Technical Report 24-97, November 1997.
Dipartimento di Informatica e Sistemistica, Università di Roma “La Sapienza”, ITALY.
URL = http://www.dis.uniroma1.it/PUB/AI/papers/cado-etal-97-d.ps.gz

Preprocessing of Intractable Problems*

Marco Cadoli   Francesco M. Donini   Paolo Liberatore
Marco Schaerf

Dipartimento di Informatica e Sistemistica,
Università di Roma “La Sapienza”,
Via Salaria 113, I-00198, Roma, Italy
Email: {cadoli|donini|liberatore|schaerf}@dis.uniroma1.it
WWW: http://www.dis.uniroma1.it/PUB/AI/

November 10, 1997

Abstract

Some computationally hard problems –e.g., deduction in logical knowledge bases– are such that part of an instance is known well before the rest of it, and remains the same for several subsequent instances of the problem. In these cases, it is meaningful to preprocess *off-line* this known part so as to simplify the remaining *on-line* problem. In this paper we investigate such a technique in the context of intractable, i.e., NP-hard, problems. Recent results in the literature show that not all NP-hard problems behave in the same way: for some of them preprocessing yields polynomial-time on-line simplified problems (we call them *compilable*), while for other ones there is strong evidence that this should not happen. Our primary goal is to provide a sound methodology that can be used either to prove or disprove that a problem is compilable. To this end, we define new models of computation, complexity classes, and reductions. We find complete problems for such classes, completeness meaning they are “the less likely to be compilable”. We also investigate preprocessing that does not yield polynomial-time on-line algorithms, but generically “decreases” complexity. This leads us to define “hierarchies of compilability”, that are the analog of the polynomial hierarchy. A detailed comparison of our framework to the related idea of “parameterized tractability” shows the differences between the two approaches.

*This paper is an extended and revised version of [7].

1 Introduction

1.1 Background and Motivations

In this paper we analyze in a formal way the idea of processing off-line part of the data representing an instance of a problem, in order to reduce the complexity of on-line computation. We refer to the preprocessing phase as *compilation*.

The basic idea of compilation—simplifying a problem by putting off-line part of the computation—is as old as Mathematics. For example, the Babylonians used a table of numbers in order to make multiplications between integers easier (the operation of multiplication is time-consuming, and modern algorithms still try to replace multiplications with sums and subtractions whenever possible). Namely, they had tables containing, for any integer a up to a given value, the value $\lfloor a^2/4 \rfloor$. Let x and y be two integers to be multiplied. It holds that

$$x \cdot y = \frac{(x+y)^2}{4} - \frac{(x-y)^2}{4} = \left\lfloor \frac{(x+y)^2}{4} \right\rfloor - \left\lfloor \frac{(x-y)^2}{4} \right\rfloor$$

Using the table, it is possible to speed up the computation, as evaluating the right hand side of the equation requires only one sum $x + y$, one subtraction $x - y$, looking up twice the table for $\left\lfloor \frac{(x+y)^2}{4} \right\rfloor$ and $\left\lfloor \frac{(x-y)^2}{4} \right\rfloor$, and then one subtraction.

The whole procedure is a waste of resources if a *single* multiplication is to be done, since it requires to compute the squares of two numbers. However, the table allows to compute many products $x \cdot y$ in a very efficient way. In this sense, this is a typical example of compilation.

Let us now consider a simple problem on graphs: given an undirected graph and two of its nodes, determine whether they are connected or not. Off-line processing gives computational advantages also in this case. The simplest algorithm to solve this problem visits the whole graph, and its cost is $O(n+m)$ (n being the number of nodes and m the number of edges). Consider instead the following algorithm: determine the connected components of the graph, number them, and then write down a table with n entries which contains, for each node, the number of the connected component it belongs to. To determine whether two nodes are connected, it's enough to find out which connected components they belong to: if they are the same, the two nodes are connected, otherwise they are not.

Of course, determining the connected components is not any easier than searching the graph for a single path. However, if the graph is known in advance, then the table can be computed once and for all, and then verifying the existence of a path only takes $O(1)$ time. This preprocessing makes sense if two conditions hold:

1. Part of the instance (the graph) is known *in advance*, that is, some time before the nodes are known and the result is needed.

2. It is necessary to perform several connectedness checks on the same graph.

If the former condition does not hold, preprocessing cannot be done. If the latter does not hold, preprocessing could still give some computational gain, although its benefits would be limited to a single instance. With respect to the first condition, note that, if the graph is not known in advance, processing it could be useful for the following instances.

Another, more realistic, example of preprocessing can be found in database technology, with the so-called *data-warehousing*. The results of specific queries to very large databases (usually called *views*) are stored separately from the database itself, thus allowing fast answers. Maintaining the data-warehouse, e.g., with respect to updates, is worthwhile if there are many queries. The cost of queries is reduced by posing them to the data-warehouse rather than to the whole database.

From the formal point of view, the notion of *amortized* computational complexity [28] is quite useful in analyzing the benefits of off-line processing of data. In fact, amortized computational complexity is an adequate tool when several instances of the same problem must be solved. However, amortized complexity has been applied to polynomial-time solvable problems.

The focus of this paper is on problems that are (probably) not polynomial-time solvable: we want to investigate under what conditions, after preprocessing, it is possible to solve them in polynomial time. If this is possible, we say that the problem is *compilable to* P, otherwise we say that it is *non-compilable to* P (the formal definitions are given in Section 2). Artificial Intelligence and Knowledge Representation offer a very simple paradigmatic problem that fits in the model described above: to decide whether a propositional formula γ is implied by a propositional knowledge base K, i.e., whether $K \models \gamma$. It is well known that this problem is coNP-complete, when both K and γ are part of the input. In many AI domains however, a single knowledge base K must be queried many times, and it makes sense to preprocess it, putting it into a form that allows the solution of $K \models \gamma$ in polynomial time. Unfortunately, if the query γ can be any clause the problem is not compilable to P. In fact, we have shown in [8], slightly generalizing a result of Kautz and Selman in [19], that the existence of a data structure whose size is polynomial w.r.t. the size of the knowledge base, and from which one can correctly answer all queries in polynomial time, is equivalent to NP $\subseteq$ P/poly (which implies that Σ_2^p =PH [18], i.e., that the polynomial hierarchy collapses at the second level). However, the problem is compilable to P if we constrain γ to be a single literal.

We consider any decision problem whose instances can be partitioned into two parts, one which we call *fixed*, and one which we call *varying*. Compilation is a preprocessing of the fixed part, and delivers a *compiled structure*. In our model we do not impose any limit on the time of compilation, but only on the size of the compiled structure: in order to be feasible, the compiled structure must have size polynomial w.r.t. the size of the fixed part.

While keeping the constraint on the size of the compiled structure, it is possible to give up polynomiality of the on-line part of the computation, as long the complexity is decreased. As an example, starting from a PSPACE-complete problem, it could make sense to have "just" an on-line NP-complete problem to deal with.

1.2 State of the Art

In the last years many researchers in AI have introduced various forms of *knowledge compilation* [25, 26] or *off-line reasoning* [23]. Specifically, they mainly focused on the problem of deciding whether a set of propositional clauses logically entails a clause. As an example, in [23] Moses and Tennenholtz show that, under some non-trivial restrictions on the query language, it is possible to modify off-line the knowledge base so that clause inference can be decided, on-line, in polynomial time. In general, proposals for representation of all the theorems of a propositional knowledge base can be classified in three main methods: 1) use of prime implicants or prime implicates; 2) adding to the knowledge base only those prime implicates that make any deduction possible by unit resolution; 3) use of prime implicates with respect to a tractable theory. An early discussion on the usefulness of knowledge compilation in the context of AI can be found in [20, 3]. A more recent survey is [5]. Benefits offered by the technique have been analyzed from the experimental point of view (cf. e.g., [24]).

Feasibility/unfeasibility of off-line processing is strictly related to possibility/impossibility of representing each formula of logic $\mathcal{L}_1$ as a formula of polynomial size in logic $\mathcal{L}_2$. "Translations" of this kind have recently been investigated in the AI literature, specifically in the context of logics for common-sense and non-monotonic reasoning. As an example, several researchers investigated the feasibility of representing propositional circumscriptions [14] or propositional default theories [1] as purely propositional formulae. In some sense this is also a form of knowledge compilation, because reasoning in such non-monotonic formalisms is typically a problem complete for the second level of the polynomial hierarchy. As an example, determining whether $CIRC(T) \models \gamma$, i.e., whether γ is true in the circumscription of T, or in other words whether γ is true in all minimal models of T, T and γ being propositional formulae, is Π_2^p-complete [12] (cf. Section 3 for the definition of minimal model). If for each propositional formula T there is a formula T' of size polynomial in the size of T such that for all formulae δ, $CIRC(T) \models \delta$ iff $T' \models \delta$, then the inference problem after compilation becomes "just" coNP-complete. Computational properties of such translations have been investigated in several papers, here briefly listed:

- [6, 21] for formulae which are the outcome of the revision or update of a propositional formula,
- [9] for circumscription, and
- [15, 8] for default logic.

In each of the above listed papers the authors show cases in which this form of compilation is possible, and cases in which it does not seem to be possible. As an example, in [9] it is shown that, if circumscription can be translated in the way specified before, then NP $\subseteq$ coNP/poly, which implies that Σ_3^p =PH [29], i.e., that the polynomial hierarchy collapses at the third level.

Another body of research bearing similarities with the idea of off-line processing is "fixed parameter tractability" [10, 11]. There are problems such that the input can be naturally split into two parts, e.g., whether a graph G has a clique of size k or not. In this case k is a parameter that can be fixed. Roughly speaking, a problem is fixed parameter tractable if it can be solved in time $f(k) \cdot n^c$, where n is the size of the input, c is a constant, and f is a function independent of the input. Although in both approaches the input to a problem is split into two parts, there are some major differences between the fixed parameter tractability of a problem and the feasibility of its off-line processing. In a nutshell, in our setting we assume that there are several instances of the same problem, all sharing a fixed part, and we are not interested in finding an expression of the complexity in terms of both parts of the input. An extensive analysis of the similarities and differences of the two approaches is done in Section 5.

To sum up, the state of the art shows several non-compilability results already appearing in the literature. Nevertheless there is no clear understanding of the properties that make some problems compilable and others non-compilable. Moreover, there is no general methodology to prove non-compilability.

1.3 Goal and results

The goal of this paper is to formally investigate and characterize the notion of compilability of an intractable problem. We show that proving non-compilability in the absolute sense amounts to solve some long-standing problems in the theory of computational complexity, such as non-collapse of the polynomial hierarchy. Therefore we devise tools for proving a problem to be "probably" non-compilable —in the same way as the notions of NP class and polynomial many-one reduction are tools for proving that a problem is "probably" intractable.

We also investigate off-line processing that decreases complexity, without necessarily leading to a polynomial-time on-line problem, e.g., a Σ_2^p-complete problem processed for obtaining an on-line problem which is in NP. For this reason, we introduce a "compilability hierarchy". Since the polynomial hierarchy [27] has both theoretical and methodological importance (as an example, AI research has recently proved it to be useful in the design of algorithms for theorem proving in non-standard logics, cf. [16]), we believe in the importance of devising an analogous notion for compilability.

In particular, we define two hierarchies of problems, that are to compilation what the polynomial hierarchy is to efficiency (without compilation). For doing that, we introduce suitable models of computation, that allow an instance of a problem to be split into two different (fixed and varying) parts.

The first hierarchy is mainly used to define the properties of compilability of problems. We have for example a class containing all the problems that are *compilable to* P, that is, all the problems that can be solved in polynomial time if a preprocessing is allowed. Another class contains problems which are in NP after the compilation, and so on. A concept of reduction for this hierarchy is defined, along with the definition of complete problems for each class of the hierarchy.

The second hierarchy is the non-uniform version of the first one. In this sense, it generalizes both the first one, and the classical non-uniform hierarchy [18, 29]. A suitable concept of reduction is defined for this hierarchy. We show that it is much easier to find complete problems for these classes, than for the corresponding uniform ones. Thus, the completeness for one class of the non-uniform hierarchy is better suited for proving the non-compilability of problems.

Provided that the polynomial hierarchy does not collapse, we prove that both our hierarchies are proper, that is, all the classes in them are distinct. The proposed framework allows for much simpler compilability and non-compilability proofs, using complete problems and reductions. We demonstrate the benefits of our proposal by providing several examples of compilable and non-compilable problems, taken both from AI and from graph theory. These examples show that complexity and compilability of a problem are distinct. In fact we provide problems with the same complexity that have different properties for what concerns compilability.

1.4 Outline

The structure of the paper is as follows. Section 2 contains definitions for the models of computation, reductions, complexity classes and hierarchies. It also contains basic properties of reductions and classes. Section 3 lists some problems which do not appear to be compilable, while Section 4 lists some which do. In Section 5 a detailed comparison of our approach with fixed parameter tractability is performed. Section 6 concludes the work. Some proofs appear in the Appendix.

2 Definitions and Properties

We assume that languages are built over an alphabet Σ that contains a special symbol # denoting blanks. The *length* of a string $x \in \Sigma^*$ is denoted by $|x|$. In this paper we are concerned with problems whose input is composed of two parts, where one part is fixed (the part that is preprocessed) and the second one is varying (only accessible on-line). Therefore, the problems we are interested in can be formally defined as sets of pairs of strings. A *language of pairs* S is a subset of $\Sigma^* \times \Sigma^*$. A function $f : \Sigma^* \to \Sigma^*$ is called *poly-size* if there exists a polynomial p such that for all $x \in \Sigma^*$ it holds $|f(x)| \leq p(|x|)$.

