In Pursuit of an Efficient SAT Encoding for the Hamiltonian Cycle Problem

Neng-Fa Zhou

CUNY Brooklyn College & Graduate Center

Abstract. SAT solvers have achieved remarkable successes in solving various combinatorial problems. Nevertheless, it remains a challenge to find an efficient SAT encoding for the Hamiltonian Cycle Problem (HCP), which is one of the most well-known NP-complete problems. A central issue in encoding the HCP into SAT is how to prevent sub-cycles, and one well-used technique is to map vertices to different positions. The HCP can be modeled as a single-agent path-finding problem. If the agent occupies vertex i at time t, and occupies vertex j at time $t+1$, then vertex j's position must be the successor of vertex i's. In this paper, we compare three encodings for the successor function, namely, the *unary encoding*, the *binary adder encoding*, and the *LFSR encoding* that uses a linear-feedback-shift register. We also propose a preprocessing technique that rules out a position from consideration for a vertex and a time if the agent cannot occupy the vertex at the time. Our study has surprisingly revealed that, with optimizations and preprocessing, the binary adder encoding is a clear winner: it solved some instances of the knight's tour problem that had been beyond reach for eager encoding methods, and solved more instances of the Flinders HCP challenge set than other encodings.

1 Introduction

The Hamiltonian Cycle Problem (HCP) is one of the most well-known NP-complete problems. Given a graph, which is either directed or undirected, the goal of the HCP is to find a cycle in the graph that includes each and every vertex exactly once. As the HCP occurs in many combinatorial problems, the global constraint `circuit(`G`)` has become indispensable in constraint programming (CP) systems. Given a graph G represented by a list of domain variables, the constraint ensures that any valuation of the variables constitutes a Hamiltonian cycle.

SAT solvers have achieved remarkable successes in solving combinatorial problems, ranging from formal methods [9, 19, 24], planning [22, 33], answer set programming [4, 12], to general constraint satisfaction problems (CSPs) [2, 17, 20, 30, 35, 36, 38]. The key issue in encoding the HCP into SAT is how to prevent sub-cycles. A naive encoding, which bans sub-cycles in every subset of vertices, requires an exponential number of clauses. One common technique used in SAT encodings for the HCP is to map vertices to different positions so that no sub-cycles can be formed during search. The direct encoding of the mapping, which

requires $O(n^3)$ clauses for a graph of n vertices in the worst case, does not scale well for large graphs [16, 26, 31]. In order to circumvent the explosive code size of the eager encoding approach, researchers have proposed lazy approaches, such as satisfiability modulo acyclicity [1] that incorporates reachability checking during search, and incremental SAT solving that incrementally adds clauses to ban sub-cycles [34]. Recently inspired by the log encoding [18], Johnson proposed a compact encoding for the HCP, which employs a linear-feedback-shift register (LFSR) for the successor function [21].

This paper continues the pursuit of an efficient SAT encoding for the HCP. The HCP can be modeled as a single-agent path-finding problem. Given a graph of n vertices, the agent resides at the start vertex at time 1, moves to a neighboring vertex in each step, and at time $n + 1$ be back at the start vertex after having visited all the vertices. Each vertex is mapped to a distinct position. If the agent occupies vertex i at time t, and occupies vertex j at time $t + 1$, then vertex j's position must be the successor of vertex i's. We call this encoding *distance encoding*. In this paper, we compare three encodings for the successor function, namely, the *unary encoding*, the *binary adder encoding*, and the *LFSR encoding*. We also propose a preprocessing technique that rules out a position from consideration for a vertex and a time if the agent cannot occupy the vertex at the time.

The experimental results show that, with optimizations and preprocessing, the binary adder encoding outperforms the unary and the LFSR encodings. The binary adder encoding solved some instances of the knight's tour problem that had been beyond reach for eager encoding methods, and solved more instances of the Flinders HCP challenge set[1] than other encodings.

2 Preliminaries

2.1 The `circuit` Constraint

A well-known representation of a directed graph in CP is to use a domain variable for each vertex in the graph, where the domain represents the set of neighboring vertices. For example, Figure 1 gives a directed graph and its representation using domain variables, where vertex `i` is represented by the domain variable V_i (i = 1,2,3,4), and the domain of V_i indicates the outgoing arcs from vertex `i`.

Let G `= [`$V_1, V_2, \ldots, V_n$`]` be a list of domain variables representing a graph. A valuation $V_i = j$ of the domain variables represents a subgraph of G that consists of arcs (i, j) $(i \in 1 \ldots n, j \in 1 \ldots n)$. The `circuit(`$G$`)` constraint enforces that the subgraph represented by a valuation of the domain variables forms a Hamiltonian cycle. For example, for the graph in Figure 1, `[2,4,1,3]` is a solution because `1→2, 2→4, 4→3, 3→1` is a Hamiltonian cycle, but `[2,1,4,3]` is not because the graph `1→2, 2→1, 3→4, 4→3` contains two sub-cycles. In the following, we assume that $n > 1$, the vertices are numbered from 1, and the graph is anti-reflexive, meaning $V_i \neq i$ for $i \in 1 \ldots n$.

