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 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. This paper compares three encodings for the successor function, namely, a *unary encoding* that uses a Boolean variable for each vertex-time pair, an optimized *binary adder encoding* that uses a special incrementor with no carry variables, and a *LFSR encoding* that uses a linear-feedback-shift register. This paper also proposes 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 optimization 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 approaches, and performed the best on the HCP instances used in the 2019 XCSP competition.

1 Introduction

The Hamiltonian Cycle Problem (HCP) is one of the most well-known NP-complete problems. Given a graph, the goal of HCP is to find a cycle in the graph that includes each and every vertex exactly once. As 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 [2, 27], planning [24, 37], answer set programming [6, 14], to general constraint satisfaction problems (CSPs) [4, 20, 22, 33, 39, 40, 43]. The key issue in encoding HCP into SAT is how to prevent sub-cycles. A naive encoding, which bans sub-cycles in every proper subset of vertices, requires an exponential number of clauses. One common technique used in SAT encodings for 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 to large graphs [19, 28, 35]. In order to circumvent the explosive encoding size of the eager approach, researchers have proposed lazy approaches, such as satisfiability modulo acyclicity [3] that incorporates reachability checking during search, and incremental SAT solving that incrementally adds clauses to ban sub-cycles [38]. Recently inspired by the log encoding [21], Johnson proposed a compact encoding for HCP, which employs a linear-feedback-shift register (LFSR) for the successor function [23, 18].

This paper continues the pursuit of an efficient SAT encoding for HCP. 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 $n+1$ comes back at the start vertex after having visited each and every vertex exactly once. 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. This encoding is called *distance encoding*. This paper compares three encodings for the successor function, namely, a *unary encoding* that uses a Boolean variable for each vertex-time pair, an optimized *binary adder encoding* that uses a special incrementor with no carry variables, and a *LFSR encoding* that uses a linear-feedback-shift register. This paper also proposes 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, using two SAT solvers, show that, with preprocessing, the optimized binary adder encoding significantly outperformed 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 approaches, and solved more instances of the HCP benchmark used in the 2019 XCSP competition than other solvers.

2 Preliminaries

This section defines HCP, and gives the basic SAT encodings for domain variables and the at-most-one constraint that are employed in the SAT encodings for HCP.

2.1 The HCP and the `circuit` Constraint

Given a directed graph, the goal of HCP is to find a cycle in the graph that includes each and every vertex exactly once. In CP, HCP can be described as a global constraint `circuit(`G`)`, where G `= [`$V_1, V_2, \ldots, V_n$`]` is a list of domain variables representing the graph.

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`. A valuation $V_i = j$ of the domain variables represents a subgraph of G that consists

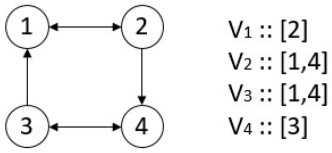

Fig. 1. A directed graph and its representation using domain variables

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.

2.2 Direct Encoding

Let $X :: \{a_1, a_2, \ldots, a_n\}$ be a domain variable. The *direct encoding* [11] 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 $\geq_1 (B_1, B_2, \ldots, B_n)$, and an *at-most-one* constraint $\leq_1 (B_1, B_2, \ldots, B_n)$. The at-least-one constraint is encoded as the clause $B_1 \vee B_2 \vee \ldots \vee B_n$.

2.3 SAT Encodings for the At-Most-One Constraint

The at-most-one constraint has numerous encodings into SAT (see [42] for the latest comparison).

The ***pairwise*** (PW) encoding for $\leq_1 (B_1, B_2, \ldots, B_n)$ decomposes the constraint into $\neg B_i \vee \neg B_j$, for $i \in 1..n-1$ and $j \in i+1..n$. PW generates $O(n^2)$ clauses, and is therefore not viable for large n.

