

In Search of the Smallest Vocabulary Set in a Language Using Formal Methods

Po-Chun Wu
B11901054
b11901054@ntu.edu.tw

Yu-Chia Kuo
B11901047
b11901047@ntu.edu.tw

Abstract—This paper addresses the Smallest Vocabulary Set (SVS) problem. We prove SVS is NP-hard via reduction to Boolean satisfiability. We evaluate several formal methods, with MaxSAT and Bounded Model Checking (BMC) showing the best results. Heuristic algorithms we propose offer near-optimal solutions much faster, especially when combined with MaxSAT. Experiments on a large dictionary graph show that optimal sets cover about 40% of total words, and our hybrid method achieves close-to-optimal results with significantly reduced runtime.

Our Presentation Video:

https://www.youtube.com/watch?v=rbsxcR_iScY

Index Terms—SAT, BMC, BDD, TSS, Heuristic, Ric

I. INTRODUCTION

Learning a new language has always been a headache for many of us, especially memorizing the thousands of vocabulary that are required to communicate in a smooth and confident way. Today, a foreign language is often learned by translation, but doing so sacrifices the sense of language that native speakers develop by learning their mother tongue using the language itself. Hence, a problem emerges: When is it possible for people to learn a new language without translation? Or, more specifically, how many words should we initially know to learn the rest of the vocabulary in the language using a monolingual dictionary?

A. The Smallest Vocabulary Set Problem

To answer the question above, we can model the monolingual dictionary as a directed graph, with n vertices $V = \{v_0, v_1, \dots, v_{n-1}\}$ being the set of words, and e directed edges $E = \{e_0, e_1, \dots, e_{e-1}\}$ defined as

$$E = \{(v_i, v_j) | v_i \text{ is used in the definition of } v_j\} \quad (1)$$

. For example, if there is an entry *Cat: An animal of the family Felidae*, then $\{(an, cat), (animal, cat), (of, cat), (the, cat), (family, cat), (felidae, cat)\}$ is a subset of E .

Assuming no guesses, to learn a new word, we must understand all the words in the definition of that word. Hence, the Smallest Vocabulary Set Problem is defined as follows:

Smallest Vocabulary Set Problem (SVS)

For a directed graph $G = (V, E)$, find the smallest set $S_0 \subset V$ such that all vertices in V can be partitioned into layers $S_0, S_1, S_2, \dots, S_k$ satisfying the following conditions:

- S_0 is the initial set of words that can be learned without any prerequisites.
- For each layer S_i where $i > 0$, every word $v \in S_i$ has all of its direct predecessors in $R_{i-1} = \bigcup_{j=0}^{i-1} S_j$.

That is, each word in a given layer can be defined entirely using words from earlier layers.

B. Difficulty of the SVS Problem

The SVS Problem is a special case of the other commonly-discussed problem in marketing and epidemiology: the Target Set Selection (TSS) problem. The TSS problem was first introduced by Richardson and Domingos [2] and later refined by Kempe et al. [3], as discussed in [1], and it is described as follows:

Target Set Selection Problem (TSS)

Given a graph $G = (V, E)$, each vertex $v \in V$ is associated with a threshold $\text{thr}(v) \in \mathbb{N}$. The goal is to find the smallest subset $S \subseteq V$ (called the *target set*) such that activating S initially results in activating all vertices in G (or at least ℓ vertices, in some variants), according to the following rules:

- All vertices in S are activated at step $t = 0$.
- In each subsequent step $t \geq 1$, a vertex $v \in V$ becomes activated if at least $\text{thr}(v)$ of its neighbors are already activated.

In the case of the SVS problem, the $\text{thr}(v)$ function is valued as *all predecessors of v* .

The difficulty of this problem was proved in Kempe et al. [2] as NP-hard. We will later see in that it can be split into steps that are NP-complete and can be reduced to the Boolean Satisfiability problem in polynomial time.

C. Dataset

To study the SVS problem, we used the Free Dictionary API [4] of the English language to generate the graph. The modeled graph has $|V| = 7753$ vertices and $|E| = 27612$ directed edges (removed self-edges). For simplification, we used the subgraphs G_n to run our algorithms, where $G_n = (V_n, E_n)$, $V_n \subset V, |V_n| = n, E_n = \{(v_i, v_j) | (v_i \in V_n) \wedge (v_j \in V_n) \wedge ((v_i, v_j) \in E)\}$.

II. ALGORITHMIC APPROACH

Before using the SAT solver to solve this problem, we tried to find the best possible way to use algorithms to solve it. First, we aimed for an algorithm that can verify that a solution is true. Next, we used this algorithm to search for the optimal solution.

A. Solution Verification Algorithm

Inspired by the Ford-Fulkerson algorithm for finding the maximum flow of a graph, we thought of an approach that deletes edges along the breadth-first search iterations. The algorithm is as follows:

Algorithm 1 Implication Algorithm

- 1) For any graph G_n , we are given $S_0 \subset V_n$.
 - 2) For every $v_i \in S_0$, remove all incoming edges.
 - 3) For $k = 0$ to $n - 1$:
 - a) For every $v_i \in S_k$, remove all outgoing edges.
 - b) Add all the vertices that doesn't have incoming edges to S_{k+1}
 - c) Stop when every vertex is included in $R_{k+1} = \bigcup_{j=0}^{k+1} S_j$ and return *Valid Solution*.
 - 4) If there are still vertices that are not in R_n , return *Invalid Solution*
-

This algorithm is called the *Implication Algorithm* in our program because, in each iteration, we add all the vertices that can be implied completely by the previous vertices in R_k to S_{k+1} .

The time complexity of the algorithm is $\Theta(n + e)$ since it traverses every vertex and every edge exactly once. Hence, the SVS problem is of the nondeterministic polynomial class (NP).

B. Optimization Algorithm

Having the *Implication Algorithm* able to verify the answer in linear time, the most straightforward way to solve this problem is to call it for every combination of $S_0 \subset V_n$. The algorithm is as follows:

Algorithm 2 Kuo's Algorithm

- 1) Find all vertices without predecessors and put them in S_0 .
 - 2) For $k = 1$ to $n - 1$:
 - a) Generate all combinations of S_0 with size k .
 - b) Compute *Implication*(S_0), if true, return S_0
 - 3) If no solutions are found, return V_n .
-

Since this algorithm runs through each combination of n vertices and each iteration has a time complexity of $\Theta(n + e)$, the overall time complexity is $O(n \cdot 2^n)$, assuming $e = c \cdot n$ for some finite c . This algorithm performs quite badly, but since the similar TSS problem is proved by [3] as NP-Hard, we stopped looking for better algorithms from scratch and instead aimed for the formal approach.

III. FORMAL APPROACH: BOUNDED MODEL CHECKING

The SVS problem is an optimization problem, which can be solved by iteratively solving the following sub-problem for $k = 1$ to n :

Vocabulary Set Satisfiability Problem (VS-SAT)

For a directed graph $G = (V, E)$, find a set $S_0 \subset V$ such that $|S_0| = k$ for some integer $k \leq n$, and all vertices in V can be partitioned into layers $S_0, S_1, S_2, \dots, S_k$ satisfying the following conditions:

- S_0 is the initial set of words that can be learned without any prerequisites.
- For each layer S_i where $i > 0$, every word $v \in S_i$ has all of its direct predecessors in $R_{i-1} = \bigcup_{j=0}^{i-1} S_j$.

If no such S_0 can be found, the problem is UNSAT.

