Generating Efficient Solvers from Constraint Models

Shu Lin
Peking University
China
fzlinshu@pku.edu.cn

Na Meng
Virginia Tech
USA
nm8247@cs.vt.edu

Wenxin Li
Peking University
China
lwx@pku.edu.cn

ABSTRACT

Combinatorial problems (CPs) arise in many areas, and people use constraint solvers to automatically solve these problems. However, the state-of-the-art constraint solvers (e.g., Gecode and Chuffed) have overly complicated software architectures; they compute solutions inefficiently. This paper presents a novel and model-driven approach—SoGen—to synthesize efficient problem-specific solvers from constraint models. Namely, when users model a CP with our domain-specific language PDL (short for Problem Description Language), SoGen automatically analyzes various properties of the problem (e.g., search space, value boundaries, function monotonicity, and overlapping subproblems), synthesizes an efficient solver algorithm based on those properties, and generates a C program as the problem solver. SoGen is unique because it can create solvers that resolve constraints via dynamic programming (DP) search.

For evaluation, we compared the solvers generated by SoGen with two state-of-the-art constraint solvers: Gecode and Chuffed. SoGen's solvers resolved constraints more efficiently; they achieved up to 6,058x speedup over Gecode and up to 31,300x speedup over Chuffed. Additionally, we experimented with both SoGen and the state-of-the-art solver generator—Dominion. We found SoGen to generate solvers faster and the produced solvers are more efficient.

CCS CONCEPTS

• **Theory of computation** → **Theory and algorithms for application domains**; • **Software and its engineering** → **Domain specific languages**; **Automatic programming**.

KEYWORDS

Combinatorial problems (CP), constraint solvers, static analysis of problem properties, automated DP optimization

ACM Reference Format:
Shu Lin, Na Meng, and Wenxin Li. 2021. Generating Efficient Solvers from Constraint Models. In *Proceedings of the 29th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE '21), August 23–28, 2021, Athens, Greece.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3468264.3468566

1 INTRODUCTION

Combinatorial problems (CPs) [20] popularly exist in a variety of domains. Typical CPs include:

(1) *Register Allocation [21]*: How can we assign a large number of variables to a few registers?
(2) *Winner Determination in Combinatorial Auction [27]*: Given a set of bids in an auction, what is the allocation of items to bidders that maximizes the auctioneer's revenue?

Each CP involves finding a grouping, ordering, or assignment of a discrete and finite set of objects that satisfies given conditions. Constraint solving provides a means of solving CPs automatically. When using an existing constraint solver (e.g., Gecode [28], Chuffed [7], or Minion [12]) to solve a problem, users go through two stages. First, users adopt a domain-specific language (e.g., MiniZinc [24]) to *model the problem* as (1) a set of decision variables and (2) a set of constraints on those variables. Here, a **decision variable** represents a choice that must be made to solve the problem. Second, a constraint solver is used to *search for a solution* to the model, i.e., value assignment to variables such that all constraints are satisfied.

A major limitation of existing generic solvers is the scalability issue. Namely, current solvers cannot quickly solve certain CPs even if the problem size is moderate (i.e., the problem has a decent number of variables and each variable has a reasonable value range). For instance, based on our experience, although Gecode can find the shortest path between two nodes in a given graph, it cannot solve this problem within a reasonable amount of time (e.g., 3 hours) when the graph has 100 or more nodes. One reason to explain such deficiency is the inefficient backtracking search algorithm popularly used by solvers. As problem size increases, the search space can grow exponentially and the naïve search can become very slow and ineffective. Some researchers recently explored to use SAT/SMT solvers to solve CP constraints [5, 34, 37]. However, SAT/SMT solvers only handle SAT/SMT problems—a subset of CPs. They cannot solve CPs when objective functions are complex (e.g., composed of nonlinear functions).

Additionally, existing constraint solvers are usually provided as toolkits or software libraries [6, 29]. To mitigate the scalability issue mentioned above, users are supported to configure and tune the optimization options offered by solvers via programming or machine learning [13, 15, 35, 36]. However, constraint toolkits have become overly complex in an effort to support more functionalities [26]; such complexity negatively impacts the efficiency and scalability of solvers in two ways:

(1) Configuring or tuning existing solvers for large and realistic problems requires users of a great deal of expertise, and demands lots of fine-tuning effort by humans or machines.

(2) The complex software architecture of these toolkits can introduce high overheads into the constraint solving procedure, compromising any performance gain due to optimizations.

In this paper, we introduce a novel approach—SoGen (short for Solver Generator)—that generates efficient solvers from CP models to better address the scalability issue. There are two components in SoGen: a constraint modeling language PDL and a generation engine. To solve a CP with SoGen, users need to first use PDL (short for Problem Description Language) to model the problem from two or three aspects: (1) input parameters and their **domains (i.e., value ranges)**, (2) decision variables and their relations with inputs, and (3) (optionally) an objective function for optimization. When the PDL model M is sent to the generation engine, the engine analyzes M to identify i) the minimum value range of each variable, ii) any value dependency between variables, and iii) the independent variables with minimum search space. Such model analysis facilitates the engine to characterize four problem properties:

Prop1 **the search space** that can be used to decide the search strategy of any synthesized algorithm,
Prop2 **value ranges of subfunction(s)** to decide how to optimize synthesized algorithms via feasibility-based branch pruning,
Prop3 **the monotonicity of objective function** to determine how to prune branches based on the objective function in CP, and
Prop4 **overlapping subproblems** that help decide whether a search algorithm can be optimized via dynamic programming (DP).

By analyzing and using the above-mentioned properties, the engine generates an efficient problem solver implemented in C.

We did two experiments to evaluate SoGen. The first experiment compared nine solvers generated by SoGen with two state-of-the-art solvers: Gecode and Chuffed. By applying alternative solvers to 45 constraint-solving tasks and setting time limit to 30 minutes, we observed SoGen's solvers to find solutions for 43 tasks; while Gecode and Chuffed separately handled 40 and 37 tasks. More importantly, based on the property characteristics of individual problems, SoGen optimized six solver algorithms: three algorithms automatically optimized via branch pruning and another three optimized by DP. SoGen's solvers achieved up to 6,058x speedup over Gecode and up to 31,300x speedup over Chuffed. The second experiment compares SoGen with another solver generator Dominion [1, 2]. In all 15 experimented cases, SoGen's solvers worked more efficiently than the solvers created by Dominion. One of SoGen's solvers achieved up to 593x speedup.

In summary, this paper makes the following contributions:

- We developed SoGen—an approach that statically analyzes CPs and synthesizes solvers accordingly. Different from prior work, SoGen does not require users to configure any parameter; it can create problem-specific DP search algorithms.
- Our novel analysis on CPs statically extracts and characterizes four problem properties. Based on these properties of any given CP, SoGen shrinks the search space, generates a solver algorithm, and opportunistically optimizes the algorithm with branch pruning and dynamic programming.
- We evaluated SoGen by comparing it with another solver generator—Dominion, and by comparing the problem-specific solvers it generated with Gecode and Chuffed. No prior work conducted such a comprehensive evaluation as we did.

```
int main() {
  best_result = NONOPTIMAL_VALUE;
  for (v1 in range)
    for (v2 in range)
      for (...) {
        if (violateConstraint(...)) continue;
        if (findBetterSolution(...))
          update(best_result, v1, v2,...);
      }
}
```

Listing 1: An algorithm skeleton for iteration-based search

```
int main() { best_result = NONOPTIMAL_VALUE; rec(1); }
void rec(int i) {
  if (i > # of controlling variables) {
    if (violateConstraint(...)) continue;
    if (findBetterSolution(...))
      update(best_result, v1, v2,...);
    return;
  }
  for (vi in range_i) rec(i+1);
}
```

Listing 2: An algorithm skeleton for recursion-based search

In the following sections, we will introduce the background knowledge of our research (Section 2), present a running example (Section 3), and describe the two components of SoGen: PDL (Section 4) and the engine (Section 5). Our PDL Manual, program, and data are available at https://github.com/fzlinshu/SoGen.

2 BACKGROUND AND TERMINOLOGY

This section introduces CPs (Section 2.1) and two typical search strategies (Section 2.2) involved in constraint solving.

2.1 Combinatorial Problems (CP)

A typical CP searches for one solution in a finite data space. Formally, a problem is CP if given

- a set of parameter variables $\mathcal{A} = \{\alpha_1, \cdots, \alpha_M\}$ and their domains (i.e., value ranges),
- a set of decision variables $\mathcal{V} = \{v_1, \cdots, v_N\}$ and their domains, and
- a set of constraints on those variables in $\mathcal{A}$ and $\mathcal{V}$: $\mathcal{R} = \{r_1(\mathcal{A}, \mathcal{V}), \cdots, r_L(\mathcal{A}, \mathcal{V})\}$,

we are supposed to find a value assignment for $\{v_1, \cdots, v_N\}$ such that all constraints $\mathcal{R}$ are satisfied.

