Optimizing Constraint Solving via Dynamic Programming

Shu Lin[1*], **Na Meng**[2] and **Wenxin Li**[1]
[1]Department of Computer Science and Technology, Peking University, Beijing, China
[2]Department of Computer Science, Virginia Tech, Blacksburg, VA, USA
fzlinshu@pku.edu.cn, nm8247@vt.edu, lwx@pku.edu.cn

Abstract

Constraint optimization problems (COP) on finite domains are typically solved via search. Many problems (e.g., 0-1 knapsack) involve redundant search, making a general constraint solver revisit the same subproblems again and again. Existing approaches use caching, symmetry breaking, subproblem dominance, or search with decomposition to prune the search space of constraint problems. In this paper we present a different approach—$DPSolver$—which uses dynamic programming (DP) to efficiently solve certain types of constraint optimization problems (COPs). Given a COP modeled with MiniZinc, $DPSolver$ first analyzes the model to decide whether the problem is efficiently solvable with DP. If so, $DPSolver$ refactors the constraints and objective functions to model the problem as a DP problem. Finally, $DPSolver$ feeds the refactored model to Gecode—a widely used constraint solver—for the optimal solution. Our evaluation shows that $DPSolver$ significantly improves the performance of constraint solving.

1 Introduction

When solving a constraint optimization problem (COP), a general solver searches for an assignment of variables to (1) satisfy the constraints on those variables and (2) optimize an objective function. Such search may require repetitive computation when different branches on a search tree lead to the same subproblem. The unnecessary redundant work can cause the worst case complexity of search to be $O(M^N)$, where N is the number of variables for assignments and M is the number of branches at each node.

Existing methods avoid or reduce redundant search via caching [Smith, 2005], symmetry breaking [Gent *et al.*, 2006], subproblem dominance [Chu *et al.*, 2012], problem decomposition [Kitching and Bacchus, 2007], branch-and-bound pruning [Marinescu and Dechter, 2005], lazy clause generation [Ohrimenko *et al.*, 2009], or auto-tabling [Dekker *et al.*, 2017; Zhou *et al.*, 2015].

*Contact Author

Dynamic programming (DP) is a classical method for solving complex problems. Given a problem, DP decomposes it into simpler subproblems, solves each subproblem once, stores their solutions with a table, and conducts table lookup when a subproblem reoccurs [Bertsekas, 2000]. Although DP seems a nice search algorithm for COP solutions, we have not seen it to be used in solving general constraint models. Therefore, this paper explores how well DP helps optimize constraint solving. There are three major research challenges:

- Given a COP, how can we automatically decide whether the problem is efficiently solvable with DP?
- If a problem can be solved with DP, how can we implement the DP search?
- When there are multiple ways to conduct DP search for a problem, how can we automatically choose the best one with optimal search performance?

To address these challenges, we designed and implemented a novel approach $DPSolver$, which opportunistically accelerates constraint solving in a non-intrusive way. Specifically, given a COP described in MiniZinc—a widely used constraint modeling language [Nethercote *et al.*, 2007], $DPSolver$ determines whether the problem has (1) optimal substructures and (2) overlapping subproblems; if so, the problem is efficiently solvable with DP. Next, for each solvable problem, $DPSolver$ converts the original model to a DP-oriented model, such that a general constraint solver (i.e., Gecode [Schulte *et al.*, 2006]) essentially conducts DP search when processing the new model. Third, if multiple DP-oriented models are feasible, $DPSolver$ estimates the computation complexity of each model to choose the fastest one.

We applied $DPSolver$ to nine optimization problems, including seven DP problems and two non-DP ones. $DPSolver$ significantly speeded up the constraint solving process for all problems. We also applied two state-of-the-art optimized constraint solvers—CHUFFEDC (caching) and CHUFFEDL (LCG)—to the same dataset for comparison, and observed that $DPSolver$ significantly outperformed both techniques. We open sourced $DPSolver$ and our evaluation data set at https://github.com/fzlinshu/DPSolver.

```
% Input arguments
    int: N;
    int: C;
    array[1..N] of int: V;
    array[1..N] of int: W;

% Variables
    var set of 1..N: knapsack;

% Constraints
    constraint sum (i in knapsack)(W[i]) <= C;

% Objective function
    solve maximize sum (i in knapsack)(V[i]);
```

Figure 1: Model of the 0-1 knapsack problem

2 A Motivating Example

To facilitate discussion, we introduce the 0-1 knapsack problem as an exemplar problem efficiently solvable with DP.

> **Problem Description:** There are N items. There is a knapsack of capacity C. The i^{th} item ($i \in [1, N]$) has value V_i and weight W_i. Put a set of items $S \subseteq \{1, \ldots, N\}$ in the knapsack, such that the sum of the weights is at most C while the sum of values is maximal.

The 0-1 knapsack problem is a typical COP and can be modeled with MiniZinc in the following way (see Figure 1):
Given the above model description, a general constraint solver (e.g., Gecode) typically enumerates all possible subsets of N to find the maximum value summation. Such naïve search has $O(2^N)$ time complexity. Our research intends to significantly reduce this complexity.