Hence, if we can find a way to solve the VS-SAT problem, the SVS problem would be solved by one of the following procedures:

Algorithm 3 Incremental Algorithm

For $k = 1$ to n :

- 1) if not VS-SAT(k): Continue.
 - 2) else: Return S_0 found by VS-SAT.
-

Algorithm 4 Decremental Algorithm

For $k = n$ to 1 :

- 1) if VS-SAT(k): Save the S_0 found and continue.
 - 2) else: Return the saved S_0 from the previous iteration.
-

A. Reduction of VS-SAT to the Boolean Satisfiability Problem

To solve VS-SAT, we can use an encoding to reduce the problem to the clauses that form the conjunctive normal form Boolean satisfiability problem (CNF-SAT). Since CNF-SAT can be regarded as a Boolean representation of a circuit, we can also find similarities in the VS-SAT structure that resemble states and transition relations in the circuit.

1) *States*: For any graph $G_n = (V_n, E_n)$, every subset S of V_n can be represented as a Boolean state $X = x_{n-1}x_{n-2}\dots x_2x_1x_0$, where $x_i = v_i \in S$.

2) *Initial State*: Since we are finding the S_0 that eventually implies V_n , we cannot set the initial state to S_0 because we don't know what S_0 is. However, we know the final state, which is V_n ; hence, we can reverse the set expansion process as follows: "Starting from V_n , remove all the states that can be implied by the other vertices one by one. Eventually, there would be no states that could be removed and S_0 would be made up of the remaining states". Therefore, the initial state is defined as $S^0 = V_n$. In the form of Boolean states, $X^0 = 11...111$,

3) *time frame*: In circuits, each clock cycle defines a time frame, and reachable states start from the initial state and eventually expand to a fixed point. In this problem, since the *initial state* is $X^0 = 11...111$, the state at time t , $X^t = x_{n-1}^t x_{n-2}^t \dots x_2^t x_1^t x_0^t$ is defined as *the set of vertices remaining after removing some vertices in the previous time frames*.

4) *4. State Transitions*: The state transition clauses are the rules to remove vertices from a state X^t to form the next state X^{t+1} . The high-level concept of the clauses is as follows: *For a vertex to be in the current state, it should be in the next state, or all its predecessors are in the next state*. Therefore, the transition clauses are:

$$\forall i, x_i^t \leftrightarrow x_i^{t+1} \vee \left(\bigwedge_{x_j \in \text{pred}(x_i)} x_j^{t+1} \right) \quad (2)$$

, where $\text{pred}(x_i)$ denotes all the predecessors of v_i representing x_i in the graph.

5) *Fixed point*: A fixed point reached means that for the graph at time t , the next state can only be generated by every vertex itself being in the current state and in the next state. In the boolean language, it means $\forall i, x_i^t \leftrightarrow x_i^{t+1}$ is the only way to generate the next state. In terms of the graph, there are no vertices that can be removed and S_0 is found.

6) *Assumptions*: For each state transition, if the fixed point has not yet been reached, there exists at least one vertex that can be removed, that is, $\exists i, x_i^t \leftrightarrow \bigwedge_{x_j \in \text{pred}(x_i)} x_j^{t+1}$. Therefore, as t goes from some t_0 to $t_0 + 1$, the set size constraint k can be decremented from k_0 to $k_0 - 1$. Since in the initial state, $t = 0$ and $k = n$, by induction, for each t , $k = n - t$. Henceforth, for each iteration of solving VS-SAT at time t , we add the assumption $|X^t| \leq k = n - t$ to the SAT solver, where the symbol $|X^t|$ refers to the number of 1s in X^t .

In conclusion, the SVS problem can be reduced to a sequential circuit SAT problem, and solving the VS-SAT problem is equivalent to solving the combinational SAT problem of that circuit.

Reduced VS-SAT(k) Problem

Check if the following Boolean formula is satisfiable. If

it is satisfiable, return a solution.

Initial State

$$\bigwedge_{i=0}^n x_i^0 \quad (3)$$

time frame Expansion to $t = n - k$

$$\bigwedge_{t=0}^{n-k} \bigwedge_{i=0}^n x_i^t \leftrightarrow x_i^{t+1} \vee \left(\bigwedge_{x_j \in \text{pred}(x_i)} x_j^{t+1} \right) \quad (4)$$

Assumption at $t = n - k$

$$\sum_{i=0}^n x_i^{n-k} \leq k \quad (5)$$

The symbol \sum indicates algebraic sum over Boolean variables x_i^{n-k} .

B. Bounded Model Checking

To solve the SVS problem, it is required to iteratively solve the VS-SAT problem from $k = n$ to $k = 1$. However, from the second set of clauses (time frame expansion) in the reduced VS-SAT problem, it is apparent that as k decreases ($n - k$ increases), the same clauses would be built and solved again and again. Hence, rather than solving VS-SAT by iteration over k , we can use the bounded model checking procedure to reduce the time it takes to build the same clauses over and over again. The algorithm is as follows:

Algorithm 5 SVS Algorithm Using Bounded Model Checking

- 1) Build the clauses for X^0 : $\bigwedge_{i=0}^n x_i^0$
- 2) For $t = 0$ to n :
 - a) Build the clauses for $X^t \rightarrow X^{t+1}$:

$$x_i^t \leftrightarrow x_i^{t+1} \vee \left(\bigwedge_{x_j \in \text{pred}(x_i)} x_j^{t+1} \right) \quad (6)$$

- b) Build the clauses for the assumption for $k = n - t$:

$$\sum_{i=0}^n x_i^t \leq n - t \quad (7)$$

- c) Solve the CNF with the current clauses. If UNSAT, return the last saved solution.
- d) If SAT, save the solution and release the previous assumption.

The clauses that represent the initial state are the variables themselves, so the number of clauses is n and the number of variables is also n . The transition clauses are implemented using the Tesitin transformation, so

$$x_i^t \leftrightarrow x_i^{t+1} \vee \left(\bigwedge_{x_j \in \text{pred}(x_i)} x_j^{t+1} \right) \quad (8)$$

is transformed into

$$\bigwedge_{x_j \in \text{pred}(x_i)} (\bar{x}_i^{t-1} \vee x_i^t \vee x_j^t) \quad (9)$$

$$\wedge (\bar{x}_i^t \vee x_i^{t-1}) \wedge \left(x_i^{t-1} \vee \bigvee_{x_j \in \text{pred}(x_i)} x_j^{t-1} \right) \quad (10)$$

. Thus, for each incrementation of t , there are $3n + e$ clauses generated with a total of $3n + 4e$ literals. The total number of clauses and literals generated for a solution of size k is $n + (n - k) \cdot (3n + e)$ and $n + (n - k) \cdot (3n + 4e)$, respectively. In conclusion, without taking account of the assumption clauses, the number of clauses grows linearly for each step. Since clauses are generated by traversal of the graph, the time complexity of reducing one time frame of the SVS problem to the SAT problem is $O(n + e) = O(n)$, assuming $e \leq c \cdot n$ for some fixed and finite integer c .

C. Refining the Assumption Clauses

Having known that the initial states and the state transitions can be transformed into CNF in $O(n)$ time, the next question is how to generate the clauses for the assumption $\sum_{i=0}^n x_i^t \leq n - t$?

The most intuitive way is by encoding all the possible cases of X^t such that the requirement is met. However, all combinations of $X^t \leq n - t$ are exponential to n , and they significantly increase the number of generated clauses compared to the linear behavior of transition clauses.

