Fundamenta Informaticae XX (2019) 1–22
DOI 10.3233/FI-2012-0000
IOS Press

Applying Modern SAT-solvers to Solving Hard Problems

Artur Niewiadomski, Piotr Switalski
Siedlce University, Faculty of Science, Institute of Computer Science,
3-Maja 54, 08-110 Siedlce, Poland
artur.niewiadomski@uph.edu.pl; piotr.switalski@uph.edu.pl

Teofil Sidoruk, Wojciech Penczek*
Institute of Computer Science, Polish Academy of Sciences
Jana Kazimierza 5, 01-248 Warsaw, Poland
t.sidoruk@ipipan.waw.pl, penczek@ipipan.waw.pl

Abstract. We present nine SAT-solvers and compare their efficiency for several decision and combinatorial problems: three classical NP-complete problems of the graph theory, bounded Post correspondence problem (BPCP), extended string correction problem (ESCP), two popular chess problems, PSPACE-complete verification of UML systems, and the Towers of Hanoi (ToH) of exponential solutions. In addition to several known reductions to SAT for the problems of graph k-colouring, vertex k-cover, Hamiltonian path, and verification of UML systems, we also define new original reductions for the N-queens problem, the knight's tour problem, and ToH, SCP, and BPCP. Our extensive experimental results allow for drawing quite interesting conclusions on efficiency and applicability of SAT-solvers to different problems: they behave quite efficiently for NP-complete and harder problems but they are by far inferior to tailored algorithms for specific problems of lower complexity.

1. Introduction

The Boolean satisfiability problem, abbreviated SAT, was the first known NP-complete problem [21]. The efforts towards improving efficiency of SAT-solving algorithms have long passed beyond purely

Address for correspondence: Institute of Computer Science, Siedlce University of Natural Sciences and Humanities, 3-Maja54, 08-110 Siedlce, Poland
*Also affiliated with: Institute of Computer Science, Siedlce University of Natural Sciences and Humanities

academic discourse and currently have many important practical applications, such as verification [1, 2], bounded model checking [6, 7, 18, 41, 23, 29, 42], planning [22], and composition of web services [31] to name only some of them. On the other hand, scientific interest in the topic is by no means fading. As a seminal NP-complete problem, SAT, and efficient solving thereof, is of key importance to the P = NP hypothesis, which, without any exaggeration, can be counted among the most notable unsolved problems of not just information technology, but all modern science.

In this paper, our aim is to present modern SAT-solvers and to compare their efficiency for several decision and combinatorial problems: three classical NP-complete problems of the graph theory, bounded Post correspondence problem (BPCP), extended string correction problem (ESCP), two popular chess problems, verification of UML systems, and the Towers of Hanoi (ToH) problem. We use several known reductions to SAT for the problems of graph k-colouring, vertex k-cover, Hamiltonian path, and verification of UML systems, but also define new original reductions for the N-queens problem, the knight's tour problem, BPCP, ESCP, and the ToH problem. The new reductions are exploited in our experimental studies, but, especially these for BPCP, ESCP, and ToH, which are an important contribution of this paper, are of more general value and could be used in practice for solving these problems. Each reduction is followed by two results stating the complexity of the reduction and its correctness, i.e., that the formula coding the problem is of a given size and it is satisfiable iff the problem has a solution.

It is important to mention that our investigations are dealing not only with NP-complete problems, but also with problems of polynomial complexity (a chess problem) as well as with a PSPACE-complete problem (UML verification) and a combinatorial problem of exponential complexity (the ToH solutions are exponential in the input size). This is motivated by the idea to compare SAT-solvers applied not only to problems for which they have been originally designed, but also to find out how SAT-solvers behave when applied to problems of lower complexity and to a problem for which the optimal recursive algorithm requires an exponential time. Our experimental results show that SAT-solvers do not offer quick solutions to problems of complexity lower than NP.

Additionally, our intention is to showcase the extent of improvements in the area of SAT solving that have occurred since the DPLL algorithm was first published, and especially in the last decade. To this end, the efficiency of modern SAT-solvers is compared against that of older programs. While starting our investigations, we expected the latter to perform noticeably worse than recent, state-of-the-art SAT-solvers. Moreover, we separate satisfiable and unsatisfiable instances of the tested problems, in order to thoroughly analyse efficiency of the SAT-solvers considered.

The rest of this paper is organized as follows. In the next section we present shortly SAT-solving algorithms and modern SAT-solvers. Reductions to SAT for NP-complete and polynomial problems are discussed in Section 3. Reductions to SAT for PSPACE- and EXPTIME- problems are presented in Section 4. In Section 5 experimental results and comparisons are discussed. The final section contains conclusions.

2. Overview of SAT-solvers

In this section we briefly present SAT-solving algorithms and modern SAT-solvers. The original Davis-Putnam algorithm, first presented in 1960 [10], is based upon the resolution rule. It is a theorem proving technique that leads to a proof by contradiction. Thus, the rule is iteratively applied to the negation of the formula until eventually there remains an empty clause (whose logical value is false). The reductions

are equisatisfiable, that is, at every iteration the output formula is satisfiable if and only if so is the input, so the inference must have started at a false formula too, which in turn was a negation, meaning the original formula is satisfiable. On the other hand, the more recent DPLL algorithm [9] employs another approach, i.e., the splitting rule. DPLL is commonly referred to as a modification of the original DP algorithm. Indeed, the only alteration consists in using the splitting rule in place of resolution. However, this has radically changed the nature of the algorithm, turning DPLL into a backtracking scheme.

Though it has been more than half a century since the DPLL algorithm was published, its core idea remains the basis of modern SAT-solvers. Of course, this by no means indicates that the progress made in the last fifty years has been insignificant. Most importantly, the satisfiability problem has matured, with its efficient solving having found specific applications in many production processes, such as the design and formal verification of integrated circuits (including Intel Core CPUs [19]), automated software verification (used by Microsoft, among others [11]), or managing dependencies between optional software components (i.e. plug-ins for the Java Eclipse development environment [24]), and many more. This further bolsters the efforts towards improving the efficiency of SAT-solvers, as well as reinforces general interest in the topic, an example of which are International SAT Competitions, held mostly annually since 2002 and open to all interested developers. Crucially, the projects taking part in the competition are publicly available with full source code and are among the leading SAT-solvers today.

Solver	**Release/Year**	**Notes**
Lingeling	2016	Winner of several recent SAT Competitions.
Plingeling	2016	Parallel version of Lingeling.
Glucose	4.1 (2016)	Originally based on Minisat.
Glucose-syrup	4.1 (2016)	Parallel version of Glucose. Winner of the most recent SAT Competition's Parallel Track (2017).
Clasp	3.3.0 (2017)	Combines ASP (Answer Set Programming) with state-of-the-art CDCL SAT-solving techniques.
Minisat	2.2.0 (2010)	Older, classic solver developed at MIT. Served as base for many other projects, including Glucose.
ManySAT	2.0 (2008)	Variant of Minisat that adds support for parallel processing.
zChaff	2007	Oldest of tested solvers. Winner of the inaugural SAT Competition in 2002.
Z3	4.5.0 (2017)	Powerful SMT-solver developed at Microsoft Research. Also supports DIMACS input.

Table 1. SAT-solvers selected for comparison

Modern SAT-solvers can be classified in two main groups, depending on the algorithm used. The first approach constitutes an evolution of the original DPLL algorithm and is referred to as CDCL (Conflict-Driven Clause Learning). It augments the classic DPLL backtracking scheme with mechanisms such as non-chronological backjumping, clause learning, periodic restarts and various heuristics, both deterministic and non-deterministic. Currently, CDCL algorithms are represented in several state-of-the-art

SAT-solvers, including Lingeling [5], Glucose [3] and Clasp [15]. Next to further improvements in decision-making upon encountering conflicts and generally more efficient search and data structures, parallel processing is another important direction of their development. While still at an early stage, it is already supported by two of the aforementioned projects, namely Lingeling and Glucose, and their number is fully expected to increase in the coming years.

Another possible approach to solving SAT involves the use of local search methods. The algorithm starts by assigning random truth values to propositional variables, and then proceeds to change one in each step, depending on the resulting increase in the number of satisfied clauses. This is continued for a pre-determined number of steps, or until an assignment satisfying all clauses is found. One of the first SAT-solvers to employ local search algorithms was GSAT (Greedy SAT) [36]. WalkSAT [35] is another example of such SAT-solver.

Given the ever-changing landscape of modern SAT-solvers, many of which also exist in multiple code branches, i.e., specifically optimized for parallel processing, it is all but impossible to include every notable SAT-solver in the comparison. The following have been chosen: Lingeling and Plingeling [5], Glucose and Glucose-syrup [3], Clasp [15], Minisat [37], ManySAT [17], Microsoft Z3 [12], and zChaff [27]. All in all, including the parallel versions of Lingeling and Glucose, nine SAT-solvers have been included in the comparison. They are briefly presented in Table 1.

3. Reductions to SAT for problems in P and NP

Below we discuss translations to SAT for 7 problems: three classical NP-complete graph problems [21], the bounded Post correspondence problem, extended string correction problem, and two chess problems.

