of the formula changes, if for example the total size is either $k + 0$ or $k' + 1$ where k' is less than k.

Lemma 6 only predicates about the presence of l in a formula, but this tells its overall size only when the rest of the formula has a fixed form. This is ensured by the concept of superirredundancy, which requires a separate section because of the length of its analysis.

3.4 Size of forgetting

Many formulae B express forgetting the same variables X from a formula A. Some may be large and some small. Producing an artificially large formula is straightforward: if $\{a \vee b, b \vee c\}$ expresses forgetting, also $\{a \vee b, b \vee c, \neg a \vee a, a \vee b \vee \neg c\}$ does: adding tautologies and consequences does not change the semantics of a formula. The question is not whether a large expression of forgetting exists.

The question is whether a small expression of forgetting exists. In this context, "small" means "of polynomial size". Technically: given a formula A and a set of variables X, does any formula of polynomial size express forgetting X from A?

This is not the case for all formulae. That would have unlikely consequences on the complexity hierarchy [23] Yet, it is the case for some formulae. It depends on the formula. For example, all negation-free CNF formulae can be forgotten variables by removing the clauses that contain them. The question is whether expressing forgetting can be done in small space for a specific formula. This will be proved D^p-hard and in Σ_2^p for Horn formulae, and D^p2-hard and in Σ_3^p for unrestricted CNF formulae.

The existing literature provide mechanisms for forgetting variables from a formula and results about the minimal size of expressing forgetting for all formulae. They leave open the question in between: the minimal size of expressing forgetting for a formula. An example result of the first kind is: "Salient features of the solution provided include linear time complexity, and linear size of the output of (iterated) forgetting" [1]: given a formula, the size of forgetting is linear. An example result of the second kind is: "the size of the result of forgetting may be exponentially large in the size of the input program" [8]: forgetting may produce exponentially large formulae when considering all possible input formulae. It may, not must. For some formulae, forgetting may not increase size. The only hardness result about this problem has been published by Zhou [47]; it is discussed in a following section. Other authors reported worst-case results [14, 13], and some the opposite, as certain forgetting mechanisms of certain logics can be expressed in polynomial size [19].

Forgetting is also called variable elimination, especially in the context of automated reasoning [38]. It is also identified by its dual concept of uniform interpolation, especially in first-order, modal and description logics [4]. While forgetting is always expressible in exponential space in propositional logics, uniform interpolants in other logics may be larger, if they exists at all. For example, their size is at least triple-exponential in certain description logics, provided that they exist [33]. Analogous to the question of checking their size is checking their existence [2].

4 Superredundancy

The previous section ends with a cliffhanger: a formula may or may not contain a literal depending on a certain condition, but unless the rest of the formula is fixed its overall size may not be controlled by the condition.

Controlling size is required for a reduction. For example, a reduction from propositional satisfiability to the problem of size of forgetting is not just something that takes a formula F and produces a formula A and some variables to forget. It is a reduction only if the satisfiability of F commands the space needed to express forgetting the variables from A: if F is satisfiable forgetting is expressed by a formula of size k or less; if F is unsatisfiable forgetting it is not expressed by any formula of size k or less. Otherwise, this is not a reduction.

Lemma 6 tells how to make size jump over the k boundary: building A so that it activates the lemma if F is unsatisfiable. Unsatisfiability pushes a literal l into every formula expressing forgetting, a $+1$ in size. This is however not enough. It may work or not depending on the rest of the formula.

An example where it works is when a satisfiable F translates into $\{a \vee \neg b, a \vee c\}$ and an unsatifiable F into $\{a \vee \neg b, a \vee c, l\}$. The first formula has size 4, the second $4 + 1$. The presence of l provides the required increase in size.

An example where it does not work is when a satisfiable F translates into $\{a \vee \neg b, a \vee c\}$ and an unsatifiable F into $\{\neg a \vee c, l\}$. Lemma 6 may still force l into the second formula, but size changes from 4 to $2 + 1$. The $+1$ is the second formula is overcome by the decrease of size of the other clauses from 4 to 2.

The mechanism of having or not having l in the formula only works if the rest of the formula is fixed. Otherwise, the presence of l may invalidate the construction like the last addition to an house of cards crushes its lower layers: the unsatisfiability of F may force l into the formula, but if it also allows removing a clause of two literals the total size change is -1, not the required $+1$.

The lower layers, the other clauses, are fixed in place by superirredundancy. Its formal definition is below, but what counts is that a superirredundant clause of a formula belongs in every minimal equivalent formula. The number of its literals is never subtracted from the overall size of the formula. The literals of the superirredundant clauses are fixed; the other clauses may provide the required $+1$ addition in size.

Another use of superirredundancy is proving the formula A minimal. The reason for this requirement is detailed in the next section. How it is meet in this: by having all its clauses superirredundant. If A contains only superirredundant clauses every minimal-size equivalent formula contains them as well; therefore, it is a superset of A; if it contains other clauses it would not be minimal; as a result, it does not contain other clauses: it coincides with A. This argument is made the formal proof of a following lemma, but it already shows that superirredundancy of all clauses implies minimality.

Superirredundancy is based on resolution. Summarizing the notation introduced in Section 2: $\text{resolve}(F, G)$ are the clauses obtained by resolving each clause of the first formula with each of the second; $F \vdash G$ means that the clauses of the second formula are obtained by repeatedly resolving the clauses of the first; the set of all clauses obtained by repeatedly resolving the clauses of a formula F is denoted $\text{ResCn}(F)$.

Resolution does not just tell whether a clause is implied from a set. It also tells why: the implied clause is consequence of other two, in turns consequence of others and so on. This structure is necessary in many proofs below, and is the very reason why they employ resolution $\vdash$ instead of entailment $\models$.

A clause c of F may or may not be redundant in $\text{ResCn}(F)$. It is redundant if it is a consequence of $\text{ResCn}(F) \backslash \{c\}$, like in Figure 1.

An example is a in $F = \{a, \neg a \vee b, \neg b \vee a\}$. The resolution closure of this formula is $\text{ResCn}(F) = \{a, b, \neg a \vee b, \neg b \vee a\}$, where a is redundant since it is entailed by b and $\neg b \vee a$.

Such a redundancy is not always the case. For example, the resolution closure of $F = \{a, b\}$ is itself, since its two clauses do not resolve. As a result, a is not redundant in $\text{ResCn}(F) = \{a, b\}$.

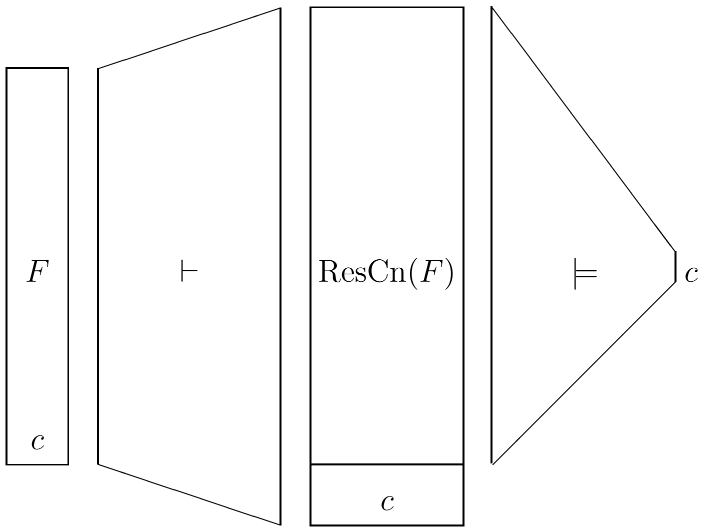


Figure 1: A superredundant clause

In the first case, when c is redundant in the resolution closure, $\mathrm{ResCn}(F)\backslash\{c\}$ is equivalent to $\mathrm{ResCn}(F)$. Even if c is not redundant in F, it is not truly necessary as a formula $\mathrm{ResCn}(F)\backslash\{c\}$ not containing it is equivalent to F. In the first example, a is not redundant in $F = \{a, \neg a \vee b, \neg b \vee a\}$, yet F is equivalent to $\mathrm{ResCn}(F)\backslash\{a\} = \{b, \neg a \vee b, \neg b \vee a\}$, which does not contain the clause a.

This is a weak version of redundancy: while c may not be removed from F, it can be replaced by other consequences of F. The converse is therefore a strong version of minimality: c cannot be removed even adding all other resolution consequences of F. This is why it is called superirredundant. A superirredundant clause cannot be removed even adding resolution consequences in its place. It is irredundant even expanding F this way, even switching to such supersets of F.

The opposite to superirredundancy is unsurprisingly called superredundancy: a clause is superredundant if it is redundant in the superset $\mathrm{ResCn}(F)$ of F. Such a clause can be replaced by other resolution consequences of F.

Definition 6 *A clause c of a formula F is superredundant if it is redundant in the resolution closure of the formula: $ResCn(F)\backslash\{c\} \models c$. It is superirredundant if it is not superredundant.*

A superirredundant clause of a formula will be proved to be in all minimal formulae equivalent to that formula. It is necessary in them. This is how fixing a part of the formula is achieved: by ensuring that its clauses are superirredundant. The opposite concept of superredundancy is introduced because it simplifies a number of technical results.

Contrary to what it may look, creating superirredundant clauses is not difficult.

A typical situation with a superredundant clause c is that F has some minimal equivalent formulae like D which contains c but also others A, B and C which do not, like in Figure 2. This is possible because of $\mathrm{ResCn}(F)\backslash\{c\} \models c$, which allows some subsets of $\mathrm{ResCn}(F)\backslash\{c\}$ like D to entail c. Some other subsets like A, B and C may still be minimal even if they contain c.

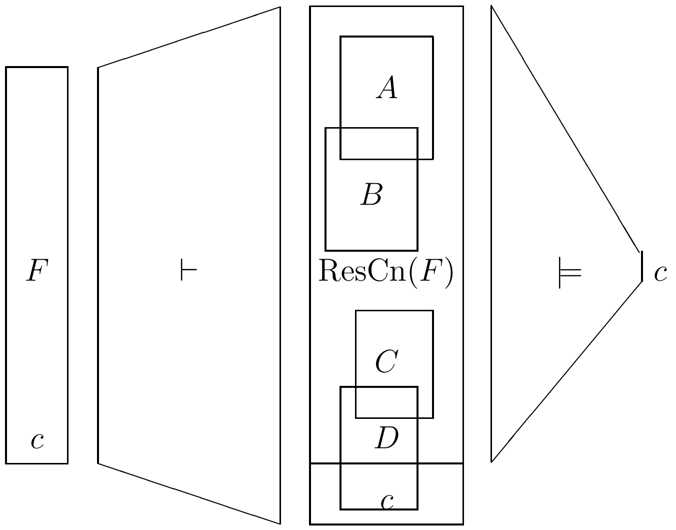


Figure 2: A superredundant clause and some minimal equivalent formulae

Superredundancy is the same as redundancy in the resolution closure. It is not the same as redundancy in the deductive closure, which is always the case unless the clause contains all variables in the alphabet. Otherwise, if a is a variable not in c, then $c \models c \vee a$ and $c \models c \vee \neg a$; as a result, if $c \in F$ then $Cn(F) \backslash \{c\}$ contains $c \vee a$ and $c \vee \neg a$, which imply c. The same argument does not apply to resolution because neither $c \vee a$ nor $c \vee \neg a$ follow from c by resolution. This is why superredundancy is defined in terms of resolution and not entailment.

Redundancy implies superredundancy: if c follows from $F \backslash \{c\}$ it also follows from $\text{ResCn}(F) \backslash \{c\}$ by monotonicity of entailment. Not the other way around. For example, a is irredundant in $F = \{a, \neg a \vee b, \neg b \vee a\}$ but is superredundant: a and $\neg a \vee b$ resolve to b, which resolves with $\neg b \vee a$ back to a. This formula is in the `irredundantsuperredundant.py` file of `minimize.py`.

The formula in this example is not minimal, as it is equivalent to $\{a, b\}$. It shows that a non-minimal formula may contain a superredundant clause. This will be proved to be always the case. The main usage of superredundancy is exactly this one, but in reverse: a formula entirely made of superirredundant clauses is minimal.

Proving superirredundancy of all clauses of a formula is easier than proving minimality since clauses can be checked individually. Yet, not all minimal formulae are superirredundant. This limits the usefulness of superredundancy: it is not the right tool for checking minimality of an arbitrary formula; it is for building a formula that is required to be minimal. Hardness proofs are an example. They can be designed to only produce superirredundant clauses, making the generated formulae minimal.

Lemma 7 *If c is a superirredundant clause of F, it is contained in every minimal CNF formula equivalent to F.*

Proof. Let B be a minimal formula that is equivalent to F. By Property 1, $B \subseteq \text{ResCn}(F)$. If B does not contain c, this containment strengthens to $B \subseteq \text{ResCn}(F) \backslash \{c\}$. A consequence of the equivalence between B and F is that B implies every clause of F, including c. Since

$B \models c$ and $B \subseteq \text{ResCn}(F) \backslash \{c\}$, by monotonicity $\text{ResCn}(F) \backslash \{c\} \models c$ follows. This is the opposite of the assumed superirredundancy of c. □

This lemma provides a sufficient condition for a clause being in all minimal formulae equivalent to the given one. Not a necessary one, though. A clause that is not superirredundant may still be in all minimal formulae. A counterexample only requires three clauses.

Counterexample 1 *The first clause of* $F = \{a, \neg a \vee b, a \vee \neg b\}$ *is superredundant but is in all minimal formulae equivalent to* F*.*

Proof. The formula is in the `minimize.py` test file `allminimal.py`. Its first clause is a. It resolves with $\neg a \vee b$ into b. The resolution closure of F contains a, b and $\neg b \vee a$. The first is redundant as it is entailed by the second and the third. This proves it superredundant.

Yet, the only minimal formula equivalent to F is $F' = \{a, b\}$, which contains a in spite of its superredundancy. That $\{a, b\}$ is minimal is proved by the superirredundancy of its clauses: its resolution closure is $\{a, b\}$ itself since its clauses do not resolve; none of the two clauses is redundant in it. □

The proof of this counterexample relies on the syntactic dependency of superredundancy: a is superredundant in $\{a, \neg a \vee b, a \vee \neg b\}$ but superirredundant in its minimal equivalent formula $\{a, b\}$.

Lemma 8 *There exists two equivalent formulae that both contain a clause that is superredundant in one but not in the other.*

Proof. The formulae are $F = \{a, \neg a \vee b, a \vee \neg b\}$ and $F' = \{a, b\}$.

The resolution closure of the first is $\text{ResCn}(F) = \{a, b, \neg a \vee b, a \vee \neg b\}$ since the only possible resolutions in F are $a, \neg a \vee b \vdash b$ and $b, a \vee \neg b \vdash a$. While $\neg a \vee b$ and $a \vee \neg b$ have opposite literals, they would only resolve into a tautology. Since the clauses of F' do not resolve, its resolution closure is F' itself: $\text{ResCn}(F') = \{a, b\}$.

The clause a is redundant in $\text{ResCn}(F) = \{a, b, \neg a \vee b, a \vee \neg b\}$ since it is entailed by b and $a \vee \neg b$. It is not redudant in $\text{ResCn}(F') = \{a, b\}$ since it is not entailed by b. □

Being dependent on the syntax of the formula, superredundancy and superirredundancy are not the same as any condition that is independent from the syntax, such as:

- redundancy in the set of prime implicates, which is also employed in formula minimization [20];
- essentiality of prime implicates, defined as containment of a prime implicate in all prime CNFs equivalent to the formula [20]
- presence in all minimal CNF formulae equivalent to F.

These conditions are independent on the syntax since the prime implicantes, the prime equivalent CNFs and the minimal equivalent formulae are the same for two equivalent formulae. Being independent on the syntax, they are not the same as superredundancy or superirredundancy, which are proved dependent on the syntax by Lemma 8.

Lemma 7 does not contradict the inequality of superirredundancy and presence in all minimal equivalent formulae. It only proves that a superirredundant clause of a formula is in all minimal formulae equivalent to it. Not the other way around. A clause may be in all minimal formulae while not being superirredundant.

Lemma 9 *There exists a formula that contains a superredundant clause that is in all its minimal CNF equivalent formulae.*

Proof. Let c be a clause that is superredundant in F and superirredundant in F' with $F \equiv F'$. Such a condition is possible according to Lemma 8.

Since c is superirredundant in F', it is contained in all minimal formulae equivalent to F'. Since this formula is equivalent to F, their minimal equivalent formulae are the same. □

Superirredundancy in a formula is a strictly stronger condition than membership in all its minimal equivalent formulae. It implies that, but is not the same.

This is to be kept in mind when superirredundancy is used as a precondition. Some lemmas below prove that if a clause is superirredundant in a formula then it is superirredundant in a similar formula. The precondition of such a result is the superirredundancy of the clause; its membership in all minimal equivalent formulae may not be enough. Superirredundancy is required. Membership to all minimal equivalent formulae is a consequence of both the premise and the conclusion, not a substitute the premise.

The typical use of superirredundancy is to build a formula that is guaranteed to be minimal.

Lemma 10 *If a formula contains only superirredundant clauses, it is minimal.*

Proof. Let B be a minimal formula equivalent to A. By Lemma 7, the superirredundant clauses of A are in all minimal formulae that are equivalent to A. Therefore, B contains all of them. If A only comprises superirredundant clauses, B contains all of them: $A \subseteq B$. The only case where B could not be the same as A is when this containment is strict, but that would imply that B is not minimal since A is equivalent but smaller. □

If a formula is built so that all its clauses are superirredundant, it is guaranteed to be minimal. Not the other way around. Rather the opposite: a minimal formula may be made only of superredundant clauses. An example is $\{\neg a \vee b, \neg b \vee c, \neg c \vee a\}$. Resolving the first two clauses $\neg a \vee b$ and $\neg b \vee c$ generates $\neg a \vee c$. Resolving the other pairs produces the opposite cycle of clauses $\{\neg a \vee c, \neg c \vee b, \neg b \vee a\}$, which is equivalent to the original. The clauses of the original are therefore entailed by some of their resolution consequences. Yet, the original is minimal; proving it minimal is long but tedious, which is why the proof is delegated to the file `superredundantminimal.py` of `minimize.py`.

Superirredundancy differs from minimality, but is easier than that to ensure. Simplicity is its motivation. Therefore, it makes sense to simplify it further. The following lemma gives a number of equivalent conditions, all based on F deriving another formula G that in turn derives c. The first equivalent condition is exemplified by Figure 3. Instead of proving that c is a consequence of $\mathrm{ResCn}(F)\backslash\{c\}$, these equivalent conditions allows proving the existence of a possibly smaller formula G that entails c.

Lemma 11 *A clause c of a formula F is superredundant if and only if a formula G satisfying either one of the following conditions exists:*

1. $F \vdash G \models c$ *where* $c \not\in G$
2. $F \vdash G \vdash c'$ *where* $c \not\in G$ *and* $c' \subseteq c$

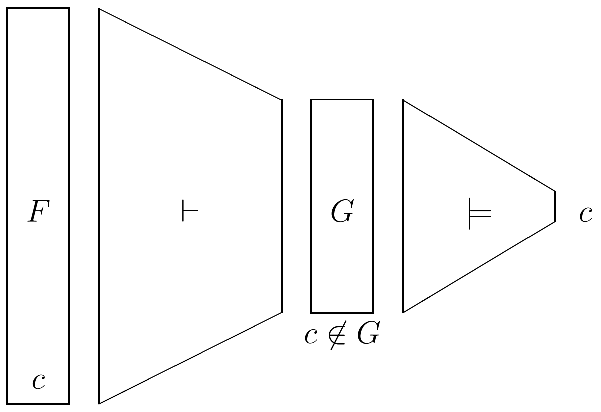


Figure 3: An example of a superredundant clause

3. *$F \vdash G \vdash c$ with $c \notin G$ or $F \vdash c'$ with $c' \subset c$*

4. *$F \vdash G \vdash c$ with $c \notin G$ or $F \models c'$ with $c' \subset c$*

Proof. Equivalence with the first condition is proved in the two directions. A clause c of F is superredundant if and only if $\mathrm{ResCn}(F)\backslash\{c\} \models c$. If this condition is true the claim holds with $G = \mathrm{ResCn}(F)\backslash\{c\}$, since c is not in $\mathrm{ResCn}(F)\backslash\{c\}$ by construction and the definition of resolution closure implies $F \vdash c'$ for every $c' \in \mathrm{ResCn}(F)$. In the other direction, if $F \vdash G$ then $G \subseteq \mathrm{ResCn}(F)$. Since $c \notin G$, this containment strenghtens to $G \subseteq \mathrm{ResCn}(F)\backslash\{c\}$. Since $G \models c$, it follows $\mathrm{ResCn}(F)\backslash\{c\} \models c$ by monotonicity.

The first condition is equivalent to the second because $G \models c$ is the same as $G \vdash c'$ with $c' \subseteq c$.

The second condition is proved equivalent to the third considering the two directions separately. The second condition includes $c' \subseteq c$, which comprises two cases: $c' = c$ and $c' \subset c$. If $c' = c$ the second condition becomes $F \vdash G \vdash c$ with $c \notin G$, the same as the first alternative of the third condition. If $c' \subset c$ the second condition is $F \vdash G \vdash c'$, which implies $F \vdash c'$ with $c' \subset c$; this is the second alternative of the third condition. In the other direction, the first alternative of the third condition is $F \vdash G \vdash c$ with $c \notin G$, which is the same as the second condition with $c' = c$. The second alternative is $F \vdash c'$ with $c' \subset c$; the second condition holds with $G = \{c'\}$.

Equivalence with the fourth condition holds because $F \vdash c'$ implies $F \models c'$, and in the other direction $F \models c'$ implies $F \vdash c''$ with $c'' \subset c$, and the third equivalent condition holds with c'' in place of c'. □

A clause c is superredundant if it follows from F by a resolution proof that contains a set of clauses G sufficient to prove c. As such, G is a sort of "cut" in a resolution tree $F \vdash c$, separating c from F. This cut can be next to the root, next to the leaves, or somewhere in between. In practice, it is useful at the first resolution steps (next to the leaves) or at the last (next to the root).

The next equivalent condition to superredundancy cuts the resolution tree at its very last point, one step short of regenerating c. It comprises two alternatives, depicted in Figure 4. They are due to the two possibilities contemplated by the third condition of Lemma 11: either F implies a proper subset of c or c itself with a resolution proof cut by a set G.

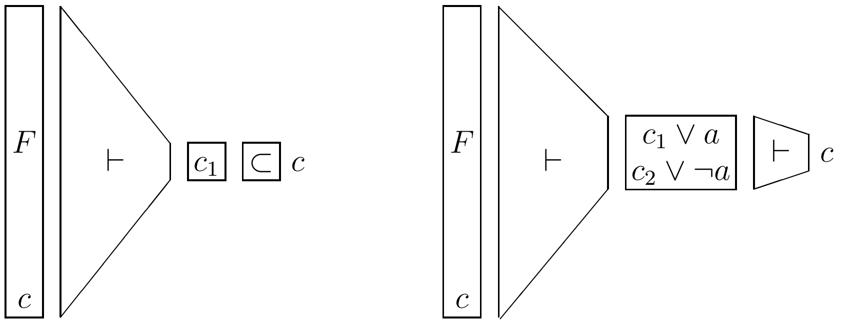


Figure 4: Superredundancy proved by the last step of resolution

Lemma 12 *A clause c of a formula F is superredundant if and only if either $F \vdash c_1$ where $c_1 \subset c$ or $F \vdash c_1 \vee a, c_2 \vee \neg a$ for some variable a not in c and clauses c_1 and c_2 such that $c = c_1 \vee c_2$.*

Proof. By Lemma 11, superredundancy is equivalent to $F \vdash G \vdash c$ with $c \not\in G$ or $F \vdash c'$ with $c' \subset c$.

The second part of this condition is the same as $F \vdash c_1$ and $c_1 \subset c$ with $c_1 = c'$, the first alternative in the statement of the lemma.

The first part $F \vdash G \vdash c$ with $c \not\in G$ is now proved to be the same as the second alternative in the statement of the lemma: $F \vdash c_1 \vee a, c_2 \vee \neg a$ where $a \not\in c$ and $c = c_1 \vee c_2$.

If $F \vdash G \vdash c$ with $c \not\in G$, since c is not in G the derivation $G \vdash c$ contains at least a resolution step. Let c' and c'' be the two clauses that resolve to c in this derivation. Since they resolve to c, they have the form $c' = c_1 \vee a$ and $c'' = c_2 \vee \neg a$ for some variable a. Their resolution $c = c_1 \vee c_2$ does not contain a by Lemma 1. These two clauses are obtained by resolution from G. Since $F \vdash G$, they also derive by resolution from F.

In the other direction, $F \vdash c_1 \vee a, c_2 \vee \neg a$ implies superredundancy with $G = \{c_1 \vee a, c_2 \vee \neg a\}$, since $G \vdash c$ and $c \not\in G$ since c does not contain a by assumption while the two clauses of G both do. □

This lemma tells that looking at all possible sets of clauses G when checking superredundancy is a waste of time. The sets comprising pairs of clauses containing an opposite literal suffice. Their form provides an even further simplification: they are obtained by splitting the clause in two and adding an opposite literal to each. Superredundancy is the same as resolution deriving either such a pair or a proper subset of the clause.

