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 [20] 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 [14,15]. Besides, HCP is tackled in Operations Research (OR). For instance, Jäger and Zhang [19] 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

*A preliminary version of this paper was submitted to a workshop Pragmatics of SAT 2014 with a non-archived option. This workshop has limited number of audience and fulfills the submission policy of JELIA.

hybrid algorithm and a Mixed Integer Programming (MIP) model for HCP on undirected graphs [10].

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 [16]. An encoding method was proposed in 90's by [18,17] named later *absolute encoding*. Following that, in 2003, Prestwich proposed the *relative encoding* [25] 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 [30] which achieved indeed 4 orders of magnitude speedup on satisfiable structured graphs from the DIMACS graph coloring instances compared to the one by Prestwich [25]. 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 while other specialized method comparatively easily solves instances with 10,000 nodes.

In this paper, we escape the current limitations of SAT-based methods using an abstraction/refinement approach and by handling specifically some constraints. Note that we consider in our encoding undirected graphs as in [25,30].

- **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 [23].
- **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 which is tightly integrated with Sat4j.

We carried out experiments on three benchmark sets. One is `color04` which is used in [30] and comes from DIMACS graph coloring instances [1]. The second one is `knight` which is a set of knight's tour instances used in [10]. The third one is `tsplib` which is the whole set of HCP instances in TSPLIB [4]. In addition to the above benchmark sets, we generated random graph instances and carried out experiments to evaluate the scalability of proposed methods. On those benchmark sets, we compare the proposed incremental SAT based methods against the previous eager SAT-based method by Velev and Gao [30], a HCP solving method by Eshragh et al. [10], 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 [25], 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 [30]. 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 while the state-of-the-art TSP solver LKH easily solves instances of HCP with over 10,000 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 in $O(n^3)$ clauses.

```
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

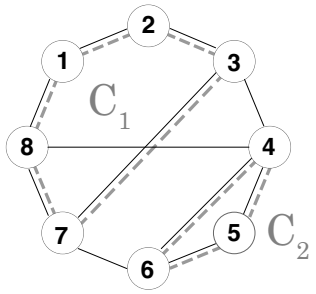

Fig. 2: Counter Example

To solve large HCP, 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 [8] and depicted in Fig. 1.

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.

$$\neg x_{12} \vee \neg x_{23} \vee \neg x_{37} \vee \neg x_{78} \vee \neg x_{81} \qquad \text{(block } C_1 \text{ clockwise)}$$
$$\neg x_{87} \vee \neg x_{73} \vee \neg x_{32} \vee \neg x_{21} \vee \neg x_{18} \qquad \text{(block } C_1 \text{ counter-clockwise)}$$
$$\neg x_{46} \vee \neg x_{65} \vee \neg x_{54} \qquad \text{(block } C_2 \text{ clockwise)}$$
$$\neg x_{45} \vee \neg x_{56} \vee \neg x_{64} \qquad \text{(block } C_2 \text{ counter-clockwise)}$$

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 [17,25,30]. 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 [27,6,26,11]. When we use *binomial encoding*, $\binom{m}{k}$ clauses are needed. It is improved by using *Totalizer* ($O(m^2)$) [6], or *Sequential Counter* ($O(m \cdot k)$) [27]. 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 [23] started in 2004 as an implementation in Java of the original Minisat specification [9]. 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.

3.3 System Tightly Integrated with SAT Solvers.

Thanks to the good performances of modern SAT solvers, a number of efficient SAT-based constraint solvers such as BEE [24] and Sugar [29] have been developed recently. Those systems are loosely integrated with the SAT solvers, i.e. they consider SAT solvers as black boxes. An advantage of this architecture is that we can upgrade regularly at no cost to the latest state-of-the art solvers.

There are however benefits to design tools with a more intimate communication with the SAT solver, among them to use the specificities offered by the

underlying solver such as Native BC in Sat4j. Scarab [28] is a constraint programming DSL in Scala tightly integrated with the SAT solver Sat4j. The whole system is thus almost OS agnostic since it runs on a Java virtual machine. The tight integration allows Scarab and Sat4j to communicate in memory without interprocess communication, which results in quick feedback from the solver.

Scarab allows us to write the procedure of incremental HCP solving almost as is — we need only about 100 lines of Scarab program (see detail in the supplemental page[1]).