The ***bisect*** (BS) encoding for $\leq_1 (B_1, B_2, \ldots, B_n)$ splits the variables into two groups $G_1 = \{B_1, B_2, \ldots, B_m\}$ and $G_2 = \{B_{m+1}, \ldots, B_n\}$, where $m = \lfloor \frac{n}{2} \rfloor$. It introduces a new variable T as the *commander* for G_1, and uses $\neg T$ as the commander for G_2. BS decomposes the constraint into the following: **(BS-1)** For $i \in 1..m$: $B_i \Rightarrow T$; **(BS-2)** For $i \in m+1..n$: $B_i \Rightarrow \neg T$; **(BS-3)** $\leq_1 (B_1, B_2, \ldots, B_m)$; **(BS-4)** $\leq_1 (B_{m+1}, \ldots, B_n)$. Constraint **BS-1** forces T to be 1 if any of the variables in G_1 is 1. Constraint **BS-2** forces T to be 0 if any of the variables in G_2 is 1. Since T cannot be both 0 and 1 at the same time, it is impossible for one variable in G_1 and another variable in G_2 to be 1 simultaneously. Constraints **BS-3** and **BS-4** recursively enforce at-most-one on the two groups. The BS is a special case of the *commander encoding*[25]. The number of clauses generated by BS is characterized by $f(n) = n + 2f(n/2)$, and the number of new variables is characterized by $g(n) = 1 + 2g(n/2)$.

The ***product*** (PD) encoding [7] for $\leq_1 (B_1, B_2, \ldots, B_n)$ arranges the variables on an $m \times m$ matrix M, where $m = \sqrt{n}$. It introduces two vectors of new variables $\langle R_1, R_2, \ldots, R_m \rangle$ and $\langle C_1, C_2, \ldots, C_m \rangle$, where R_i represents row i and C_j represents column j. In case n is not a square number, the extra entries of M are filled with 0. PD decomposes the constraint into the following: **(PD-1)** For $i \in 1..m$, $j \in 1..m$: $M_{ij} \Rightarrow R_i \wedge C_j$; **(PD-2)** $\leq_1 (R_1, R_2, \ldots, R_m)$; **(PD-3)** $\leq_1 (C_1, C_2, \ldots, C_m)$. The number of clauses generated by PD is characterized by $f(n) = 2n + 2f(\sqrt{n})$, and the number of new variables is characterized by $g(n) = 2\sqrt{n} + 2g(\sqrt{n})$.

A hybrid encoding for $\leq_1 (B_1, B_2, \ldots, B_n)$ is employed for HCP. When $n \leq 4$, PW is used. Otherwise, the constraint is divided into smaller at-most-one constraints using BS or PD, depending on the cost function $f(n) + \alpha g(n)$, where α is the penalty for introducing a new variable. For example, for $n = 64$ and $\alpha = 3$, the constraint is first divided using PD into two sub-constraints, each of which involves 8 variables, and then the sub-constraints are divided using BS into base ones, which are encoded using PW.

2.4 Log Encoding and Logic Optimization

The *log encoding* [21] 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 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 [29, 36] 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 [12]. The Espresso logic optimizer [5] only computes a partial set of prime implicants based on heuristics, and therefore a smaller set-covering problem.

For example, consider the domain variable $X :: [1, 2, 5, 6]$. The log encoding uses a sequence of three Boolean variables, $X_2X_1X_0$, to encode the domain. It is possible to represent 8 different values with three Boolean variables, including the values in X's domain and the no-good values in the set $\{0, 3, 4, 7\}$. A naive encoding with *conflict clauses* [13] for the domain requires four clauses:

$$X_2 \vee X_1 \vee X_0 \qquad (X \neq 0)$$

$$X_2 \vee \neg X_1 \vee \neg X_0 \qquad (X \neq 3)$$
$$\neg X_2 \vee X_1 \vee X_0 \qquad (X \neq 4)$$
$$\neg X_2 \vee \neg X_1 \vee \neg X_0 \qquad (X \neq 7)$$

Each of these clauses corresponds to a no-good value. The logic optimizer Espresso only uses two clauses:

$$X_1 \vee X_0$$
$$\neg X_1 \vee \neg X_0$$

Each clause corresponds to a prime implicant in the disjunctive normal form. Note that the variable X_2 is optimized away.