3.1. Reductions for classical NP-complete problems

We begin with the three NP-complete graph problems, firstly because these are classical problems whose reductions to SAT are well known, and secondly, on account of the encoding for Hamiltonian path being later reused for knight's tour. Thus, translations for graph k-colouring and vertex k-cover are only shortly recalled below; for a full coverage the interested reader is referred to [38]. The reduction to SAT for Hamiltonian path, as well as further reductions featuring our original encodings, are subsequently discussed in more detail.

Graph k-colouring. Given a graph of n vertices, the graph k-colouring problem consists in finding an assignment of one of k possible colours to each vertex of the graph, such that no two adjacent vertices are of the same colour. Let $p_{i,j}$, for $i = 1..n, j = 1..k$, denotes that the i-th vertex has the j-th colour. The number of propositional variables is thus $n * k$. Taking the number of subformulas into account, the encoding proposed in [38] is of size $O(n^2 * k)$, since the number of edges is n^2 for a full, directed graph.

Vertex k-cover problem. Given a graph of n vertices, the vertex k-cover problem consists in finding a subset of vertices of a given size k, such that each edge of the graph is incident to one of them. Let $p_{i,j}$, for $i = 1..n, j = 1..k$, denotes that the i-th vertex has been included to the covering subset on the j-th 'position'. Obviously, the ordering is for the convenience of the encoding only, since every linearisation of a covering subset is a proper solution. Thus, the number of propositional variables used to encode this problem in [38] equals $n * k$ and the resulting formula is of size $O(n^2 * k)$.

Hamiltonian path. The Hamiltonian path problem consists in finding a path (over edges) in a given graph such that it visits each of its vertices exactly once. Below, we recall the required constraints

from [38] on account of the translation being later applied for knight's tour. Let variable $p_{i,j}$, where $i, j \in \{1, 2, ..., n\}$, represent the i-th vertex being at j-th position in the Hamiltonian path. The total number of variables is thus n^2. Below, the formula $\mathcal{P}(n)$ encodes a one-to-one assignment of vertices to positions on the path, and vice versa:

$$\mathcal{P}(n) = \bigwedge_{j=1}^{n}\Big(\bigvee_{i=1}^{n} p_{i,j} \wedge \big(\bigwedge_{k\in\{1..n\}\setminus\{i\}} \neg p_{k,j}\big)\Big) \wedge \bigwedge_{i=1}^{n}\Big(\bigvee_{j=1}^{n} p_{i,j} \wedge \big(\bigwedge_{k\in\{1..n\}\setminus\{j\}} \neg p_{i,k}\big)\Big)$$

Consequently, the whole formula encoding the Hamiltonian path for a graph of n vertices and the neighbourhood relation N (by $N(i)$ we denote the set of the neighbours of the i-th vertex) is as follows:

$$H(n, N) = \mathcal{P}(n) \wedge \bigwedge_{i=1}^{n}\bigwedge_{j=1}^{n-1}\big(p_{i,j} \implies \bigvee_{v\in N(i)} p_{v,j+1}\big)$$

The two subsequent lemmas follow from the construction of the formula $H(n, N)$.

Lemma 3.1. The formula $H(n, N)$ is of size $O(n^3)$.

Proof. The size of $\mathcal{P}(n)$ is $2n^3$, because we iterate three times over n while building the nested formula. The remaining part of $H(n, N)$ can potentially be of size n^3, since in the inner disjunction we have to consider the possibility of up to n neighbours for any given vertex. The entire formula is thus of size $3n^3$, and so is in $O(n^3)$. □

Lemma 3.2. Given a graph of n vertices and the neighbourhood relation N, there exists a Hamiltonian path iff the formula $H(n, N)$ is satisfiable.

Proof. It is easily inferred from the definition of the Hamiltonian path. The subformula $\mathcal{P}(n)$ comprises of two conjunctions of constraints 'at least one' and 'at most one', thus representing selections of 'exactly one', first assigning vertices to positions on the path, and then vice versa. The following part of $H(n, N)$ ensures that the assignment from $\mathcal{P}(n)$ is indeed a valid Hamiltonian path, that is, every vertex but the last one[1] has a successor within its neighbourhood, as per the definition. Because we have a one-to-one assignment of vertices to positions on the path, and subsequent vertices have successors within their neighbourhoods, all requirements for a Hamiltonian path to exist are met. □

3.2. Reduction for bounded Post correspondence problem

Post correspondence problem (PCP) was originally introduced in [34]. There are several equivalent formulations of PCP. One of them is as follows. Given a finite alphabet Σ containing at least two symbols, let Σ^+ be the set of all non-empty words over Σ. Let $W = (w_1, \ldots, w_n)$, $V = (v_1, \ldots, v_n)$, where $\forall_{i=1..n}(w_i, v_i) \in \Sigma^+ \times \Sigma^+$, be two non-empty, finite sequences of n words of Σ^+. The problem consists in finding a sequence of indices $(i_1, \ldots, i_k)$, such that $\overline{w} = w_{i_1} \cdot w_{i_2} \cdot \ldots \cdot w_{i_k}$, $\overline{v} = v_{i_1} \cdot v_{i_2} \cdot \ldots \cdot v_{i_k}$, and $\overline{w} = \overline{v}$. Intuitively, the problem is to find a sequence of indices for which the concatenation of the corresponding words of the lists W and V are equal. In general, when there is no upper bound on a

[1]The last vertex is omitted as we consider a Hamiltonian path and not necessarily a circuit.

solution length k, PCP is undecidable, as shown by Post in [34]. Thus, we deal with a bounded version of PCP (BPCP, for short), where k is bounded. BPCP is in NP [14] and was also studied in different contexts, for example, using DNA-based bio-computations [20]. An instance of PCP can be visualised using a list of tiles similar to domino bricks where the i-th tile contains the word w_i on the top, and the word v_i at the bottom. Each tile has assigned an id equal to the position of the tile in the input list and we assume that at least k copies of each tile is at our disposal. Thus, a solution is the sequence of tile's ids, and a single element of the sequence is also called a *solution step*. An example instance and a solution is depicted in Fig. 1.

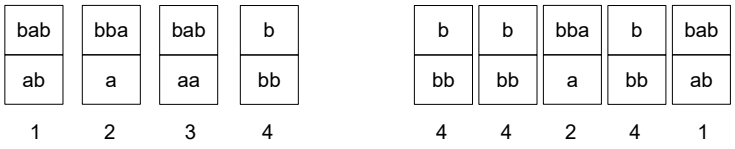

Figure 1. An example instance of PCP, for $n = 4$ and $\Sigma = \{a, b\}$, $W = (bab, bba, bab, b)$, $V = (ab, a, aa, bb)$. The solution $(4, 4, 2, 4, 1)$ corresponds to the word $\overline{w} = \overline{v} = bbbbabbab$.

Translation to SAT. In addition to the assumptions above, we restrict the length of each word of W and V to some $m \in \mathbb{N}_+$. In this setting, for a given k, the resulting words $\overline{w}$ and $\overline{v}$ are of length $l \in \mathbb{N}_+$, where $k \leq l \leq m * k$. To avoid ambiguity, we refer to the words of W and V as to *elements*, while the concatenated word $\overline{w}$ and $\overline{v}$ is called the 'upper' and the 'bottom' word, respectively. Moreover, to enable the standard binary encoding, we treat the symbols of Σ as the consecutive natural numbers $\{0, \ldots, |\Sigma| - 1\}$.

Let PV be a set of propositional variables used to encoding the problem as a Boolean formula. In order to keep our presentation as clear as possible, we do not show the very details of the binary encoding over propositional variables. Instead, we use propositions of PV to form *symbolic variables*, i.e., vectors of propositional variables, (often called *bitvectors*), over which natural numbers are binary encoded[2]. To this aim, we use expressions of the form $(variable := value)$. For example, let $\mathbf{x}$ be a bitvector (x_1, x_2).Thus, the value 2 ($(10)_2$) is encoded over $\mathbf{x}$ by $(x_1 \wedge \neg x_2)$, which is denoted by $(\mathbf{x} := 2)$. Moreover, we group symbolic variables into vectors, and refer to the single ones using indices. Let $\mathbf{w}$ and $\mathbf{v}$ be vectors of $k * m$ symbolic variables to represent numbers corresponding to the alphabet symbols, let vectors $\mathbf{p}^w$ and $\mathbf{p}^v$ (both of size $k + 1$) encode positions in words $\overline{w}$ and $\overline{v}$, respectively, whereas $\mathbf{id}$ is a vector of k symbolic variables encoding a solution, so a sequence of tile ids[3]. Thus, for example, by $\mathbf{w}_i$ and $\mathbf{v}_i$, for $i = 1..k * m$, we denote the symbolic variables encoding consecutive letters of words $\overline{w}$ and $\overline{v}$, respectively. In other words, $\mathbf{w}_i$ and $\mathbf{v}_i$ can be seen as 'slots' for numbers representing the alphabet letters, and the variables $\mathbf{id}_j$ as 'slots' for numbers corresponding to tile ids.

Thus the total number of symbolic variables used in our encoding equals $2km + 2(k + 1) + k = 2km + 3k + 2$, so it is in $O(km)$. After multiplying it by the length of the longest bitvector: $b =$

[2]We use the standard binary encoding, similar to the one shown in equation (5).

[3]In the simplest case, when Σ contains only two symbols (letters), the bitvector encoding a single symbol consists of only one propositional variable. However, to encode positions in a word we need $\lceil log_2(k * m) \rceil$ propositional variables, while to encode a tile id we need $\lceil log_2(n) \rceil$ propositions.