3 Approach

$DPSolver$ consists of three phases: DP problem recognition (Section 3.1), DP-oriented model description generation (Section 3.2), and description selection (Section 3.3). Phase II is taken only when Phase I identifies one or more solvable problems in a given MiniZinc model; and Phase III is taken only when Phase II generates multiple alternative models.

3.1 Phase I: DP Problem Recognition

If a problem can be efficiently solved with dynamic programming, it must have two properties [Cormen *et al.*, 2009]:

P1. Optimal Substructures. An optimal solution can be constructed from optimal solutions of its subproblems. This property ensures the usability of DP, because DP saves and uses only the optimal instead of all solutions to subproblems for optima calculation.

P2. Overlapping Subproblems. When a problem is decomposed into subproblems, some subproblems are repetitively solved. This property ensures the usefulness of DP, because by memoizing solutions to subproblems, DP can eliminate repetitive computation.

Essentially, DP is applicable to a COP when optimal solutions to subproblems can be repetitively reused for optima calculation. If a COP has both properties, we name it a ***DP problem***.

Given a COP modeled in MiniZinc, $DPSolver$ recognizes a DP problem by taking three steps: 1) identifying any array variable and accumulative function applied to those array elements, 2) converting constraints and objective functions to recursive functions of array elements (i.e., subproblems), and 3) checking recursive functions for the two properties.

Step 1: Candidate Identification

$DPSolver$ checks for any declared variable of the array data type because DP is usually applied to arrays. Additionally, if a variable is a set, as shown by the `knapsack` variable in Figure 1, $DPSolver$ converts it to a boolean array `b` such that "`b[i]`" indicates whether the i^{th} item in the set is chosen or not. By doing so, $DPSolver$ can also handle problems with set variables. We name the identified array or set variables ***candidate variables***.

Next, $DPSolver$ checks whether any candidate variable is used in at least one constraint and one objective function; if so, $DPSolver$ may find optimization opportunities when enumerating all value assignments to the variable. In Figure 1, both the constraint and objective can be treated as functions of the newly created array b as below:

$$\sum_{i=1}^{N} b[i] * W[i] \leq C, \text{ where } b[i] \in \{0, 1\} \tag{3.1}$$

$$maximize \sum_{i=1}^{N} b[i] * V[i], \text{ where } b[i] \in \{0, 1\} \tag{3.2}$$

When these functions have any accumulative operator (e.g., sum), it is feasible to further break down the problem into subproblems. Thus, $DPSolver$ treats the variable `b` together with related functions as a candidate for DP application.

Step 2: Function Conversion

$DPSolver$ tentatively converts relevant constraint and objective functions to step-wise recursive functions in order to identify subproblems and prepare for further property check. Specifically, $DPSolver$ unfolds all accumulative functions recursively. For instance, the constraint formula (3.1) can be converted to

$$\begin{aligned} f_0(W, b) &= 0 \\ f_1(W, b) &= f_0(W, b) + b[1] * W[1] \\ c_1(W, b) &= C - f_1(W, b) \\ c_1(W, b) &\geq 0 \\ &\ldots \\ f_N(W, b) &= f_{N-1}(W, b) + b[N] * W[N] \\ c_N(W, b) &= C - f_N(W, b) \\ c_N(W, b) &\geq 0 \end{aligned} \tag{3.3}$$

Here, $f_i(W, b)(i \in [1, N])$ computes weight sums for the first i items given (1) the item weight array W and (2) the boolean array b. The function $c_i(W, b)$ subtracts $f_i(W, b)$ from capacity C, to define the constraint for each subproblem. Here, c_i means "*the remaining capacity limit for the last*

$(N-i)$ *items*". The value of c_i helps decide whether an item should be added to the knapsack. Namely, if the weight of the $(i+1)^{th}$ item is greater than c_i's value, the item should not be added. Actually, the c_i's value is limited by the lower bound $min\{0, (N-i) \cdot min\{W[1], \ldots, W[N]\}\}$. Because $DPSolver$ scanned the input data and found all weight values to be greater than 0, the lower bound used in (3.3) was simplified to 0.

Similarly, the objective function (3.2) can be transferred to

$$\begin{aligned} &o_0(V,b) = 0 \\ &o_1(V,b) = o_0(V,b) + b[1] * V[1] \\ &opt_1(V) = max\ o_1(V,b) \\ &\quad \ldots \\ &o_N(V,b) = o_{N-1}(V,b) + b[N] * V[N] \\ &opt_N(V) = max\ o_N(V,b) \end{aligned} \tag{3.4}$$

Here, $o_N(V,b)$ calculates the value summation. The function $opt_i(V)$ defines the optimization goal for each subproblem related to the first i items. We use max instead of $maximize$ to indicate that the maximal value can be calculated by simply enumerating all possible values of variables without using any optimization or advanced algorithm such as search. When all possible value assignments are explored for b, $o_N(V,b)$ and $opt_N(V)$ functions can be executed to obtain the maximal summation.

Step 3: Property Check

With the converted functions, $DPSolver$ checks for two properties in sequence.

