Incremental SAT-Based Method with Native Boolean Cardinality Handling for the Hamiltonian Cycle Problem

Takehide Soh, Daniel Le Berre, Stéphanie Roussel, Mutsunori Banbara, Naoyuki Tamura

Incremental SAT-based Method with Native Boolean Cardinality Handling for the Hamiltonian Cycle Problem

Takehide Soh[1], Daniel Le Berre[2], Stéphanie Roussel[2], Mutsunori Banbara[1], and Naoyuki Tamura[1]

[1] Kobe University 1-1, Rokko-dai, Nada, Kobe, Hyogo 657-8501 Japan
{soh@lion.,tamura@,banbara@}kobe-u.ac.jp
[2] CNRS - Université d'Artois, Rue Jean Souvraz, SP-18, F-62307, Lens, France
{leberre,sroussel}@cril.univ-artois.fr

Abstract. The Hamiltonian cycle problem (HCP) is the problem of finding a spanning cycle in a given graph. HCP is NP-complete and has been known as an important problem due to its close relationship to the travelling salesman problem (TSP), which can be seen as an optimization variant of finding a minimum cost cycle. In a different viewpoint, HCP is a special case of TSP. In this paper, we propose an incremental SAT-based method for solving HCP. The number of clauses needed for a CNF encoding of HCP often prevents SAT-based methods from being scalable. Our method reduces that number of clauses by relaxing some constraints and by handling specifically cardinality constraints. Our approach has been implemented on top of the SAT solver Sat4j using Scarab. An experimental evaluation is carried out on several benchmark sets and compares our incremental SAT-based method against an existing eager SAT-based method and specialized methods for HCP.

1 Introduction

The Hamiltonian cycle problem (HCP) is the problem of finding a spanning cycle, called *Hamiltonian cycle*, in a given graph. HCP is listed in Karp's 21 NP-complete problems [24] and has been known as an important problem due to its close relationship to the travelling salesman problem (TSP). On the one hand, HCP is a special case of TSP. On the other hand, TSP can be seen as an optimization variant of HCP and the development of an effective method for TSP would have a significant impact in computer science.

HCP has been theoretically studied in graph theory [16,17]. Besides, HCP is tackled in Operations Research (OR). For instance, Jäger and Zhang [21] shows a method based on the Hungarian algorithm and Karp-Steele patching for solving HCP on directed graphs. More recently, Eshragh *et. al.* shows a hybrid algorithm and a Mixed Integer Programming (MIP) model for HCP on undirected graphs [12].

HCP also has been studied in Artificial Intelligence using propositional satisfiability (SAT). In SAT-based methods, the main issue for solving HCP is how

to encode connectivity constraints. Those constraints can also be seen as permutation constraints which have been studied in Constraint Programming [18]. An encoding method was proposed in 90's by [20,19] named later *absolute encoding*. Following that, in 2003, Prestwich proposed the *relative encoding* [28] which requires fewer clauses than the absolute encoding. In 2009, Velev and Gao further improve the relative encoding by merging encoding variables and applying triangulation to a given graph [32] which achieved indeed 4 orders of magnitude speedup on satisfiable structured graphs from the DIMACS graph coloring instances compared to the one by Prestwich [28]. However, the number of clauses in the encoding is increasing by $O(n^3)$ and it is still difficult to solve graph instances which consist in over 1,000 nodes.

In this paper, we escape the current limitations of SAT-based methods using an abstraction/refinement approach and by natively handling Boolean cardinality constraints. Note that we consider in our encoding for undirected graphs as in [28,32].

- **Incremental HCP Solving.** The encoding of the connectivity constraints often causes the generation of a huge amount of clauses which prevent SAT-based methods from being scalable. Our method thus relaxes the connectivity constraints to reduce the number of clauses and incrementally refines the encoding by adding new clauses when sub-cycles are detected.
- **Native Boolean Cardinality Handling.** Another issue when translating HCP to SAT is to express Boolean Cardinality (BC) constraints, for which various encoding into CNF exists. In addition to using those existing BC encodings, we propose to use a solver with native support for BC constraints, called *Native BC*. The Native BC has the advantages to reduce encoding time and memory usage. Native BC is provided as a specific constraint in the SAT solver Sat4j [27].
- **Implementation on a System Tightly Integrated with SAT Solvers.** Since SAT solvers are necessary to invoke many times in incremental HCP solving, communication cost is not negligible. We thus implement the first version of our method on Scarab [31] which is tightly integrated with Sat4j.