$max(\lceil log_2(km) \rceil, \lceil log_2(n) \rceil)$, we have $O(kmb)$, but since in most cases we have $km > n$, the number of propositional variables is in $O(km * log(km))$.

Now, we are at a position to show the reduction of BPCP to SAT in a top-down manner. Overall, the formula encoding BPCP is as follows.

$$bpcp(k, m, W, V) = \bigvee_{l=k}^{m*k} \Big(vectorsEq(l, k) \wedge \bigwedge_{j=1}^{k} \big(word(j, W, \mathbf{w}, \mathbf{p}^w, \mathbf{id}) \wedge word(j, V, \mathbf{v}, \mathbf{p}^v, \mathbf{id}) \big) \Big) \quad (1)$$

where $vectorsEq(l, k)$ is explained later, and $word(j, A, \mathbf{u}, \mathbf{p}, \mathbf{id})$ is a formula encoding the choice of the j-th solution step and its consequences. The symbols of the element corresponding to the j-th solution step are encoded over symbolic vector $\mathbf{u}$, between positions indicated by variables $\mathbf{p}_j$ and $\mathbf{p}_{j+1}$:

$$word(j, A, \mathbf{u}, \mathbf{p}, \mathbf{id}) = \bigvee_{i=1}^{|A|} \Big((\mathbf{id}_j := i) \wedge \bigvee_{p=j}^{(j-1)*m+1} \Big((\mathbf{p}_j := p) \wedge (\mathbf{p}_{j+1} := p + |a_i|) \wedge \bigwedge_{x=1}^{|a_i|} (\mathbf{u}_{p+x-1} := at(a_i, x)) \Big) \Big) \quad (2)$$

where a_i is the i-th word of the sequence A, $|a_i|$ denotes the length of the word a_i, and $at(a_i, x)$ is the natural number denoting the x-th symbol of a_i. Note that this formula is used twice in (1). It exploits $\mathbf{w}$ and $\mathbf{p}^w$ to encode symbols of $\overline{w}$ and the positions of its components, and, respectively, $\mathbf{v}$ and $\mathbf{p}^v$ to encode $\overline{v}$. However, both instances use the same vector ($\mathbf{id}$) to encode a solution. The formula $vectorsEq(l, k)$ encodes the equality of the vectors $\mathbf{w}$ and $\mathbf{v}$ up to length l, as well as the fact that the vectors begin at position 1 and end at position l.

$$vectorsEq(l, k) = \bigwedge_{i=1..l} (\mathbf{w}_i \Leftrightarrow \mathbf{v}_i) \wedge (\mathbf{p}^w_1 := 1) \wedge (\mathbf{p}^v_1 := 1) \wedge (\mathbf{p}^w_{k+1} := l + 1) \wedge (\mathbf{p}^v_{k+1} := l + 1) \quad (3)$$

Lemma 3.3. Given k, m, V, and W, where $|W| = |V| = n$. The size of the formula $bpcp(k, m, W, V)$ is in $O(k^3m^3n)$ and in $O(k^4m^4)$ if $km > n$.

Proof. Note that for a given l the size of the formula (3) is $5l$, so it is in $O(l) = O(mk)$, because $(m - 1)k$ is the highest value of l. On the other hand, for a given j, the size of the formula (2) equals to $n * (((j - 1) * m + 1 - j) * (2 + m))$, so it is in $O(m^2jn)$, and since the highest value of j is k, we have $O(m^2kn)$. Thus, for the whole formula $bpcp(k, m, W, V)$ we have $O(km * (km + k * m^2kn)) = O(k^2m^2 + k^3m^3n) = O(k^3m^3n)$. □

Theorem 3.4. Given k, m, V, and W. BPCP has a solution iff $bpcp(k, m, W, V)$ is satisfiable.

Proof. Notice that we consider the input words of length from 1 to m, so the concatenation of k such words is of length from k to $m * k$.

($\Longrightarrow$) Assume that for a given k, m, W, V there exists a solution $sol = (s_1, \ldots, s_k)$ which defines the word $\overline{w} = \overline{v}$ of length len. We shall show that $[vectorsEq(len, k) \wedge \bigwedge_{j=1..k} \big(word(j, W, \mathbf{w}, \mathbf{p}^w, \mathbf{id}) \wedge word(j, V, \mathbf{v}, \mathbf{p}^v, \mathbf{id}) \big]$ is satisfied.

Let us analyse the formula $word(j, W, \mathbf{w}, \mathbf{p}^w, \mathbf{id})$, that is the instance of the formula (2) for $A = W$, $\mathbf{u} = \mathbf{w}$, and $\mathbf{p} = \mathbf{p}^w$, i.e., we consider the 'upper' word $\overline{w}$. For a given j, that is, while deciding the j-th solution step the formula contains an alternative choice over all elements of W, as the variable i

ranges over all available ids. Thus, for $i = s_j$, the formula contains also the element of W matching the j-th solution step of sol. Note that the value s_j is encoded over $\mathbf{id}_j$. Next, in order to satisfy the formula (2), at least one disjunct of the inner alternative has to evaluate to $true$. So, there exists some $p \in \{j, \ldots, (j-1) * m + 1\}$ corresponding to the starting position of the j-th element of the 'upper' word. Obviously, this position is the total length of first $j-1$ elements of the solution increased by 1. According to our assumptions, if all $j-1$ elements are of the shortest possible length 1, their total length equals $j-1$ and the next (j-th) element begins at position $j-1+1 = j$. On the other hand, if all previous elements are of the maximal length m, the j-th element starts at position $(j-1) * m + 1$. The inner alternative considers both boundary values and also all values between, thus one of them matches the starting position of the j-th element. Observe, that in the case of the first element, i.e., for $j = 1$, 1 is the only value of p. Then, for the given p, the inner conjunction is satisfied. Moreover, p is encoded over the symbolic variable $\mathbf{p}^w_j$, the starting position of the next ($(j+1)$-th) element is encoded over $\mathbf{p}^w_{j+1}$, and the consecutive symbols of w_{s_j} are binary encoded over the symbolic variables $(\mathbf{w}_p, \ldots, \mathbf{w}_{p+|w_{s_j}|-1})$.

Similarly, we can analyse the formula $word(j, V, \mathbf{v}, \mathbf{p}^v, \mathbf{id})$ for the 'bottom' word. Finally, since sol is a solution, the consecutive len symbols of $\overline{w}$ and $\overline{v}$ match. Thus, $vectorsEq(len, k)$ is satisfied as the consecutive len variables of $\mathbf{w}$ and $\mathbf{v}$ are equal, and the values of $\mathbf{p}^w_1$, $\mathbf{p}^v_1$, $\mathbf{p}^w_{k+1}$, and $\mathbf{p}^v_{k+1}$ required by $vectorsEq(len, k)$ are set by the appropriate instances of the formula (2).
($\Longleftarrow$) This part of the proof follows directly from the structure of the formula $bpcp(k, m, W, V)$. If the formula is satisfiable, then we obtain a valuation M of the symbolic variables which satisfies the formula. Since $vectorsEq(l, k)$ is satisfied by M, we have $M(\mathbf{w}_i) = M(\mathbf{v}_i)$, for $i = 1, \ldots, l$. That is, the upper word is equal to the bottom one, both start at position 1, and end at position l. Observe that the formula (2) implies that the upper word is built out of words of W, while the bottom one out of words of V. For $j = 1, \ldots, k$, the consecutive components of the words start at positions $M(\mathbf{p}^w_j)$ and $M(\mathbf{p}^v_j)$, respectively, and every consecutive component of the upper and of the bottom word is the $M(\mathbf{id}_j)$-th word of the input sequence W, and V, respectively. Thus, we have a solution for a given BPCP instance. □

3.3. Reduction for string correction problem

String correction problem (SCP) is a well-studied class of problems originated from string edit distance. The edit distance defines a set of some simple operators conducted on strings. SCP consists in finding out whether a string A can be transformed into a string B in a finite k edit distance operations. If three single-character operators: $insertion$, $deletion$ or $substitution$, are allowed, then the edit distance is called Levenshtein's one [25]. This problem can be solved in time proportional to the product of the lengths of A and B as shown in [40]. A more interesting extension of edit distance is Damerau-Levenshtein distance [8], where the set of operators is extended with the operator exchanging the positions of any two symbols in a string. This problem is called the *extended string-to-string correction problem* (ESCP). In this case the problem is of the same complexity as before [40]. However, if one limits the available operations to single character deletions and swapping the adjacent symbols only, and allows for using at most k edits, where $k \in \mathbb{N}$, then the problem becomes NP-complete [40].