(a) *Verifying optimal substructures.* We first defined and proved the following theorem:

Theorem 3.1. *Given two sets of functions, $O = \{o_0(\cdot), o_1(\cdot), o_2(\cdot), \ldots, o_n(\cdot)\}$ and $Opt = \{opt_1(\cdot), opt_2(\cdot), \ldots, opt_n(\cdot)\}$ ("$\cdot$" is a placeholder for arguments), for any $i \in [1,n]$, suppose that*

- *$o_i(\cdot) = h(o_{i-1}(\cdot), b[i])$ where b is a candidate variable and h is monotonically increasing in $o_{i-1}(\cdot)$,*
- *$opt_i(\cdot) = max\ o_i(\cdot)$.*

Then $opt_i(\cdot)$ is monotonically increasing in $opt_{i-1}(\cdot)$.

In this theorem, each $o_i(.)$ is a function, representing all possible value sums produced when $b[1..i]$ is assigned with different vector values, while $max\ o_i(.)$ is a value, representing the maximum among those value sums.

Proof. For any $i \in [1,n]$,

$$\begin{aligned} &\because o_{i-1}(\cdot) \leq max\ o_{i-1}(\cdot) \\ &\therefore h(o_{i-1}(\cdot), b[i]) \leq h(max\ o_{i-1}(\cdot), b[i]) \\ &\therefore o_i(\cdot) \leq h(max\ o_{i-1}(\cdot), b[i]) \\ &\therefore max\ o_i(\cdot) = h(max\ o_{i-1}(\cdot), b[i]), i.e., \\ &opt_i(\cdot) = h(opt_{i-1}(\cdot), b[i]) \end{aligned}$$

Therefore, $opt_i(\cdot)$ monotonically increases in $opt_{i-1}(\cdot)$. The optimal solution can be composed with the optimal solution to a subproblem. ■

Similarly, we defined and proved a related theorem when the max function used in Theorem 3.1 is replaced with min. Furthermore, there are problems whose $opt_i(\cdot)$ functions are expressions of $max\ o_i$ (e.g., $max\ o_i + 3$) instead of $max\ o_i$ itself. To ensure the generalizability of our approach, we also consider such problems to have optimal substructures as long as $max\ o_i$ is a function of $max\ o_{i-1}$. This is because when the ***extreme value*** (max or min) related to a problem's optimal solution can be computed with the extreme values derived for subproblems, we can always construct the optimal solution by reusing extreme values from subproblems.

Based on the above theorems, given converted objective functions, $DPSolver$ locates the used max or min function and tentatively matches h. For each matched h function, $DPSolver$ takes the derivative to check for any monotonicity property. For our example (function sets (3.4)), we have

$$h_{0/1}(o_{i-1}(\cdot)) = o_{i-1}(\cdot) + b[i] * V[i].$$

Assuming $h_{0/1}$ to be continuous, $DPSolver$ finds the derivative to be $\partial o_i(\cdot)/\partial o_{i-1}(\cdot) = 1 > 0$. Therefore, $h_{0/1}$ increases monotonically in o_{i-1}. Solutions of the 0-1 knapsack problem have optimal substructures.

(b) *Verifying overlapping subproblems.* We defined and proved another theorem to facilitate property checking.

Theorem 3.2. *Given two sets of functions, $F = \{f_0(\cdot), f_1(\cdot), \ldots, f_n(\cdot)\}$ and $Con = \{c_0(\cdot), c_1(\cdot), \ldots, c_n(\cdot)\}$ ("$\cdot$" is a placeholder for arguments), for any $i \in [1,n]$, suppose that*

- *$f_0(\cdot) = v_0$ where v_0 is a constant,*
- *$f_i(\cdot) = p(f_{i-1}(\cdot), b[i])$ where b is a variable, and*
- *$c_i(\cdot) = q(f_{i-1}(\cdot), b[i])$.*

Then there exist overlapping subproblems of $f_n(\cdot)$ and $c_n(\cdot)$ between different value assignments of b.

Proof. This theorem includes two parts: (1) $f_n(\cdot)$ has overlapping subproblems; and (2) $c_n(\cdot)$ has overlapping subproblems. Here we demonstrate the proof by induction for $f_n(\cdot)$. The proof for $c_n(\cdot)$ is similar.

1. **n=2:** For any two assignments of b: b'_{A2} and b''_{A2}, where $b'_{A2} \neq b''_{A2}$ but $b'_{A2}[1] = b''_{A2}[1]$,

   $\because$ The first elements of both arrays are identical,

   $\therefore$ The evaluation procedures of $f_1(\cdot) = p(f_0(\cdot), b[1])$ remains the same given b'_{A2} and b''_{A2}.

   $\therefore$ When $f_2(\cdot)$ is computed based on $f_1(\cdot)$, the calculation procedure of $f_1(\cdot)$ overlaps between b'_{A2} and b''_{A2}. Thus, $f_n(\cdot)$ has overlapping subproblems.

2. **n=k (k>2):** Assume the theorem to hold. Namely, there exist two assignments b'_{Ak} and b''_{Ak} ($b'_{Ak} \neq b''_{Ak}$), between which the evaluation procedures of $f_n(\cdot)$ have overlapping subproblems.