3 The Distance Encoding for the `circuit` Constraint

The `circuit(`G`)` constraint, where G is a list of domain variables $[V_1, V_2, \ldots, V_n]$, 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 to 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,[1] and $s^k(p)$ be the kth successor of p. Assume that vertex 1 is visited first, and it is mapped to position 1.[2] The following constraints ensure that the cycle starts at 1 and ends at 1:

[1] The successor function, such as the LFSR described below, may generate a different sequence of numbers from the natural number sequence.

[2] A good heuristic is to start with a vertex that has the smallest degree [41].

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 theorem shows that it is sufficient to use one-way entailment constraints in (3) - (5) instead of stronger equivalence constraints.

The final encoding size depends on how the successor function is encoded. The code size of constraints (1) and (2) 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 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 for 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 the following constraint must hold:

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

For each vertex i ($i \in 1..n$), there is exactly one 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:

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

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 log encoding:

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

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

Let X's log encoding be $\langle X_{m-1}X_{m-2}\ldots X_1X_0\rangle$ and Y's log encoding be $\langle Y_{m-1}Y_{m-2}\ldots Y_1Y_0\rangle$. Consider the unsigned 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 using ripple carry adders from the least significant bit to the most significant bit. If a half-adder is used for each bit position, then the addition requires, in total, $m-1$ carry variables and $7 \times m$ clauses.

The following sequential incrementor performs the addition using no carry variables:[3]

- For bit position 0, $Y_0 = \neg X_0$, which is encoded as two clauses: $Y_0 \vee X_0$ and $\neg Y_0 \vee \neg X_0$.
- For bit position 1, the input carry from bit position 0 is X_0, so the following constraints must hold:

  $$X_0 \Rightarrow Y_1 = \neg X_1$$
  $$\neg X_0 \Rightarrow Y_1 = X_1$$

  These two constraints are converted into 4 clauses.
- For each other bit position i ($i > 1$), the carry from bit position $i-1$ is 1 iff $Y_{i-1} = 0$ and $X_{i-1} = 1$, so the following constraints must hold:

  $$\neg Y_{i-1} \wedge X_{i-1} \Rightarrow Y_i = \neg X_i$$
  $$\text{otherwise} \Rightarrow Y_i = X_i$$

  These constraints can be encoded using 6 clauses.

The total number of clauses used for the addition is $2 + 4 + (m-2) * 6$.

The sequential incrementor is improved as follows. For bit position i ($i > 1$), instead of considering one bit at a time, the improved version considers two bits at a time, imposing the following constraints:

$$\neg Y_{i-1} \wedge X_{i-1} \Rightarrow Y_i = \neg X_i$$
$$\neg Y_{i-1} \wedge X_{i-1} \wedge X_i \Rightarrow Y_{i+1} = \neg X_{i+1}$$
$$\text{otherwise} \Rightarrow Y_i = X_i \wedge Y_{i+1} = X_{i+1}$$

These constraints can be encoded using 11 clauses, resulting in a reduction of one clause for each two bits.

Furthermore, the improved incrementor treats the top 4 bits as a whole using the adder in Figure 2.[4] The carry from bit position $m-5$ to bit position $m-4$ is 1 iff $Y_{m-5} = 0$ and $X_{m-5} = 1$. The adder uses 21 clauses, resulting in a reduction of 3 clauses from 24 clauses needed by the one-bit incrementor and 1 clause from 22 clauses needed by the two-bit incrementor.

[3] This encoding is based one suggested by Vitaly Lagoon via personal communication.

[4] One may wonder why the chosen number is 4, not 3 or 5. Interestingly, a choice of 3, 5, or any other number will increase the overall number of clauses for Espresso.