A clause is proved superredundant by such a splitting. Yet, proving superredundancy is not much useful. Proving minimality is the goal. Minimality follows from superirredundancy, not superredundancy. It is in this task that the lemma is useful. Instead of checking all possible sets of clauses G, it allows concentrating only on the pairs obtained by splitting the clause.

When the clauses of F do not resolve, the second alternative offered by Lemma 12 never materializes: a clause is superredundant if and only if it is a superset of a clause of F. This quite trivial specialization looks pointless, but turns essential when paired to the subsequent Lemma 17.

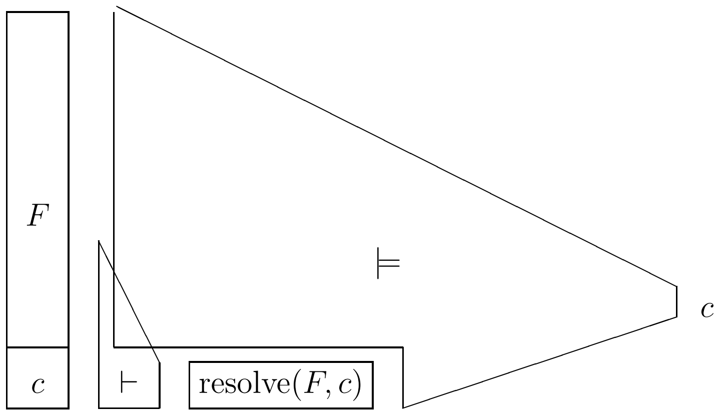


Figure 5: Superredundancy proved by immediate resolution consequences

Lemma 13 *If no two clauses of F resolve then a clause of F is superredundant if and only if F contains a clause that is a strict subset of it.*

Proof. By Lemma 12, $c \in F$ is superredundant if and only if $F \vdash c_1$ with $c_1 \subset c$ or $F \vdash c_1 \vee a, c_2 \vee \neg a$ with $c = c_1 \vee c_2$. The second condition implies $c_1 \vee a, c_2 \vee \neg a \in F$ since the clauses of F do not resolve; this contradicts the assumption since these two clauses resolve. As a result, the only actual possibility is the first: $F \vdash c_1$ with $c_1 \subset c$. Since the clauses of F do not resolve, c_1 cannot be the result of resolving clauses. Therefore, it is in F. □

Formula G of Lemma 11 can be seen as a cut in a resolution tree from F to c. It separates all occurrences of $c \in F$ in the leaves from c in the root. Lemma 12 places the cut next to the root. The following places it next to the leaves. It proves that resolving c with clauses of F only is enough. The clauses obtained by these resolutions are resolve(c, F), according to Definition 4. This situation is shown by Figure 5.

Lemma 14 *A clause c of F is superredundant if and only if $F \backslash \{c\} \cup resolve(c, F) \models c$.*

Proof. The first equivalent condition to superredundancy offered by Lemma 11 is the existence of a set G such that $F \vdash G$, $G \models c$ and $c \not\in G$. The proof is composed of two parts: the first is that $F \backslash \{c\} \cup$ resolve(c, F) is such a set G if it entails c; the second is that if such a set G exists, the derivation of $F \vdash G$ can be rearranged so that c resolves only with other clauses of F. The first is almost trivial, the second is not because c may resolve with clauses obtained by resolution in $F \vdash G$. The rearranged derivation begins with a batch of resolutions of c with other clauses of F, and c is then no longer used. The resolvents of these first resolutions and the rest of F makes the required set G.

If $F \backslash \{c\} \cup$ resolve$(c, F) \models c$ then $G = F \backslash \{c\} \cup$ resolve(c, F) proves c superredundant: $F \vdash G$, $G \models c$ and $c \not\in G$. The first condition $F \vdash G$ holds because the only clauses of G that are not in F are the result of resolving $c \in F$ with a clause of F; the second condition $G \models c$ holds by assumption; the third condition is that G does not contain c, and it holds because G is the union of $F \backslash \{c\}$ and resolve(c, F), where $F \backslash \{c\}$ does not contain c by construction and resolve(c, F) because resolving a clause does not generate the clause itself by Lemma 1.

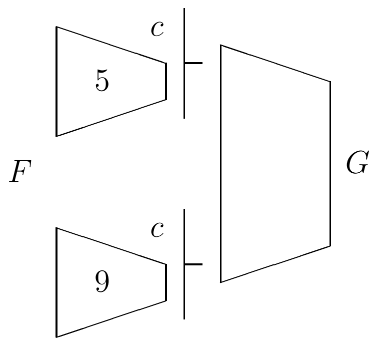


Figure 6: An example of the rise of a clause

The rest of the proof is devoted to proving the converse: $F \vdash G$, $G \models c$ and $c \notin G$ imply $F \backslash \{c\} \cup \text{resolve}(c, F) \models c$.

The claim is proved by repeatedly modifying G until it becomes a subset of $F \backslash \{c\} \cup \text{resolve}(c, F)$ while still maintaining its properties $F \vdash G$, $G \models c$ and $c \notin G$.

This process ends because a measure defined on the derivation $F \vdash G$ decreases until reaching zero. This measure is the almost-size of the derivation $F \vdash G$ plus the rise of c in it. Both are based on the size of the subtrees of $F \vdash G$: each clause in the derivation is generated independently of the others, and is therefore the root of its own tree.

The size of the derivation $F \vdash G$ is the number of clauses it contains. Its almost-size is the number of clauses except the roots.

The derivation $F \vdash G$ may contain some resolutions of c with other clauses. The rise of an individual resolution of c with a clause c'' in $F \vdash G$ is the number of nodes in the tree rooted at c'' minus one. The total rise of c in $F \vdash G$ is the sum of all resolutions of c in it. It measures the overall distance of c from the other leaves of the tree, its elevation from the ground. Figure 6 shows an example.

Both the almost-size and the rise of c are not negative. They are sums, each addend being the size of a nonempty tree minus one; since each tree is not empty its size is at least one; the addend is at least zero. Their sums are at least zero.

If G is a subset of $F \cup \text{resolve}(c, F)$, then $c \notin G$ implies it is also a subset of $F \backslash \{c\} \cup \text{resolve}(c, F)$. Since G implies c, also does its superset $F \backslash \{c\} \cup \text{resolve}(c, F)$. This is the claim.

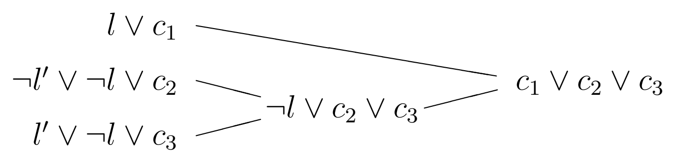
Otherwise, G is not a subset of $F \cup \text{resolve}(c, F)$. This means that G contains a clause $c' \in G$ that is not in F and is not the result of resolving c with a clause of F. Since c' is not in F it is the result of resolving two clauses c'' and c'''. One of them may be c or not; if it is, the other one is not in F and is therefore the result of resolving two other clauses.

If neither c'' nor c''' is equal to c, the modified set $G' = G \backslash \{c'\} \cup \{c'', c'''\}$ has the same properties of G that prove c superredundant: $F \vdash G'$, $G' \models c$ and $c \notin G'$. Let a and b be the size of the trees rooted in c'' and c''' in $F \vdash G$. Since $F \vdash G$ has c' as a root, its almost-size includes $a + b + 1 - 1 = a + b$. Instead, $F \vdash G'$ has c'' and c'' as roots in place of c'; therefore, its almost-size includes $(a - 1) + (b - 1) = a + b - 2$. The almost-size of $F \vdash G'$ is smaller than $F \vdash G$. The rise of c is the same, since the resolutions of c are the same in the two derivations. Summarizing, almost-size decreases while rise maintains its value.

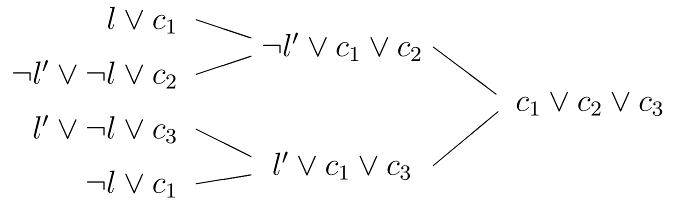
If either c'' or c''' is equal to c, the same set $G' = G \backslash \{c'\} \cup \{c'', c'''\}$ does not work because it does not maintain $c \notin G'$. Since the two cases $c'' = c$ and $c''' = c$ are symmetric, only the second is analyzed: $c' \in G$ is generated by $c, c'' \vdash c'$ in $F \vdash G$. If c'' is in F then c' is in resolve(c, F) because it is the result of resolving c with a clause of F. Otherwise, c'' is a clause obtained by resolving two other clauses.

These two clauses resolve in c'', which resolves with c. Two resolutions, two pairs of opposite literals. Let l be the literal of c that is negated in c'' and l' the literal that is resolved upon in the resolution that generates c''. At least one of the two clauses that generate c'' contains $\neg l$ since c'' does. At least means either one or both.

The first case is that both clauses that resolve in c'' contain $\neg l$. Since they also contain l' and $\neg l'$, they can be written $\neg l' \vee \neg l \vee c_2$ and $l' \vee \neg l \vee c_3$. They resolve in $c'' = \neg l \vee c_2 \vee c_3$. Since c contains l, it can be written $l \vee c_1$. It resolves with $c'' = \neg l \vee c_2 \vee c_3$ to $c' = c_1 \vee c_2 \vee c_3$.


$l \vee c_1$
$\neg l' \vee \neg l \vee c_2$
$l' \vee \neg l \vee c_3$
$\neg l \vee c_2 \vee c_3$
$c_1 \vee c_2 \vee c_3$

A different derivation from the same clauses resolves $l \vee c_1$ with $\neg l' \vee \neg l \vee c_2$, producing $c_1 \vee \neg l' \vee c_2$, and with $l' \vee \neg l \vee c_3$, producing $c_1 \vee l' \vee c_3$. The produced clauses $c_1 \vee \neg l' \vee c_2$ and $c_1 \vee l' \vee c_3$ resolve to $c_1 \vee c_2 \vee c_3$, the same conclusion of the original derivation. This is a valid derivation, with an exception discussed below.


$l \vee c_1$
$\neg l' \vee \neg l \vee c_2$
$l' \vee \neg l \vee c_3$
$\neg l \vee c_1$
$\neg l' \vee c_1 \vee c_2$
$l' \vee c_1 \vee c_3$
$c_1 \vee c_2 \vee c_3$

This derivation proves $F \vdash G'$ where $G' = G \backslash \{c_1 \vee c_2 \vee c_3\} \cup \{c_1 \vee \neg l' \vee c_2, c_1 \vee l' \vee c_3\}$. The only clause of G that G' does not contain is $c_1 \vee c_2 \vee c_3$, which is implied by resolution from its clauses $c_1 \vee \neg l' \vee c_2$ and $c_1 \vee l' \vee c_3$. Therefore, $G' \models G$. This implies $G' \models c$ since $G \models c$. Since $c_1 \vee \neg l' \vee c_2$ is obtained by resolving two clauses over l, it does not contain l by Lemma 1. The same applies to $c_1 \vee l' \vee c_3$. Since c contains l, it is not equal to either of these two clauses. It is not equal to any other clause of G' either, since these are also clauses of G and c is not in G. This proves that G' has the same properties that prove c superredundant: $F \vdash G'$, $G' \models c$ and $c \notin G'$.

Since $\neg l' \vee \neg l \vee c_2$ and $l' \vee \neg l \vee c_3$ are generated by resolution from F, they are the root of resolution trees. Let a and b be their size. The almost-size of the original derivation $F \vdash G$ includes $a + b + 2$, that of $F \vdash G'$ has $a + b + 2$ in its place. Almost-size does not change.

The rise of resolving $c = l \vee c_1$ with $c'' = \neg l \vee c_2 \vee c_3$ in the original derivation is one less the size of the tree rooted in c''. This tree comprises c'' and the trees rooted in $\neg l' \vee \neg l \vee c_2$ and $l' \vee \neg l \vee c_3$. Its size is therefore $a + b + 1$. The rise of c is therefore $a + b$. The two resolutions of $c = l \vee c_1$ in the modified derivation are with $\neg l' \vee \neg l \vee c_2$ and $l' \vee \neg l \vee c_3$. The rise of c in the first is the size of tree rooted in $\neg l' \vee \neg l \vee c_2$ minus one: $a - 1$; the rise of c in the second is $b - 1$. Their sum is $a + b - 2$, which is strictly less than $a + b$.

In summary, switching from $F \vdash G$ to $F \vdash G'$ maintains the almost-size and decreases the rise of c.

The exception mentioned above is that the new resolutions may generate tautologies, which are not allowed. Since $\neg l' \vee \neg l \vee c_2$, $l' \vee \neg l \vee c_3$ and $c_1 \vee c_2 \vee c_3$ are in the original derivation, they are not tautologies. As a result, c_1, c_2 and c_3 do not contain opposite literals, c_2 does not contain l' and c_3 does not contain $\neg l'$. The new clause $c_1 \vee \neg l' \vee c_2$ is tautological only if l' is in c_1, and $c_1 \vee l' \vee c_3$ only if $\neg l'$ is in c_1. By symmetry, only the second case is considered: $c_1 = \neg l' \vee c_1'$.

$$\begin{tikzcd}
l \vee \neg l' \vee c_1' \arrow[drr, dash] & & \\
\neg l' \vee \neg l \vee c_2 \arrow[dr, dash] & & \neg l' \vee c_1' \vee c_2 \vee c_3 \\
 & \neg l \vee c_2 \vee c_3 \arrow[ur, dash] & \\
l' \vee \neg l \vee c_3 \arrow[ur, dash] & &
\end{tikzcd}$$

An alternative derivation resolves $l \vee \neg l' \vee c_1$ with $\neg l' \vee \neg l \vee c_2$, resulting in $\neg l' \vee c_1 \vee c_2$, a subset of the original result $\neg l' \vee c_1 \vee c_2 \vee c_2$.

$$\begin{tikzcd}
l \vee \neg l' \vee c_1' \arrow[dr, dash] & \\
 & \neg l' \vee c_1' \vee c_2 \\
\neg l' \vee \neg l \vee c_2 \arrow[ur, dash] & \\
l' \vee \neg l \vee c_3 &
\end{tikzcd}$$

Since $\neg l' \vee c_1' \vee c_2 \subseteq \neg l' \vee c_1' \vee c_2 \vee c_3$, it holds $\neg l' \vee c_1' \vee c_2 \models \neg l' \vee c_1' \vee c_2 \vee c_3$, which implies $G' \models G$ where $G' = G \backslash \{\neg l' \vee c_1' \vee c_2 \vee c_3\} \cup \{\neg l' \vee c_1' \vee c_2\}$, which in turns implies $G' \models c$ since $G \models c$. This set G' is still obtained by F by resolution. It does not contain c because $c \notin G$ and the added clause $\neg l' \vee c_1' \vee c_2$ is not c. It is not c because it does not contain l while c does, and it does not contain l by Lemma 1 because it is the result of resolving two clauses over l. This proves that G' inherit from G all properties that prove c superredundant: $F \vdash G'$, $G' \models c$ and $c \notin G'$.

If the tree rooted in $\neg l' \vee \neg l \vee c_2$ has size a and the tree rooted in $l' \vee \neg l \vee c_3$ has size b, the derivation $F \vdash G$ includes $a + b + 2$ in its almost-size. The derivation $F \vdash G'$ has $a + 1$ in its place, a decrease in almost-size. The rise of this resolution of c in $F \vdash G$ is $a + b$. In $F \vdash G'$, it is $a - 1$. Both almost-size and rise of c decrease.

The second case is that only one of the clauses that resolve into c'' contains $\neg l$. Their resolution literal is still denoted l'; therefore, they can be written $\neg l' \vee \neg l \vee c_2$ and $l' \vee c_3$. The result of resolving them is $c'' = \neg l \vee c_2 \vee c_3$, which resolves with $c = l \vee c_1$ to generate $c' = c_1 \vee c_2 \vee c_3$.

$$\begin{tikzcd}
l \vee c_1 \arrow[drr, dash] & & \\
\neg l' \vee \neg l \vee c_2 \arrow[dr, dash] & & c_1 \vee c_2 \vee c_3 \\
 & \neg l \vee c_2 \vee c_3 \arrow[ur, dash] & \\
l' \vee c_3 \arrow[ur, dash] & &
\end{tikzcd}$$

Since $l \vee c_1$ and $\neg l' \vee \neg l \vee c_2$ oppose on l, they resolve. The result is $c_1 \vee \neg l' \vee c_2$, which resolves with $l' \vee c_3$ into $c_1 \vee c_2 \vee c_3$. The same three clauses produce the same clause. This is a valid derivation with an exception discussed below.

$$\begin{tikzcd}
l \vee c_1 \arrow[dr, dash] & & \\
 & \neg l' \vee c_1 \vee c_2 \arrow[r, dash] & c_1 \vee c_2 \vee c_3 \\
\neg l' \vee \neg l \vee c_2 \arrow[ur, dash] & & \\
l' \vee c_3 \arrow[uurr, dash] & &
\end{tikzcd}$$

This derivation proves $F \vdash G'$ where $G' = G \backslash \{c_1 \vee c_2 \vee c_3 \cup \{c_1 \vee \neg l' \vee c_2, l' \vee c_3\}$. The only clause of G that G' does not contain is $c_1 \vee c_2 \vee c_3$, but this is the result of resolving $c_1 \vee \neg l' \vee c_2$ and $l' \vee c_3$, two clauses of G'. As a result, $G' \models G$. This proves $G' \models c$ since $G \models c$. Finally, c is not in G'. Since $c \notin G$, suffices to prove that c is not any of the two clauses that G' contains while G does not. This is the case because $c = l \vee c_1$ contains l while the two clauses does not. The original derivation contains $c_1 \vee c_2 \vee c_3$ as the result of resolving two clauses over l; Lemma 1 tells that l is not in $c_1 \vee c_2 \vee c_3$. As a result, l is in $c_1 \vee \neg l' \vee c_2$ or $l' \vee c_3$ only if either $l = \neg l'$ or $l = l'$. That implies that $\neg l \vee c_2 \vee c_3$ contains l' while it is obtained in the original derivation by resolving two clauses over l', contradicting Lemma 1. This proves that G' inherits all properties that prove c superredundant: $F \vdash G'$, $G' \models c$ and $c \notin G'$.

Since $\neg l' \vee \neg l \vee c_2$ and $l' \vee c_3$ are obtained by resolution from F, they are the root of resolution trees. Let a and b be their size.

The almost-size of $F \vdash G$ includes a part for the tree rooted in $c_1 \vee c_2 \vee c_3$; that part is $a + b + 2$. The derivation $F \vdash G'$ is the same except that it has the parts for $c_1 \vee \neg l' \vee c_2$ and $l' \vee c_3$ instead: $a + 1$ and $b - 1$. The almost-size decreases from $a + b + 2$ to $a + b$.

The rise of the resolution of c with $\neg l \vee c_2 \vee c_3$ in $F \vdash G$ is the size of three tree rooted in $\neg l \vee c_2 \vee c_3$; this tree contains $a + b + 1$ nodes; the rise in $F \vdash G$ is therefore $a + b$. The rise of the resolution of c with $\neg l' \vee \neg l \vee c_2$ in $F \vdash G'$ is instead the size of tree rooted in $\neg l' \vee \neg l \vee c_2$ minus one: $a - 1$. This is smaller than $a + b$ since b is nonnegative.

Summarizing, the change in the derivation strictly decreases both its overall size and its rise of c.

The exception that makes the new derivation invalid is when the new clause $c_1 \vee \neg l' \vee c_2$ is a tautology. Valid resolution derivations do not contain tautologies. This also applies to the original one. Since $c_1 \vee c_2 \vee c_3$ is not a tautology, $c_1 \vee c_2$ is neither. Since $\neg l' \vee \neg l \vee c_2$ is not a tautology, c_2 does not contain l'. The new clause $c_1 \vee \neg l' \vee c_2$ is a tautology only if l' is in c_1. Equivalently, $c_1 = l' \vee c'_1$ for some c'_1.

$$\begin{tikzcd}
l \vee l' \vee c'_1 \arrow[drr, dash] & & \\
\neg l' \vee \neg l \vee c_2 \arrow[dr, dash] & & l' \vee c'_1 \vee c_2 \vee c_3 \\
 & \neg l \vee c_2 \vee c_3 \arrow[ur, dash] & \\
l' \vee c_3 \arrow[ur, dash] & &
\end{tikzcd}$$

The root of the derivation is $l' \vee c_1 \vee c_2 \vee c_3$ in this case. This is a superset of its grandchildren $l' \vee c_3$. Since subclauses imply superclauses, $G' = G \backslash \{c_1 \vee c_2 \vee c_3\} \cup \{l' \vee c_3\}$ implies G. By transitivity, it implies c.

The clause $l' \vee c'_1 \vee c_2 \vee c_3$ does not contain l by Lemma 1 because it is the result of resolving two clauses upon l in the original derivation. As a result, its subset $l' \vee c_3$ does not contain l either. It therefore differs from c, which contains l. This implies $c \not\in G'$ since $c \not\in G$ and $c \neq l' \vee c_3$.

Since $F \vdash G$ includes the derivation of $l' \vee c_3$, also $F \vdash G'$ holds.

All three properties of G are inherited by G': $F \vdash G'$, $G' \models c$ and $c \not\in G'$.

Let a and b be the size of the trees rooted at $\neg l' \vee \neg l \vee c_2$ and $l' \vee c_3$. The almost-size of $F \vdash G$ has a component for its root $l' \vee c_1 \vee c_2 \vee c_3$ of value $a + b + 2$. The derivation $F \vdash G'$ has the root $l' \vee c_3$ in its place, contributing only $a - 1$ to the almost-size. The almost-size decreases. The rise of c also decreases. In the original derivation c resolves with $\neg l \vee c_2 \vee c_3$, contributing $a + b$ to the overall rise. In $F \vdash G'$ this resolution is absent, contributing 0 to the overall rise. Since a and b are the size of non-empty trees, they are greater than zero. The rise of c decreases of $a + b$, which is at least 2.

All of this proves that if G is not a subset of $F \cup \mathrm{resolve}(c, F)$ it can be changed to decrease its overall measure while still proving c superredundant. This change preserves $F \vdash G$, $G \models c$ and $c \not\in G$ and strictly decreases the measure of $F \vdash G$, defined as the sum of its almost-size and rise of c.

This change can be iterated as long as G is not a subset of $F \cup \mathrm{resolve}(c, F)$. It terminates because the measure strictly decreases at each step but is never negative as proved above. When it terminates, G is a subset of $F \cup \mathrm{resolve}(c, F)$ because otherwise it could be iterated. Since $c \not\in G$ is one of the preserved properties, G is also a subset of $F \backslash \{c\} \cup \mathrm{resolve}(c, F)$. Since G implies c, its superset $F \backslash \{c\} \cup \mathrm{resolve}(c, F)$ implies c too. This is the claim. □

Lemma 14 allows for a simple algorithm for checking superredundancy: resolve c with all other clauses of F and then remove it. If c is still entailed, it is superredundant. This proves that checking superredundancy is polynomial-time for example in the Horn and Krom case (the latter is that all clauses contain two literals at most). More generally, it is polynomial in all restrictions where inference is polynomial-time and are closed under resolution.

Theorem 2 *Checking superredundancy is polynomial-time in the Horn and Krom case.*

Proof. The clauses in $\mathrm{resolve}(F, c)$ can be generated in polynomial time by resolving each clause of F with c. The result is still Horn or Krom because resolving two Horn clauses or two Krom clauses respectively generates a Horn and Krom clause. Checking $F \backslash \{c\} \cup \mathrm{resolve}(F, c) \models c$ therefore only takes polynomial time. □

If c is a single literal l, it only resolves with clauses containing $\neg l$. The condition is very simple in this case.

Corollary 3 *A single-literal clause l of F is superredundant if and only if*

$$\{c \in F \mid \neg l \not\in c,\ c \neq l\} \cup \{c \mid c \vee \neg l \in F\} \models l$$

Proof. When a clause c comprises a single literal l, Lemma 14 equates its superredundancy to $F \backslash \{l\} \cup \mathrm{resolve}(F, l) \models l$. The first part $F \backslash \{l\}$ of the formula comprises clauses that resolve with l and clauses that do not:

$$F \backslash \{l\} = \{c \in F \mid \neg l \not\in c,\ c \neq l\} \cup \{c \in F \mid \neg l \in c\}$$

If c is in the second set, then resolve(F, l) contains resolve$(c, l) = c\backslash\{\neg l\}$, which implies c. Therefore, $F\backslash\{l\} \cup \text{resolve}(F, l)$ is equivalent to $\{c \in F \mid \neg l \notin c,\ c \neq l\} \cup \text{resolve}(F, l)$. The clauses that resolve with l are all of the form $c \vee \neg l$, and the result of the resolution is c. Therefore, resolve(F, l) can be rewritten as $\{c \mid c \vee \neg l \in F\}$. This proves the claim. □

This lemma shows how to check the superredundancy of a single-literal clause of a formula. All it takes is a simple transformation of the formula: the unit clause l is removed, and the literal $\neg l$ is deleted from all clauses containing it. If what remains implies l, then l is superredundant.