3. **n=k+1:** To prove the theorem, we compose two assignments: $b'_{A(k+1)}$ and $b''_{A(k+1)}$, such that $b'_{A(k+1)}[1:k] =$

b'_{Ak} and $b''_{A(k+1)}[1:k] = b''_{Ak}$. Intuitively, these arrays separately include b'_{Ak} and b''_{Ak} as the first k elements.

$\because f_{k+1}(\cdot) = p(f_k(\cdot), b[k+1])$

$\therefore$ The function depends on the evaluation result of $f_k(\cdot)$, a function computed based on the first k elements

$\because$ The evaluation processes of $f_k(\cdot)$ between b'_{Ak} and b''_{Ak} have overlapping subproblems

$\therefore$ The evaluation processes of $f_{k+1}(\cdot)$ between $b_{A(k+1)'}$ and $b''_{A(k+1)}$ also overlap.

Therefore, when different value assignments of b are explored, there are overlapping subproblems to resolve. ■

Based on Theorem 3.2, given converted constraint functions, $DPSolver$ tries to match p and q by using or unfolding the formulas of f and c functions. Thus, for our example (function sets (3.3)), the matched functions are

$$p_{0/1}(f_{i-1}(\cdot), b[i]) = f_{i-1}(\cdot) + b[i] * W[i],$$
$$q_{0/1}(f_{i-1}(\cdot), b[i]) = C - f_{i-1}(\cdot) - b[i] * W[i].$$

Notice that $p_{0/1}$ is derived from the f formulas, while $q_{0/1}$ is obtained when $DPSolver$ replaces the occurrence of f in c formulas. With such matched functions found, the 0-1 knapsack problem passes the property check.

DP trades space for time by storing and reusing optimal solutions to subproblems. If a problem has the first property only, $DPSolver$ does not proceed to Phase II. Because there is no reuse of intermediate results, the extra space consumption for caching those results will not bring any execution speedup. However, if a problem has the second property only, even though DP is not applicable, $DPSolver$ still creates a table to memoize *all* intermediate results for data reuse and runtime overhead reduction.

3.2 Phase II: DP-Oriented Model Generation

The above-mentioned recursive functions (e.g., function sets (3.3) and (3.4)) are not directly usable by DP for efficient search because no redundant computation is eliminated. We need to rewrite those functions such that intermediate results for subproblems are stored to a table and are used to replace repetitive computation.

Specifically, given (1) a candidate variable b, (2) a series of constraint functions $c(\cdot)$, (3) a series of objective functions $o(\cdot)$, and (4) the extreme value we care about (i.e., max or min), $DPSolver$ creates a two-dimension array M such that

$$\begin{aligned} M[i, c_i] &= extreme\ o_i(\cdot) \\ &= extreme\ h(extreme\ o_{i-1}(\cdot), b[i]) \\ &= extreme\ h(M[i-1, c_{i-1}], b[i]) \end{aligned} \quad (3.5)$$

In M, i corresponds to the array index range of b, c_i represents the corresponding valid constraint value, while the cell $M[i, c_i]$ saves the extreme value computed for each subproblem, meaning "*given the first i elements and the constraint value c_i, what is the extreme value of o_i?*". When different values of $b[i]$ are enumerated, the corresponding c_{i-1} can be different. Thus, multiple cells in the row $M[i-1]$ may be reused for computation, and $M[i, c_i]$ is usually decided by the value comparison between these cells. By generating such table-driven recursive functions to model a DP problem, $DPSolver$ can produce a DP-oriented MiniZinc description.

```
% Input arguments
    ... % unchanged N, C, V, W

% Variables
    array[0..N, 0..C] of var int: dpvalue;

% Constraints
    constraint forall(j in 0..C)
      (dpvalue[0,j] = if j==0 then 0 else -1 endif);
    function var int: calcValue(int: i,int: j,int: k) =
      if j-W[i]*k>=0 then
        if dpvalue[i-1,j-W[i]*k] != -1 then
          dpvalue[i-1,j-W[i]*k]+V[i]*k
        else -1 endif
      else -1 endif;
    constraint
      forall(i in 1..N, j in 0..C) (
        dpvalue[i,j] = max(k in 0..1)(calcValue(i,j,k))
      );

% Objective function
    solve maximize max(j in 0..C)(dpvalue[N,j]);
```

Figure 2: DP-oriented model of the 0-1 knapsack problem

Figure 2 shows the newly generated model for our example. The space complexity of creating the $dpvalue$ table is $O(NC)$, while the time complexity is $O(NC)$. This time complexity refers to the whole resolution process because the DP-oriented model is resolved by propagation only.

Handling of Side Constraints. Some COP models are defined with *side constraints* in addition to accumulative functions. $DPSolver$ can still process such models by converting each side constraint to an `if`-clause. For instance, a variant of the 0-1 knapsack problem can have an additional requirement as "*Item 3 must be put in the knapsack*", which can be expressed as a side constraint "`3 in knapsack`". For this side constraint, $DPSolver$ generates an `if`-clause "`if i==3 /\ k!=1 then -1 else ...`" and inserts it to the beginning of the `calcValue(...)` function in Figure 2.