$X_{m-1} \vee X_{m-4} \vee \neg Y_{m-1}$
$X_{m-1} \vee \neg Y_{m-1} \vee \neg Y_{m-2}$
$X_{m-1} \vee \neg Y_{m-1} \vee \neg Y_{m-3}$
$X_{m-1} \vee \neg Y_{m-1} \vee \neg Y_{m-4}$
$X_{m-3} \vee \neg Y_{m-3} \vee \neg Y_{m-4}$
$\neg X_{m-4} \vee \neg Y_{m-5} \vee Y_{m-4}$
$X_{m-2} \vee \neg X_{m-3} \vee Y_{m-2} \vee Y_{m-3}$
$X_{m-4} \vee X_{m-5} \vee \neg Y_{m-4}$
$\neg X_{m-1} \vee Y_{m-1}$
$\neg X_{m-2} \vee \neg X_{m-3} \vee \neg Y_{m-2} \vee Y_{m-3}$
$\neg X_{m-3} \vee \neg X_{m-4} \vee \neg X_{m-5} \vee Y_{m-5} \vee \neg Y_{m-3}$
$X_{m-2} \vee X_{m-4} \vee \neg Y_{m-2}$
$X_{m-3} \vee X_{m-4} \vee \neg Y_{m-3}$
$X_{m-2} \vee \neg Y_{m-2} \vee \neg Y_{m-3}$
$X_{m-2} \vee \neg Y_{m-2} \vee \neg Y_{m-4}$
$\neg X_{m-4} \vee X_{m-5} \vee Y_{m-4}$
$X_{m-1} \vee \neg X_{m-2} \vee Y_{m-1} \vee Y_{m-2}$
$X_{m-4} \vee \neg X_{m-5} \vee Y_{m-5} \vee Y_{m-4}$
$X_{m-4} \vee \neg Y_{m-5} \vee \neg Y_{m-4}$
$\neg X_{m-1} \vee \neg X_{m-2} \vee \neg Y_{m-1} \vee Y_{m-2}$
$X_{m-3} \vee \neg X_{m-4} \vee \neg X_{m-5} \vee Y_{m-5} \vee Y_{m-3}$

Fig. 2. 4-bit adder $\langle X_{m-1}X_{m-2}X_{m-3}X_{m-4}\rangle + (\neg Y_{m-5} \wedge X_{m-5}) = \langle Y_{m-1}Y_{m-2}Y_{m-3}Y_{m-4}\rangle$

Under log encoding, each of the position variables P_i ($i \in 2..n$) uses $log_2(n)$ Boolean variables. 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 ($i \in 1..n$) [23]. Given a binary number X, the Fibonacci LFSR 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 $\langle X_3X_2X_1X_0\rangle$, the next binary number $\langle Y_3Y_2Y_1Y_0\rangle$ 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 use any carry variables either. 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 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, since the agent must be back at vertex 1 at time $n+1$, 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.

It is expensive to check if there is a path of a given length t from one vertex to another if t is large.[5] For long paths, the *shortest-distance heuristic* is used. 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,[6] 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 [17].

6 Experimental Results

All the encodings described above have been implemented in the Picat compiler.[7] An experiment was conducted to compare the three encodings for the successor function on the knight's tour problem and the HCP benchmark used in the 2019 XCSP solver competition[8], using the SAT solver MapleLCMDiscChronoBT-DL-v3, the winner of the main track of the 2019 SAT Race[9]. In order to show how the SAT solutions perform in a broader context, the experiment also included **or-tools** (version 7.6)[10], **clingon** (version 5.4.0)[11], and an incremental SAT-based approach in the comparison. For **or-tools**, a lazy clause generation CP solver, HCP is encoded as as a `circuit` constraint, and the first-fail strategy is utilized to label variables. For **clingon**, an answer-set programming system that performs search-time reachability testing [3], there are several encodings for HCP. The following ASP encoding, which has been shown to perform the best, was used in the experiment:

```
{hpath(X,Y) : link(X,Y)} = 1 :- node(X).
```

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

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

[7] picat-lang.org

[8] http://xcsp.org/competition

[9] http://sat-race-2019.ciirc.cvut.cz/

[10] https://developers.google.com/optimization

[11] https://potassco.org/

```
{hpath(X,Y) : link(X,Y)} = 1 :- node(Y).

reach(1).

reach(Y) :- reach(X),hpath(X,Y).

:- not reach(X),node(X).
```

The incremental approach treats HCP as an assignment problem. If a solution does not contain any sub-cycles, then the solution is returned as a valid solution to the original HCP. If the solution contains a sub-cycle that includes a set of vertices S, then the clause $\texttt{sum}(H_{i \in S j \notin S}) > 1$ is added into the encoding to ban the sub-cycle. The incremental approach uses the same SAT solver, and restarts from scratch after each new sub-cycle elimination clause is added.