[1] `http://fhcp.edu.au/fhcpcs`

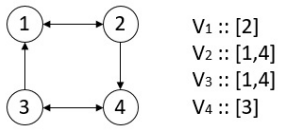

Fig. 1. A directed graph and its representation using domain variables

2.2 Direct Encoding of Domain Variables

Let $X :: \{a_1, a_2, \ldots, a_n\}$ be a domain variable. The *direct encoding* [8] introduces a Boolean variable B_i for $B_i \Leftrightarrow X = a_i$ $(i \in 1..n)$, and generates the constraint *exactly-one*$(B_1, B_2, \ldots, B_n)$, which is converted to a conjunction of an *at-least-one* constraint and an *at-most-one* constraint. The *at-least-one*$(B_1, B_2, \ldots, B_n)$ is encoded as the clause $B_1 \vee B_2 \vee \ldots \vee B_n$. The *at-most-one*$(B_1, B_2, \ldots, B_n)$ can be naively encoded as $\neg B_i \vee \neg B_j$, for $i \in 1..n$, $j \in 1..n$, and $i \neq j$. The 2-product encoding [5] for *at-most-one* is more compact than the naive encoding, but it introduces temporary Boolean variables.

2.3 Log Encoding of Domain Variables

The *log encoding* [18] is more compact than the direct encoding. The *sign-and-magnitude* log encoding uses a sequence of Boolean variables for the magnitude. If there are values of both signs in the domain, then the encoding uses another Boolean variable for the sign. Each combination of values of the Boolean variables represents a value for the domain variable.

Under the log encoding, each domain variable can be treated as a truth table, and a logic optimizer can be utilized to find CNF clauses for it. The Quine-McCluskey (QM) algorithm [32, 27] is popular for two-level logic optimization. A *product* is a conjunction of literals. Given a truth table, each tuple is a product, called a *minterm*, that involves all the inputs. A minterm is in the *on-set* if its output is required to be 1, in the *off-set* if the output is required to be 0, and in the *don't-care-set*, otherwise. A product of literals is an *implicant* of a truth table if it entails no minterms in the off-set. A *prime implicant* is an implicant that is not implied by any other implicant. For a truth table, the QM algorithm first computes all the prime implicants of the table, and then finds a minimal set of prime implicants that covers all the minterms in the on-set and none of the minterms in the off-set. The second step of the QM algorithm requires solving the minimum set-covering problem, which is NP-hard [10]. The Espresso logic optimizer [3] only computes a partial set of prime implicants based on heuristics, and therefore a smaller set-covering problem.

For example, consider the domain variable $X :: [-2, -1, 2, 1]$. The log encoding uses one Boolean variable, S, to encode the sign, and two variables, X_1 and X_0, to encode the magnitude. A naive encoding with *conflict clauses* [11] for the domain requires four clauses:

$$\begin{array}{ll} \neg S \vee \neg X_1 \vee \neg X_0 & (X \neq -3) \\ \neg S \vee X_1 \vee X_0 & (X \neq -0) \\ S \vee X_1 \vee X_0 & (X \neq 0) \\ S \vee \neg X_1 \vee \neg X_0 & (X \neq 3) \end{array}$$

Each of these clauses corresponds to a no-good value in $\{-3, -0, 0, 3\}$, where -0 denotes the negative 0. The logic optimizer Espresso only uses two clauses:

$$\begin{array}{l} X_0 \ \vee \ X_1 \\ \neg X_0 \ \vee \ \neg X_1 \end{array}$$

Note that the sign variable is optimized away.

3 The Distance Encoding for HCP

The `circuit(`G`)` constraint, where G `= [`$V_1, V_2, \ldots, V_n$`]`, enforces the following: (1) each of the vertices has exactly one incoming arc and exactly one outgoing arc; (2) each of the proper subgraphs of G is a tree, meaning that the subgraph is connected and the number of vertices is 1 greater than the number of arcs. A SAT encoding based on these properties does not use any extra variables, but requires an exponential number of clauses.

The *distance encoding* for HCP employs a matrix of Boolean variables H of size $n \times n$ for the Hamiltonian cycle. The entry H_{ij} is 1 if and only if the arc (i, j) occurs in the resulting Hamiltonian cycle.

The following *channeling constraints* connects the matrix H and the original domain variables $[V_1, V_2, \ldots, V_n]$:

$$\text{For each } i \in 1..n,\ j \in 1..n,\ i \neq j\text{: } H_{ij} \Leftrightarrow V_i = j \qquad (1)$$

Since each variable V_i takes only one value, constraint (1) entails that each vertex has exactly one outgoing arc. The following *degree* constraints ensure that each vertex has exactly one incoming arc:

$$\text{For each } j \in 1..n\text{: } \textstyle\sum_{i=1}^{n} H_{ij} = 1 \qquad (2)$$

For each pair of vertices (i, j) $(i \in 1..n, j \in 1..n)$, if the arc (i, j) is not in the original graph G, then the entry H_{ij} is set ot 0. Therefore, the number of Boolean variables in H equals the number of arcs in G.