The solution we resorted to is *Sequential Counter Encoding* by Sinz, 2005 [5]. The encoding uses a counter circuit structure to generate the signal for $X^t \leq n - t$ and uses auxiliary variables $s_{i,j}$ to represent the partial sum of the counter. The encoding is as follows (Sinz, 2005):

Case $k = n - t = 1$

$$(\bar{x}_1^t \vee s_{1,1}) \wedge (\bar{x}_n^t \vee \bar{s}_{n-1,1}) \wedge \bigwedge_{1 < i < n} ((\bar{x}_i^t \vee s_{i,1}) \wedge (\bar{s}_{i-1,1} \vee s_{i,1}) \wedge (\bar{x}_i^t \vee s_{i-1,1})) \quad (11)$$

Case $k = n - t \neq 1$

$$(\bar{x}_1^t \vee s_{1,1}) \wedge (\bar{x}_n^t \vee \bar{s}_{n-1,n-t}) \wedge \left(\bigwedge_{1 < j \leq n-t} \bar{s}_{1,j} \right) \wedge \bigwedge_{1 < i < n} \left[(\bar{x}_i^t \vee s_{i,1}) \wedge (\bar{s}_{i-1,1} \vee s_{i,1}) \wedge (\bar{x}_i^t \vee \bar{s}_{i-1,n-t}) \wedge \bigwedge_{1 < j \leq n-t} ((\bar{x}_i^t \vee s_{i-1,j-1} \vee s_{i,j}) \wedge (\bar{s}_{i-1,j} \vee s_{i,j})) \right]. \quad (12)$$

From [5], the $n - t = 1$ case requires $3n - 4$ clauses, while the $n - t \neq 1$ case requires $2n^2 - 2nt - 2n + 3t - 1$ clauses,

Performance of C++ and Python Implementations

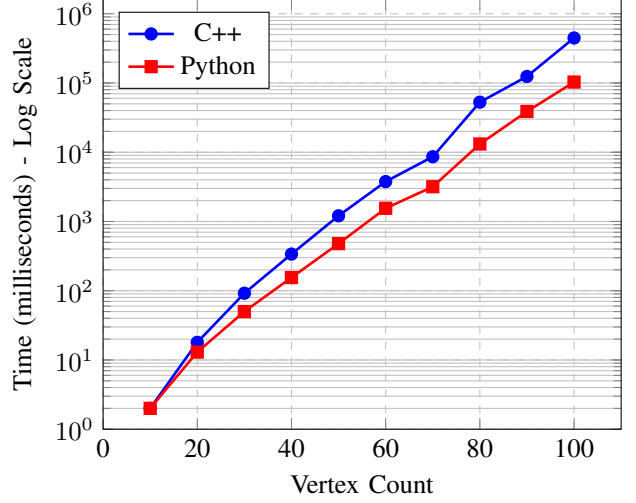


Fig. 1. Log-scale plot comparing the performance of C++ and Python implementations across different vertex counts.

with $(n - 1)(n - t)$ auxiliary variables for encoding. Since the generation of clauses are independent of the graph, the time complexity of generating the assumption clauses is $O(n^2)$.

D. NP-Hardness

From the previous discussions, the initial and transition clauses in each time frame can be generated in $O(n)$ time, and the assumption clauses can be generated in $O(n^2)$ time, so the overall time complexity of reducing the SVS problem at one time frame (VS-SAT) to the Boolean satisfiability problem (CNF-SAT) in polynomial time. Since CNF-SAT is a known NP-Complete problem, this demonstrates that VS-SAT is NP-Complete. Since the SVS problem is an optimization problem whose decision version (VS-SAT) is NP-Complete, it follows directly that the SVS problem is NP-Hard.

E. Implementation

Our implementation of algorithms to solve the SVS problem includes some algorithms using C++ and others using Python. The C++ algorithms are used to adapt to existing gv data structures and algorithms, as well as the BDD approach, which will be discussed later. The Python algorithms are used to test out experimental approaches faster and can be run on on-line workbench. To compare the run-time of C++ and Python algorithms, they both contain the same bounded model checking (BMC) approach. By scaling the run-time of the Python algorithms by the constant $\frac{T_{C++,BMC}}{T_{Python,BMC}}$, other algorithms exclusive to Python can be compared with those exclusive to C++.

The details of the execution of this program can be found in the Appendix.

F. Execution Results

Figure 1 shows the run time of the C++-based and the Python-based BMC algorithms. The run time is exponential to

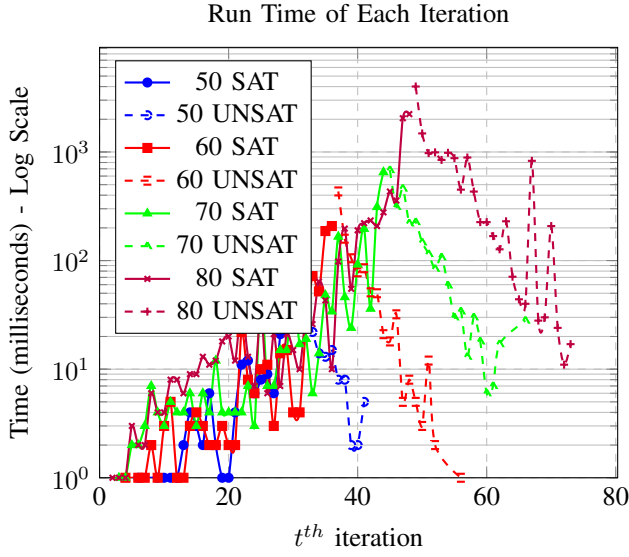


Fig. 2. Run time of each iteration for different problem sizes (50, 60, 70, and 80) on a logarithmic Y-axis.

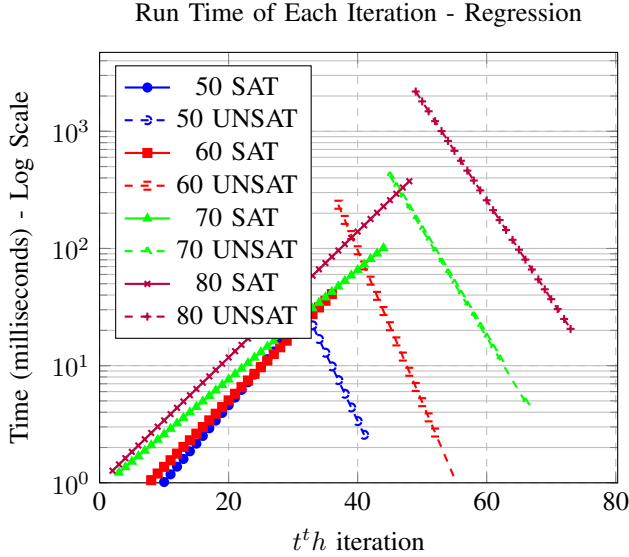


Fig. 3. Run time of each iteration applied linear regression in log scale.

the number of vertices n , which is as expected. The exponential regressions of the two curves are: $T_{C++}(n) = 1.24e^{0.131n}$ and $T_{Python}(n) = 1.16e^{0.116n}$.