Moreover, we introduce some terminology for the logical problems we consider in the paper. We use a finite alphabet of propositional symbols $L = \{a, b, c, \ldots\}$, possibly with subscripts. We admit the usual boolean connectives ($\wedge$, $\vee$, and $\neg$) for constructing well-formed formulae. A *literal* is a propositional symbol or its negation. A *clause* is a disjunction of literals. An *interpretation* for L is a mapping from L in $\{\mathsf{true}, \mathsf{false}\}$. A *model* of a formula T is an interpretation M that maps T to true (written $M \models T$). A formula F is a logical consequence of another formula T (denoted as $T \models F$) if and only if $\forall M : M \models T$ implies $M \models F$.

According to our intuitive notion of compilability, a problem S is compilable if, given an instance $\langle x, y \rangle$ of it, the fixed part (x) can be preprocessed (thus obtaining $f(x)$) so that solving the problem on-line is simpler. Of course, this process is worthwhile if the cost of solving the problem, given the varying part (y) and the result of the preprocessing ($f(x)$), is much simpler than solving the problem given the (non-preprocessed) fixed and varying part. Consider for example the problem 3CNF CLAUSE INFERENCE (CI). This is the problem of deciding, given a set of clauses (each composed of three literals) and a query (a clause), whether the set implies the query. In terms of languages:

$$\text{CI} = \{\langle x, y \rangle \mid x \text{ is a 3CNF formula, } y \text{ is a clause, and } x \models y\}$$

As already pointed out, this is a prototypical problem in logic, AI, and computational complexity.

With the aim of making on-line reasoning polynomial, in the preprocessing phase we can generate and order all the clauses composed by the variables of x that are implied by x. The number of such clauses is exponential, but finite. Now, to decide whether $x \models y$, we have only to check whether there exists a clause implied by x which is a subset of y.

Let us analyze the computational cost of these operations: the preprocessing requires exponential time, and the result might require an exponential amount of memory to store the set of clauses. However, given the query y, deciding if it is implied by x is now only linear in the size of the initial set of clauses, since binary search can be applied to the ordered set of implied clauses.

This compilation is only useful if we can afford enough room to store the compiled structure. Since in most cases this is not realistic, we impose that the result of the compilation must be of polynomial size. This is why we introduce poly-size functions. We use a poly-size function f in our definitions to formalize the fact that given x, this string is processed obtaining a polynomial-size data structure $f(x)$.

Let now formally define the concept of compilation. In the above example we considered a coNP-complete problem, and the definition of compilability was: the problem is compilable if, after preprocessing the fixed part, the solution can be found

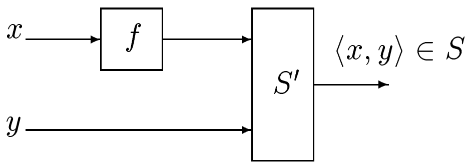


Figure 1: Machine for the comp-C class

in polynomial time. This definition can be generalized in two directions: the first is to consider problems in higher classes of the polynomial hierarchy; the second is to allow the compilation to be used to allow a simpler-but-not-polynomial solving algorithm. For example, we can have a Σ_2^p-complete problem, and it makes sense to compile the fixed part in such a way that the complexity of the resulting problem decreases to NP.

Formally, given a complexity class C, we introduce the class of problems compilable to C, that we denote as comp-C. The complement of this class is denoted as co-(comp-C). Throughout this paper we assume that C denotes a complexity class containing P.

Definition 1 (comp-C) *A language of pairs $S \subseteq \Sigma^* \times \Sigma^*$ belongs to* comp-C *iff there exist a poly-size function f and a language of pairs S' such that for all $\langle x, y \rangle \in S$ it holds that:*

1. $\langle f(x), y \rangle \in S'$ *iff* $\langle x, y \rangle \in S$;
2. $S' \in$ C.

Notice that no restriction is imposed on the *time* needed to compute the function f, but only on the *size* of the result, i.e., f is a poly-size function. This definition can be represented in terms of a computing machine as in Figure 1.

This machinery captures our intuitive notion of compilability into C of a problem S with fixed and varying parts. The function f represents the compilation of the fixed part. In order to decide whether $\langle x, y \rangle \in S$, we process off-line the fixed part x, thus obtaining $f(x)$, and then we decide whether $\langle f(x), y \rangle \in S'$. The whole process is convenient if deciding $\langle f(x), y \rangle \in S'$ is easier than deciding $\langle x, y \rangle \in S$.

Since most of the problems we discuss in this paper belong to classes of the polynomial hierarchy (PH), it is handy to define the *polynomial compilability hierarchy* as the set of classes comp-C such that C belongs to PH.

In particular, the class comp-P contains all the problems that can be solved in polynomial time on-line after an off-line processing of the fixed part. Many problems belong to this class, including problems where for every given fixed part x the number of distinct varying parts is bounded by a polynomial in $|x|$. This is shown by the following theorem (other problems are shown in Section 4).

Theorem 1 *Let S be a language of pairs and $W(S,x) = \{y \mid \langle x,y\rangle \in S\}$. If there exists a polynomial p such that for every x it holds $|W(S,x)| \leq p(|x|)$, then $S \in$ comp-P.*

Note that the class comp-C contains problems that, when both the fixed and the varying part are given on-line, are undecidable. This proves that comp-C is an extension of the class C, in that more computational power is allowed in the preprocessing phase. In order to prove the non-membership of a problem to a class, we introduce now suitable definitions of reduction and hardness. We cannot use the usual many-one polynomial reductions. Indeed, the polynomial reductions are too weak, in the sense that no problem would be comp-C-complete w.r.t. polynomial reductions (more details follow).

We now recall the basic properties that must be satisfied by all reductions. We define these properties for languages composed of one input, but they can be readily applied to languages composed of pairs. First of all, if $A, B \in \Sigma^*$ are two languages, then $f : \Sigma^* \rightarrow \Sigma^*$ is a reduction between A and B if for any $x \in \Sigma^*$ it holds $x \in A$ iff $f(x) \in B$. Let $\leq$ be a set of reductions (that is, a set of functions from strings to strings). We say that the problem A is $\leq$-reducible to B (written $A \leq B$) if and only if $\leq$ contains a reduction f from A to B.

Following [17] we introduce three additional properties that our reductions should satisfy.

Property 1 [Transitivity]
The reductions $\leq$ are transitive, i.e., $A \leq B$ and $B \leq C$ implies $A \leq C$.

Property 2 [Compatibility]
The reductions $\leq$ are compatible with the class $\mathrm{L} \subseteq \Sigma^*$, i.e., for any pair of problems A and B, if $B \in \mathrm{L}$ and $A \leq B$, then $A \in \mathrm{L}$.

Property 3 [Existence of hard problems]
There exists a problem $B \in \mathrm{L}$ such that, for any $A \in \mathrm{L}$ it holds $A \leq B$.

The above definitions easily generalize to languages of pairs.

Usual polynomial-time reductions cannot satisfy compatibility for classes comp-C. In order to have a class of reductions that are compatible with respect to the classes comp-C, and are powerful enough to allow the definition of complete problems, we introduce the notion of *comp-reduction*.

Definition 2 ($\leq_{comp}$ reduction) *A $\leq_{comp}$ reduction between two languages of pairs A and B is a triple $\langle f_1, f_2, g\rangle$, where f_1 and f_2 are poly-size unary functions and g is a binary polynomial function, such that for any pair of strings $\langle x,y\rangle$ it holds*

$$\langle x,y\rangle \in A \quad \textit{iff} \quad \langle f_1(x), g(f_2(x),y)\rangle \in B$$

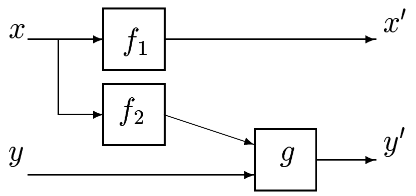


Figure 2: $\leq_{comp}$ reductions

Given two problems A and B, we say that A is *comp-reducible* to B iff there exists a $\leq_{comp}$ reduction between them, that is, $A \leq_{comp} B$.

Intuitively, $A \leq_{comp} B$ if: 1) the fixed part of B can be obtained from the fixed part of A using a poly-size function (f_1), and 2) the varying part of B can be constructed using both a poly-size function (f_2) applied to the fixed part of A, and a polynomial-time function (g) applied to f_2 and to the varying part of A. We remark that the existence of a connection between the fixed part of the first problem and the varying one of the second (f_2) is essential for obtaining some of the forthcoming results. However, in Section 3 we show various reductions that do not need to use the full power of the $\leq_{comp}$ reductions, i.e., they do not use the function f_2. The $\leq_{comp}$ reductions can be represented as in Figure 2.

These reductions satisfy the basic properties of a reduction, as shown by the following result.

Theorem 2 *The comp-reductions $\leq_{comp}$ satisfy transitivity and satisfy compatibility w.r.t.* comp-C.

Therefore, it is possible to define a notion of *hardness* and *completeness* for comp-C using this kind of reductions.

Definition 3 (Hardness and Completeness) *A language of pairs B is* comp-C-hard *iff for any $A \in$* comp-C *it holds $A \leq_{comp} B$. Moreover, B is* comp-C-complete *iff B is in* comp-C *and is* comp-C*-hard.*

Using the above definition we can find complete problems. Given a problem S with one input, we call ϵS the problem with two inputs defined as:

$$\epsilon S = \{\epsilon\} \times S = \{\langle x, y\rangle \mid x \text{ is the empty string and } y \in S\}$$

Theorem 3 *For every complexity class* C*, if S is* C*-complete (under polynomial many-one reductions) then ϵS is complete (under $\leq_{comp}$ reductions) for the corresponding compilability class* comp-C.

The above result is not surprising since the fixed part is empty, and therefore, there is no possibility of taking advantage of preprocessing. As an example, starting from the NP-complete problem 3SAT, we obtain the comp-NP-complete problem ϵ3SAT. Of course, problems of this kind are not usual in practice: the fixed part is empty, and there is no possibility to take advantage of preprocessing. However, they will be used throughout the paper, since it is easy to prove the hardness of a problem by reducing an ϵS problem to it. For several important problems we can prove that they probably do not belong to comp-P by proving their comp-C-completeness for some C above P (cf. Theorem 9), and this is usually done by reducing another comp-C-complete problem to it (and often this comp-C-complete problem is an ϵS problem).

For example, it can be easily shown that the problem FORMULA INFERENCE (FI) defined as

$$\text{FI} = \{\langle x, y\rangle \mid x \text{ and } y \text{ are propositional formulae and } x \models y\}$$

is comp-coNP-complete.

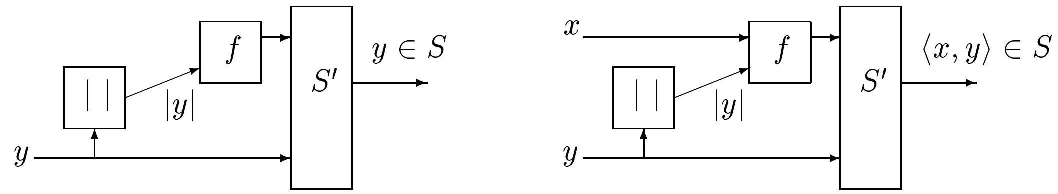
Theorem 4 *The problem* FI *is* comp-coNP-complete.

These classes characterize our notion of compilability. However, they suffer from a severe technical problem. In fact, while we can prove that FI is non-compilable by showing that FI is comp-coNP-complete, the non-compilability proofs appeared in the literature [19, 8, 9, 6, 15, 21] for many natural problems cannot be rephrased as proofs of comp-C-completeness. As a consequence, our classes fail to show that these problems are non-compilable.

Consider, for instance, the problem 3CNF CLAUSE INFERENCE (CI). A more formal statement of the result presented in Section 1.1, is that such a problem belongs to comp-P iff NP is included in P/poly [8]. This problem belongs to comp-coNP (just take f as the identity function) but it is probably not comp-coNP-complete, as proven by the following result.

Theorem 5 *If* CI *is* comp-coNP-*complete then* P = NP.

This suggests that the notion of comp-C-hardness is not a completely adequate tool for our purposes. Looking at proofs for non-compilability appearing in the literature gives another, orthogonal, perspective on the inadequacies of the formal tools introduced so far. In fact, non-compilability for CI in [19, 8] is proven by showing that if the problem is compilable, then NP $\subseteq$ P/poly. In short, the proof goes as follows: start from the NP-complete problem 3SAT, and reduce it to CI in such a way that the fixed part of CI depends only on the *size* of the instance of 3SAT. This is "almost" a $\leq_{comp}$ reduction, since it can be viewed as a reduction from ϵ3SAT to CI. However, the poly-size functions of a $\leq_{comp}$ reduction are not allowed to use the size of the varying part (in this case, the size of the instance of 3SAT).



The C/poly machine The nu-comp-C machine

Figure 3: Comparing different machines

Intuitively, in these proofs of non-compilability there is an element that is not present in $\leq_{comp}$ reductions, that is, usage of some information (i.e., the size) of the varying part in the poly-size functions.

In order to overcome this drawback we introduce the class of problems *non-uniformly compilable* to a class C, denoted as nu-comp-C. This class generalizes both comp-C and C/poly and, in the following, is mainly used to prove non-compilability results.

Definition 4 (nu-comp-C) *A language of pairs $S \subseteq \Sigma^* \times \Sigma^*$ belongs to* nu-comp-C *iff there exists a binary poly-size function f and a language of pairs S' such that for all $\langle x, y \rangle \in S$ it holds:*

1. $\langle f(x, |y|), y \rangle \in S'$ *iff* $\langle x, y \rangle \in S$;
2. $S' \in$ C.

Notice that now the poly-size function f takes as input both x and the size of y. In Figure 3 the machines corresponding to C/poly and nu-comp-C are compared.

The class nu-comp-C directly extends C/poly by allowing for a fixed part x. It also generalizes comp-C by allowing the preprocessing phase to use the size of the varying part. This aspect makes a difference whenever both the fixed part x and the size of the varying part $|y|$ are known in advance. As we show later in the paper, we only need to know an *upper bound* on the size of the varying part to apply the definition.