Combinatorial optimization problems (COPs) are a subset of CPs because in addition to the $\mathcal{A}$, $\mathcal{V}$, and $\mathcal{R}$ mentioned above, each COP also defines an objective function $f(\mathcal{A}, \mathcal{V})$. To solve a COP, we need to find the optimal value assignment for $\mathcal{V}$ such that (1) all constraints are satisfied and (2) the value of the objective function is optimal.

2.2 Two Typical Search Strategies

Generally speaking, a CP can be solved when a search algorithm enumerates all data points in a space either iteratively or recursively. Thus, there are two alternative ways to implement a constraint solver: **iteration-based search** and **recursion-based search**.

As shown in Listing 1, iteration-based search adopts one or more variables and their domains to control the number of loop iterations. In each iteration, the algorithm checks whether the current value assignment of variables (1) satisfies all constraints and (2) (optionally) leads to a better value of the objective function; if so, the algorithm updates its record to track a better solution. The algorithm returns

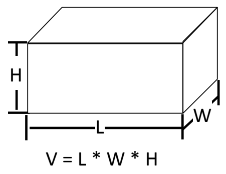

Figure 1: A cuboid with the volume V

```
#input
    V of int in [1,10^5];
#required
    L of int in [1,?];
    W of int in [1,?];
    H of int in [1,?];
    V = L * W * H;
#objective
    minimize (2 * (L * W + L * H + W * H));
```

Listing 3: The PDL model for the cuboid problem

one (optimal) solution after all iterations. Similarly, recursion-based search uses at least one variable and related domain(s) to control the number of recursive function calls (see Listing 2). In each recursive function call, the algorithm checks whether all variables have values assigned. If so, the algorithm checks if the value assignment (1) satisfies all constraints and (2) (optionally) gets a better objective function value. We use **controlling variables** to refer to the variables that control loop iterations or recursive function calls. The domains of these controlling variables define the search space.

When developing SoGen to synthesize efficient CP solvers, we tried to tackle two challenges:

C1. Given a CP, how can our approach decide which search strategy to adopt?

C2. Once a search strategy is selected, how does our approach decide which of the following two optimizations to apply: pruning and DP?

SoGen overcame both challenges by automatically characterizing properties for any given CP and making decisions accordingly.

3 A RUNNING EXAMPLE

This section overviews our approach with an exemplar COP.

3.1 Problem Statement

Given an integer V ($\leq 10^5$), find a cuboid with the smallest surface area such that: (1) the volume is V and (2) the lengths of all edges (e.g., L, W, and H in Figure 1) are integers.

3.2 Constraint Modeling with PDL

Suppose that a user Alex wants to build an automatic solver for the problem. By analyzing the problem, Alex can identify

- one input parameter `V`,
- three positive integer variables whose values will be computed: `L`, `W`, and `H`, and
- an objective function to minimize the surface area.

With PDL, Alex can create a constraint model for the problem. As shown in Listing 3, the model includes three segments: `#input`, `#required`, and `#objective`. The `#input` segment defines the input parameter `V` and its domain `[1, 10^5]`. The `#required` segment declares decision variables (i.e., `L`, `W`, and `H`), their domain `[1, ?]` (i.e., any positive integer), and their value constraint with the input: `V = L * W * H`. The `#objective` segment defines the objective function.

3.3 Algorithm Synthesis

Given the constraint model in PDL, the SoGen engine automatically synthesizes a solver algorithm by taking four steps: bound tightening, independent variable set selection, algorithm generation, and algorithm optimization.

Bound Tightening. To characterize the problem property Prop1, SoGen iteratively tightens variable domains with an off-the-shelf technique FBBT [3]. Intuitively, in the first iteration, by converting the given formula to $L = V/(W \times H)$, FBBT refines the domain of L as $D'_L = D_L \cap \big([1, 10^5]/([1, +\infty) \times [1, +\infty))\big) = [1, +\infty] \cap (0, 10^5] = [1, 10^5]$. In the second iteration, FBBT similarly refines the domain of W by using L's updated domain. This process continues until the domain of each variable is fixed as $[1, 10^5]$.

Independent Variable Set Selection. Given a set of variables, not every variable should be used as a controlling variable in the synthesized search algorithm. This is because when variables have mathematical relations with each other, the values of some variables can uniquely determine the values of other variables. To further refine Prop1 and to focus search on only feasible or promising value assignments, we defined three terms:

Definition 1. ***Value Dependencies****: Given formulas or equations, e.g., $r_j(\mathcal{A}, \mathcal{V})$, if the value of certain variable v_i is uniquely determined by the value assignment of other variables, we say that v_i has value dependencies on others.*

Definition 2. ***Independent Variable Set****: This is a subset of $\mathcal{V}$. The values of variables in this subset can uniquely determine the values of other variables in $\mathcal{V}$.*

Definition 3. ***Dependent Variable Set****: This is the complementary set of an independent variable set in $\mathcal{V}$. All variables in this subset have value dependencies on the independent variables.*

In this example, `L`, `W`, and `H` are interdependent. Namely, given the value assignment of any two variables, the third variable is computable. Therefore, `(L, W)` can be considered as an independent variable set. This step chooses the independent variable set with

Algorithm 1: The optimized iteration-based solver algorithm created by SoGen for the cuboid problem

```
     Input: $V$ /* input parameter $V$ */
     Output: $best\_result, L, W, H$ /* the values of (1) objective function and
        (2) variables */
1.1  $best\_result \leftarrow MAX\_VALUE$;
     /* 1. enumerate values of controlling variables */
1.2  foreach $L \in [1, 100000]$ do
1.3      if $V$ mod $L \neq 0$ then
1.4          continue;
1.5      foreach $W \in [1, 100000]$ do
             /* 2. check constraints on variable values */
1.6          if $V/L$ mod $W$ then
1.7              continue;
             /* 3. calculate values of dependent variables */
1.8          $H \leftarrow V/L/W$;
             /* 4. calculate the value of objective function */
1.9          $result \leftarrow 2 * (L * W + L * H + W * H)$;
             /* 5. update record if a better solution is found */
1.10         if $result \geq best\_result$ then
1.11             continue;
1.12         $best\_result \leftarrow result$;
1.13         record($L, W, H$);
```

minimum domains as the controlling variables in an algorithm-to-design. In this way, SoGen refines its characterization for Prop1.

Algorithm Generation. This step relies on Prop1 characteristics to synthesize a basic search algorithm. If there are only a few controlling variables and no variable is of any composite data type (e.g., `set`), this step creates an iteration-based search algorithm; otherwise, it synthesizes a recursion-based algorithm in order to make the generated code more compact and readable. For this example, since there are only two primitive-typed variables (i.e., `L` and `W`), an iteration-based algorithm with the two-level nested loop structure can solve the problem. In particular, inside the inner loop, the constraint-related `if`-condition is: `(V == L * W * H)`.

Algorithm Optimization. This step characterizes and uses properties Prop2–Prop4 to opportunistically optimize the basic algorithm. Specifically, SoGen tentatively decomposes each constraint/objective function into smaller subfunctions. Among all **valid solutions** (e.g., the value assignments that can satisfy all constraints), if the potential values of any constraint subfunction are bounded by the tightened variable domains, Prop2 is characterized accordingly. If the objective function monotonically increases or decreases, Prop3 is consider satisfied. If (1) Prop2 is characterized, (2) Prop3 is satisfied, and (3) the search space of a potential DP algorithm (estimated by Prop2) is much smaller than that of the basic algorithm, Prop4 is considered satisfied. SoGen applies branch pruning if Prop2 tightens the value ranges of subfunctions or Prop3 holds. SoGen applies DP if Prop4 holds.

In our example, the constraint formula `(V == L * W * H)` can be decomposed to two subfunctions: `V/L` and `V/L/W`. Among all valid solutions, the potential values of these subfunctions are actually bounded by variable domains (e.g., because `V/L==W*H`, values of `V/L` should be positive integers). Thus, SoGen characterizes Prop2 and applies branch pruning, to avoid enumerating invalid solutions. The synthesized algorithm by SoGen is illustrated in Algorithm 1.

3.4 Code Generation

Based on the synthesized algorithm, SoGen produces a constraint solver implemented in C. The solver program implements not only the algorithm, but also two utility functions: `_input()`—to read values of input parameters from console, and `_output()`—to write the optimal value of objective function and variable values to console. With a solver generated for the cuboid problem, Alex can enter any `V` value to instantiate the COP. Then the solver responds with the minimum surface area and related values of `L`, `W`, and `H`.

4 PDL

PDL is a constraint modeling language for users to describe CPs. As illustrated in Listing 3, a PDL model consists of three segments: `#input`, `#required`, and `#objective`.

Input Segment (`#input`) declares all input parameters and their domains. These inputs will be declared as formal arguments by a generated program. This segment can have zero or more input parameter declaration. A parameter can be declared with any primitive type (e.g., `int`) or composite type (e.g., `list`).

Required Segment (`#required`) declares decision variables, their domains, and their relations with inputs (i.e., data constraints). A required segment has zero or more statement. A statement can be a variable declaration, expression statement, or `forall`-enumeration.