TABLE I
RUN TIME COMPARISON

n	SAT cases	UNSAT cases
50	215.0	210.0
60	678.0	1068.0
70	1907.0	2594.0
80	7043.0	14213.0

Figure 2 shows the time spent solving the CNF-SAT problem for every iteration of t , and figure 3 is the linear regression

TABLE II
SPECIAL CASES OF RUN TIME

n	min SAT run time	max UNSAT run time	$\frac{T_{2cases}}{\text{total run time}}$
50	112	59	0.8
60	208	433	0.7
70	651	681	0.6
80	2233	4008	0.59

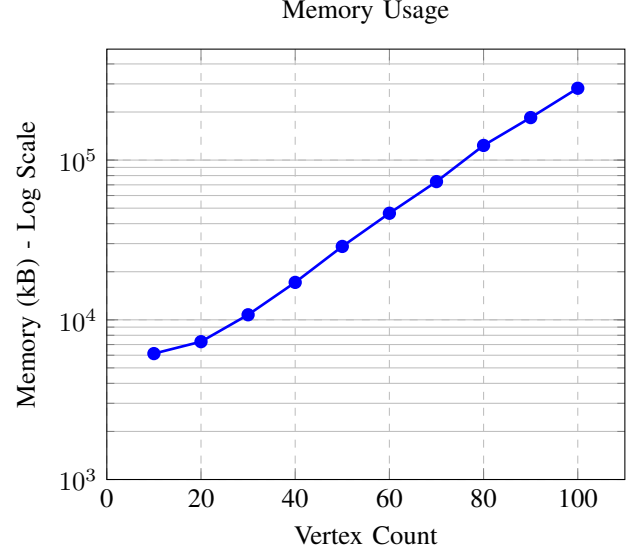


Fig. 4. Log-scale plot comparing the memory usage of different number of vertices.

on the log scale plot of figure 2. Table I is the summation of the run time of the previous data. Table II shows the two special cases of t : the iteration solving for the optimal solution (min SAT) and the iteration right after the optimal solution (max UNSAT). It also shows the percentage of the two cases in all values of t . From the data provided above, we can make the following observations.

- 1) The solving process becomes exponentially more time consuming when $k = n - t$ reaches the optimal solution of S_0 .
- 2) The cases that use up the most time are the min SAT case and the max UNSAT case. Their sum accounts for over 50% of the run time, and even more if we only consider iteration over one side.
- 3) The solution is closer to 1 than n , but the total time it takes from 1 to the optimal is less than that from n to the optimal. Hence, the procedure of increasing t runs faster than that of decreasing t .
- 4) For a fixed t , when n is larger, each SAT iteration does not consume significantly more time, but each UNSAT iteration increases in run time.

Figure 4 shows the memory usage of different numbers of vertices on a logarithmic scale. The straight line infers that despite the linear and quadratic size of literals and clauses, the solver still needs an exponential amount of memory.

In order to use the BMC approach to solve the complete

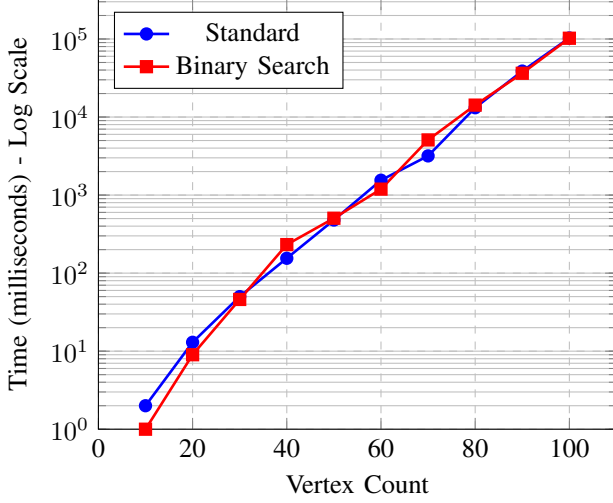


Fig. 5. Log-scale plot comparing the performance of standard BMC and binary search implementations across different vertex counts. Solver: Python MiniSAT.

$n = 7753$ English dictionary, 10^{560} millisecond-CPU's are required along a memory size of 10^{585} KB, which is impossible to obtain. Currently, using the BMC approach, our local computers can run a maximum of $n = 140$, which is far from the destination. Therefore, in the next sections, we will discuss other methods that we have tried to improve the performance of solving the SVS problem.

IV. OTHER FORMAL APPROACHES

A. Binary Search BMC

Our previous approach to solving the SVS problem is to run through $t = 1$ to the optimal $t = n - k + 1$ to the minimum t that leads to UNSAT. Since the problem is a search problem to find k such that $VS - SAT(k) = SAT$ and $VS - SAT(k - 1) = UNSAT$, it can be implemented using binary search instead.

From figure 5 it is apparent that the binary search approach does not make much difference for the large cases. The main reason is because, in the previous section, we have seen the exponential growth of run time of the cases near the optimal solution, and the binary search algorithm skips the cases far away from the optimal solution and refines them near the optimal one. Therefore, the binary search approach does not speed up the process of searching for the optimal solution.

B. Interpolation-Based Unbounded Model Checking (ItpUBMC)

As we have seen in the BMC case, the UNSAT cases often run much slower than the SAT cases, one part is due to the need to build a lot more time frames $t > n - k_{optimal}$ than the SAT cases, creating a lot more variables than the SAT cases, and thus reversely iterating from $t = n$ to $t = 1$ would not be faster than forward iteration. Therefore, we resort to

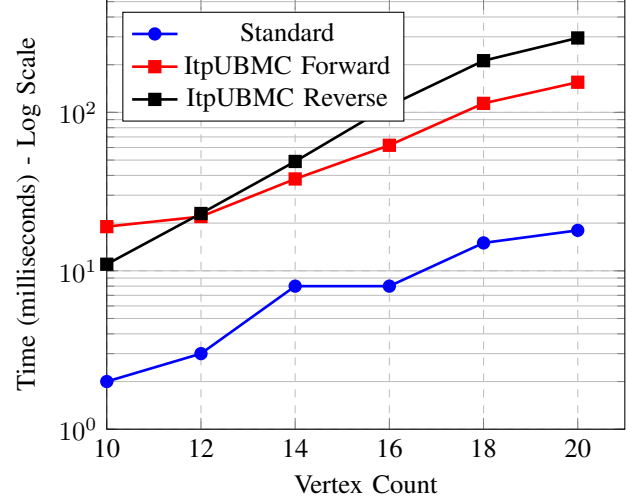


Fig. 6. Log-scale plot comparing the performance of standard BMC and ItpUBMC implementations across different vertex counts. Solver: C++ MiniSAT.

the unbounded model checking approach. Since it might find UNSAT fixed points before the time frame is expanded to $t = n - l_{optimal}$ reverse iteration using UBMC might possibly beat the forward iteration using BMC.

From figure 6, it is apparent that the UBMC algorithm is not performing well.

Consider the forward iteration approach. In the SAT region, it is trying to find a solution such that $VS - SAT(n - t) = SAT$. However, it has to expand all the time frames to reach the solution, which is similar to the BMC case. Moreover, the algorithm encourages finding fixed points before expanding the time frame and thus would spend a lot of time on finding fixed points when it should expand the time frame. Hence, the forward iteration approach is significantly slower than the BMC approach.

Consider the backward iteration approach. In the UNSAT region, it is trying to find a fixed point. However, from the run-time logs, it is seen that for every UNSAT case, the algorithm reaches the preset bound without ever successfully finding a fixed point. Thus, it uses up an enormous time finding interpolations and calls the SAT solver again and again without any useful progress.

C. Property-Directed Reachability (PDR)

time frame expansion is expensive, both in time and memory. The reason why the forward UBMC approach failed is also because it requires time frame expansion. Hence, we tried to use the PDR algorithm to avoid time frame expansion.

From Figure 7, it is apparent that the PDR algorithm behaves very poorly. One reason is because it requires a complete rerun of the algorithm for every t , compared to the BMC approach, which only needs to add one set of transition clauses and continue solving. Another reason is similar to the

Performance of Standard and PDR Implementations

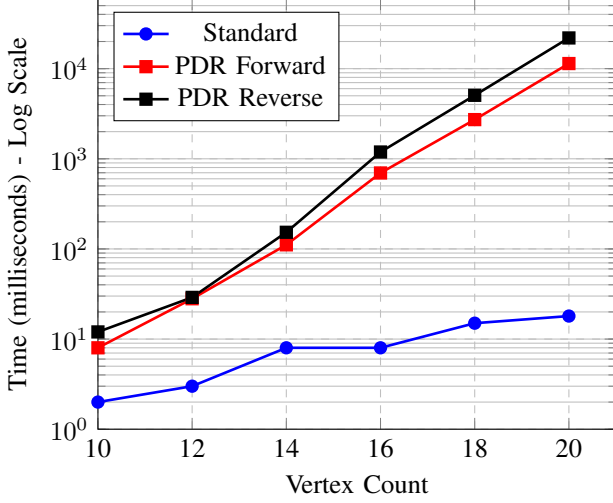


Fig. 7. Log-scale plot comparing the performance of standard BMC and PDR implementations across different vertex counts. Solver: C++ MiniSAT.