4 Experimental Results

This section provides experimental results to evaluate the effectiveness of (a) the incremental HCP solving, (b) Native BC, and (c) their implementation on Scarab, which are respectively explained in Section 3.1, 3.2, and 3.3. In addition, we also have (d) a comparison with other specialized methods. To measure the above, 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 [30]. 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 across calls did not accelerate searches but other heuristic values are kept through all iterations.
- In addition to S4J-S and S4J-N on Scarab, we prepared an implementation of S4J-S on loosely integrated system to measure the implementation difference (referred to as S4J-S-Loose).

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. To select the benchmark sets, we adopted the ones used in the literature of the previous eager SAT-based method [30], and a HCP solving method by Eshragh et al. [10].

[1] `http://kix.istc.kobe-u.ac.jp/~soh/scarab/jelia2014/`

– `color04` comes from DIMACS graph coloring instances [1] used in the eager SAT-based method [30]. 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 [10]. 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 [10] and we additionally use the remaining 7 instances for wider comparisons.
– Also, in the latter part of this section, we use random graph instances to check the scalability of compared methods.

4.1 Comparisons on #Instances Solved and CPU Time

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 62 instances but slows down in early stage. A reason is the size of encoded clauses which explodes to over 100 million even when the input graph size is 500. It is obviously closed to the limit of SAT solvers. Moreover, the time of encoding also cannot be ignored. In fact, Velev cannot encode given instance within 500 seconds when the size of the graph becomes greater than 900. The number of encoded clauses of SAT-based methods is detailed in a later section. LKH solved 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.

Fig. 4 (left) shows the log-log scale scatter plot for the detailed comparisons of Velev v.s. S4J-S for commonly solved instances. Although S4J-S solved 42 more instances than Velev, Velev is faster than S4J-S for those limited instances. From this result, we can read that the eager SAT-based method Velev is fast if a given instance is of tractable size. Fig. 4 (right) shows the comparison of S4J-S v.s. S4J-N. S4J-N is constantly faster than S4J-S. In particular, the difference slightly tends to be large as the CPU time increases.

In addition to those experimental comparisons, we have a literature based comparison with a HCP solving method by Eshragh *et. al.* [10]. They propose a hybrid algorithm and a Mixed Integer Programming model for HCP. Since they do not provide the executable program of their method and the machine spec on which their program runs, direct comparison is difficult. However, for reference, we provide a literature-based comparison with their method using the same instances in [10]. Table 1 shows the comparison between S4J-N and [10]. Although direct comparison is difficult, there is obvious difference in computational performance between [10] and S4J-N. S4J-N solves those problems within 8 seconds while the runtimes of their method range from 2 seconds to 165600 seconds.

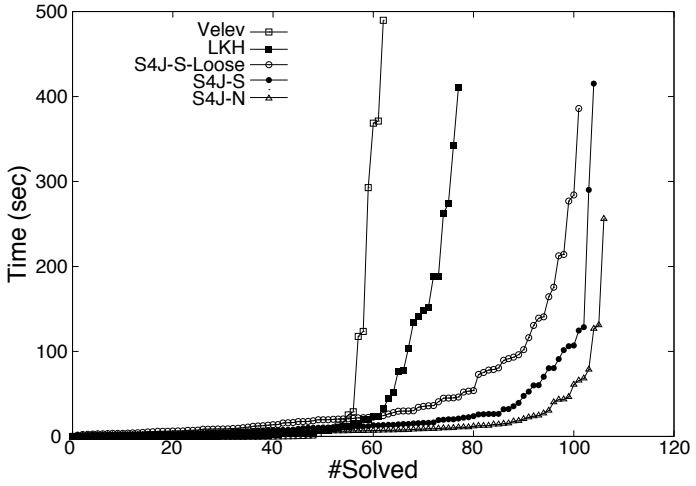

Fig. 3: Cactus Plot on `color04`, `knight`, and `tsplib`

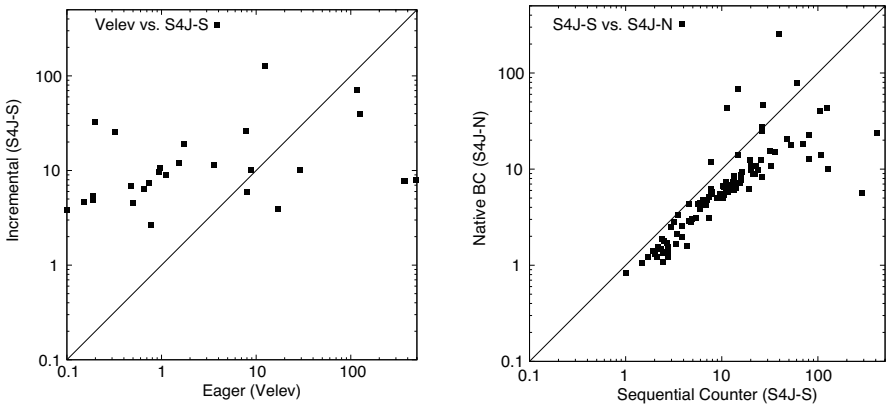

Fig. 4: CPU Time: (left) Velev vs. S4J-S (right) S4J-S vs. S4J-N

Table 1: Literature-based Comparison with [10]

Name	Nodes	Edges	CPU Time (sec.)	