Table 1: Built-in operators and functions in PDL

Category	Symbols
Relational Operators	`=, !=, >, <, >=, <=,`
Logical Operators	`and, or, not, xor`
Arithmetic Operators	`+, -, *, /, mod`
Exponent Operators	`^`
Aggregation Functions	`min, max, summation, product`

Each expression statement has an expression to describe data constraints as mathematical formulas. The `forall`-enumeration is a higher-order function, which applies a given function to all elements in a composite data structure.

PDL supports common data types, including primary types (i.e., int, real, char, and bool), and composite types (i.e., array, set, and struct). PDL supports various expression formats, such as

- a parenthesized expression,
- an atomic expression (e.g., a variable `A`),
- a unary expression (e.g., `not A`),
- a binary expression (e.g., `A+B`),
- an aggregate result of `forall`-enumeration (e.g., `summation [A[i]: forall i in [1, 10]]`),
- a quantifier function to check whether there exist certain value(s) to meet certain conditions (e.g., `exists a (a in [1, 10])`), and
- a conditional expression (e.g., `if a (b=1) else (b=0)`).

PDL also defines built-in operators and functions (see Table 1).

Objective Segment (#objective) declares zero or one objective function. If no objective function is defined, the resulting program stops search after finding *a* solution; otherwise, it searches for an *optimum* that acquires the best objective value among all solutions.

PDL has a similar type system to C and shares type inference rules. However, PDL supports fewer data types, including int, real, char, bool, array, set, and struct. To learn more about PDL, please refer to our PDL Manual on GitHub.

5 THE SOGEN ENGINE

As shown in Figure 2, there are five major steps in SoGen's generation engine. Steps 1-4 automate algorithm synthesis, and Step 5 automates algorithm implementation. Sections 5.1–5.5 explain each step in detail.

5.1 Bound Tightening

Given a PDL model, SoGen tokenizes the model and conducts both syntax and semantic analysis to build an **identifier table** (i.e., a table to record parameters, variables, and their domains), constraint formulas, and any objective function. Specifically, we built a PDL compiler frontend with JavaCC [17] and Java Tree Builder (JTB) [14]. JavaCC is a scanner and parser generator for LL(k) grammars. It takes in the token patterns defined with regular expressions to generate a lexical analyzer (i.e., scanner); it also generates a parser from the given syntax grammar defined in EBNF. We used the generated scanner and parser to create a parsing tree for any given PDL model. Additionally, we used JTB to implement a tree visitor for semantic analysis. With the visitor, SoGen traverses each parsing tree to extract all constraints and any objective formula, and to create an identifier table.

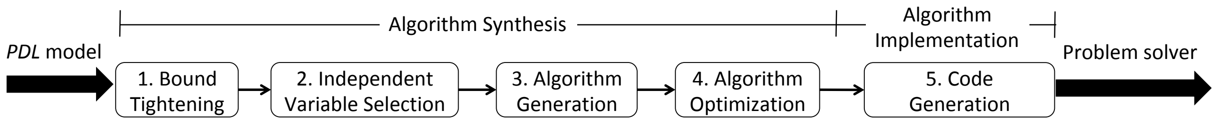

Figure 2: The SoGen engine takes five steps to generate a solver from the given PDL model

Table 2: Identifier table for the cuboid problem

Identifier	Parameter/Variable	Type & Range
V	Parameter	int in $[1,10^5]$
L	Variable	int in $[1,10^5]$
W	Variable	int in $[1,10^5]$
H	Variable	int in $[1,10^5]$

```
#required
    A of int in [1, 5];
    B of int in [1, 10];
    C of int in [1, 500];
    D of int in [1, 250000];
    C = A * B ^ 2;
    D = B ^ 2 * C ^ 2;
```

Listing 4: An exemplar `#required` segment

Similar to the symbol table produced by a traditional compiler, our identifier table records both the names and types of parameters and variables. However, different from symbol tables, identifier tables also record (1) whether an identifier is a parameter or variable, and (2) the value range of an identifier (see an exemplar identifier table in Table 2). These value ranges are computed with an off-the-shelf range reduction technique FBBT [3]. Intuitively, given $a \in [0,0], b \in [0,1], c \in [0,1], a = b + c$, FBBT first converts the formula to $b = a - c$, and then refines the domain for b as $D'_b = D_b \cap ([0,0] - [0,1]) = [0,1] \cap [-1,0] = [0,0]$. By reducing the specified value ranges of variables using FBBT, SoGen intends to shrink the search space and characterize Prop1.

5.2 Independent Variable Selection

This step identifies **controlling variables**—variables used to control the number of iterations or recursions in search algorithms. Specifically, according to the identifier table and related constraint formulas, SoGen reveals dependencies between variables, and recognizes all alternative sets of independent variables. It selects the minimum set with minimum value ranges as controlling variables for two purposes. First, controlling variables define the search space of the algorithm-to-design. The fewer controlling variables there are and the smaller value ranges they have, the smaller search space needs to be explored. Second, SoGen relies on these variables to choose the search strategy (i.e., either iteration-based or recursion-based).

Value Dependency Identification. We explain this process with a concrete example. Listing 4 shows an exemplar `#required` section, which declares four variables (i.e., A, B, C, and D) and defines two constraints. According to the first constraint, C has value dependencies on A and B, because C can be computed based on the values of those variables. Formally, we represent this dependency as:

$$C \lhd A \land B \tag{1}$$

Similarly, we can identify another dependency relationship by converting the formula to $A = C/B^2$:

$$A \lhd B \land C \tag{2}$$

However, B is not dependent on A or C. This is because given values of A and C, we cannot determine a single value for B based on the converted formula $B = \pm\sqrt{C/A}$. According to the second constraint, we can similarly identify the following value dependency:

$$D \lhd B \land C \tag{3}$$

Variable Subset Enumeration. With all dependencies identified, SoGen sorts all variables in an ascending order of their value ranges. It then enumerates all possible variable subsets in a depth-first manner to find the minimum independent variable set with the minimum search space. For the example in Listing 4, the sorted list is $[A, B, C, D]$. Thus, the enumeration procedure first includes A into a candidate set $\mathcal{I}_1$. As A alone cannot uniquely determine the value of any other variable, the procedure continues to include B into the set, obtaining $\mathcal{I}_1 = \{A, B\}$. Based on Relations (1) and (3), the procedure concludes that all remaining variables are dependent on $\mathcal{I}_1$, so $\mathcal{I}_1$ is an independent variable set. The search space defined by $\mathcal{I}_1$ is the Cartesian production of all included variables' value ranges, whose size is $s_1 = range(A) \times range(B)$.

In the next round of subset exploration, the procedure tentatively includes B into a new candidate set $\mathcal{I}_2$. Since no variable solely depends on B, we can further add C into $\mathcal{I}_2$ to build another independent variable set. Correspondingly, the search space size is $s_2 = range(B) \times range(C)$. Since $s_2 > s_1$, we consider $\mathcal{I}_1$ to be better than $\mathcal{I}_2$. Our search continues until every subset is enumerated. We use the variables in the optimal independent variable set as controlling variables, which characterize Prop1.

5.3 Automatic Algorithm Generation

This step relies on Prop1 to decide whether an iterative or recursive algorithm should be created.

Generation of Iteration-Based Algorithms. If Prop1 corresponds to a few controlling variables (e.g., < 10) and all variables have primitive types (e.g., `bool`), SoGen creates an iterative algorithm based on the skeleton shown in Listing 1. This is because compared with recursion-based algorithms, iteration-based algorithms have higher efficiency. Please refer to Algorithm 1 for an exemplar iterative algorithm produced by SoGen.

Generation of Recursion-Based Algorithms. If Prop1 corresponds to a large number of controlling variables (e.g., ≥10) or any of the variable has a composite data type (e.g., `list`), SoGen chooses to create a recursive algorithm based on the skeleton in Listing 2. SoGen's decision making is based on four rationales. First, any composite data type can be treated as a set of independent controlling variables. Second, when there are many controlling variables, recursion-based algorithms are more compact and readable. Compactness ensures software reusability and maintainability. Readability facilitates experts to further optimize generated solvers as needed. Third, recursion-based algorithms can effectively solve CPs that have more controlling variables. Iteration is limited by the maximum depth of nested loops (e.g., 127 for C), while the recursion depth can be much larger (e.g., some thousands). Fourth, the extra runtime overhead of recursive function calls over loop iterations is negligible, compared with the overall solving time.

Natural Language	PDL Model

There are N items.
There is a knapsack of capacity C.
The i^{th} item ($i \in N$) has value V_i and weight W_i.
 	
Put a set of items $S \subseteq N$ in the knapsack, such that the sum of weights is at most C.
 	
