Efficient SAT Techniques for Absolute Encoding of Permutation Problems: Application to Hamiltonian Cycles

Miroslav N. Velev[‡] **Ping Gao**

Aries Design Automation, LLC

[‡]Contact author: `miroslav.velev@aries-da.com`

Abstract

We study novel approaches for solving of hard combinatorial problems by translation to Boolean Satisfiability (SAT). Our focus is on combinatorial problems that can be represented as a permutation of *n* objects, subject to additional constraints. In the case of the Hamiltonian Cycle Problem (HCP), these constraints are that two adjacent nodes in a permutation should also be neighbors in the graph for which we search for a Hamiltonian cycle. We use the absolute SAT encoding of permutations, where for each of the *n* objects and each of its possible positions in a permutation, a predicate is defined to indicate whether the object is placed in that position. For implementation of this predicate, we compare the direct and logarithmic encodings that have been used previously, against 16 hierarchical parameterizable encodings of which we explore 416 instantiations. We propose the use of enumerative adjacency constraints—that enumerate the possible neighbors of a node in a permutation—instead of, or in addition to the exclusivity adjacency constraints—that exclude impossible neighbors, and that have been applied previously. We study 11 heuristics for efficiently choosing the first node in the Hamiltonian cycle, as well as 8 heuristics for static CNF variable ordering. We achieve at least 4 orders of magnitude average speedup on HCP benchmarks from the phase transition region, relative to the previously used encodings for solving of HCPs via SAT, such that the speedup is increasing with the size of the graphs.

Introduction

In the last decade, dramatic improvements were achieved in both the speed and capacity of SAT solvers, which are now up to 5 orders of magnitude faster and can handle problems that are up to 4 – 5 orders of magnitude bigger, e.g., (Eén and Sörensson 2005; Pipatsrisawat and Darwiche 2007; Huang 2007b). The new efficient SAT solvers open new possibilities for applying this technology. By translating hard Computer Science problems to equivalent SAT problems, we can directly benefit from the recent tremendous advances in SAT, and the constant stream of innovations in this extremely active research field, without having to reimplement the same optimizations in specialized tools for solving of specific problems. That approaches based on efficient translation to SAT can outperform other methods was demonstrated in the recent International Planning Competitions (`http://zeus.ing.unibs.it/ipc-5`), where first places in the optimal planning category were won by SAT-based planners (Kautz et al. 2006; Xing et al. 2006).

In this paper, we investigate efficient SAT techniques for solving of problems that can be reformulated as permutations, subject to additional constraints. Particularly, we do an in-depth study of Hamiltonian Cycle Problems (HCPs)—where the goal is to find a route in a graph by visiting each node exactly once and returning to the starting node—a known class of hard combinatorial problems, classified as NP-complete—see Prob. GT37 on p. 199 of (Garey and Johnson 1979). Another hard combinatorial problem, quasigroup completion (Kautz et al. 2001; Gomes and Shmoys 2002; Ansótegui et al. 2004; Velev and Gao 2009), can be reformulated as multiple permutations, subject to additional constraints; it has applications to design of experiments, and wavelength routing in switches on optical networks. Other combinatorial problems that can be viewed as permutations with constraints are discussed in (Hnich et al. 2004; Cadoli and Schaerf 2005). At NASA, problems that can be reformulated as permutations of tasks, subject to additional constraints, arise in preparing sites for human habitation (Frank 2009). The efficient encoding of real-world problems as equivalent SAT formulas is a challenge identified by Kautz and Selman (2007).

Previous methods for HCP solving by translation to SAT (Iwama and Miyazaki 1994; Hoos 1999; Prestwich 2003; Cadoli and Schaerf 2005) exploited only 2 simple SAT encodings for solving Constraint Satisfaction Problems (CSPs) via SAT—the log encoding (Iwama and Miyazaki 1994), and the direct encoding (de Kleer 1989). However, those authors present results for graphs with at most 24 nodes. Indeed, our experiments indicate that the previous encodings do not scale for graphs with 30 nodes—both the log and direct encoding do not complete the solving of a suite of 100 graphs, each with 30 nodes and in the phase-transition region (Cheeseman et al. 1991), in 300,000 seconds—while many of our strategies solve all instances in that suite in 30 seconds or less, resulting in an average speedup of at least 4 orders of magnitude, such that the speedup is increasing with the size of the graphs.

This paper makes five contributions: 1) the first study of the benefits from 16 hierarchical parameterizable SAT encodings for CSPs (Velev 2007)—where a hierarchy of

Encoding	Required Clauses			Predicate $index(v, i)$
	at-least-one	at-most-one	excluded-illegal-values	
log	———	———	$\neg l_{v2} \vee \neg l_{v1}$	$index(v, 0) = \neg l_{v2} \wedge \neg l_{v1}$ $index(v, 1) = \neg l_{v2} \wedge l_{v1}$ $index(v, 2) = l_{v2} \wedge \neg l_{v1}$
direct	$x_{v0} \vee x_{v1} \vee x_{v2}$	$\neg x_{v0} \vee \neg x_{v1}$ $\neg x_{v0} \vee \neg x_{v2}$ $\neg x_{v1} \vee \neg x_{v2}$	———	$index(v, 0) = x_{v0}$ $index(v, 1) = x_{v1}$ $index(v, 2) = x_{v2}$
muldirect	$x_{v0} \vee x_{v1} \vee x_{v2}$	———	———	$index(v, 0) = x_{v0}$ $index(v, 1) = x_{v1}$ $index(v, 2) = x_{v2}$