Similarly to the compilability classes, we define the *non-uniform polynomial compilability hierarchy* as the set of classes nu-comp-C such that C belongs to PH.

Since the nu-comp-C classes are larger than the corresponding comp-C ones, we need also a new kind of reduction, more powerful than the $\leq_{comp}$ reductions.

Definition 5 ($\leq_{nu-comp}$ **reduction**) *A $\leq_{nu-comp}$ reduction between two languages of pairs A and B is a triple $\langle f_1, f_2, g \rangle$, where f_1 and f_2 are poly-size binary functions and g is a binary polynomial function, such that for any pair of strings $\langle x, y \rangle$ it holds*

$$\langle x, y \rangle \in A \quad \textit{iff} \quad \langle f_1(x, |y|), g(f_2(x, |y|), y) \rangle \in B$$

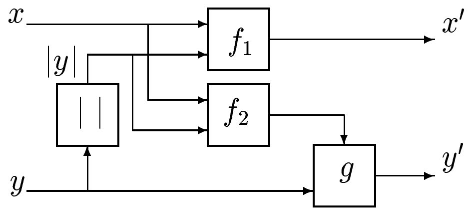


Figure 4: $\leq_{nu-comp}$ reductions

The $\leq_{nu-comp}$ reductions can be represented as in Figure 4. Given two problems A and B, we say that A is *non-uniformly-comp-reducible* to B iff there exists a $\leq_{nu-comp}$ reduction between them, that is, $A \leq_{nu-comp} B$.

These reductions satisfy all important properties of a reduction. Indeed, it holds that:

Theorem 6 *The reductions $\leq_{nu-comp}$ satisfy transitivity and compatibility w.r.t. the class* nu-comp-C.

Therefore, it is possible to define a notion of *hardness* and *completeness* for nu-comp-C.

Definition 6 (Hardness and Completeness) *Let S be a language of pairs. S is* nu-comp-C-hard *iff for all problems $A \in$* nu-comp-C *it holds $A \leq_{nu-comp} S$. The language S is* nu-comp-C-complete *if it is in* nu-comp-C *and is* nu-comp-C*-hard.*

Just like in the polynomial hierarchy and the polynomial compilability hierarchy, a problem is usually proved to be hard for a class in the non-uniform polynomial compilability hierarchy by showing that another problem, already known to be hard, can be reduced to it. As a result, for starting we need some nu-comp-C hard problem. The following theorem shows that comp-C and nu-comp-C share some complete problems.

Theorem 7 *For every class* C*, if S is* C*-complete (under polynomial many-one reductions) then ϵS is complete (under $\leq_{nu-comp}$ reductions) for the corresponding non-uniform compilability class* nu-comp-C.

Note, however, that complete problems for the two classes do not coincide, as shown by the following result, which shows that there is an adequate complexity class characterizing the problem CI.

Theorem 8 CI *is* nu-comp-coNP*-complete.*

The behavior of our classes w.r.t. complementation is quite interesting. In fact, for each complexity class C, it holds that co-(comp-C) = comp-(co-C) and co-(nu-comp-C) = nu-comp-(co-C). Some more properties of the (non-uniform) compilability classes are the following.

Theorem 9 comp-C $\subseteq$ comp-C$'$ *iff* C $\subseteq$ C$'$.

Theorem 10 nu-comp-C $\subseteq$ nu-comp-C$'$ *iff* C/poly $\subseteq$ C$'$/poly.

Theorem 10 implies (cf. [29]) that if nu-comp-Σ_i^p = nu-comp-Σ_{i+1}^p then $\Sigma_{i+2}^p = \Sigma_{i+3}^p$. For example, since CI is nu-comp-coNP-complete, if it belongs to comp-P, then it follows that nu-comp-coNP= nu-comp-P. Since nu-comp-P is closed under complementation, this implies that nu-comp-P = nu-comp-NP, and thus $\Sigma_3^p = \Sigma_4^p$.

Theorem 11 *The following relations between classes hold for every uniform and decidable complexity class* C *(where $\subset$ denotes strict containment):*

- C $\subset$ comp-C
- C $\subset$ C/poly
- comp-C $\subset$ nu-comp-C

A slightly weaker separation result is obtained for the classes C/poly and nu-comp-C.

Theorem 12 *For each class* C *such that there exists at least a language $L \notin$* C/poly, *it holds* C/poly $\subset$ nu-comp-C.

As a corollary of the above theorem, since by Muller theorem [2, Th. 2.4] there exists a language L which is not in P/poly, it holds that P/poly $\subset$ nu-comp-P. Another corollary of Theorem 12 is the following.

Theorem 13 *If the polynomial hierarchy is proper, then for each class* C *of the polynomial hierarchy it holds* C/poly $\subset$ nu-comp-C.

Relations given by Theorems 9, 10, 11, 12, and 13 are summarized in Figure 5.

We close the section by giving a summary of the complexity classes that we defined. comp-P captures the idea of "compilable problem". In general, a problem is in comp-C if, after adequate preprocessing of its fixed part, solving it on-line is a problem in C. nu-comp-NP-complete problems (under $\leq_{nu-comp}$ reductions) are what we call "non-compilable", as from Theorem 10 we know that if there exists a preprocessing of their

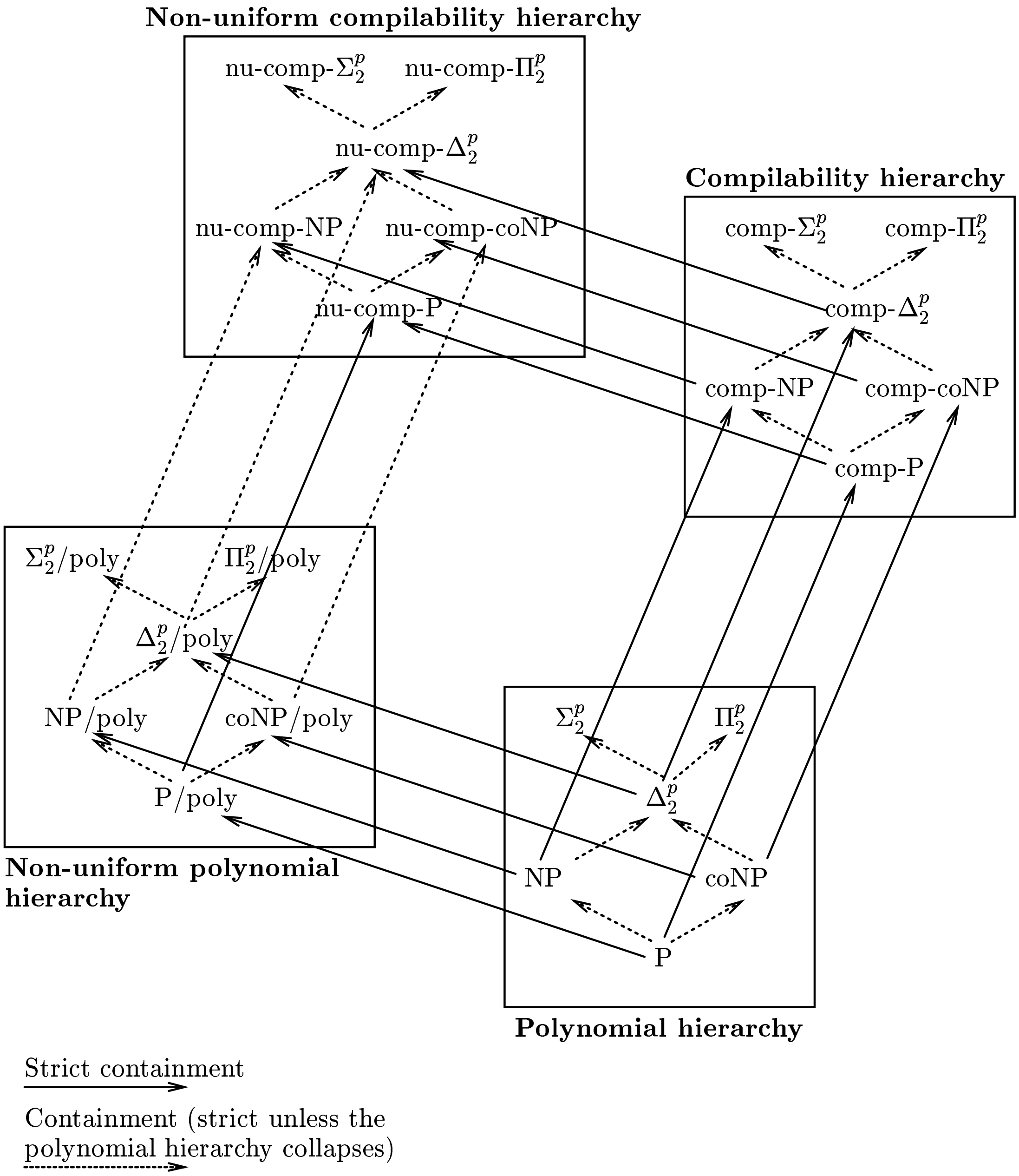


Figure 5: Relations between complexity and compilability classes.

fixed part that makes them on-line solvable in polynomial time, then NP/poly is included in P/poly. The same holds for nu-comp-coNP-complete problems. In general, a problem which is nu-comp-C-complete for a class C containing P can be regarded as the "toughest" problem in C, even after arbitrary preprocessing of the fixed part. As for comp-NP- and comp-coNP-complete problems (under $\leq_{comp}$ reductions), they are also suggestive of "non-compilability", but, as we saw in Theorem 5, the notion of comp-C-completeness is not powerful enough to capture non-compilability of problems such as CI.

3 Examples of Non-Compilable Problems

As mentioned in Section 1, we are concerned with "probably" non-compilable problems. Machinery and definitions set up so far offer a formal and simple way of proving non-compilability of a problem π: for some complexity class C above P in the polynomial hierarchy, we prove that π is nu-comp-C-complete. Then, if π is in comp-P or nu-comp-P, the two upper hierarchies of Figure 5 collapse, which implies that the polynomial hierarchy collapses at some level. This allows us to conclude that it is very unlikely that π is compilable. In this section we prove in this way non-compilability of four problems. Recall that $\leq_{nu-comp}$ reductions transform an instance $\langle x, y\rangle$ into an instance $\langle x', y'\rangle$ as follows (see Figure 4):

$$\begin{aligned} x' &= f_1(x, |y|) \\ y' &= g(f_2(x, |y|), y) \end{aligned}$$

Referring to such transformation, the proofs we are going to present are of increasing complexity, in the following sense:

1. In the first proof, only the function g is used, while $f_1(x, n) = x$, (i.e., f_1 just projects its first argument) and $f_2(x, n) = \epsilon$ (i.e., f_2 is a constant). In formulae:

$$\begin{aligned} x' &= x \\ y' &= g(y) \end{aligned}$$

2. In the second proof, $f_1(x, n) = f_1(x)$, (i.e., f_1 uses the fixed part only), while again $f_2(x, n) = \epsilon$, i.e., f_2 is not used. Hence $g(f_2(x, n), y) = g(\epsilon, y) = g(y)$ (i.e., g uses the varying part only). In formulae:

$$\begin{aligned} x' &= f_1(x) \\ y' &= g(y) \end{aligned}$$

   This is still a simple transformation, in which both x and y are separately mapped into x' and y'. Intuitively, this suggests that the structure of fixed-varying part of the two problems is very similar.

3. In the third proof, all three functions are used, but none depends on the size of the varying part $|y|$.

$$\begin{array}{rcl} x' & = & f_1(x) \\ y' & = & g(f_2(x), y) \end{array}$$

Observe that this transformation uses the machinery of a $\leq_{comp}$ reduction (hence it is still simpler than the general case).

4. The fourth proof is a true $\leq_{nu-comp}$ reduction. We start from a problem of the form ϵS. Hence, the fixed part x is the constant ϵ, and both f_1 and f_2 use only the size of the varying part:

$$\begin{array}{rcl} x' & = & f_1(|y|) \\ y' & = & g(f_2(|y|), y) \end{array}$$

3.1 Constrained Satisfiability (C-SAT)

This problem is as SAT, but with on-line constraints. Let x be a 3CNF propositional formula and y be a partial truth assignment to variables of x. C-SAT is the problem of deciding if y can be extended to a complete truth assignment satisfying x. We consider x being given off-line, while y is given on-line.

$$\text{C-SAT} = \{\langle x, y\rangle \mid y \text{ can be extended to a truth assignment satisfying } x\}$$

Proposition 14 C-SAT *is* nu-comp-NP-*complete.*

Proof. Membership in the class nu-comp-NP is obvious. For the hardness, we reduce to C-SAT the complement of CI, which is nu-comp-coNP-complete. We prove this by giving a simple $\leq_{comp}$ reduction (i.e., we do not use the size of the varying part). We have to prove that there exist two poly-size functions f_1 and f_2, and a polynomial-time function g such that, for any pairs $\langle x, y\rangle$, it holds

$$\langle x, y\rangle \notin \text{CI} \text{ iff } \langle f_1(x), g(f_2(x), y)\rangle \in \text{C-SAT}$$

(note that here $\langle x, y\rangle$ denotes the generic instance of CI to be solved using a reduction to C-SAT).

The functions that compose the reduction are defined as follows. The function f_1 is the identity, that is, $f_1(x) = x$. The function f_2 is not used (i.e., $f_2(x) = \epsilon$). Finally g builds the partial truth assignment induced by $\neg y$, that is, $g(\epsilon, y)$ is the (partial) truth assignment that maps x_i into true if x_i appears as $\neg x_i$ in the clause y, and false if it appears positively.

As a whole, given a formula x and a clause y, we have that $\langle f_1(x), g(f_2(x), y)\rangle$ is the pair $\langle x, z\rangle$, where z is the partial truth assignment defined from y as above. Using a well-known property of propositional calculus, we have that $x \not\models y$ if and only if z can be extended to form a truth assignment satisfying x.