Graph H that satisfies constraints (1) and (2) may contain sub-cycles. In order to ban sub-cycles, the distance encoding maps each vertex to a distinct position. Let $p(i)$ be the position of vertex i, $s(p)$ denote the successor of position p,[2] and $s^k(p)$ be the kth successor of p. Assume that vertex 1 is visited first, and it is mapped to position 1.[3] The following constraints ensure that the cycle starts at 1 and ends at 1:

[2] The successor function may generate a different sequence of numbers from the natural number sequence. So the successor of 1 may not be 2, depending on the encoding of the successor function.

[3] A good heuristic is to start with a vertex that has the smallest degree [37].

For each $i \in 2..n$:

$$H_{1i} \Rightarrow p(i) = s(1) \quad (3)$$

$$H_{i1} \Rightarrow p(i) = s^{n-1}(1) \quad (4)$$

Constraint (3) ensures that if there is an arc from vertex 1 to vertex i then i's position is the successor of 1. Constraint (4) ensures that if there is an arc from vertex i to vertex 1 then i's position is the $(n-1)$th successor of 1.

In addition to the above constraints, the following constraints ensure that the arcs are connected and the vertices are positioned successively:

For each $i \in 2..n,\ j \in 2..n,\ i \neq j$:

$$H_{ij} \Rightarrow p(j) = s(p(i)) \quad (5)$$

Constraint (5) ensures that vertex j is positioned immediately after vertex i if arc (i, j) is in the Hamiltonian cycle.

Theorem 1. *Constraints (1) - (5) guarantee that the graph represented by H is Hamiltonian.*

Proof. Constraints (1) and (2) entail that each vertex in graph H has exactly one incoming arc and exactly one outgoing arc, and therefore they guarantee that graph H is cyclic. Assume that the cycle in which vertex 1 occurs is:

$$1 \rightarrow v_2 \rightarrow v_3 \rightarrow \ldots \rightarrow v_k \rightarrow 1$$

According to constraints (3) - (5), the following conditions hold:

$$p(v_2) = s(1)$$
$$p(v_i) = s(p(v_{i-1})) \text{ for } i \in 3..k$$
$$p(v_k) = s^{n-1}(1)$$

These conditions entail $k = n$. Therefore, graph H includes all the vertices and is Hamiltonian.

The final code size of the distance encoding depends on how the successor function is encoded. We analyze here the code size of constraints (1) and (2), which is not dependent on the successor function. The number of Boolean variables in H equals the number of arcs in G. Constraint (1) mimics the direct encoding of domain variables. Both constraint (1) and constraint (2) are encoded as *exactly-one* constraints. Let d be the maximum degree in G. If the 2-product encoding [5] is used for *at-most-one*, then constraints (1) and (2) introduce $O(n \times \sqrt{d})$ new Boolean variables and require $O(n \times d)$ clauses.

4 Three Encodings of the Successor Function

There are several different ways to encode the successor function $p(j) = s(p(i))$ used in constraint (5). This section gives three such encodings, namely, the unary encoding, the binary adder encoding, and the linear-feedback-shift-register (LFSR) encoding.

4.1 Unary Encoding

The unary encoding of the successor function employs a matrix U of Boolean variables of size $n \times n$, where $U_{ip} = 1$ iff vertex i's position is p for $i \in 1..n$ and $p \in 1..n$. Since vertex 1 is visited first, U_{11} is initialized to 1. Each vertex is visited exactly once, so we have:

$$\text{For each } i \in 1..n\text{: } \sum_{p=1}^{n} U_{ip} = 1 \qquad (6)$$

For each vertex i ($i \in 1..n$), there is exactly on position p ($p \in 1..n$) for which U_{ip} is 1.

Constraints (3)-(5) given in the previous section are translated into the following under the unary encoding:

$$\begin{aligned}
&\text{For each } i \in 2..n\text{:}\\
&\qquad H_{1i} \Rightarrow U_{i2} && (3')\\
&\qquad H_{i1} \Rightarrow U_{in} && (4')\\
&\text{For each } i \in 2..n,\ j \in 2..n,\ i \neq j,\ p \in 2..(n-1)\text{:}\\
&\qquad H_{ij} \wedge U_{ip} \Rightarrow U_{j(p+1)} && (5')
\end{aligned}$$

Constraint (3') ensures that if there is an arc from vertex 1 to vertex i then vertex i's position is 2. Constraint (4') ensures that if there is an arc from vertex i to vertex 1 then vertex i's position is n. Constraint (5') ensures that if arc (i, j) is in the Hamiltonian cycle, and vertex i's position is p, then vertex j's position is $p + 1$. The constraints (3')-(5') entail that for each position there is exactly one vertex mapped to it ($\sum_{i=1}^{n} U_{ip} = 1$ for $p \in 1..n$).