			[Eshragh '11]	S4J-N
knight 8x8	64	168	2	1.2
knight 12x12	144	440	5	1.7
knight 20x20	400	1368	1731	4.3
alb1000	1000	1998	1907	5.8
alb2000	2000	3996	165600	7.2

4.2 Comparisons of CPU Time on Random Instances

Using random instances, we compared characteristic methods Velev, S4J-N, and LKH to check their scalability. Instances are generated at the phase transition region using parameters $m = cn(\ln n + \ln \ln n)/2$, where m is the number of edges, n is the number of nodes, c ranges 1.08 to 1.10. The size of those random graphs varies as follows: by 100 from 100 to 1000, by 500 from 1000 to 10000. Each family contains 10 graphs: totally 280 graphs are generated. Since unsatisfiable instances were relatively easy to solve for all approaches, we here use satisfiable 144 instances.

Fig. 5 shows average CPU time for each method on each graph size. If a method cannot solve some instance, then the time-limit (500 seconds) is plotted as its average cpu time. The eager method Velev runs out of memory on graphs with size greater than 300. On instances whose graph sizes are greater than 2000, S4J-N is better than others. However, S4J-N is inferior to LKH when graph size is less than 2000. One reason is the difference of implementation language: LKH is written in C++ while S4J-N is written in `Scala` which is executed on a Java virtual machine.

4.3 Comparisons of constraints encodings

In Section 4.1 and 4.2, we compare computational performance of all SAT-based and other specialized methods. This section provides how large SAT instances are encoded by each SAT-based method, which explains a reason why incremental HCP solving with Native BC is fast. Fig. 6 shows log-log scale scatter plots for the comparisons on the number of clauses in the CNF encoding for commonly solved instances. The plots for incremental methods include the number of blocking clauses added after the initial encoding. The left figure shows the comparison between Velev and S4J-S. Even we consider the additional blocking clauses, S4J-S encodes much smaller clauses. In particular, the difference of the number of encoded clauses increases as the graph size grows. A reason is that incremental methods practically do not need so many iterations and blocking clauses as is explained in the next section. The right one shows the comparison between S4J-S and S4J-N. S4J-N further reduces the number of encoded clauses. The difference comes from not-encoding in/out-degree constraint as is explained in Section 3.2.

Table 2 shows the initially encoded problem size, i.e., without blocking clauses, of three instances sampled from `color04`. The first column shows instance names and following two columns show the number of nodes and edges. The 4th to 9th columns show the number of variables, clauses, and cardinality constraints in the solver. '—' denotes that encoding cannot be done in 500 seconds. This table shows how the size of encoded problems explodes with the size of the graphs. Some instances even cannot be encoded within the time limit. Using Velev, the number of encoded clauses reaches almost 200 million. It is closed to the limit of modern SAT solvers. S4J-S reduces the number of clauses by 2 orders of magnitude. However, it cannot encode `qg.order100` within 500