Formally, the problem is formulated as follows. Given a finite alphabet Σ, two non-empty strings $A, B \in \Sigma^+$, and $k \in \mathbb{N}_+$. Assume that $A = (a_1, \ldots, a_n)$ and $B = (b_1, \ldots, b_m)$, where $a_i, b_j \in \Sigma$ for $i = 1..n$ and $j = 1..m$. The problem is whether one can transform A into B by a sequence of at most k edit operations including *swap* and *deletion*. The swap operator occurs when two consecutive

symbols switch their positions. Deletion is the removal of an individual instance of a symbol from a string, therefore shortening its length by 1. The deleted symbols are replaced with the special value ε denoting the empty character. Note that we have $n \geq m$ as otherwise the problem is unsolvable without an insert operator. Consider an example presented in Fig. 2. A transformation from $A = abcaab$ to

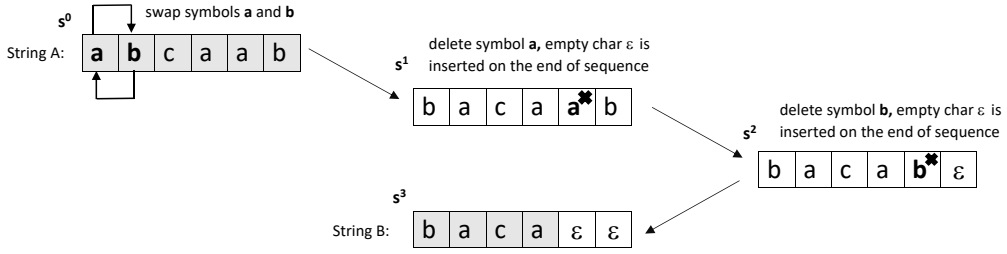

Figure 2. An example of the problem with inputs: A = abcaab, B = baca, and the parameter $k = 3$.

$B = baca$ is obtained by swapping the first two symbols (a and b) of A and then deleting the last two symbols (ab) of A.

Translation to SAT. Let PV be a set of propositional variables needed to encode ESCP. Similarly to encoding of BPCP (see Sec. 3.2), we use vectors of symbolic variables in order to binary encode the alphabet symbols and positions inside words. We need to encode an initial string A and its k copies. To this aim we allocate $k + 1$ vectors of symbolic variables, $\mathbf{s}^i$ for $i = 0..k$, each representing a possible evolution of the string A after applying i edit operations. The strings are of length n, and so the vectors $\mathbf{s}^i$ consist of n symbolic variables, $\mathbf{s}^i_j$, for $j = 1..n$. Going into details, each symbolic variable $\mathbf{s}^i_j$ is a sequence of propositions $s^i_{j,m} \in PV$, where $m = 0..\lceil log_2(|\Sigma + 1|)\rceil$. That is, it contains the number of bits necessary to binary encode all symbols of Σ and ε. Overall, we use $(k + 1) * n * \lceil log_2(|\Sigma + 1|)\rceil$ propositional variables to encode ESCP.

The reduction of ESCP to SAT is presented in a top-down fashion. The whole formula is as follows:

$$escp(k, A, B) = \bigwedge_{i=1}^{n} \Big(\mathbf{s}^0_i := a_i\Big) \wedge \Big(\bigvee_{j=1}^{k} \bigwedge_{i=1}^{m} (\mathbf{s}^j_i := b_i)\Big) \wedge \bigwedge_{j=1}^{k} step(j).$$

The meaning of $:=$ is the same as in the reduction for BPCP. The first part of the formula is a binary encoding of the input word A over the vector $\mathbf{s}^0$. Next, the disjunctions encode the desired states of a transformation after j operations have been performed, that is, the situation when the binary representation of the word B is encoded by vector $\mathbf{s}^j$. Finally, the last part of the formula encodes all possible changes introduced in the consecutive steps, as explained below:

$$step(j) = \bigvee_{p=1..n-1} \Big(del(j, p) \vee swap(j, p)\Big) \vee del(j, n). \tag{4}$$

The formula (4) describes all possible positions of delete and swap operations. The formula encoding a delete operation execued at step j and position p is as follows:

$$del(j, p) = (\mathbf{s}^{j-1}_p \neq \varepsilon) \wedge \bigwedge_{i=1..p-1} \Big(\mathbf{s}^j_i \Leftrightarrow \mathbf{s}^{j-1}_i\Big) \wedge \bigwedge_{i=p..n-1} \Big(\mathbf{s}^j_i \Leftrightarrow \mathbf{s}^{j-1}_{i+1}\Big) \wedge \Big(\mathbf{s}^j_n := \varepsilon\Big)$$

Its first conjunct is the precondition ensuring that there is a non-empty symbol at position p. Then, all symbols up to position p are 'copied' from the previous state, which is encoded as the equivalence of the consecutive $p-1$ symbolic variables of $\mathbf{s}^{j-1}$ and $\mathbf{s}^{j}$. Next, starting from position p, the consecutive symbols are 'copied' from the previous state and shifted left by one position. At the very end the empty symbol is added. Finally, the encoding of swap operation at the j-th step and at position p is defined as:

$$swap(j,p) = \left(\mathbf{s}^{j-1}_{p+1} \neq \varepsilon\right) \wedge \bigwedge_{i=1..n, i\neq p, i\neq p+1} \left(\mathbf{s}^{j}_{i} \Leftrightarrow \mathbf{s}^{j-1}_{i}\right) \wedge \left(\mathbf{s}^{j}_{p} \Leftrightarrow \mathbf{s}^{j-1}_{p+1}\right) \wedge \left(\mathbf{s}^{j}_{p+1} \Leftrightarrow \mathbf{s}^{j-1}_{p}\right),$$

where the first conjunct is a precondition ensuring that a non-empty symbol is affected by the operation, the next part 'copies' all unchanged symbols (i.e., all but those at the positions p and $p+1$), and the last two conjuncts encode swap of the two symbols at positions p and $p+1$.

Lemma 3.5. The formula $escp(k, A, B)$ is of size $O(kn^2)$.

Proof. The formula consists of three conjuncts; the first two are of size n and $k * n$, respectively. For the latter, we bound the number of the inner conjunctions m with n; recall that $n \geq m$ as the problem is otherwise unsolvable without additional operators. Finally, the last subformula is the conjunction over k steps of the ESCP algorithm. Each of these contains the disjunction over n positions in the input string on which operations $swap$ or $delete$ are performed, and whose respective formulas are both of size n. As such, the size of the third conjunct is $k * n^2$. The whole formula $escp(k, A, B)$ is thus of size $n + k * n + k * n^2$, and so is in $O(kn^2)$. □

Theorem 3.6. Given A, B, and k. ESCP has a solution iff the formula $escp(k, A, B)$ is satisfiable.

Proof. ($\Longrightarrow$) Assume there exists a solution, i.e., a sequence of k operations such that A is transformed into B. We will show that the formula $escp(k, A, B)$ is then satisfied. Any valid solution obviously originates from the input string A, represented in our encoding by the initial state vector $\mathbf{s}^0$. Thus, n symbolic variables of $\mathbf{s}^0$ correspond to the respective symbols of A, as enforced in the first conjunct of the main formula $escp(k, A, B)$. Note that the operator $:=$ indicates equality w.r.t. binary encoding. At some point during the subsequent k operations, the desired state where A equals B is reached. As such, there is an operation $j \leq k$, after which all symbolic variables of $\mathbf{s}^j$ binary encode the respective symbols of B. This is represented by the second conjunct of $escp(k, A, B)$. Finally, at each step between these initial and final states, either $delete$ or $swap$ is performed, as permitted by the ESCP conditions. This is reflected in the third and final conjunct of $escp(k, A, B)$. We will now analyse it from the bottom up, starting with the two subformulas encoding operations $delete$ and $swap$. Suppose the character at position p is deleted from A in the j-th operation. If so, then p must be within the non-empty part of the string, as the deletion of the empty character ε is not valid. Furthermore, following the operation, all characters before the deletion point, that is, at $n < p$, are left unchanged, while those after p only have their positions shifted by one to the left. These constraints are encoded in $del(j, n)$. Following a similar pattern, suppose the j-th operation consists of swapping the character at position p. Recall that the $swap$ operation in ESCP is defined as exchanging the positions of two consecutive characters; since the last character has no subsequent, it is only applicable up to and including $p = n-1$. All but the two swapped characters, that is, positions ranging over $\{1, \ldots, p-1\}$ and $\{p+2, \ldots, n\}$, are left unchanged. These constraints are encoded in $swap(j, n)$. The disjunction of $swap(j, n)$ and $del(j, n)$ over the positions

$p \in \{1, \ldots, n\}$ (with the exception of n for the former) constitutes the subformula $step(j)$.
($\Longleftarrow$) Assuming $escp(k, A, B)$ evaluates to true, we can obtain its satisfying assignment, that is, the initial state vector $\mathbf{s}^0$ and the vectors $\mathbf{s}^1 ... \mathbf{s}^k$, corresponding to the state of the input in each of the k steps. They in turn, through binary encoding using n symbolic variables per vector, represent a valid solution of ESCP where the input string A (encoded by $\mathbf{s}^0$) is transformed into B (encoded by $\mathbf{s}^j$) using at most k operations. Note that $escp(k, A, B)$ contains a regular disjunction (as opposed to XOR) of operations $swap(j, n)$ and $del(j, n)$, even though only one of these can be performed in a single step. This is because they already encode the requirement that all characters unaffected by the operation are left unchanged. As such, it is not possible for $swap$ and $delete$ to occur simultaneously in a satisfying assignment of $step(j)$ or, consequently, of the whole formula $escp(k, A, B)$. □

3.4. Reductions for chess problems

In this section, we discuss encodings to SAT for two popular chess problems, N-queens and knight's tour. The former is often incorrectly claimed to be NP-hard or NP-complete [16]. The decision version of N-queens is in fact of constant complexity (since valid arrangements have been proven to exist for all $n \geq 4$), while a single witness can be easily constructed as shown in [4]. However, any solution requires $n \log n$ bits, and as such is not polynomial w.r.t. input size, which is only $\log n$. As for Knight's tour, for arbitrarily sized $n \times m$ chessboards, an algorithm running in time $O(nm)$ is shown in [26].