The two dimensional array U has $O(n^2)$ variables. In addition, some temporary Boolean variables are introduced by the *exactly-one* constraints in (6). The number of clauses is dominated by constraint (5'), which requires $O(n^2 \times d)$ clauses to encode, where d is the maximum degree in G.

4.2 Binary Adder Encoding

The binary adder encoding of the successor function employs a log-encoded domain variable P_i for each vertex i, whose domain is the set of possible positions for the vertex. As all the positions are positive, no sign variables are needed in the encoding.

Since vertex 1 is visited first, $P_1 = 1$. Constraints (3)-(5) given above are translated into the following under the log encoding:

$$\begin{aligned}
&\text{For each } i \in 2..n\text{:}\\
&\qquad H_{1i} \Rightarrow P_i = 2 && (3'')\\
&\qquad H_{i1} \Rightarrow P_i = n && (4'')\\
&\text{For each } i \in 2..n,\ j \in 2..n,\ i \neq j\text{:}\\
&\qquad H_{ij} \Rightarrow P_j = P_i + 1 && (5'')
\end{aligned}$$

The efficiency of the encoding heavily depends on the encoding of the successor function $P_j = P_i + 1$ used in constraint (5").

$\neg X_4 \vee \neg X_3 \vee \neg Y_4 \vee Y_3$
$X_4 \vee \neg X_3 \vee Y_4 \vee Y_3$
$X_2 \vee C_1 \vee \neg Y_2$
$\neg X_3 \vee \neg X_2 \vee \neg X_1 \vee \neg C_1 \vee \neg Y_3$
$X_5 \vee X_3 \vee \neg Y_5$
$X_3 \vee \neg X_2 \vee \neg X_1 \vee \neg C_1 \vee Y_3$
$\neg X_2 \vee C_1 \vee Y_2$
$X_3 \vee \neg C_6$
$\neg X_5 \vee \neg X_4 \vee \neg Y_5 \vee Y_4$
$X_4 \vee X_3 \vee \neg Y_4$
$X_5 \vee \neg Y_5 \vee \neg Y_3$
$X_3 \vee X_2 \vee \neg Y_3$
$X_2 \vee \neg X_1 \vee \neg C_1 \vee Y_2$
$X_5 \vee \neg X_4 \vee Y_5 \vee Y_4$
$\neg C_6 \vee \neg Y_3$
$\neg X_3 \vee X_2 \vee Y_3$
$X_2 \vee \neg Y_2 \vee \neg Y_1$
$X_4 \vee \neg Y_4 \vee \neg Y_3$
$X_5 \vee \neg Y_5 \vee \neg Y_4$
$X_3 \vee \neg Y_3 \vee \neg Y_2$
$\neg X_1 \vee C_1 \vee Y_1$
$X_1 \vee \neg C_1 \vee Y_1$
$\neg X_2 \vee Y_2 \vee \neg Y_1$
$\neg X_3 \vee Y_3 \vee \neg Y_2$
$\neg C_6 \vee \neg Y_5$
$\neg X_5 \vee C_6 \vee Y_5$
$\neg C_6 \vee \neg Y_4$
$X_1 \vee C_1 \vee \neg Y_1$

Fig. 2. The five-bit adder for $<X_5X_4X_3X_2X_1> + C_1 = <C_6Y_5Y_4Y_3Y_2Y_1>$

Let X's log encoding be $<X_{m-1}X_{m-2}\ldots X_1X_0>$ and Y's log encoding be $<Y_{m-1}Y_{m-2}\ldots Y_1Y_0>$. Consider the addition:

$$\begin{array}{rr} & X_{m-1} \ldots X_1 \; X_0 \\ + & \qquad\qquad 1 \\ \hline & Y_{m-1} \ldots Y_1 \; Y_0 \end{array}$$

A naive encoding performs the addition bit-by-bit from the lowest bit position to the highest bit position. If a half-adder is used for each bit position, then the addition requires $m-1$ carry variables and 7 clauses for each bit position.[4]

A more efficient encoding performs the addition as follows: For the lowest bit position, it imposes $Y_0 = \neg X_0$ and $C_1 = X_0$, meaning that the result bit Y_0 is the negation of X_0, and the carry out C_1 is the same as X_0. The constraint $Y_0 = \neg X_0$ is encoded as two clauses: $Y_0 \vee X_0$ and $\neg Y_0 \vee \neg X_0$, and the constraint $C_1 = X_0$ only passes X_0 as the carry out, and requires no clauses. For the next five bit positions, it uses the five-bit adder given in Figure 2 to perform $<X_5X_4X_3X_2X_1> + C_1 = <C_6Y_5Y_4Y_3Y_2Y_1>$, which uses a new Boolean variable (C_6) and 25 clauses. For the remaining bit positions, it performs the addition five bits a time, using the same five-bit adder.