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 [8]. The Warnsdorff's rule [34], 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 are able to solve very large instances. Regarding SAT-based solvers, no eager approaches have been reported to be able to solve instances of size 30 or larger. The HCP benchmark used in the XCSP competition contains 10 instances selected from the Flinders challenge set[12] with numbers of vertices ranging from 338 to 1584.

All the CPU times reported below were measured on Linux Ubuntu with an Intel i7 3.30GHz CPU and 32G RAM. The time limit used was 40 minutes per instance.

Tables 1 and 2 compare the encodings on, respectively, the number of variables and the number of clauses. For each encoding, results from two separate settings are included, one with preprocessing (pp) and the other with no 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**). When preprocessing is excluded, **adder** and **lfsr** use the same number of variables because both of them use log encoding for position variables. When preprocessing is included, however, **adder** uses slightly fewer variables than **lfsr**. This is because preprocessing produces holes scattered in the domains for **lfsr**, sometimes requiring more prime implicants to cover than the original domains, while preprocessing narrows bounds of the domains or produces holes concentrated in the domains for **adder**, and Picat is able to fix some of the bits at translation time.

Table 3 compares the encodings on CPU time, which includes both the translation and solving times. The column **inc** gives the time taken by the incremental approach. The entry *MO* indicates out-of-memory. Preprocessing is generally

[12] http://fhcp.edu.au/fhcpcs

Table 1. A comparison on number of variables

Benchmark	**adder**		**lfsr**		**unary**	
	pp	no-pp	pp	no-pp	pp	no-pp
knight-8	820	920	912	920	2,958	5,582
knight-10	1,428	1,481	1,479	1,481	6,883	12,767
knight-12	2,375	2,454	2,442	2,454	13,596	25,477
knight-14	3,217	3,320	3,310	3,320	24,032	46,415
knight-16	4,994	5,386	5,376	5,386	40,385	77,806
knight-18	5,972	6,141	6,131	6,141	62,109	121,129
knight-20	8,065	8,272	8,266	8,272	94,974	182,236
knight-22	10,073	10,318	10,308	10,318	136,612	263,410
knight-24	12,152	12,451	12,443	12,451	189,984	376,426

Table 2. A comparison on number of clauses

Benchmark	**adder**		**lfsr**		**unary**	
	pp	no-pp	pp	no-pp	pp	no-pp
knight-8	11,161	13,939	8,964	9,284	17,271	33,623
knight-10	22,453	23,535	15,641	14,831	42,802	85,836
knight-12	39,784	41,647	39,214	40,163	92,130	184,775
knight-14	55,846	58,272	56,326	54,044	173,041	345,322
knight-16	81,672	96,232	73,231	76,329	306,414	604,730
knight-18	112,465	116,527	89,033	90,088	486,798	970,960
knight-20	147,513	152,184	120,849	107,883	746,299	1,499,129
knight-22	183,121	188,641	148,018	114,412	1,101,488	2,213,986
knight-24	237,071	244,714	215,516	231,325	1,563,133	3,102,627

effective in reducing the time. The results of **adder** are very interesting: when preprocessing was turned off, **adder** even failed to solve size 12; with preprocessing, however, it efficiently solved all of the instances. 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 uses slightly fewer clauses. One explanation, as shown in Table 1, could be that preprocessing helps **adder** more than **lfsr** in reducing the number of variables. The **inc** is generally not competitive; it ran out of time on two instances, and ran out of memory on another two instances. The **inc** keeps track of all the sub-cycles that have been found, and adds clauses to ban them in subsequent searches. The result indicates that **inc** is not feasible when there are a huge number of sub-cycles in the graph. The solvers **or-tools** and **clingo** are very fast on these instances; **or-tools** solved all in less than 2 seconds each, and **clingo** solved all in less than 1 second each.

Table 4 gives several knight's tour instances solved by the binary adder encoding. These instances are easy for **or-tools** and **clingon** to solve, but had been out of reach for eager SAT encoding approaches.