3.3 Phase III: DP-Oriented Model Selection

For some COP problems, $DPSolver$ can create multiple alternative optimization strategies. Given the space limit (e.g., 4GB) specified by users when they run the tool via commands, $DPSolver$ automatically estimates the time/space complexity of each alternative, chooses the models whose table sizes are within the space limit, and then recommends the model with the maximum speedup.

To better explain this phase, we take the nested 0-1 knapsack problem as another exemplar DP problem (see Figure 3).

When analyzing the problem, $DPSolver$ identifies two candidate variables—b_1 and b_2—to separately represent subsets $K1$ and $K2$. Since both variables are related to constraints, $DPSolver$ explores and assesses three potential ways to create DP-oriented models:

(1) Optimization based on b_2: When exhaustive search is

```
% Input arguments
    int: N1, N2, C1;
    array[1..N1] of int: V1, W1;
    array[1..N2] of int: V2, W2;

% Variables
    var int: C2;
    var set of 1..N1: K1;
    var set of 1..N2: K2;

% Constraints
    constraint sum (i in K1)(W1[i]) <= C1;
    constraint C2 = sum (i in K1)(V1[i]);
    constraint sum (i in K2)(W2[i]) <= C2;

% Objective function
    solve maximize sum (i in K2)(V2[i]);
```

Figure 3: Model of the nested 0-1 knapsack problem. There are two sets of items ($N1$ and $N2$). Choose items in both sets to separately fill two knapsacks: $K1$ and $K2$. If the value sum in $K1$ is used as the capacity of $K2$, try to maximize the value sum in $K2$.

leveraged to fill $K1$ in various ways, for each produced $C2$ value, the optimization problem becomes a regular 0-1 knapsack problem—a DP problem. Therefore, $DPSolver$ can create a DP-oriented model, with both time and space complexity as $O(N2C2)$. Since the exhaustive search handles such 0-1 knapsack problems with $O(2^{N2})$ time complexity, the optimization speedup is estimated as $(2^{N2}/N2C2)$.

(2) Optimization based on b_1: When exhaustive search is used to fill $K2$ in various ways, for each generated weight sum sum_w in $K2$, we need the value sum in $K1$ (i.e., C2) to be no less than sum_w while the weight sum in $K1$ to be no more than $C1$. Because these converted problems do not have any objective function, they are not considered as DP problems. $DPSolver$ does not create any model for optimization.

(3) Optimization based on b_1 and b_2: $DPSolver$ concatenates b_1 and b_2 to create a larger array b_3. In b_3, the first $N1$ elements are involved in the first two constraints, while the last $N2$ elements are related to the third constraint and objective function. Thus, b_3 is still a candidate variable and the given problem is a DP problem. The exhaustive search obtains $O(2^{N1}2^{N2})$ time complexity. In comparison, DP search has both time and space complexity as $O((N1 + N2)C1Sum^2)$ time complexity, where Sum is the upper bound of sum_w. Consequently, the estimated speedup is $(2^{N1}2^{N2}/\left[(N1 + N2)C1Sum^2\right])$.

When the two separately generated models described in (1) and (3) both have their tables smaller than the space limit, $DPSolver$ suggests the one with more speedup (i.e., (3)). Figure 4 shows the DP-oriented model resulted from (3). Intuitively, the more array variables are involved in optimization (e.g., $(b1, b2)$ vs. $b2$), the more performance gain $DPSolver$ is likely to obtain.

```
% Input arguments
    ... % unchanged N1, N2, C1, V1, W1, V2, W2
    int: sum = 20;
    % the upper bound of sum_w obtained from the input

% Variables
    array[0..N1+N2,0..C1,0..sum,0..sum] of var int:
      dpvalue;

% Constraints
    constraint
      forall(i2 in 0..C1, i3 in 0..sum, i4 in 0..sum) (
          dpvalue[0,i2,i3,i4] =
            if i2==0/\i3==0/\i4==0 then 0 else -1 endif
      );
    function var int: calcValue(int:i1,int:i2,int:i3,
                                int:i4,int:i5) =
      if i1 <= N1 then
        if i2-W1[i1]*i5>=0 /\ i3-V1[i1]*i5>=0 then
          dpvalue[i1-1,i2-W1[i1]*i5,i3-V1[i1]*i5,i4]
        else -1 endif
      elseif i4<=i3 /\ i4-W2[i1-N1]*i5>=0 then
        if dpvalue[i1-1,i2,i3,i4-W2[i1-N1]*i5]!=-1 then
          dpvalue[i1-1,i2,i3,i4-W2[i1-N1]*i5]
            +V2[i1-N1]*i5
        else -1 endif
      else -1 endif;
    constraint
      forall(i1 in 1..N1+N2, i2 in 0..C1,
             i3 in 0..sum, i4 in 0..sum) (
        dpvalue[i1,i2,i3,i4] =
          max(i5 in 0..1)(calcValue(i1,i2,i3,i4,i5))
      );

% Objective function
    solve maximize
      max(i2 in 0..C1, i3 in 0..sum, i4 in 0..sum)
         (dpvalue[N1+N2,i2,i3,i4]);
```