This condition can be further simplified when not only the clause to be checked is a literal but its converse does not even occur in the formula.

Corollary 4 *If $\neg l$ does not occur in F, the single-literal clause l of $F \cup \{l\}$ is superredundant if and only if $F \models l$.*

Proof. A clause l of $F \cup \{l\}$ is superredundant if and only if $F' \models l$ where $F' = \{c \in F \cup \{l\} \mid \neg l \notin c,\ c \neq l\} \cup \{c \mid c \vee \neg l \in F \cup \{l\}\} \models l$ thanks to Corollary 3 applied to $F \cup \{l\}$. If $\neg l$ does not occur in F then F' simplifies as follows.

$$\begin{aligned} F' &= \{c \in F \cup \{l\} \mid \neg l \notin c,\ c \neq l\} \cup \{c \mid c \vee \neg l \in F \cup \{l\}\} \\ &= \{c \in F \cup \{l\} \mid c \neq l\} \\ &= F \end{aligned}$$

This proves that l is superredundant in $F \cup \{l\}$ if and only if $F \models l$ in this case. □

The formula where l is superredundant is $F \cup \{l\}$. Alternatively, l is superredundant in F if and only if $F\backslash\{l\} \models l$, provided that $\neg l$ does not occur in F.

Talking about pure literals, another equivalent condition exists. A variable may always occur with the same sign in a formula; if so, the clauses containing it are irrelevant to the superredundancy of the others.

Lemma 15 *If $\neg l$ does not occur in F, $l \notin c$ and $l \in c'$, the clause c of F is superredundant if and only if it is superredundant in $F\backslash\{c'\}$.*

Proof. No derivation $F \vdash c$ involves c'. This is proved by contradiction. Clause c' may resolve with other clauses of F, but the resolving variable cannot be l since no clause of F contains $\neg l$. Therefore, the resulting clauses all contain l. The same applies to them: they may resolve, but the result contains l. Inductively, this proves that l is also in the root of the derivation tree. The root is c, which does not contain l by assumption. This contradiction proves that c' is not involved in $F \vdash c$.

A derivation $F \vdash G \vdash c$ is a resolution tree with leaves F and root c. It is a particular case of $F \vdash c$. Therefore, it does not contain c'. As a result, it can be rewritten $F\backslash\{c'\} \vdash G \vdash c$.

A derivation $F \vdash c''$ with $c'' \subset c$ does not contain c' because c'' does not contain l. Therefore, $F \vdash c''$ is the same as $F\backslash\{c\} \vdash c''$ for every subset c'' of c.

The third equivalent condition to superredundancy in Lemma 11 is: $F \vdash G \vdash c$ with $c \notin G$ or $F \vdash c''$ with $c'' \subset c$. These conditions are respectively equivalent to $F\backslash\{c'\} \vdash G \vdash c$

and $F\backslash\{c'\} \vdash c''$, the third equivalent condition to the superredundancy of c in $F\backslash\{c'\}$ of Lemma 11. □

A particular case meeting the assumption of the lemma is when a variable only occurs in one clause. The lemma specializes as follows.

Corollary 5 *If a variable occurs in F only in the clause c, then $c' \neq c$ is superredundant in F if and only if it is superredundant in $F\backslash\{c\}$.*

A number of equivalent conditions have been provided. Time to turn to sufficient conditions. The next corollary shows a sufficient condition to superredundanty. The following results are about superirredundancy.

Corollary 6 *If $F \models c'$ and $c' \subset c \in F$, then c is superredundant in F.*

Proof. Immediate consequence of the fourth equivalent condition of Lemma 11. □

While proving superredundancy is sometimes useful, its main aim is to prove a formula minimal, which is the case if all its clauses are superirredundant. This is why much effort is devoted to proving superirredundancy.

Proving that all clauses of a formula are superirredundant is complicated. More so on a family of formulae. These occurs in hardness proofs. For example, a certain logical problem may be shown NP-hard by reduction from vertex cover. Each graph is translated into a different formula. If formula minimality is required, it is required for all formulae obtained by translating all possible graphs. All clauses of all these formulae have to be proved superirredundant.

A way to prove superirredundancy is by first simplifying the formula and then proving superirredundancy on the result. Of course, not all simplifications work. Proving the superirredundancy of $a \vee b \vee c$ in $F = \{a \vee b, a \vee b \vee c\}$ is a counterexample. Removing the first clause from F makes $a \vee b \vee c$ superirredundant in what remains, $F' = \{a \vee b \vee c\}$. Yet, $a \vee b \vee c$ is not superirredundant in F.

A simplification works only if the superirredundancy of the clause in the simplified formula implies its superirredundancy in the original formula. In the other way around, superredundancy in the original implies superredundancy in the simplification.

What required is that "c superredundant in F" implies "c superredundant in the simplified F". One direction is enough. "Implies", not "if and only if".

The following lemmas are formulated in the direction where superredundancy implies superredundancy. This simplifies their formulation and their proofs, but they are mostly used in reverse: superirredundancy implies superirredundancy.

Lemma 16 *If a clause c of F is superredundant, it is also superredundant in $F \cup \{c'\}$.*

Proof. The assumed superredundancy of a clause c of F is by definition $\mathrm{ResCn}(F)\backslash\{c\} \models c$. The derivations by resolution from F are also valid from $F \cup \{c'\}$, where c' is just not used. Therefore, $\mathrm{ResCn}(F) \subseteq \mathrm{ResCn}(F \cup \{c'\})$. This implies $\mathrm{ResCn}(F)\backslash\{c\} \subseteq \mathrm{ResCn}(F \cup \{c'\})\backslash\{c\}$, which implies $\mathrm{ResCn}(F \cup \{c'\})\backslash\{c\} \models \mathrm{ResCn}(F)\backslash\{c\}$. By transitivity of entailment, the claim follows: $\mathrm{ResCn}(F \cup \{c'\})\backslash\{c\} \models c$. □

How this lemma is used: a formula F may simplify when a clause is added to it, making superirredundancy easy to prove. For example, adding the single-literal clause x allows removing from F all clauses that contain x. If all clauses but c contain x, the superirredundancy of c in F follows from the superirredundancy of c in $\{c, x\}$.

This example of adding a single-literal clause extends to a full-fledged sufficient condition to superredundancy. Adding x to F has the same effect of replacing x with true and simplifying the formula. This transformation is defined as follows.

$$\begin{aligned} F[\mathsf{true}/x] &= \{c[\mathsf{true}/x] \mid c \in F,\ c[\mathsf{true}/x] \neq \top\} \\ c[\mathsf{true}/x] &= \begin{cases} c\backslash\{\neg x\} & \text{if } x \not\in c \\ \top & \text{otherwise} \end{cases} \end{aligned}$$

The symbol $\top$ used in the definition does not occur in the final formula since clauses that are turned into $\top$ are removed from F. In other words, $F[\mathsf{true}/x]$ is a formula built over variables and propositional connective; it does not contain any special symbol for true or false.

Swapping x and $\neg x$ turns the definition of $F[\mathsf{true}/x]$ into $F[\mathsf{false}/x]$.

The next lemma shows that such substitutions often preserve superirredundancy. This would be obvious if superirredundancy were the same as clause primality or a similar semantical notion, but it has been proved not to be by Lemma 8.

Lemma 17 *A clause c of $F[\mathsf{true}/x]$ is superredundant if it is superredundant in F, it contains neither x nor $\neg x$ and F does not contain $c \vee \neg x$. The same holds for $F[\mathsf{false}/x]$ if F does not contain $c \vee x$.*

Proof. The claim is proved for $x = \mathsf{true}$. It holds for $x = \mathsf{false}$ by symmetry.

The assumption that c is superredundant in F is equivalent to $F\backslash\{c\} \cup \text{resolve}(F, c) \models c$ thanks to Lemma 14. The claim is the superredundancy of c in $F[\mathsf{true}/x]$, which is equivalent to $F[\mathsf{true}/x]\backslash\{c\} \cup \text{resolve}(F[\mathsf{true}/x], c)) \models c$ still thanks to Lemma 14.

The claim is the last of a chain of properties that follow from the assumption $F\backslash\{c\} \cup \text{resolve}(F, c) \models c$.

1. $(F\backslash\{c\} \cup \text{resolve}(F, c))[\mathsf{true}/x] \models c$

   Let $H = F\backslash\{c\} \cup \text{resolve}(F, c)$. The assumption is $H \models c$, the claim $H[\mathsf{true}/x] \models c$. By Boole's expansion theorem [5], H is equivalent to $x \wedge H[\mathsf{true}/x] \vee \neg x \wedge H[\mathsf{false}/x]$. The assumption $H \models c$ is therefore the same as $x \wedge H[\mathsf{true}/x] \vee \neg x \wedge H[\mathsf{false}/x] \models c$. Since a disjunction is implied by any of its disjuncts, $x \wedge H[\mathsf{true}/x] \models c$ follows. Since neither c nor $H[\mathsf{true}/x]$ contain x, this is the same as $H[\mathsf{true}/x] \models c$.

2. $(F\backslash\{c\} \cup \text{resolve}(F, c))[\mathsf{true}/x] = (F\backslash\{c\})[\mathsf{true}/x] \cup \text{resolve}(F, c)[\mathsf{true}/x]$

   Formula $F\backslash\{c\} \cup \text{resolve}(F, c)$ is a union. The substitution therefore applies to each of its sets.

   The two parts of the formula are considered separately.

3. $(F \backslash \{c\})[\mathsf{true}/x] = F[\mathsf{true}/x] \backslash \{c\}$ if $c \vee \neg x \notin F$

   Both formulae are made of some clauses of F with the substitution $[\mathsf{true}/x]$ applied to them. They differ on whether c is subtracted before or after the substitution. The claim is proved by showing that for every $c' \in F$, the clause $c'[\mathsf{true}/x]$ is in $(F \backslash \{c\})[\mathsf{true}/x]$ if and only if it is in $F[\mathsf{true}/x] \backslash \{c\}$. The two cases $x \in c'$ and $x \notin c'$ are considered separately.

   If c' contains x, then $c'[\mathsf{true}/x] = \top$. As a result, $F[\mathsf{true}/x]$ does not contain $c'[\mathsf{true}/x]$; its subset $F[\mathsf{true}/x] \backslash \{c\}$ does not either. Neither does $(F \backslash \{c\})[\mathsf{true}/x]$; indeed, $c' \in F \backslash \{c\}$ since c' contains x while c does not, but still $c'[\mathsf{true}/x]$ is equal to $\top$ because it contains x, and is not therefore in $(F \backslash \{c\})[\mathsf{true}/x]$.

   If c' does not contain x, then $c'[\mathsf{true}/x] = c' \backslash \{\neg x\}$. This clause is equal to c if and only if c' is either c or $c \vee \neg x$; the second cannot be the case since F by assumption contains c' but not $c \vee \neg x$. As a result, $c'[\mathsf{true}/x] = c$ if and only if $c' = c$. If $c' = c$ then $c'[\mathsf{true}/x] = c$ is removed from $F[\mathsf{true}/x]$ when subtracting c and c' is removed from F when subtracting c. If $c' \neq c$ then $c'[\mathsf{true}/x] = c$ is not removed from $F[\mathsf{true}/x]$ and is therefore in $F[\mathsf{true}/x] \backslash \{c\}$; also c' is not removed from F and is therefore in $F \backslash \{c\}$, which means that $c'[\mathsf{true}/x]$ is in $(F \backslash \{c\})[\mathsf{true}/x]$. In both cases, either $c'[\mathsf{true}/x]$ is in both sets or in none.

4. $\mathrm{resolve}(F, c)[\mathsf{true}/x] = \mathrm{resolve}(F[\mathsf{true}/x], c)$

   Expanding the definitions of $\mathrm{resolve}(F, c)$ and $\mathrm{resolve}(F[\mathsf{true}/x], c)$ shows that the claim is:

$$(\bigcup_{c' \in F} \mathrm{resolve}(c', c))[\mathsf{true}/x] = \bigcup_{c'' \in F[\mathsf{true}/x]} \mathrm{resolve}(c'', c)$$

   The first substitution is applied to a union, and can therefore equivalently be applied to each of its members:

$$\bigcup_{c' \in F} (\mathrm{resolve}(c', c)[\mathsf{true}/x]) = \bigcup_{c'' \in F[\mathsf{true}/x]} \mathrm{resolve}(c'', c)$$

   If x is in c' it is also in the result of resolving c' with c since c does not contain $\neg x$ by assumption. As a result, $\mathrm{resolve}(c', c)[\mathsf{true}/x] = \top$: the clauses c' that contain x do not contribute to the first union. The claim therefore becomes:

$$\bigcup_{c' \in F,\ x \notin x} (\mathrm{resolve}(c', c)[\mathsf{true}/x]) = \bigcup_{c'' \in F[\mathsf{true}/x]} \mathrm{resolve}(c'', c)$$

   The formula $F[\mathsf{true}/x]$ comprises by definition the clauses $c'' = c'[\mathsf{true}/x]$ such that $c' \in F$ and $c'[\mathsf{true}/x] \neq \top$. The second condition $c'[\mathsf{true}/x] \neq \top$ is false if $x \in c'$. The clauses containing x do not contribute to the second union either.

$$\bigcup_{c' \in F,\ x \notin x} (\mathrm{resolve}(c', c)[\mathsf{true}/x]) = \bigcup_{c' \in F,\ x \notin x} \mathrm{resolve}(c'[\mathsf{true}], c)$$

This equality is proved as a consequence of the pairwise equality of the elements of the unions.

$$\text{resolve}(c', c)[\mathsf{true}/x] = \text{resolve}(c'[\mathsf{true}/x], c) \text{ for every } c' \in F \text{ such that } x \notin c'$$

Neither c nor c' contain x: the first by the assumption of the lemma, the second because of the restriction in the above equality. Since $\text{resolve}(c', c)$ only contains literals of c and c', it does not contain x either. Replacing x with true in a clause that does not contain x is the same as removing $\neg x$.

$$\text{resolve}(c', c)\backslash\{\neg x\} = \text{resolve}(c'\backslash\{\neg x\}, c) \text{ for every } c' \in F \text{ such that } x \notin c'$$

If l is a literal of c' such that $\neg l \in c$, then $l \neq \neg x$ because c does not contain x. As a result, $l \in c'$ implies $l \in c'\backslash\{\neg x\}$. The converse also holds because of set containment. This proves that c resolves with c' on a literal if and only it resolves with $c'\backslash\{\neg x\}$ on the same literal.

If these clauses do not resolve both sides of the equality are empty and therefore equal. Otherwise, both c' and $c'\backslash\{\neg x\}$ resolve with c on the same literal l. The result of resolving c' with c is $\text{resolve}(c', c) = c \cup c'\backslash\{l, \neg l\}$; as a result, the left-hand side of the equality is $\text{resolve}(c', c)\backslash\{\neg x\} = c \cup c'\backslash\{l, \neg l\}\backslash\{\neg x\}$. This is the same as the right hand side $\text{resolve}(c'\backslash\{x\}, c) = c \cup (c'\backslash\{\neg x\})\backslash\{l, \neg l\}$ since c does not contain $\neg x$ by assumption.

Summing up, the superredundancy of c in F expressed as $F\backslash\{c\} \cup \text{resolve}(F, c) \models c$ thanks to Lemma 14 implies $(F\backslash\{c\} \cup \text{resolve}(F, c))[\mathsf{true}/x] \models c$, and the formula in this entailment is the same as $F[\mathsf{true}/x]\backslash\{c\} \cup \text{resolve}(F[\mathsf{true}/x], c)$. The conclusions $F[\mathsf{true}/x]\backslash\{c\} \cup \text{resolve}(F[\mathsf{true}/x], c) \models c$ is equivalent to the superredundancy of c in $F[\mathsf{true}/x]$ thanks to Lemma 14. □

Is the assumption $c \vee \neg x \notin F$ necessary? A counterexample disproves the lemma without it: the clause a is superredundant in $F = \{a \vee \neg x, a, x\}$ because it is redundant, but is superirredundant in $F[\mathsf{true}/x] = \{a\}$. This formula is in the `notsetvalue.py` file of `minimize.py`.

A way to prove superirredundancy is by applying Lemma 17 coupled with Lemma 13. A suitable evaluation of the variables not in c removes or simplifies the other clauses of F to the point they do not resolve, where Lemma 13 shows that c is superirredundant. Lemma 17 proves that c is also superirredundant in F.

An example is $F = \{a \vee b, b \vee c, \neg b \vee \neg d, \neg c \vee d \vee e\}$, the formula in the `setvaluenoresolution.py` file of `minimize.py`. Replacing c with true and d with false deletes the second and third clause and simplifies the fourth, leaving $F[\mathsf{true}/c][\mathsf{false}/d] = \{a \vee b, e\}$, where $a \vee b$ is superirredundant because no clause resolve in this formula. This proves that $a \vee b$ is also superirredundant in F.

A substitution may not prevent all resolutions, but still breaks the formula in small unlinked parts. Such parts can be worked on separately.

Lemma 18 *If F' does not share variables with F and F' is satisfiable, a clause c of F is superredundant if and only if it is superredundant in $F \cup F'$.*

Proof. Since c is in F, it is also in $F \cup F'$. By Lemma 14, the superredundancy of c in $F \cup F'$ is equivalent to $(F \cup F' \cup \text{resolve}(F \cup F', c)) \backslash \{c\} \models c$. Since c is in F and F does not share variables with F', the clause c does not share variables with F' and therefore does not resolve with any clause in F'. This proves $\text{resolve}(F \cup F', c) = \text{resolve}(F, c)$. The entailment becomes $(F \cup F' \cup \text{resolve}(F, c)) \backslash \{c\} \models c$, and also $(F \cup \text{resolve}(F, c)) \backslash \{c\} \cup F' \models c$ since $c \not\in F'$. This is the same as $(F \cup \text{resolve}(F, c)) \backslash \{c\} \models c$ because F' is satisfiable and because of the separation of the variables. This is equivalent to the superredundancy of c in F by Lemma 14. □

Short guide on using Lemma 17 and Lemma 18: to prove c superirredundant in F, all clauses c' that resolve with it are found and all their variables not in c collected; these variables are set to values that satisfy as many clauses c' as possible. All these clauses link c with the rest of F, and removing them makes c isolated and therefore superirredundant.

For example, the clause $c = a \vee b$ is proved superirredundant in $F = \{a \vee b, \neg a \vee c \vee d, \neg b \vee \neg c \vee \neg f, \neg d \vee f \vee g, d \vee h\}$ by a substitution that removes the clauses of F that share variables with c.

The clauses of F that share variables with $c = a \vee b$ are $\neg a \vee c \vee d$ and $\neg b \vee \neg c \vee \neg f$. They are to be removed by substituting variables other than a and b. For example, $c = \mathsf{true}$ removes the first and simplifies the second into $\neg b \vee \neg f$, which is removed by $f = \mathsf{false}$. This substitution turns F into $F[c/\mathsf{true}][f/\mathsf{false}] = \{a \vee b, \neg d \vee g, d \vee h\}$. Since its clause $c = a \vee b$ does not share variables with the other two clauses, it is superirredundant. As a result, it is also superredundant in F.

A final sufficient condition to superirredundancy is given by the following lemma. It is specular to Corollary 4: that lemma applies when $\neg l$ is not in F, this one when l is not in F.

Lemma 19 *If l does not occur in F, then l is superirredundant in $F \cup \{l\}$ if this formula is satisfiable.*

Proof. By Corollary 3, the superredundancy of l in $F \cup \{l\}$ is the same as $F' \models l$, where F' is:

$$\begin{aligned} F' &= \{c \in F \cup \{l\} \mid \neg l \not\in c,\ c \neq l\} \cup \{c \mid c \vee \neg l \in F \cup \{l\}\} \\ &= \{c \in F \mid \neg l \not\in c\} \cup \{c \mid c \vee \neg l \in F\} \end{aligned}$$

The first part of the union is a subset of F; the second comprises only subclauses of F. Since F does not contain l, this union F' does not contain l either. Therefore, F' entails l only if it is unsatisfiable.

The unsatisfiability of F' is proved to contradict the assumption of the lemma. The first part of F' is a subset of F; each clause c of its second part is a consequence of $c \vee \neg l \in F$ and l, and is therefore entailed by $F \cup \{l\}$. Therefore, this union is entailed by $F \cup \{l\}$. It unsatisfiability implies the unsatisfiability of $F \cup \{l\}$, which is contrary to an assumption of the lemma. □

5 Size after forgetting, Horn case

How much forgetting variables increases or decreases size? Decreasing size may be the whole reason for forgetting: make the formula take less storage space, make it easier to be worked on (inference, satisfiability), make it show its epistemological properties (presence of literals, dependence between literals). Even if decreasing size is not the reason for forgetting, it may be an added benefit of it; increasing size a drawback.

In both cases, the question remains: given a formula A and a set of variables Y, how much space forgetting Y from A takes? Technically, how large is a formula expressing forgetting Y from A? A complexity analysis of a decision problem requires turning that question into a yes/no problem. Given k, A and Y, does a formula B of size bounded by k express forgetting?

This is a decision problem: its instances comprise a formula A and a number k each; the solution is yes or no. Yet, it may not always capture the question of interest. For example, A may be a formula of size 100 that can be reduced to size 20 by forgetting the variables Y. This looks like a good result: the resulting formula takes much less space to be stored, checking what can be inferred from it is usually easier, and its literals are probably related in some simple way. Yet, all of this may be illusory: formula A has size 100, but only because it is extremely redundant; it could be reduced to size 10 just by rearrangements, without forgetting anything. That forgetting can be expressed in size 20 no longer looks good. It is not even a size decrease, it is a size doubling.

If forgetting was required independent on size, and checking size is a side question, the problem still makes sense: can forgetting A be expressed by a formula of size 20? If forgetting is done for size reasons, or for reasons that depend on size, the problem is not this but rather "does forgetting reduce size?" or "how much forgetting increases or decreases size?" This question depends on the original size of the formula. The answer is not "20" but rather "forgetting increases size from 10 to 20". It is certainly not "forgetting decreases size from 100 to 20", since the formula can be shrunk more without forgetting.

A simple variant of the problem eliminates this difference: formula A is assumed to be already reduced to its bare minimum. Membership is proved in the general case, but hardness in this restricted case. This way, the complexity classification also holds when the question is to establish whether forgetting reduces or increases size, rather than just establishing the size after forgetting.

If the formula is not assumed minimal, the problem is Σ_2^p-complete [47]. The hardness proof does not just lack the minimality assumption; it hinges around the minimal size of the formula. Forgetting is expressed in a certain size or not depending on whether the formula can. The size of forgetting is more or less the size without forgetting. This is not the question of whether forgetting increases or decreases size, it is mostly whether the formula itself can be reduced size.

Checking whether forgetting Y from A can be expressed in space k is easy to be proved in Σ_2^p if k is unary or polynomially bounded by the size of A: all it takes is checking all formulae of size k for their equisatisfiability with all sets of literals S over the remaining variables by Lemma 1. Hardness is not so easy to prove, and in fact leaves a gap to membership: it is only proved D^p-hard in this article.

Not that D^p-hardness is easy to prove. It requires two long lemmas, one for a coNP-hardness reduction and one for a NP-hardness reduction. These reductions have a form that allows them to be merged into a single D^p-hardness reduction.

A generic NP-hardness reduction is "if F is satisfiable then forgetting takes space less than or equal to k and greater otherwise". Such reductions cannot be merged. An additional property is required: forgetting can never be expressed in size less than k. If this is also a property of a coNP-hardness reduction where the size bound is l, the overall size is always $k + l$ or greater, with $k + l$ being only possible when the first formula is satisfiable and the second unsatisfiable.

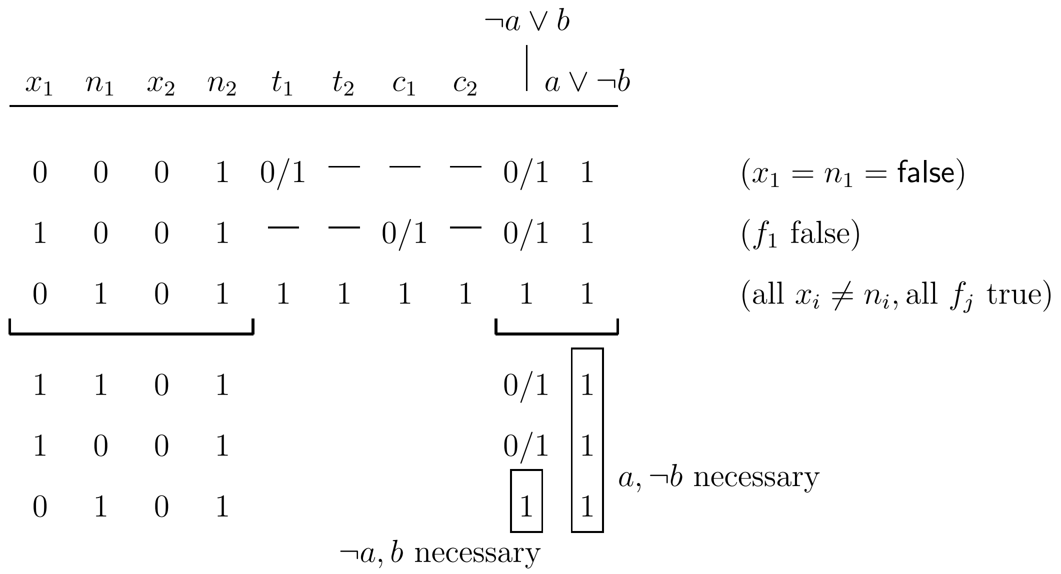
This explains why the lemmas are formulated with "equal to k" in one case and "greater than k" in the other. Their other peculiarity, that the formula generated by the reduction is required to be minimal, is due to the the reasons explained before.