Table 1. The simple SAT encodings log, direct, and muldirect for indexing the values of a CSP variable *v* that has a domain of 3 values {0, 1, 2}. A dash indicates the absence of clauses of the corresponding type. The predicate *index*(*v*, *i*) is true when the domain value *i* is selected as the value of the CSP variable *v*.

simple SAT encodings is used to index the domain of a CSP variable, while benefiting from the characteristics of the different simple encodings—of which we explore 416 instantiations, when applied to solving of HCPs; 2) the use of enumerative adjacency constraints when representing an HCP as an equivalent SAT problem—using a SAT encoding for CSPs to enumerate the possible successors and/or predecessors of a node in the Hamiltonian cycle—instead of, or in addition to the exclusivity adjacency constraints that have been used previously to prevent two nodes that are not neighbors in the graph from appearing in consecutive positions on the Hamiltonian cycle; 3) 11 heuristics for choosing the first node in the Hamiltonian cycle, resulting in fewer constraints, and simpler formulas; 4) 8 heuristics for static CNF variable ordering based on the structure of the graph; and 5) experimental results, indicating at least 4 orders of magnitude average speedup—for both satisfiable and unsatisfiable benchmarks—relative to the previously used encodings, such that the speedup is increasing with the size of the graphs.

Background

Simple SAT Encodings for CSPs

Previous work on HCP solving by translation to SAT has used only two encodings—the log encoding in (Iwama and Miyazaki 1994; Hoos 1999), and the direct encoding in (Hoos 1999; Prestwich 2003; Cadoli and Schaerf 2005). Hnich et al. (2004) also used the direct encoding when translating to SAT other types of permutation problems. We will illustrate these encodings with the clauses that they require for a CSP variable *v* with a domain of 3 values, {0, 1, 2}—see Table 1:

Log. Uses a logarithmic number of Boolean variables in the size of the domain of each CSP variable, by employing all of the Boolean variables in order to select a value from the domain of that CSP variable. Requires clauses to exclude illegal values that are not in the domain of a CSP variable. This encoding was proposed by Iwama and Miyazaki (1994).

Direct. A new Boolean variable x_{vi} is introduced to encode whether a CSP variable *v* is assigned value *i* from its domain. For each CSP variable, the introduced Boolean variables are constrained with an at-least-one clause that ensures that the CSP variable is assigned at least one value, and at-most-one clauses that guarantee that only one value is assigned. This encoding was proposed by de Kleer (1989).

A variation of the direct encoding is:

Muldirect. The multivalued direct encoding is a variant of the direct encoding, where the at-most-one clauses are omitted (Selman et al. 1992). Therefore, a SAT solution could assign several domain values to a CSP variable, so that there is no longer a 1-to-1 correspondence between SAT and CSP solutions. From a multivalued SAT solution we extract a CSP solution by taking any one of the allowed values for a CSP variable.

Given a CSP variable *v*, its domain, and the *indexing Boolean variables* introduced for SAT encoding that domain, we will refer to an assignment to those indexing Boolean variables that selects the i^{th} domain value as an *indexing Boolean pattern* for that domain value for the given CSP variable, and will denote it with the predicate *index*(*v*, *i*). For a summary of other simple SAT encodings for CSPs, the reader is referred to (Velev 2007). In the current work, every indexing Boolean pattern is either a single literal (a variable or its negation), or a conjunction of literals, but in general can be an arbitrary Boolean function.

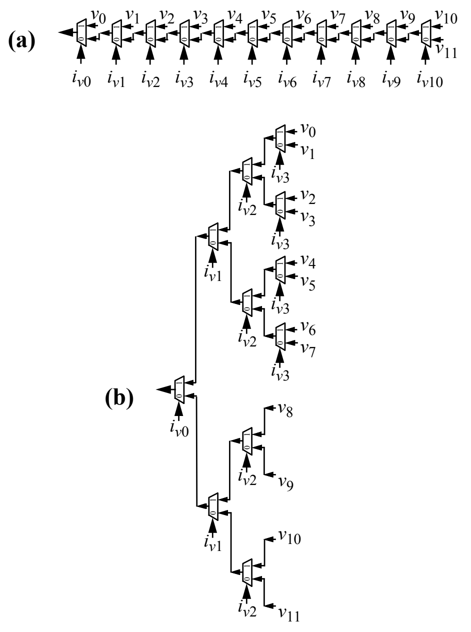
Figure 1. Two structural SAT encodings based on ITE trees for CSP variable v with domain of 12 values, $\{v_0, v_1, ..., v_{11}\}$: (a) ITE-linear; and (b) ITE-log. ITEs are shown as multiplexors. In general, the ITE tree for a CSP variable can have any structure.