The sum of values should be maximal.
	`#input`
	`    N of int in [1,100];`
	`    C of int in [1,1000];`
	`    W of (int in [1,1000])[1~N];`
	`    V of (int in [1,1000])[1~N];`
	`#required`
	`    sel of (int in [1,N]){};`
	`    summation [W[i] : forall i (i in sel)] <= C;`
	`#objective`
	`    maximize summation [V[i] : forall i (i in sel)];`

Figure 3: The description of the 0/1 knapsack problem in natural language vs. in PDL

Algorithm 2: The `main()` function in the recursion-based algorithm for the 0/1 knapsack problem

```
    Input: $N, C, W, V$ /* input parameters */
    Output: $best\_result$, $\_sel$/* the values of (1) objective function and (2)
        variable */
2.1 $best\_result \leftarrow 0$;
2.2 $wsum \leftarrow 0$, $vsum \leftarrow 0$; /* two local variables used to accumulate the
        weights and values of selected items */
2.3 rec(1, $wsum$, $vsum$, $N$, $C$, $W$, $V$);
```

Algorithm 3: The `rec(...)` function invoked by the `main()` function mentioned in Algorithm 2

```
     Input: $step, wsum, vsum, N, C, W, V$/* $step$ implies which controlling
         variable has values enumerated */
     Output: $\emptyset$
     /* 1. check whether all controlling variables have values assigned
         already */
3.1  if $step > N$ then
         /* 2. check constraints */
3.2      if $wsum > C$ then
3.3          continue;
         /* 3. update record if a better solution is found */
3.4      if $vsum \le best\_result$ then
3.5          continue;
3.6      $best\_result \leftarrow vsum$;
3.7      record($\_sel$);
3.8      return;
     /* 4. explore all possible values for the controlling variable _sel[step]
         */
3.9  foreach $\_sel[step] \in [0, 1]$ do
         /* 5. evaluate the objective function based on the variable values
             assigned so far */
3.10     rec($step + 1$, $wsum + W[step] * \_sel[step]$, $vsum + V[step] * \_sel[step]$,
           $N, C, W, V$);
```

Figure 3 shows the natural-language description and PDL model of another exemplar CP: the 0/1 knapsack problem. In the constraint model, the only decision variable `sel` is a set that holds the indexes of items put into a knapsack. To generate a solver algorithm for this problem, SoGen first converts `sel` to an N-length boolean array `_sel`, where `_sel[i]` indicates whether the number i is in the set. Next, SoGen creates a recursion-based algorithm based on inputs, `_sel`, the constraint, and the objective (see Algorithms 2 and 3).

5.4 Automatic Algorithm Optimization

SoGen characterizes Prop2–Prop4, and optimizes any synthesized algorithms based on those properties. This section first clarifies notations (Section 5.4.1) and describes the property characterization process (Section 5.4.2); next, it introduces SoGen's decision-making to optimize algorithms based on properties (Section 5.4.3).

5.4.1 Notations. To simplify explanation, suppose that the PDL model has m constraints. A solver algorithm has n controlling variables. At step $i \in [1, n]$ (i.e., the i^{th} loop structure or i^{th} recursion), there are i variables with value assignment and $(n - i)$ variables without value assignment; we denote these two sets of variables separately as $A_i = \{v_1, v_2, \ldots, v_i\}$ and $U_i = \{v_{i+1}, v_{i+2}, \ldots, v_n\}$. For any COP, in the search procedure, we denote the suboptimal value of objective function obtained so far (i.e., intermediate optimal value) as $best\_result$.

5.4.2 Property Characterization. To characterize Prop2 (value range of subfunctions), SoGen tentatively converts each constraint to subfunctions P and Q matching the following formats:

$$P(A_i)\ Cmp\ Q(U_i), \tag{4}$$

where P and Q derive from the original constraint, and Cmp represents any comparison operator (e.g., =, ≤, and ≥) used in the constraint. Suppose that the value ranges of A_i and U_i separately define value ranges for P and Q functions as r_1 and r_2. Given Cmp and r_2, SoGen checks whether there is any value in r_1 that can never satisfy the converted mathematic relationship; if so, SoGen discards those values and characterizes Prop2 to refine P's value range. SoGen later uses such refined ranges to prune search branches (see Section 5.4.3). For our running example, when $V = L \times W \times H$, SoGen can generate two converted formulas:

$$V/L = W \times H \tag{5}$$
$$V/L/W = H \tag{6}$$

Because the potential value range of $W \times H$ in Formula (5) is positive integers, SoGen infers that V/L should be an integer. Similarly, as the domain of H in Formula (6) is positive integers, SoGen infers $V/L/W$ to be an integer. Prop2 captures all such derived value constraints for P functions, e.g., $\{V\ mod\ L == 0, V/L\ mod\ W == 0\}$.

To characterize Prop3 (the monotonicity property of objective function), SoGen decides whether the objective function monotonically increases (or decreases) with all n variables by checking the coefficients of each variable. If the coefficients are all positive (or negative) values, Prop3 holds and SoGen later prunes search branches accordingly (see Section 5.4.3). For the 0/1 knapsack problem in Figure 3, the objective function is:

$$Obj = \sum_{j=1}^{N} V[j] \times sel[j] (where\ V[j] > 0). \tag{7}$$

As the coefficients of $sel[j]$ (i.e., V[j]) are all positive, Prop3 holds.

SoGen checks Prop4 (overlapping subproblems) only if Prop2 bounds every constraint subfunction involving any controlling variable and Prop3 holds; if Prop4 holds, SoGen later converts the synthesized algorithm to a dynamic programming (DP) algorithm for optimization (see Section 5.4.3). Specifically, SoGen relies on Prop2 to estimate the search space S_1 of a potential DP algorithm; it also uses the value range of controlling variables to estimate the search space S_2 of a brute-force search algorithm. If the size of S_2 is

Algorithm 4: The `rec(...)` function produced by SoGen when it applies branch pruning

```
Input: $step$, $wsum$, $vsum$, $N$, $C$, $W$, $V$ /* $step$ implies which controlling
    variable has values enumerated */
Output: $\emptyset$
/* check whether all controlling variables have values assigned already
    */
4.1  if $step > N$ then
         /* update related record if a better solution is found */
4.2      if $vsum \le best\_result$ then
4.3          continue;
4.4      $best\_result \leftarrow vsum$;
4.5      record($\_sel$);
4.6      return;
/* explore all possible values for the controlling variable $\_sel[step]$ */
4.7  foreach $\_sel[step] \in [0, 1]$ do
         /* evaluate the objective function based on the variable values
            assigned so far */
4.8      $wsum' \leftarrow wsum + W[step] * \_sel[step]$;
4.9      $vsum' \leftarrow vsum + V[step] * \_sel[step]$;
         /* FBP: check constraint violation based on $wsum'$ */
4.10     if $wsum' > C$ then
4.11         continue;
         /* OBP: assess the potential upper bound value of objective function
            based on the value sum so far */
4.12     if $vsum' + 1000 * (N - step) \le best\_result$ then
4.13         continue;
4.14     rec($step + 1$, $wsum'$, $vsum'$, $N$, $C$, $W$, $V$);
```

much larger than that of S_1 (i.e., $|S_2| >> |S_1|$), SoGen concludes that there are overlaps between subproblems of the objective function and Prop4 holds. For the 0/1 knapsack problem, Prop3 is true,

$$\begin{aligned} Prop2 &= \{0 \le \sum_{j=1}^{i} W[j] \times sel[j] \le C\}, \\ |S_1| &= \Theta(N \times C), \\ |S_2| &= 2^N. \end{aligned}$$

Θ means there exist positive constants c_1 and c_2 ($c_1 < c_2$) such that $c_1 \times N \times C \le |S1| \le c_2 \times N \times C$. As $|S_2| >> |S_1|$, Prop4 holds.

5.4.3 Optimization Application.

SoGen is capable of applying two types of algorithm optimizations: branch pruning and DP.

Branch pruning adds, restructures, or moves `if`-branches to reduce the number of explicitly enumerated values. There are two types of pruning SoGen automates:

Feasibility-Based Pruning (FBP): Based on the value assignment of some instead of all controlling variables, this optimization makes early decisions on constraint violation. If any constraint is definitely violated even if some involved variables have values unassigned, the search tree is pruned.

Objective-Based Pruning (OBP): This optimization is applied to COPs. Given the value assignment of some instead of all controlling variables, OBP predicts the potential values of objective function when all variables have values assigned. If the predicted values are unpromising (e.g., worse than the intermediate best result), the search tree is pruned.

Dynamic Programming (DP) breaks a given COP into simpler subproblems. It first solves subproblems and caches optimal solutions in a table for reuse. Then it solves the overall problem based on optimal solutions to subproblems.

Decision-Making for Feasibility-Based Pruning (FBP). When Prop2 is not empty, SoGen replaces all `if`-condition checks on original constraints with those on derived subfunctions in Prop2. When any subfunction only involves a subset of controlling variables, SoGen further moves the related `if`-condition from the innermost loop (or recursion) to some outer loop (or recursion) such that unpromising search subtrees are pruned as early as possible. Please refer to Algorithm 4 for an exemplar FBP applied by SoGen.

Decision-Making for Objective-Based Pruning (OBP). When Prop3 holds, SoGen inserts an `if`-condition check that compares the partial evaluation result of objective function based on A_i, the intermediate optimum $best\_result$, and U_i. If no matter how variables in U_i get values assigned, the existing partial evaluation result is unpromising to lead to a better objective value than $best\_result$, then SoGen uses the inserted check to prune branches. Algorithm 4 shows an exemplar OBP SoGen applied.