We carried out experiments on three benchmark sets. One is `color04` which is used in [32] and comes from DIMACS graph coloring instances [1]. The second one is `knight` which is a set of knight's tour instances used in [12]. The third one is `tsplib` which is the whole set of HCP instances in TSPLIB [4]. On those benchmark sets, we compare the proposed incremental SAT-based methods against the previous eager SAT-based method by Velev and Gao [32], a HCP solving method by Eshragh et al. [12], and the state-of-the-art TSP solver LKH. The latter provided the best answers for instances with unknown optima from DIMACS TSP Challenge [2] and provides an interface for HCP. In our experiments, we used the latest version 2.0.7 of LKH, whose performance on HCP is improved from previous versions [3]. All benchmark, programs, experimental results explained in this paper are available in: `http://kix.istc.kobe-u.ac.jp/~soh/scarab/jelia2014/`

2 Hamiltonian Cycle Problems

The Hamiltonian cycle problem (HCP) is the problem of finding a spanning cycle in a given graph. Let $G = (V, E)$ be a graph where V is a set of n nodes and E is a set of edges. A set of auxiliary arcs $A = \{(i, j),\ (j, i) \mid \{i, j\} \in E\}$ is also introduced for simple modeling. Let $x_{ij} (i \neq j)$ be a Boolean variable for each arc $(i, j) \in A$, which is equal to 1 when (i, j) is used in a solution cycle. Then, a direct modeling of HCPs would be using the following constraints.

$$\sum_{(i,j)\in A} x_{ij} = 1 \qquad \text{for each node } i = 1, \ldots, n. \qquad \text{(out-degree)}$$

$$\sum_{(i,j)\in A} x_{ij} = 1 \qquad \text{for each node } j = 1, \ldots, n. \qquad \text{(in-degree)}$$

$$\sum_{i,j\in S} x_{ij} \leq |S| - 1, \qquad S \subset V,\ 2 \leq |S| \leq n - 2 \qquad \text{(connectivity)}$$

The *out-degree* and *in-degree* constraints force that, for each node, in-degree and out-degree are respectively exactly one in a solution cycle. The *connectivity* constraint prohibits the formation of sub-cycles, i.e., cycles on proper subsets of n nodes. HCPs have been tackled by SAT-based methods. In [28], transitive relations for all possible permutations of three nodes are used to represent the connectivity constraint, which however results in $O(n^3)$ clauses. Velev and Gao follow this encoding, i.e., it basically needs $O(n^3)$ clauses, but they practically reduce the number of clauses by a triangulation for a given graph [32]. Besides, they also improve encoding by merging ordering variables. As a result, their SAT-based method achieves 4 orders of magnitude speedup on satisfiable structured graphs from the DIMACS graph coloring instances. However, it struggles to find a Hamiltonian cycle when the graph has over 1,000 nodes.

3 Proposal

3.1 Incremental HCP Solving

Previous SAT-based methods encode all constraints of HCP into SAT and compute its solution using a single execution of the SAT solver: we call those methods "eager". The main drawback of those eager methods for HCP is the encoding of connectivity constraints which results basically in $O(n^3)$ clauses.