N-queens. This is a chess problem which consists in placing the n queens on the $n \times n$ chessboard such that they are not attacking one another. Let the assignment of 'true' to variable $p_{i,j}$ represent the placement of a queen in the i-th row and j-th column. Conversely, any empty square is equivalent to its respective variable being assigned 'false'. The formula encoding the N-queens problem is then given as $\mathcal{P}(N) \wedge \mathcal{D}(N)$, where $\mathcal{P}(N)$ is the subformula previously used for the Hamiltonian path encoding (see Sec. 3.1), and $\mathcal{D}(N)$ is given as follows:

$$\mathcal{D}(n) = \bigwedge_{k=1}^{n} \bigwedge_{l=1}^{n} \Big(\bigwedge_{1 \leq i \neq k < n} \bigwedge_{1 \leq j \neq l \leq n-i} \big((\neg p_{j,i+j} \vee \neg p_{l,k+l}) \wedge (\neg p_{i+j,j} \vee \neg p_{k+l,l}) \wedge$$
$$(\neg p_{j,n-1-(i+j)} \vee \neg p_{l,n-1-(k+l)}) \wedge (\neg p_{i+j,n-1-j} \vee \neg p_{k+l,n-1-l}) \big) \Big)$$

The subsequent two lemmas follow from the construction of the formula $\mathcal{P}(N) \wedge \mathcal{D}(N)$.

Lemma 3.7. The formula $\mathcal{P}(N) \wedge \mathcal{D}(N)$ is of size $O(n^4)$.

Proof. As per Sec. 3.1, the size of $\mathcal{P}(N)$ is $2n^3$. However, for $\mathcal{D}(N)$, which has four nested conjunctions, it is not possible to treat the inner conjunction as a single subformula in the same manner because of the presence of the disjunction operator. As such, the size of $\mathcal{P}(N) \wedge \mathcal{D}(N)$ is $2n^3 + n^4$, and the formula is thus in $O(n^4)$. □

Lemma 3.8. The N-queens problem has a solution iff the formula $\mathcal{P}(N) \wedge \mathcal{D}(N)$ is satisfiable.

Proof. Since a queen moves and attacks along rows, columns, and diagonals on the chessboard, there can be at most one per row, column, and diagonal. Moreover, since the number of queens equals the number

of the rows and the columns, it is easily inferred that any valid solution will have *exactly* one queen per row and column. The required 'exactly one' constraint for rows and columns is thus the same as in the case of the Hamiltonian path, and is analogously enforced by the subformula $\mathcal{P}(N)$. For diagonals, the disjunctions in the first line of $\mathcal{D}(N)$ concern diagonals parallel to the main diagonal, while those in the second line involve those parallel to the antidiagonal. Altogether, the conjunction $\mathcal{P}(N) \wedge \mathcal{D}(N)$ covers all required constraints for a valid solution of the N-queens problem. □

Knight's tour is a chess problem which consists in finding a sequence of knight's moves such that it visits every square exactly once. For the purposes of encoding the knight's tour as a propositional formula, it can be considered a specific instance of the Hamiltonian path problem, with the chessboard being represented by a graph of n^2 vertices (each vertex represents a square of the $n \times n$ chessboard) whose are adjacent if and only if a valid knight's move is possible between the respective squares. The reduction to SAT follows exactly as previously described in Sec. 3.1, but we need more (i.e., n^4) variables to encode the consecutive moves: by $p_{i,j}$, for $i, j = 1..n^2$, we denote the j-th move of the knight to position i. Thus, the resulting formula is of size $O(n^4)$.

4. Reductions to SAT for problems in EXPTIME and PSPACE

This section discusses translations to SAT for two problems: one of exponential time complexity and one of polynomial space complexity.

4.1. Towers of Hanoi

The ToH problem [28] is a well-known mathematical puzzle where n discs of different sizes can be placed on one of the three towers. Initially, all the disks are aligned on the first tower, such that the biggest one is on the bottom and every other disc is located on a bigger one. The solution of ToH consists in finding a sequence of disc movements in order to place all of them on another tower preserving their original order. However, every move is restricted only to one of the discs being on the top of a tower, and it cannot be placed on the top of a smaller disc. Below, we present our original propositional encoding of the ToH problem.

The i-th state is represented by the tuple $(d_{1,i}, d_{2,i}, \ldots, d_{n,i})$, where every $d_{j,i} \in \{0, 1, 2\}$, for $j = 1..n$, corresponds to the location of the j-th disc (either on the first (0), second (1), or the third (2) tower). By $D(j, i, t)$ we denote $d_{j,i} = t$, where $j = 1$ stands for the smallest disc, while $j = n$ corresponds to the biggest one, $i = 0$ denotes the initial state, and $t \in \{0, 1, 2\}$ corresponds to a tower. We use the standard binary encoding of integers using two propositional variables:

$$D(j,i,t) = \begin{cases} \neg p_{j_1,i} \wedge \neg p_{j_2,i} \text{ for } t = 0, \\ \neg p_{j_1,i} \wedge p_{j_2,i} \quad \text{for } t = 1, \\ p_{j_1,i} \wedge \neg p_{j_2,i} \quad \text{for } t = 2. \end{cases} \tag{5}$$

The initial and final states are encoded as follows:

$$\mathcal{I} = \bigwedge_{j=1}^{n} D(j, 0, 0), \qquad \mathcal{F} = \bigwedge_{j=1}^{n} D(j, max, 2),$$

where $max = (2^n - 1)$ is the number of moves. All possible moves in the i-th state are encoded as:

$$\mathcal{M}(i) = \bigvee_{t=0}^{2} \Big(\bigvee_{j=1}^{n} \big(move(j,i,t) \wedge \bigwedge_{k \in \{1,\ldots,n\} \setminus \{j\}} d_{k,i} = d_{k,i+1} \big) \Big),$$

where all but the j-th disc remain on the same tower, and the j-th one is moved:

$$move(j,i,t) = pre(j,i,t) \wedge \Big(\bigvee_{t' \in \{0,1,2\} \setminus \{t\}} post(j,i+1,t,t') \Big),$$

$$pre(j,i,t) = D(j,i,t) \wedge \big(\bigwedge_{k=1}^{j-1} \neg D(k,i,t) \big), \quad post(j,i,t,t') = \big(\bigwedge_{k=1}^{j-1} \neg D(k,i-1,t') \big) \wedge D(j,i,t')$$

Thus, the whole formula encoding the ToH problem is as follows:

$$ToH(n) = \mathcal{I} \wedge \mathcal{F} \wedge \bigwedge_{i=0}^{max} \mathcal{M}(i).$$

The subsequent two lemmas follow from the construction of the formula $ToH(n)$.

Lemma 4.1. The formula ToH(n) is of size $O(2^n)$.

Proof. The solution of the ToH puzzle is a sequence of 2^n states, in each of which we have n discs encoded using two propositional variables. Thus, we need $2^n * 2n$ variables, and the formula is of size $O(2^n)$. □

Theorem 4.2. The ToH problem for n discs has a solution iff the formula ToH(n) is satisfiable.

Proof. Let us examine the subformulas used in our encoding, beginning with the initial and final states, denoted by $\mathcal{I}$ and $\mathcal{F}$, respectively. The former is straightforward and ensures that the starting position is correct, i.e. all discs are on the first tower. Note, however, that the latter does not represent the actual end state of a ToH puzzle solution. Instead, it corresponds to a state reached after about half the total moves, where only the largest disc is on the destination tower. This is motivated by the observation, following [28], that all remaining moves can be trivially obtained by mirroring past steps. $\mathcal{M}(i)$ is a disjunction over towers $t \in \{0,1,2\}$ and then discs $j \in \{1,\ldots,n\}$ of possible moves $move(j,i,t)$, in conjunction with an additional constraint ensuring every disc but the selected j-th one remains in place, as per the rules of the ToH puzzle. Thus, $\mathcal{M}(i)$ encodes all possible moves in the i-th state. Individual moves comprise of conjunctions of precondition $pre(j,i,t)$ and postcondition $post(j,i,t,t')$. Intuitively, the former ensures that all required conditions for a valid move are met: firstly, the j-th disc is indeed on the tower t, and secondly, there is no smaller disc on the same tower (and so the $j-th$ one can be moved). Conversely, the latter verifies post-move conditions, specifically, that the j-th disc is no longer on its original tower t, has moved to destination tower t' and is the smallest disc on t'. □

4.2. Model checking of UML systems

Unified Modeling Language (UML) [39] is a graphical specification language. It consists of over a dozen types of diagrams which allow for describing a modelled system from different points of view. Nowadays, model checking techniques are able to verify crucial properties of systems at early stages of design. Here, we recall some results of our previous work resulted in development of a symbolic (bounded) model-checking method operating on systems specified in a subset of UML, and performing a direct translation of all the possible executions of a system (unfolded to a given depth) to SAT.

The tested benchmark is a variant of the well known Generalised Railroad Crossing (GRC). The system, operating a gate at a railroad crossing, consists of a gate, a controller, and N tracks which are occupied by trains. Each track is equipped with sensors that indicate a position of a train and send an appropriate message to the controller. Depending on the track occupancy the controller can either open or close the gate [32]. Due to improper time constraints, the benchmark contains a subtle error which allows a train to enter the crossing while the gate is not yet fully closed, and we test the reachability of such a state.