Decision-Making for DP. When Prop4 holds, SoGen restructures the synthesized recursion-based algorithm to have two parts: (1) table creation and (2) table access. For table creation, SoGen initializes a table of $(m + 1)$ dimensions. Among these dimensions, the first one has values within [1, n] and corresponds to the n steps of function recursion. The other m dimensions correspond to the derived subfunctions in Prop2. Each table cell caches an optimal subproblem solution. For table accesses, SoGen defines two `if`-structures in the algorithm. One structure checks whether a given subproblem is already solved, and looks up the table if so. The other structure puts the result of a newly solved subproblem to table if the result is better than the value on record. Please refer to Algorithm 5 for the DP algorithm SoGen generated for 0/1 knapsack problem.

Algorithm 5: The `rec(...)` function optimized by SoGen when it applies DP and branch pruning

```
Input: $step$, $wsum$, $vsum$, $N$, $C$, $W$, $V$ /* step implies which controlling
    variable has values enumerated */
Output: $\emptyset$
/* reuse the result if the subproblem has been solved before */
5.1  if $DP\_sel[step][wsum] \ne null$ then
5.2      $vsum \leftarrow vsum + DP\_sel[step][wsum]$;
5.3      $step \leftarrow N + 1$;
5.4      copy $\_sel[step..N]$ from the recorded subproblem solution;
5.5  if $step > N$ then
5.6      $result \leftarrow vsum$;
5.7      if $wsum > C$ then
5.8          return $-1$;
5.9      if $result \ge best\_result$ then
5.10         $best\_result \leftarrow result$;
5.11         record($\_sel$);
5.12         return $vsum$;
5.13 foreach $\_sel[step] \in [0, 1]$ do
5.14     $wsum' \leftarrow wsum + W[step] * \_sel[step]$;
5.15     $vsum' \leftarrow vsum + V[step] * \_sel[step]$;
5.16     if $wsum' > C$ then
5.17         continue;
5.18     if $vsum' + 1000 * (N - step) \le best\_result$ then
5.19         continue;
5.20     $tmp \leftarrow$ rec($step + 1$, $wsum'$, $vsum'$, $N$, $C$, $W$, $V$) $- vsum$;
         /* if the result is better than the recorded subproblem solution,
            use it to update the record */
5.21     if $tmp > DP\_sel[step][wsum]$ then
5.22         $DP\_sel[step][wsum] \leftarrow tmp$;
/* return the optimal value of objective function */
5.23 return $vsum + DP\_sel[step][wsum]$;
```

The rationale of SoGen's decision-making for DP is as below. DP can optimize search when a problem has two properties: **optimal**

substructures and **overlapping subproblems** [9]. Optimal substructures mean that an optimal solution can be constructed with the optimal solutions to its subproblems; this property corresponds to Prop3. Overlapping subproblems indicate that a naïve search algorithm repetitively solves some subproblems; this property is actually Prop4. As SoGen checks Prop4 only if Prop2 is not empty and Prop3 is true, it applies DP based on the fact that both Prop3 and Prop4 are satisfied. In this way, SoGen ensures that its generated DP algorithms can improve solver efficiency.

5.5 Code Generation

To generate code for each synthesized algorithm, SoGen has six predefined code templates:

- `_input(...)` reads parameters from the console.
- `_output(...)` prints the optimal value of objective function and related variable values.
- `_update(...)` compares a new solution `result` with `best_result`, and updates recorded values if `result` is better.
- `_find(...)` implements the recursive function for recursion-based search algorithms.
- `_solve(...)` enumerates values of controlling variables, checks constraint formulas, evaluates the objective function, and updates records. It calls `_find(...)` (optional) and `_update(...)`.
- `main()` integrates all functions into a program.

We designed SoGen to implement algorithms in C because the language is simple and efficient. However, our methodology is not limited to C and can be implemented to generate code in other languages as well.

6 EVALUATION

To assess the usefulness of SoGen, we conducted two experiments and explored two research questions (RQs):

- **RQ1:** How do the solvers generated by SoGen compare with state-of-the-art CP solvers?
- **RQ2:** How does SoGen compare with the state-of-the-art solver generator?

6.1 Empirical Comparison with CP Solvers

This section first introduces our dataset and experiment settings, and then discusses our results.

6.1.1 Dataset. We created a dataset based on nine classical and representative CPs from undergraduate programming courses and CSPLib [10]—a program library for constraints. The problems are different in terms of constraints, objective functions, variables, sizes of the search spaces, and possible solver optimization techniques. Therefore, the included problems are diverse; they represent a much larger set of CPs. As shown in Table 3, depending on the problem, the complexity of a naïve backtracking algorithm can be $O(N^C)$, $O(C^N)$, or $O(N!)$. Here N is a parameter mentioned in the original problem description. Complexity reflects the space size of each backtracking search. As the search space grows with N, we defined five constraint-solving tasks/instances for each problem by setting N to distinct values:

(1) If the complexity is polynomial (i.e., $O(N^C)$), we set N to 10^3, 10^4, 10^5, 10^6, and 10^7 because current solvers usually solve the problems efficiently.

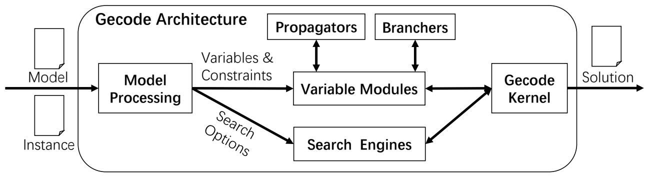

Figure 4: The architecture of Gecode

(2) If the complexity is exponential (i.e., $O(C^N)$), we set N to smaller numbers: 50, 100, 150, 200, and 250. This is because when N is too large, existing solvers do not respond in a timely manner.

(3) If the complexity is factorial (i.e., $O(N!)$), we set N to even smaller numbers: 5, 10, 15, 20, and 25. This is because current solvers can only solve these problems when N is small. P7 is an outlier. Although the theoretical complexity is $O(N!)$, P7 is a constraint satisfaction problem that only requires for one feasible solution. Therefore, P7 can be effectively solved by current solvers in practice; and we set N to 50, 100, 150, 200, and 250.

With the above-mentioned method, we created 45 tasks.

6.1.2 Experiment Settings. We compared the solvers generated by SoGen with generic solvers Gecode [28] and Chuffed [7], by applying them to our dataset. We did not experiment with any SMT solver, because SMT solvers only handle SMT problems—a subset of CPs, which are decision problems for logical formulas with respect to combinations of background theories expressed in first-order logic with equality. The state-of-the-art SMT solver—Z3 [4, 23]—cannot solve CPs when objective functions are non-linear.

Gecode and Chuffed are two state-of-the-art generic constraint solvers widely used by people to solve CPs. Both Gecode and Chuffed have complex architectures. Chuffed adapts techniques from SAT solving, such as conflict clause learning, to speed up constraint solving [31]. As shown in Figure 4, Gecode has multiple components. In particular, the *Search Engines* component supports alternative search strategies, while *Propagators* and *Branchers* offer a variety of configurable optimizations [29]. To solve CPs with these solvers, we used MiniZinc [24], a high-level and solver-independent constraint modeling language, to describe each CP task by specifying input parameters, a set of decision variables, constraint formulas, and (optionally) the objective function. Then we used a standard third-party tool mzn2fzn [33] to convert MiniZinc models to FlatZinc models, as FlatZinc [32] is a low-level and solver-dependent modeling language acceptable by Gecode and Chuffed.

Solvers by SoGen were generated from PDL models. We used PDL to describe the nine problems in Table 3 and used SoGen to generate nine solvers accordingly. Among these solvers, SoGen optimized three solvers via branch pruning (i.e., solvers for P4, P7, and P8); it optimized another three solvers via both pruning and DP (i.e., solvers for P5, P6, and P9). In our experiment, we applied each solver to the five constraint-solving tasks mentioned in Table 3.