Table 3. A comparison on CPU time (seconds)

Benchmark	**adder**		**lfsr**		**unary**		**inc**
	pp	no-pp	pp	no-pp	pp	no-pp	
knight-8	2.88	2.39	2.93	3.4	8.92	11.26	0.17
knight-10	2.39	113.18	3.67	3.83	11.76	18.31	0.47
knight-12	4.46	>2400	11.12	81.33	22.15	42.51	5.16
knight-14	4.59	>2400	13.31	126.52	48.96	89.43	88.63
knight-16	7.83	>2400	30.16	52.50	92.61	225.47	>2400
knight-18	10.16	>2400	35.85	153.99	137.91	436.13	*MO*
knight-20	9.39	>2400	505.40	>2400	208.53	512.10	625.25
knight-22	25.84	>2400	1243.22	>2400	358.51	>2400	>2400
knight-24	9.64	>2400	>2400	>2400	>2400	>2400	*MO*

Table 4. Knight's tour instances solved by the binary adder encoding (seconds)

size	26	28	30	32	34	36	38
time	57.45	83.04	62.90	346.57	188.00	310.09	304.13

Table 5 compares the solvers on CPU time using the XCSP competition instances.[13] The number in parentheses indicates the number of vertices in the graph. Preprocessing was enabled for the eager encoding approaches. Overall, **adder** performed the best. It solved all the instances, none of which took more than 250 seconds. The **lfsr** failed on one instance. For the solved instances, the times taken by **lfsr** are much longer than those taken by **adder**. The **unary** failed on 6, and **inc** failed on 7 instances, due to time out or memory out. While **or-tools** and **clingo** demonstrated superior performance on the knight's tour benchmark, they are not as competitive as **adder** on these instances; **or-tools** failed on 7 instances, and **clingo** failed on 1 instance and took more than 500 seconds to solve two of the instances each.

For comparison, the same experiment was also conducted using CaDiCaL, the second-place winner in the 2019 SAT Race. With preprocessing, **adder** using CaDiCaL also solved all the instances in Tables 3 and 5, while **lfsr** failed on knight-24 and graph48, and **unary** failed on 4 knight's tour instances and 6 XCSP instances.

[13] All the participating solvers in the 2019 XSCP competition, except PicatSAT and **Choco**, failed on every single instance. PicatSAT, which is based on an early version of **adder**, solved all of the 10 instances, the sequential version of **Choco** solved 4 instances, and the parallel version of **Choco** solved 7 instances.

Table 5. XCSP competition instances (CPU time)

benchmark	adder	lfsr	unary	inc	or-tools	clingo
graph162 (909)	184.18	676.91	*MO*	21.98	49.62	16.94
graph171 (996)	27.25	51.58	1887.64	>2400	>2400	52.50
graph197 (1188)	64.18	270.74	>2400	>2400	>2400	1821.98
graph223 (1386)	68.16	144.20	1279.99	>2400	>2400	17.92
graph237 (1476)	91.6	388.25	>2400	>2400	>2400	23.78
graph249 (1558)	52.88	112.60	494.45	336.21	307.30	13.47
graph252 (1572)	98.22	403.25	>2400	>2400	>2400	541.48
graph254 (1582)	63.65	268.23	541.97	168.32	>2400	12.7
graph255 (1584)	45.86	144.83	>2400	>2400	171.0	>2400
graph48 (338)	213.0	>2400	>2400	*MO*	>2400	0.97

7 Related Work

Various approaches have been proposed for HCP [15]. As HCP is a special variant of the Traveling Salesman Problem (TSP), many approaches proposed for TSP [9, 16] can be tailored to HCP.

Recently several studies have used SAT solvers for HCP. A common technique utilized in encoding HCP into SAT in order to prevent sub-cycles is to impose a strict ordering on the vertices. The *bijection* encoding [19] 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 [35] 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 [28], 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 [41] that using a hierarchical encoding for domain variables significantly reduces the encoding 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 [31], 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 [30].

The log encoding [21] resembles the binary representation of numbers used in computer hardware. Despite its compactness, log encoding is not popular due to its poor propagation strengths [26]. Johnson first came up with the idea of using log encoding for position variables and the LFSR for encoding the successor function [23]. The binary adder encoding for $Y = X + 1$ proposed in this paper is a special optimized incrementor that does not use any carry variables.