To solve large HCPs, instead of encoding connectivity constraints into CNF and run a SAT solver once, we relax those constraints and incrementally execute the SAT solver on an abstraction of the problem. If the solution found contains sub-cycles, we prevent them in the new abstraction by adding new clauses. As such, we generate the clauses encoding the connectivity constraints "on demand", or "lazily". Such approach correspond to a *Counterexample-Guided Abstraction Refinement* (CEGAR) loop for HCP which was originally proposed in the context of model checking [9] and depicted in Fig. 1.

```
1: Ψ := initial abstraction of G ;
2: while (Ψ is satisfiable)
3:     if (Solution contains only one cycle)
           // we found a Hamiltonian cycle of G
4:         return Solution
5:     Ψ_block := Construct blocking clauses;
                  // (two for each sub-cycle)
6:     Ψ := Ψ ∧ Ψ_block ;
7: return there is no Hamiltonian cycle;
```

Fig. 1: CEGAR Iteration for Solving HCP

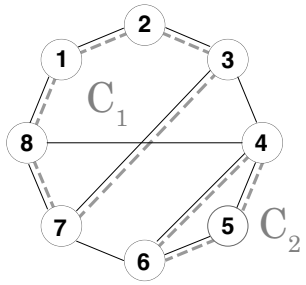

Fig. 2: Counter Example

The initial abstraction is built by omitting the connectivity constraint. That is, cardinality constraints corresponding to in/out-degree constraints are encoded. We also encode $x_{ij}+x_{ji} \leq 1$ for each edge $\{i,j\} \in E$ to prevent sub-cycles between two nodes. Those encoding results in a CNF formula Ψ (Line 1), which represents an abstract HCP constraint model. It ensures that every node must belong to some cycle but it does not ensure that the cycle is a Hamiltonian cycle. Fig. 2 shows such a case: every node belongs to a cycle but there is more than one cycle. SAT solving is then executed and the CEGAR iteration starts (Line 2). Whenever the formula is unsatisfiable, the iteration ends and it is decided that there is no Hamiltonian cycle (Line 8). If the formula is satisfiable and its model contains a single cycle then it must be a Hamiltonian cycle (Line 4). Otherwise, the solution consists of multiple sub-cycles which represent counter examples. To refine the constraints, some blocking clauses are added to Ψ to block each sub-cycle clockwise and counterclockwise (Line 6 and 7). This procedure is iterated until a Hamiltonian cycle is found or Ψ becomes unsatisfiable. Blocking clauses are generated to prevent the sub-cycles to appear again. In the case of Fig. 2, the following four clauses are generated: $\neg x_{12} \vee \neg x_{23} \vee \neg x_{37} \vee \neg x_{78} \vee \neg x_{81}$ to block C_1 clockwise. $\neg x_{87} \vee \neg x_{73} \vee \neg x_{32} \vee \neg x_{21} \vee \neg x_{18}$ to block C_1 counter-clockwise. $\neg x_{46} \vee \neg x_{65} \vee \neg x_{54}$ to block C_2 clockwise. $\neg x_{45} \vee \neg x_{56} \vee \neg x_{64}$ to block C_2 counter-clockwise. Note that, even in the worst case, we do not always need to block all sub-cycles in a given graph since in/out-degree constraints ensure that every node belongs to some cycle. For instance, in Fig. 2, it is not necessary to block a sub-cycle $(1,2,3,4,8)$ since the remaining nodes $\{5,6,7\}$ cannot construct any sub-cycles.

3.2 Native Boolean Cardinality Handling

By the relaxation of connectivity constraints, we may reduce considerably the number of clauses compared to eager SAT-based methods [19,28,32]. This section discusses how to encode the remaining in/out-degree constraints, which form Boolean cardinality (BC) constraints $\sum_{i=1}^{m} x_i \ \# \ k$ where $x_i \in \{0,1\}$ are

Boolean variables, m is an integer represents the number of variables, the relational operator $\#$ is one of $\{\leq, \geq, =\}$, k is an integer represents the degree (threshold) of the constraint.

Boolean cardinality encoding into CNF has been actively studied [30,6,29,13]. When we use *binomial encoding*, $\binom{m}{k}$ clauses are needed. It is improved by using *Totalizer* ($O(m^2)$) [6], or *Sequential Counter* ($O(m \cdot k)$) [30]. However, even when using the Sequential Counter for encoding the BC constraints in HCP, $O(n^2)$ clauses are needed for graph instances consisting of n nodes.

One way to avoid generating those clauses is to support natively a specific representation of those cardinality constraints in the SAT solver. It is expected that such specialized SAT-based systems could benefit from avoiding the time of CNF encoding, and reducing the number of constraints in the solver, which reduces the amount of memory used.