Lemma 20 *There exists a polynomial algorithm that turns a CNF formula F into a minimal-size Horn formula A, a subset $X_C \subseteq Var(A)$ and a number k such that forgetting all variables but X_C from A is expressed by a Horn formula of size k if F is unsatisfiable and only by CNF formulae of size greater than or equal to $k + 2$ if F is satisfiable.*

Proof. Let $F = \{f_1, \ldots, f_m\}$ be a CNF formula built over the alphabet $X = \{x_1, \ldots, x_n\}$. The reduction employs the fresh variables $E = \{e_1, \ldots, e_n\}$, $T = \{t_1, \ldots, t_n\}$, $C = \{c_1, \ldots, c_m\}$ and $\{a, b\}$. The formula A, the set of variables X_C and the number k are:

$$\begin{array}{rcl}
A & = & \{\neg x_i \vee \neg e_i, \neg x_i \vee t_i, \neg e_i \vee t_i \mid x_i \in X\} \cup \\
& & \{\neg x_i \vee c_j \mid x_i \in f_j\} \cup \{\neg e_i \vee c_j \mid \neg x_i \in f_j\} \cup \\
& & \{\neg t_1 \vee \cdots \vee \neg t_n \vee \neg c_1 \vee \cdots \vee \neg c_m \vee \neg a \vee b\} \cup \\
& & \{a \vee \neg b\} \\
X_C & = & X \cup E \cup \{a, b\} \\
k & = & 2 \times n + 2
\end{array}$$

Before formally proving the claim, how the reduction works is summarized. Some literals are still necessary after forgetting, and some of them are necessary only if F is satisfiable. The clauses $\neg x_i \vee \neg e_i$ make $\neg x_i$ and $\neg e_i$ necessary. The clause $a \vee \neg b$ makes a and $\neg b$ necessary. If F is always false then for every value of the variables $X \cup E$ either some t_i can be set to false (if $x_i = e_i = \mathsf{false}$) or some c_j can be set to false (because e_i is the negation of x_i, and at least a clause of F is false). This makes the clause $\neg t_1 \vee \cdots \vee \neg t_n \vee \neg c_1 \vee \cdots \vee \neg c_m \vee \neg a \vee b$ satisfied regardless of a and b. Instead, if the formula is satisfied by an evaluation of X and E is its opposite, then all c_j and t_i have to be true, turning $\neg t_1 \vee \cdots \vee \neg t_n \vee \neg c_1 \vee \cdots \vee \neg c_m \vee \neg a \vee b$ into $\neg a \vee b$. This makes $\neg a$ and b necessary as well.



x_1	n_1	x_2	n_2	t_1	t_2	c_1	c_2	$\neg a \vee b$	$a \vee \neg b$	
0	0	0	1	0/1	—	—	—	0/1	1	($x_1 = n_1 =$ false)
1	0	0	1	—	—	0/1	—	0/1	1	(f_1 false)
0	1	0	1	1	1	1	1	1	1	(all $x_i \neq n_i$, all f_j true)

1	1	0	1	0/1	$\boxed{1}$
1	0	0	1	0/1	$\boxed{1}$
0	1	0	1	$\boxed{1}$	$\boxed{1}$

The figure shows three models as an example. In the first model, the assignments $x_1 = e_1 =$ false allow t_1 to take any value (denoted 0/1); regardless of the other variables (irrelevant values are marked $-$), $t_1 =$ false satisfies the clause $\neg t_1 \vee \cdots \vee \neg t_n \vee \neg c_1 \vee \cdots \vee \neg c_m \vee \neg a \vee b$ without the need to also satisfy its subset $\neg a \vee b$; this subclause can be false and still A is true. In the second model the values of x_i and e_i are opposite to each other for every i, but the clause $f_2 \in F$ is false; c_1 can take any value, including false; this again allows A to be true even if $\neg a \vee b$ is false. In the third model, the variables x_i and e_i are all opposite to each other and all clauses of F true; all t_i and c_i are forced to be true, making $\neg a \vee b$ the only way to satisfy the clause $\neg t_1 \vee \cdots \vee \neg t_n \vee \neg c_1 \vee \cdots \vee \neg c_m \vee \neg a \vee b$. When removing the intermediate variables t_i and c_i, all that matters is whether $\neg a \vee b$ was allowed to be false for some values of the removed variables or not. This is the case for the first two models but not the third, where $\neg a$ and b are necessary.

Minimality. The minimality of A is proved applying Lemma 17 to remove some clauses so that the remaining ones do not resolve and Lemma 13 applies. Lemma 10 proves A minimal since it only contains superirredundant clauses.

Substituting the variables a, b with false removes $\neg t_1 \vee \cdots \vee \neg t_n \vee \neg c_1 \vee \cdots \vee \neg c_m \vee \neg a \vee b$ and $a \vee \neg b$ from A. The remaining clauses contain x_i, e_i only negative and t_i, c_j only positive. Therefore, these clauses do not resolve. Since they do not contain each other, Lemma 13 proves them superirredundant. They are also superirredundant in A by Lemma 17 since A does not contain any of their supersets.

The superirredundancy of the remaining two clauses is proved by substituting all x_i, e_i with false. This substitution removes the clauses $\neg x_i \vee \neg e_i$, $\neg x_i \vee t_i$, $\neg e_i \vee t_i$, $\neg x_i \vee c_j$ and $\neg e_i \vee c_j$ because they all contain either $\neg x_i$ or $\neg e_i$. The two remaining clauses are $\neg t_1 \vee \cdots \vee \neg t_n \vee \neg c_1 \vee \cdots \vee \neg c_m \vee \neg a \vee b$ and $a \vee \neg b$. They have opposite literals, but resolving them results in tautologies. As a result, $F = \mathrm{ResCn}(F)$. Since none of the two entails the other, they are irredundant in $\mathrm{ResCn}(F)$ and are therefore superirredundant. By Lemma 17, they are superirredundant in A as well since A does not contain a superset of them.

Formula F is unsatisfiable. Forgetting all variables but X_C from A is expressed by $B = \{\neg x_i \vee \neg e_i\} \cup \{a \vee \neg b\}$, a Horn formula of the required variables $X_C = X \cup E \cup \{a, b\}$ and size $||B|| = 2 \times n + 2 = k$.

Corollary 2 proves that B expresses forgetting if every set of literals S that contains exactly all variables $X_C = X \cup E \cup \{a, b\}$ is satisfiable with A if and only if it is satisfiable with B. Two cases are possible.

$\{x_i, e_i\} \subseteq S$ **for some** i ; the clause $\neg x_i \vee \neg e_i$ in both A and B is falsified by S; both $A \cup S$ and $B \cup S$ are unsatisfiable;

$\{x_i, e_i\} \subseteq S$ **for no** i ; since S contains either x_i or $\neg x_i$ for each i and either e_i or $\neg e_i$ for each i, either $\neg x_i \in S$ or $\neg e_i \in S$; as a result, all clauses $\neg x_i \vee \neg e_i$ are satisfied in $A \cup S$ and $B \cup S$, and can therefore be disregarded from this point on; the only remaining clause of B is $a \vee \neg b$;

if S contains $\neg a$ and b, then B is not satisfied; but A contains the same clause $a \vee \neg b$, so it is not satisfied either; if S contains both a and b or both $\neg a$ and $\neg b$, then B is satisfied, and A is also satisfied by setting all variables t_i and c_j to true; therefore, the only sets S that may differ when added to A and B are those containing a and $\neg b$; these sets are consistent with B; they make the clause $a \vee \neg b$ of A redundant, and resolve with the clause $\neg t_1 \vee \cdots \vee \neg t_n \vee \neg c_1 \vee \cdots \vee \neg c_m \vee \neg a \vee b$ making it subsumed by $\neg t_1 \vee \cdots \vee \neg t_n \vee \neg c_1 \vee \cdots \vee \neg c_m$.

Two subcases are considered:

$\{\neg x_i, \neg e_i\} \subseteq S$ **for some** i the remaining clauses of A are satisfied by setting t_i to false, all t_z with $z \neq i$ to true and all c_j to true; in particular, the clause $\neg t_1 \vee \cdots \vee \neg t_n \vee \neg c_1 \vee \cdots \vee \neg c_m$ is satisfied because of $t_i = \mathsf{false}$;

$\{\neg x_i, \neg e_i\} \subseteq S$ **for no** i ; at this point, also $\{x_i, e_i\} \subseteq S$ for no i; as a result, S contains either $\{x_i, \neg e_i\}$ or $\{\neg x_i, e_i\}$, which means that it implies $x_i \not\equiv e_i$; the clauses $\neg e_i \vee c_j$ are therefore equivalent to $x_i \vee c_j$; by assumption, at least a clause of F is false for every possible value of the variables X; let f_j be such a clause for the only truth evaluation on X that satisfies S; by setting all variables t_i and all c_z with $z \neq j$ to true and c_j to false, this assignment satisfies all clauses; in particular, the clauses $\neg x_i \vee c_j$ and $x_i \vee c_j$ are satisfied even if c_j is false because f_j is false in S, which implies that all literals x_i and $\neg x_i$ it contains are false; the clause $\neg t_1 \vee \cdots \vee \neg t_n \vee \neg c_1 \vee \cdots \vee \neg c_m$ is satisfied because of $c_j = \mathsf{false}$.

All of this proves that B is the result of forgetting all variables but X_C from A.

Minimal number of literals. Every CNF formula B that expresses forgetting all variables but $X_C = X \cup E \cup \{a, b\}$ from A contains at least $k = 2 \times n + 2$ literal occurrences.

This is proved by showing that B contains the literals $\neg x_i$, $\neg e_i$, a and $\neg b$. This is in turn proved by Lemma 6: a set S satisfying each of them l is shown consistent with A while $S \backslash \{l\} \cup \{\neg l\}$ is not.

For the literals $\neg x_i$ and a the set S contains all $\neg x_i$, all e_i, a and b. It is consistent with A because both are satisfied by the model that sets all x_i to false and all e_i, t_i, c_i, a and b

to true. Replacing $\neg x_i$ with x_i makes S inconsistent with $\neg x_i \vee \neg e_i$. Replacing a with $\neg a$ makes S inconsistent with $a \vee \neg b$.

For the literals $\neg e_i$ and $\neg b$, the set S contains all x_i, all $\neg e_i$, $\neg a$ and $\neg b$. It is consistent with A because both are satisfied by the model that assigns all x_i, t_i and c_i to true and all e_i, a and b to false. Replacing $\neg e_i$ with e_i makes S inconsistent with $\neg x_i \vee \neg e_i$. Replacing $\neg b$ with b makes it inconsistent with $a \vee \neg b$.

This proves that every formula obtained by forgetting all variables but X_C from A contains all the $k = 2 \times n + 2$ literals $\neg x_i$, $\neg e_i$, a and $\neg b$.

Formula F is satisfiable. If this is the case, every CNF formula B that expresses forgetting all variables but X_C from A contains the literals $\neg a$ and b. These two literals are in addition to the k literals of the previous point, raising the minimal number of literals to $k + 2$.

That B contains $\neg a$ is proved by exhibiting a set of literals S that is consistent with A while $S \backslash \{l\} \cup \{\neg l\}$ is not where $l = \neg a$. This implies that B contains $\neg a$ by Lemma 6. A similar set with $l = b$ shows that B also contains b.

Let M be a model of F. The set of literals S contains x_i or $\neg x_i$ depending on whether M satisfies x_i; it contains e_i or $\neg e_i$ depending on whether M falsifies x_i; it also contains $\neg a$ and $\neg b$. This set is consistent with A because they are both satisfied by the model that extends M by setting each e_i opposite to x_i, all t_i and c_i to true and a and b to false. In particular, the clause $\neg t_1 \vee \cdots \vee \neg t_n \vee \neg c_1 \vee \cdots \vee \neg c_m \vee \neg a \vee b$ is satisfied because it contains $\neg a$.

Replacing $\neg a$ with a makes S no longer consistent with A. Let $S' = S \backslash \{\neg a\} \cup \{\neg\neg a\}$. This set has the same literals over x_i and e_i of S. Since M satisfies F, for each of its clauses f_j at least a literal in f_j is true in M. If this literal is x_i, then S' contains x_i; since x_i is in f_j, formula A contains the clause $\neg x_i \vee c_j$; therefore, $S' \cup A \models c_j$. If the literal of f_j that is true in M is $\neg x_i$, then S' contains e_i; since $\neg x_i$ is in f_j, formula A contains $\neg e_i \vee c_j$; therefore, $S' \cup A \models c_j$. This proves that regardless of whether the literal of f_j that is true in M is positive or negative, if F is consistent then $S' \cup A$ implies c_j. This is the case for every j since M satisfies all clauses of F. Since S' contains either x_i or e_i for every i and A contains both $\neg x_i \vee t_i$ and $\neg e_i \vee t_i$ for every i, $S' \cup A$ also implies all variables t_j. Since $S' = S \backslash \{\neg a\} \cup \{a\}$ also contains a and $\neg b$, it is inconsistent with $\neg t_1 \vee \cdots \vee \neg t_n \vee \neg c_1 \vee \cdots \vee \neg c_m \vee \neg a \vee b$. This proves that $\neg a$ is in B.

The similar set S that contains a and b leads to the same point where all variables c_j and t_j are implied, making $\neg t_1 \vee \cdots \vee \neg t_n \vee \neg c_1 \vee \cdots \vee \neg c_m \vee \neg a \vee b$ consistent with $\{a, b\}$ but not with $\{a, \neg b\}$. This proves that b is also in B. □

This lemma shows a polynomial reduction from propositional unsatisfiability to the problem of forget size in the Horn case. It is therefore a proof of coNP-hardness. Yet, it is not formulated this way. It instead predicates about the translation itself. Only this way it could include the additional property that the minimal size is either k or at least $k+2$. This allows merging it with another reduction to form a proof of D^p-hardness.

The minimal size after forgetting is not k or $k+2$. Rather, it is either k or at least $k+2$. If F is unsatisfiable the minimal size is k; otherwise it is at least $k + 2$; "at least" means that the minimal size may not be $k + 2$ but larger. The difference depends on what the formula B resulting from forgetting contains in the two cases. If F is satisfiable then B is $B = \{\neg x_i \vee \neg e_i\} \cup \{a \vee \neg b\}$, which has size k. If F is satisfiable B is only proved to also contain $\neg a$ and b, not to contain any specific clause over them; rather the opposite: these literals are somehow linked to the variables x_i and e_i. In particular, if the values of x_i satisfy

F then $\neg a \vee b$ has to be true; otherwise it does not. This is only possible if B contains some clauses that link the literals x_i with $\neg a \vee b$. The reduction produces a formula of size k after forgetting if F is unsatisfiable, but it otherwise produces a formula of size at least $k+2$. If it produced a formula of size exactly $k+2$, it would prove the problem even harder than D^p.

The following lemma shows the problem NP-hard: a formula F is satisfiable if and only if forgetting some variables from A can be expressed in a certain space.

Forgetting these variables from the formulae A produced by the reduction can be done in polynomial time, just not minimally. In other words, if the reduction produces a formula A and variables X_C, forgetting X_C from A can always be expressed by a possibly non-minimal formula B in polynomial time. The complete translation from F to B and k provides an alternative proof of NP-hardness of the minimality problem without forgetting [20]: given a Horn formula B, is there any formula of size k equivalent to it?

In the other way around, the existing proof of NP-hardness of the minimality problem without forgetting [20] could be used to show that the problem with forgetting is NP-hard: a formula A is equivalent to one of size k if and only if forgetting no variable from A can be expressed in space k. Such a proof cannot be used here for two reasons. The first is that adding the requirement that A is minimal invalidates it: if A is minimal then forgetting no variables from it can always be stored in space $||A||$ and never in less space; the constraint of minimality trivializes the problem of minimality. The second is the need for merging the reduction with the previous one, which requires forgetting to be either expressible with size exactly equal k or larger, never smaller.

Lemma 21 *There exists a polynomial algorithm that turns a CNF formula F into a minimal-size Horn formula A, a subset $X_C \subseteq Var(A)$ and a number k such that forgetting all variables but X_C from A is expressed by a Horn formula of size k if F is satisfiable and only by CNF formulae of size greater than k otherwise.*

Proof. Let the formula be $F = \{f_1, \ldots, f_m\}$ and $X = \{x_1, \ldots, x_n\}$ its variables. The formula A is built over an extended alphabet comprising the variables $X = \{x_1, \ldots, x_n\}$ and the additional variables $O = \{o_1, \ldots, o_n\}$, $E = \{e_1, \ldots, e_n\}$, $P = \{p_1, \ldots, p_n\}$, $T = \{t_1, \ldots, t_n\}$, $C = \{c_1, \ldots, c_m\}$, $R = \{r_1, \ldots, r_n\}$, $S = \{s_1, \ldots, s_n\}$ and q.

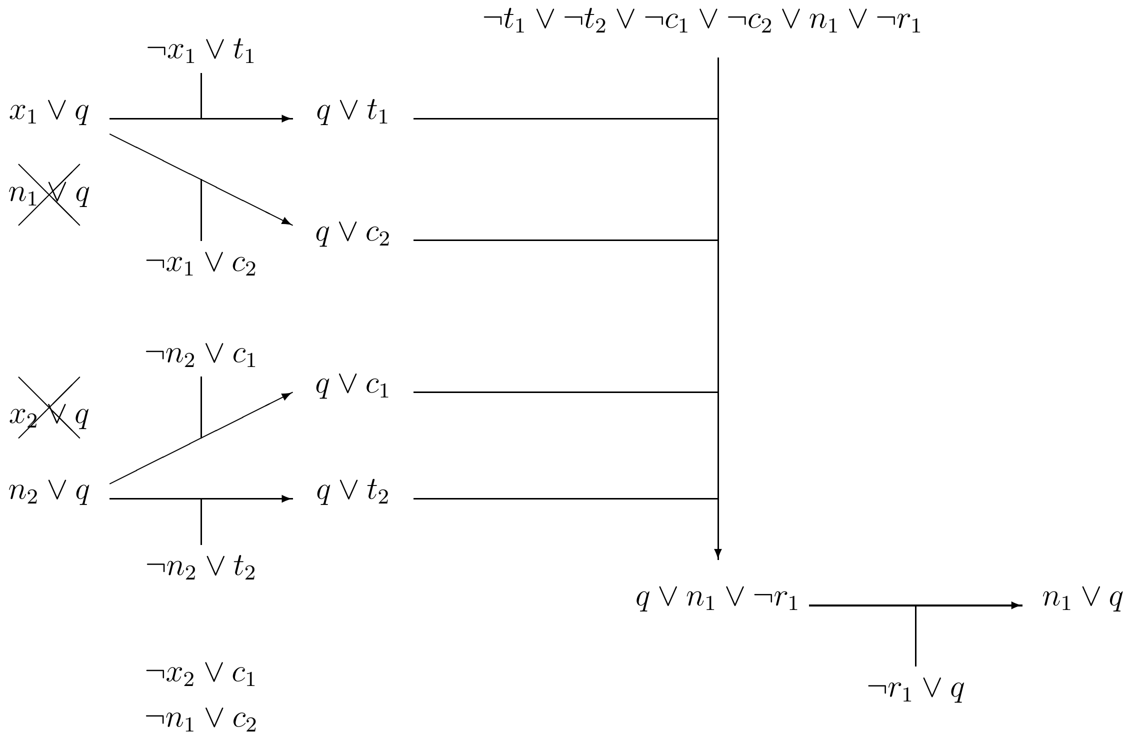
The formula A, the set of variables X_C and the integer k are as follows.

$$\begin{array}{rcl}
A & = & A_F \cup A_T \cup A_C \cup A_B \\
A_F & = & \{x_i \vee \neg o_i, o_i \vee \neg q \mid x_i \in X\} \cup \{e_i \vee \neg p_i, p_i \vee \neg q \mid x_i \in X\} \\
A_T & = & \{\neg x_i \vee t_i, \neg e_i \vee t_i \mid x_i \in X\} \\
A_C & = & \{\neg x_i \vee c_j \mid x_i \in f_j\} \cup \{\neg e_i \vee c_j \mid \neg x_i \in f_j\} \\
A_B & = & \{\neg t_1 \vee \cdots \vee \neg t_n \vee \neg c_1 \vee \cdots \vee \neg c_m \vee x_i \vee \neg r_i, r_i \vee \neg q \mid x_i \in X\} \cup \\
 & & \{\neg t_1 \vee \cdots \vee \neg t_n \vee \neg c_1 \vee \cdots \vee \neg c_m \vee e_i \vee \neg s_i, s_i \vee \neg q \mid x_i \in X\} \\
X_C & = & X \cup E \cup T \cup C \cup R \cup S \cup \{q\} \\
k & = & 2 \times n + ||A_T|| + ||A_C|| + ||A_B||
\end{array}$$

Before formally proving that the reduction works, a short summary of why it works is given. The variables to forget are $O \cup P$. A way to forget them is to turn A_F into

$A_R = \{x_i \vee \neg q, e_i \vee \neg q \mid x_i \in X\}$. The other clauses of A are superirredundant: all minimal equivalent formulae contain them. The bound k allows only one clause of A_R for each i. Combined with the clauses of A_T they entail $t_i \vee \neg q$. If F is satisfiable, they also combine with the clauses A_C to imply all clauses $c_j \vee \neg q$. Resolving these clauses with A_B produces all clauses $x_i \vee \neg q$ and $e_i \vee \neg q$, including the ones not in the formula. This way, a formula that contains one clause of A_R for each index i implies all of A_R, but only if F is satisfiable.

The following figure shows how $e_1 \vee q$ is derived from $x_1 \vee q$ and $e_2 \vee q$, when the formula is $F = \{f_1, f_2\}$ where $f_1 = x_1 \vee x_2$ and $f_2 = \neg x_1 \vee \neg x_2$. These clauses translate into $A_C = \{\neg x_1 \vee c_1, \neg x_2 \vee c_1, \neg e_1 \vee c_2, \neg e_2 \vee c_2\}$.



For each index i, at least one among $x_i \vee q$ and $e_i \vee q$ is necessary for deriving $q \vee t_i$, which is required for these derivations to work. Alternatively, $q \vee t_i$ may be selected. Either way, for each index i at least a two-literal clause is necessary.

The claim is formally proved in four steps: first, a non-minimal way to forget all variables but X_C is shown; second, its superirredundant clauses are identified; third, an equivalent formula of size k is built if F is satisfiable; fourth, the necessary clauses in every equivalent formula are identified; fifth, if F is unsatisfiable every equivalent formula is proved to have size greater than k.

Effect of forgetting.

Theorem 1 proves that forgetting all variables not in X_C, which are $O \cup P$, is expressed by resolving out these variables. Since o_i occurs only in $x_i \vee \neg o_i$ and $o_i \vee \neg q$, the result is $x_i \vee \neg q$. The same holds for p_i. The resulting clauses are denoted A_R:

$$A_R = \{x_i \vee \neg q, e_i \vee \neg q \mid x_i \in X\}$$

Superirredundancy.

The claim requires A to be minimal, which follows from all its clauses being superirredundant by Lemma 10. Most of them survive forgetting; the reduction is based on these being superirredundant. Instead of proving superirredundancy in two different but similar formulae, it is proved in their union.

In particular, the clauses $A_F \cup A_T \cup A_C \cup A_B$ are shown superirredundant in $A_F \cup A_R \cup A_T \cup A_C \cup A_B$. Lemma 16 implies that they are also superirredundant in its subsets $A_F \cup A_T \cup A_C \cup A_B$ and $A_R \cup A_T \cup A_C \cup A_B$, the formula before and after forgetting.

To be precise, the latter is just one among the formulae expressing forgetting. Yet, its superirredundant clauses are in all minimal CNF formulae equivalent to it. Therefore, all minimal CNF formulae expressing forgetting contain them.

Superirredundancy is proved via Lemma 17: a substitution simplify $A_F \cup A_R \cup A_T \cup A_C \cup A_B$ enough to prove superirredundancy easily, for example because its clauses do not resolve and Lemma 13 applies.

- Replacing all variables x_i, e_i, t_i and c_j with true removes from $A_F \cup A_R \cup A_T \cup A_C \cup A_B$ all clauses in $A_R \cup A_T \cup A_C$, all clauses of A_F but $o_i \vee \neg q$ and $p_i \vee \neg q$ and all clauses of A_B but $r_i \vee \neg q$ and $s_i \vee \neg q$. The remaining clauses contain only the literals o_i, p_i, r_i, s_i and $\neg q$. Therefore, they do not resolve. Since none is contained in another, they are all superirredundant by Lemma 13. This proves the superirredundancy of all clauses $o_i \vee \neg q$, $p_i \vee \neg q$, $r_i \vee \neg q$ and $s_i \vee \neg q$.

- Replacing all variables q, o_i, p_i, r_i and s_i with false removes from $A_F \cup A_R \cup A_T \cup A_C \cup A_B$ all clauses but $A_T \cup A_C$. These clauses contain only the literals $\neg x_i$, $\neg e_i$, t_i and c_j. Therefore, they do not resolve. Since they are not contained in each other, Lemma 13 proves them superirredundant.