[4] For the lowest and highest bit positions, the half-adder can be optimized so that it uses 4 clauses.

Under the log encoding, each of the position variables P_i ($i \in 2..n$) uses $log_2(n)$ Boolean variables. In addition, temporary Boolean variables are used for carries in the encoding of $P_j = P_i + 1$ in constraint (5"). The number of clauses is dominated by constraint (5"), which requires $O(n \times log_2(n) \times d)$ clauses to encode, where d is the maximum degree in G.

4.3 LFSR Encoding

The LFSR encoding of the successor function also employs a log-encoded domain variable P_i for each vertex i ($i \in 1..n$) [21]. Given a binary number X, the Fibonacci LFSR[5] determines the next binary number Y by shifting the bits of X one position to left and computing the lowest bit of Y by applying xor on the *taps* bits of X. For a given length of n, the LFSR is able to generate all $2^n - 1$ non-zero numbers from any non-zero start number.

For example, consider the length $n = 4$ and the taps $\{2, 3\}$. Given a binary number $<X_3X_2X_1X_0>$, the next binary number $<Y_3Y_2Y_1Y_0>$ is calculated as follows: $Y_3 = X_2$, $Y_2 = X_1$, $Y_1 = X_0$, $Y_0 = X_2 \oplus X_3$. Assume the start number is 0001, the LFSR produces the following sequence:

$$0001 \rightarrow 0010 \rightarrow 0100 \rightarrow 1001 \rightarrow$$
$$0011 \rightarrow 0110 \rightarrow 1101 \rightarrow 1010 \rightarrow$$
$$0101 \rightarrow 1011 \rightarrow 0111 \rightarrow 1111 \rightarrow$$
$$1110 \rightarrow 1100 \rightarrow 1000 \rightarrow 0001$$

The LFSR encoding is more compact than the binary adder encoding. The LFSR encoding does not need any carry variables. For a number, in order to produce its successor the LFSR encoding uses two clauses for each bit except the lowest bit, for which it uses 4 clauses if the number of taps is 2, and 16 clauses if the number of taps is 4.

5 Preprocessing

The distance encoding treats the HCP as a single-agent path-finding problem. At time 1, the agent resides at vertex 1. In each step, the agent moves to a neighboring vertex. The agent cannot reach a vertex at time t ($t \in 2..n$) if there are no paths of length $t - 1$ from vertex 1 to the vertex. Similarly, the agent cannot occupy a vertex at time t ($t \in 2..n$) if there are no paths of length $n - t + 1$ from the vertex to vertex 1. This simple reasoning rules out impossible positions from consideration for vertices during preprocessing.

If the agent cannot occupy vertex i at time t, then vertex i cannot be mapped to position t. Under the unary encoding, the variable U_{it} is set to 0; under binary encoding, the value $s^{t-1}(1)$ is excluded from the domain of P_i.

[5] `https://en.wikipedia.org/wiki/Linear-feedback_shift_register#Fibonacci_LFSRs`

It is expensive to check if there is a path of a given length t from one vertex to another if t is large.[6] For long paths, we use the *shortest-distance heuristic*. For each vertex i ($i \in 2..n$), if the shortest distance from vertex 1 to vertex i is t, then the agent cannot occupy vertex i at times 1, 2, ..., t. Similarly, if the shortest distance from vertex i to vertex 1 is t, then the agent cannot occupy vertex i at times $n-t+2$, $n-t+3$, ..., n.

If the graph is undirected, and the start vertex 1 has exactly two neighbors,[7] then the agent must visit one of the neighbors at time 2, and visit the other neighbor at time n. This means that the agent cannot occupy either neighbor at times 3, 4, ... $n-1$. This idea can be seen as a special case of the Hall's theorem [15].

6 Experimental Results

We experimentally compared the three encodings of the successor function on the knight's tour problem and the Flinders challenge set, using the Maple SAT solver [25]. The knight's tour problem is a popular benchmark that has been utilized to evaluate solvers. The problem can be solved algorithmically in linear-time [6]. The Warnsdorff's rule, which always proceeds to the square from which the knight has the fewest onwards moves, is a very effective heuristic used in backtracking search. With Warnsdorff's rule, called *first-fail* principle in CP, and the reachability-checking capability during search, CP solvers can solve very large instances. Regarding SAT-based solvers, there are reports of successes in this problem using lazy approaches [1, 34], but no eager approaches have been reported to be able to solve instances of size 30×30 or larger. The Flinders challenge set contains instances with various graph structures, and is very comprehensive for evaluating HCP solvers.