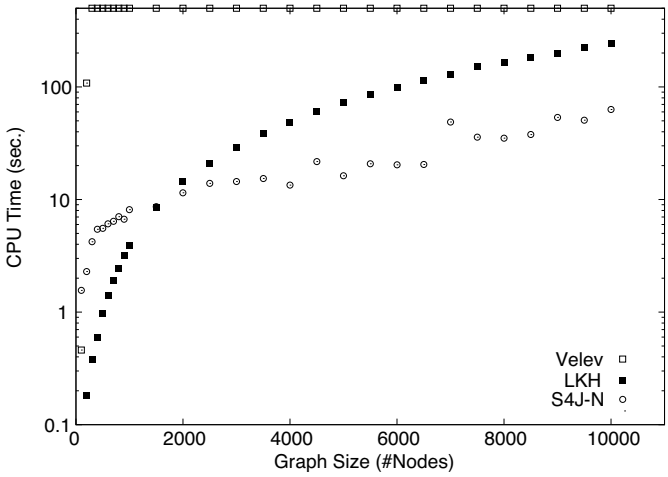

Fig. 5: Average CPU Time on Random Graph Instances

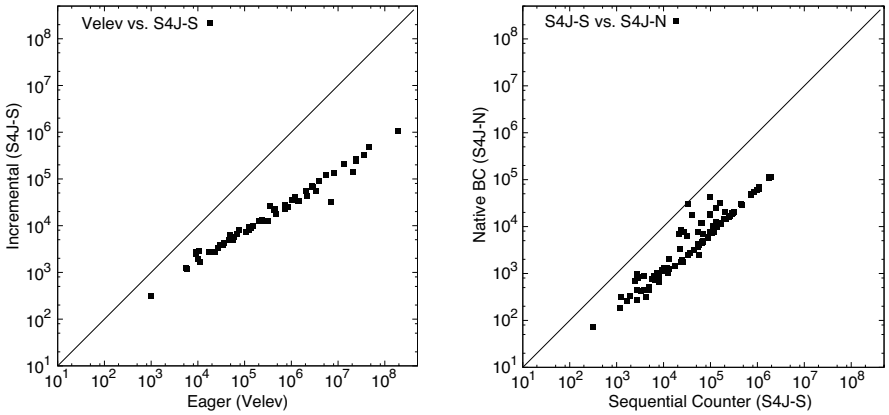

Fig. 6: #Clauses Encoded: (left) Velev vs. S4J-S (right) S4J-S vs. S4J-N

Table 2: Number of Encoded Variables, Clauses, Constraints

Instance	Nodes	Edges	Velev		S4J-S		S4J-N	
			Vars.	Cls.	Vars.	Cls.	Vars.	Cons.
`DSJC500.5`	500	62,624	248,690	194,186,195	375,744	1,063,608	125,248	64,624
`latin_square`	900	307350	—	—	1,844,100	5,223,150	614,700	310,950
`qg.order100`	10,000	990,000	—	—	—	—	1,980,000	1,030,000

Table 3: Statistics For All Solved and Satisfiable Instances

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

seconds. In contrast, S4J-N keeps the number of constraints to at most 1 million even when the numbers of nodes and clauses become 10 and 100 thousand.

4.4 Numbers of Iterations and Cycles to Reach Solutions

We report how many number of iterations and cycles are needed to reach a solution. Table 3 shows the median, average, and maximum numbers of iterations and cycles for all satisfiable instances solved by each of S4J-S and S4J-N. For those benchmarks, all but a few unsatisfiable instances are solved with one single SAT call. 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.

4.5 Discussion: Performance of Other SAT Solvers

In the previous section, we have experiments to evaluate the performance of incremental HCP solving using the SAT solver Sat4j with embedded Native BC. It shows good performance compared with the previous SAT-based method and other specialized methods.

One interesting point is that Sat4j runs on Java virtual machine and executed on a pure bytecode which is hardly competitive with the state-of-the-art solvers of the SAT competition. One question arising: is it possible to accelerate the incremental HCP solving method by using those award winning SAT solvers like Lingeling-ats, Glucose3 (core), and Minisat2.2 (core)? To answer this question, we carried out preliminary experiments using those solvers with their default settings. If a solver has both core (without pre-processing) and simp (with pre-processing) package, core is used. Note that all systems in this paper are implemented on a prototyping tool Scarab. Thus, when using Sat4j, the communication can be done on memory without invoking outer processes because

Table 4: Number of Solved Instances

Benchmark		Lin-S	Glu-S	Min-S	S4J-S	S4J-N
`color04`	(119)	43	88	88	90	92
`knight`	(11)	3	7	6	5	5
`tsplib`	(9)	2	9	9	9	9
Total	(129)	48	104	103	104	106

both Scarab and Sat4j run on a Java virtual machine. However, in the case of using other C++/C implementation solvers, the communication is done in a naive way: write/read text files.