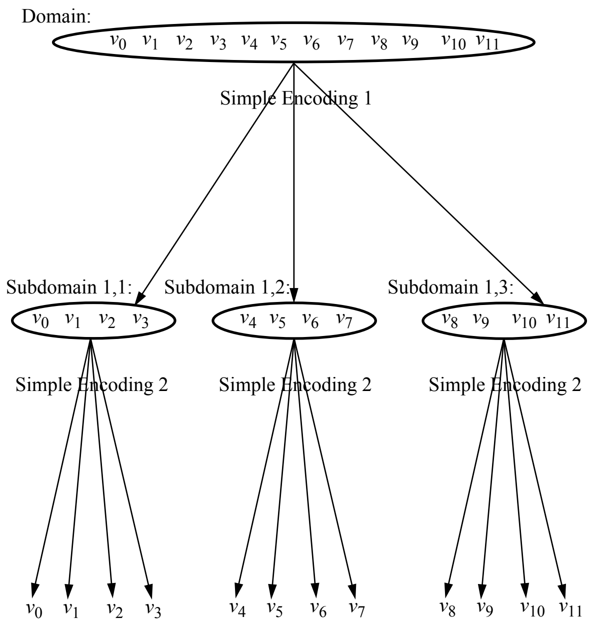

Figure 2. Example of a hierarchical, recursive, hybrid encoding. A domain of 12 values, $\{v_0, v_1, ..., v_{11}\}$, is divided into three subdomains by Simple Encoding 1 at level 1. Each subdomain is further divided into four parts by Simple Encoding 2 at level 2, where each part is a different domain value.

Structural SAT Encodings for CSPs

We can represent each CSP variable with a tree of ITE (for "if-then-else") operators that selects a value from this CSP variable's set of k domain values (Velev 2007). In this representation, an ITE operator ITE(i, t, e) takes three arguments: a Boolean variable i, such that if i is *true* then the ITE selects the then-argument t and otherwise the else-argument e, where the then- and else-arguments are either ITE subtrees or domain values that appear as leaves of the tree. Depending on the structure of the ITE tree introduced for a CSP variable, we can obtain various encodings. One extreme case is a chain of ITE operators—see Fig. 1.a. We call this SAT encoding *ITE-linear*, due to the linear structure of the ITE tree. Another extreme case is when the ITE tree is balanced, with every path in the tree going through either $\lceil \log_2(k) \rceil$ or $\lceil \log_2(k) \rceil - 1$ ITE operators—see Fig. 1.b. We call the resulting SAT encoding *ITE-log*. It can be viewed as a variant of the log encoding (see Table 1), where some of the indexing Boolean patterns do not contain the last indexing Boolean variable, so that no constraints are needed to exclude illegal indexing Boolean patterns, as in the log encoding. In general, the ITE tree for a CSP variable can have any structure.

Hierarchical SAT Encodings for CSPs

We can use a hierarchy of different simple SAT encodings (Velev 2007) in order to select the domain values of a CSP variable. Namely, we can first use one simple SAT encoding to partition the domain of that CSP variable into subdomains, and then a different simple SAT encoding to either select the values in each subdomain, or to further partition it into smaller subdomains, and so on. For a given domain or a subdomain, we restrict its subdomains to not overlap. Then, we can use the same SAT encoding (with the same set of indexing Boolean variables) to select the values in each subdomain at the same new level in the hierarchy, and/or to further divide the subdomains at that level into smaller subdomains—see Fig. 2. A domain value is selected if it gets selected in its corresponding subdomain at the lowest level in the hierarchy, and for each of the higher levels in the hierarchy, the corresponding larger subdomain that contains this value also gets selected by the SAT encoding for that level of the hierarchy. The global predicate *index*(v, i) is defined as the conjunction of the corresponding predicates for all levels of the hierarchy. If the total number of indexing Boolean patterns in a hierarchical encoding is greater than the number of domain values, the unused indexing Boolean patterns are prevented from occurring by clauses that require the negation of those indexing Boolean patterns to be true.

Kwon and Klieber (2007) proposed SAT encodings for CSP variables that can be viewed as hierarchical encodings with the direct encoding used at each level. In contrast, we consider hierarchical encodings that can have any simple encoding at each level of the hierarchy.

SAT Representation of HCPs

In this work, we apply the absolute SAT encoding of permutations in order to translate HCPs to equivalent SAT problems, as proposed by Iwama and Miyazaki (1994), and used by Hoos (1999), Prestwich (2003), and Cadoli and Schaerf (2005). The HCP instances are encoded as CSPs by enumerating vertex permutations of the given graph as solution candidates. For every vertex v, we introduce a CSP variable, whose value represents v's position in the permutation. Three types of constraints are introduced: 1) *complete occupancy constraints*, enforcing that each position in the permutation is occupied by a vertex; 2) *exclusivity positional constraints*, ensuring that only one vertex can appear at a given position in the permutation; and 3) *exclusivity adjacency constraints*, guaranteeing that if two vertices are not adjacent in the graph, then they cannot appear in consecutive positions in the permutation. Note that if the graph has n vertices, and so the domain of the CSP variable for each vertex is of size n, then the exclusivity positional constraints also ensure that the complete occupancy constraints will be satisfied, and so we do not need to impose them explicitly.