The functions f_1, f_2 and g constitute a $\leq_{comp}$ reduction and, therefore, also a $\leq_{nu-comp}$ reduction. Hence, C-SAT is nu-comp-NP-complete. □

3.2 Constrained Vertex Cover (C-VC)

Let $G = \langle V, E\rangle$ be a graph, k be an integer and V' be a subset of V. C-VC is the problem of deciding whether there exists a vertex cover of G including V' of cardinality less than or equal to k. We assume G and k are off-line and V' is on-line. More formally,

$$\begin{aligned}\text{C-VC} \;=\; & \{\langle (G, k), V'\rangle \mid \\ & \text{there is a vertex cover of } G \text{ of cardinality} \leq k \text{ including } V'\}\end{aligned}$$

Proposition 15 C-VC *is* nu-comp-NP-*complete.*

Proof. The polynomial many-one reduction from 3SAT to VERTEX COVER in [13, pg.55] can be easily adapted to obtain a $\leq_{comp}$ reduction from C-SAT to C-VC. Note that we just need a $\leq_{comp}$ reduction, i.e., we do not use the size of the varying part.

Let an instance $\langle x, y\rangle$ of C-SAT being made as $x = \{c_1, \ldots, c_m\}$, where each c_i is a three-literals clause over the alphabet $U = \{u_1, \ldots, u_n\}$, and let $y : U' \rightarrow \{\mathsf{true}, \mathsf{false}\}$ be the partial truth assignment to U, with $U' \subseteq U$. Given x, the function f_1 constructs the graph G_x as in [13, pg.55]: for each propositional atom $u \in U$, there are in G_x two nodes $u, \bar{u}$ and an edge between them. For each clause $c_i \in x$, there is a clique of three nodes $a_1[i], a_2[i], a_3[i]$ in G_x. Then for each clause $c_i = b_1 \vee b_2 \vee b_3$ (where for $j = 1, 2, 3$ each b_j is either u_h or $\bar{u}_h$) there are three edges $(a_1[i], b_1), (a_2[i], b_2), (a_3[i], b_3)$. Moreover, f_1 counts the number of clauses m and the number of atoms n and sets $k = n + 2m$.

We define $f_2(x) = \epsilon$ for every x – that is, we do not use function f_2.

The function $g(f_2(x, n), y) = g(\epsilon, y)$ just sets $V' = U'$. Continuing to adapt the proof in [13, pg.55] , it can be shown that for each $\langle x, y\rangle$, $\langle x, y\rangle \in$C-SAT iff the the graph G_x has a vertex cover including V' of size less or equal to k.

Since a $\leq_{comp}$ reduction is also a $\leq_{nu-comp}$ reduction, this proves that C-VC is nu-comp-NP-hard. To prove membership, it is sufficient to observe that, without any preprocessing, C-VC already belongs to NP. □

In the above cases we were able to show nu-comp-NP-completeness by means of a simple usage of the $\leq_{comp}$ reduction, since the function f_2 is not used. A more complex $\leq_{comp}$ reduction – using f_2 this time – is needed for the next result.

3.3 Minimal Model Checking (MMC)

This problem is also known as model checking in circumscription. Let x be any propositional formula. The *minimal* models of x are those truth assignments which satisfy x, and have as few positive values as possible (w.r.t. set containment). MMC is the problem of deciding whether a given truth assignment y is a minimal model of x. We consider the formula x as given off-line, and the truth assignment y as given on-line.

$$\text{MMC} = \{\langle x, y\rangle \mid y \text{ is a minimal model of } x\ \}$$

Theorem 16 MMC *is* nu-comp-coNP-*complete.*

Proof. Membership in the class nu-comp-coNP is immediate, since MMC is already a coNP-complete problem [4] with no compilation at all.

Hardness is proved by showing that CI $\leq_{comp}$ MMC. Again, we do not use the size of the varying part in this reduction, so we use a $\leq_{comp}$ reduction. Let $\langle x, y\rangle$ be an instance of CI. Define the function f_1 as follows: given x, first f_1 lists all variables of x; let $a_1, \ldots, a_n$ be such variables. Then f_1 constructs a new formula x' using the set of $5n+1$ variables $\{a_1, \ldots, a_n, b_1, \ldots, b_n, c_1, \ldots, c_n, d_1, \ldots, d_n,\ e_1, \ldots, e_n, p\}$. To improve readability, let $\neq$ denote exclusive-or, and $\rightarrow$ implication. The formula x' is defined as

$$x' = (x \wedge (\bigwedge_{i=1}^{n} (b_i \rightarrow a_i) \wedge (c_i \rightarrow \neg a_i)) \vee (p \wedge (\bigwedge_{i=1}^{n} a_i))) \wedge (b_i \neq d_i) \wedge (c_i \neq e_i))$$

The function f_2 passes to g the list of all $5n+1$ variables of x'.

The function g takes also the clause y, and outputs the truth assignment v defined as follows: $v(p) = \mathsf{true}$, and for $i \in 1..n$

- $v(a_i) = \mathsf{true}$,
- $v(b_i) = \mathsf{true}$ iff a_i occurs negatively in y, $v(b_i) = \mathsf{false}$ otherwise,
- $v(c_i) = \mathsf{true}$ iff a_i occurs positively in y, $v(c_i) = \mathsf{false}$ otherwise,
- $v(d_i) = \neg v(b_i)$, and $v(e_i) = \neg v(c_i)$.

We now prove that $\langle x, y\rangle \in$ CI iff $\langle x', y'\rangle \in$ MMC, that is, $x \models y$ iff y' is a minimal model of x'. Consider the restriction v' of v to $\{b_1, \ldots, b_n, c_1, \ldots, c_n, d_1, \ldots, d_n, e_1, \ldots, e_n\}$, and substitute values of v' in x', simplifying whenever possible. Observe that all exclusive-or's $\neq$ simplify to true, and that each implication $(b_i \rightarrow a_i) \wedge (c_i \rightarrow \neg a_i)$ simplifies to:

- a_i if a_i occurs negatively in y,
- $\neg a_i$ if a_i occurs positively in y,

- true if a_i does not occur in y.

Therefore, x' is equivalent to $(x \wedge \neg y) \vee (p \wedge \bigwedge_{i=1}^{n} a_i)$ under the partial assignment v'.

Only-if part. Let $x \models y$. Suppose v is not a minimal model of x', i.e., there exists another model u assigning false to a variable which is assigned true by v, and no variable assigned false in v is assigned true by u. Observe that u and v must agree on $\{b_1, \ldots, b_n, c_1, \ldots, c_n, d_1, \ldots, d_n, e_1, \ldots, e_n\}$, because of the exclusive-or's, hence the restriction of u to $\{b_1, \ldots, b_n, c_1, \ldots, c_n, d_1, \ldots, d_n, e_1, \ldots, e_n\}$ is the same as v', namely, v'. Then, u must assign false to a variable in $\{p, a_1, \ldots, a_n\}$. Now observe that $x \models y$ implies that $x \wedge \neg y$ is unsatisfiable, i.e., $x \wedge \neg y$ is equivalent to false. In this case, x' under v' further simplifies to $(p \wedge \bigwedge_{i=1}^{n} a_i)$. Clearly, the only way of satisfying this formula is to assign true to $\{p, a_1, \ldots, a_n\}$, as v does. Therefore, there is no such a u, and v is a minimal model of x'.

If part. Let $x \not\models y$. Then $x \wedge \neg y$ is satisfiable; let t be a model of $x \wedge \neg y$, which is a truth assignment to $\{a_1, \ldots, a_n\}$. Let u be the complete truth assignment obtained by merging t, v', and $u(p) = \mathsf{false}$. Clearly, u satisfies x'. Moreover, variable p is assigned true by v and false by u, and no variable assigned false by v is assigned true by u. Hence, v is not a minimal model of x'. □

3.4 Clause Minimal Inference (CMI)

Finally, we present the problem (CMI), which is at the second level of our hierarchy of compilable problems. It is defined as

$$\text{CMI} = \{\langle x, y \rangle \mid y \text{ is a clause true in all minimal models of } x\}$$

Rephrasing results from [9], we can prove the following.

Theorem 17 CMI *is* nu-comp-Π_2^p-*complete.*

The proof (see appendix) makes use of the size of the varying part.

4 Examples of Compilable Problems

We already know by Theorem 1 that all problems whose varying part is bounded by a polynomial are in comp-P. As a consequence the problem k-CI, which is the version of CI where the size of the clause is bounded by a constant k, is in comp-P.

There are problems whose varying part is not bounded by a polynomial, but can be easily "translated" into problems obeying to such a constraint. As an example, in the problem CONJUNCTION INFERENCE, defined as

$$\text{CONJ-I} = \{\langle x, y\rangle \mid \begin{array}{l} x \text{ is a propositional formula,} \\ y \text{ is a conjunction of literals and} \\ x \models y\} \end{array}$$

the varying part is not bounded by a polynomial, as there are exponentially many conjunctions of literals. Nevertheless, the answer to a conjunctive query can be easily built using the answers to atomic queries, as $x \models y_1 \wedge \cdots \wedge y_k$ iff $x \models y_1$, and $\ldots$, and $x \models y_k$. The problem of answering literal queries has clearly a varying part which is bounded by a polynomial.

Of course, the notion of compilability is more interesting for problems whose varying part is not bounded by a polynomial, and such that there is no immediate translation into a problem of this kind. As an example the following problem is in comp-P.

Given a graph $G = \langle V, E\rangle$, an edge which does not occur in any of its Hamiltonian cycles is called *H-inessential*. The subgraph $\langle V, E'\rangle$ of G ($E' \subseteq E$) which contains no H-inessential edges is called the *Hamiltonian Reduction* of G, and is denoted as $HR(G)$. Given a graph G and a subset S of its nodes, deciding whether there is a cycle in $HR(G)$ that uses a subset of the nodes in S is an NP-hard problem: take $S = V$ (the whole set of nodes in G); if G has no Hamiltonian cycle then $HR(G)$ has no edges ($E' = \emptyset$, since every edge in G is H-inessential); otherwise, if G has at least one Hamiltonian cycle then there is a cycle in $HR(G)$ involving all nodes of S.

Nevertheless, the problem CYCLE IN A HAMILTONIAN REDUCTION, defined as:

$$\text{CHR} = \{\langle G, S\rangle \mid \begin{array}{l} \text{there is a cycle in } HR(G) \text{ that uses} \\ \text{a subset of the nodes in } S\ \} \end{array}$$

belongs to comp-P: computing $HR(G)$ can be done off-line using polynomial space, and then checking whether there is a cycle that uses a subset of the nodes in S is clearly polynomial. It is worthwhile to notice that the varying part of CHR is not bounded by a polynomial, as there are exponentially many possible subsets of nodes. Also, asking for the existence of cycles of fixed length does not help.

Another example of problem in comp-P is the following. Given a propositional formula T, its *generalized closure* [22] $GCWA(T)$ is defined as $T \cup \{\neg p \mid p$ is a literal which is false in all minimal models of $T\}$. Given a formula T and an interpretation M of its boolean letters, deciding whether M is a model of $GCWA(T)$ is a coNP-hard problem: given any formula F on alphabet $A = \{a_1, \ldots, a_n\}$, and another atom $u \notin A$, define $T = (F \wedge \neg u) \vee (u \wedge a_1 \wedge \cdots \wedge a_n)$. Let $M = \{u\} \cup A$. We can show that F is satisfiable iff $M \not\models GCWA(T)$. Note that $M \not\models GCWA(T)$ is equivalent to M not being a minimal model of T. Therefore, this reduction also shows that MMC, when both x and y are given on-line, is coNP-hard.

Again, the problem GENERALIZED CLOSURE MODEL CHECKING, defined as:

$$\text{GMC} = \{\langle x, y\rangle \mid y \models GCWA(x)\ \}$$

belongs to comp-P: computing $GCWA(x)$ can be done off-line using $O(|x|)$ space, and then checking whether $y \models GCWA(x)$ is a polynomial-time problem. The varying part of GMC is not bounded by a polynomial, as there are exponentially many truth assignments to letters. Moreover, the two problems MMC and GMC, are proven to be coNP-hard by means of the same many-one polynomial-time reduction, and still GMC is compilable while MMC is not.

GCWA gives us the opportunity to show a comp-coNP problem. Given a formula T and a clause γ, to know whether $GCWA(T) \models \gamma$ is a Π_2^p-hard problem [12]. Nevertheless, the problem GENERALIZED CLOSURE INFERENCE, defined as:

$$\text{GI} = \{\langle T, \gamma\rangle \mid GCWA(T) \models \gamma\ \}$$

belongs to comp-coNP, as checking whether $GCWA(T) \models \gamma$ —after $GCWA(T)$ has been computed off-line— is in coNP. Actually, it is easy to show that GI is nu-comp-coNP-complete, i.e., it has the same "compilability degree" of CI. This confirms that our compilability classes account for the complexity of a problem *after* preprocessing.

5 Related Work

In this section we compare our definitions of compilability with the idea of *fixed parameter tractability* as defined in [11]. We show similarities and differences between the two ideas, from both the conceptual and the technical point of view.

As in the approach proposed in this paper, in [11] the concern is on problems with two inputs. A language $L \subseteq \Sigma^* \times \Sigma^*$ is called a *parameterized language*, and if $\langle x, k\rangle \in L$, k is called a *parameter*. Usually, the parameter is a positive integer. Several decision problems can be modeled as parameterized languages; as an example, in the parameterized version of the VERTEX COVER problem, the input is a graph G, the parameter is a positive integer k, and the question is whether G has a vertex cover of size $\leq k$. Although the VERTEX COVER problem is NP-hard when k is not fixed, it can be solved by a single algorithm which runs in time $O(2^k|G|)$ for each k.