Tables 1 and 2 compare the encodings on code size. For each encoding, we compared two settings, one with preprocessing (pp) and the other without preprocessing (no-pp). The results are roughly consistent with the theoretical analysis: The LFSR encoding (**lfsr**) generates the most compact code, then followed by the binary adder encoding (**adder**), and finally by the unary encoding (**unary**). Preprocessing has different levels of effectiveness in reducing code size. While preprocessing reduces the code size to half for **unary** and always reduces the code size to some extent for **adder**, it increases **lfsr**'s number of clauses for instance knight-20. Although both **adder** and **lfsr** use log encoding for position variables, preprocessing produces holes scattered in the domains for **lfsr**, and these domains sometimes require more prime implicants to cover than domains that have holes concentrated.

Table 3 compares the encodings on CPU time, which includes both the compile and solving times. The times were measured on Linux Ubuntu with an Intel

[6] Let M be the adjacency matrix of the graph. A naive algorithm that finds all paths of length t requires computing M^t.

[7] The knight's tour problem belongs to this case if one of the corner squares is chosen as vertex 1.

Table 1. A comparison of the encodings on code size (number of variables)

Benchmark	**adder**		**lfsr**		**unary**	
	pp	no-pp	pp	no-pp	pp	no-pp
knight-8	808	1,220	880	888	2,246	4,416
knight-10	1,928	1,981	1,407	1,409	5,259	12,497
knight-12	3,123	3,202	2,314	2,326	13,186	25,063
knight-14	4,261	4,364	3,110	3,120	23,446	45,045
knight-16	6,382	6,774	5,088	5,098	39,591	75,988
knight-18	7,752	7,921	5,739	5,749	61,075	118,799
knight-20	10,285	10,492	7,754	7,760	92,080	179,330
knight-22	12,781	13,026	9,660	9,670	133,078	259,864
knight-24	15,396	15,695	11,643	11,651	185,746	364,126

Table 2. A comparison of the encodings on code size (number of clauses)

Benchmark	**adder**		**lfsr**		**unary**	
	pp	no-pp	pp	no-pp	pp	no-pp
knight-8	10,531	14,527	9,220	9,540	37,909	146,460
knight-10	23,009	24,683	16,217	15,406	135,200	89,482
knight-12	40,828	43,547	40,241	41,187	96,382	192,806
knight-14	57,465	61,115	57,924	55,644	181,604	364,667
knight-16	84,026	100,212	75,561	78,633	320,668	639,929
knight-18	115,631	121,835	92,201	93,225	509,231	1,029,328
knight-20	151,630	159,013	124,927	111,979	788,490	1,591,404
knight-22	188,325	197,181	153,243	119,595	1,166,274	2,351,996
knight-24	239,456	251,113	222,159	237,725	1,659,575	3,348,802

i7 3.30GHz CPU and 32GHz RAM, and the time limit used was 20 minutes per instance. Preprocessing is generally effective in reducing the time. The results for **adder** are very interesting: when preprocessing was turned off, **adder** even failed to solve size 12×12; with preprocessing, however, it efficiently solved all of the instances. **adder** even succeeded in solving larger instances. For example, it solved size 30×30 in 1 minute, and size 36×36 in 34 minutes. There have been no reports of such successes by eager encoding approaches in the literature. It is also interesting to note that **lfsr** does not scale up as well as **adder**, although **lfsr** also uses log encoding for position variables, and preprocessing also reduces the domains of these variables.[8] With preprocessing, **unary** also solved all of the instances. However, the time taken for each instance is much longer than that taken by **adder**.

Table 4 shows the number of solved instances in the Flinders challenge set. All the encodings were tested with preprocessing enabled, and the time limit used was 10 minutes per instance. The 1001 instances were divided into five groups based on the numbers of vertices in the graphs. Except for the last group in which each graph has more than 4000 vertices, **adder** solved more instances than **lfsr** and **unary** in each of the groups.

[8] The experiment produced similar results when Lingeling (http://fmv.jku.at/lingeling/) was used as the SAT solver.

Table 3. A comparison of the encodings on CPU time (seconds)

Benchmark	**adder**		**lfsr**		**unary**	
	pp	no-pp	pp	no-pp	pp	no-pp
knight-8	2.19	2.59	2.28	2.51	5.80	7.59
knight-10	3.32	3.79	3.48	3.29	7.08	11.21
knight-12	4.60	>1200	6.48	17.59	9.62	37.24
knight-14	3.39	>1200	89.73	62.89	16.37	95.68
knight-16	6.92	>1200	18.14	33.66	79.03	185.33
knight-18	4.97	>1200	52.71	83.12	110.39	265.12
knight-20	9.37	>1200	90.36	145.63	133.25	422.83
knight-22	14.05	>1200	573.61	517.69	341.84	>1200
knight-24	24.75	>1200	>1200	>1200	1027.11	>1200

Table 4. A comparison of the encodings on the Flinders challenge set (solved instances)

n	**adder**	**lfsr**	**unary**
1..1000 (171)	156	152	119
1001..2000 (167)	137	95	13
2001..3000 (175)	75	20	0
3001..4000 (185)	7	2	0
> 4000 (303)	0	0	0
total (1001)	375	269	132

7 Related Work

Various approaches have been proposed for the HCP [13]. As the HCP is a special variant of the Traveling Salesman Problem (TSP), many approaches proposed for TSP [7, 14] can be tailored to the HCP.