The reduction to SAT is reported in [32], and presented in more detail in [30]. The number of propositional variables needed to encode the transition relation is polynomial w.r.t. several parameters (like, e.g., the length of the event queues, the number of objects in the system, etc.) except for the case when the multiplication is used in the specification of the system, which makes the encoding exponential, which does not concern GRC.

5. Experimental results and comparisons

In this section we discuss experimental results and compare the efficiency of the SAT-solvers applied. The test platform was a desktop PC with a quad-core Intel Core i5-3350 CPU operating at 3.1 Ghz, without HyperThreading support. The system had 16 GB of RAM and was running Windows 7 Professional. For time measurement, built-in statistics were used for every SAT-solver. Given the computational complexity of SAT, not all test instances were expected to be processed in a reasonable amount of time. It is of concern especially in the case of older, classic SAT-solvers, i.e., zChaff, as well as unsatisfiable instances of selected problems. The timeout was set at 60 minutes.

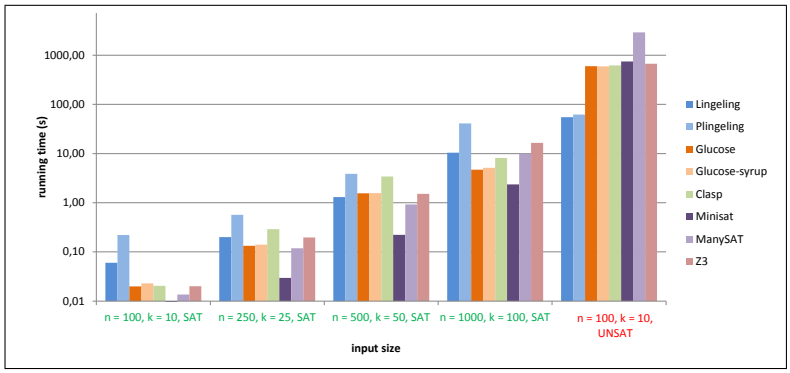

Figure 3. Average results for the graph k-colouring problem, grouped by input size

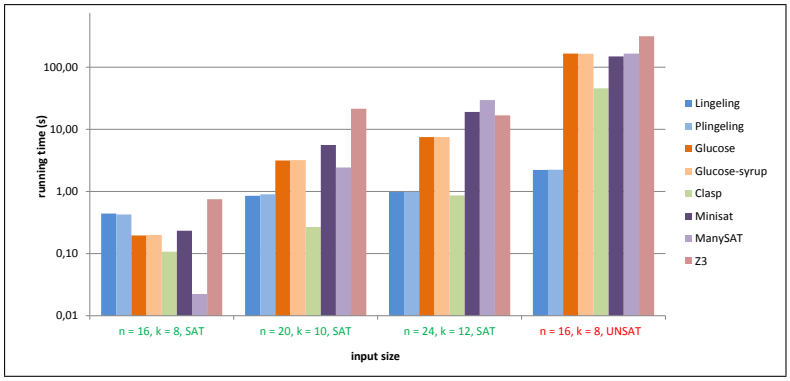

Figure 4. Average results for the vertex k-cover problem, grouped by input size

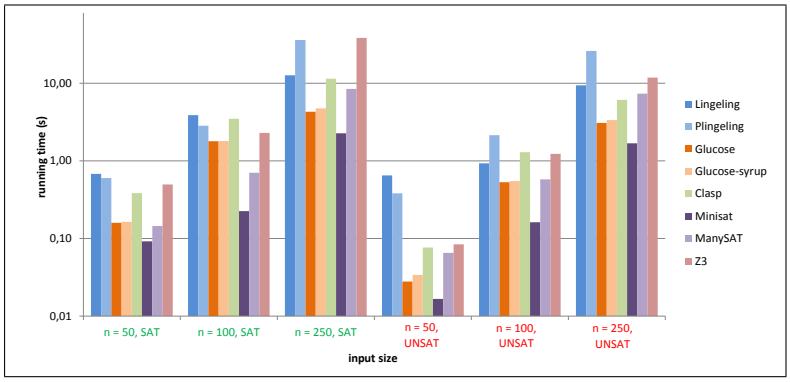

Figure 5. Average results for the Hamiltonian path problem, grouped by input size

5.1. Results for NP-complete problems

Because the three discussed classical NP-complete problems come from graph theory, creating input DIMACS files for SAT-solvers involves the random generation of graphs with a given number of vertices n and, where applicable, an additional parameter k. The input strings for the bounded Post correspondence problem and the String correction problem are also randomized. Thus, to ensure a fair comparison and reliable results, the running time was averaged over multiple files generated for each test instance.

The standard deviation varied significantly depending on the problem encoded and satisfiability of the instance, but one consistent observation is that the results were much more stable for the classical graph problems (vertex k-coloring and Hamiltonian path in particular), as opposed to bounded PCP and string correction. For the latter two, especially in unsatisfiable instances, relative standard deviation often exceeded 100%.

Older SAT-solvers generally performed in line with expectations. Of particular note here is Minisat's excellent performance when dealing with classical graph problems. On the other hand, zChaff was unable to handle most of the larger input files, often running out of memory even though a 64-bit version of the solver is available and was the one tested. Consequently, it is usually omitted from our results for better chart readability.

The situation is not nearly as clear-cut at the opposite end of the spectrum. Modern SAT-solvers, like Lingeling, Glucose and Clasp, though often offered far superior performance, did not come out on top in every single test instance. This clearly shows just how challenging the solving of an NP-complete problem such as SAT can be, while at the same time emphasizing the continuing need for further improvements in many areas of the algorithms.

More specifically, of NP-complete problems, Hamiltonian path (Fig. 5) and satisfiable instances of graph k-colouring (Fig. 3) appeared to be the least challenging for tested SAT-solvers. Even for the largest of generated input files, representing formulas with about 15 million clauses, finding a solution did not prove to be very difficult, and all solvers (with the exception of the aforementioned zChaff) were able to do so in 30 seconds or less. Lingeling, and especially its parallel version Plingeling, actually recorded the slowest times in both of these tests, which might be seen as surprising for a very recent, modern, state-of-the-art SAT-solver. It appears that in this particular case the time overhead introduced by additional operations such as file preprocessing for more efficient solving, and especially dividing the task between multiple threads in the case of Plingeling, simply exceeded the actual solving time. While less than ideal in this specific situation, it is in no way indicative of poor performance of the more sophisticated SAT-solvers in general, as evidenced by their behaviour in the difficult unsatisfiable instance of graph k-colouring, as well as in the following test.

The vertex k-cover test (Fig. 4) proved to be more difficult, scaling worse with an increased input size compared to the other two classical graph problems (hence the relatively small values of n and k chosen). In these circumstances, Lingeling was able to leverage its CDCL advancements to achieve significantly faster processing time than other SAT-solvers for unsatisfiable input. Overall, the test clearly showed that the more difficult the instance, the more likely it is for modern SAT-solvers to outperform older, less advanced algorithms. This was evidenced with not only Lingeling, but also with Clasp being significantly faster than the older Minisat. Glucose's results were on par with the latter, however, possibly stemming from the fact it was originally based on Minisat's source code. The same held true for ManySAT, while Z3's results were below average in this test.

Generally, it was expected for the satisfiable instances of the problems discussed to be solved faster, on account of not having to explore the remainder of the solution space once a satisfying assignment was found. While this was indeed the case for both graph k-colouring and vertex k-cover, an anomaly could be observed in the case of the Hamiltonian path test, where unsatisfiable instances were actually verified faster. This could be attributed to the relative ease with which the non-existence of a Hamiltonian path can be proved in a graph, requiring only a single unconnected vertex.

The results for the bounded Post correspondence problem, separated between satisfiable and unsatisfiable instances, are presented in Fig. 6. Glucose was the fastest solver in both cases (except for the largest UNSAT test where it finished marginally behind Lingeling), although its parallel version offered little to no performance increase, or even introduced slight time overhead. Plingeling, meanwhile, performed significantly better than its sequential counterpart, and this difference only became more evident as the size of test instances increased. Equally noteworthy is the fact Lingeling (along with its parallel branch) was the only SAT-solver in this test whose running time was shorter for unsatisfiable instances; otherwise, as expected, they took longer to verify. This difference, however, was not as dramatic as in the case of several other tests. Minisat, its parallelization ManySAT, and zChaff were largely unable to compete against state-of-the-art SAT-solvers in this test, requiring times at least an order of magnitude longer to complete verification. The results of Clasp and Z3 were also significantly worse than those achieved by Glucose and Plingeling.