When searching for Hamiltonian paths that visit each vertex exactly once, the exclusivity adjacency constraints are imposed only for consecutive positions in the permutation. However, when searching for Hamiltonian cycles that must return to the first node, exclusivity adjacency constraints are also imposed between the last and first positions in the permutation. For another SAT encoding of permutations that was used for solving of HCPs, see (Prestwich 2003).

Using Hierarchical SAT Encodings to Represent Permutations

We can use any SAT encoding for CSP variables, including hierarchical encodings, in order to index the possible positions of an object in a permutation. While hierarchical encodings were shown to produce up to 3 orders of magnitude speedup on the DIMACS graph-coloring problems (Velev 2007), and up to 4 orders of magnitude speedup on detailed routings of complex FPGAs (Velev and Gao 2008), they did not accelerate the solving of quasigroup completion problems in our experiments for (Velev and Gao 2009), and have not been applied to SAT-based solving of HCPs, to the best of our knowledge. Thus, the question whether they would result in a speedup, given the structure of HCPs. Furthermore, we implemented new hierarchical parameterizable encodings than the ones studied in (Velev 2007; Velev and Gao 2008).

Enumerative Adjacency Constraints

Instead of the exclusivity adjacency constraints—which prevent two nodes that are not neighbors in the graph from appearing in consecutive positions in the permutation that represents the ordering of nodes in the Hamiltonian cycle—we can use *enumerative adjacency constraints* that, given a node at position i in the cycle, enumerate all of its neighbors from the graph and ensure that one of them will appear at position $i + 1$. The set of m neighbors of the node at position i is indexed with a new indexing scheme (either simple or hierarchical) that uses fresh indexing Boolean variables and selects one out of m elements.

Because of the indexing scheme for the position of each node in the permutation and the exclusivity positional constraints, which map a given node to exactly one position in the permutation for each assignment to all indexing Boolean variables, the new indexing scheme for choosing a neighbor can be reused for every position in the permutation.

Let $index\_fwd\_neighbor(v_p, v_q)$ be the indexing predicate introduced for selecting the neighbor of v_p to appear in the next consecutive position of the permutation, i.e., $index\_fwd\_neighbor(v_p, v_q)$ is true if neighbor v_q is selected to follow node v_p in the Hamiltonian cycle. Then, for each position i and each neighbor v_q, we add the corresponding enumerative adjacency constraint:

$$index(v_p, i) \wedge index\_fwd\_neighbor(v_p, v_q) \Rightarrow index(v_q, i + 1),$$

where $index(v_p, i)$ is the indexing Boolean pattern that maps node v_p to position i, and $index(v_q, i + 1)$ is the indexing Boolean pattern that maps node v_q to position $i + 1$. In the case where the indexing Boolean pattern $index(v_q, i + 1)$ is a conjunction of k literals, $l_1 \wedge l_2 \wedge ... \wedge l_k$, then the above constraint will result in k clauses in the CNF formula, one for each of the literals:

$$index(v_p, i) \wedge index\_fwd\_neighbor(v_p, v_q) \Rightarrow l_j, \quad j = 1, ..., k,$$

thus ensuring that if node v_p is at position i, and node v_q is chosen as its successor in the Hamiltonian cycle, then each literal in the indexing Boolean pattern $index(v_q, i + 1)$ will be assigned a value so that $index(v_q, i + 1)$ will be true, i.e., v_q will be in position $i + 1$. Since the above constraints are between a node and its successor in the permutation, we call them *forward enumerative adjacency constraints*.

We can define similar constraints between a node and its predecessor in the permutation, but based on a different indexing predicate with fresh indexing Boolean variables, $index\_bck\_neighbor(v_p, v_q)$:

$$index(v_p, i) \wedge index\_bck\_neighbor(v_p, v_q) \Rightarrow index(v_q, i - 1).$$

We call the resulting constraints *backward enumerative adjacency constraints*. Again, if the indexing Boolean pattern $index(v_q, i - 1)$ is a conjunction of k literals, the above constraint will result in k clauses in the CNF formula, one for each of the literals.

Note that the same neighbor v_q of node v_p cannot be selected simultaneously to appear as both a successor and a predecessor of v_p, because the indexing scheme for node v_q and the exclusivity positional constraints ensure that v_q can be in only one place in the permutation for each assignment to all indexing Boolean variables. Then, if both forward and backward enumerative adjacency constraints are used, we can optionally add the redundant constraints:

$$\neg index\_fwd\_neighbor(v_p, v_q) \vee \neg index\_bck\_neighbor(v_p, v_q),$$

for each neighbor v_q of each node v_p. We call these *exclu-*

sivity constraints between the forward and backward enumerative adjacency constraints.

The new indexing schemes for forward and backward enumerative adjacency constraints can be multivalued, i.e., can select several neighbors to appear at the forward or backward adjacent position at the same time, since the exclusivity positional constraints will ensure that only one of those nodes will actually appear in that position.