The VERTEX COVER problem is therefore polynomial if the parameter k is fixed, as captured by the following definition.

Definition 7 (Uniform fixed parameter tractability, Definition 2.4(i) of [11]) *Let A be a parameterized problem. A is* uniformly fixed parameter tractable *($A \in$* uFPT*) if there is an algorithm Φ, a constant c, and an arbitrary function $f : \mathbb{N} \longrightarrow \mathbb{N}$ such that*

(a) the running time of $\Phi(\langle x, k\rangle)$ is at most $f(k)|x|^c$, and

(b) $\langle x, k\rangle \in A$ iff $\Phi(\langle x, k\rangle) = 1$.

VERTEX COVER is uniformly fixed parameter tractable (actually, since the function $f(k) = 2^k$ is recursive, the problem is *strongly* uniformly fixed parameter tractable, cf. [11, Definition 2.4(ii)]).

The main similarities between fixed parameter tractability and compilability are the following:

- in both cases the input to a problem is split in two parts, which have different status;
- in both cases the aim is to prove that some NP-hard problems become tractable if one part of the input is treated in some special way.

The former similarity immediately raises a question: how do the inputs relate to each other in the two approaches? Clearly there are two possible answers, that we analyze considering the VERTEX COVER problem. From the formal point of view, it seems that the k of the fixed parameter tractability approach should be mapped into the fixed part and the graph G into the varying part, since the algorithm that solves the problem is exponential in k. On the other hand –from the conceptual point of view– the opposite choice makes much more sense, because the graph should be known in advance and the size of the covering should be known on-line. With the latter choice, the VERTEX COVER problem belongs to comp-P.

On the other hand, it is not obvious how a problem with two inputs which are strings, e.g., 3CNF CLAUSE INFERENCE, should be encoded in the fixed parameter tractability approach, which requires one input to be a number. One approach, which we use later in this section, is Gödelization.

The main conceptual differences are the following:

- in fixed parameter tractability "the main idea is to study languages that are tractable by the slice", and the focus is on the cost of an algorithm when a parameter of the input is "fixed";
- in compilability, the main idea is "to investigate the idea of processing off-line part of the input data", the only bound being that the compiled data structure does not have exponential size.

As a consequence, compilation makes sense only if either:

- several instances of the problem must be solved, i.e., there are many varying parts, or
- the fixed part is known well before the varying part.

On the other hand, fixed parameter tractability is defined:

- for a single instance, and
- if the algorithm comes to know both parts of the input at the same time.

In the rest of the section, we focus on formal differences between the two approaches, showing in particular that:

- comp-P $\not\subseteq$ uFPT (unless the polynomial hierarchy collapses), and
- uFPT $\not\subseteq$ comp-P (unconditionally).

To prove the former property, let's consider the CI problem (3CNF CLAUSE INFERENCE, cf. Section 2), which we recall here for convenience:

$$\text{CI} = \{\langle x, y\rangle \mid x \text{ is a 3CNF formula, } y \text{ is a clause and } x \models y\}.$$

CI is probably not compilable, since it belongs to comp-P iff NP is included in P/poly.

Nevertheless, CI can be easily shown to belong to uFPT (we assume that the off-line part x is treated as a fixed parameter). Given x and y, generate all the models of x, and for each such model, verify whether it implies y. The running time of this procedure is $O(2^{|x|} \cdot |y|)$, and is thus uniformly fixed parameter tractable. Such an algorithm is exponential only in the size of the fixed part, but it requires the knowledge of the varying part to work. Obviously, this procedure does not prove that CI is compilable to P.

The following remarks are in order:

- As for syntax, in the approach of [11] the parameter is the second argument of $\langle x, k\rangle$, while in our approach the fixed part, which is conceptually analogous, is the first argument.
- In our approach, the fixed part is a string, not an integer. This difficulty can be easily bypassed, and a more formal proof that comp-P $\subseteq$ uFPT implies the collapse of the polynomial hierarchy can be obtained by the following steps:

  1. Introduce two functions:
     - $\mathfrak{G} : \Sigma^* \longrightarrow \mathbb{N}$, which associates a positive integer to each string x representing a 3CNF formula;
     - $\mathfrak{H} : \mathbb{N} \longrightarrow \Sigma^*$, which associates a string representing a 3CNF formula to each positive integer k.

     It is easy to design $\mathfrak{G}$ and $\mathfrak{H}$, in such a way that they have the following properties (cf. Proposition 19 in the Appendix):

     - $\mathfrak{G}(x)$ can be computed in time polynomial in the size of x.

- $\mathfrak{H}(k)$ can be computed in time polynomial in the size of k (i.e., in time $\log k$).
- $\mathfrak{H}(\mathfrak{G}(x)) = x$ for any x.

2. Define the problem FCI (FIXED PARAMETER 3CNF CLAUSE INFERENCE) as (notation as in [11]: Q = input string; k = parameter)

$$\text{FCI} = \{ \langle Q, k \rangle \bigm| Q \text{ is a clause and } \mathfrak{H}(k) \models Q \}$$

3. Prove that FCI is nu-comp-coNP-hard, by reducing the problem CI to FCI via a $\leq_{nu-comp}$ reduction (cf. Proposition 20 in the Appendix).
4. Prove that FCI $\in$ uFPT (cf. Proposition 21 in the Appendix).

To prove that comp-P $\not\subseteq$ uFPT we consider the problem HALT defined as follows:

$$\text{HALT} = \{\langle x, y \rangle \mid x \text{ represents an always-terminating Turing machine } \}$$

This problem is clearly compilable to P, since the decision of the termination can be made off-line. On the other hand, there is no algorithm that can decide the termination of a Turing machine, thus it is not uniformly fixed parameter tractable.

By allowing the solution of undecidable problems in the preprocessing phase, we are just strengthening non-compilability results such as nu-comp-coNP-hardness of CI: any problem that is, e.g., nu-comp-coNP-hard is not compilable *even if we can solve arbitrarily hard problem in the preprocessing phase.* If we were to limit such power, e.g., by defining comp-P$'$ to be the class of problems that can be solved in polynomial time by preprocessing the fixed part with a terminating procedure, the corresponding reduction (to be compatible with the classes of this new hierarchy) must have f_1 and f_2 decidable too. As a result, each reduction of this kind is also a $\leq_{nu-comp}$ reduction, but not the other way around. This implies that there are problems that can be proven to be not compilable using $\leq_{nu-comp}$ reductions and hardness, but can not if decidability is imposed. Thus, limiting the preprocessing phase to decidable problems does not give any advantage in proving that a problem is not compilable.

It is interesting to note that comp-P$'$ $\subset$ uFPT: for a single instance of the problem, the total time for the preprocessing plus the time for the on-line computation is $h(|x|) + g(|x| + |y|)$, where h is a generic function, and g is a polynomial. This is a case of fixed parameter tractability, since the non-polynomial part of this function depends only on the size of the fixed part. On the other hand, the classes of compilability characterize the complexity of solving many instances sharing the same fixed part, or the case of a fixed part which is known in advance, which are are scenarios not covered by the fixed parameter tractability theory.

6 Conclusions

We applied theoretical computer science techniques to analyze intractable problems where part of the input—the off-line part—can be preprocessed. We imposed on the preprocessing the sole condition that its output must have polynomial size w.r.t. the preprocessed input data.

We defined suitable computational models with a preprocessing phase, followed by a (ordinary) computation taking as inputs both the result of preprocessing and some other (on-line) input.

We proposed two new classes of problems, namely compilable and non-uniformly compilable problems, and generalized this notions to hierarchies which are analogous (and related) to the polynomial hierarchy and its non-uniform version.

Our proposal systematizes many proofs of the impossibility of representing a formula in logic $\mathcal{L}_1$ as a formula of polynomial size in logic $\mathcal{L}_2$ (e.g., [9, 15]). These proofs can be now rephrased in terms of membership/completeness in a specific class of the hierarchies.

Moreover, our proofs of the impossibility of having a preprocessing with a polynomial-size output provide worst cases for compilation methods in Artificial Intelligence problems.

Finally, we compared our classes with the class of fixed parameter tractable problems, and showed with both intuitive arguments and formal results that the classes do not coincide.

Acknowledgments

The authors are grateful to Pierluigi Crescenzi, Georg Gottlob, and Riccardo Silvestri for interesting discussions on the topic of the paper. They also thank Pino Italiano and Fabio Massacci for useful comments on an earlier draft. This work has been supported by ASI (Italian Space Agency), MURST (Italian Ministry for University and Scientific and Technological Research) and CNR (Italian Research Council) under the SARI project.

References

[1] R. Ben-Eliyahu and R. Dechter. Default reasoning using classical logic. *Artificial Intelligence Journal*, 84(1–2):113–150, 1996.

[2] R. Boppana and M. Sipser. The complexity of finite functions. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science*, volume A, chapter 14. Elsevier Science Publishers (North-Holland), Amsterdam, 1990.

[3] A. Borgida, R. J. Brachman, D. W. Etherington, and H. A. Kautz. Vivid knowledge and tractable reasoning: Preliminary report. In *Proceedings of the Eleventh International Joint Conference on Artificial Intelligence (IJCAI-89)*, pages 1146–1152, 1989.

[4] M. Cadoli. The complexity of model checking for circumscriptive formulae. *Information Processing Letters*, 44:113–118, 1992.

[5] M. Cadoli and F. M. Donini. A survey on knowledge compilation. *AI Communications—The European Journal for Artificial Intelligence*, 1997. To appear. Preliminary version available on the WWW; URL= `http://www.dis.uniroma1.it/pub/AI/papers/cado-doni-97.ps.gz`.

[6] M. Cadoli, F. M. Donini, P. Liberatore, and M. Schaerf. The size of a revised knowledge base. In *Proceedings of the Fourteenth ACM SIGACT SIGMOD SIGART Symposium on Principles of Database Systems (PODS-95)*, pages 151–162, 1995. Extended version available as Technical Report DIS 34-96, Dipartimento di Informatica e Sistemistica, Università di Roma "La Sapienza", November 1996.

[7] M. Cadoli, F. M. Donini, P. Liberatore, and M. Schaerf. Feasibility and unfeasibility of off-line processing. In *Proceedings of the Fourth Israeli Symposium on Theory of Computing and Systems (ISTCS-96)*, pages 100–109. IEEE Computer Society Press, 1996.

[8] M. Cadoli, F. M. Donini, and M. Schaerf. Is intractability of non-monotonic reasoning a real drawback? *Artificial Intelligence Journal*, 88(1–2):215–251, 1996.

[9] M. Cadoli, F. M. Donini, M. Schaerf, and R. Silvestri. On compact representations of propositional circumscription. *Theoretical Computer Science*, 182:183–202, 1997.

[10] R. G. Downey and M. F. Fellows. Fixed-parameter tractability and completeness: Basic results. *SIAM Journal on Computing*, 24(4):873–921, 1995.

[11] R. G. Downey and M. F. Fellows. Parameterized complexity, February 1995. Manuscript, 411 pages. To appear by Springer in 1997.

[12] T. Eiter and G. Gottlob. Propositional circumscription and extended closed world reasoning are Π_2^p-complete. *Theoretical Computer Science*, 114:231–245, 1993.

[13] M. R. Garey and D. S. Johnson. *Computers and Intractability, A Guide to the Theory of NP-Completeness*. W.H. Freeman and Company, San Francisco, Ca, 1979.

[14] M. Gelfond, H. Przymusinska, and T. Przymusinsky. On the relationship between circumscription and negation as failure. *Artificial Intelligence Journal*, 38:49–73, 1989.

[15] G. Gogic, H. A. Kautz, C. Papadimitriou, and B. Selman. The comparative linguistics of knowledge representation. In *Proceedings of the Fourteenth International Joint Conference on Artificial Intelligence (IJCAI-95)*, pages 862–869, 1995.

[16] G. Gottlob. Complexity and expressive power of KR formalisms. In *Proceedings of the Fifth International Conference on the Principles of Knowledge Representation and Reasoning (KR-96)*, pages 647–649, 1996.

[17] D. S. Johnson. A catalog of complexity classes. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science*, volume A, chapter 2. Elsevier Science Publishers (North-Holland), Amsterdam, 1990.

[18] R. M. Karp and R. J. Lipton. Some connections between non-uniform and uniform complexity classes. In *Proceedings of the Twelfth ACM Symposium on Theory of Computing (STOC-80)*, pages 302–309, 1980.

[19] H. A. Kautz and B. Selman. Forming concepts for fast inference. In *Proceedings of the Tenth National Conference on Artificial Intelligence (AAAI-92)*, pages 786–793, 1992.

[20] H. J. Levesque. Making believers out of computers. *Artificial Intelligence Journal*, 30:81–108, 1986.

[21] P. Liberatore. Compact representation of revision of Horn clauses. In Xin Yao, editor, *Proceedings of the Eighth Australian Joint Artificial Intelligence Conference (AI-95)*, pages 347–354. World Scientific, 1995.

[22] J. Minker. On indefinite databases and the closed world assumption. In *Proceedings of the Sixth International Conference on Automated Deduction (CADE-82)*, pages 292–308, 1982.

[23] Y. Moses and M. Tennenholtz. Off-line reasoning for on-line efficiency: knowledge bases. *Artificial Intelligence Journal*, 83:229–239, 1996.

[24] R. Schrag and J. Crawford. Implicates and prime implicates in random 3SAT. *Artificial Intelligence Journal*, 81:199–222, 1996.

[25] B. Selman and H. A. Kautz. Knowledge compilation using Horn approximations. In *Proceedings of the Ninth National Conference on Artificial Intelligence (AAAI-91)*, pages 904–909, 1991.

[26] B. Selman and H. A. Kautz. Knowledge compilation and theory approximation. *Journal of the ACM*, 43:193–224, 1996.

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

[28] R. E. Tarjan. Amortized computational complexity. *SIAM Journal on Algebraic and Discrete Methods*, 6:306–318, 1985.