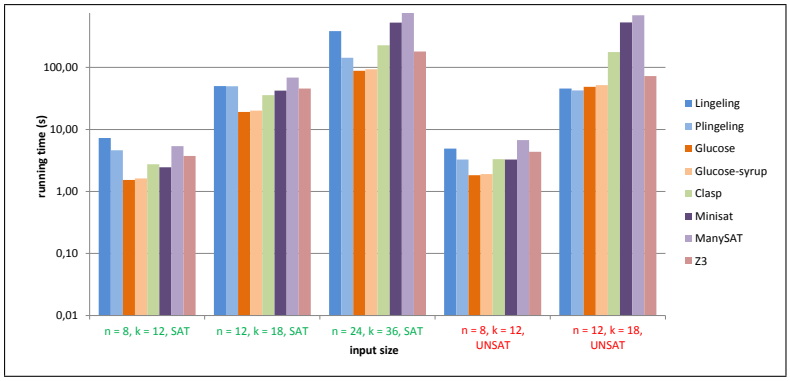

Figure 6. Results for bounded Post correspondence problem, grouped by input size

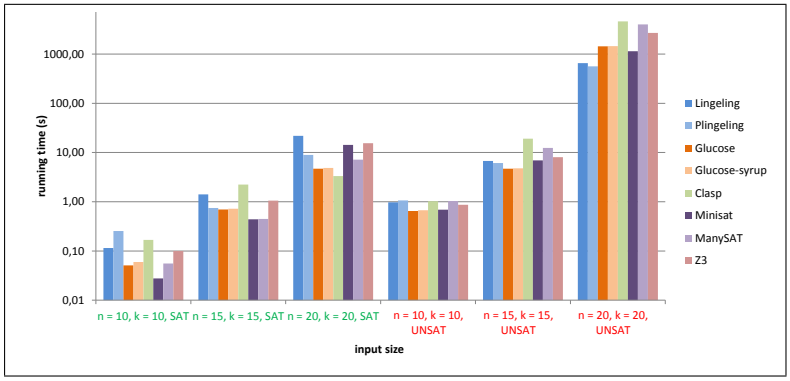

Figure 7. Results for the string correction problem, grouped by input size

Finally, in the string correction test a particularly large difference in verification time of satisfiable and unsatisfiable instances could be observed (Fig. 7), with the latter at least an order of magnitude longer. On the other hand, individual solvers were relatively closely matched compared to previous tests, although older ones like Minisat and ManySAT, and even the modern Clasp, noticeably lagged behind when processing the largest input files. Overall, Lingeling and Glucose verified the most difficult instances in shortest time, with the former slightly more effective at satisfiable instances.

5.2. Results for chess problems

The N-queens problem has several known solutions exploiting certain patterns and symmetries that were discovered to repeatedly occur in valid arrangements [4]. As such, it was expected that a reduction to SAT and employing a SAT-solver would not be optimal (Fig. 8). That is indeed the case: while there always remains a possibility that the solver will be able to encounter a valid assignment relatively early, even in the most optimistic scenario the processing time is several orders of magnitude worse than that of algorithms tailored for the N-queens problem. For instance, taking advantage of the aforementioned patterns and symmetries, the well known constraint solver OptaPlanner [13] is capable of finding solutions for as many as 100000 queens in a matter of seconds.

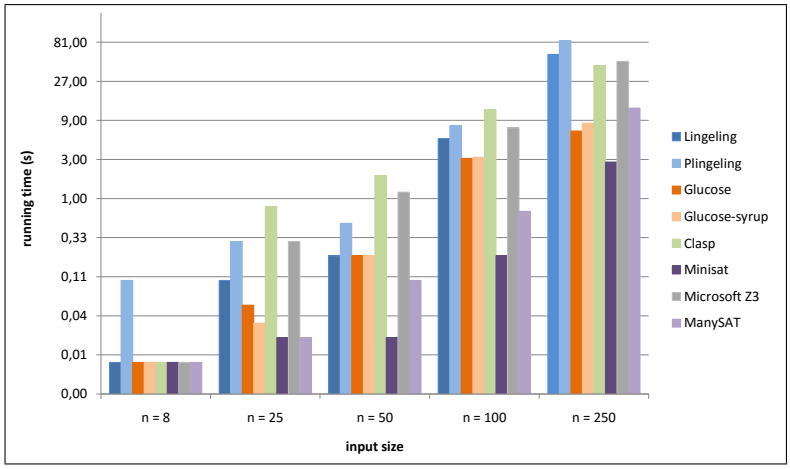

Figure 8. Results for the N-queens problem, grouped by input size

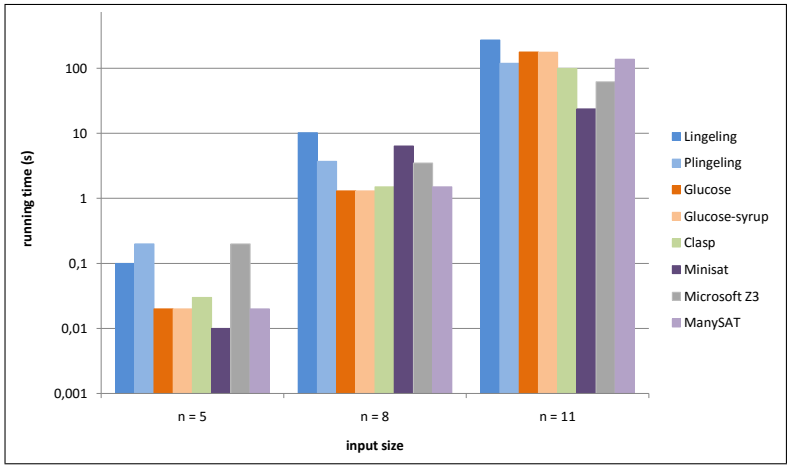

Figure 9. Results for the knight's tour problem, grouped by input size

The non-viability of using reductions to SAT as a means of efficiently solving chess problems that are not specifically NP-complete or NP-hard is even more clear in the case of knight's tour (Fig. 9). Since the chessboard has n^2 squares, the number of variables required to represent a unique knight's move drastically increases compared to that of a regular instance of the Hamiltonian path problem. Consequently, so does the rate at which input files increase in size. As early as at $n = 17$, or only slightly more than twice the size of a standard chessboard, none of the tested SAT-solvers was able to verify satisfiability within 60 minutes.

Practically, this all but disqualifies any use of SAT-solvers for this specific problem. Instead, attention should be directed towards specialized algorithms for the knight's tour problem that are known to have lower computational complexity [33, 26].

5.3. Results for EXPTIME and PSPACE problems

In Fig. 10 we present the comparison of SAT-solvers' performance using the Towers of Hanoi problem for $n \in \{5, 7, 9, 11, 13\}$. Because the puzzle has a solution for any number of discs n, there are, of course, no unsatisfiable instances to be tested.

As expected, the running time scales very poorly given that the problem is exponential in the size of the input. Lingeling and Glucose were by far the most efficient solvers in this competition, with the former slightly faster for the largest of tested instances. However, it should also be noted that in either case, parallel versions provided no noticeable advantage.

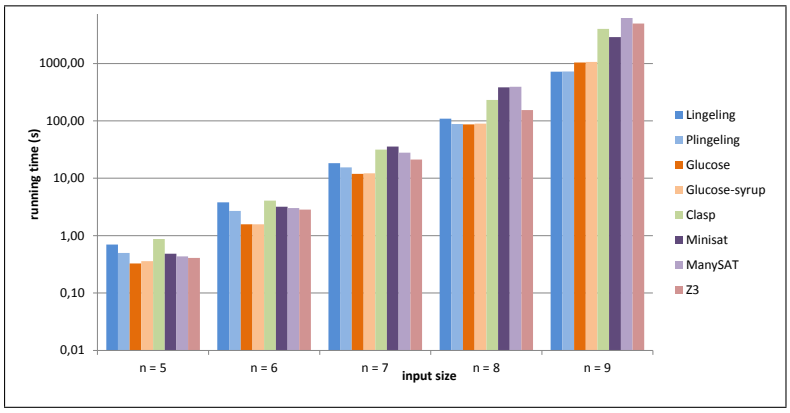

Figure 10. Results for the Towers of Hanoi problem

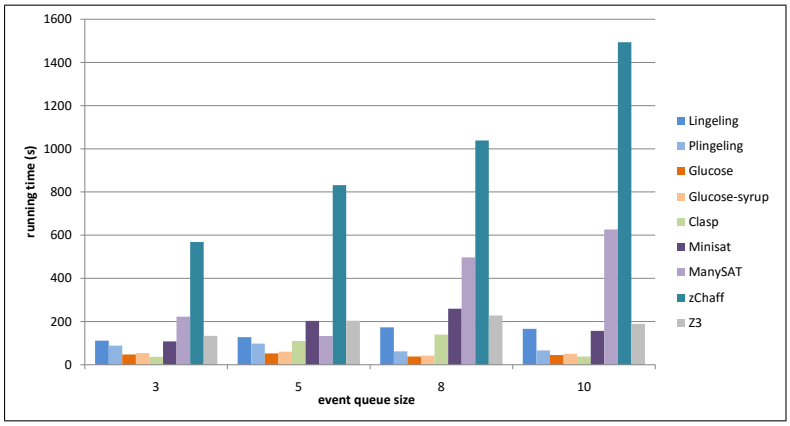

Figure 11. Results for model checking of UML systems

Finally, in Fig. 11 we present the results of GRC benchmark for 20 tracks, scaled w.r.t. the length of the event queues (3, 5, 8, and 10), where the obtained formula is satisfiable (at depth 18) and the integers are encoded over 10 propositional variables. It is easy to observe that modern solvers, especially Glucose and Clasp, handle the benchmarks almost effortlessly, taking advantage of the system symmetry. For older SAT-solvers, running time does increase with the size of the input, but nowhere near as significantly as in previous examples. This is also evidenced by the fact even zChaff was able to handle all instances

	Lingeling	Plingeling	Glucose	Gl.-syrup	Clasp	Minisat	ManySAT	Z3	zChaff
Sat	2277	1663	**1655**	1716	5061	4684	9449	6357	61848
Unsat	782	**710**	2261	2269	5484	2585	7806	3779	36000
Total	3059	**2373**	3916	3986	10545	7269	17255	10136	97848

Table 2. Total time consumed by the solvers for satisfiable and unsatisfiable benchmarks

in this test, albeit, as expected, it was by far the slowest solver in contention, once again emphasizing the dramatic progress made in the area since the beginning of the twenty-first century.