The preprocessing technique for excluding unreachable positions from the domains of position variables is well-used in constraint programming. Similar

techniques have been used for maintaining consistency of some of the global constraints, such as the *regular* constraint [32], and for eliminating variables in multi-agent path finding [1]. This work has shown, for the first time, that when preprocessing is effective the binary adder encoding of the successor function significantly outperforms the unary and LFSR encodings for HCP.

In order to circumvent the explosive encoding sizes of eager approaches, researchers have proposed lazy approaches, such as satisfiability modulo acyclicity [3] and incremental SAT solving [38] for HCP. The idea to incrementally add constraints to avoid code explosion is the pillar of the cutting-plane method [9, 10]. The incremental approach may suffer if the problems require repeated addition of sub-cycle elimination clauses.

8 Conclusion

A central issue in encoding HCP into SAT is how to prevent sub-cycles, and one well-used technique is to map vertices to different positions. This paper has compared three encodings for the successor function used in the distance encoding of HCP, and proposed a preprocessing technique that rules out unreachable positions from consideration. Our study has surprisingly revealed that, with preprocessing and optimization, the binary adder encoding outperforms the unary and the LFSR encodings. While no eager SAT encoding approaches have been reported to be able to solve size 30 or larger of the knight's tour problem, the binary adder encoding, using the SAT solver MapleLCMDiscChronoBT-DL-v3, succeeded in solving all instances up to size 38 in less than 6 minutes each. This is a remarkable advancement of the state of the art. While there is still a long way to go for eager SAT encoding approaches to be competitive with CP and ASP solvers on the knight's tour problem, this paper has showed that the binary adder encoding is competitive with the best CP and ASP solvers on the HCP benchmark used in the 2019 XCSP competition.

An efficient SAT encoding for HCP will expand the successes of SAT solvers in solving combinatorial problems, such as the travelling salesman problem (TSP), which is a generalization of HCP, and its variants. Further improvements include exploiting special graph structures and symmetry-breaking techniques in SAT encodings.

Acknowledgement

The author would like to thank Håkan Kjellerstrand for helping test and tune Picat's SAT compiler, Marijn Heule for pointing out Andrew Johnson's work on the LFSR encoding, Andrew Johnson for clarifications on his LFSR encoding, and the anonymous reviewers for helpful comments. This work is supported in part by the NSF under the grant number CCF1618046.

References

1. Roman Barták, Neng-Fa Zhou, Roni Stern, Eli Boyarski, and Pavel Surynek. Modeling and solving the multi-agent pathfinding problem in Picat. In *29th IEEE International Conference on Tools with Artificial Intelligence*, pages 959–966, 2017.
2. Armin Biere, Marijn Heule, Hans van Maaren, and Toby Walsh. *Handbook of Satisfiability*. IOS Press, 2009.
3. 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.
4. Lucas Bordeaux, Youssef Hamadi, and Lintao Zhang. Propositional satisfiability and constraint programming: A comparative survey. *ACM Comput. Surv.*, 38(4):1–54, 2006.
5. Robert King Brayton, Gary D. Hachtel, Curtis T. McMullen, and Alberto L. Sangiovanni-Vincentelli. *Logic Minimization Algorithms for VLSI Synthesis*. Kluwer Academic Publishers, 1984.
6. Gerhard Brewka, Thomas Eiter, and Miroslaw Truszczyński. Answer set programming at a glance. *Commun. ACM*, 54(12):92–103, 2011.
7. Jingchao Chen. A new SAT encoding of the at-most-one constraint. In *Proc. of the Int. Workshop of Constraint Modeling and Reformulation*, 2010.
8. 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.
9. William J. Cook. *In Pursuit of the Traveling Salesman: Mathematics at the Limits of Computation*. Princeton University Press, 2012.
10. G. B. Dantzig, R. Fulkerson, and S. Johnson. Solution of a large-scale traveling-salesman problem. *Operations Research*, 2:393–410, 1954.
11. Johan de Kleer. A comparison of ATMS and CSP techniques. In *IJCAI*, pages 290–296, 1989.
12. Michael R. Garey and David S. Johnson. *Computers and Intractability*. W.H. Freeman and Co., 1979.
13. Marco Gavanelli. The log-support encoding of CSP into SAT. In *CP*, pages 815–822, 2007.
14. Martin Gebser, Benjamin Kaufmann, André Neumann, and Torsten Schaub. Conflict-driven answer set solving. In *IJCAI*, pages 386–, 2007.
15. Ronald J. Gould. Recent advances on the Hamiltonian problem: Survey III. *Graphs and Combinatorics*, 30(1):1–46, 2014.
16. Gregory Gutin and Abraham P. Punnen. *The Traveling Salesman Problem and Its Variations (Combinatorial Optimization)*. Springer, 2007.
17. Philip Hall. Representatives of subsets. *J. London Math. Soc.*, 10(1):26–30, 1935.
18. Michael Haythorpe and Andrew Johnson. Change ringing and Hamiltonian cycles: The search for erin and stedman triples. *EJGTA*, 7(1):61–75, 2019.
19. Alexander Hertel, Philipp Hertel, and Alasdair Urquhart. Formalizing dangerous SAT encodings. In *Proceedings of SAT*, pages 159–172, 2007.
20. Jinbo Huang. Universal Booleanization of constraint models. In *CP*, pages 144–158, 2008.
21. Kazuo Iwama and Shuichi Miyazaki. SAT-variable complexity of hard combinatorial problems. In *IFIP Congress (1)*, pages 253–258, 1994.
22. Peter Jeavons and Justyna Petke. Local consistency and SAT-solvers. *JAIR*, 43:329–351, 2012.