- Replacing all variables q, r_i and s_i with false and all variables t_i and c_i with true removes from $A_F \cup A_R \cup A_T \cup A_C \cup A_B$ all clauses but $x_i \vee \neg o_i$ and $e_i \vee \neg p_i$. They do not resolve because they do not share variables. Lemma 13 proves them superirredundant because they do not contain each other.

- Replacing all variables with false except for all variables t_i and c_j and the two variables x_h and r_h removes all clauses from $A_F \cup A_R \cup A_T \cup A_C \cup A_B$ but $\neg x_h \vee t_h$, $\neg t_1 \vee \cdots \vee \neg t_n \vee \neg c_1 \vee \cdots \vee \neg c_m \vee x_h \vee \neg r_h$ and all clauses $\neg x_h \vee c_j$ with $x_h \in f_j$. They only resolve in tautologies. Therefore, their resolution closure only contains them. Removing $\neg t_1 \vee \cdots \vee \neg t_n \vee \neg c_1 \vee \cdots \vee \neg c_m \vee x_h \vee \neg r_h$ from the resolution closure leaves only $\neg x_h \vee t_h$ and all clauses $\neg x_h \vee c_j$ with $x_h \in f_j$. They do not resolve since they do not contain opposite literals. Since $\neg t_1 \vee \cdots \vee \neg t_n \vee \neg c_1 \vee \cdots \vee \neg c_m \vee x_h \vee \neg r_h$ is not contained in them, it is not entailed by them. This proves it superirredundant. A similar replacement proves the superirredundancy of each $\neg t_1 \vee \cdots \vee \neg t_n \vee \neg c_1 \vee \cdots \vee \neg c_m \vee e_h \vee \neg s_h$.

These points prove that the clauses $A_F \cup A_T \cup A_C \cup A_B$ are superirredundant in the formula before forgetting and the clauses $A_T \cup A_C \cup A_B$ are superirredundant in the formula after forgetting. The only clauses that may be superredundant are A_R in the formula after forgetting.

Formula F is satisfiable.

Let M be a model satisfying F. Forgetting all variables but X_C is expressed by $A'_R \cup A_T \cup A_C \cup A_B$, where A'_R comprises the clauses $x_i \vee \neg q$ such that $M \models x_i$ and the clauses $e_i \vee \neg q$ such that $M \models \neg x_i$. This Horn formula has size k. It expresses forgetting because it is equivalent to $A_R \cup A_T \cup A_C \cup A_B$. This is proved by showing that it entails every clause in A_R.

Since M satisfies every clause $f_j \in F$, it satisfies at least a literal of f_j: for some x_i, either $x_i \in f_j$ and $M \models x_i$ or $\neg x_i \in f_j$ and $M \models \neg x_i$. By construction, $x_i \in f_j$ implies $\neg x_i \vee c_j \in A_C$ and $\neg x_i \in f_j$ implies $\neg e_i \vee c_j \in A_C$. Again by construction, $M \models x_i$ implies $x_i \vee \neg q \in A'_R$ and $M \models \neg x_i$ implies $e_i \vee \neg q \in A'_R$. As a result, either $x_i \vee \neg q \in A'_R$ and $\neg x_i \vee c_j \in A_C$ or $e_i \vee \neg q \in A'_R$ and $\neg e_i \vee c_j \in A_C$. In both cases, the two clause resolve in $c_j \vee q$.

Since M satisfies either x_i or $\neg x_i$, either $x_i \vee \neg q \in A'_R$ or $e_i \vee \neg q \in A'_R$. The first clause resolve with $\neg x_i \vee t_i$ and the second with $\neg e_i \vee t_i$. The result is $t_i \vee \neg q$ in both cases.

Resolving all these clauses $t_i \vee \neg q$ and $c_j \vee q$ with $\neg t_1 \vee \cdots \vee \neg t_n \vee \neg c_1 \vee \cdots \vee \neg c_m \vee x_i \vee \neg r_i$ and then with $r_i \vee \neg q$, the result is $x_i \vee \neg q$. In the same way, resolving these clauses with $\neg t_1 \vee \cdots \vee \neg t_n \vee \neg c_1 \vee \cdots \vee \neg c_m \vee e_i \vee \neg s_i$ and $s_i \vee \neg q$ produces $e_i \vee \neg q$. This proves that all clauses of A_R are entailed.

Necessary clauses

All CNF formulae that are equivalent to $A_R \cup A_T \cup A_C \cup A_B$ and have minimal size contain $A_T \cup A_C \cup A_B$ because these clauses are superirredundant. Therefore, these formulae are $A_N \cup A_T \cup A_C \cup A_B$ for some set of clauses A_N. This set A_N is now proved to contain either $x_h \vee \neg q$, $x_h \vee \neg r_h$, $e_h \vee \neg q$, $e_h \vee \neg s_h$ or $t_h \vee \neg q$ for each index h. Let M and M' be the following models.

$$\begin{aligned}
M &= \{x_i = e_i = t_i = \mathsf{true} \mid i \neq h\} \cup \{x_h = e_h = t_h = \mathsf{false}\} \cup \\
&\quad \{c_j = \mathsf{true}\} \cup \{q = \mathsf{true}\} \cup \{r_i = \mathsf{true}, s_i = \mathsf{true}\} \\
M' &= \{x_i = e_i = t_i = \mathsf{true} \mid i \neq h\} \cup \{x_h = e_h = t_h = \mathsf{true}\} \cup \\
&\quad \{c_j = \mathsf{true}\} \cup \{q = \mathsf{true}\} \cup \{r_i = \mathsf{true}, s_i = \mathsf{true}\}
\end{aligned}$$

The five clauses are falsified by M. Since the two of them $xh \vee \neg q$ and $e_h \vee \neg q$ are in A_R, this set is also falsified by M. As a result, M is not a model of $A_R \cup A_T \cup A_C \cup A_B$. This formula is equivalent to $A_N \cup A_T \cup A_C \cup A_B$, which is therefore falsified by M. In formulae, $M \not\models A_N \cup A_T \cup A_C \cup A_B$.

The formula $A_N \cup A_T \cup A_C \cup A_B$ contains a clause falsified by M. Since $M \models A_T \cup A_C \cup A_B$, this clause is in A_N but not in $A_T \cup A_C \cup A_B$. In formulae, $M \not\models c$ for some $c \in A_N$ and $c \notin A_T \cup A_C \cup A_B$. This clause is entailed by $A_R \cup A_T \cup A_C \cup A_B$ because this formula entails all of $A_N \cup A_T \cup A_C \cup A_B$, and c is in A_N. In formulae, $A_R \cup A_T \cup A_C \cup A_B \models c$.

This clause c contains either x_h, e_h or t_h. This is proved by deriving a contradiction from the assumption that c does not contain any of these three literals. Since $M \not\models c$, the clause c contains only literals that are falsified by M. Not all of them: it does not contain x_h, e_h and t_h by assumption. It does not contain $\neg x_h$, $\neg e_h$ and $\neg t_h$ either because it would otherwise be satisfied by M. As a result, c is also falsified by M', which is the same as M

but for the values of x_h, e_h and t_h. At the same time, M' satisfies $A_R \cup A_T \cup A_C \cup A_B$, contradicting $A_R \cup A_T \cup A_C \cup A_B \models c$. This contradiction proves that c contains either x_h, e_h or t_h.

From the fact that c contains either x_h, e_h or t_h, that is a consequence of $A_R \cup A_T \cup A_C \cup A_B$, and that is in a minimal-size formula, it is now possible to prove that c contains either $x_h \vee \neg q$, $x_h \vee \neg r_h$, $e_h \vee \neg q$, $e_h \vee \neg s_h$ or $t_h \vee \neg q$.

Since c is entailed by $A_R \cup A_T \cup A_C \cup A_B$, a subset of c follows from resolution from it: $A_R \cup A_T \cup A_C \cup A_B \vdash c'$ with $c' \subseteq c$. This implies $A_N \cup A_T \cup A_C \cup A_B \models c'$ by equivalence. If $c' \subset c$ then $A_N \cup A_T \cup A_C \cup A_B$ would not be minimal because it contained a non-minimal clause $c \in A_N$. Therefore, $A_R \cup A_T \cup A_C \cup A_B \vdash c$.

The only two clauses of $A_R \cup A_T \cup A_C \cup A_B$ that contain x_h are $x_h \vee \neg q$ and $\neg t_1 \vee \cdots \vee \neg t_n \vee \neg c_1 \vee \cdots \vee \neg c_m \vee x_h \vee \neg r_h$. They contain either $\neg q$ or $\neg r_h$. These literals are only resolved out by clauses containing their negations q and r_h. No clause contains q and the only clause that contains r_h is $r_h \vee \neg q$, which contains $\neg q$. If a result of resolution contains x_h it also contains either $\neg q$ or $\neg r_h$. This applies to c because it is a result of resolution.

The same applies if c contains e_h: it also contains either $\neg q$ or $\neg s_i$.

The case of $t_h \in c$ is a bit different. The only two clauses of $A_R \cup A_T \cup A_C \cup A_B$ that contain t_h are $\neg x_h \vee t_h$ and $\neg e_h \vee t_h$. Since both are in A_T and $c \not\in A_T$, they are not c. The first clause $\neg x_h \vee t_h$ only resolves with $x_i \vee \neg q$ or $\neg t_1 \vee \cdots \vee \neg t_n \vee \neg c_1 \vee \cdots \vee \neg c_m \vee x_h \vee \neg r_h$, but resolving with the latter generates a tautology. The result of resolving $\neg x_h \vee t_h$ with $x_i \vee \neg q$ is $t_h \vee \neg q$; no clause contains q. Therefore, c can only be $t_h \vee \neg q$. The second clause $\neg e_h \vee t_h$ leads to the same conclusion.

In summary, c contains either $x_h \vee \neg q$, $x_h \vee \neg r_i$, $e_h \vee \neg q$, $e_h \vee \neg s_i$ or $t_h \vee \neg q$. In all these cases it contains at least two literals. This is the case for every index h; therefore, A_N contains at least n clauses of two literals. Every minimal CNF formula equivalent to $A_R \cup A_T \cup A_C \cup A_B$ has size at least $2 \times n$ plus the size of $A_T \cup A_C \cup A_B$. This sum is exactly k. This proves that every minimal CNF formula expressing forgetting contains at least k literal occurrences. Worded differently, every CNF formula expressing forgetting has size at least k.

Formula F is unsatisfiable

The claim is that no CNF formula of size k expresses forgetting if F is unsatisfiable. This is proved by deriving a contradiction from the assumption that such a formula exists.

It has been proved that every CNF formula expressing forgetting is equivalent to $A_R \cup A_T \cup A_C \cup A_B$ and that the minimal equivalent CNF formulae are $A_N \cup A_T \cup A_C \cup A_B$ for some set A_N that contains clauses that include either $x_h \vee \neg q$, $x_h \vee \neg r_i$, $e_h \vee \neg q$, $e_h \vee \neg s_i$ or $t_h \vee \neg q$ for each index h.

If A_N contains other clauses, or more than one clause for each h, or these clauses contain other literals, the size of $A_N \cup A_T \cup A_C \cup A_B$ is larger than $k = 2 \times n + ||A_T|| + ||A_C|| + ||A_B||$, contradicting the assumption. This proves that every formula of size k that is equivalent to $A_R \cup A_T \cup A_C \cup A_B$ is equal to $A_N \cup A_T \cup A_C \cup A_B$ where A_N contains exactly one clause among $x_h \vee \neg q$, $x_h \vee \neg r_i$, $e_h \vee \neg q$, $e_h \vee \neg s_i$ or $t_h \vee \neg q$ for each index h.

The case $x_h \vee \neg r_h \in A_N$ is excluded. It would imply $A_R \cup A_T \cup A_C \cup A_B \models x_h \vee \neg r_h$, which implies the redundancy of $\neg t_1 \vee \cdots \vee \neg t_n \vee \neg c_1 \vee \cdots \vee \neg c_m \vee x_h \vee \neg r_h \in A_B$ contrary to its previously proved superirredundancy. A similar argument proves $e_h \vee \neg s_h \not\in A_N$.

The conclusion is that every formula of size k that is equivalent to $A_R \cup A_T \cup A_C \cup A_B$ is equal to $A_N \cup A_T \cup A_C \cup A_B$ where A_N contains exactly one clause among $x_h \vee \neg q$, $e_h \vee \neg q$, $t_h \vee \neg q$ for each index h.

If F is unsatisfiable all such formulae are proved to be satisfied by a model that falsifies $A_R \cup A_T \cup A_C \cup A_B$, contrary to the assumed equivalence.

Let M be the model that assigns $q = \mathsf{true}$ and $t_i = \mathsf{true}$, and assigns $x_i = \mathsf{true}$ and $e_i = \mathsf{false}$ if $x_i \vee \neg q \in A_N$ and $x_i = \mathsf{false}$ and $e_i = \mathsf{true}$ if $e_i \vee \neg q \in A_N$ or $t_i \vee \neg q \in A_N$. All clauses of A_N and A_T are satisfied by M.

This model M can be extended to satisfy all clauses of $A_C \cup A_B$. Since F is unsatisfiable, M falsifies at least a clause $f_j \in F$. Let M' be the model obtained by extending M with the assignments of c_j to false, all other variables in C to true and all variables r_i and s_i to true. This extension satisfies all clauses of A_B either because it sets c_j to false or because it sets r_i and s_i to true. It also satisfies all clauses of A_C that do not contain c_j because it sets all variables of C but c_j to true.

The only clauses that remain to be proved satisfied are the clauses of A_C that contain c_j. They are $\neg x_i \vee c_j$ for all $x_i \in f_j$ and $\neg e_i \vee c_j$ for all $\neg x_i \in f_j$. Since M' falsifies f_j, it falsifies every $x_i \in f_j$; therefore, it satisfies $\neg x_i \vee c_j$. Since M' falsifies f_j, it falsifies every $\neg x_i \in f_j$; since by construction it assigns e_i opposite to x_i, it falsifies e_i and therefore satisfies $\neg e_i \vee c_j$.

This proves that M' satisfies $A_N \cup A_T \cup A_C \cup A_B$. It does not satisfy $A_R \cup A_T \cup A_C \cup A_B$. If $x_1 \vee \neg q \in A_N$ then M' sets x_1 to true and e_1 to false; therefore, it does not satisfy $e_1 \vee \neg q \in A_R$. Otherwise, M' sets x_1 to false and e_1 to true; therefore, it does not satisfy $x_1 \vee \neg q$.

This contradicts the assumption that $A_N \cup A_T \cup A_C \cup A_B$ is equivalent to $A_R \cup A_T \cup A_C \cup A_B$. The assumption that it has size k is therefore false. □

The problem of size after forgetting is the target of both a reduction from propositional satisfiability and from propositional unsatisfiability. This alone proves it both NP-hard and coNP-hard. The form of the reductions allows them to be merged into a single D^p-hardness proof. The key is that in both cases the minimal formula contains exactly k literal occurrences or is larger. Reductions where the size could be less than k do not work.

Lemma 22 *Checking whether forgetting some variables from a minimal-size Horn formula is expressed by a CNF or Horn formula bounded by a certain size is* D^p*-hard.*

Proof. For every CNF formula F, Lemma 20 ensures the existence of a minimal-size Horn formula A, a set of variables X_A and an integer k such that forgetting all variables but X_A from A is expressed by a Horn formula of size k if F is unsatisfiable and is only expressed by larger CNF formulae otherwise.

For every CNF formula G, Lemma 21 ensures the existence of a minimal-size Horn formula B, a set of variables X_B and an integer l such that forgetting all variables but X_B from B is expressed by a Horn formula of size l if G is satisfiable and is only expressed by larger CNF formulae otherwise.

The prototypical D^p-hard problem is that of establishing whether a formula F is satisfiable and another G is unsatisfiable. If the alphabets of the two formulae G and F are not disjoint they can be made so by renaming one of them to fresh variables because renaming does not affect satisfiability. The same applies to the formulae B and A respectively build from them according to Lemma 20 and Lemma 21 because renaming does not change the minimal size of forgetting either. Lemma 4 proves that $A \cup B$ can be minimally expressed by $C \cup D$ where

C minimally expresses forgetting from A and D from B. The size of these two formulae are l and k if G is unsatisfiable and F satisfiable. If G is satisfiable then D is larger than k while C is still large at least l; the minimal expression of forgetting $A \cup B$ is therefore strictly larger than $k + l$. The same happens if F is unsatisfiable.

This proves that the problem of checking the satisfiability of a formula and the unsatisfiability of another reduces to the problem of checking the size of the minimal expression of forgetting from Horn formulae. □

Proving hardness takes most of this section, but still leaves a gap between the complexity lower bound it shows and the upper bound in the next theorem. The problem is D^p-hard, which is just a bit above NP-hardness and coNP-hardness, but belongs to a class of the next level of the polynomial hierarchy: Σ_2^p.

Theorem 3 *Checking whether forgetting some variables from a Horn formula is expressed by a CNF or Horn formula bounded by a certain size expressed in unary is* D^p*-hard and in* Σ_2^p*, and remains hard even if the formula is restricted to be of minimal size.*

Proof. The problem belongs to Σ_2^p because it can be expressed as the existence of a formula of the given size or less that expresses forgetting the given variables from the formula. In turn, expressing forgetting is by Corollary 2 the same as the equiconsistency with a set of literals containing all variables not to be forgotten. This condition can be expressed by the following metaformula where A is the formula, Y are the variables not to be forgotten and k the size bound.

$$\exists B \; . \; ||B|| \leq k \text{ and } \forall S \; . \; Var(S) \subseteq Y \Rightarrow S \cup A \not\models \bot \Leftrightarrow S \cup B \not\models \bot$$

Both B and S are bounded in size: the first by k, the second by the number of variables in Y. Since consistency is polynomial for Horn formulae, this is a $\exists\forall$QBF, which proves membership to Σ_2^p.

Hardness for D^p is proved by Lemma 22. □

The assumption that k is represented in unary is technical. When formulated as a decision problem, the size of forgetting is the question of whether forgetting from a formula A is expressed by a formula of size k, but the actual problem is to find such a formula. If k is exponential in the size of A, a formula of size k may very well exist but is unpractical to represent. Unless A is very small. The requirement that k is in unary forces the input of the problem to be as large as the expected output. If the available space is enough for storing a resulting formula of size k, it is also enough for storing an input string of length k, which k in unary is. In the other way around, representing k in unary witnesses the ability of storing a resulting formula of size k. The similar assumption "k is polynomial in the size of A" fails to include the case where A is very small but the space available for expressing forgetting is large.

The problem is D^p-hard, and belongs to Σ_2^p. Theorem 3 leaves a gap between the lower bound of D^p and the upper bound of Σ_2^p. According to what proved so far, the problem could be as easy as D^p-complete or as hard as Σ_2^p-complete. Nothing in the results obtained so far favors either possibility. Actually, nothing indicates for certain that the problem is complete for either class; it could be complete for any class in between, like $\Delta_2^p[\log n]$ or Δ_2^p.

Anecdotal evidence hints that the problem is Σ_2^p-complete. The analogous problems without forgetting for unrestricted formulae kept a gap between NP and Σ_2^p for twenty years before being closed as Σ_2^p-complete [37, 40]. Proving membership was easy; proving hardness was not.

This is a common pattern, not limited to formula minimization: in many cases, hardness is more difficult to prove than memership. Not always, but hardness proofs are often more complicated than membership proofs. The above lemmas are an example: several pages of proof for hardness, ten lines for membership. A proof of Σ_2^p-hardness may very well exists but is just difficult to find. As it was for the problem without forgetting.

All of this is anecdotal. Technically, the complexity of the problem could be anything in between D^p and Σ_2^p.

As a personal opinion, not based on the technical results, the author of this article would bet on the problem being Σ_2^p-complete. The missing proof of Σ_2^p-hardness could be an extension of that of Lemma 21, since both Σ_2^p and NP are based on an existential quantification. The extension of an already difficult proof would be further complicated by the addition of an inner universal quantification.

A way to partly close the issue is to further restrict the Horn case to make the problem to be Δ_2^p-complete. The gap would close to its lower end for such a class of formulae.

6 Size after forgetting, general case

The complexity analysis for general CNF formulae mimics that of the Horn case. Two reductions prove the problem hard for the two basic classes of a level of the polynomial hierarchy. They are merged into a single proof that slightly increases the lower bound. A membership proof for a class of the next level ends the analysis.

The difference is that the level of the polynomial hierarchy is the second instead of the first. The two reductions prove the problem hard for Π_2^p and Σ_2^p. They are merged into a D_2^p-hardness proof. Finally, the problem is located within Σ_3^p.

As for the Horn case, the first lemma proves the problem Π_2^p-hard, but is formulated in terms of the reduction because the reduction is needed to raise the lower bound to D_2^p-hard.

Lemma 23 *There exists a polynomial algorithm that turns a CNF formula F into a minimal-size CNF formula A, a subset $X_C \subseteq Var(A)$ and a number k such that forgetting all variables from A but X_C is expressed by a CNF formula of size k if $\forall X \exists Y.F$ is valid and only by CNF formulae of size $k+2$ or greater otherwise.*

Proof. Let $F = \{f_1, \ldots, f_m\}$ and $X = \{x_1, \ldots, x_n\}$. Checking the validity of $\forall X \exists Y.F$ remains Π_2^p-hard even if F is satisfiable: if F is not satisfiable, $\forall X \exists Y.F$ can be turned into the equivalent formula $\forall X \cup \{s\} \exists Y.s \vee F$, and $s \vee F$ is satisfiable.

The reduction is based on an extended alphabet with the additional fresh variables $E = \{e_1, \ldots, e_n\}$, $D = \{d_1, \ldots, d_m\}$ and $\{a, b, q, r\}$. The formula A, the set of variables X_C and the number k are:

$$\begin{aligned} A \;=\; & \{f_j \vee c_j \vee q \mid f_j \in F\} \cup \\ & \{\neg c_j \vee r \mid f_j \in F\} \cup \\ & \{\neg r \vee \neg a \vee b \vee q\} \cup \end{aligned}$$

$$\begin{aligned} & \{a \vee \neg b \vee q\} \cup \\ & \{x_i \vee e_i \mid x_i \in X\} \\ X_C &= X \cup E \cup \{a, b, q\} \\ k &= 2 \times n + 3 \end{aligned}$$

A short explanation of how the reduction works precedes its formal proof. The key is how a model over $X \cup \{q\}$ extends to a model of A, in particular its possible values of a and b. All models over $X \cup \{q\}$ that satisfy q can be extended to satisfy A: all clauses not containing q are satisfied by setting $r = \mathsf{true}$ and e_i opposite to x_i; satisfaction is not affected by the values a and b. The remaining models set $q = \mathsf{false}$. For these models, the truth of a clause f_j makes $f_j \vee c_j \vee q$ satisfied even if $c_j = \mathsf{false}$. In turn, $c_j = \mathsf{false}$ satisfies $\neg c_j \vee r$ even if $r = \mathsf{false}$, which satisfies $\neg r \vee \neg a \vee b \vee q$ regardless of the values of a and b; the values of a and b only need to satisfy $a \vee \neg b \vee q$. Otherwise, the falsity of f_j imposes $c_j = \mathsf{true}$ to satisfy $f_j \vee c_j \vee q$, which makes $\neg c_j \vee r$ require $r = \mathsf{true}$, which turns $\neg r \vee \neg a \vee b \vee q$ into $\neg a \vee b \vee q$, making the literals $\neg a$ and b necessary in addition to a and $\neg b$.

The proof comprises four steps: first, A is proved minimal as required by the claim of the lemma; second, k literals that are in every formula that expresses forgetting regardless of the validity of the QBF are identified; third, a formula of size k expressing forgetting when the QBF is valid is determined; fourth, every formula expressing forgetting contains at least two further literals if the QBF is invalid.

Minimality of A.

Follows from Lemma 10 since all clauses of A are superirredundant. This is in turn proved by showing substitutions that disallow all resolutions, which proves the superredundancy of the remaining clauses by Lemma 17 and Lemma 13.

The substitution that replaces with true the variables a, b, r, all e_i and all c_j with $j \neq h$ removes all clauses but $f_h \vee c_h \vee q$, which is therefore superirredundant.

The clauses $\neg c_j \vee r$ are proved superirredundant by substituting q and e_i with true, which removes all other clauses. The clauses $\neg c_j \vee r$ do not resolve because they do not contain opposite literals.

Two other clauses are proved superirredundant by the substitution that replaces all e_i with true, all c_j with false, and $X \cup Y$ with some values that satisfy F; such values exist because F is by assumption satisfiable. This substitution removes all clauses but $\neg r \vee \neg a \vee b \vee q$ and $a \vee \neg b \vee q$, which only resolve in tautologies.

Finally, the clauses $x_h \vee e_h$ are proved superirredundant by replacing q and r with true, which removes all other clauses. Since the clauses $x_h \vee e_h$ only contain positive literals, they do not resolve.

Necessary literals.

Regardless of the validity of $\forall X \exists Y.F$, the literals $X \cup E \cup \{a, \neg b, q\}$ are necessary in every CNF formula that expresses forgetting all variables but X_C from A. This is proved by Lemma 6, exhibiting a set of literals S such that $S \cup A$ is consistent but $S \backslash \{l\} \cup \{\neg l\} \cup A$ is not for every $l \in X \cup E \cup \{a, \neg b, q\}$.