The Sat4j library [27] started in 2004 as an implementation in Java of the original Minisat specification [11]. In contrast with recent versions of Minisat, and most SAT solvers, the underlying SAT solver is still designed to work with custom constraints, not just clauses. Sat4j has a native representation of BC constraints, denoted *Native BC* in the rest of the paper. It currently emulates a BC constraint $\sum_{i=1}^{n} x_i \geq k$. This specific constraints generates clauses of size $n - k + 1$ when it detects a conflict with the current assignment. In addition, whenever it detects that $n - k$ variables are already assigned to 0, it forces the remaining variables to be 1 using the $n - k$ falsified literals as an explanation for those propagation. One can consider that such constraint generates "on demand" or lazily the clauses of the binomial encoding.

4 Experimental Results

This section provides experimental results to evaluate the effectiveness of the incremental HCP solving, Native BC, and their implementation on Scarab. We also have a comparison with other specialized methods. The following systems are used:

- Eager SAT-based method (referred to as Velev) is our implementation of the previous SAT-based method by Velev and Gao [32]. It runs with Minisat2.2.
- HCP/TSP Solver LKH is the state-of-the-art TSP solver which provided the best answers for instances with unknown optima from DIMACS TSP Challenge [2] and provides an interface for HCP. In our experiments, we used the latest version 2.0.7 of LKH, whose performance on HCP is improved from previous versions [3].
- Incremental HCP Solving (referred to as S4J-S, S4J-N) is the proposed methods implemented on Scarab. Two versions are prepared to measure the effectiveness of using Sequential Counter or Native BC, respectively. We have also tested another encoding method Totalizer in all instances but omit their results since they are similar (or slightly inferior) to Sequential Counter. Readers can check the results of Totalizer in the supplemental web page. Note that learned clauses are cleared after each iteration since keeping them

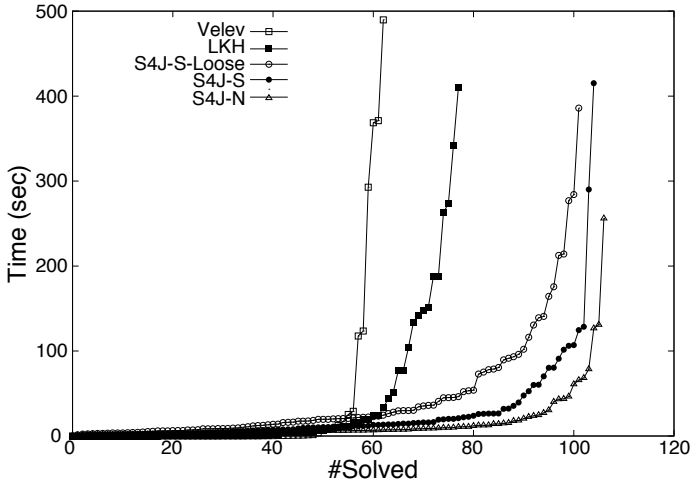

Fig. 3: Cactus Plot on `color04`, `knight`, and `tsplib`

across calls did not accelerate searches but other heuristic values are kept through all iterations.

– We also prepared, S4J-S-Loose, a variant of S4J-S, which is implemented on loosely integrated system to measure the implementation difference.

All experiments are carried out on Intel Xeon 2.93 GHz within the timelimit of 500 seconds. 4GB heap memory is allowed in the Java virtual machine settings (`-Xms4g -Xmx4g`). Sat4j with the prebuilt solver "Glucose21" is used for incremental HCP solving, which gave the best overall results from the available solvers of the library on the benchmarks used. Benchmark sets are selected from the literature of the previous eager SAT-based method [32] and a HCP solving method by Eshragh et al. [12]: `color04` comes from DIMACS graph coloring instances [1] used in the eager SAT-based method [32]. It consists of 119 instances whose number of nodes ranges from 11 to 10,000. `knight` is a set of knight's tour instances used in [12]. In the literature, only 3 instances of sizes 8x8, 12x12, 20x20 are used. In the experiments, we additionally use 8 instances of sizes 30x30, 40x40, ..., and 100x100 for wider comparisons. `tsplib` is the whole set of HCP instances of TSPLIB [4]. Similar to `knight`, two of them are used in [12] and we additionally use the remaining 7 instances for wider comparisons.