Figure 4: DP-oriented model of the nested 0-1 knapsack problem when the optimization is based on both $b1$ and $b2$

4 Implementation

$DPSolver$ is implemented based on a widely used open-source constraint solver Gecode [Schulte *et al.*, 2006]. Given a user-provided MiniZinc model, $DPSolver$ analyzes the problem, converts the model to a DP-oriented model when possible, and passes the model to Gecode. If $DPSolver$ cannot optimize a model, it passes the original model to Gecode. If multiple alternative DP-oriented models are generated, $DPSolver$ passes the best one.

For some problems, $DPSolver$ implements extra handling to optimize constraint solving as much as possible.

Problems with aftereffects. Some problems (e.g., longest increasing subsequence) have the ***aftereffect property***. Namely, the value of $o_i(\cdot)$ not only depends on $o_{i-i}(\cdot)$ or $b[i]$, but also on elements in $b[1 : i-1]$ (e.g., $b[i-1]$). $DPSolver$ handles such problems by saving more data for each subproblem. If the computation of $o_i(\cdot)$ depends on last T elements in b, i.e., $b[i-T : i-1]$, then $DPSolver$ increases the table by T dimensions to save the involved T elements.

Problems without optimal substructures. Some problems (e.g., blackhole) have overlapping subproblems but no optimal substructures. Even though they are not DP problems, $DPSolver$ still optimizes them by memoizing solutions to subproblems for data reuse. Instead of

saving data as "$M[i, c_i] = extreme\ o_i(\cdot)$", $DPSolver$ saves "$M[i, c_i, j] = true/false$" to indicate "*given i elements and constraint value c_i, whether or not o_i is equal to j.*". All saved data will be enumerated when $DPSolver$ searches for the best solution.

Problems with two or more candidate variables. Some problems (see Figure 3) have multiple candidate variables for DP optimization. $DPSolver$ enumerates and assesses different optimization strategies by concatenating arrays. With more detail, given multiple array variables B=$\{b_1, b_2, \ldots, b_k\}$, $DPSolver$ enumerates subsets in B. For instance, if $k = 3$, $DPSolver$ first analyzes each array for any optimization opportunity. Next, $DPSolver$ enumerates all array pairs for concatenation (e.g., (b_1, b_2), (b_2, b_3), and (b_1, b_3)), and analyzes whether any combined array can be used for optimization. Finally, $DPSolver$ concatenates all arrays to obtain a larger one (e.g., (b_1, b_2, b_3)), and decides whether any optimization is applicable. Such analysis can be time-consuming when k is large. In such scenarios, $DPSolver$ ranks arrays based on their lengths and focuses the analysis on longer arrays.

5 Evaluation

This section first introduces our evaluation data set (Section 5.1). It then describes the experiment settings (Section 5.2), and then finally explains our results (Section 5.3).

5.1 Data Set

To evaluate the effectiveness of $DPSolver$, we created a data set of 9 representative COP tasks:

- 0-1 knapsack.
- Complete knapsack: Similar to 0-1 knapsack but each item can be selected multiple times.
- Nested 0-1 knapsack.
- Shortest path: Given a graph of N nodes and the weighted edges between them, find a path between S and T to minimize the weight sum.
- Longest increasing subsequence: Find the longest ascending subsequence in a given sequence of length N.
- Longest common subsequence: Find the longest common subsequence between two given sequences of length N.
- Radiation therapy [Baatar *et al.*, 2007]: Given an $M \times N$ integer matrix describing the radiation dose to deliver to each area, decompose the matrix into a set of patterns for a radiation source to deliver. Minimize the working time of the radiation source and the number of used patterns.
- Modulus 0-1 knapsack: Similar to 0-1 knapsack but the objective is to maximize the last digit of the value sum.
- Blackhole: A player starts this solitaire card game by (1) moving Ace of Spades to the blackhole, and (2) arranging the other 51 cards as 17 piles of 3 cards each. In each turn, a card at the top of any pile can be moved to the blackhole if it is +1/-1 from the previous one. Find a way to move all cards.

The first seven problems are DP problems, and the last two problems are non-DP ones because they do not have the P1 property. We chose these problems in our evaluation for three reasons. First, the DP structures of the problems are distinct. Second, they are representative and cover most of the common DP structures from DP exercises. Third, the variants of knapsack are easy to explain and understand.

5.2 Experiment Settings

We applied $DPSolver$, Gecode [Schulte *et al.*, 2006], and CHUFFED [Chu *et al.*, 2012] to the data set. Specifically, CHUFFED can be executed as a naïve solver (denoted as CHUFFED), a solver with automatic caching (CHUFFEDC) [Smith, 2005], or a solver with Lazy Clause Generation [Ohrimenko *et al.*, 2009]. Although both CHUFFEDC and CHUFFEDL reduce redundant search, CHUFFEDC reuses active and non-satisfied constraints while CHUFFEDL uses constraints involved in conflicts to generate new constraints (nogoods).