Performance of Standard and BDD Implementations

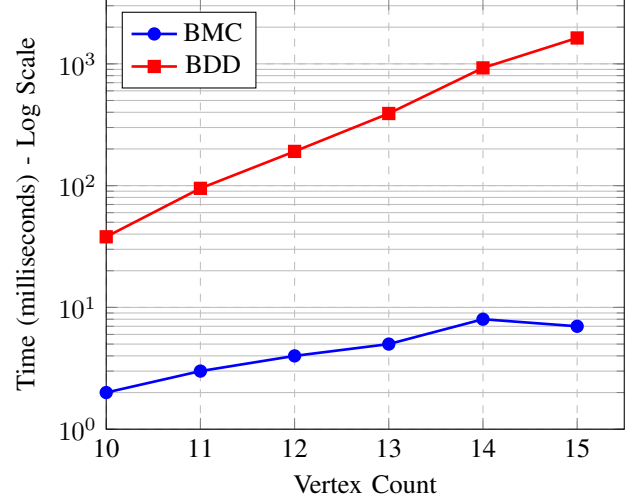


Fig. 9. Log-scale plot comparing the performance of standard BMC BDD implementations across different vertex counts. Solver: C++ MiniSAT.

Memory Usage of BMC, ItpUBMC, and PDR

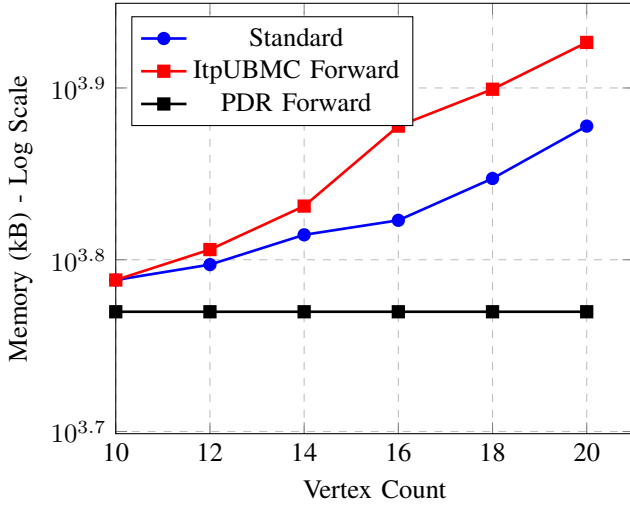


Fig. 8. Log-scale plot comparing the memory usage of BMC, ItpUBMC, and PDR implementations across different vertex counts. Solver: C++ MiniSAT.

Memory Usage of BMC and BDD

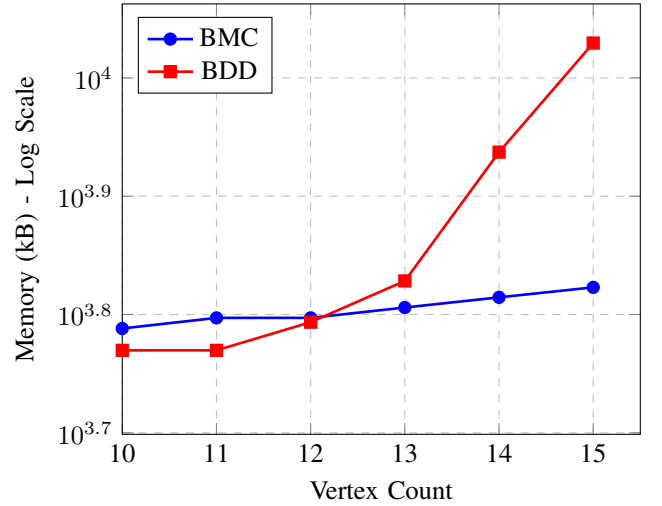


Fig. 10. Log-scale plot comparing the memory usage of BMC and BDD implementations across different vertex counts. Solver: C++ MiniSAT.

ItpUBMC case, that it does not find fixed points before the preset bound in the UNSAT cases. However, in Figure 8, we can see the advantages of the PDR algorithm, which is its memory usage is constant compared to the exponential growth of the BMC and ItpUBMC algorithms because it does not require time frame expansion. Hence, it has the potential to solve larger graphs if the algorithm is improved in the future.

D. Binary Decision Diagram (BDD)

The approaches we discussed above are SAT-based formal approaches. Since in the BMC, ItpUBMC, and PDR approaches, BMC is currently the best-performing algorithm; therefore, we tried to push the time frame expansion approach

to the limit, which is using BDD to build all solutions at all time frames at once.

From Figures 9 and 10 we can see that BDD performs poorly in both time and memory, and the execution of $n = 15$ is almost the limit our local computers can reach. Since the BDD approach is widely regarded as "inferior to SAT" in the modern day of large SoCs, we will not discuss how to improve the BDD variable order to achieve a better result.

E. MaxSAT

This is the fastest approach to find the minimum initial set size.

Time Performance Comparison: BMC vs MaxSAT

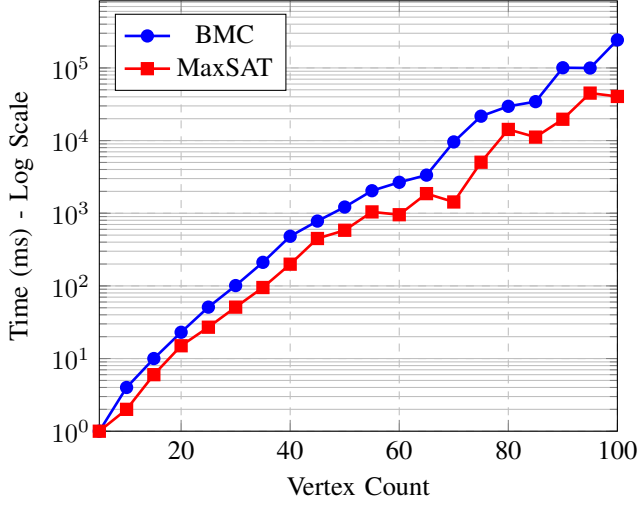


Fig. 11. Log-scale plot comparing time performance of Binary Search BMC and MaxSAT across different problem sizes.

MaxSAT Method Flow:

- 1) **Problem Formulation:** The MaxSAT approach reformulates the dictionary learning problem as a **Weighted Conjunctive Normal Form Satisfiability Problem (WCNF-SAT)**, which consists of:
 - **Hard Constraints:** Mandatory logical conditions that must be satisfied
 - **Soft Constraints:** Optimization objectives that are preferred but can be violated
- 2) **Variable Declaration:**
 - x_i : Boolean variable indicating whether word i is in the initial set.
 - r_i^t : Boolean variable indicating whether word i has been learned by time t .
 - $aux_and_i_t$: Auxiliary variables for Tseitin transformation of complex logic
- 3) **Hard Constraints Encoding:** The method uses the same three core constraints as Sequential SAT:
 - **Constraint 1 - Initial State:** $r_i^0 \leftrightarrow x_i$
 - **Learning Rule:** $r_i^t \leftrightarrow r_i^{t-1} \vee \bigwedge_{r_i \in \text{predecessors}} r_i^{t-1}$
 - **Constraint 3 - Completion Condition:** All words must be learned by maximum time step
- 4) **Soft Constraints Encoding (Optimization Objective)**
To minimize the initial set size, soft constraints are added for each non-forced initial word: `wcnf.append($x_i, 1$)`. For this constraint, the solver will prefer each $x_i = \text{False}$.