Table 4 shows the number of instances solved by Lingeling-ats, Glucose3 (core), and Minisat2.2 (core) with Sequential Counter in addition to S4J-S and S4J-N. Those 3 SAT solvers do not have Native BC constraints. From the table, Glucose3 (core) and Minisat2.2 (core) solved almost the same number of instances as Sat4j solved. Considering the communication between those external SAT solvers and Scarab is done in a naive way, we can expect to accelerate the method by implementing tight integrated systems with those SAT solvers and implementing Native BC into them.

With this result, another question is why incremental methods using Lingeling-ats are much slower than the ones based on other solvers. To seek a reason, we tested several configurations of Lingeling-ats and found that pre-processing and/or in-processing would be not good for this incremental HCP solving. Specifically, they make the number of iterations larger. In Lingeling-ats, those x-processing is executed by default and it can be turned off using the option `--plain=1`, which also stops in-processing. To check this hypothesis in other solvers, using Sequential Counter, we additionally tested Minisat2.2 (simp) and Glucose3 (simp), which are Minisat2.2 (core) and Glucose3 (core) plus pre-processing.

	Lingeling-ats		Minisat2.2		Glucose3	
pre-processing?	yes	no	yes	no	yes	no
# of Solved	165	207	192	280	201	279
Median # of Iterations	22	5	47	7	54	7
Maximum # of Iterations	112	30	378	44	203	59

This table reports the number of instances solved using Lingeling-ats, Minisat2.2, and Glucose3 with or without pre-processing. We use a random benchmark sets the same as Section 4.2. The results clearly show the bad effect of pre-processing in terms of the number of instances solved and the convergence of iterations of incremental HCP solving. This results also provide us some light to tackle a future subject – how to reduce and control the number of iterations to reach a solution.

5 Related Work

In 2000, Clarke *et al.* proposed Counterexample-Guided Abstraction Refinement (CEGAR) in the context of model checking [8], which receives a program text and abstract functions are extracted from it. Following their work, there are some applications of CEGAR to Presburger Arithmetic [21], deciding the theory of Arrays [12], and the RNA-folding problem [13]. Although there are few applications for combinatorial problems such as graph problems, we apply CEGAR to HCP by relaxing connectivity constraints.

In the context of solving TSP, there is a traditional OR technique proposed in 80's which translates TSP into an assignment problem [7,22]. Jäger and Zhang [19] 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 [19] 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.

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. 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.
8. Edmund M. Clarke, Orna Grumberg, Somesh Jha, Yuan Lu, and Helmut Veith. Counterexample-guided abstraction refinement. In *CAV*, pages 154–169, 2000.
9. 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.
10. 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.
11. 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.
12. Vijay Ganesh and David L. Dill. A decision procedure for bit-vectors and arrays. In *CAV*, pages 519–531, 2007.
13. 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.
14. Ronald J. Gould. Advances on the Hamiltonian problem - a survey. *Graphs and Combinatorics*, 19(1):7–52, 2003.
15. Ronald J. Gould. Recent advances on the Hamiltonian problem: Survey III. *Graphs and Combinatorics*, 30(1):1–46, 2014.
16. Brahim Hnich, Toby Walsh, and Barbara M. Smith. Dual modelling of permutation and injection problems. *J. Artif. Intell. Res. (JAIR)*, 21:357–391, 2004.
17. 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.
18. 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.
19. 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.
20. Richard M. Karp. Reducibility among combinatorial problems. In *Complexity of Computer Computations*, pages 85–103, 1972.
21. Daniel Kroening, Joël Ouaknine, Sanjit A. Seshia, and Ofer Strichman. Abstraction-based satisfiability solving of presburger arithmetic. In *CAV*, pages 308–320, 2004.
22. Gilbert Laporte. The traveling salesman problem: An overview of exact and approximate algorithms. *European Journal of Operational Research*, 59(2):231–247, June 1992.
23. 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.
24. Amit Metodi and Michael Codish. Compiling finite domain constraints to sat with bee. *TPLP*, 12(4-5):465–483, 2012.

25. Steven David Prestwich. SAT problems with chains of dependent variables. *Discrete Applied Mathematics*, 130(2):329–350, 2003.
26. 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.
27. 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.
28. Takehide Soh, Naoyuki Tamura, and Mutsunori Banbara. Scarab: A rapid prototyping tool for SAT-based constraint programming systems. In *SAT*, pages 429–436, 2013.
29. Naoyuki Tamura, Akiko Taga, Satoshi Kitagawa, and Mutsunori Banbara. Compiling finite linear CSP into SAT. *Constraints*, 14(2):254–272, 2009.
30. 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.