23. 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.
24. Henry A. Kautz and Bart Selman. Planning as satisfiability. In *ECAI*, pages 359–363, 1992.
25. Will Klieber and Gihwon Kwon. Efficient CNF encoding for selecting 1 from n objects. In *the Fourth Workshop on Constraints in Formal Verification(CFV)*, 2007.
26. Donald Knuth. *The Art of Computer Programming, Volume 4, Fascicle 6: Satisfiability*. Addison-Wesley, 2015.
27. Leslie Lamport. *Specifying Systems: The TLA+ Language and Tools for Hardware and Software Engineers*. Addison-Wesley, 2004.
28. 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.
29. Edward J. McCluskey. Minimization of Boolean functions. *Bell System Technical Journal*, 35(6):1417V–1444, 1956.
30. C. E. Miller, A. W. Tucker, and R. A. Zemlin. Integer programming formulation of traveling salesman problems. *J. ACM*, 7(4):326–329, 1960.
31. 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.
32. Gilles Pesant. A regular language membership constraint for finite sequences of variables. In *CP*, pages 482–495, 2004.
33. Justyna Petke. *Bridging Constraint Satisfaction and Boolean Satisfiability*. Artificial Intelligence: Foundations, Theory, and Algorithms. Springer, 2015.
34. Ira Pohl. A method for finding Hamilton paths and knight's tours. *Commun. ACM*, 10:446–449, 1967.
35. Steven David Prestwich. SAT problems with chains of dependent variables. *Discrete Applied Mathematics*, 130(2):329–350, 2003.
36. Willard Van Orman Quine. The problem of simplifying truth functions. *The American Mathematical Monthly*, 59(8):521V–531, 1952.
37. Jussi Rintanen. Planning as satisfiability: Heuristics. *Artif. Intell.*, 193:45–86, 2012.
38. 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.
39. Mirko Stojadinovic and Filip Maric. meSAT: multiple encodings of CSP to SAT. *Constraints*, 19(4):380–403, 2014.
40. Naoyuki Tamura, Akiko Taga, Satoshi Kitagawa, and Mutsunori Banbara. Compiling finite linear CSP into SAT. *Constraints*, 14(2):254–272, 2009.
41. 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.
42. Neng-Fa Zhou. Yet another comparison of SAT encodings for the at-most-k constraint. *ArXiv*, abs/2005.06274, 2020.
43. Neng-Fa Zhou and Håkan Kjellerstrand. Optimizing SAT encodings for arithmetic constraints. In *CP*, pages 671–686, 2017.