Fig. 3 shows a cactus plot denoting all results of compared systems: Velev, LKH, S4J-S, S4J-N, and S4J-S-Loose. In the result, the eager SAT-based method Velev solved 60 instances but slows down in early stage. A reason is the number of encoded clauses which explodes to over 100 million even when #nodes of the input graph is 500. It is obviously closed to the limit of SAT solvers. For instance, it generates 194,186,195 clauses for `DSJC500.5` (#nodes is 500) and could not encode `latin_square` (#nodes is 900) within 500 seconds. LKH solved

Table 1: #Solved per Graph-Size

Graph Size	#Ins.	Velev [32]		S4J-N	
		#S	(%)	#S	(%)
$n \leq 200$	54	45	(83)	48	(89)
$200 < n \leq 2000$	63	15	(23)	49	(78)
$2000 < n$	21	0	(0)	8	(38)

Table 2: Statistics

	S4J-S		S4J-N	
	#Ite.	#Cyc.	#Ite.	#Cyc.
Median	10	48	7	20
Average	60.0	311.8	37.9	310.5
Maximum	3332	9188	761	7604

81 instances, which is more than Velev but less than the incremental HCP solving methods: S4J-S-Loose, S4J-S, and S4J-N. Among them, the difference of S4J-S-Loose and S4J-S is not small: S4J-S is faster especially until 100 seconds. Consequently, incremental HCP solving with Native BC S4J-N solved the most instance – it is always faster than other methods. A reason is that incremental methods can start with much less clauses and practically do not need so many iterations and blocking clauses as is explained in the latter part of this section. We also have a literature-based comparison with results provided by Eshragh *et. al.* [12]. They carried experiments on knight's tour problems of 8x8, 12x12, and 20x20, and TSPLIB problems of `alb1000` and `alb2000`. Runtimes of S4J-N range from 1 second to 8 seconds while runtimes of their method range from 2 seconds to 165,600 seconds.

Table 1 shows the distribution of the number of solved instances for the number n of nodes on Velev and S4J-N. In case of $n \leq 200$, both Velev and S4J-N solved more than 80% of instances. However, in case of $200 < n \leq 2000$, Velev could solve only 23% of instances while S4J-N solves 78% of instances. Moreover, even the case of $2000 < n$, S4J-N still solves 38% of instances. With regard to the number of nodes of graphs, the largest satisfiable instance solved by Velev is `1-Insertions_6` ($n = 607$), one by S4J-N is `alb5000` ($n = 5,000$) [1]. In the literature [7,32], triangulation techniques are proposed to reduce the number of transitivity constraints and they are supposed to be effective for sparse graphs. If we select graph instances whose density are less than 0.03 and number of nodes are less than 2000 (33 out of 138), the difference between Velev and S4J-N becomes smaller but S4J-N still solves 25 instances while Velev solves 16 instances.

Table 2 shows the median, average, and maximum numbers of iterations and cycles for all satisfiable instances solved by each of S4J-S and S4J-N. We can read the followings from this table. The maximum number of cycles found for one instance is less than 10 thousands, that is, we need at most 20 thousands clauses in addition to the base clauses for solving those instances. Also, the median numbers of cycles show that we generally need much less additional clauses. The median numbers of iterations and cycles are almost stable in two encoding methods. In some cases, from the maximum numbers of iterations, we need to launch the SAT solver over thousands times. Considering that the given time limit is 500 seconds, the cost of the invocation of SAT solving procedure is preferred to be low in incremental HCP solving.

[1] S4J-N solved `qg.order100` ($n = 10,000$) with 512 seconds a bit longer than timelimit.

In addition to above experiments and analyses, readers can find further experiments and comparisons (e.g. using other SAT solvers) in:
`http://kix.istc.kobe-u.ac.jp/~soh/scarab/jelia2014/`.