We can use only forward enumerative adjacency constraints, or both forward and backward enumerative adjacency constraints, or both forward and backward enumerative adjacency constraints with exclusivity constraints between them. Furthermore, we can use each of these combinations either alone or together with exclusivity adjacency constraints.

The enumerative adjacency constraints add extra Boolean variables and clauses to the CNF representation of the HCP. If the direct or muldirect encoding is used to enumerate the neighbors of each node, the number of extra variables is $O(n \cdot d)$, where n is the number of nodes in the graph, and d is the maximum degree of a node. However, it is well known that the number of Boolean variables and clauses is not indicative of the complexity of solving a CNF formula, and previous work has shown that overconstraining a CNF formula with redundant constraints may make it easier to solve, e.g., see (Kautz et al. 2001; Gomes and Shmoys 2002; Ansótegui et al. 2004).

While the exclusivity adjacency constraints (used by all previous researchers) exclude infeasible portions of the solution space, they leave it up to the SAT solver to deduce the feasible portions. In contrast, the enumerative adjacency constraints enumerate the feasible portions of the solution space, thus ensuring that the search will stay within them, eliminating inefficiency, and resulting in faster convergence to a solution.

Choosing the First Node in a Permutation When Solving HCPs via SAT

We exploit the observation that when the search is for a Hamiltonian cycle, as opposed to a Hamiltonian path, the first node in the cycle can be selected in any way, since all nodes have to be included in the cycle, and if a cycle exists then all nodes are symmetrical in it. Thus, if a cycle does not exist when a particular node is selected as a first node, then a cycle would not exist when any other node is selected as a first node.

By selecting the first node in the cycle, we have to enumerate all possible permutations of the other $n - 1$ nodes, where n is the number of nodes in the graph. However, we need to impose the exclusivity and/or enumerative adjacency constraints between the chosen first node and the second node (i.e., the first node in the permutation of the other $n - 1$ nodes), as well as between the chosen first node and the last node in the permutation. This eliminates the need for indexing Boolean variables for the chosen first node to encode its placement in a permutation of n nodes, possibly reduces the number of indexing Boolean variables used for each of the other $n - 1$ nodes since now they need to be placed in a permutation of $n - 1$ elements, and eliminates an $O(n^2)$ exclusivity positional constraints—if they are used—between the chosen first node and each of the other nodes for every position in a permutation of n nodes, thus resulting in simpler CNF formulas.

We implemented the following 11 heuristics for selecting the first node:

- *f1*—choose the first node in the graph description;
- *f2*—choose the first node of max. degree in the graph description;
- *f3*—choose the first node of min. degree in the graph description;
- *f4*—choose the first node of average degree in the graph description;
- *f5*—random choice;
- *f6*—choose a node of max. degree, and break ties based on a lesser sum of neighbors' degrees;
- *f7*—choose a node of max. degree, and break ties based on a greater sum of neighbors' degrees;
- *f8*—choose a node of average degree, and break ties based on a lesser sum of neighbors' degrees;
- *f9*—choose a node of average degree, and break ties based on a greater sum of neighbors' degrees;
- *f10*—choose a node of min. degree, and break ties based on a lesser sum of neighbors' degrees;
- *f11*—choose a node of min. degree, and break ties based on a greater sum of neighbors' degrees.

Static CNF Variable Ordering

In order to explore ways to reflect the structure of a graph when solving the HCP for it by translation to SAT, we implemented heuristics for static CNF variable ordering that is done in two steps. First, the nodes in the graph are sorted according to their degrees, using one of the following four approaches:

- *o1*—descending order of the node degrees, with ties broken based on greater sum of neighbors' degrees;
- *o2*—descending order of the node degrees, with ties broken based on lesser sum of neighbors' degrees;
- *o3*—ascending order of the node degrees, with ties broken based on greater sum of neighbors' degrees;
- *o4*—ascending order of the node degrees, with ties broken based on lesser sum of neighbors' degrees.

Second, following the node order from the first step above, the CNF variables introduced for each node are assigned priority levels in one of two methods:

- *A*—individually, first listing the indexing variables that encode the node's position in the permutation—such that if a hierarchical encoding is used then priority is given to the indexing variables for higher levels in the hierarchy where the domain is partitioned into larger subdomains—followed by the indexing variables that encode the enumerative adjacency constraints, if applicable;

- *B*—in one or two groups, such that the CNF variables in a group have the same priority level, where the first group consists of the indexing variables that encode the node's position in the permutation, and the second group (with a lower priority level), if applicable, consists of the indexing variables that encode the enumerative adjacency constraints.

Thus, the actual static CNF variable ordering is produced by applying one of the 4 node ordering approaches *o1 – o4*, followed by one of the 2 methods to assign priority levels to the CNF variables introduced for a node, resulting in 8 static CNF variable-ordering heuristics. When using a static variable order, a SAT solver would assign values to the CNF variables starting from the highest and proceeding to the lowest priority level. Hence, when method *A* is used to assign priority levels, resulting in a different priority level for each variable, the SAT solver will not use its dynamic variable ordering heuristic, but only its dynamic heuristic for selecting the value to assign to the next (statically determined) decision variable. In contrast, when method *B* is used to assign priority levels, all the variables in a group will have the same priority level, and the SAT solver will apply both its dynamic variable ordering heuristic—to order the variables from a group with the same priority level—and its dynamic heuristic to select the value of each variable. We implemented method *B* to explore the benefits from the corresponding capability in the SAT solver `rsat_3` (Pipatsrisawat and Darwiche 2007; Pipatsrisawat and Darwiche 2008).