[29] C. K. Yap. Some consequences of non-uniform conditions on uniform classes. *Theoretical Computer Science*, 26:287–300, 1983.

A Appendix

A.1 Proofs of Section 2

Theorem 1 *Let S be a language of pairs and $W(S,x) = \{y \mid \langle x, y\rangle \in S\}$. If there exists a polynomial p such that for every x it holds $|W(S,x)| \leq p(|x|)$, then $S \in$ comp-P.*

Proof. Given off-line x, we simply build a table, where we record the solution to the decision problem $\langle x, y\rangle$ for each y. Thus on-line we read y and access such a table. This access is obviously feasible in polynomial time w.r.t. the size of the table. Since $W(S,x)$ has size polynomial in $|x|$, this table has polynomial size too.

Formally, given x let $W(S,x) = \{y_1, \ldots, y_m\}$, with m bounded by a polynomial $p(|x|)$. Let @ be a new symbol, and

$$
\begin{aligned}
f(x) &= y_1 @ \cdots @ y_m \\
S' &= \{\langle a_1 @ \cdots @ a_m, y\rangle \mid y = a_i \text{ for some } i\}
\end{aligned}
$$

Then, we have

$$
\begin{aligned}
\langle x, y\rangle \in S &\Leftrightarrow y \in W(S,x) \\
&\Leftrightarrow y = \text{ some element of } f(x) \\
&\Leftrightarrow \langle f(x), y\rangle \in S'
\end{aligned}
$$

□

Theorem 2 *The comp-reductions $\leq_{comp}$ satisfy transitivity and satisfy compatibility w.r.t.* comp-C.

Proof. We have to prove that the relation $\leq_{comp}$ is transitive and is compatible with the compilable uniform hierarchy, that is, if $A \leq_{comp} B$ and $B \in$ comp-C then $A \in$ comp-C.

Transitivity Suppose $A \leq_{comp} B$ and $B \leq_{comp} C$. By definition there exist four poly-size functions f_1, f_2, f_1' and f_2' and two polynomial reductions g and g' such that

$$\begin{array}{rcl}\langle x, y\rangle \in A & \Leftrightarrow & \langle f_1(x), g(f_2(x), y)\rangle \in B \\ \langle x', y'\rangle \in B & \Leftrightarrow & \langle f_1'(x'), g'(f_2'(x'), y')\rangle \in C\end{array}$$

Now let

$$\begin{array}{rcl}f_1''(x) & = & f_1'(f_1(x)) \\ f_2''(x) & = & f_2'(f_1(x))@f_2(x) \\ g''(x, y) & = & \left\{\begin{array}{ll} g'(r, g(s, y)) & \text{if @ appears exactly once in } x \text{ and } x = r@s \\ \epsilon & \quad\text{otherwise}\end{array}\right.\end{array}$$

where @ is a new symbol. Note that the functions f_1'' and f_2'' are poly-size, and g'' is a polynomial reduction. Then, we have

$$\begin{array}{rcl}\langle x, y\rangle \in A & \Leftrightarrow & \langle f_1(x), g(f_2(x), y)\rangle \in B \\ & \Leftrightarrow & \langle f_1'(f_1(x)), g'(f_2'(f_1(x)), g(f_2(x), y))\rangle \in C \\ & \Leftrightarrow & \langle f_1''(x), g''(f_2'(f_1(x))@f_2(x), y\rangle \in C \\ & \Leftrightarrow & \langle f_1''(x), g''(f_2''(x), y)\rangle \in C\end{array}$$

Compatibility Let B be a comp-C problem, and $A \leq_{comp} B$. We prove that A is a comp-C problem. By definition, there are three poly-size functions f_1, f_2 and f', a polynomial function g and a C language S' such that

$$\begin{array}{rcl}\langle x, y\rangle \in A & \Leftrightarrow & \langle f_1(x), g(f_2(x), y)\rangle \in B \\ \langle x', y'\rangle \in B & \Leftrightarrow & \langle f'(x'), y'\rangle \in S\end{array}$$

Now, let $f''(x) = f'(f_1(x))@f_2(x)$ and S'' be the C language characterized by

$$\langle a@b, c\rangle \in S'' \Leftrightarrow \langle a, g(b, c)\rangle \in S'$$

We have that

$$\begin{array}{rcl}\langle x, y\rangle \in A & \Leftrightarrow & \langle f_1(x), g(f_2(x), y)\rangle \in B \\ & \Leftrightarrow & \langle f'(f_1(x)), g(f_2(x), y)\rangle \in S \\ & \Leftrightarrow & \langle f''(x), y\rangle \in S''\end{array}$$

so A is a comp-C language. $\square$

Theorem 3 *For every complexity class* C*, if S is* C*-complete (under polynomial many-one reductions) then ϵS is complete (under $\leq_{comp}$ reductions) for the corresponding compilability class* comp-C*.*

Proof. Suppose that $A \in$ comp-C. Then there exists a poly-size function f and a C language S' such that

$$\langle x, y \rangle \in A \;\Leftrightarrow\; \langle f(x), y \rangle \in S'$$

Now, S is a C-complete problem, so given $S' \in$ C, there exists a polynomial function g such that

$$a \in S' \;\Leftrightarrow\; g(a) \in S$$

Now, consider the following functions:

$$\begin{aligned} f_1(x) &= \epsilon \\ f_2(x) &= f(x) \\ g'(a,b) &= g(a,b) \end{aligned}$$

Using these poly-size functions, A can be reduced to ϵS since

$$\begin{aligned} \langle x, y \rangle \in A &\Leftrightarrow \langle f(x), y \rangle \in S' \\ &\Leftrightarrow g(f(x), y) \in S \\ &\Leftrightarrow \langle \epsilon, g(f(x), y) \rangle \in S \\ &\Leftrightarrow \langle f_1(x), g'(f_2(x), y) \rangle \in S \end{aligned}$$

so $A \leq_{comp} \epsilon S$. $\square$

Theorem 4 *The problem* FI *is* comp-coNP-complete.

Proof. Since the decision problem FI where both the knowledge base and the query are given on-line is in coNP then FI is also in comp-coNP. By theorems (2,3) it suffices to show that ϵ3UNSAT $\leq_{comp}$ FI. Given an instance $\langle x, y \rangle$ we define two poly-size functions f_1, f_2 and a polynomial function g such that

$$\langle x, y \rangle \in \epsilon 3\text{UNSAT} \;\Leftrightarrow\; \langle f_1(x), g(f_2(x), y) \rangle \in \text{FI}$$

Let

$$\begin{aligned} f_1(x) &= \mathsf{true} \\ f_2(x) &= x \\ g(x,y) &= \begin{cases} \mathsf{false} & \text{if } x \neq \epsilon \\ \neg y & \text{otherwise} \end{cases} \end{aligned}$$

We have two cases, either $x = \epsilon$ or not. In the first case, we need to show that $\langle \epsilon, y\rangle \in \epsilon$3UNSAT if and only $\langle \mathsf{true}, \neg y\rangle \in$ FI. This is equivalent to say that y is unsatisfiable iff true *models*$\neg y$, that trivially holds. In the second case ($x \neq \epsilon$), we show that $\langle x, y\rangle \in \epsilon$3UNSAT if and only $\langle \mathsf{true}, \mathsf{false}\rangle \in$ FI. Notice that $\langle x, y\rangle \notin \epsilon$3UNSAT when $x \neq \epsilon$ and, at the same time, it is not the case that $\mathsf{true} \models \mathsf{false}$. □

Theorem 5 *If* CI *is* comp-coNP-*complete then* P = NP.

Proof. Let UNSAT be the complementary problem of SAT. Assume that CI is a comp-coNP-complete problem. Then the problem ϵUNSAT is reducible to it, that is, there are two poly-size functions f_1, f_2 and a polynomial function g such that

$$\langle x, y\rangle \in \epsilon\text{UNSAT} \;\Leftrightarrow\; \langle f_1(x), g(f_2(x), y)\rangle \in \text{CI}$$

where $f_1(x)$ is a 3-CNF formula and $g(f_2(x), y)$ is a clause. Now, let F be a propositional formula. We have

$$\begin{array}{rcl} F \text{ unsatisfiable} & \Leftrightarrow & \langle \epsilon, F\rangle \in \epsilon\text{UNSAT} \\ & \Leftrightarrow & \langle f_1(\epsilon), g(f_2(\epsilon), F)\rangle \in \text{CI} \end{array}$$

The rightmost formula is equivalent to $f_1(\epsilon) \models g(f_2(\epsilon), F)$. Now, $f_1(\epsilon)$ is a constant formula and $g(f_2(\epsilon), F)$ is a clause. Deciding if a clause is implied by a constant formula is a polynomial-time problem. But this solves also the coNP-complete problem of deciding whether F is unsatisfiable or not. Hence, if CI is a comp-coNP-complete problem, then P=coNP. Since P is closed under complementation, then P=NP. □

Theorem 6 *The reductions* $\leq_{nu-comp}$ *satisfy transitivity and compatibility w.r.t. the class* nu-comp-C*.*

Proof. We have to prove that the relation $\leq_{nu-comp}$ is transitive, and that if $A \leq_{nu-comp} B$ and $B \in$ nu-comp-C, then $A \in$ nu-comp-C.

Transitivity

By hypothesis there exist four poly-size functions f_1, f_2, f'_1 and f'_2, and two polynomial ones g and g', such that

$$\begin{array}{rcl} \langle x, y\rangle \in A & \Leftrightarrow & \langle f_1(x, |y|), g(f_2(x, |y|), y)\rangle \in B \\ \langle x', y'\rangle \in B & \Leftrightarrow & \langle f'_1(x', |y'|), g'(f'_2(x', |y'|), y')\rangle \in C \end{array}$$

Since g is a polynomial function, we can determine a polynomial p such that $|g(a, b)| \leq p(|a| + |b|)$, for any a, b. Now let

$$\begin{array}{rcl} f''_1(x, |y|) & = & f'_1(f_1(x, |y|), p(|f_2(x, |y|)| + |y|) \\ f''_2(x, |y|) & = & f''_2(f_1(x, |y|), p(|f_2(x, |y|)| + |y|)@f_2(x, |y|) \end{array}$$

Let $r(n) = \# \ldots \#$ (a sequence of n blanks) and $k = p(|a| + |b|) - |g(a,b)|$. Notice that, by the definition of p, k is strictly positive. Now, let $g_r(a,b) = g(a,b)r(k)$, that is $g(a,b)$ followed by k blanks. The polynomial function g'' is defined as:

$$g''(x,y) = \begin{cases} g'(r, g_r(s,y)) & \text{if @ appears exactly once in } x \text{ and } x = r@s \\ \epsilon & \text{otherwise} \end{cases}$$

We can now prove the claim.

$$\begin{array}{rcl}
\langle x,y\rangle \in A & \Leftrightarrow & \langle f_1(x,|y|), g(f_2(x,|y|),y)\rangle \in B \\
& \Leftrightarrow & \langle f_1(x,|y|), g_r(f_2(x,|y|),y)\rangle \in B \\
& \Leftrightarrow & \langle f_1'(f_1(x,|y|), |g_r(f_2(x,|y|),y)|), g'(f_2'(f_1(x,|y|), |g_r(f_2(x,|y|),y)|), \\
& & g_r(f_2(x,|y|),y))\rangle \in C \\
& \Leftrightarrow & \langle f_1'(f_1(x,|y|), p(|f_2(x,|y|)| + |y|), g'(f_2'(f_1(x,|y|), p(|f_2(x,|y|)| + |y|)), \\
& & g_r(f_2(x,|y|),y))\rangle \in C \\
& \Leftrightarrow & \langle f_1''(x,|y|), g''(f_2''(x,|y|),y)\rangle \in C
\end{array}$$

Compatibility

Let us suppose $A \leq_{nu-comp} B$ and $B \in$ nu-comp-C. We can prove that $A \in$ nu-comp-C. By hypothesis there exist two poly-size functions f_1 and f_2 and a polynomial one g such that

$$\langle x,y\rangle \in A \quad \Leftrightarrow \quad \langle f_1(x,|y|), g(f_2(x,|y|),y)\rangle \in B$$

and there is a poly-size function f and a C language S' such that

$$\langle x',y'\rangle \in B \quad \Leftrightarrow \quad \langle f(x',|y'|), y'\rangle \in S'$$

Now, let f' and S'' be

$$\begin{array}{rcl}
f'(x,|y|) & = & f(f_1(x,|y|),0)@f(f_1(x,|y|),1)@\cdots@f(f_1(x,|y|),p(|x|+|y|))@f_2(x,|y|) \\
S'' & = & \{\langle a_0@\cdots@a_n@b, c\rangle \mid \langle a_i, g(b,c)\rangle \in S' \text{ where } i = |g(b,c)|\}
\end{array}$$

In other words, f' concatenates the results of the function f for all values from 0 up to $p(|x|+|y|)$), while the language S'' picks and concatenates some of the first elements of the language S'. Notice that S'' is a C language since the complexity of checking membership is polynomially related to the complexity of checking membership in S'.