5 Related Work

In 2000, Clarke *et al.* proposed Counterexample-Guided Abstraction Refinement (CEGAR) in the context of model checking [9], which receives a program text and abstract functions are extracted from it. Following their work, there are some applications of CEGAR to Presburger Arithmetic [25], deciding the theory of Arrays [14], and the RNA-folding problem [15]. Recently, the use of CEGAR was proposed to solve QBF [23], Circumscription [22] and argumentation inference [10]. We believe that such approach can be applied to even more cases in Artificial Intelligence.

In the context of solving TSP, there is a traditional OR technique proposed in 80's which translates TSP into the assignment problem [8,26]. Jäger and Zhang [21] apply this OR technique to HCP on directed graphs by using the Hungarian algorithm and Karp-Steele patching. Though only for a small proportion of instances, a SAT approach is used in their rare last step (14 out of 4266 instances) to guarantee completeness. It is described in the literature [21] that their method is less effective to undirected graphs, in particular, in the case that a given graph have no Hamiltonian cycle their method will enumerate all sub-cycles in the main step which cause a long running time. In our method, the SAT approach is central and part of a CEGAR loop, which practically performs well on undirected graphs for both SAT/UNSAT problems. Comprehensive experiments are carried by using several encoding/solvers. Our work provides some hints on the importance of (not) encoding cardinality constraints into CNF.

6 Conclusion

In this paper, we proposed an incremental SAT-based method with Native BC for solving HCP. It overcomes other methods by reducing the cost of full encoding of connectivity constraints and CNF encoding of BC constraints. Our work gives analyses for encoded clauses and iterations, and also points out that pre-processing affects the convergence of CEGAR iterations for solving HCP. Recently, Abío *et. al.* presented an approach which balance the use of encoding and the use of custom propagators within SMT [5]. In our work, a custom propagator is used for BC while a lazy encoding of the combination constraints is performed using CEGAR. It is another kind of balance between encoding and propagation.

Acknowledgements

This work was partially funded by JSPS KAKENHI Grant Numbers 24300007 and 25730042.

References

1. DIMACS Graph Coloring. `http://mat.gsia.cmu.edu/COLOR/instances.html`.
2. DIMACS TSP Challnege. `http://dimacs.rutgers.edu/Challenges/TSP/`.
3. LKH. `http://www.akira.ruc.dk/ keld/research/LKH/`.
4. TSPLIB. `http://comopt.ifi.uni-heidelberg.de/software/TSPLIB95/`.
5. Ignasi Abío, Robert Nieuwenhuis, Albert Oliveras, Enric Rodríguez-Carbonell, and Peter J. Stuckey. To encode or to propagate? the best choice for each constraint in SAT. In *CP*, pages 97–106, 2013.
6. Olivier Bailleux, Yacine Boufkhad, and Olivier Roussel. A translation of pseudo boolean constraints to SAT. *Journal on Satisfiability, Boolean Modeling and Computation*, 2(1-4):191–200, 2006.
7. Randal E. Bryant and Miroslav N. Velev. Boolean satisfiability with transitivity constraints. *ACM Trans. Comput. Log.*, 3(4):604–627, 2002.
8. Giorgio Carpeneto and Paolo Toth. Some new branching and bounding criteria for the asymmetric travelling salesman problem. *Management Science*, 26(7):pp. 736–743, 1980.
9. Edmund M. Clarke, Orna Grumberg, Somesh Jha, Yuan Lu, and Helmut Veith. Counterexample-guided abstraction refinement. In *CAV*, pages 154–169, 2000.
10. Wolfgang Dvorák, Matti Järvisalo, Johannes Peter Wallner, and Stefan Woltran. Complexity-sensitive decision procedures for abstract argumentation. *Artif. Intell.*, 206:53–78, 2014.
11. Niklas Eén and Niklas Sörensson. An extensible SAT-solver. In *Proceedings of the 6th International Conference on Theory and Applications of Satisfiability Testing (SAT 2003), LNCS 2919*, pages 502–518, 2003.
12. Ali Eshragh, Jerzy A. Filar, and Michael Haythorpe. A hybrid simulation-optimization algorithm for the Hamiltonian cycle problem. *Annals OR*, 189(1):103–125, 2011.
13. Alan M. Frisch and Paul A. Giannaros. SAT encodings of the at-most-k constraint: Some old, some new, some fast, some slow. In *Proceedings of the The 9th International Workshop on Constraint Modelling and Reformulation (ModRef 2010)*, 2010.
14. Vijay Ganesh and David L. Dill. A decision procedure for bit-vectors and arrays. In *CAV*, pages 519–531, 2007.
15. Vijay Ganesh, Charles W. O'Donnell, Mate Soos, Srinivas Devadas, Martin C. Rinard, and Armando Solar-Lezama. Lynx: A programmatic SAT solver for the rna-folding problem. In *SAT*, pages 143–156, 2012.
16. Ronald J. Gould. Advances on the Hamiltonian problem - a survey. *Graphs and Combinatorics*, 19(1):7–52, 2003.
17. Ronald J. Gould. Recent advances on the Hamiltonian problem: Survey III. *Graphs and Combinatorics*, 30(1):1–46, 2014.
18. Brahim Hnich, Toby Walsh, and Barbara M. Smith. Dual modelling of permutation and injection problems. *J. Artif. Intell. Res. (JAIR)*, 21:357–391, 2004.
19. Holger H. Hoos. SAT-encodings, search space structure, and local search performance. In *Proceedings of the 16th International Joint Conference on Artificial Intelligence (IJCAI 1999)*, pages 296–303, 1999.
20. Kazuo Iwama and Shuichi Miyazaki. SAT-variable complexity of hard combinatorial problems. In *Proceedings of the IFIP 13th World Computer Congress*, pages 253–258, 1994.