The first set is $S = \{x_i, \neg e_i, a, b, \neg q\}$, which is consistent with A because of the model that satisfies S and assigns r and all variables c_j to true. Changing x_i to $\neg x_i$ violates the clause $x_i \vee e_i$. Changing a to false violates $a \vee \neg b \vee q$. This proves that a and all variables x_i are necessary by Lemma 6.

The second set is $S = \{\neg x_i, e_i, \neg a, \neg b, \neg q\}$, which is consistent with A because of the model that satisfies S and assigns r and all variables c_j to true. Changing e_i to $\neg e_i$ violates $x_i \vee e_i$, changing b to true violates $a \vee \neg b \vee q$. This proves that e_i and $\neg b$ are necessary by Lemma 6.

The third set is $\{S = x_i, e_i, \neg a, b, q\}$, which is consistent with A because of the model that satisfies S and assigns r and all variables c_j to true. Changing q to false violates the clause $a \vee \neg b \vee q$, proving that q is necessary.

In summary, all literals in $X \cup E \cup \{a, \neg b, q\}$ are in every CNF formula expressing forgetting all variables but X_C from A. These literals are $2 \times n + 3$. This is a part of the claim: no CNF formula expressing forgetting is smaller than $2 \times n + 3$.

Forgetting when $\forall X \exists Y.F$ is valid

If $\forall X \exists Y.F$ is valid, forgetting is expressed by $B = \{a \vee \neg b \vee q\} \cup \{x_i \vee e_i \mid x_i \in X\}$, which has the required size $k = 2 \times n + 3$ and variables $XC = X \cup E \cup \{a, b, q\}$. Corollary 2 proves that this formula expresses forgetting: every set S of literals of X_C that contains all variables of X_C is consistent with B if and only if it is consistent with A.

Since B only contains clauses of A, every set of literals S that is consistent with A is also consistent with B. The claim follows from proving the converse for every set of literals S over X_C that contains all variables of X_C.

The assumption is that $S \cup B$ is consistent; the claim is that $S \cup A$ is consistent. Since $S \cup B$ is consistent, it has a model M. Let M_X be its restriction to the variables X and M'_Y to Y. By assumption, $\forall X \exists Y.F$ is valid. Therefore, $M_X \cup M_Y$ satisfies F for some truth evaluation M_Y over Y. Since S is satisfied by M and does not contain any variable Y, it is also satisfied by $M \backslash M'_Y \cup M_Y$. The truth evaluation $M_C = \{c_j = \mathsf{false} \mid f_j \in F\} \cup \{r = \mathsf{false}\}$ satisfies all clauses $\neg c_j \vee r$ and $\neg r \vee \neg a \vee b \vee q$. Since $M_X \cup M_Y$ satisfies all clauses $f_j \in F$, the union $M \backslash M'_Y \cup M_Y \cup M_C$ satisfies all clauses $f_j \vee c_j \vee q$ of A. This proves that $M \backslash M'_Y \cup M_Y \cup M_C \cup M_O$ satisfies all clauses of A that B does not contain.

Forgetting when $\forall X \exists Y.F$ is invalid

All CNF formulae that express forgetting have been proved to contain $X \cup E \cup \{a, \neg b, q\}$. If $\forall X \exists Y.F$ is invalid, they all contain $\neg a$ and b as well.

This is proved by Lemma 6: a set of literals S over X_C is shown to be consistent with A while $S \backslash \{\neg a\} \cup \{a\}$ is not. A similar set is shown for b.

Since $\forall X \exists Y.F$ is invalid, for some model M_X over X the model $M_X \cup M_Y$ falsifies F for every model M_Y over Y. The required set S is built from M_X: it contains the literals over x_i that are satisfied by M_X and $\neg a$, $\neg b$ and $\neg q$.

$$S = \{x_i \mid M_X \models x_i\} \cup \{\neg x_i \mid M_X \models \neg x_i\} \cup \{\neg a, \neg b, \neg q\}$$

By construction, M_X satisfies the first part of S. The model $M_O = \{a = \mathsf{false}, b = \mathsf{false}, q = \mathsf{false}\}$ satisfies the second. Therefore, $M_X \cup M_O$ satisfies S.

The consistency of $S \cup A$ is shown by proving that $M_X \cup M_O$ can be extended to the other variables to satisfy A. This extension is $M_X \cup M_Y \cup M_O \cup M_N \cup M_C$, where M_Y is an arbitrary model over Y, M_N assigns every e_i opposite to x_i in M_X and M_C is $\{c_j = \mathsf{true} \mid$

$f_j \in F\} \cup \{r = \textsf{true}\}$. The clauses $f_j \vee c_j \vee q$ are satisfied because c_j is true, the clauses $\neg c_j \vee r$ because r is $\textsf{true}$, the clause $\neg r \vee \neg a \vee b \vee q$ because a is $\textsf{false}$, $a \vee \neg b \vee q$ because b is $\textsf{false}$, the clauses $x_i \vee e_i$ because $M_N \models e_i$ if $M_X \not\models x_i$.

This proves that $M_X \cup M_O \cup M_N \cup M_C$ satisfies $S \cup A$, which is therefore satisfiable.

The claim is a consequence of $S' = S \backslash \{\neg a\} \cup \{a\}$ being inconsistent with A.

$$S' = \{x_i \mid M_X \models x_i\} \cup \{\neg x_i \mid M_X \models \neg x_i\} \cup \{a, \neg b, \neg q\}$$

This is proved by contradiction: a model M' is assumed to satisfy $S' \cup A$. Since M' satisfies S', it assigns the variables x_i the same as M_X. Let M_Y be the restriction of M' to the variables Y. By assumption, M_X is a model over X that cannot be extended to Y to satisfy F. As a result, $M_X \cup M_Y \not\models F$. Therefore, M' falsifies at least a clause $f_j \in F$. Since M' satisfies $f_j \vee c_j \vee q$ but falsifies both f_j and q, it satisfies c_j. It also satisfies r because it satisfies $\neg c_j \vee r$ and falsifies c_j. Since M' satisfies S' it satisfies a and falsifies b and q. The conclusion is that all literals of $\neg r \vee \neg a \vee b \vee q \in A$ are false, contrary to the assumption that M' satisfies A.

A similar set S with a and b in place of $\neg a$ and $\neg b$ proves that expressing forgetting also requires b. □

The second lemma is again about a reduction. Its statement implies that the problem is Σ_2^p-hard, but it predicates about the reduction rather than the hardness. This allows it to be merged with the first lemma into a proof of D_2^p-hardness. The existing proof of Σ_2^p-hardness of the problem without forgetting [40] also proves the problem with forgetting Σ_2^p-hard, but does not allow such a merging and does not hold in the restriction of minimal formulae.

Lemma 24 *There exists a polynomial algorithm that turns a DNF formula $F = f_1 \vee \cdots \vee f_m$ over variables $X \cup Y$ into a minimal-size CNF formula A, a subset $X_C \subseteq Var(A)$ and a number k such that forgetting all variables but X_C from A is expressed by a CNF formula of size k if $\exists X \forall Y.F$ is valid, and only by larger CNF formulae otherwise.*

Proof. Let $F = f_1 \vee \cdots \vee f_m$ be the DNF formula over variables $X \cup Y$. The reduction employs additional variables: $O = \{o_i \mid x_i \in X\}$, $E = \{e_i \mid x_i \in X\}$, $P = \{p_i \mid x_i \in X\}$, $T = \{t_i \mid x_i \in X\}$, $D = \{d_j \mid f_j \in F\}$, $R = \{r_i \mid x_i \in X\}$, $S = \{s_i \mid x_i \in X\}$ and q. The formula A, the alphabet X_C and the number k are as follows.

$$\begin{array}{rcl}
A & = & A_F \cup A_T \cup A_D \cup A_B \\
A_F & = & \{x_i \vee \neg o_i, o_i \vee q \mid x_i \in X\} \cup \{e_i \vee \neg p_i, p_i \vee q \mid x_i \in X\} \\
A_T & = & \{\neg x_i \vee t_i,\ \neg e_i \vee t_i \mid x_i \in X\} \\
A_D & = & \{(\neg f_j[e_i/\neg x_i]) \vee d_j \mid f_j \in F\} \\
A_B & = & \{\neg t_1 \vee \cdots \vee \neg t_n \vee \neg d_j \vee x_i \vee \neg r_i, r_i \vee q \mid x_i \in X\ , f_j \in F\} \cup \\
 & & \{\neg t_1 \vee \cdots \vee \neg t_n \vee \neg d_j \vee e_i \vee \neg s_i, s_i \vee q \mid x_i \in X,\ f_j \in F\} \\
X_C & = & X \cup E \cup Y \cup T \cup D \cup R \cup S \cup \{q\} \\
k & = & 2 \times n + ||A_T \cup A_D \cup A_B||
\end{array}$$

The reduction works because every minimal CNF formula that expresses forgetting contains at least one among $x_h \vee q$, $e_h \vee q$ and $t_h \vee q$ for each h, and all of $A_T \cup A_D \cup A_B$. This proves the lower bound k. If the QBF is valid, for some evaluation over X the formula F is true regardless of Y. Choosing the clauses $x_h \vee q$, $e_h \vee q$ or $t_h \vee q$ that correspond to this model, some clause A_D imply d_j, which allows A_B to entail all remaining clauses. If the QBF is not valid, no literal d_j is entailed.

The formal proof requires five steps: first, every formula expressing forgetting is equivalent to a certain formula $A_R \cup A_T \cup A_D \cup A_B$; second, A is a minimal CNF formula and the clauses of $A_T \cup A_D \cup A_B$ are in all minimal CNF formulae equivalent to $A_R \cup A_T \cup A_D \cup A_B$; third, forgetting is expressed by a formula of size k if the QBF is valid; fourth, every minimal CNF formula expressing forgetting contains either $x_h \vee q$, $e_h \vee q$ or $t_h \vee q$ for each h; fifth, if the QBF is invalid then forgetting is only expressed by formulae larger than k.

Effect of forgetting.

The variables to forget are $O \cup P$. Each is contained only in two clauses of A, with opposite signs. Resolving them produces the clauses in the following set A_R.

$$A_R = \{x_i \vee q, e_i \vee q \mid x_i \in X\}$$

By Theorem 1, forgetting is expressed by $A_R \cup A_T \cup A_D \cup A_B$. Therefore, all formulae that express forgetting are equivalent to this formula.

Superirredundancy.

All clauses of $A_F \cup A_T \cup A_D \cup A_B$ are proved superirredundant in $A_F \cup A_R \cup A_T \cup A_D \cup A_B$. Both A and $A_R \cup A_T \cup A_D \cup A_B$ are subsets of this formula; therefore, the superirredundant clauses are superirredundant in both formulae by Lemma 16. Since A comprises exactly them, it is minimal thanks to Lemma 10. Since all formulae expressing forgetting are equivalent to $A_R \cup A_T \cup A_D \cup A_B$, where $A_T \cup A_D \cup A_B$ are superirredundant, these clauses are in all formulae expressing forgetting.

Superirredundancy is proved applying a substitution to the formula so that the resulting clauses do not resolve and are not contained in one another. This condition proves them superirredundant by Lemma 13. Lemma 17 implies their superirredundancy in the original formula.

Replacing all variables X, E, T and D with true removes from the formula $A_F \cup A_R \cup A_T \cup A_D \cup A_B$ all clauses but $o_i \vee q$, $p_i \vee q$, $r_i \vee q$ and $s_i \vee q$. These clauses do not resolve because they only contain positive literals. None is contained in another.

Replacing all variables R and S with false and all variables T, D and q with true removes from the formula $A_F \cup A_R \cup A_T \cup A_D \cup A_B$ all clauses but the clauses $x_i \vee \neg o_i$ and $e_i \vee \neg p_i$. They are not contained in one another; they do not resolve because they do not contain opposite literals.

Replacing all variables O, P, R and S with false and D and q with true removes all clauses but $\neg x_i \vee t_i$ and $\neg e_i \vee t_i$. These clauses do not resolve because they do not contain opposite literals; they are not contained in one another.

Replacing all variables O, P, R and S with false and T, $D \backslash \{d_h\}$ and q with true removes all clauses but $(\neg f_h[e_i/\neg x_i]) \vee d_h$, which is therefore superirredundant.

The last substitution replaces all variables $X \backslash \{x_h\}$, E, O, P, $R \backslash \{r_h\}$ and S with false, all variables $D \backslash \{d_l\}$ and q with true, all variables y_i such that $y_i \in \neg f_l[e_i/\neg x_i]$ to true and all such that $\neg y_i \in \neg f_l[e_i/\neg x_i]$ to false. This substitution removes all clauses but $\neg x_h \vee t_h$,

$\neg t_1 \vee \cdots \vee \neg t_n \vee \neg d_l \vee x_h \vee \neg r_i$ and possibly $(\neg f_l[e_i/\neg x_i]) \vee d_l$. The latter clause is removed if it contains some variable y_i. It is removed if it contains some literal $\neg x_i$ with $i \neq h$. It is removed if it contains some literal $\neg e_i$. The only other literals it may contain are $\neg x_h$ and d_l; it contains both: d_l by construction, $\neg x_h$ because otherwise f_l would be empty. The remaining clauses are therefore $\neg x_h \vee t_h$, $\neg t_1 \vee \cdots \vee \neg t_n \vee \neg d_l \vee x_i \vee \neg r_i$ and possibly $\neg x_h \vee d_h$. These clauses only resolve in tautologies, which proves the second superirredundant. A similar argument holds for $\neg t_1 \vee \cdots \vee \neg t_n \vee \neg d_l \vee e_i \vee \neg s_i$.

Validity of $\exists X \forall Y . F$.

Let M be a model over variables X that makes F true regardless of the values of Y. Let $A'_R \subseteq A_R$ be the set of clauses $x_i \vee q$ such that $M \models x_i$ and $e_i \vee q$ such that $M \models \neg x_i$. This set has size $2 \times n$. Therefore, $A'_R \cup A_T \cup A_D \cup A_B$ has size $k = 2 \times n + ||A_T \cup A_D \cup A_B||$. This formula expresses forgetting if it is equivalent to $A_R \cup A_T \cup A_D \cup A_B$, which is the case if $A'_R \cup A_T \cup A_D \cup A_B \models A_R$. The claim is proved by showing that $A'_R \cup A_T \cup A_D \cup A_B$ entails A_R.

Either $x_i \vee q$ or $e_i \vee q$ is in A'_R for every i and these clauses respectively resolve with $\neg x_i \vee t_i$ and $\neg e_i \vee t_i$, producing $t_i \vee q$ in both cases. Each clause $\neg t_1 \vee \cdots \vee \neg t_n \vee \neg d_j \vee x_i \vee \neg r_i$ resolve with them and with $r_i \vee q$ to $\neg d_j \vee x_i \vee q$. This clause further resolves with $(\neg f_j[e_i/\neg x_i]) \vee d_j$ to produce $(\neg f_j[e_i/\neg x_i]) \vee x_i \vee q$. This proves that $A'_R \cup A_T \cup A_D \cup A_B$ implies every clause $(\neg f_j[e_i/\neg x_i]) \vee x_i \vee q$ with $f_j \in F$. The following equivalence holds.

$$\begin{aligned}
\{\neg f_j[e_i/\neg x_i] \vee x_i \vee q | f_j \in F\} &\equiv \left(\bigwedge \{\neg f_j[e_i/\neg x_i] \mid f_j \in F\}\right) \vee x_i \vee q \\
&\equiv \neg \left(\bigvee \{f_j[e_i/\neg x_i] \mid f_j \in F\}\right) \vee x_i \vee q \\
&\equiv \neg F[e_i/\neg x_i] \vee x_i \vee q
\end{aligned}$$

Since $A'_R \cup A_T \cup A_D \cup A_B$ implies the first set, it implies the last formula: $A'_R \cup A_T \cup A_D \cup A_B \models \neg F[e_i/\neg x_i] \vee x_i \vee q$.

Since M satisfies F regardless of Y, it follows that $\{x_i \mid M \models x_i\} \cup \{\neg x_i \mid M \models \neg x_i\} \models F$. Replacing each $\neg x_i$ with e_i in both sides of this entailment turns into $\{x_i \mid M \models x_i\} \cup \{e_i \mid M \models \neg x_i\} \models F[e_i/\neg x_i]$. Disjoining both terms with q results into $A'_R \models F[e_i/\neg x_i] \vee q$.

This entailment and the previously proved $A'_R \cup A_T \cup A_D \cup A_B \models \neg F[e_i/\neg x_i] \vee x_i \vee q$ imply $A'_R \cup A_T \cup A_D \cup A_B \models x_i \vee q$.

The same holds for $e_i \vee q$ by symmetry. Therefore, $A'_R \cup A_T \cup A_D \cup A_B$ implies every clause of A_R.

Necessary clauses.

All formulae that express forgetting are equivalent to $A_R \cup A_T \cup A_D \cup A_B$ and therefore contain all its superirredundant clauses $A_T \cup A_D \cup A_B$. As a result, they have the form $A_N \cup A_T \cup A_D \cup A_B$ for some set of clauses A_N. It is now shown that all equivalent CNF formulae of minimal size contain a clause that include either $x_h \vee q$, $x_h \vee \neg r_h$, $e_h \vee q$, $e_h \vee \neg s_h$, or $t_h \vee q$ for each h.

Since $A_N \cup A_T \cup A_D \cup A_B$ is equivalent to $A_R \cup A_T \cup A_D \cup A_B$, it entails $x_h \vee q \in A_R$. This clause is not satisfied by the following model.

$$M = \{x_i = e_i = t_i = \mathsf{true} \mid i \neq h\} \cup \{x_h = e_h = t_h = \mathsf{false}\} \cup$$

$$\{d_j = \mathsf{true} \mid f_j \in F\} \cup \{r_i = s_i = \mathsf{true}\} \cup \{q = \mathsf{false}\}$$

This model satisfies all clauses of $A_T \cup A_D \cup A_B$. If A_N also satisfied it, $A_N \cup A_T \cup A_D \cup A_B$ would have a model that falsifies $x_h \vee q$, which it instead entails. As a result, A_N contains a clause c that M falsifies. Since $A_N \cup A_T \cup A_D \cup A_B$ is a formula of minimal size, it entails no proper subset of c. By equivalence, the same applies to $A_R \cup A_T \cup A_D \cup A_B$.

$$\begin{aligned}
&M \not\models c\\
&A_R \cup A_T \cup A_D \cup A_B \models c\\
&A_R \cup A_T \cup A_D \cup A_B \models c' \text{ implies } c' \not\subset c
\end{aligned}$$

If c contains neither x_h, e_h nor t_h, it would still be falsified by the model that is the same as M except that it assigns x_h, e_h and t_h to true. This model satisfies $A_R \cup A_T \cup A_D \cup A_B$. As a result, $A_R \cup A_T \cup A_D \cup A_B \cup \neg(c)$ is consistent, contradicting $A_R \cup A_T \cup A_D \cup A_B \models c$. This proves that c contains either x_h, e_h or t_h.

Since these three variables are negative in M and $M \not\models c$, they are positive in c. In other words, c contains either x_h, e_h or t_h unnegated.

Since c is entailed by $A_R \cup A_T \cup A_D \cup A_B$ but none of its proper subsets does, it follows from resolution: $A_R \cup A_T \cup A_D \cup A_B \vdash c$.

If c contains x_h, it also contains either q or $\neg r_h$. This is proved as follows. Since c is the tree of a resolution tree and contains x_h, this literal is also in one of the leaves of resolution. The only clauses of $A_R \cup A_T \cup A_D \cup A_B$ containing x_h are $x_h \vee q$ and all clauses $\neg t_1 \vee \cdots \vee \neg t_n \vee \neg d_j \vee x_h \vee \neg r_h$. The first does not resolve over q because the formula does not contain $\neg q$. The other clauses only resolve over r_h with $r_h \vee q$, which introduces q, which again cannot be removed by resolution. Since c is obtained by resolution, if it contains x_h it also contains either $\neg r_h$ or q.

By symmetry, if c contains e_h it also contains either $\neg s_h$ or q.

The other case is that c contains t_h. The only clauses of $A_R \cup A_T \cup A_D \cup A_B$ that contain t_h are $\neg x_h \vee t_h$ and $\neg e_h \vee t_h$. These clauses are satisfied by M while c is not, therefore c is not one of them. The first clause $\neg x_h \vee t_h$ only resolves over x_h with $x_h \vee q$ and all clauses $\neg t_1 \vee \cdots \vee \neg t_n \vee \neg d_j \vee x_h \vee \neg r_h$, but resolving with the latter only generates tautologies. Therefore, the first step of resolution is necessarily $\neg x_h \vee t_h, x_h \vee q \vdash t_h \vee q$. Since the formula does not contain q, every clause obtained from resolution that contains t_h also contains q. This also includes c. The same holds for symmetry for $\neg e_h \vee t_h$.

This proves that every minimal-size CNF formula expressing forgetting contains a clause that includes either $x_h \vee q$, $x_h \vee \neg r_h$, $e_h \vee q$, $e_h \vee \neg s_h$, or $t_h \vee q$ for each h.

Falsity of $\exists X \forall Y.F$.

The falsity of $\exists X \forall Y.F$ contradicts the existence of a minimal-size CNF formula of size k expressing forgetting. The relevant results proved so far are: every CNF formula expressing forgetting has size k or more and is equivalent to $A_R \cup A_T \cup A_D \cup A_B$; the minimal-size such formulae are $A_N \cup A_T \cup A_D \cup A_B$ where A_N contains a clause that includes either $x_h \vee q$, $x_h \vee \neg r_h$, $e_h \vee q$, $e_h \vee \neg s_h$, or $t_h \vee q$ for each h.

A formula $A_N \cup A_T \cup A_D \cup A_B$ of size k expressing forgetting, if any, is minimal since no smaller formula expresses forgetting. Therefore, A_N includes a clause containing one of the five disjunctions for each h. Since these are not in $A_T \cup A_D \cup A_B$, the size of such formulae is $k = 2 \times n + ||A_T \cup A_D \cup A_B||$ if every clause of A_N is exactly one of the above disjunction for each h. If A_N contains more than one clause for some h or the clause for some h contains more than two literals or A_N contains other clauses, the formula is not minimal.

The case $x_h \vee \neg r_h \in A_N$ can be excluded: it makes $\neg t_1 \vee \cdots \vee \neg t_n \vee \neg d_j \vee x_h \vee \neg r_h \in A_N$ redundant in $A_N \cup A_T \cup A_D \cup A_B$, contradicting the minimality of this formula. The case $e_h \vee \neg s_h \in A_N$ is excluded in the same way.

These exclusion leaves A_N to contain exactly one among $x_h \vee q$, $e_h \vee q$, and $t_h \vee q$ for each h and nothing else.

The final step of the proof is that no such A_N makes $A_N \cup A_T \cup A_D \cup A_B$ equivalent to $A_R \cup A_T \cup A_D \cup A_B$ if $\exists X \forall Y.F$ is invalid. Nonequivalence is proved by exhibiting a model of the first formula that does not satisfy the second.

Let M_X be the model over X that contains $x_i = \mathsf{true}$ if $x_i \vee q \in A_N$ and $x_i = \mathsf{false}$ otherwise. Let M_N be the model that assigns every e_i opposite to x_i and $M_T = \{t_i = \mathsf{true}\}$. By construction, $M_X \cup M_N \cup M_T \cup \{q = \mathsf{false}\}$ satisfies all clauses of A_N. It also falsifies either $x_i \vee q$ or $e_i \vee q$ for each i because it assigns false to q and to either x_i or e_i. It therefore falsifies A_R.

Since $\exists X \forall Y.F$ is invalid, every model over X falsifies F with a model over Y. Let M_Y be the model over Y such that $M_X \cup M_Y \models \neg F$. Since $F = f_1 \vee \cdots \vee f_m$, it holds $M_X \cup M_Y \models \neg f_j$ for every $f_j \in F$. It follows $M_X \cup M_Y \cup M_N \models \neg f_j[e_i/\neg x_i]$ since M_N assigns every e_i opposite to x_i.

Merging the results proved in the latter two paragraphs, $M_X \cup M_T \cup \{q = \mathsf{false}\} \cup M_N \cup M_Y$ satisfies both A_N and $\neg f_j[e_i/\neg x_i]$ for every $f_j \in F$.

This model can be extended to a model of $A_N \cup A_T \cup A_D \cup A_B$ by adding $M_O = \{d_j = \mathsf{false}\} \cup \{r_i = \mathsf{true}\} \cup \{s_i = \mathsf{true}\}$. The clauses of A_N are already proved satisfied. The clauses $\neg x_i \vee t_i \in A_T$ are satisfied because M_T contains $t_i = \mathsf{true}$. The clauses $(\neg f_j[e_i/\neg x_i]) \vee d_j$ are satisfied because $\neg f_j[e_i/\neg x_i]$ is. The clauses of A_B are satisfied because each contains either $\neg d_j$, r_i or s_i, and these literals are true in M_O.