6. Conclusions

We have presented nine SAT-solvers and compared their efficiency for several decision and combinatorial problems: five NP-complete problems including three classical graph problems and two others featuring our original SAT encodings, bounded Post correspondence problem and string correction problem, two popular chess problems, verification of UML systems in PSPACE, and the Towers of Hanoi (ToH) problem, whose solutions are of exponential complexity. Our experimental results allow for drawing conclusions on efficiency and applicability of modern SAT-solvers to problems of different complexity. As one could expect, while SAT-solvers behave quite efficiently for NP-complete and harder problems, they are by far inferior to tailored algorithms for specific problems of lower complexity, like N-queens. Another main observation is that modern SAT-solvers, like Lingeling, Glucose and Clasp, despite their sophistication and often indeed superior performance, especially when dealing with the largest and most difficult (unsatisfiable) test instances, did not come out on top in every single test. However, their efficiency is unquestionable, if we take into account the overall results. Table 2 presents the total time needed by each tool to solve all presented benchmarks. The values in bold indicate the shortest total times consumed for satisfiable and unsatisfiable instances, and the best total time. As for the satisfiable benchmarks, the winner is Clasp with only a slight advantage over Plingeling, which, in turn consumed the least time for the unsatisfiable instances and in total. Thus, Plingeling is the champion of our comparison.

Overall, the results clearly show how challenging the solving of the SAT problem can be, while at the same time emphasizing the continuing need for further improvements in many areas of the algorithms.

References

[1] P. A. Abdulla, P. Bjesse, and N. Eén. Symbolic reachability analysis based on SAT-solvers. In *Proc. of the 6th Int. Conf. on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'00)*, volume 1785 of *LNCS*, pages 411–425. Springer-Verlag, 2000.

[2] A. Armando and L. Compagna. An optimized intruder model for SAT-based model-checking of security protocols. In *Electronic Notes in Theoretical Computer Science*, volume 125, pages 91–108. Elsevier Science Publishers, March 2005.

[3] G. Audemard and L. Simon. Glucose and syrup in the SAT race 2015. *SAT Race*, 2015.

[4] J. Bell and B. Stevens. A survey of known results and research areas for n-queens. *Discrete Mathematics*, 309(1):1 – 31, 2009.

[5] A. Biere. Lingeling and friends entering the SAT challenge 2012. *Proceedings of SAT Challenge*, pages 33–34, 2012.

[6] A. Biere et al. Symbolic model checking using SAT procedures instead of BDDs. In *In Proc. of the ACM/IEEE Design Automation Conference (DAC)*, pages 317–320, 1999.

[7] E. Clarke, A. Biere, R. Raimi, and Y. Zhu. Bounded model checking using satisfiability solving. *Formal Methods in System Design*, 19(1):7–34, 2001.

[8] F. J. Damerau. A technique for computer detection and correction of spelling errors. *Commun. ACM*, 7(3):171—-176, 1964.

[9] M. Davis, G. Logemann, and D. Loveland. A machine program for theorem-proving. *Commun. ACM*, 5(7):394–397, July 1962.

[10] M. Davis and H. Putnam. A computing procedure for quantification theory. *J. ACM*, 7(3):201–215, July 1960.

[11] L. de Moura and N. Bjørner. *Bugs, Moles and Skeletons: Symbolic Reasoning for Software Development*, pages 400–411. Springer Berlin Heidelberg, Berlin, Heidelberg, 2010.

[12] L. M. de Moura and N. Bjørner. Z3: An efficient SMT solver. In *Proc. of TACAS'08*, volume 4963 of *LNCS*, pages 337–340. Springer-Verlag, 2008.

[13] G. De Smet et al. Optaplanner user guide. *http://www. optaplanner. org.*, 2017.

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

[15] M. Gebser et al. clasp : A conflict-driven answer set solver. In *LPNMR*, 2007.

[16] I. P. Gent, Ch. Jefferson, and P. Nightingale. Complexity of n-queens completion. *J. Artif. Int. Res.*, 59(1):815–848, May 2017.

[17] Y. Hamadi, S. Jabbour, and L. Sais. ManySAT: a Parallel SAT Solver. *JSAT*, 6(4):245–262, 2009.

[18] M. Kacprzak and W. Penczek. Unbounded model checking for alternating-time temporal logic. In *3rd International Joint Conference on Autonomous Agents and Multiagent Systems (AAMAS 2004), 19-23 August 2004, New York, NY, USA*, pages 646–653, 2004.

[19] R. Kaivola et al. Replacing Testing with Formal Verification in Intel Core I7 Processor Execution Engine Validation. In *Proceedings of the 21st International Conference on Computer Aided Verification*, CAV '09, pages 414–429, Berlin, Heidelberg, 2009. Springer-Verlag.

[20] Lila Kari, Greg Gloor, and Sheng Yu. Using DNA to solve the bounded post correspondence problem. *Theor. Comput. Sci.*, 231(2):193–203, 2000.

[21] R. Karp. Reducibility among combinatorial problems. In *Complexity of Computer Computations*, pages 85–103. Plenum Press, 1972.

[22] H. Kautz and B. Selman. Planning as satisfiability. In *In ECAI 92: Proceedings of the 10th European conference on Artificial intelligence*, pages 359–363, 1992.

[23] M. Knapik and W. Penczek. Bounded model checking for parametric timed automata. *Trans. Petri Nets and Other Models of Concurrency*, 5:141–159, 2012.

[24] D. Le Berre and P. Rapicault. Dependency management for the eclipse ecosystem: Eclipse p2, metadata and resolution. In *Proceedings of the 1st International Workshop on Open Component Ecosystems*, IWOCE '09, pages 21–30, New York, NY, USA, 2009. ACM.

[25] V. I. Levenshtein. Binary codes capable of correcting deletions, insertions, and reversals. *Soviet Physics Doklady*, 10(8):707—-710, 1966.

[26] Shun-Shii Lin and Chung-Liang Wei. Optimal algorithms for constructing knight's tours on arbitrary n x m chessboards. *Discrete Applied Mathematics*, 146(3):219 – 232, 2005.

[27] Y. Mahajan, Z. Fu, and S. Malik. Zchaff2004: An efficient sat solver. In *Int. Conf. on Theory and Applications of Satisfiability Testing*, pages 360–375. Springer, 2004.

[28] R. Martins and I. Lynce. Effective cnf encodings for the towers of hanoi. In *International Conference on Logic for Programming Artificial Intelligence and Reasoning*, 2008.

[29] A. Męski, W. Penczek, M. Szreter, B. Woźna-Szcześniak, and A. Zbrzezny. BDD-versus SAT-based bounded model checking for the existential fragment of linear temporal logic with knowledge: algorithms and their performance. *Autonomous Agents and Multi-Agent Systems*, 28(4):558–604, 2014.

[30] A. Niewiadomski. *Automatyczna weryfikacja systemow specyfikowanych w UML (in Polish)*. PhD thesis, Polish Academy of Science, ICS, January 2011.

[31] A. Niewiadomski et al. Towards automatic composition of web services: SAT-based concretisation of abstract scenarios. *Fundam. Inform.*, 120(2):181–203, 2012.

[32] A. Niewiadomski, W. Penczek, and M. Szreter. A new approach to model checking of UML state machines. *Fundamenta Informaticae*, 93(1-3):289–303, 2009.

[33] I. Parberry. An efficient algorithm for the knight's tour problem. *Discrete Appl. Math.*, 73(3):251–260, 1997.

[34] Emil L Post. A variant of a recursively unsolvable problem. *Bulletin of the American Mathematical Society*, 52(4):264–268, 1946.

[35] B. Selman, H. Kautz, and B. Cohen. Local search strategies for satisfiability testing. In *Dimacs Series in Discrete Mathematics and Theoretical Computer Science*, pages 521–532, 1995.

[36] B. Selman, H. Levesque, and D. Mitchell. A new method for solving hard satisfiability problems. In *Proceedings of the Tenth National Conference on Artificial Intelligence*, AAAI'92, pages 440–446. AAAI Press, 1992.

[37] N. Sorensson and N. Een. Minisat v1. 13 - a SAT solver with conflict-clause minimization. *SAT*, 2005(53):1–2, 2005.

[38] H. Stamm-Wilbrandt. Programming in propositional logic or reductions: Back to the roots (satisfiability), 1993.

[39] OMG UML. Unified modeling language. *Infrastructure Specification,*, 2(1), 2007.

[40] R. A. Wagner and M. J. Fischer. The string-to-string correction problem. *Journal of the ACM*, 21(1):168––173, 1974.

[41] B. Woźna, A. Zbrzezny, and W. Penczek. Checking reachability properties for Timed Automata via SAT. *Fundamenta Informaticae*, 55(2):223–241, 2003.

[42] B. Woźna-Szcześniak. SAT-based bounded model checking for weighted deontic interpreted systems. *Fundam. Inform.*, 143(1-2):173–205, 2016.