Procedure. We conducted the experiment on a computer that has an Intel Core i5-7300HQ 2.5GHz CPU and 8G RAM. We applied three solvers to each task: one solver generated by SoGen, Gecode, and Chuffed. For fair comparison, We used the default settings of all experimented tools, and modeled the problems with PDL/MiniZinc in semantically equivalent ways. To ensure the representativeness of our results, we applied each solver to every task three times. We recorded solvers' runtime and memory costs, and averaged the

Table 3: The nine problems and related constraint-solving tasks used in the first experiment

Id	Problem Summary	Complexity	Constraint-Solving Tasks
P1	*Greatest Common Divisor:* Find the greatest common divisor of given N positive integers.	$O(N)$	N = 10^3, 10^4, 10^5, 10^6, 10^7
P2	*Cake Baking:* There are two types of cakes. Given (1) both the ingredients and profit of each cake and (2) the total amounts of ingredients available, suppose that at most N cakes of each type can be made. How many cakes of each type should a baker make to maximize the profit?	$O(N^2)$	N = 10^3, 10^4, 10^5, 10^6, 10^7
P3	*Cuboid Problem:* Given V and N, find a cuboid with the smallest surface area such that (1) the volume is V and (2) the lengths of all edges are integers and not exceeding N.	$O(N^2)$	N = 10^3, 10^4, 10^5, 10^6, 10^7
P4	*Map Coloring:* Color the N nodes of a graph with four colors so that no two adjacent nodes have the same color.	$O(4^N)$	N = 50, 100, 150, 200, 250
P5	*0/1 Knapsack Problem:* See Figure 3	$O(2^N)$	N = 50, 100, 150, 200, 250
P6	*Teamwork:* Given the cooperation value and working value of each candidate, select a subset among N candidates such that (1) the total cooperation value is positive and (2) the total working value is maximum.	$O(2^N)$	N = 50, 100, 150, 200, 250
P7	*N Queens:* Put N queens on an $N \times N$ chess board so that no queen can attack others.	$O(N!)$	N = 50, 100, 150, 200, 250
P8	*Traveling Salesman Problem:* Find the shortest Hamiltonian cycle in a graph of N nodes.	$O(N!)$	N = 5, 10, 15, 20, 25
P9	*Shortest Path:* Find the shortest path from Node 1 to Node N in a given graph.	$O(N!)$	N = 5, 10, 15, 20, 25

Table 4: The time and memory costs of each solver on 45 constraint-solving tasks

	Time Cost (Second)			Memory Cost (MB)		
Id	SoGen	Gecode	Chuffed	SoGen	Gecode	Chuffed
P1	[0.19, 0.30, 0.38, 0.41, 4.09]	[0.66, 1.15, 2.00, 10.51, 96.50]	[0.90, 2.54, 2.79, 20.54, 325.89]	3–12	16–136	20–139
P2	[0.02, 0.03, 0.03, 0.05, 0.24]	[0.63, 0.63, 0.65, 0.81, 1.31]	[0.61, 0.67, 0.68, 3.25, 147.80]	3	13	13
P3	[0.02, 0.02, 0.04, 0.05, 0.36]	[0.61, 0.60, 0.82, 1.24, 5.72]	[1.49, 7.56, 48.05, >1800, >1800]	3	13-21	N/A
P4	[0.10, 4.23, 47.52, >1800, >1800]	[0.66, 1.14, 6.88, 395.32, >1800]	[0.59, 0.79, 3.25, 247.52, >1800]	N/A	N/A	N/A
P5	[0.03, 0.06, 0.11, 0.32, 0.56]	[4.35, 363.49, >1800, >1800, >1800]	[2.95, 191.22, 1523.72, >1800, >1800]	3–5	N/A	N/A
P6	[0.02, 0.03, 0.06, 0.11, 0.28]	[0.73, 3.24, 60.53, 134.24, 863.72]	[0.70, 1.92, 5.83, 14.98, 71.29]	3–4	15–21	16–22
P7	[0.02, 0.02, 0.14, 0.36, 0.55]	[0.67, 0.74, 3.99, 6.07, 45.41]	[0.72, 0.75, 1.86, 2.08, 23.50]	3	13–17	13–19
P8	[0.02, 0.07, 9.23, 56.24, 842.73]	[0.64, 0.74, 24.87, 532.91, >1800]	[0.60, 3.56, 72.13, >1800, >1800]	3–8	N/A	N/A
P9	[0.02, 0.02, 0.03, 0.04, 0.06]	[0.78, 0.95, 4.52, 65.36, 242.91]	[0.79, 1.59, 82.67, 1252.01, >1800]	3–4	13–220	N/A

"N/A" means that the memory cost cannot be calculated because some constraint-solving processes were interrupted due to timeout.

values among all three runs. Due to the time limit, if a solver did not respond within 30 minutes, we terminated the execution.

6.1.3 Results. Table 4 presents our results. Due to the space limit, each row shows results for five tasks related to the same problem. Among the five tasks, since the time costs of solvers change more significantly than memory costs, for each solver, this table shows all measured values of time costs and value ranges of memory costs.

According to the table, all solvers have relatively low memory costs but their time costs vary a lot. It implies that CPU instead of memory is the performance bottleneck. This is expected because constraint-solving is computationally intensive; all solvers focus their efforts on the enumeration of variable values and the evaluation of constraint formulas as well as objective functions. Compared with Gecode and Chuffed, SoGen's solvers successfully handled more tasks and often used less time and memory. Specifically, the solvers by SoGen fulfilled 43 tasks and triggered timeouts for only P4. Meanwhile, Gecode handled 40 tasks and obtained timeouts when solving P4, P5, and P8; Chuffed handled only 37 tasks and acquired timeouts when dealing with P3, P4, P5, P8, and P9.

Among the 35 tasks commonly solved by distinct solvers, on average, SoGen's solvers achieved 401x speedup over Gecode and 1,167x speedup over Chuffed; their average memory costs are 20% of Gecode's and 19% of Chuffed's. In particular, compared with Gecode, SoGen's P5 solver achieved the highest speedup—6,058x when $N = 100$. In comparison with Chuffed, SoGen's P9 solver obtained the highest speedup—31,300x when $N = 20$. Two reasons can explain the better performance of SoGen's solvers. First, SoGen's solvers are small C code; they have no software infrastructure to consume additional resources. Second, SoGen's property characterization enables it to minimize the search space, select the best search strategy, and apply optimizations as needed. By minimizing search space, SoGen could select the the most efficient search strategy and avoid fruitless value enumeration; thus, it outperformed Gecode and Chuffed when solving P1–P3 tasks even though no optimization was applied. When solving P5–P9, SoGen's solvers worked better because they optimized search based on properties.

When solving P4, the efficiency comparison is $Chuffed > Gecode$ > SoGen. As mentioned earlier, each CP corresponds to multiple solving tasks (i.e., problem instances). For P4, a solving task is determined by an N value (i.e., 4) and a map, while the map concretizes constraints between specific controlling variables (i.e., adjacent nodes have distinct colors). Such constraints are specific to individual tasks; they are not statically inferable from the PDL model of P4. They are not leveraged by SoGen for optimization either, as SoGen applies problem-specific optimizations based on static analysis of problem properties. Meanwhile, Gecode and Chuffed apply task-specific optimizations (e.g., variable reordering) based on dynamic analysis of the solving procedure for individual tasks.

Finding 1: *SoGen's solvers worked better than Gecode and Chuffed by handling more tasks given limited time. SoGen's solvers accelerated the solving process by up to 3–4 orders of magnitude.*

6.2 Empirical Comparison with Dominion

Dominion [2] is the state-of-the-art solver generator; it is based on the generic solver Minion [12]. Given a problem described in Dominion Input Language (DIL), Dominion takes four steps: (1) it selects potentially relevant components (e.g., optimizers and search strategies) from Minion's component library; (2) it randomly creates a solver to include some selected components; (3) it mutates the solver by adding/removing one component at a time, tests each mutant with constraint-solving tasks, and adjusts mutation

Table 5: The solver generation time by SoGen and Dominion (second)

Id	Solver Generation by SoGen	Solver Generation by Dominion										
		1	2	3	4	5	6	7	8	9	10	Sum
Q1	<0.001	3.42	4.92	2.13	3.42	8.35	6.46	8.76	7.13	2.42	2.56	49.57
Q2	<0.001	9.42	13.24	8.93	25.63	11.72	33.87	6.85	5.56	19.46	14.92	149.60
Q3	<0.001	3.56	11.72	19.63	29.45	11.81	14.62	17.93	15.82	22.8	27.21	174.55

Table 6: The constraint-solving time for each task (T1–T15) by SoGen's solvers and Dominion's fastest solvers (second)

	T1	T2	T3	T4	T5	T6	T7	T8	T9	T10	T11	T12	T13	T14	T15
Solvers by SoGen	0.02	0.02	0.14	0.36	0.55	0.02	0.04	0.16	2.90	26.51	16.42	47.78	134.21	784.64	1061.12
The Best Solvers by Dominion	0.32	0.53	0.80	3.29	32.93	1.03	18.42	94.92	1192.32	>1800	70.83	452.98	1642.93	>1800	>1800

accordingly until finding a best solver; (4) it repeats (2) and (3) for 10 times and creates 10 solvers.

6.2.1 Dataset. To understand how SoGen compares with Dominion, we did a comparative experiment by applying both tools to the same dataset. Due to the difficulty of DIL usage, we were unable to use DIL to describe all nine problems mentioned in the dataset shown in Section 6.1.1. Therefore, we created a second dataset by referring to the problems mentioned in the Dominion paper [2]. Among the six problems discussed in that paper, we managed to express three problems in DIL. As a result, our new dataset includes 15 constraint-solving tasks related to 3 CPs:

- Q1. *N Queens*: See P7 in Table 3.
- Q2. *Golomb:* A Golomb ruler is defined as a set of N integers $0 = a_1 < a_2 < \ldots < a_N$ such that the $N(N-1)/2$ differences (i.e., $a_j - a_i, 1 \le i < j \le N$) are distinct. Find a ruler with the minimum length.
- Q3. *Non-Monochromatic Rectangles:* Color $N \times N$ grids with a fixed number of colors (i.e., C), such that there is no rectangle with all four corners to have the same color.

The complexities of Q2 and Q3 are separately $O(N^{2N})$ [11] and $O(C^{N^2})$, higher than those mentioned in Table 3. Thus, we set N to very small numbers when defining tasks: 8, 9, 10, 11, and 12.