The static CNF variable order is produced by our tool—that given a graph, generates a CNF formula encoding the feasibility of a Hamiltonian cycle in the graph—and is passed to the SAT solver together with the CNF formula. Since we use only complete SAT solvers, a static variable ordering will not affect the result, but may help to solve the problem faster.

In previous work on solving of CSPs by translation to SAT, static variable ordering was used when solving of graph-coloring problems (Velev 2007; Van Gelder 2008; Velev and Gao 2008), and quasigroup completion problems (Velev and Gao 2009). Hnich et al. (2004) researched static variable orderings for non-SAT-based CSP solvers applied to permutation problems.

Results

The experiments were run on a Dell Precision T7400 workstation with two 3.2-GHz quad-core Intel Xeon processors, 32 GB of 800-MHz memory, and Red Hat Enterprise Linux v5.3. (Only one CPU core was used for each experiment.) We used the graph generator by Vandegriend and Culberson[1] to produce random graphs.

We started with satisfiable benchmarks (guaranteed to have Hamiltonian cycles) by generating suites of 100 graphs from the phase-transition region (Cheeseman et al. 1991) that satisfy the ratio e / (n log n) = 1, where e is the number of edges and n the number of nodes in the graph, since that ratio was shown to result in the hardest instances (Frank and Martel 1995).

We compared 16 hierarchical parameterizable SAT encodings—of which we explored 416 instantiations—when used to represent the permutation of the vertices: ITE-linear-k+direct and ITE-linear-k+muldirect for k = 2, ..., 12; ITE-log-k+direct and ITE-log-k+muldirect for k = 2, ..., 12; direct-k+direct, direct-k+muldirect, muldirect-k+direct, and muldirect-k+muldirect for k = 2, ..., 12, such that the number of indexing Boolean variables used in the second level was computed based on the number of variables in the first level and the domain size; direct-k+(direct-n)*, direct-k+(muldirect-n)*, muldirect-k+(direct-n)*, muldirect-k+(muldirect-n)*, ITE-linear-k+(direct-n)*, ITE-linear-k+(muldirect-n)*, ITE-log-k+(direct-n)*, and ITE-log-k+(muldirect-n)* for k = 2, ..., 8 and n = 3, ..., 8, where the * means that the corresponding encoding is repeated starting from level 2, with a fresh set of indexing Boolean variables for each level, such that the number of levels introduced is determined by the domain size. We also ran experiments with 5 simple encodings: direct, muldirect, log, ITE-linear, and ITE-log.

We compared four SAT solvers: `satz215.2` (Li and Anbulagan 1997; Li 1999); `tinisat` (Huang 2007a; Huang 2007b); `minisat_1.14` (Eén and Sörensson 2005); and `rsat_3` (Pipatsrisawat and Darwiche 2007; Pipatsrisawat and Darwiche 2008). We found `rsat_3` to have the best performance on graphs of 30 or more vertices, and so used it for the experiments that we present.

We started with experiments for graphs with 20 vertices. Unlike Hoos (1999), who identified the direct encoding to outperform the log encoding when using SAT solvers that are no longer competitive, we found the log encoding to outperform the direct encoding. In particular, the log encoding resulted in 319 sec to solve all 100 CNF formulas from graphs with 20 vertices, while the direct encoding took approximately 1,500 sec. Of the hierarchical encodings, the best performance was due to the ITE-linear-2+direct encoding, which required 85 sec, closely followed by the ITE-log-2+direct encoding with 89 sec, and then the ITE-linear-2+muldirect with 113 sec. We repeated these experiments for the 100 graphs with 25 vertices: the log encoding required approximately 40,000 sec, in contrast to the ITE-linear-2+direct encoding that resulted in approximately 4,000 sec, i.e., an average speedup of an order of magnitude, with the speedups on individual benchmarks reaching 3 orders of magnitude. However, the hierarchical SAT encodings did not scale well for larger graphs.

We implemented the enforcement of enumerative adjacency constraints, using either the direct or muldirect encoding to index the neighbors. The enumerative adjacency constraints could be used instead of, or in addition to the exclusivity adjacency constraints that have been used previously. We found that the direct encoding, when used to encode the permutation of the vertices, produced the greatest speedup with enumerative adjacency constraints, rela-

1.http://web.cs.ualberta.ca/~joe/Theses/HCarchive/main.html tive to the log encoding or any of the hierarchical encodings.