21. Gerold Jäger and Weixiong Zhang. An effective algorithm for and phase transitions of the directed Hamiltonian cycle problem. *J. Artif. Intell. Res. (JAIR)*, 39:663–687, 2010.
22. Mikolás Janota, Radu Grigore, and João Marques-Silva. Counterexample guided abstraction refinement algorithm for propositional circumscription. In *JELIA*, pages 195–207, 2010.
23. Mikolás Janota, William Klieber, João Marques-Silva, and Edmund M. Clarke. Solving qbf with counterexample guided refinement. In *SAT*, pages 114–128, 2012.
24. Richard M. Karp. Reducibility among combinatorial problems. In *Complexity of Computer Computations*, pages 85–103, 1972.
25. Daniel Kroening, Joël Ouaknine, Sanjit A. Seshia, and Ofer Strichman. Abstraction-based satisfiability solving of presburger arithmetic. In *CAV*, pages 308–320, 2004.
26. Gilbert Laporte. The traveling salesman problem: An overview of exact and approximate algorithms. *European Journal of Operational Research*, 59(2):231–247, June 1992.
27. Daniel Le Berre and Anne Parrain. The Sat4j library, release 2.2. *Journal on Satisfiability, Boolean Modeling and Computation*, 7:59–64, 2010. system description.
28. Steven David Prestwich. SAT problems with chains of dependent variables. *Discrete Applied Mathematics*, 130(2):329–350, 2003.
29. João P. Marques Silva and Inês Lynce. Towards robust CNF encodings of cardinality constraints. In Christian Bessiere, editor, *Proceedings of the 13th International Joint Conference on Principles and Practice of Constraint Programming (CP 2007)*, volume 4741 of *Lecture Notes in Computer Science*, pages 483–497. Springer, 2007.
30. Carsten Sinz. Towards an optimal CNF encoding of boolean cardinality constraints. In *Proceedings of the 11th International Joint Conference on Principles and Practice of Constraint Programming (CP 2005), LNCS 3709*, pages 827–831, 2005.
31. Takehide Soh, Naoyuki Tamura, and Mutsunori Banbara. Scarab: A rapid prototyping tool for SAT-based constraint programming systems. In *SAT*, pages 429–436, 2013.
32. Miroslav N. Velev and Ping Gao. Efficient SAT techniques for relative encoding of permutations with constraints. In *Australasian Conference on Artificial Intelligence*, pages 517–527, 2009.