6.2.2 Experiment Settings. We compared SoGen with Dominion in two aspects: (1) the speed of solver generation and (2) the efficiency of generated solvers.

Procedure. We modeled each CP with both PDL and DIL in semantically equivalent ways, and fed those models separately to SoGen and Dominion using the default settings. For each CP, SoGen generated a single solver while Dominion created 10 solvers. By applying each solver to every task three times, we recorded the solvers' costs and averaged the values among three runs.

6.2.3 Results. As shown in Table 5, SoGen generated solvers more efficiently than Dominion. Specifically, SoGen produced a single solver for each CP and the generation time is negligible (i.e., < 0.001 second). Meanwhile, Dominion generated 10 solvers for each CP. The generation time varies from 2.13 to 33.87 seconds, with the average as 12.46 seconds per solver. The overall solver generation time by Dominion for each CP is 49.57–174.55 seconds; as the 10 solvers created for each CP were optimized differently, some solvers resolved constraints faster than the others.

To compare the quality of generated solvers, we applied SoGen's solvers and Dominion's fastest solvers to the 15 tasks in our dataset. Namely, each task was resolved by two separate solvers, and we recorded the incurred time/memory costs. As shown in Table 6, SoGen's solvers fulfilled all 15 tasks, while Dominion's solvers only resolved 12 tasks. Among these 12 tasks, SoGen's solvers worked more efficiently and obtained 4x–593x speedup over Dominion's solvers. We also compared the memory costs. Among the 12 tasks successfully resolved by both types of solvers, SoGen's solvers used only 19%–60% of the memory space used by Dominion's.

Three reasons can explain SoGen's higher effectiveness and efficiency. First, SoGen synthesized solver programs from scratch instead of tailoring existing solver architectures; so its solvers incurred no runtime overhead for complex software infrastructures or poorly configured optimizations. Second, SoGen generates efficient solvers based on rigorous static reasoning of problem properties, while Dominion explores and validates each solver based on random search and dynamic solver execution. Third, SoGen can synthesize DP search algorithms and Dominion cannot do that. Therefore, our approach is more rigorous and efficient.

Finding 2: *SoGen outperformed Dominion by generating solvers with lower runtime overheads; its solvers also resolved more constraints by using less time and memory.*

7 DISCUSSION

PDL vs. Existing Domain-Specific Languages. PDL is similar to existing constraint modeling languages (e.g., MiniZinc [24] and DIL [1]), as it also supports users to describe inputs, decision variables, constraints, and (optionally) the optimization objective. However, we chose to define PDL instead of reusing existing languages, because we need users to also specify the *domains* of all inputs and decision variables. Such domain information is mandatory by SoGen, since SoGen relies on the info to reason about problem properties, and to use those properties for algorithm synthesis and optimization. Meanwhile, the domain information is optional in other languages; it is used by existing constraint solvers only for input validation. None of existing solvers characterizes Prop1-Prop4 or synthesizes DP search algorithms as SoGen does.

The Novelty of SoGen. As a solver generator, SoGen takes in CP models and outputs problem-specific optimized constraint solvers. On the other hand, existing constraint solvers (e.g., Gecode and Chuffed) take in CP models and output solutions to the described constraint-solving tasks. SoGen is unique in three aspects:

- It statically reasons about four problem properties: the search space, value ranges of subfunctions, the monotonicity of objective functions, and overlapping subproblems.
- Based on problem properties, SoGen automates the decision-making process to adopt two optimization strategies: branch pruning and dynamic programming.

- It automatically designs and implements dynamic programming algorithms.

SoGen's good performance is due to the novel approach design of unique property reasoning, and problem-specific C solver generation. We did not propose any new optimization technique. Instead, SoGen characterizes CPs from different angles, and automatically applies existing optimization techniques as needed to achieve high problem-solving efficiency. Due to the space and time limit, we did not measure the impact of different optimization strategies. We will analyze that in the future.

The Correctness of CP Models. We carefully did constraint modeling for all experimented CPs using MiniZinc (for Gecode and Chuffed), PDL (for SoGen), and DIL (for Dominion), in order to define correct models. As Gecode and Chuffed take in FlatZinc instead of MiniZinc directly, we used a standard third-party tool mzn2fzn [33] to translate models from MiniZinc to FlatZinc, and to ensure the translation correctness. As all experimented tools support declarative programming, we modeled every problem with different languages in semantically equivalent ways, and open-sourced all models at GitHub. We observed correct results by all solvers, which indicate the correctness of problem specifications.

8 THREATS TO VALIDITY

Threats to External Validity. All the findings and observations mentioned in this paper are based on our evaluation datasets. We believe our results to generalize well to unexplored CPs for two reasons. First, the experimented CPs are from undergraduate programming courses and CSPLib, so they are popularly used and representative. Second, the two optimization techniques automated are pruning and DP, which have been widely applied for search optimization. Thus, SoGen is likely to considerably accelerate the solving process for unexplored CPs. In the future, we plan to experiment with more CPs so as to explore the generalizability of our observations.

Threats to Construct Validity Although we had no difficulty writing models with MiniZinc and PDL, DIL seems quite different from other modeling languages and its documentation is not quite helpful. Consequently, we created a dataset by reading the Dominion paper [2] and modeling three CPs mentioned there with our best effort. Because (1) neither source code nor data of the Dominion paper is publicly available and (2) Dominion works nondeterministically, we do not guarantee that the solvers used in our evaluation are the best solvers that Dominion can generate.

9 RELATED WORK

The related work of this research includes optimizers of constraint solving and solver generators.

Optimizers of Constraint Solving. To overcome the scalability issue of constraint solving, researchers proposed a variety of methods to optimize the solving process [8, 16, 18, 25, 30]. For instance, caching [30] memoizes search states to prevent the same state from being recomputed. Subproblem dominance [8] reasons about the dominance relationship between states to reduce unnecessary state exploration. Lazy Clause Generation (LCG) [25] analyzes failures at backtracking points and derives new constraints (i.e., nogoods) to reduce the search space. Given a MiniZinc model, DPSolver [18] analyzes the model and checks whether the described problem has two properties: (1) optimal substructures and (2) overlapping subproblems. If so, DPSolver refactors the MiniZinc model such that Gecode solves the problem in a DP manner.

SoGen is similar to existing optimizers by automating both property characterization for CPs and decision-making for optimizations. However, SoGen focuses on a unique set of properties and conducts novel property reasoning. Instead of adding optimizers to existing solvers, SoGen creates solvers from CP models to eliminate the high overheads incurred by complex software architectures and to synthesize efficient solvers that perform DP search.

Solver Generators. Generic constraint solvers are usually provided as configurable "toolbox" systems. By manually configuring the system-provided search strategies and optimizers, users can tailor a generic solver for any CP. Because manual configuration is challenging and time-consuming for users, prior work [2, 13, 15, 22, 35, 36] proposed solver generators to automate configuration tuning. For example, given a problem specification and several training instances (i.e., constraint-solving tasks), MULTI-TAC adopts the backtracking schema to search for an optimized configuration among candidates, and evaluates each configuration based on those instances. Some approaches [13, 15, 35, 36] exploit machine learning to tune the parameters or select the solvers in an algorithm portfolio (i.e., alternative algorithms usable to solve the same problem).

Although existing techniques customize generic solvers for given problems to accelerate constraint solving, they cannot fully bypass the extra runtime or memory overheads introduced by the complex infrastructure of toolbox systems. Meanwhile, the time-consuming tuning procedure can be ineffective and inaccurate, making the generated solvers inefficient. In comparison, SoGen synthesizes solver algorithms from scratch, produces efficient and correct solvers in C, and thus incurs no infrastructure-related costs.

10 CONCLUSION

Existing constraint solvers cannot efficiently solve CPs. This paper presents SoGen—a novel approach of generating efficient solvers to alleviate the scalability issue of existing solvers. Different from the state-of-the-art solver generator, SoGen analyzes constraint models written in PDL to (1) identify controlling variables and minimize their value ranges, (2) compute value boundaries for any subfunction of constraint formulas, (3) check whether a given objective function is monotonic, and (4) decide whether an objective function has overlapping subproblems. With its novel and static property characterization, SoGen synthesizes optimized solvers that significantly outperformed existing ones in most scenarios. We recently also applied SoGen to the context of CS education: we provided our PDL and tool as a scaffolding technique that intends to help students program for COPs, and observed very impressive results [19]. In the future, we will evaluate SoGen with more CPs and improve it to characterize more properties.

ACKNOWLEDGMENTS

This work was supported by Science and Technology Innovation 2030 No. 2018AAA0100901 of China, Baidu Foundation No. 2020BD003 of Peking University, and National Science Foundation No. 1845446 of USA. We thank reviewers and Dr. Yingfei Xiong for their valuable feedback. We also thank Mr. Qinjian Zhang for providing experiment environment.