This proves that $M_X \cup M_T \cup \{q = \mathsf{false}\} \cup M_N \cup M_Y \cup M_O$ satisfies $A_N \cup A_T \cup A_D \cup A_B$. It does not satisfy A_R, which means that it falsifies $A_R \cup A_T \cup A_D \cup A_B$. This proves that $A_N \cup A_T \cup A_D \cup A_B$ is not equivalent to $A_R \cup A_T \cup A_D \cup A_B$.

In summary, assuming that the QBF is not valid and that a CNF formula of size k expresses forgetting, it is proved that the formula does not express forgetting. This contradiction shows that no formula of size k expresses forgetting if the QBF is not valid. □

As anticipated, the two reductions merge into one that proves the problem of forgetting size D_2^p-hard.

Lemma 25 *Checking whether forgetting a given set of variables from a minimal-size CNF formula is expressed by a CNF formula bounded by a certain size is D_2^p-hard.*

Proof. For every ∀QBF Lemma 23 ensures the existence of a minimal-size CNF formula A, a set of variables X_A and an integer k such that forgetting all variables but X_A from A is expressed by a CNF formula of size k if the QBF is valid and is only expressed by larger CNF formulae otherwise.

For every ∃QBF Lemma 24 ensures the existence of a minimal-size CNF formula B, a set of variables X_B and an integer l such that forgetting all variables but X_B from B is expressed by a CNF formula of size l if the QBF is valid and is only expressed by larger CNF formulae otherwise.

A D_2^p-hard problem is that of establishing whether an $\exists$QBF and a $\forall$QBF are both valid. If their alphabets are not disjoint they can be made so by renaming one of them to fresh variables since renaming does not affect validity. The same applies to the formulae B and A respectively build from them according to Lemma 23 and Lemma 24 because renaming does not change the minimal size of forgetting either. Lemma 4 proves that forgetting from $A \cup B$ is expressed by $C \cup D$ where C expresses forgetting from A and D from B. The minimal size of two such CNF formulae are respectively k and l. If the QBFs are both valid they are exactly k and l large. Otherwise, they are strictly larger than either k or l. The sum is $k + l$ if both QBFs are valid and is larger than $k + l$ otherwise. □

The next theorem adds a complexity class membership to the hardness of the problem of size of forgetting proved in the previous lemma.

Theorem 4 *Checking whether forgetting some variables from a CNF formula is expressed by a CNF formula of a certain size expressed in unary is D_2^p-hard and in Σ_3^p, and remains hard even if the CNF formula is restricted to be of minimal size.*

Proof. Membership to Σ_3^p is proved first. The problem is the existence of a CNF formula of the given size or less that expresses forgetting the given variables from the formula. Corollary 2 reformulates forgetting in terms of equiconsistency with a set of literals containing all variables not to be forgotten. Forgetting withing a certain size is formalized by the following metaformula where A is the formula, Y the variables not to be forgotten and k the size bound.

$$\exists B \,.\, Var(B) \subseteq Y,\ ||B|| \leq k \text{ and } \forall S \,.\, Var(S) \subseteq Y \Rightarrow \exists M \,.\, M \models S \cup A \Leftrightarrow \exists M' \,.\, M' \models S \cup B$$

All four quantified entities are bounded in size: B by k, S and M' by the number of variables in Y and M by the number of variables in A. This is therefore a $\exists\forall\exists$QBF, which proves membership to Σ_3^p.

Hardness to D_2^p is proved by Lemma 25 in the restriction where A is minimal. □

The technical assumption that k is in unary is the same as in Theorem 3; it is motivated after that theorem.

7 Ensuring superirredundancy

As it often happens, things appear magically in proofs. Here is a formula and, what a surprise, it is minimal. By chance, forgetting certain variables is expressed by a formula comprising mostly superirredundant clauses. Many of them are like $x_i \vee \neg o_i$ and $o_i \vee q$, pairs with a shared variable occurring nowhere else — for no particular reason. Except that these clauses turn out superirredundant. Magically.

The proofs are written this way to go straight to their aim: proving that the claim follows from the assumptions. The process of their creation is hidden, only the steps from assumptions to claim remains.

This is often sufficient. The proof is disposable: once concluded, once "Q.E.D." is reached nothing of it matters to the rest of the article. Only the statement of the lemma or theorem remains. Only that is referenced in the following: "by Lemma 21", "as Theorem 5 shows", "thanks to Lemma 14". The proof itself is forgotten. For good reasons: mentioning its internals is like every driver had to know about differential steering to be able to pull over at the grocery store.

Sometimes, some parts of the mostly neglected proofs have their revenge. They turn out to have a sufficient range of applicability to be promoted to the status of lemmas, sometimes even theorems. Superirredundancy proved some formulae minimal; it also proved that the minimal forms of others always contain some clauses. Such properties are not limited to forgetting. They matter when minimizing a formula without forgetting [22, 7]; they matter when minimizing the result of revising a formula [6, 25, 27] or circumscribing it [32]; they matter whenever a formula is produced, and minimizing it makes sense.

Some clauses may be superirredundant out of the box. A formula is created to some aim and, this time really by chance, some of its clauses like $\neg x_i \vee t_i$ are already superirredundant. Some others like $x_i \vee q$ do not have the same luck. They are not superirredundant; or maybe they are, but proving it appears just too complicated. They are ripped apart, their pieces hold together only by a new variable, like $x_i \vee \neg o_i$ and $o_i \vee q$. The original clause $x_i \vee q$ is what really needed; but only its two superirredundant replacements $x_i \vee \neg o_i$ and $\neg o_i \vee q$ appear in the proof.

Does this trick work in general?

Can every clause be made superirredundant just by cutting it in two, like $x_i \vee q$ into $x_i \vee \neg o_i$ and $o_i \vee \neg q$? Not always, but often. If one of the two parts is already superredundant without the holding variable, isolating it is of no help. Otherwise, it is. But this is not the only caveat.

Making a clause superirredundant would be counterproductive if it makes others superredundant. Fix one, break two. The formula would never become minimal this way. Even if only two clauses are required to be superirredundant they may never become both so at the same time if making one superirredundant makes the other superredundant.

Making a clause superirredundant is not enough; preserving the superirredundancy of the others is also a must. This is often the case, but not always. The exception is when another clause resolves with each of the two pieces.

Making a clause superirredundant and preserving the superirredundancy of the others are two requirements. An even more basic one, so basic that it went unnoticed, is that the resulting formula is the same as the original. Clauses could be made superirredundant just by renaming the variables of each on a new alphabet, like turning $x_i \vee q$ into $x'_i \vee q'$; the result is superirredundant because each clause resolves with no other; yet, the resulting formula is just a bunch of unrelated clauses without any of the properties the original may have had.

The new clauses $x_i \vee \neg o_i$ and $o_i \vee q$ resolve into the original $x_i \vee q$. The transformation does not exactly preserve the semantics of the formula because of the new variable o_i. Yet, the resulting formula is almost the same as the original. Except for o_i. All models not including o_i are still (partial) models. All consequences that do not include o_i are still consequences. All properties not involving o_i are preserved. This is forgetting: the original formula expresses forgetting o_i from the generated one. This is always the case. No additional requirement is needed.

In summary, splitting a clause works if:

- the resulting formula is similar enough to the original;

- the split clause is superirredundant;
- the other clauses remain superirredundant.

The first point is formalized as: the original formula expresses forgetting the new variable from the generated formula. The second and the third points have additional requirements, they are not always the case. They are proved in reverse, by showing the consequences of superredundancy.

An example illustrates the method. The first clause is superredundant in the following formula, the first in the `split.py` file of `minimize.py`.

$$\{a \vee b \vee c, \neg a \vee d, \neg c \vee d, \neg d \vee a \vee c\}$$

The last three clauses are the same as $a \vee c \equiv d$. They make $a \vee c$ equivalent to d. Consequently, the first clause $a \vee b \vee c$ is replaceable by $d \vee b$ and therefore superredundant.

If it is required to be superirredundant, it can be made so by splitting it into $a \vee x$ and $\neg x \vee b \vee c$.

$$\{a \vee x, \neg x \vee b \vee c, \neg a \vee d, \neg c \vee d, \neg d \vee a \vee c\}$$

The last three clauses still make $a \vee c$ equivalent to d, but this is no longer a problem because a and c are now separated: a is in $a \vee x$ and c is in $\neg x \vee b \vee c$. The only way to join them back to apply their equivalence to d is to resolve the two parts into $a \vee b \vee c$. This removes x and $\neg x$, which are necessary to derive the two clauses $a \vee x$ and $\neg x \vee b \vee c$ back. All clauses in this formula are superirredundant, as shown by the second formula in the `split.py` file of `minimize.py`.

The first point to prove is that splitting a clause does not change the meaning of the formula. The semantics changes slightly since the original formula does not mention x at all while the modified one does. In the example, $a = \mathsf{false}$, $b = \mathsf{true}$ and $x = \mathsf{false}$ satisfy the original clause $a \vee b \vee c$ but not its part $a \vee x$. The modified formula cannot be equivalent to the original since it contains the new variable. Yet, it is equivalent apart from it. This is what forgetting does: it removes a variable while semantically preserving everything else.

Lemma 26 *Every CNF formula F that contains a clause $c_1 \vee c_2$ and does not mention x expresses forgetting x from $F \backslash \{c_1 \vee c_2\} \cup \{c_1 \vee x, c_2 \vee \neg x\}$.*

Proof. The formula $F \backslash \{c_1 \vee c_2\} \cup \{c_1 \vee x, c_2 \vee \neg x\}$ in the statement of the lemma is denoted F''. Theorem 1 proves that a formula expresses forgetting x from it is $F'' \backslash (F'' \cap x) \backslash (F'' \cap \neg x) \cup \mathrm{resolve}(F'' \cap x, F'' \cap \neg x)$. The claim is proved by showing that this formula is F.

The only clause of F'' containing x is $c_1 \vee x$ and the only clause containing $\neg x$ is $c_2 \vee \neg x$. Therefore, $F'' \cap x$ is $\{c_1 \vee x\}$ and $F'' \cap \neg x$ is $\{c_2 \vee \neg x\}$. The formula that expresses forgetting is therefore $F'' \backslash \{c_1 \vee x\} \backslash \{c_2 \vee \neg x\} \cup \mathrm{resolve}(c_1 \vee x, c_2 \vee \neg x)$, which is equal to $F'' \backslash \{c_1 \vee x\} \backslash \{c_2 \vee \neg x\} \cup \{c_1 \vee c_2\}$ since $\mathrm{resolve}(c_1 \vee x, c_2 \vee \neg x) = \{c_1 \vee c_2\}$. Replacing F'' with its definition turns this formula into $F \backslash \{c_1 \vee c_2\} \cup \{c_1 \vee x, c_2 \vee \neg x\} \backslash \{c_1 \vee x\} \backslash \{c_2 \vee \neg x\} \cup \{c_1 \vee c_2\}$. Computing unions and set subtractions shows that this formula is F. □

This lemma tells that $F \backslash \{c_1 \vee c_2\} \cup \{c_1 \vee x, c_2 \vee \neg x\}$ is like F apart from x. This is the basic requirement for the split: it preserves the semantics as much as possible. The modified formula has the same consequences of the original that do not involve x.

The aim of the split is not just to preserve the semantics but also to make the split clause superirredundant.

The addition of x and $\neg x$, more than the split itself, is what creates superirredundancy. The two parts contain x and $\neg x$, and are the only clauses containing them. They are necessary to derive every other clause containing them, including themselves.

An exception is when the part containing x derives another containing x which derives it back. The presence of x in the whole derivation sequence ensures that removing x everywhere does not invalidate the derivation. The result is a derivation from a part of the original clause (without x added) to other clauses and back. It proves the superredundancy of that part. This explains the exception: superirredundancy is only obtained if none of the two parts of the clause is superredundant by itself.

Lemma 27 *If $c_1 \vee x$ is superredundant in $F \backslash \{c_1 \vee c_2\} \cup \{c_1 \vee x, c_2 \vee \neg x\}$ then c_1 is superredundant in $F \cup \{c_1\}$, provided that:*

- *$c_1 \vee c_2$ is in F;*
- *c_1 is not in F; and*
- *x does not occur in F.*

Proof. Lemma 14 reformulates the superredundancy in the assumption and in the claim as entailments.

$$\begin{aligned}
&F \backslash \{c_1 \vee c_2\} \cup \{c_1 \vee x, c_2 \vee \neg x\} \backslash \{c_1 \vee x\} \cup \text{resolve}(c_1 \vee x, F \backslash \{c_1 \vee c_2\} \cup \{c_1 \vee x, c_2 \vee \neg x\}) \\
&\qquad \models c_1 \vee x \\
&F \cup \{c_1\} \backslash \{c_1\} \cup \text{resolve}(c_1, F \cup \{c_1\}) \\
&\qquad \models c_1
\end{aligned}$$

The claim is proved if the first entailment implies the second. Since F contains $c_1 \vee c_2$, it is the same as $F \backslash \{c_1 \vee c_2\} \cup \{c_1 \vee c_2\}$. This allows reformulating the second entailment in terms of $F' = F \backslash \{c_1 \vee c_2\}$; the first can be as well.

$$\begin{aligned}
F' \cup \{c_1 \vee x, c_2 \vee \neg x\} \backslash \{c_1 \vee x\} \cup \text{resolve}(c_1 \vee x, F' \cup \{c_1 \vee x, c_2 \vee \neg x\}) &\models c_1 \vee x \\
F' \cup \{c_1 \vee c_2\} \cup \{c_1\} \backslash \{c_1\} \cup \text{resolve}(c_1, F' \cup \{c_1 \vee c_2\} \cup \{c_1\}) &\models c_1
\end{aligned}$$

The set subtractions can be computed immediately.

In the first formula, $F' \cup \{c_1 \vee x, c_2 \vee \neg x\} \backslash \{c_1 \vee x\}$ is equal to $F' \cup \{c_2 \vee \neg x\}$ since neither $c_2 \vee \neg x$ nor any clause in F' is equal to $c_1 \vee x$. The former is not because it contains $\neg x$, the latter are not because F' is a subset of F, which does not mention x.

In the second formula, $F' \cup \{c_1 \vee c_2\} \cup \{c_1\} \backslash \{c_1\}$ is equal to $F' \cup \{c_1 \vee c_2\}$ since neither $c_1 \vee c_2$ nor any clause in F' is equal to c_1. The first is not because it is in F while c_1 is not, the second are not because F' is a subset of F, which does not contain c_1.

$$\begin{aligned} F' \cup \{c_2 \vee \neg x\} \cup \text{resolve}(c_1 \vee x, F' \cup \{c_1 \vee x, c_2 \vee \neg x\}) &\models c_1 \vee x \\ F' \cup \{c_1 \vee c_2\} \cup \text{resolve}(c_1, F' \cup \{c_1 \vee c_2\} \cup \{c_1\}) &\models c_1 \end{aligned}$$

Both entailments contain the resolution of a clause with a union. This is the same as the resolution of the clause with each component of the union.

$$\begin{aligned} F' \cup \{c_2 \vee \neg x\} \cup \text{resolve}(c_1 \vee x, F') \cup \text{resolve}(c_1 \vee x, \{c_1 \vee x\}) \cup \text{resolve}(c_1 \vee x, \{c_2 \vee \neg x\}) &\models c_1 \vee x \\ F' \cup \{c_1 \vee c_2\} \cup \text{resolve}(c_1, F') \cup \text{resolve}(c_1, \{c_1 \vee c_2\}) \cup \text{resolve}(c_1, \{c_1\}) &\models c_1 \end{aligned}$$

Some parts of these formulae are empty because clauses do not resolve with themselves or with their superclauses.

$$\begin{aligned} F' \cup \{c_2 \vee \neg x\} \cup \text{resolve}(c_1 \vee x, F') \cup \text{resolve}(c_1 \vee x, \{c_2 \vee \neg x\}) &\models c_1 \vee x \\ F' \cup \{c_1 \vee c_2\} \cup \text{resolve}(c_1, F') &\models c_1 \end{aligned}$$

Since $c_1 \vee c_2$ is in F and formulae are assumed not to contain tautologies, the two subclauses c_1 and c_2 do not contain opposite literals. Therefore, resolving $c_1 \vee x$ and $c_2 \vee \neg x$ only generates $c_1 \vee c_2$, which is not a tautology. This simplifies $\text{resolve}(c_1 \vee x, \{c_2 \vee \neg x\})$ into $\{c_1 \vee c_2\}$.

$$\begin{aligned} F' \cup \{c_2 \vee \neg x\} \cup \text{resolve}(c_1 \vee x, F') \cup \{c_1 \vee c_2\} &\models c_1 \vee x \\ F' \cup \{c_1 \vee c_2\} \cup \text{resolve}(c_1, F') &\models c_1 \end{aligned}$$

The set $\text{resolve}(c_1 \vee x, F')$ contains the result of resolving $c_1 \vee x$ with the clauses of F'. Since F' is a subset of F, it does not contain x. Therefore, the resolving literal of $c_1 \vee x$ with a clause of $c'' \in F'$ is not x if any. If $c_1 \vee x$ resolves with $c'' \in F$, then c_1 does as well. Adding x to the resolvent generates the resolvent of $c_1 \vee x$ and c''. Formally, $\text{resolve}(c_1 \vee x, F') = \{c' \vee x \mid c' \in \text{resolve}(c_1, F')\}$.

$$\begin{aligned} F' \cup \{c_2 \vee \neg x\} \cup \{c' \vee x \mid c' \in \text{resolve}(c_1, F')\} \cup \{c_1 \vee c_2\} &\models c_1 \vee x \\ F' \cup \{c_1 \vee c_2\} \cup \text{resolve}(c_1, F') &\models c_1 \end{aligned}$$

Replacing x with false in the first entailment results in $F' \cup \{c_2 \vee \neg \mathsf{false}\} \cup \{c' \vee \mathsf{false} \mid c' \in \text{resolve}(c_1, F')\} \cup \{c_1 \vee c_2\} \models c_1 \vee \mathsf{false}$. Simplifying according to the rules of propositional logic $\mathsf{true} \vee G = \mathsf{true}$ and $\mathsf{false} \vee G = G$ turns this entailment into $F' \cup \{\mathsf{true}\} \cup \{c' \mid c' \in \text{resolve}(c_1, F')\} \cup \{c_1 \vee c_2\} \models c_1$, which is the same as $F' \cup \text{resolve}(c_1, F') \cup \{c_1 \vee c_2\} \models c_1$, the second entailment.

This proves that the first entailment implies the second: the assumption implies the claim. □

The intended usage of the lemma is to split a clause $c_1 \vee c_2$ of F into $c_1 \vee x$ and $c_2 \vee \neg x$, where x is a new variable. Being new, x does not occur in the rest of the formula. If the lemma is used this way, its first and last assumptions are met. The second may not, and the lemma fails if F contains c_1. Actually, the lemma fails if any of its three assumptions does not hold.

If $c_1 \vee c_2$ is not in F, the lemma fails. A counterexample is $c_1 = a \vee b$, $c_2 = a$ and $F = \emptyset$. The other preconditions of the lemma are satisfied: c_1 is not in F, where x does not occur; $c_1 \vee x$ is superredundant in $F \backslash \{c_1 \vee c_2\} \cup \{c_1 \vee x, c_2 \vee \neg x\}$ since this formula is $\{a \vee b \vee x, a \vee \neg x\}$, whose two clauses resolve in $a \vee b$, which entails $c_1 \vee x = a \vee b \vee x$. The conclusion that $c_1 = a \vee b$ is superredundant in $F \cup \{c_1\} = \emptyset \cup \{a \vee b\} = \{a \vee b\}$ is false since this formula allows no resolution.

If c_1 is in F, the lemma fails. A counterexample is $c_1 = a \vee b$, $c_2 = a$ and $F = \{a \vee b\}$. The other preconditions of the lemma are satisfied: $c_1 \vee c_2$ is in F, where x does not occur; $c_1 \vee x$ is superredundant in $F \backslash \{c_1 \vee c_2\} \cup \{c_1 \vee x, c_2 \vee \neg x\}$ since this formula is $\{a \vee b \vee x, a \vee \neg x\}$, whose two clauses resolve in $a \vee b$, which entails $c_1 \vee x = a \vee b \vee x$. The conclusion that $c_1 = a \vee b$ is superredundant in $F \cup \{c_1\} = \{a \vee b\} \cup \{a \vee b\} = \{a \vee b\}$ is false since this formula allows no resolution.

If F mentions x, the lemma fails. A counterexample is $c_1 = a$, $c_2 = b$ and $F = \{a \vee b, x\}$. The other preconditions of the lemma are satisfied: $c_1 \vee c_2 = a \vee b$ is in F, while c_1 is not; $c_1 \vee x$ is superredundant in $F \backslash \{c_1 \vee c_2\} \cup \{c_1 \vee x, c_2 \vee \neg x\}$, which is $\{a \vee b, x\} \backslash \{a \vee b\} \cup \{a \vee x, b \vee \neg x\}$, which is the same as $\{x\} \cup \{a \vee x, b \vee \neg x\}$, where $c_1 \vee x = a \vee x$ is superredundant because it is entailed by x. The conclusion that $c_1 = a$ is superredundant in $F \cup \{c_1\}$ is false since this formula is $\{a \vee b, x\} \cup \{a\} = \{a \vee b, x, a\}$, where no clause resolve.

The three assumptions do not hinder the intended usage of the lemma: make a clause $c_1 \vee c_2$ of F superirredundant by splitting it into $c_1 \vee x$ and $c_2 \vee \neg x$ on a new variable x. The first assumption is met because $c_1 \vee c_2$ is a clause of F to be made superirredundant. The third is met because x is new. The second is met in the sense that $c_1 \vee c_2$ can just be removed if c_1 is also in the formula.

When the three assumptions are met, the lemma tells that c_1 is superredundant in $F \cup \{c_1\}$ if $c_1 \vee x$ is superredundant in $F \backslash \{c_1 \vee c_2\} \cup \{c_1 \vee x, c_2 \vee \neg x\}$. This implication is useful in reverse: $c_1 \vee x$ is superirredundant in $F \backslash \{c_1 \vee c_2\} \cup \{c_1 \vee x, c_2 \vee \neg x\}$ unless c_1 is superredundant in $F \cup \{c_1\}$. The goal of making $c_1 \vee x$ superirredundant is hit, but only if c_1 is superirredundant in $F \cup \{c_1\}$.

This condition is necessary. The following example shows it cannot be lifted; it is file `alreadybefore.py` of `minimize.py`.

$$F = \{a \vee b, \neg a \vee c, a \vee \neg c\}$$

The last two clauses are equivalent to $a \equiv c$. They make the first clause superredundant because $a \vee b$ derives $c \vee b$ which derives $a \vee b$ back. Splitting does not make the clause superirredundant: $a \vee x$ still derives $c \vee x$, which derives $a \vee x$ back. Removing x from this derivation results in a that derives c that derives a back. Splitting $a \vee b$ does not work because a alone is already superredundant. Adding a new variable x does not change the situation.

Preserving the semantics of the formula and making a clause superirredundant is not enough. The other clauses must remain superirredundant. Otherwise, the process may go on forever. Even attempting to have two clauses superirredundant would fail if making one so makes the other not. The final requirement of clause splitting is that the other clauses remain superirredundant.

This is mostly the case, with an exception: a clause may lose its superirredundancy if it resolves with both parts of the splitted clause. This case is discussed after the proof of the lemma.

The lemma is formulated in reverse. Instead of "superirredundancy is maintained except in this condition", it states "superredundancy is generated only in this condition".

Lemma 28 *If c and $c_1 \vee c_2$ are two different clauses of F and c is superredundant in $F \backslash \{c_1 \vee c_2\} \cup \{c_1 \vee x, c_2 \vee \neg x\}$ and x does not occur in F then either:*

- *c resolves with both c_1 and c_2; or*
- *c is superredundant in F.*

Proof. Lemma 14 reformulates the superredundancy in the assumption and in the claim.

$$\begin{aligned} F \backslash \{c_1 \vee c_2\} \cup \{c_1 \vee x, c_2 \vee \neg x\} \backslash \{c\} \cup \mathrm{resolve}(c, F \backslash \{c_1 \vee c_2\} \cup \{c_1 \vee x, c_2 \vee \neg x\}) &\models c \\ F \backslash \{c\} \cup \mathrm{resolve}(c, F) &\models c \end{aligned}$$

Since F contains $c_1 \vee c_2$, it is the same as $F \backslash \{c_1 \vee c_2\} \cup \{c_1 \vee c_2\}$. Both this expression and $F \backslash \{c_1 \vee c_2\} \cup \{c_1 \vee x, c_2 \vee \neg x\}$ contain $F' = F \backslash \{c_1 \vee c_2\}$.

$$\begin{aligned} F' \cup \{c_1 \vee x, c_2 \vee \neg x\} \backslash \{c\} \cup \mathrm{resolve}(c, F' \cup \{c_1 \vee x, c_2 \vee \neg x\}) &\models c \\ F' \cup \{c_1 \vee c_2\} \backslash \{c\} \cup \mathrm{resolve}(c, F' \cup \{c_1 \vee c_2\}) &\models c \end{aligned}$$

Resolving a clause with a set is the resolution of the clause with each clause in the set.

$$\begin{aligned} F' \cup \{c_1 \vee x, c_2 \vee \neg x\} \backslash \{c\} \cup \mathrm{resolve}(c, F') \cup \mathrm{resolve}(c, \{c_1 \vee x\}) \cup \mathrm{resolve}(c, \{c_2 \vee \neg x\}) &\models c \\ F' \cup \{c_1 \vee c_2\} \backslash \{c\} \cup \mathrm{resolve}(c, F') \cup \mathrm{resolve}(c, \{c_1 \vee c_2\}) &\models c \end{aligned}$$

The clauses in these entailments are the same except for:

- the first formula contains $c_1 \vee x$, $c_2 \vee \neg x$, $\mathrm{resolve}(c, c_1 \vee x)$ and $\mathrm{resolve}(c, c_2 \vee \neg x)$
- the second formula contains $c_1 \vee c_2$ and $\mathrm{resolve}(c, c_1 \vee c_2)$

The difference depends on whether c resolves with $c_1 \vee x$, $c_2 \vee \neg x$ and $c_1 \vee c_2$. Since c does not contain x, it resolves with $c_1 \vee x$ if and only if it resolves with c_1, and the same for c_2. It resolves with $c_1 \vee c_2$ if it resolves with either c_1 or c_2. All depends on whether c resolves with c_1 or with c_2.