For satisfiable benchmarks, the greatest speedup and scalability were obtained when we used direct encoding for indexing the position of each node in the permutation, both forward and backward enumerative adjacency constraints and exclusivity constraints for them with the direct encoding used to index the neighbors, keeping the exclusivity adjacency constraints from the previous methods, choosing the first node in the cycle based on heuristic *f11*, and using static CNF variable ordering heuristic *o4.B*. This strategy took 11 seconds for the suite with 20-node graphs; 12 seconds for the suite with 25-node graphs; 20 seconds for the suite with 30-node graphs; 31 seconds for the suite with 40-node graphs; 207 seconds for the suite with 50-node graphs; 291 seconds for the suite with 60-node graphs; 2,395 seconds for the suite with 70-node graphs; 5,291 seconds for the suite with 80-node graphs; approximately 68,000 seconds for the suite with 90-node graphs; and approximately 120,000 seconds for the suite with 95-node graphs, where each of these suites had 100 graphs. (The time to generate each CNF formula, as well as to map each satisfying assignment to a sequence of nodes and verify that they are indeed a Hamiltonian cycle, took less than 1% of the SAT solving time and is included in the times reported above.) In contrast, when we ran experiments with the previously used encodings log and direct on the same suite of 100 graphs with 30 nodes each, the SAT solving with either encoding took more than 300,000 seconds, such that the log encoding could complete only the first 10 benchmarks in that time interval, while the direct encoding could not complete even the first benchmark. That is, the time for solving of that suite was reduced from more than 300,000 seconds—with only 10 out of 100 benchmarks solved—to 20 seconds with all of the benchmarks solved, or an average speedup of at least 4 orders of magnitude. Furthermore, the speedup is increasing with the size of the graphs.

Of the 11 heuristics for choosing the first node in the cycle, the best performance had *f11* and *f10*, which produced an average speedup of up to 3×, relative to using the first node in the graph description. Of the 8 heuristics for static CNF variable ordering, *o4.B* had the best performance for satisfiable instances, resulting in an average speedup of up to 5× relative to using the dynamic variable ordering heuristic in the SAT solver.

We also conducted experiments with 50 graphs of 25 nodes each that do not have Hamiltonian cycles; they were also generated randomly, but satisfying the ratio $e / (n \log n) = 0.5$. Using the best strategy from solving satisfiable benchmarks—the direct encoding for indexing the position of each node in the permutation, both forward and backward enumerative adjacency constraints and exclusivity constraints for them with the direct encoding used to index the neighbors, keeping the exclusivity adjacency constraints from the previous methods, and choosing the first node in the cycle based on heuristic *f11*, but without static variable ordering—took a total of 8 seconds to generate all CNF formulas and find each of them unsatisfiable. (Using static variable ordering resulted in longer CPU time.) In contrast, with both the log and direct encoding, the solving of those benchmarks did not complete in 800,000 seconds. Thus, our techniques result in at least 5 orders of magnitude of average speedup for unsatisfiable benchmarks as well.

When we increased the ratio $e / (n \log n)$ to 1.25 and then to 1.5, the graphs became much easier to find Hamiltonian cycles in, and the resulting suites of 100 graphs were solved up to 2 orders of magnitude faster than the corresponding satisfiable suites where the graphs were generated with this ratio set to 1.

Conclusion

We did an in-depth study of HCP solving based on the absolute SAT encoding of permutations with constraints. We found the hierarchical SAT encoding ITE-linear-2+direct to result in an average speedup of an order of magnitude relative to the log encoding (that significantly outperformed the direct encoding) on a suite of 100 graphs with 25 nodes each, generated in the phase-transition region. However, the hierarchical SAT encodings did not scale well for larger graphs.

The main contribution of the paper is the use of forward and backward enumerative adjacency constraints, together with exclusivity constraints between them, in addition to the exclusivity adjacency constraints that have been applied previously. We achieved at least 4 orders of magnitude average speedup when solving the HCP for both satisfiable and unsatisfiable benchmarks—relative to the previously used direct and log encodings—such that the speedup is increasing with the size of the graphs. Heuristic *f11* for selecting the first node in the cycle produced speedup of up to 3 times. For satisfiable benchmarks, the best performance was due to the static CNF variable ordering heuristic *o4.B* that resulted in up to a factor of 5 speedup, relative to using the SAT solver's dynamic variable ordering heuristic, which had the best performance for unsatisfiable benchmarks.

We expect that the presented techniques can be adapted to efficiently solve other classes of permutation problems. The availability of many efficient translations to SAT will allow us to design robust portfolios of parallel strategies for solving of HCPs, which we will investigate in our future research.

References

Ansótegui, C.; del Val, A.; Dotú, I.; Fernández, C.; and Manyà, F. 2004. Modeling Choices in Quasigroup Completion: SAT vs. CSP. In *Proceedings of the Nineteenth National Conference on Artificial Intelligence, Sixteenth Conference on Innovative Applications of Artificial Intelligence*, 137–142.

Cadoli, M.; and Schaerf, A. 2005. Compiling Problem Specifications into SAT. *Artificial Intelligence* 162(1–2): 89–120.

Cheeseman, P.; Kanefsky, B.; and Taylor, W. M. 1991. Where the Really Hard Problems Are. In P*roceedings of the 12th International Joint Conference on AI (IJCAI'91)*, 331–337.

de Kleer, J. 1989. A Comparison of ATMS and CSP Techniques. In *Proceedings of the 11th Int'l. Joint Conference on*

Artificial Intelligence (IJCAI'89), 290–296.