Recently several studies have used SAT solvers for the HCP. A common technique utilized in encoding the HCP into SAT in order to prevent sub-cycles is to impose a strict ordering on the vertices. The *bijection* encoding [16] uses an *edge* constraint for each non-arc pair (i, j) that bans vertex j from immediately following vertex i in the ordering. This encoding is compact for dense graphs. The *relative* encoding [31] imposes transitivity on the ordering: if vertex i reaches vertex k, and vertex k reaches vertex j, then vertex i reaches vertex j. The *reachability* encoding, which is used in translating answer-set programs with loops into SAT [26], also imposes transitivity on the ordering. All these encodings use direct encoding for positions, and require $O(n^3)$ clauses in the worst case. It is reported in [37] that using a hierarchical encoding for domain variables significantly reduces the code size and increases the solving speed for HCP. However, hierarchical encoding still suffers from code explosion for large graphs.

The distance encoding for HCP is not new. It is based on the standard decomposer used in MiniZinc [29], which uses an order variable O_i for each vertex i, and ensures that if $V_i = j$ then $O_j = O_i + 1$. The idea of using order or position variables could be traced back to the integer programming formulation that uses dummy variables to prevent sub-cycles [28]. Johnson first came up with the idea of using the LFSR to encode the successor function [21]. His HCP solver, which employs the LFSR encoding and graph partition techniques, took the second place in the Flinders challenge.

The log encoding [18] resembles the binary representation of numbers used in computer hardware. Despite its compactness, log encoding is not popular due to its poor propagation strengths [23]. This work has shown for the first time that, with optimizations and preprocessing, the binary adder encoding of the successor function outperforms the unary and LFSR encodings for the HCP.

In order to circumvent the explosive code size of eager encoding approaches, researchers have proposed lazy approaches, such as satisfiability modulo acyclicity [1] and incremental SAT solving [34] for the HCP. The lazy approaches may be able to deal with large graphs. However, they limit the choice of SAT solvers.

8 Conclusion

A central issue in encoding the HCP into SAT is how to prevent sub-cycles, and one well-used technique is to map vertices to different positions. In this paper, we have compared three encodings for the successor function used in the distance encoding of the HCP, and proposed a preprocessing technique that rules out unreachable positions from consideration. Our study has surprisingly revealed that, with optimizations and preprocessing, the binary adder encoding outperforms the unary and the LFSR encodings. The binary encoding solved some instances of the knight's tour problem that had been beyond reach for eager encoding methods, and solved more instances of the Flinders HCP challenge set than other encodings.

An efficient SAT encoding for the HCP will expand the successes of SAT solvers in solving combinatorial problems. We plan to further improve the distance encoding for the HCP by exploiting special graph structures. We also plan to generalize the encoding for the TSP, of which the HCP is a special variant.

Acknowledgement

The author would like to Marijn Heule for pointing out Andrew Johnson's work on the LFSR encoding, and Andrew Johnson for helpful discussions. This work is supported in part by the NSF under the grant number CCF1618046.

References

1. Jori Bomanson, Martin Gebser, Tomi Janhunen, Benjamin Kaufmann, and Torsten Schaub. Answer set programming modulo acyclicity. In *Logic Programming and Nonmonotonic Reasoning (LPNMR)*, pages 143–150, 2015.
2. Lucas Bordeaux, Youssef Hamadi, and Lintao Zhang. Propositional satisfiability and constraint programming: A comparative survey. *ACM Comput. Surv.*, 38(4):1–54, 2006.
3. Robert King Brayton, Gary D. Hachtel, Curtis T. McMullen, and Alberto L. Sangiovanni-Vincentelli. *Logic Minimization Algorithms for VLSI Synthesis*. Kluwer Academic Publishers, 1984.