We have

$$\begin{array}{rcl}
\langle x,y\rangle \in A & \Leftrightarrow & \langle f_1(x,|y|), g(f_2(x,|y|),y)\rangle \in B \\
& \Leftrightarrow & \langle f(f_1(x,|y|), |g(f_2(x,|y|)|), g(f_2(x,|y|),y)\rangle \in S' \\
& \Leftrightarrow & \langle f'(x,|y|), y\rangle \in S''
\end{array}$$

$\square$

Theorem 7 *For every class* C*, if* S *is* C*-complete (under polynomial many-one reductions) then* ϵS *is complete (under* $\leq_{nu-comp}$ *reductions) for the corresponding non-uniform compilability class* nu-comp-C*.*

Proof. Let A be a generic nu-comp-C problem. By hypothesis,

$$\langle x, y\rangle \quad \Leftrightarrow \quad \langle f(x, |y|), y\rangle \in S'$$

But $S' \in C$, thus it can be reduced to S using a polynomial function g such that

$$\langle x, y\rangle \in S' \quad \Leftrightarrow \quad f(x, y) \in S$$

Now, let

$$\begin{aligned} f_1(x, |y|) &= \epsilon \\ f_2(x, |y|) &= f(x, |y|) \\ g'(a, b) &= g(a, b) \end{aligned}$$

We have

$$\begin{aligned} \langle x, y\rangle \in A &\Leftrightarrow \langle f(x, |y|), y\rangle \in S' \\ &\Leftrightarrow g(f(x, |y|), y) \in S \\ &\Leftrightarrow \langle \epsilon, g(f(x, |y|), y)\rangle \in S \\ &\Leftrightarrow \langle f_1(x, |y|), g(f_2(x, |y|), y)\rangle \in \epsilon S \end{aligned}$$

$\square$

Theorem 8 CI *is* nu-comp-coNP*-complete.*

Proof. Since the decision problem CI where both the knowledge base and the query are given on-line is in coNP then CI is also in nu-comp-coNP. By theorems (6,7) it suffices to show that ϵ3UNSAT $\leq_{nu-comp}$ CI. Given an instance $\langle \delta, \pi\rangle$ we define two poly-size functions f_1, f_2 and a polynomial function g such that

$$\langle \delta, \pi\rangle \in \epsilon\text{3UNSAT} \quad \Leftrightarrow \quad \langle f_1(\delta, |\pi|), g(f_2(\delta, |\pi|), \pi)\rangle \in \text{CI}$$

Let $n = |\pi|$ and L the alphabet of (at most) n atoms used in π. We exploit the fact that each three-literals clause over L can be given a name: we denote with C a set of atoms one-to-one with possible three-literals clauses of L

$$C = \{c_i \mid \gamma_i \text{ is a three-literals clause of } L\} \tag{1}$$

Let

$$\begin{aligned}
f_1(\delta, n) &= \begin{cases} \mathsf{true} & \text{if } \delta \neq \epsilon \\ \bigwedge_{i=1..max(|\delta|,n)} c_i \rightarrow \gamma_i & \text{otherwise} \end{cases} \\
f_2(\delta, n) &= \delta \\
g(\delta, \pi) &= \begin{cases} \mathsf{false} & \text{if } \delta \neq \epsilon \\ (\bigvee_{\gamma_i \notin \pi} c_i) \vee (\bigvee_{\gamma_i \in \pi} \neg c_i) & \text{otherwise} \end{cases}
\end{aligned}$$

We prove that if $T = f_1(\delta, \pi)$ and $q = g(f_2(\delta, \pi), \pi)$, then $\langle \delta, \pi \rangle \in \epsilon$UNSAT if and only if $\langle T, q \rangle \in$ CI.

Suppose $\langle \delta, \pi \rangle \in \epsilon$UNSAT. By definition, $\delta = \epsilon$, hence we have

$$\begin{aligned}
T \wedge \neg q &= \{c_i | \gamma_i \in \pi\} \wedge \{\neg c_i | \gamma_i \notin \pi\} \wedge \{c_i \rightarrow \gamma_i\} \\
&= \bigwedge \{c_i | \gamma_i \in \pi\} \wedge \bigwedge \{\neg c_i | \gamma_i \notin \pi\} \wedge \pi
\end{aligned}$$

Now, π is unsatisfiable, thus $T \wedge \neg q$ is unsatisfiable too. This implies $T \models q$.

Suppose $\langle \delta, \pi \rangle \notin \epsilon$UNSAT. This implies either $\delta \neq \epsilon$ or π satisfiable. In the first case, we obtain $T = \mathsf{true}$ and $q = \mathsf{false}$, so $\langle T, q \rangle \notin$ CI. In the second case, we have

$$T \wedge \neg q = \bigwedge \{c_i | \gamma_i \in \pi\} \wedge \bigwedge \{\neg c_i | \gamma_i \notin \pi\} \wedge \pi$$

Since π is satisfiable, we have a model of $T \wedge \neg q$. This is a model of T that does not imply q. Hence $T \not\models q$. This proves that ϵUNSAT $\leq_{nu-comp}$ CI. □

Theorem 9 comp-C $\subseteq$ comp-C$'$ *iff* C $\subseteq$ C$'$.

Proof. Let us assume C $\subseteq$ C$'$, and let S be a problem that belongs to comp-C. We prove $S \in$ comp-C'. By definition, there exist a poly-size function f and a C language S' such that

$$\langle x, y \rangle \in S \quad \text{iff} \quad \langle f(x), y \rangle \in S'$$

Since C $\subseteq$ C', the language S' is also a C$'$ language. As a result, the above identity is indeed a proof of comp-C$'$ membership for S.

Let us assume comp-C $\subseteq$ comp-C'. Let $S \subseteq \Sigma^*$ be a C-complete problem (w.r.t. polynomial reductions). We prove that $S \in$ C', thus proving that C $\subseteq$ C'.

By Theorem 3, ϵS is in comp-C, thus it is also in comp-C$'$ by hypothesis, thus there exist a poly-size function f and a C$'$ language S' such that

$$\langle x, y \rangle \in \epsilon S \quad \text{iff} \quad \langle f(x), y \rangle \in S'$$

In order to prove $S \in$ C$'$, we reduce it to S'. Indeed,

$$y \in S \quad \text{iff} \quad \langle \epsilon, y \rangle \in S \quad \text{iff} \quad \langle f(\epsilon), y \rangle \in S'$$

Since $f(\epsilon)$ is a constant, S can be polynomially reduced to S', thus S is in C$'$. Since S is C-complete, we obtain C $\subseteq$ C'. $\square$

Theorem 10 nu-comp-C $\subseteq$ nu-comp-C$'$ *iff* C/poly $\subseteq$ C$'$/poly.

Proof. Let us assume C/poly $\subseteq$ C$'$/poly, and let S be a C-complete problem. This implies that ϵS is nu-comp-C-complete and that $S \in$ C'/poly. Thus, it holds that $\epsilon S \in$ nu-comp-C'. As a result, there is a nu-comp-C-complete problem that belongs to nu-comp-C$'$. Thus, nu-comp-C $\subseteq$ nu-comp-C$'$.

Let now assume nu-comp-C $\subseteq$ nu-comp-C$'$. Let S be a generic C/poly problem. We prove that S is also C$'$/poly. By definition there exist a poly-size function f and a C language S' such that

$$y \in S \quad \text{iff} \quad \langle f(|y|), y\rangle \in S'$$

As a result, the language of pairs ϵS is in nu-comp-C. Therefore, it is also in nu-comp-C$'$ and there exist a poly-size function f and a C$'$ language S'' such that

$$\langle x, y\rangle \in \epsilon S \quad \text{iff} \quad \langle f(x, |y|), y\rangle \in S''$$

As a consequence, we have

$$y \in S \quad \text{iff} \quad \langle \epsilon, y\rangle \in \epsilon S \quad \text{iff} \quad \langle f(\epsilon, |y|), y\rangle \in S''$$

Since $f(\epsilon, |y|)$ is indeed a poly-size function of $|y|$ alone, and S'' is in C$'$, we conclude that S is in C$'$/poly. $\square$

Theorem 11 *The following relations between classes hold for every uniform and decidable complexity class* C *(where $\subset$ denotes strict containment):*

- C $\subset$ comp-C
- C $\subset$ C/poly
- comp-C $\subset$ nu-comp-C

Proof. All containments trivially hold by definition. We must show that three of the containments (C $\subset$ comp-C, C $\subset$ C/poly and comp-C $\subset$ nu-comp-C) are strict.

Let HALT be the following (undecidable) language

$$\text{HALT} = \{x \mid \text{ the Turing machine associated with } x \text{ always terminates } \}$$

Using this base language, we define:

$$\text{HALT}\epsilon = \{\langle x, y\rangle \mid x \in \text{HALT}, y = \epsilon\}$$

Notice that HALTϵ does not belong to any class C, while it belongs to all classes comp-C. Hence, we have that C $\subset$ comp-C. Consider now the following language:

$$S_1 = \{\langle x, y\rangle \mid |x| \in \text{HALT}\}$$

Notice that S is in C/poly, but not in C. Hence, C $\subset$ C/poly. Consider now the "reverse" language S_2:

$$S_2 = \{\langle x, y\rangle \mid |y| \in \text{HALT}\}$$

We prove that S_2 is in nu-comp-C, but not in comp-C. The first part is easy to prove, since

$$\langle x, y\rangle \in S_2 \quad \text{iff} \quad \langle f(x, |y|), y\rangle \in S'$$

where f is the function that gives 1 if the second argument is in HALT, 0 otherwise, and S' is the language that accept exactly the pairs that have 1 in the first place. Clearly, f is poly-size and S' is polynomial.

Now, let us assume that S_2 is in comp-C. Then we have a poly-size function f and a C language S such that

$$\langle x, y\rangle \quad \text{iff} \quad \langle f(x), y\rangle \in S'$$

Given a generic string s, we can decide whether $s \in \text{HALT}$ in the following manner: let g be the function that, given a string s, gives a string of n ones, where n is the number associated to s. Now,

$$s \in \text{HALT} \quad \text{iff} \quad \langle \epsilon, g(s)\rangle \in S_2 \quad \text{iff} \quad \langle f(\epsilon), g(s)\rangle \in S'$$

Since $f(\epsilon)$ is a constant and $g(s)$ is decidable, we have reduced an undecidable problem to S', which is by hypothesis a class of the polynomial hierarchy, thus decidable. Hence, S_2 is not in comp-C. □

Theorem 12 *For each class* C *such that there exists at least a language* $L \notin$ C/poly, *it holds* C/poly $\subset$ nu-comp-C.

Proof. Let L be the language that does not belong to C/poly. Consider the language

$$L' = \{\langle x, y\rangle \mid x \in L\}$$

This second language L' is not in C/poly, either. Suppose there exist a poly-size function f and a C language S such that

$$\langle x, y\rangle \in L' \quad \text{iff} \quad \langle f(|x| + |y|), \langle x, y\rangle\rangle \in S$$

Then, L would be in C/poly as well. Indeed, just consider that

$$\begin{array}{rll} x \in L & \text{iff} & \langle x, \epsilon \rangle \in L' \\ & \text{iff} & \langle f(|x| + |\epsilon|)), \langle x, \epsilon \rangle \rangle \in S \\ & \text{iff} & \langle f(|x|), \langle x, \epsilon \rangle \rangle \in S \end{array}$$

Now, take

$$S' = \{ \langle a, b \rangle \mid \langle a, \langle b, \epsilon \rangle \rangle \in S \}$$

which is in C. As a result,

$$x \in L \quad \text{iff} \quad \langle f(|x|), x \rangle \in S'$$

where f is poly-size and S' is in C. This implies $L \in$ C/poly. Since this is false by hypothesis, it holds $L' \notin$ C/poly.

On the other hand, L' is in nu-comp-C. Indeed, let

$$f(x, k) = \begin{cases} 1 & \text{if } x \in A \\ 0 & \text{otherwise} \end{cases}$$

and $S = \{ \langle 1, y \rangle \mid y \text{ is any string } \}$. It holds $\langle x, y \rangle \in B$ if and only if $\langle f(x), y \rangle \in S$, where f is polysize. As a result, B is in nu-comp-C, while it is not in C/poly. □

Theorem 13 *If the polynomial hierarchy is proper, then for each class* C *of the polynomial hierarchy it holds* C/poly $\subset$ nu-comp-C*.*

Proof. If the polynomial hierarchy is proper, then (cf. [29]) also the non-uniform polynomial hierarchy is proper. Hence for each class C of the polynomial hierarchy there exists a language L such that $L \notin$ C/poly. □

A.2 Proofs of Section 3

Theorem 17 CMI *is* nu-comp-Π_2^p*-complete.*

Proof. We give a $\leq_{nu-comp}$ reduction of a nu-comp-Π_2^p-complete problem. The prototypical Π_2^p-complete problem is deciding the truth of a (restricted form of) quantified boolean formula. More precisely, given two sets of propositional atoms $X = \{x_1, \ldots, x_n\}$, $Y = \{y_1, \ldots, y_m\}$, and a 3CNF formula E containing literals on the alphabet $X \cup Y$, we call a $\forall\exists$-QBF the following quantified boolean formula F:

$$F = \forall x_1, \ldots, x_n \exists y_1, \ldots, y_m . E \tag{2}$$

We call E the *matrix* of F. By Theorem 7, we have that the problem $\epsilon\forall\exists$-QBF is nu-comp-Π_2^p-complete. Thus, we now show a reduction from $\epsilon\forall\exists$-QBF to CMI.

To simplify the definition of the three functions f_1, f_2, g needed in the $\leq_{nu-comp}$ reduction, we assume the following: in any $\forall\exists$-QBF F the existentially quantified variables (the set Y) have indices from 1 to (at most) $|F|$, and the universally quantified variables (the set X) have indices from $|F|+1$ to $2|F|$. Observe that this is always possible, since a formula F cannot use more than $|F|$ variables. From now on, we assume that $|F| = |X| = |Y| = k$.

Let C be a set of new atoms, one for each three-literals clause over $X \cup Y$, i.e., $C = \{c_i \mid \gamma_i \text{ is a three-literals clause of } X \cup Y\}$. Moreover, let D be a set of new atoms in one-to-one correspondence with atoms of C, and let Z be as above. Finally, let W be the set of atoms $C \cup D \cup \{u\}$, where u is a distinguished atom. To avoid conflicts, we assume that the indices of all these new variables are greater than $2k$. However, we keep the notation z_i for a variable in Z to highlight the correspondence between z_i and x_i, and the same for the variables in C and D.