Eén, N.; and Sörensson, N. 2005. MiniSat—A SAT Solver with Conflict-Clause Minimization. In *Proceedings of the Eighth International Conference on Theory and Applications of Satisfiability Testing*.

Frank, J.; and Martel, C. U. 1995. Phase Transitions in the Properties of Random Graphs. In *Proceedings of Principles and Practice of Constraint Programming (CP'95)*.

Frank, J. 2009. Personal Communication.

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

Gomes, C. P.; and Shmoys, D. B. 2002. The Promise of LP to Boost CSP Techniques for Combinatioral Problems. In *Proceedings of the International Conference on Integration of AI and OR Techniques in Constraint Programming for Combinatorial Optimization Problems (CP-AI-OR'02)*, 291–305.

Hnich, B.; Walsh, T.; and Smith, B. M. 2004. Dual Modelling of Permutation and Injection Problems. *Journal of Artificial Intelligence Research (JAIR)* 21: 357–391.

Hoos, H. H. 1999. SAT-Encodings, Search Space Structure, and Local Search Performance. In *Proceedings of the 16th Int'l. Joint Conference on Artificial Intelligence (IJCAI'99)*, 296–303.

Huang, J. 2007a. The Effect of Restarts on the Efficiency of Clause Learning. In *Proceedings of the 20th Int'l. Joint Conference on Artificial Intelligence (IJCAI'07)*, 2318–2323.

Huang, J. 2007b. A Case for Simple SAT Solvers. In *Proceedings of the 13th International Conference on Principles and Practice of Constraint Programming*, 839–846.

Iwama, K.; and Miyazaki, S. 1994. SAT-Varible Complexity of Hard Combinatorial Problems. In *Proceedings of the IFIP 13th World Computer Congress* (1): 253–258.

Li, C. M.; and Anbulagan. 1997. Heuristics Based on Unit Propagation for Satisfiability Problems. In *Proceedings of the Fifteenth International Joint Conference on Artificial Intelligence (IJCAI'97)*, 366–371.

Li, C. M. 1999. A Constraint-Based Approach to Narrow Search Trees for Satisfiability. *Inf. Process. Letters* 71(2): 75–80.

Kautz, H. A.; Ruan, Y.; Achlioptas, D.; Gomes, C. P.; Selman, B.; and Stickel, M. E. 2001. Balance and Filtering in Structured Satisfiable Problems. In *Proceedings of the 17th Int'l. Joint Conference on Artificial Intelligence (IJCAI'01)*, 351–358.

Kautz, H.; Selman, B.; and Hoffmann, J. 2006. SatPlan: Planning as Satisfiability. *5th Int'l Planning Competition, Int'l Conf. on Automated Planning and Scheduling (ICAPS'06)*.

Kautz, H. A.; and Selman, B. 2007. The state of SAT. *Discrete Applied Mathematics* 155(12): 1514–1524.

Kwon, G.; and Klieber, W. 2007. Efficient CNF Encoding for Selecting 1 from N Objects. In *Proceedings of the Fourth Workshop on Constraints in Formal Verification*.

Pipatsrisawat, K.; and Darwiche, A. 2007. A Lightweight Component Caching Scheme for Satisfiability Solvers. In *Proceedings of the 10th International Conference on Theory and Applications of Satisfiability Testing (SAT'07)*, 294–299.

Pipatsrisawat, K.; and Darwiche, A. 2008. A New Clause Learning Scheme for Efficient Unsatisfiability Proofs. In *Proceedings of the Twenty-Third AAAI Conference on Artificial Intelligence*, 1481–1484.

Prestwich, S. D. 2003. SAT Problems with Chains of Dependent Variables. *Discrete Applied Mathematics* 130(2): 329–350.

Selman, B.; Levesque, H. J.; and Mitchell, D. G. 1992. A New Method for Solving Hard Satisfiability Problems. In *Proceedings of the 10th National Conference on Artificial Intelligence (AAAI'92)*, 440–446.

Van Gelder, A. 2008. Another Look at Graph Coloring via Propositional Satisfiability. *Discrete Applied Mathematics* 156 (2): 230–243.

Velev, M. N. 2007. Exploiting Hierarchy and Structure to Efficiently Solve Graph Coloring as SAT. In *Proceedings of the International Conference on Computer-Aided Design (ICCAD'07)*, 135–142.

Velev, M. N.; and Gao, P. 2008. Comparison of Boolean Satisfiability Encodings on FPGA Detailed Routing Problems. In *Proceedings of Design, Automation and Test in Europe (DATE '08)*, 1268–1273.

Velev, M. N.; and Gao, P. 2009. Efficient SAT-Based Techniques for Design of Experiments by Using Static Variable Ordering. In *Proceedings of 10th International Symposium on Quality Electronic Design (ISQED '09)*, 371–376.

Xing, Z.; Chen, Y.; and Zhang, W. 2006. MaxPlan: Optimal Planning by Decomposed Satisfiability and Backward Reduction. *5th Int'l Planning Competition, Int'l Conf. on Automated Planning and Scheduling (ICAPS'06)*, 53–56.