4. Gerhard Brewka, Thomas Eiter, and Miroslaw Truszczyński. Answer set programming at a glance. *Commun. ACM*, 54(12):92–103, 2011.
5. Jingchao Chen. A new SAT encoding of the at-most-one constraint. In *Proc. of the 9th Int. Workshop of Constraint Modeling and Reformulation*, 2010.
6. Axel Conrad, Tanja Hindrichs, Hussein Morsy, and Ingo Wegener. Solution of the knight's Hamiltonian path problem on chessboards. *Discrete Applied Mathematics*, 50(2):125–134, 1994.
7. William J. Cook. *In Pursuit of the Traveling Salesman: Mathematics at the Limits of Computation*. Princeton University Press, 2012.
8. Johan de Kleer. A comparison of ATMS and CSP techniques. In *IJCAI*, pages 290–296, 1989.
9. Jean-Louis Boulanger (Editor). *Formal Methods Applied to Industrial Complex Systems: Implementation of the B Method*. Wiley, 2014.
10. Michael R. Garey and David S. Johnson. *Computers and Intractability*. W.H. Freeman and Co., 1979.
11. Marco Gavanelli. The log-support encoding of CSP into SAT. In *CP*, pages 815–822, 2007.
12. Martin Gebser, Benjamin Kaufmann, André Neumann, and Torsten Schaub. Conflict-driven answer set solving. In *IJCAI*, pages 386–, 2007.
13. Ronald J. Gould. Recent advances on the Hamiltonian problem: Survey III. *Graphs and Combinatorics*, 30(1):1–46, 2014.
14. Gregory Gutin and Abraham P. Punnen. *The Traveling Salesman Problem and Its Variations (Combinatorial Optimization)*. Springer, 2007.
15. Philip Hall. Representatives of subsets. *J. London Math. Soc.*, 10(1):26–30, 1935.
16. Alexander Hertel, Philipp Hertel, and Alasdair Urquhart. Formalizing dangerous SAT encodings. In *Proceedings of SAT*, pages 159–172, 2007.
17. Jinbo Huang. Universal Booleanization of constraint models. In *CP*, pages 144–158, 2008.
18. Kazuo Iwama and Shuichi Miyazaki. SAT-variable complexity of hard combinatorial problems. In *IFIP Congress (1)*, pages 253–258, 1994.
19. Daniel Jackson. *Software Abstractions: Logic, Language, and Analysis*. MIT Press, 2012.
20. Peter Jeavons and Justyna Petke. Local consistency and SAT-solvers. *JAIR*, 43:329–351, 2012.
21. Andrew Johnson. Quasi-linear reduction of Hamiltonian cycle problem (HCP) to satisfiability problem (SAT), 2014. Disclosure Number IPCOM000237123D, IP.com, Fairport, NY, June 2014. Available at https://priorart.ip.com/IPCOM/000237123.
22. Henry A. Kautz and Bart Selman. Planning as satisfiability. In *ECAI*, pages 359–363, 1992.
23. Donald Knuth. *The Art of Computer Programming, Volume 4, Fascicle 6: Satisfiability*. Addison-Wesley, 2015.
24. Leslie Lamport. *Specifying Systems: The TLA+ Language and Tools for Hardware and Software Engineers*. Addison-Wesley, 2004.
25. Jia Hui Liang, Chanseok Oh, Vijay Ganesh, Krzysztof Czarnecki, and Pascal Poupart. MapleCOMSPS LRB VSIDS and MapleCOMSPS CHB VSIDS. In *Proceedings of SAT Competition 2017: Solver and Benchmark Descriptions*, pages 20–21, 2017.
26. Fangzhen Lin and Jicheng Zhao. On tight logic programs and yet another translation from normal logic programs to propositional logic. In *IJCAI*, pages 853–858, 2003.

27. Edward J. McCluskey. Minimization of boolean functions. *Bell System Technical Journal*, 35(6):1417–1444, 1956.
28. C. E. Miller, A. W. Tucker, and R. A. Zemlin. Integer programming formulation of traveling salesman problems. *J. ACM*, 7(4):326–329, 1960.
29. Nicholas Nethercote, Peter J. Stuckey, Ralph Becket, Sebastian Brand, Gregory J. Duck, and Guido Tack. MiniZinc: Towards a standard CP modelling language. In *CP*, pages 529–543, 2007.
30. Justyna Petke. *Bridging Constraint Satisfaction and Boolean Satisfiability*. Artificial Intelligence: Foundations, Theory, and Algorithms. Springer, 2015.
31. Steven David Prestwich. SAT problems with chains of dependent variables. *Discrete Applied Mathematics*, 130(2):329–350, 2003.
32. Willard Van Orman Quine. The problem of simplifying truth functions. *The American Mathematical Monthly*, 59(8):521–531, 1952.
33. Jussi Rintanen. Planning as satisfiability: Heuristics. *Artif. Intell.*, 193:45–86, 2012.
34. Takehide Soh, Daniel Le Berre, Stéphanie Roussel, Mutsunori Banbara, and Naoyuki Tamura. Incremental SAT-based method with native Boolean cardinality handling for the Hamiltonian cycle problem. In *Logics in Artificial Intelligence (JELIA)*, pages 684–693, 2014.
35. Mirko Stojadinovic and Filip Maric. meSAT: multiple encodings of CSP to SAT. *Constraints*, 19(4):380–403, 2014.
36. Naoyuki Tamura, Akiko Taga, Satoshi Kitagawa, and Mutsunori Banbara. Compiling finite linear CSP into SAT. *Constraints*, 14(2):254–272, 2009.
37. Miroslav N. Velev and Ping Gao. Efficient SAT techniques for absolute encoding of permutation problems: Application to Hamiltonian cycles. In *Eighth Symposium on Abstraction, Reformulation, and Approximation (SARA)*, 2009.
38. Neng-Fa Zhou and Håkan Kjellerstrand. Optimizing SAT encodings for arithmetic constraints. In *CP*, pages 671–686, 2017.