Four cases are possible. Apart from the last case, the claim is proved by removing all clauses containing x from the first formula and adding their resolution, which produces the second formula. Theorem 1 proves that this procedure generates a formula that expresses forgetting x and therefore entails the same consequences that do not contain x, such as c.

1. c resolves with neither c_1 nor c_2

   The three sets resolve$(c, c_1 \vee c_2)$, resolve$(c, c_1 \vee x)$ and resolve$(c, c_2 \vee \neg x)$ are empty. The only other differing clauses are $c_1 \vee x$ and $c_2 \vee \neg x$ in the first formula and $c_1 \vee c_2$ in the second. The first two are the only clauses containing x. Resolving them results in the third. This proves the claim by Theorem 1.

2. c resolves with c_1 but not with c_2

   The two sets resolve$(c, c_1 \vee c_2)$ and resolve$(c, c_1 \vee x)$ contain a clause but resolve$(c, c_2 \vee \neg x)$ does not since c does not contain x. The only differing clauses between the two formulae are $c_1 \vee x$, $c_2 \vee \neg x$ and resolve$(c, c_1 \vee x)$ in the first and $c_1 \vee c_2$ and resolve$(c, c_1 \vee c_2)$ in the second.

   Only two pairs of clauses contain x with opposite sign: the first is $c_1 \vee x$ and $c_2 \vee \neg x$, the second is resolve$(c, c_1 \vee x)$ and $c_2 \vee \neg x$.

   The first pair resolves into $c_1 \vee c_2$, the first differing clause in the second formula.

   The second pair is shown to resolve in the second differing clause, resolve$(c, c_1 \vee c_2)$. If l is the resolving literal l between c and $c_1 \vee x$, then resolve$(c, c_1 \vee x)$ is $c \vee c_1 \vee x \backslash \{l, \neg l\}$. Since c does not contain x, the resolving literal l cannot be x. As a result, this clause contains x. It therefore resolves with $c_2 \vee \neg x$ into $c \vee c_1 \vee x \backslash \{l, \neg l\} \vee c_2 \vee \neg x \backslash \{x, \neg x\} = c \vee c_1 \backslash \{l, \neg l\} \vee c_2$. Since c_2 does not resolve with c, it does not contain $\neg l$. It does not contain l either since otherwise $c_1 \vee c_2$ would be tautological. The clause is therefore the same as $c \vee c_1 \vee c_2 \backslash \{l, \neg l\}$. This is resolve$(c, c_1 \vee c_2)$, the second differing clause in the second formula.

   This proves that replacing all clauses containing x in the first formula with their resolution produces the second. This proves the claim by Theorem 1.

3. c resolves with c_2 but not with c_1

   Same as the previous case by symmetry.

4. c resolves with both c_1 and c_2

   The claim is proved because its first alternative is exactly that c resolves with both c_1 and c_2.

All of this proves the claim in all four cases. In the first three, replacing all clauses containing x with their resolution in the first formula produces the second; this implies that the two formulae have the same consequences that do not contain x, such as c. This is the first alternative of the claim. The fourth case coincides with the second alternative of the claim. □

Ideally, all clauses would maintain their superirredundancy. This is the case for most but not all. The exception is the clauses that resolve with both parts of the clause that is split. Such clauses invalidate the proof. That raises the question: could the proof be improved to include them? Or do they falsify the statement of the lemma instead? An example proves the latter; it is in the file `bothparts.py` of `minimize.py`.

$$\begin{aligned}
F &= \{c, c_1 \vee c_2, a \vee e, \neg e \vee \neg a \vee \neg c\} \\
F'' &= \{c, c_1 \vee x, \neg x \vee c_2, a \vee e, \neg e \vee \neg a \vee \neg c\} \\
c &= a \vee b \vee c \vee d \\
c_1 &= \neg a \vee b \\
c_2 &= \neg c \vee d
\end{aligned}$$

The formula obtained by splitting $c_1 \vee c_2$ is denoted F''. The clause $c_1 \vee c_2 = \neg a \vee b \vee \neg c \vee d$ is superredundant in F because it resolves with $a \vee e$ into $e \vee b \vee \neg c \vee d$, which resolves with $\neg e \vee \neg a \vee \neg c$ back into $\neg a \vee b \vee \neg c \vee d$. To make this clause superirredundant, it is split. However, that makes the first clause $c = a \vee b \vee c \vee d$ superredundant.

That c is superirredundant in F is proved replacing e with true and simplifying the formula. That removes $a \vee e$ and turns $\neg e \vee \neg a \vee \neg c$ into $\neg a \vee \neg c$. What remains is $F[\mathsf{true}/e] = \{a \vee b \vee c \vee d, \neg a \vee b \vee \neg c \vee d, \neg a \vee \neg c\}$. The first clause resolves both with the second and the third, but the result is a tautology in both cases: $F[\mathsf{true}/e] \backslash \{c\} \cup \text{resolve}(c, F)$ is equivalent to $F[\mathsf{true}/e] \backslash \{c\}$, which does not entail c. Lemma 14 proves that c is not superredundant in $F[\mathsf{true}/e]$. Since c contains neither x nor $\neg x$ and F does not contain $c \vee \neg e$, Lemma 17 ensures that c would be superredundant in $F[\mathsf{true}/c]$ it were in F. But c is not superredundant in $F[\mathsf{true}/c]$. As a result, it is not superredundant in F.

Yet, c is superredundant in F'', the formula after the split: $c = a \vee b \vee c \vee d$ resolves with $c_1 \vee x = \neg a \vee b \vee x$ into $x \vee b \vee c \vee d$; it also resolves with $\neg x \vee c_2 = \neg x \vee \neg c \vee d$ into $\neg x \vee a \vee b \vee d$; the resulting two clauses resolve into c, and therefore imply it. They are the set G that proves c superredundant according to Lemma 11, since they are obtained from F'' by resolution, none of them is c, and they imply c.

If the target was to make both c and $c_1 \vee c_2$ superirredundant, Lemma 28 misses it. Yet, a second shot gets it: $c_1 \vee x$ and $\neg x \vee c_2$ are now superirredundant but c no longer is; splitting it makes it so:

$$F''' = \{a \vee b \vee y, \neg y \vee c \vee d, \neg a \vee b \vee x, \neg x \vee \neg c \vee d, a \vee e, \neg e \vee \neg a \vee \neg c\}$$

The split separates c into $a \vee b \vee y$ and $\neg y \vee c \vee d$. Both parts resolve with $c_1 \vee c_2$, but this is not a problem because $c_1 \vee c_2$ is no longer in the formula. It has already been split into $c_1 \vee x$ and $\neg x \vee c_2$. The first resolves with $a \vee b$ but not with $c \vee d$, the second with $c \vee d$ but not with $a \vee b$. The original clause $c_1 \vee c_2$ would be made superredundant by this splitting, but its two parts $c_1 \vee x$ and $\neg x \vee c_2$ are not. By first splitting a clause and then the other, both are made superirredundant.

Mission accomplished: if a clause is not superirredundant but should be, splitting it on a new variables makes it so. Subclauses and clauses that resolve with both parts are to be watched out, but the mechanism mostly works.

Making clauses superirredundant nails them to the formula. It forces them in all minimal equivalent CNF formulae. Every CNF formula F is $F' \cup F''$, where F' are its superirredundant clauses; every minimal formula equivalent to F is $F' \cup F'''$. The superirredundant clauses F' are always there. They provide the basement over which the other clauses build upon. They are the skeleton, with its hard bones but also its flexible joins. The muscles, the other clauses, may move it not by bending the bones but by rotating them at the joins. The superirredundant clauses are fixed but may still leave space for other clauses to change. Minimizing $F' \cup F''$ is altering F'' while keeping F'. Is finding a minimal version of F'' that is equivalent to the original formula when F' is always present.

An example is the hypothetical reduction from vertex cover to formula minimization at the begin of the section. The superredundant clauses encode the graph; the superirredundant clauses force every edge in the graph to be covered by making certain other clauses necessary otherwise. Depending on the graph, the superredundant clauses include them if no suitable cover exists.

A way to ensure superirredundancy is to make clauses superirredundant. The three lemmas in this section do this:

Lemma 26 proves that splitting a clause $c_1 \vee c_2$ into $c_1 \vee x$ and $c_2 \vee \neg x$ does not change the meaning of the formula except for the new variable x;

Lemma27 proves that the two parts $c_1 \vee x$ and $c_2 \vee \neg x$ are superirredundant unless c_1 or c_2 are superredundant when added to the formula;

Lemma 28 proves that the other clauses remain superirredundant after the split; the exception are the clauses that resolve with both parts of the splitted clause; these are made superredundant, but can themselves be splitted.

Making a clause superirredundant forces it into all mininal equivalent formulae. The number of its literals is always part of the total size of the minimal formula. It is a fixed part of it. It is a tare—it could be subtracted from both the size of the formula and its required minimal size and nothing changes. For a clause of ten literals, a minimal formula of size 100 exists if and only if the other clauses can be reduced to size 90. This subtraction allows removing 10 from the calculation while still forcing the variables to satisfy the clause.

8 minimize.py

Several examples and counterexamples in the article state that a specific clause is superredundant or superirredundant in a specific formula, or that a specific formula is minimal or not minimal before or after forgetting. Proving such specific statements in full every time would be a waste of time when a simple program can do that via resolution.

That program is `minimize.py`: it computes the redundant and superredundant clauses of a formula. It optionally computes the minimal-size formulae equivalent to it. It optionally forgets a variable and computes the minimal-size formulae expressing forgetting. Being aimed at the short example formulae in the article, it is simple rather than efficient.

It is currently available at `https://github.com/paololiberatore/minimize.py` along with the related files mentioned in this article.

The formula and variable to forget are passed from the commandline or in a file. For example:

```
minimize.py -f -minimal -forget c 'a=bc' 'c->d' 'da'
minimize.py -t test1.py
```

In the first form, `c` is the variable to forget, `'a=bc' 'c->d' 'da'` is the formula: `'da'` is the clause $d \vee a$, `'c->d'` is $\neg c \vee d$ and `'a=bc'` stands for the clauses $\neg a \vee b$, $\neg a \vee c$ and $\neg b \vee \neg c \vee a$.

In the second form, `test1.py` is a file that contains the formula and the other parameters. All other Python files in the repository are examples in this form.

Variables can only be single letters. This limits the number of variables, but avoids the need of separators between them.

The program produces: the resolution closure of the formula; the set of its prime implicates; its redundant and superredundant clauses; if requested by `-minimal`, its smallest equivalent formulae; if requested by `-forget variable`, a formula expressing forgetting that variable; namely, the set of prime implicates of the formula that do not contain the variable; also the smallest formulae equivalent to it are produced.

Further explanation of the program and its internals are in the `README.md` file.

9 Conclusions

Forgetting variables from formulae may increase size, instead of decreasing it. This phenomenon is already recognized as a problem [9, 3]. Deciding whether it takes place or not for a specific formula and variables to forget is difficult. While checking inference is polynomial in the Horn case, checking whether forgetting is expressed by a formula of a certain maximal size is at least D^p-hard, which implies it both NP-hard and coNP-hard; the same for the general case, where inference is coNP-hard but checking size after forgetting is at least D_2^p-hard. These hardness results employ the concept of superredundancy of a clause of a formula. This is the redundancy of the clause in a superset of the formula containing some of its consequences. It may be useful for proving results about formula minimality outside of forgetting.

The algorithms used in the `minimize.py` program are designed with simplicity rather than efficiency. Superredundancy is checked as redundancy in the resolution closure of the formula, but Lemma 14 provides a more space-efficient solution when a single clause is to be checked for superredundancy; resolve it with every other clause of the formula, then delete it and check whether it is still entailed. Lemma 11 provides a mechanism for detecting all superredundant clauses: resolve all clauses in all possible ways; when resolution produces a clause in the original formula, this clause is superredundant.

Superredundancy was useful in proving that certain formulae are minimal, and that certain formulae contain certain clauses in all their minimal expressions. It works because it is not based on entailment but on resolution, which is a restricted form of entailment: it does not allow adding arbitrary literals to clauses. In the other way around, entailment is resolution plus expansion. The large corpus of research on automated reasoning [18, 21] offers numerous alternative forms of derivation that work even when formulae are not in clausal form, like natural deduction and Frege systems. Suitably restricting these may allow the same conclusions as superredundancy concerning parts of the formula.

Another future direction is the precise characterization of complexity. For Horn formulae the problem is proved D^p-hard and is in Σ_2^p, which leaves a large gap between the lower and the upper bound. The problem could be complete for D^p. Or for Σ_2^p. Or for a class in between, like $\Delta_2^p[\log n]$ or Δ_2^p. The same for the general case, where the problem is in Σ_3^p

but only hard for D_2^p. The definition of the problem suggests that the actual complexity is the same as its upper bounds, Σ_2^p-complete and Σ_3^p-complete, but this is yet to be proved. Finding a subcase where the upper bound lowers to the hardness level would also be of interest.

Closing the complexity gap is just one open direction for future work.

Forgetting has variants and is defined for many logics other than propositional logic. The problem of size applies to all of them. What is its complexity? This article characterizes it for one version of forgetting in propositional logic. The other versions and the other logics are still open. Some results may apply to them as well. For example, superredundancy is based on resolution and may therefore helps in analyzing the problem for first-order logic, where resolution applied. Logic programs embed Horn clauses; the hardness results for the Horn case may hold for them as well. More generally, how hard it is to check whether forgetting in logic programs is expressed within a certain size? How hard it is in first-order logic? In description logics? How hard it is when forgetting literals rather than variables?

The size after forgetting matters not only when forgetting variables but also literals [23], possibly with varying variables [31]. All variants inhibit the values of some variables to matter: forgetting variables makes their values irrelevant to the satisfaction of the formula; forgetting literals makes only the true or false value not to matter; varying variables allows some other variables to change. These variants generalize forgetting variables, inheriting the problem of size with the same complexity at least.

Forgetting applies to frameworks other than propositional logic. The problem of size applies to them as well.

Forgetting from logic programs [43, 44, 19] is usually backed by the need of solving conflicts rather than an explicit need of reducing size. Yet, an increase in size is recognized as a problem: “Whereas relying on such methods to produce a concrete program is important in the sense of being a proof that such a program exists, it suffers from several severe drawbacks when used in practice: In general, it produces programs with a very large number of rules.” [3]; “It can also be observed that forgetting an atom results in at worst a quadratic blowup in the size of the program. [...] While this may seem comparatively modest, it implies that forgetting a set of atoms may result in an exponential blowup.” [9].

Another common area of application of forgetting is first-order logic [28]. Size after forgetting is related to bounded forgetting [48], which is forgetting with a constraint on the number of nested quantifiers. The difference is that the bound is an additional constraint rather than a limit to check. Bounded forgetting still involves a measure (the number of quantifiers), but forcing the result by that measure makes it close to bounding PSPACEproblems [26]. Enforcing size rather than checking it is another possible direction of expansion of the present article.

As are the other logics where forgetting is applied like description logics [11, 45] and modal logics [46, 41], where forgetting is often referred to as its dual concept of uniform interpolant, and also temporal logics [16], logics for reasoning about actions [14, 34], and defeasible logics [1].

References

[1] G. Antoniou, T. Eiter, and K. Wang. Forgetting for defeasible logic. In *Proceedings of the eighteenth conference on Logic for Programming, Artificial Intelligence, and Reasoning (LPAR-18)*, volume 7180 of *Lecture Notes in Computer Science*, pages 77–91. Springer, 2012.

[2] A. Artale, J.C. Jung, A. Mazzullo, A. Ozaki, and F. Wolter. Living without beth and craig: Explicit definitions and interpolants in description logics with nominals (extended abstract). In Stefan Borgwardt and Thomas Meyer, editors, *Proceedings of the 33rd International Workshop on Description Logics (DL 2020)*, volume 2663 of *CEUR Workshop Proceedings*, 2020.

[3] M. Berthold, R. Gonçalves, M. Knorr, and J. Leite. A syntactic operator for forgetting that satisfies strong persistence. Technical Report abs/1907.12501, Computing Research Repository (CoRR), 2019.

[4] M. Bílková. Uniform interpolation and propositional quantifiers in modal logics. *Studia Logica*, 85(1):1–31, 2007.

[5] G. Boole. *Investigation of The Laws of Thought, On Which Are Founded the Mathematical Theories of Logic and Probabilities.* Walton and Maberly, 1854.

[6] M. Cadoli, F. M. Donini, P. Liberatore, and M. Schaerf. The size of a revised knowledge base. *Artificial Intelligence*, 115(1):25–64, 1999.

[7] O. Čepek and P. Kučera. On the complexity of minimizing the number of literals in horn formulae. RUTCOR Research Report RRR 11-208, Rutgers University, 2008.

[8] J.P. Delgrande. A knowledge level account of forgetting. *Journal of Artificial Intelligence Research*, 60:1165–1213, 2017.

[9] J.P. Delgrande and K. Wang. A syntax-independent approach to forgetting in disjunctive logic programs. In *Proceedings of the Twenty-Ninth AAAI Conference on Artificial Intelligence (AAAI 2015)*, pages 1482–1488. AAAI Press, 2015.

[10] J.P. Delgrande and R. Wassermann. Horn clause contraction functions. *Journal of Artificial Intelligence Research*, 48:475–511, 2013.

[11] T. Eiter, G. Ianni, R. Schindlauer, H. Tompits, and K. Wang. Forgetting in managing rules and ontologies. In *2006 IEEE / WIC / ACM International Conference on Web Intelligence (WI 2006), 18-22 December 2006, Hong Kong, China*, pages 411–419. IEEE Computer Society Press, 2006.

[12] T. Eiter and G. Kern-Isberner. A brief survey on forgetting from a knowledge representation and perspective. *KI — Kuenstliche Intelligenz*, 33(1):9–33, 2019.

[13] T. Eiter and K. Wang. Forgetting and conflict resolving in disjunctive logic programming. In *Proceedings, The Twenty-First National Conference on Artificial Intelligence and the Eighteenth Innovative Applications of Artificial Intelligence Conference*, pages 238–243. AAAI Press/The MIT Press, 2006.

[14] E. Erdem and P. Ferraris. Forgetting actions in domain descriptions. In *Proceedings of the Twenty-Second AAAI Conference on Artificial Intelligence (AAAI 2007)*, pages 409–414. AAAI Press, 2007.

[15] R. Fagin, J.Y. Halpern, Y. Moses, and M. Vardi. *Reasoning about knowledge*. The MIT Press, 1995.

[16] R. Feng, E. Acar, S. Schlobach, Y. Wang, and W. Liu. On sufficient and necessary conditions in bounded CTL. Technical Report abs/2003.06492, Computing Research Repository (CoRR), 2020.

[17] P. Fišer and J. Hlavička. BOOM-A heuristic boolean minimizer. *Computing and informatics*, 22(1):19–51, 2012.

[18] M. Fitting. *First-order logic and automated theorem proving*. Springer, 2012.

[19] R. Gonçalves, M. Knorr, and J. Leite. The ultimate guide to forgetting in answer set programming. In *Proceedings of the Fifteenth International Conference on Principles of Knowledge Representation and Reasoning (KR 2016)*, pages 135–144. AAAI Press, 2016.

[20] P.L. Hammer and A. Kogan. Optimal compression of propositional horn knowledge bases: Complexity and approximation. *Artificial Intelligence*, 64(1):131–145, 1993.

[21] J. Harrison. *Handbook of practical logic and automated reasoning*. Cambridge University Press, 2009.

[22] E. Hemaspaandra and H. Schnoor. Minimization for generalized boolean formulas. In *Proceedings of the Twenty-Second International Joint Conference on Artificial Intelligence (IJCAI 2011)*, pages 566–571, 2011.

[23] J. Lang, P. Liberatore, and P. Marquis. Propositional independence — formula-variable independence and forgetting. *Journal of Artificial Intelligence Research*, 18:391–443, 2003.

[24] C.T. Lee. *A completeness theorem and computer program for finding theorems derivable from given axioms*. PhD thesis, Department of Electrical Engineering and Computer Science, University of California, 1967.

[25] P. Liberatore. Compilability and compact representations of revision of Horn knowledge bases. *ACM Transactions on Computational Logic*, 1(1):131–161, 2000.

[26] P. Liberatore. Complexity issues in finding succinct solutions of PSPACE-complete problems. Technical Report abs/cs/0503043, CoRR, 2005.

[27] P. Liberatore and M. Schaerf. The compactness of belief revision and update operators. *Fundamenta Informaticae*, 62(3-4):377–393, 2004.

[28] F. Lin and R. Reiter. Forget it! In *Proceedings of the AAAI Fall Symposium on Relevance*, pages 154–159, 1994.

[29] F. Martínez-Plumed, C. Ferri, J. Hernández-Orallo, and M. Ramírez-Quintana. Knowledge acquisition with forgetting: an incremental and developmental setting. *Adaptive Behavior*, 23(5):283–299, 2015.

[30] E.J. McCluskey. Minimization of boolean functions. *The Bell System Technical Journal*, 35(6):1417–1444, 1956.

[31] Y. Moinard. Forgetting literals with varying propositional symbols. *Journal of Logic and Computation*, 17(5):955–982, 2007.

[32] Y. Moinard and R. Rolland. Smallest equivalent sets for finite propositional formula circumscription. In *Proceedings of the First International Conference on Computational Logic (CL-2000)*, volume 1861 of *Lecture Notes in Computer Science*, pages 897–911. Springer, 2000.

[33] N. Nikitina and S. Rudolph. (Non-)succinctness of uniform interpolants of general terminologies in the description logic EL. *Artificial Intelligence*, 215:120–140, 2014.

[34] D. Rajaratnam, Levesque. H.J., M. Pagnucco, and M. Thielscher. Forgetting in action. In *Proceedings of the Fourteenth International Conference on Principles of Knowledge Representation and Reasoning (KR 2014)*. AAAI Press, 2014.

[35] J. A. Robinson. A machine oriented logic based on the resolution principle. *Journal of the ACM*, 12:397–415, 1965.

[36] J.R. Slagle, C.L. Chang, and R. Lee. Completeness theorems for semantic resolution in consequence-finding. In *Proceedings of the First International Joint Conference on Artificial Intelligence (IJCAI'69)*, pages 281–286, 1969.

[37] L. J. Stockmeyer. The polynomial-time hierarchy. *Theoretical Computer Science*, 3:1–22, 1976.

[38] S. Subbarayan and D.K. Pradhan. NiVER: Non-increasing variable elimination resolution for preprocessing SAT instances. In *International conference on theory and applications of satisfiability testing*, pages 276–291. Springer, 2004.

[39] M. Theobald, S.M. Nowick, and T. Wu. Espresso-HF: a heuristic hazard-free minimizer for two-level logic. In *Proceedings of the thirdythird Design Automation Conference*, pages 71–76, 1996.

[40] C. Umans. The minimum equivalent DNF problem and shortest implicants. *Journal of Computer and System Sciences*, 63(4):597–611, 2001.

[41] H. van Ditmarsh, A. Herzig, J. Lang, and P. Marquis. Introspective forgetting. *Synthese*, 169:809–827, 2009.

[42] G. Van Rossum and F.L. Drake. *The Python language reference manual.* Network Theory Ltd., 2011.

[43] K. Wang, A. Sattar, and K. Su. A theory of forgetting in logic programming. In *Proceedings of the Twentieth National Conference on Artificial Intelligence and the Seventeenth Innovative Applications of Artificial Intelligence*, pages 682–688. AAAI Press/The MIT Press, 2005.

[44] Y. Wang, Y. Zhang, Y. Zhou, and M. Zhang. Knowledge forgetting in answer set programming. *Journal of Artificial Intelligence Research*, 50, 05 2014.

[45] X. Zhang. Forgetting for distance-based reasoning and repair in DL-lite. *Knowledge-Based Systems*, 107:246–260, 2016.

[46] Y. Zhang and Y. Zhou. Knowledge forgetting: properties and applications. *Artificial Intelligence*, 173:1525–1537, 2009.

[47] Y. Zhou. Polynomially bounded forgetting. In *Proceedings of the Thirteenth Pacific Rim International Conference on Artificial Intelligence (PRICAI 2014)*, pages 422–434, 2014.

[48] Y. Zhou and Y. Zhang. Bounded forgetting. In *Proceedings of the Twenty-Fifth AAAI Conference on Artificial Intelligence (AAAI 2011)*. AAAI Press, 2011.