As shown in Figure 11, the MaxSAT solver achieve shorter time than binary search BMC method because of the non-iterative solver.

V. HEURISTIC APPROACHES

A. Heuristic Algorithmic Approximation Approach

Since the methods mentioned in the FORMAL APPROACH section are time consuming, we propose alternative heuristic strategies to achieve solutions significantly faster while maintaining practical effectiveness. These approaches prioritize scalability and runtime efficiency over exhaustive exploration.

1) **Heuristic Algorithm by degree ratio:** By our observation, the in/out degree of vertices play important roles in deciding the initial states. That is, the one who has much more out-degree than in-degree tends to be initial state. So we define the ratio $\frac{\text{in-degree}}{\text{out-degree}}$ and use it for later computation. The algorithm is as follows:

Algorithm 6 Heuristic Algorithm by degree ratio

- 1) For all vertices, calculate their in-degree and out-degree, and thus their degree ratio.
 - 2) Sort the vertices with their degree ratio.
 - 3) Add a vertex with the lowest degree ratio in S_0 iteratively until the current S_0 can pass the implication algorithm.
-

2) **Heuristic Algorithm by graph properties:** the first method only considers *in/out degree* of the vertices. This results in a larger initial set S_0 , so we find another method with considering more graph metrics such as *pagerank*, *betweenness centrality*, *in-closeness/out-closeness*. The algorithm is as follows:

Algorithm 7 Heuristic Algorithm by more graph metrics

- 1) Calculate graph metrics taht include *in/out degree*, *pagerank*, *betweenness centrality*, *in/out closeness*.
 - 2) Calculate the weighted score of the metrics.
 - 3) Add a vertex with the highest weighted score until the current S_0 can pass the implication algorithm.
-

3) **Heuristic Algorithm by using features from statistics analysis:** There are many graph metrics that can be taken into consideration. But the previous methods only choose the ones intuitively. And for this method, we use correlation, mean difference, and feature importance to find a better order to add the vertices to S_0 .

The algorithm is as follows:

Algorithm 8 Heuristic Algorithm by using features from statistics analysis

- 1) Analyze the feature importance by using correlation, mean difference, machine learning.(Shown in Table III, IV, V)
 - 2) Calculate score of each vertex, whose weight is dependent on the statistics analysis from *step 1*.
 - 3) Add a vertex with the highest weighted score until the current S_0 can pass the implication algorithm.
-

Comparison of these heuristic approaches

TABLE III
CORRELATION ANALYSIS

Feature	Pearson	P-val	Spearman	P-val
is_sink	0.353	0.000	0.353	0.000
closeness_out	0.353	0.000	0.374	0.000
eigenvector	0.319	0.000	0.377	0.000
in_degree	0.300	0.000	0.346	0.000
neighbor_in_count	0.300	0.000	0.346	0.000
total_degree	0.275	0.000	0.278	0.000
two_hop_in	0.262	0.000	0.252	0.000
betweenness	0.254	0.000	0.243	0.000
neighbor_total_count	0.240	0.000	0.242	0.000
pagerank	0.220	0.000	0.373	0.000
closeness_in	-0.187	0.000	0.017	0.572
degree_ratio	-0.187	0.000	-0.193	0.000
two_hop_total	0.078	0.011	0.097	0.002
is_source	-0.063	0.042	-0.063	0.042
clustering	-0.040	0.190	-0.062	0.045

TABLE IV
TOP FEATURES BY DIFFERENCE IN MEANS

Feature	S0 Mean	Non-S0 Mean	Difference
two_hop_in	47.402	36.137	11.265
total_degree	27.848	20.561	7.287
in_degree	15.633	9.105	6.529
neighbor_in_count	15.633	9.105	6.529
neighbor_total_count	25.470	19.613	5.857
two_hop_total	60.815	57.299	3.516
degree_ratio	1.189	2.065	-0.876
out_degree	12.215	11.457	0.758
neighbor_out_count	12.215	11.457	0.758
two_hop_out	40.397	40.957	-0.559
is_sink	0.194	0.000	0.194
closeness_out	0.483	0.413	0.070
closeness_in	0.406	0.469	-0.063
eigenvector	0.138	0.078	0.060
is_hub	0.708	0.744	-0.036

We’ve compared the solution size and time of different approach. These two figures and one table(Figures 12 and 13, Table VI, VII) show the comparison of solution size, execution time, and their summary.

As shown in the tables, the first heuristic method runs the fastest among these approach. However, they get the biggest solution size. On the other hand, heuristic algorithm by using features statistics and analysis can get the smallest solution size among these heuristic approaches, which is closest to the formal approach.

B. Heuristic BMC-SAT Approaches

Since the heuristic approaches mentioned above have a great improvement in runtime compared to the formal approach. However, they don’t guarantee the optimal results as the formal methods do. Hence, a hybrid solution is to use the results from heuristic approaches to guide the SAT solver in the formal approach to speed up the execution time.

1) *Lookup BMC*: The first approach we think of is simple: Cue the solver with the solution from the last iteration. Since the constraint gets tighter when t increases, we set the solver’s initial variables that are not in the last iteration to false, hence providing the cue.

TABLE V
MACHINE LEARNING FEATURE IMPORTANCE

Model	Feature	Importance / Coefficient
Random Forest	betweenness	0.1199
	pagerank	0.0883
	degree_ratio	0.0869
	eigenvector	0.0786
	closeness_in	0.0730
	two_hop_out	0.0723
	closeness_out	0.0632
	total_degree	0.0589
	out_degree	0.0566
	clustering	0.0541
	neighbor_out_count	0.0540
	neighbor_total_count	0.0401
	two_hop_total	0.0378
	two_hop_in	0.0352
	in_degree	0.0334
Logistic Regression	is_sink	4.0784
	closeness_in	-1.1016
	clustering	0.8898
	neighbor_total_count	-0.7547
	is_hub	-0.5979
	total_degree	0.3823
	betweenness	0.2865
	pagerank	0.2673
	eigenvector	0.2155
	out_degree	0.2075
	neighbor_out_count	0.2075
	closeness_out	-0.1762
	in_degree	0.1748
	neighbor_in_count	0.1748
	is_source	-0.1442
Random Forest Accuracy:		0.870
Logistic Regression Accuracy:		0.816

TABLE VI
PERFORMANCE SUMMARY STATISTICS

Method	Avg Solution Size	Avg Time (ms)
MaxSAT	21.4	7,115
Heuristic1	51.8	39
Heuristic2	40.1	401
Heuristic3	33.8	442

TABLE VII
METHOD PERFORMANCE COMPARISON

Comparison	Cases	Percentage	Description
Heu3 vs Heu2 (better)	16/20	80.0%	Solution quality
Heu3 vs Heu1 (better)	19/20	95.0%	Solution quality
Heu3 vs MaxSAT (optimal)	2/20	10.0%	Optimal match
Heu1 (fastest)	20/20	100.0%	Execution time
Heu2 (fastest)	0/20	0.0%	Execution time
Heu3 (fastest)	0/20	0.0%	Execution time
MaxSAT (fastest)	0/20	0.0%	Execution time

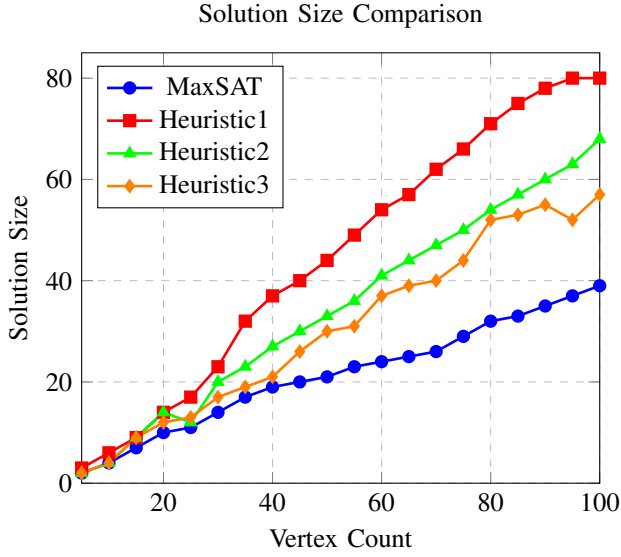


Fig. 12. Solution size comparison across MaxSAT and three heuristic methods over various problem sizes.