We conducted the experiment on a personal computer (with an Intel Core i5-7300HQ 2.5GHz CPU, 8G of RAM). Table 1 shows the measured runtime overhead of different solvers. For generality, given the size of input for each problem (Column 4), we randomly generated 10 sets of input parameters, executed each tool with these inputs, and reported the average runtime overhead among the 10 runs. We set 10000 seconds for execution timeout. If a solver does not return any result within 10000 seconds, we terminated the execution.

5.3 Results

As shown in Table 1, Gecode and CHUFFED ran overtime for four problems (No. 4 and 6-8), mainly because these solvers applied almost no optimization or only built-in basic optimizations. In contrast, $DPSolver$ solved each of these problems within 100 seconds. For the remaining problems, $DPSolver$ achieved on average 698x speedup over Gecode and 790x speedup over CHUFFED.

> **Finding 1:** *$DPSolver$ sped up general constraint solvers by several orders of magnitude.*

We compared $DPSolver$ with two optimized solvers: CHUFFEDC and CHUFFEDL. Unexpectedly, CHUFFEDL ran overtime for four problems (No. 2, 4, 6, and 8) while $DPSolver$ solved all of them. For four out of the other problems, CHUFFEDL executed more slowly than both unoptimized solvers. This may be because a lot of nogoods are generated and propagated based on conflicts, but these nogoods could not help reduce the search space. For the five problems solvable by CHUFFEDL, $DPSolver$ obtained 345x speedup over CHUFFEDL on average. When solving all 9 problems, $DPSolver$ achieved 23x speedup over CHUFFEDC.

> **Finding 2:** *$DPSolver$ outperformed caching and LCG when solving DP problems and some non-DP problems with the P2 property.*

No.	Problem	Extra Handling	Size of Input	Gecode	CHUFFED	CHUFFEDC	CHUFFEDL	$DPSolver$	
								Total	Analysis
1	0-1 knapsack	-	$N = 10^3, C = 10^3$	975	1143	47	1598	2	0.05
2	Complete knapsack	-	$N = 10^3, C = 10^3$	3985	4262	125	>10000	2	0.05
3	Nested 0-1 knapsack	With multiple candidates	$N = 100, C1 = 100, Sum = 20$	357	496	78	672	1	0.03
4	Shortest path	With aftereffects (T=1)	$N = 10^3$	>10000	>10000	784	>10000	97	0.15
5	Longest inc. subseq.	With aftereffects (T=1)	$N = 10^3$	1274	1653	34	1751	5	0.03
6	Longest common subseq.	With multiple candidates, with aftereffects (T=1)	$N = 10^3$	>10000	>10000	55	>10000	10	0.08
7	Radiation therapy	With multiple candidates, with aftereffects (T=1)	$M = 8, N = 10$	>10000	>10000	40	4	4	0.18
8	Modulus 0-1 knapsack	Without P1	$N = 10^3, C = 10^3$	>10000	>10000	58	>10000	53	0.05
9	Blackhole	Without P1, with multiple candidates	$N = 52$	168	74	46	267	43	0.16

Table 1: Constraint solving time (in second) of different tools

Actually, the execution time of $DPSolver$ is spent for two tasks: (1) model analysis (Section 3), and (2) constraint solving. To compare the benefit and cost of model analysis, we measured the analysis time cost. As shown by the last column in Table 1, $DPSolver$ spent 0.03-0.18 second in analyzing each model and generating new models as needed. The analysis time cost is negligible compared with the constraint solving time saved by $DPSolver$ over current approaches.

Finding 3: *$DPSolver$ significantly accelerated constraint solving without introducing substantial analysis overhead.*

6 Related Work

This section describes related work on automatic optimization of constraint solving (Section 6.1), constraint solving with DP (Section 6.2), and tabling (Section 6.3).

6.1 Optimizers of Constraint Solving

When constraint solvers are used to search solutions for COP tasks, a number of methods were proposed to avoid or reduce redundant search in the process [Smith, 2005; Gent *et al.*, 2006; Kitching and Bacchus, 2007; Ohrimenko *et al.*, 2009; Chu *et al.*, 2012]. Specifically, caching memoizes search states to prevent the same state from being recomputed [Smith, 2005]. Symmetry breaking and subproblem dominance identify the equivalence or dominance relationship between states to avoid useless state exploration [Gent *et al.*, 2006; Chu *et al.*, 2012]. Symmetric component caching decomposes a COP into disjoint subproblems via variable assignments, such that each subproblem is solved independently to compose the optimal solution [Kitching and Bacchus, 2007]. Lazy Clause Generation (LCG) identifies constraints related to conflicts, and generates new constraints (nogoods) to reduce search [Ohrimenko *et al.*, 2009].

Compared with prior work, $DPSolver$ reduces unnecessary search by (1) identifying optimal substructures and overlapping subproblems in given problem descriptions, and (2) formatting the descriptions as DP-oriented models when possible. Our evaluation shows that $DPSolver$ worked much better than the state-of-the-art optimizers.