REFERENCES

[1] Dharini Balasubramaniam, Lakshitha De, Chris Jefferson, Lars Kotthoff, Ian Miguel, and Peter Nightingale. 2011. Dominion: An Architecture-Driven Approach to Generating Efficient Constraint Solvers. *Proceedings - 9th Working IEEE/IFIP Conference on Software Architecture, WICSA 2011* (06 2011). https://doi.org/10.1109/WICSA.2011.37

[2] Dharini Balasubramaniam, Christopher Jefferson, Lars Kotthoff, Ian Miguel, and Peter Nightingale. 2012. An automated approach to generating efficient constraint solvers. In *2012 34th International Conference on Software Engineering (ICSE)*. 661–671. https://doi.org/10.1109/ICSE.2012.6227151

[3] Pietro Belotti, Sonia Cafieri, Jon Lee, and Leo Liberti. 2010. Feasibility-based bounds tightening via fixed points. In *International Conference on Combinatorial Optimization and Applications*. Springer, 65–76. https://doi.org/10.1007/978-3-642-17458-2_7

[4] Nikolaj Bjørner, Leonardo de Moura, Lev Nachmanson, and Christoph Wintersteiger. 2021. Programming Z3. http://theory.stanford.edu/~nikolaj/programmingz3.html.

[5] Miquel Bofill, Josep Suy, and Mateu Villaret. 2010. A System for Solving Constraint Satisfaction Problems with SMT. In *Theory and Applications of Satisfiability Testing – SAT 2010*, Ofer Strichman and Stefan Szeider (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 300–305. https://doi.org/10.1007/978-3-642-14186-7_25

[6] Choco 2020. Choco-solver. https://choco-solver.org/.

[7] Geoffrey Chu, Maria Garcia De La Banda, and Peter J. Stuckey. 2010. Automatically Exploiting Subproblem Equivalence in Constraint Programming. In *Proceedings of the 7th international conference on Integration of AI and OR Techniques in Constraint Programming for Combinatorial Optimization Problems*. https://doi.org/10.1007/978-3-642-13520-0_10

[8] Geoffrey Chu, Maria Garcia De La Banda, and Peter J. Stuckey. 2012. Exploiting subproblem dominance in constraint programming. *Constraints* 17, 1 (2012), 1–38. https://doi.org/10.1007/s10601-011-9112-9

[9] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. 2009. *Introduction to Algorithms, third edition*. MIT Press.

[10] CSPLib 2020. CSPLib: A problem library for constraints. http://csplib.org/.

[11] Apostolos Dimitromanolakis. 2002. *Analysis Of The Golomb Ruler And The Sidon Set Problems And Determination Of Large Near-Optimal Golomb Rulers*. Technical Report.

[12] Ian Philip Gent, Christopher Jefferson, and Ian Miguel. 2006. MINION: A Fast, Scalable, Constraint Solver. In *ECAI 2006, 17th European Conference on Artificial Intelligence, August 29 - September 1, 2006, Riva del Garda, Italy, Including Prestigious Applications of Intelligent Systems (PAIS 2006), Proceedings*.

[13] Carla P. Gomes and Bart Selman. 2001. Algorithm portfolios. *Artificial Intelligence* 126, 1-2 (2001), 43–62. https://doi.org/10.1016/S0004-3702(00)00081-3

[14] JTB 2020. Java Tree Builder. http://compilers.cs.ucla.edu/jtb/.

[15] Ashiqur Khudabukhsh, Lin Xu, Holger Hoos, and Kevin Leyton-Brown. 2009. SATenstein: Automatically Building Local Search SAT Solvers from Components. *IJCAI International Joint Conference on Artificial Intelligence* 232, 517–524. https://doi.org/10.1016/j.artint.2015.11.002

[16] Matthew Kitching and Fahiem Bacchus. 2007. Symmetric Component Caching. In *IJCAI*. 118–124.

[17] Viswanathan Kodaganallur. 2004. Incorporating language processing into Java applications: a JavaCC tutorial. *IEEE Software* 21, 4 (July 2004), 70–77. https://doi.org/10.1109/MS.2004.16

[18] Shu Lin, Na Meng, and Wenxin Li. 2019. Optimizing Constraint Solving via Dynamic Programming. In *Proceedings of the 28th International Joint Conference on Artificial Intelligence* (Macao, China) *(IJCAI'19)*. AAAI Press, 1146–1154. https://doi.org/10.24963/ijcai.2019/160

[19] Shu Lin, Na Meng, and Wenxin Li. 2021. PDL: Scaffolding Problem Solving in Programming Courses. In *Proceedings of the 2021 Conference on Innovation & Technology in Computer Science Education*. https://doi.org/10.1145/3430665.3456360

[20] László Lovász. 2007. *Combinatorial Problems and Exercises*. AMS Chelsea Pub., Providence, RI.

[21] Roberto Castañeda Lozano, Mats Carlsson, Gabriel Hjort Blindell, and Christian Schulte. 2016. Register Allocation and Instruction Scheduling in Unison. In *Proceedings of the 25th International Conference on Compiler Construction* (Barcelona, Spain) *(CC 2016)*. ACM, New York, NY, USA, 263–264. https://doi.org/10.1145/2892208.2892237

[22] Steven Minton. 1996. Automatically Configuring Constraint Satisfaction Programs: A Case Study. *Constraints* 1, 1-2 (1996), 7–43. https://doi.org/10.1007/BF00143877

[23] Leonardo De Moura and Nikolaj Bjrner. 2008. Z3: an efficient SMT solver. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*. https://doi.org/10.1007/978-3-540-78800-3_24

[24] Nicholas Nethercote, Peter J. Stuckey, Ralph Becket, Sebastian Brand, Gregory J. Duck, and Guido Tack. 2007. MiniZinc: Towards a Standard CP Modelling Language. In *Proceedings of the 13th International Conference on Principles and Practice of Constraint Programming* (Providence, RI, USA) *(CP'07)*. Springer-Verlag, Berlin, Heidelberg, 529–543. https://doi.org/10.1007/978-3-540-74970-7_38

[25] Olga Ohrimenko, Peter J. Stuckey, and Michael Codish. 2009. Propagation via lazy clause generation. *Constraints* 14, 3 (2009), 357–391. https://doi.org/10.1007/s10601-008-9064-x

[26] Jean-Francois Puget. 2004. Constraint Programming Next Challenge: Simplicity of Use. In *Principles and Practice of Constraint Programming – CP 2004*, Mark Wallace (Ed.). Springer Berlin Heidelberg, Berlin, Heidelberg, 5–8. https://doi.org/10.1007/978-3-540-30201-8_2

[27] Tuomas Sandholm, Subhash Suri, Andrew Gilpin, and David Levine. 2002. Winner Determination in Combinatorial Auction Generalizations. In *Proceedings of the First International Joint Conference on Autonomous Agents and Multiagent Systems: Part 1* (Bologna, Italy) *(AAMAS '02)*. ACM, New York, NY, USA, 69–76. https://doi.org/10.1145/544741.544760

[28] Christian Schulte, Mikael Z. Lagerkvist, and Guido Tack. 2006. Gecode: Generic constraint development environment. In *INFORMS Annual Meeting*.

[29] Christian Schulte, Guido Tack, and Mikael Z. Lagerkvist. 2019. Introduction. In *Modeling and Programming with Gecode*, Christian Schulte, Guido Tack, and Mikael Z. Lagerkvist (Eds.). Corresponds to Gecode 6.2.0.

[30] Barbara M. Smith. 2005. Caching Search States in Permutation Problems. In *Principles and Practice of Constraint Programming - CP 2005*, Peter van Beek (Ed.). Springer Berlin Heidelberg, Berlin, Heidelberg, 637–651. https://doi.org/10.1007/11564751_47

[32] Peter J. Stuckey, Kim Marriott, and Guido Tack. 2020. FlatZinc and Flattening. https://www.minizinc.org/doc-2.5.2/en/flattening.html.

[33] Peter J. Stuckey, Kim Marriott, and Guido Tack. 2020. The MiniZinc IDE. https://www.minizinc.org/doc-2.5.2/en/minizinc_ide.html.

[31] Peter J. Stuckey, Kim Marriott, and Guido Tack. 2021. Solving Technologies and Solver Backends. https://www.minizinc.org/doc-2.4.3/en/solvers.html.

[34] Charles Turner. 2014. *Comparing SAT and SMT Encodings of All-Different and Global Cardinality Constraints*. Master's thesis. University of York.

[35] Lin Xu, Holger H. Hoos, and Kevin Leyton-Brown. 2010. Hydra: Automatically Configuring Algorithms for Portfolio-Based Selection. In *Proceedings of the Twenty-Fourth AAAI Conference on Artificial Intelligence, AAAI 2010, Atlanta, Georgia, USA, July 11-15, 2010*.

[36] Lin Xu, Frank Hutter, Holger H. Hoos, and Kevin Leyton-Brown. 2011. SATzilla: Portfolio-based Algorithm Selection for SAT. *Journal of Artificial Intelligence Research* 32, 1 (2011), 565–606.

[37] Neng-Fa Zhou, Masato Tsuru, and Eitaku Nobuyama. 2012. A Comparison of CP, IP, and SAT Solvers through a Common Interface. In *2012 IEEE 24th International Conference on Tools with Artificial Intelligence*, Vol. 1. 41–48. https://doi.org/10.1109/ICTAI.2012.15