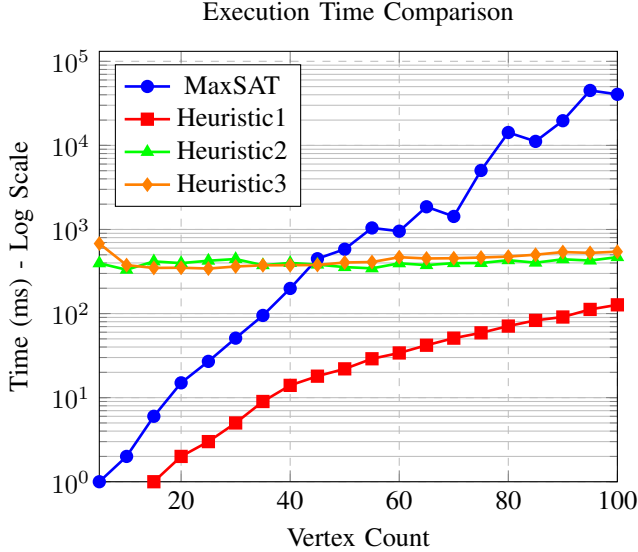


Fig. 13. Log-scale plot comparing execution times of MaxSAT and three heuristics over various problem sizes.

2) *Approximation with Forced Lookup Solution*: A more aggressive approach of lookup is to add clauses to force the solver to solve with the last solution, but it might terminate early due to the overrestriction. Therefore, this algorithm is an approximation algorithm. However, from the execution data, it seems to have found the smallest set.

3) *Heuristic Skip BMC*: From the previous data, we can observe that for any case of n , the solution size k is approximately $0.4n$, and hence our first trial of improving the algorithms is to make the program “guess the answer” and refine it. Furthermore, in section III we found out that the UNSAT cases run slower than the SAT cases, and hence we designed the algorithm to search in an “incremental

forward, leap backward” fashion instead of binary search. The procedure is as follows:

Algorithm 9 Heuristic Skip BMC

- 1) Guess that $k = 0.4n$, and thus $t = n - k = 0.6n$
 - 2) If SAT, $t = t + 1$.
 - 3) If UNSAT, $t = t - 5$.
 - 4) Continue until VS-SAT($n-t$) is SAT and VS-SAT($n-(t+1)$) = UNSAT and return VS-SAT($n-t$).
-

4) *Heuristic Skip BMC with Ratio Approximation*: Since we have analyzed the overlap between the answers from heuristic approaches and the formal approach, and we found that the first heuristic approach has the largest overlap ratio, we used this heuristic to assist the solver. The procedure is as follows:

Algorithm 10 Heuristic Skip BMC with Ratio Approximation

- 1) Run the first heuristic approach and obtain the over-approximated S_0 .
 - 2) Find the order of implication of the vertices and save the reachable sets at each time frame using the implication algorithm.
 - 3) Guess that $k = 0.4n$, and thus $t = n - k = 0.6n$.
 - 4) When building the transition clauses, also build the constraints on each layer from the order of implication of the over-approximated S_0 .
 - 5) If SAT, $t = t + 1$.
 - 6) If UNSAT, $t = t - 5$.
 - 7) Continue until VS-SAT($n-t$) is SAT and VS-SAT($n-(t+1)$) = UNSAT and return VS-SAT($n-t$).
-

This algorithm forces some values to be false, and thus is also an approximation algorithm. However, from the execution data it seems that the smallest set has also been found.

From Figure 14, we can see that the lookup methods do not perform significantly better than the standard case, but the two skip algorithms have surely skipped a lot of unrelated cases and thus reduced the run time by approximately half. However, the ratio approximation has not done a lot of help.

C. Heuristic MaxSAT Approach

We choose the first heuristic solution as an input to *MaxSAT* solver. We have assigned a higher weight to the vertices that are not in the S_0 of heuristic approach, so that the solver will try not to add these vertices into S_0 first.

Consequently, we found that in the first 100 cases, this method does not exceed the answer from the formal approach by more than 2, while achieving a much faster runtime.

In Table VIII and Figure 15, MS_{S0} is the solution size of the original MaxSAT approach, $MS_H S0$ is the solution size of the Heuristic MaxSAT approach, and the other two columns are their run-time. As shown in the figure, for case

Run Time of Heuristic BMC Approaches

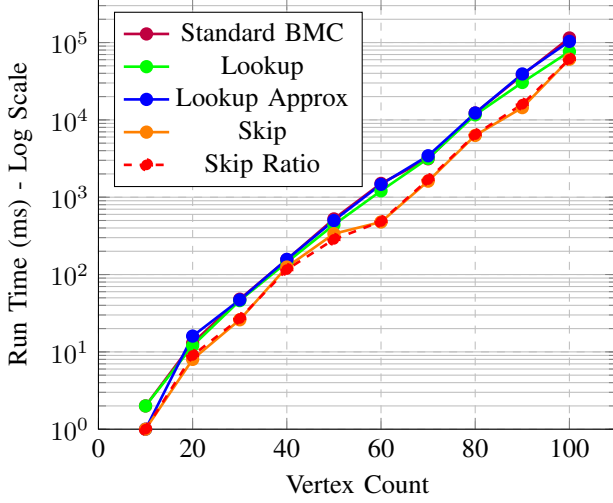


Fig. 14. Log-scale plot comparing the run time of several heuristic BMC approaches across different vertex counts. Solver: Python MiniSAT.

Run Time of MaxSAT, Heuristic MaxSAT and Standard BMC

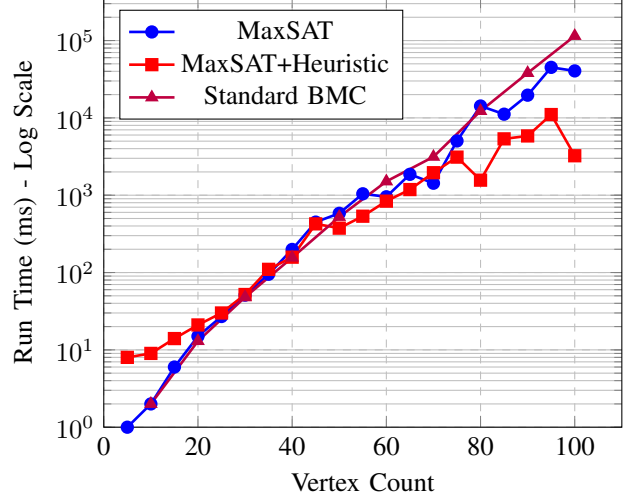


Fig. 15. Log-scale plot comparing MaxSAT, Heuristic MaxSAT and Standard BMC across different vertex counts.