We now define a formula imposing non-equivalence between atoms in X and their correspondents in Z, and the same for C and D. We call Σ the 2CNF formula made up by the following clauses:

1. for each atom x_i of X, there are two clauses $x_i \vee z_i$, and $\neg x_i \vee \neg z_i$ in Σ,
2. for each atom c_i of C, there are two clauses $c_i \vee d_i$, and $\neg c_i \vee \neg d_i$ in Σ,

Now we want to code every possible 3CNF formula over $X \cup Y$, using the atoms in C as "enabling gates". We call Γ the 4CNF formula containing, for each three-literals clause γ_i over $X \cup Y$, a clause $\gamma_i \vee \neg c_i$.

We define $f_1(\delta, k)$ as:

$$f_1(\delta, k) \quad = \quad \begin{cases} \mathsf{true} & \text{if } \delta \neq \epsilon \\ T_k & \text{otherwise} \end{cases}$$

where T_k is defined as :

$$T_k = \Sigma \wedge ((u \wedge y_1 \wedge \cdots \wedge y_n) \vee (\neg u \wedge \Gamma)). \tag{3}$$

Note that the size k of F is needed to build T_k, because Σ, Γ and the set Y have a size that depends on k. Observe that the size $|T_k|$ is $O(k^3)$, and that T_k can be rewritten as an equivalent 5CNF formula. Moreover, T_k does not depend on a specific $\forall\exists$-QBF F, but only on its size.

The function f_2 is defined as $f_2(\delta, k) = \delta$, i.e., f_2 is not needed in the reduction. The function g is defined as:

$$g(f_2(\delta, k), F) = g(\delta, F) \quad = \quad \begin{cases} \mathsf{false} & \text{if } \delta \neq \epsilon \\ Q_F & \text{otherwise} \end{cases}$$

where Q_F is defined as:

$$Q_F = \neg u \vee \neg y_1 \vee \cdots \vee \neg y_m \vee (\bigvee_{c_i \in C_E} \neg c_i) \vee (\bigvee_{c_i \in \overline{C_E}} c_i) \vee (\bigvee_{d_i \in D_E} d_i) \vee (\bigvee_{d_i \in \overline{D_E}} \neg d_i)$$

Observe that Q_F contains all atoms of $Y \cup W$.

We now show that $\langle \delta, F \rangle \in \epsilon\forall\exists$-QBF iff $\langle f_1(\delta, |F|), g(\delta, F) \rangle \in$ CMI. For the case $\delta \neq \epsilon$, the pair $\langle \delta, F \rangle$, which does not belong to $\epsilon\forall\exists$-QBF, is transformed into $\langle \mathsf{true}, \mathsf{false} \rangle$ which does not belong to CMI. For the case $\delta = \epsilon$, we show that $\langle \epsilon, F \rangle \in \epsilon\forall\exists$-QBF iff $\langle T_{|F|}, Q_F \rangle \in$ CMI. From now on, we omit the subscript of T and Q to improve readability.

Given the $\forall\exists$-QBF F with matrix E, we denote

$$C_E = \{c_i \in C \mid \gamma_i \text{ is a clause of } E\}$$

and similarly for D_E. Moreover, $\overline{C_E} = C - C_E$, and $\overline{D_E} = D - D_E$. We denote with $\mathcal{M}(T)$ the models of T, and similarly with $\mathcal{M}(\Sigma)$ the models of Σ. Given $M \in \mathcal{M}(\Sigma)$, we define its extension $\mathcal{E}(M) = M \cup Y \cup \{u\}$.

We first prove the following properties of T.

Proposition 18 *Let T be as in (3). Then:*

1. *T is always satisfiable.*
2. *If $M \in \mathcal{M}(\Sigma)$ then $\mathcal{E}(M) \in \mathcal{M}(T)$.*
3. *Every model $M \in \mathcal{M}(T)$ defines a unique matrix $E = \{\gamma_i \mid c_i \in M\}$.*
4. *For every model $M \in \mathcal{M}(T)$ such that $M \cap C = C_E$, M satisfies Γ iff M satisfies E.*

Proof.

1. Let $M = D \cup X$. Obviously $M \models \Sigma$ and $M \models \neg u \wedge \Gamma$. Therefore $M \models T$.
2. M already satisfies Σ; moreover, $Y \cup \{u\}$ satisfies the first disjunct in $(u \wedge y_1 \wedge \cdots \wedge y_n) \vee (\neg u \wedge \Gamma)$.
3. Obvious.
4. M satisfies Γ iff it satisfies all the clauses $\gamma_i \vee \neg c_i$ of Γ. Since $M \cap C = C_E$ and each clause of Γ contains a different $\neg c_i$, each clause is satisfied iff either it contains a $\neg c_i$ such that $c_i \in \overline{C_E}$ or the remaining part of the clause (γ_i) is satisfied. But the conjunction of these remaining clauses is exactly E.

□

We now prove that F is valid iff Q is true in all minimal models of T.

If part. Suppose that Q is true in all minimal models of T. Consider any model $M \in \mathcal{M}(\Sigma)$ such that $M \cap C = C_E$, that is, M contains exactly the atoms of C corresponding to clauses of E. Note that the model $\mathcal{E}(M) = M \cup Y \cup \{u\}$ cannot be a minimal model, because it does not satisfy Q. Hence for any such M there exists a different extension $\mathcal{E}_1(M)$, with $\mathcal{E}_1(M) \subset \mathcal{E}(M)$, that is a minimal model of T satisfying Q. Since $\mathcal{E}_1(M)$ satisfies Q, it must satisfy $\neg u$. As a consequence, $\mathcal{E}_1(M) \models \Gamma$ (Cf. (3)). But any model satisfying Γ and whose intersection with C equals C_E satisfies also E (Cf. Proposition 18 point 4). Hence for any $M \in \mathcal{M}(\Sigma)$ such that $M \cap C = C_E$, there is an extension $\mathcal{E}_1(M)$ satisfying E. Since each $M \in \mathcal{M}(\Sigma)$ contains a truth assignment to variables in X, it follows that for each assignment to variables in X there is an assignment to variables in Y (namely, $\mathcal{E}_1(M) \cap Y$) such that E is satisfied. Therefore, F is valid.

Only if part. Assume that there exists a minimal model M of T such that $M \models \neg Q$. Observe that $M \models u \wedge y_1 \wedge \cdots \wedge y_n$, $M \cap C = C_E$, $M \cap D = \overline{D_E}$. Let $M_\Sigma = M \cap (X \cup Z \cup C \cup D)$. Obviously, M is an extension of M_Σ. The minimality of M implies that $M \not\models \Gamma$, otherwise $M - \{u\}$ would be a model of T. Since $M \cap C = C_E$ and $M \not\models \Gamma$, by Proposition 18 (point 4) we also have that $M \not\models E$. Again from the minimality of M, it follows that no other extension M' of M_Σ satisfies T. As a consequence, there exists an assignment to the variables in X (i.e., $M_\Sigma \cap X$) for which there is no assignment to the variables in Y that makes E true. Therefore F is not valid. □

A.3 Proofs of Section 5

Proposition 19

- $\mathfrak{G}(x)$ *can be computed in time polynomial in the size of* x.
- $\mathfrak{H}(k)$ *can be computed in time polynomial in the size of* k *(i.e., in time* $\log k$*).*
- $\mathfrak{H}(\mathfrak{G}(x)) = x$ *for any* x.

Proof. Let $\gamma = l_{i_1} \vee l_{i_2} \vee l_{i_3}$ be a clause, where l_{i_j} is a literal. We associate a number to each literal in the following way:

$$\begin{array}{rcl} \neg x_1 & \rightarrow & 0 \\ x_1 & \rightarrow & 1 \\ \neg x_2 & \rightarrow & 2 \end{array}$$

$$\begin{aligned} x_2 &\rightarrow 3 \\ &\vdots \end{aligned}$$

that is, a negative literal $\neg x_i$ corresponds to $2(i-1)$, while a positive one x_i corresponds to $2i-1$. Note that this is a bijection.

Each clause is associated to a 3-tuple of numbers, therefore each 3CNF formula x can be represented with a sequence of 3-tuples. Now we show how such a sequence can be represented with a number. Let

$$[(x_{11}, x_{12}, x_{13}), \ldots, (x_{k1}, x_{k2}, x_{k3})]$$

be a sequence of 3-tuples. Represent each number x_{ij} with its auto-delimited string s_{ij}, that is, if the binary representation of x_{ij} is $b_1 \ldots b_z$, then $s_{ij} = b_1 0 b_2 0 b_3 0 \ldots b_z 1$. The number associated to the sequence is

$$rev(s_{11} \cdot s_{12} \cdot s_{13} \ldots \cdot s_{w1} \cdot s_{w2} \cdot s_{w3})$$

where rev is the operation of reverse of a string, and $(\cdot)$ represents string concatenation.

Let $\mathfrak{G}(x)$ be the number associated to x via the above procedure. Notice that each 3CNF formula x is associated to a number, but there are numbers that do not correspond to any x. We define a function $\mathfrak{H}$ that associates to any number a 3CNF formula.

Let k be a number. Consider its binary representation $k_1 \ldots k_m$. If m is odd, add a 1 on the head. Revert this string:

$$s = k_m \ldots k_1 \{1\}$$

Now, consider s as a concatenation of auto-delimited strings, that is, let

$$s = b_{11} 0 b_{12} 0 b_{13} 0 \ldots b_{1z_1} 1 \ldots b_{w1} 0 b_{w2} 0 b_{w3} 0 \ldots b_{wz_w} 1$$

In this manner, any k corresponds to a sequence of w numbers. If w is not a multiple of three, add one or two zeros to the sequence. As a result, we have a sequence of 3-tuples of numbers. At this point, we can associate a clause of three literals to any 3-tuple, by using the literal-number bijection. Let $\mathfrak{H}(k)$ be the 3CNF formula associated to this sequence of clauses. This concludes the proof.

Notice that the inverse of the last item, that is $\mathfrak{G}(\mathfrak{H}(k)) = k$ does not necessarily hold. □

Proposition 20 FCI *is* nu-comp-coNP-*hard.*

Proof. We reduce the problem CI to FCI via a $\leq_{nu-comp}$ reduction. We exhibit two poly-size functions f_1 and f_2, and a poly-time function g such that

$$\langle a, b\rangle \in \text{CI} \;\text{ iff }\; \langle g(f_2(|a|), b), f_1(a)\rangle \in \text{FCI}$$

Notice that a (the first string of the pair) is the fixed part of the problem CI, while the fixed part of a pair $\langle x, k\rangle$ of the language FCI is k, that is, the second element of the pair.

We define these three functions as

$$\begin{aligned} f_1(a) &= \mathfrak{G}(a) \\ f_2(n) &= \epsilon \\ g(c, d) &= d \end{aligned}$$

As a result

$$\langle g(f_2(|a|), b), f_1(a)\rangle = \langle b, \mathfrak{G}(a)\rangle$$

Suppose $\langle a, b\rangle \in$ CI, that is $a \models b$. Since $\mathfrak{H}(\mathfrak{G}(a))$ is equal to a by property 1, $\mathfrak{H}(\mathfrak{G}(a)) \models b$. As a result, $\langle b, \mathfrak{G}(a)\rangle \in$ FCI.

Suppose instead that $\langle a, b\rangle \notin$ CI. We have that $a \not\models b$, thus $\mathfrak{H}(\mathfrak{G}(a)) \not\models b$, thus $\langle b, \mathfrak{G}(a)\rangle \notin$ FCI. □

Proposition 21 FCI $\in$ uFPT.

Proof. We exhibit (cf. [11, Definition 2.4(i)]) an algorithm Φ, a constant c, and a function $f : \mathbb{N} \longrightarrow \mathbb{N}$ such that

(a) the running time of $\Phi(\langle Q, k\rangle)$ is at most $f(k)|Q|^c$, and

(b) $\langle Q, k\rangle \in$ FCI iff $\Phi(\langle Q, k\rangle) = 1$.

(Since f is recursive, we actually prove that FCI is strongly uniformly fixed parameter tractable, cf. [11, Definition 2.4(ii)]).

Let $Cl : \Sigma^* \longrightarrow \Sigma^*$ be the "deductive closure" operator in propositional logic, which, given a formula F in conjunctive normal form, returns the set of non-tautologous clauses that are logical consequences of F. (Note that the number of clauses returned by $Cl(F)$ is at most $2^{|F|}$, and each clause has size $\leq |F|$).

Let x be $\mathfrak{H}(k)$. We consider now the problem:

Instance: Q, $Cl(x)$
Output: "yes" iff $x \models Q$, "no" otherwise

Such a problem can be solved in time $|Cl(x)| \cdot O(|Q|)$ in the following way:

1. if Q is a tautology, then return "yes";

   such a test can be done in time $O(|Q|)$.

2. let Q' be obtained from Q by eliminating all propositional letters which do not occur in x;

   Q' can be obtained in time $|Cl(x)| \cdot O(|Q|)$.

3. $x \models Q$ iff $Q' \in Cl(x)$;

   testing whether Q' is equal to a generic element of $Cl(x)$ costs $O(|Q'|) \in O(|Q|)$. Therefore $x \models Q$ can be tested in time $|Cl(x)| \cdot O(|Q|)$.

Now, algorithm Φ works as follows:

1. transforms k into $x = \mathfrak{H}(k)$. x can be computed in $O(\log k)$ time. x has at most $O(\log k)$ clauses.

2. transforms x into $Cl(x)$. This can be done in time $|Cl(x)| \leq |x| \cdot 2^{|x|} \in O(k \cdot \log k)$

3. uses the above algorithm in time $|Cl(x)| \cdot O(|Q|) \in O(k \cdot \log k) \cdot O(|Q|)$.

The total cost of Φ is therefore $O(k) + O(k \cdot \log k) \cdot O(|Q|) \in O(k \cdot \log k) \cdot O(|Q|)$. This concludes the proof. $\square$