100, the Heuristic MaxSAT method is an order of magnitude faster than the MaxSAT method, and two orders of magnitude faster than Standard BMC.

TABLE VIII
MAXSAT VS MAXSAT HEURISTIC PERFORMANCE COMPARISON

Size	MaxSAT S0	Heuristic MaxSAT S0
5	2	2
10	4	4
15	7	7
20	10	10
25	11	11
30	14	14
35	17	17
40	19	19
45	20	20
50	21	21
55	23	23
60	24	24
65	25	25
70	26	28
75	29	30
80	32	33
85	33	34
90	35	36
95	37	38
100	39	40

VI. CONCLUSION

In this paper, we have explored the Smallest Vocabulary Set (SVS) problem through both formal verification methods and heuristic approaches. We have conclusively demonstrated that the SVS problem is NP-hard, confirming the need for efficient approaches to handle large vocabulary sets.

Among the formal methods, our experiments revealed that Bounded Model Checking (BMC) and MaxSAT are the most effective approaches. BMC with progressive time frame expansion provides a systematic way to find optimal solutions,

while MaxSAT offers better performance by directly encoding the optimization objective. The other formal methods we implemented—Binary Search BMC, Interpolation-based Unbounded Model Checking, Property-Directed Reachability, and Binary Decision Diagrams—failed to outperform the standard BMC approach.

Our results consistently showed that the optimal initial vocabulary set typically comprises approximately 40% of the total words in the dictionary graph. This finding suggests a practical rule of thumb for estimating the size of the core vocabulary needed to learn a language without translation.

The heuristic approaches we developed offer significant advantages in computational efficiency. While they do not guarantee optimal solutions, our statistical analysis-based heuristic (Heu3) consistently produced solutions within 30-40% of optimal size, with runtimes orders of magnitude faster than formal methods. The hybrid approach combining heuristic pre-computation with MaxSAT achieved the best balance between solution quality and performance, with results typically within 1-2 words of the optimal solution while reducing runtime by 75%.

For future work, it is possible that there are more hybrid approximation approaches that can result in smaller solutions compared to the heuristic approaches but faster runtimes than the standard BMC or MaxSAT approaches. Furthermore, we can apply the algorithms to other fields of knowledge, for which certain prerequisites are needed for learning a new concept. This opens the door to broader applications such as curriculum design, concept map optimization, or skill acquisition planning, where identifying minimal yet sufficient foundational elements is crucial. Ultimately, we hope this study serves as a foundation for developing scalable tools that assist in efficient learning and knowledge transfer across a variety of domains.

VII. APPENDIX: PRESENTATION VIDEO

https://www.youtube.com/watch?v=rbsxcR_iScY

VIII. APPENDIX: EXECUTION INSTRUCTIONS

A. Basic Usage

The SVS Solver Program is executed using the `run.sh` script with the following syntax:

```
./run.sh [-n start_n] [-x end_n] [-i step]
[-S solver] [-M mode] [-a]
```

This script runs the following steps:

- 1) Compile the C++ code
- 2) Generate the input subgraph G_n from the main graph G with size n for n in $range(start_n, end_n, step)$.
- 3) Execute every case n .
- 4) Combines every output into one file: `results/results.json`
- 5) Delete all intermediate files (input, output) in `working/`
- 6) Analyze the results and save them in `results/analysis/`

TABLE IX
COMMAND-LINE PARAMETERS FOR THE SVS SOLVER

Parameter	Description	Default
-n	Starting graph size n	10
-x	Ending graph size n	20
-i	Step size between graph sizes	10
-S	Solver implementation (python or cpp)	python
-M	Algorithm mode	standard
-a	Analyze results after completion	false

B. Available Algorithm Modes

The `-M` parameter specifies the algorithm mode to use. Available options depend on the selected solver implementation.

1) Python Solver Modes:

- `standard` – Classic BMC approach
- `forward` – Forward search BMC that rebuilds transition clauses every time
- `reverse` – Reverse search BMC that rebuilds transition clauses every time
- `binary_search` – Binary search BMC
- `maxsat` – MaxSAT approach using RC2 solver
- `heuristic_ratio` – Heuristic approach based on vertex in/out degree ratios
- `bmc_lookup` – BMC with phase setting from previous solutions
- `approximate_bmc_lookup` – Heuristic BMC with negative constraints from previous solutions
- `bmc_skip` – BMC with adaptive step size skipping
- `approximate_bmc_skip_ratio` – BMC with adaptive skipping and ratio-based heuristics
- `maxsat` – MaxSAT approach using RC2 solver

- `incremental` – Incremental SAT with PySAT Sequential Counter
- `heuristic` – Pure heuristic solver (v2)
- `heuristic_v3` – Enhanced pure heuristic solver (v3)
- `maxsat_heu` – MaxSAT with heuristic initialization
- `sequential_heu` – Sequential Counter with heuristic initialization
- `incremental_heu` – Incremental SAT with heuristic initialization

2) C++ Solver Modes:

- `standard` – Classic BMC approach
- `forward` – Forward search BMC that rebuilds transition clauses every time
- `reverse` – Reverse search BMC that rebuilds transition clauses every time
- `itpubmc` – Interpolation-based Unbounded BMC forward search
- `itpubmc_reverse` – Interpolation-based Unbounded BMC reverse search
- `pdr` – Property Directed Reachability
- `pdr_reverse` – Reverse search using PDR algorithm
- `bdd` – Symbolic model checking using Binary Decision Diagrams

C. Example Executions

1) *Basic Python Solver:* Execute the solver with default parameters using Python implementation:

```
./run.sh
```

2) *C++ Solver with Analysis:* Run the C++ solver and perform result analysis:

```
./run.sh -S cpp -a
```

3) *Binary Search Algorithm:* Use the Python solver with binary search BMC algorithm:

```
./run.sh -M binary_search
```

4) *BDD-Based Symbolic Model Checking:* Execute the C++ solver using Binary Decision Diagrams:

```
./run.sh -S cpp -M bdd
```

5) *Comprehensive Benchmark:* Perform extensive testing across multiple graph sizes with result analysis:

```
./run.sh -n 10 -x 100 -i 10 -S python -M bmc_skip
```

D. Output

The solver generates:

- Console output showing progress and final solution
- JSON result files (`top_n.in.result.json`) containing solution details and performance metrics
- Combined results file (`results/results.json`) when using the `-a` flag
- Analysis reports in the `results/analysis/` directory

REFERENCES

- [1] M. Dvořák, D. Knop, and Š. Schierreich, "On the complexity of target set selection in simple geometric networks," *Discrete Mathematics and Theoretical Computer Science*, vol. 26, no. 2, 11, 2024. [Online]. Available: [arXiv:2307.06976v4](https://arxiv.org/abs/2307.06976v4)
- [2] M. Richardson and P. Domingos, "Mining knowledge-sharing sites for viral marketing," in *Proc. 8th ACM SIGKDD Int. Conf. Knowledge Discovery and Data Mining*, 2002, pp. 61–70.
- [3] D. Kempe, J. Kleinberg, and É. Tardos, "Maximizing the spread of influence through a social network," *Theory of Computing*, vol. 11, no. 4, pp. 105–147, 2015.
- [4] Free Dictionary API, Dictionary API: Free Dictionary API for English definitions, Accessed: Jun. 8, 2025. [Online]. Available: <https://dictionaryapi.dev/>
- [5] C. Sinz, "Towards an optimal CNF encoding of Boolean cardinality constraints," in *Principles and Practice of Constraint Programming (CP 2005)*, 2005, pp. 827–831, doi: 10.1007/11564751_73