6.2 Constraint Solving with DP

Researchers created various methods to solve constraints via DP [Moor, 1994; Sauthoff *et al.*, 2011; Morihata *et al.*, 2014; Prestwich *et al.*, 2018]. For instance, DPE is a method to model DP in Constraint Programming (CP) [Prestwich *et al.*, 2018]. With this method, a modeler can specify a DP problem by defining a CP model via KOLMOGOROV—a constraint specification language. GAP is another domain-specific language for DP problem specification [Sauthoff *et al.*, 2011]. However, none of these approaches can automatically detect any DP problem structure in general constraint models.

Moor proved the conditions in which we can check monotonicity to verify the optimal substructures of given problems [Moor, 1994]. The research inspires our approach of DP problem recognition. Morihata et al. built a Haskell library for users to describe a problem in a naïve enumerate-and-choose style, and provided an approach that automatically derives efficient algorithms to solve the problem [Morihata *et al.*, 2014]. However, the library does not support users to specify any problem with multiple constraints, neither can users specify the three classes of problems listed in Section 4.

6.3 Tabling

Approaches were built to save intermediate computation results in certain data structures, so as to reduce or eliminate repetitive computation [Zhou *et al.*, 2015; Dekker *et al.*, 2017; de Uña *et al.*, 2019]. For example, auto-tabling provides MiniZinc annotations for users to define and insert table constraints and save calculation results to tables [Dekker *et al.*, 2017]. Similarly, the Picat tabling requires users to specify the variables for tabling [Zhou *et al.*, 2015]. Different from these tools, $DPSolver$ automatically detects memoization opportunities and creates tables without any user input.

de Uña *et al.* detected subproblem equivalence by hashing results [Chu *et al.*, 2012], and then used MDDs and formulas in d-DNNFs instead of tables to compute and store solutions [de Uña *et al.*, 2019]. To build an MDD or d-DNNF, a modeler has to specify the constraints, and the domain as well as ordering of related variables. DPSolver is different in two ways. First, it unfolds and analyzes predicates to detect overlapping subproblems and/or optimal substructures. Second, it does not require extra user input.

7 Discussion

Theoretically, $DPSolver$ is applicable when a model $\mathcal{M}$ meets two criteria:

(i) $\mathcal{M}$ describes a discrete optimization problem that has overlapping subproblems; and

(ii) $\mathcal{M}$ has at least an array or set of same-typed variables.

Essentially, (i) ensures the opportunity to trade space for time. Namely, if there is redundant computation between subproblems, we can save computed results to remove duplicated calculation. Additionally, (ii) ensures the applicability of the table-based search. With a fixed linear ordering between variables, we can index and retrieve solutions to subproblems.

In practice, $DPSolver$'s capability is also limited by another two conditions:

(iii) All constraints in $\mathcal{M}$ are defined with the built-in operations, global constraints, and non-recursive functions provided by MiniZinc. This is imposed by our current tool implementation.

(iv) The available memory space allocated by users should be sufficient to hold a table. Suppose that

a) there are N elements in a candidate array;

b) the size of each element's value range is M;

c) there are L ($L \geq 1$) accumulative functions related to the array;

d) for the i^{th} accumulative function, the size of the constraint value's range is R_i; and

e) the aftereffect value is T.

Then the space complexity is $O(NM^T \prod_{i=1}^{L} R_i)$. With this formula, $DPSolver$ decides whether the complexity value is larger than the allocated space; if not, $DPSolver$ optimizes the model.

$DPSolver$ is not limited to solving "pure" DP problems; it can handle some non-DP problems as long as the memoization of intermediate results can reduce duplicated calculation. For instance, the graph coloring problem is about coloring the N nodes in a graph with C different colors such that no two adjacent nodes have the same color. This non-DP problem can be described as a DP-oriented model with $T = N$. $DPSolver$ optimizes the solving process when N is small (e.g., $N = 10$), but does not do so when N is large (e.g., 50) because the space cost is too high.

Furthermore, we manually checked the 130 models in the MiniZinc Benchmark Suite[1], to estimate the applicability of our approach. Based on our observation, $DPSolver$ can fully optimize 9 models, and partially optimize 44 models by working for smaller sizes of inputs. The other problems have complicated constraints, and can be potentially handled by $DPSolver$ if we simplify the constraints.

[1]https://github.com/MiniZinc/minizinc-benchmarks

8 Conclusion

$DPSolver$ opportunistically optimizes constraint solving if a problem (1) uses any array variable and (2) can be reformulated as a DP-oriented problem. We focused on dynamic programming because although it is a popularly used problem-solving paradigm for large problems, it has not been fully exploited to optimize constraint solving.

Our research has made three contributions. First, $DPSolver$ analyzes a given problem modeled with MiniZinc to automatically check for two DP-related properties. Second, $DPSolver$ converts problem descriptions to DP-oriented models in novel ways such that DP problems and some non-DP problems can be efficiently resolved by generic constraint solvers. Third, we applied $DPSolver$ and related techniques to nine representative COP problems (including DP and non-DP problems). Impressively, our evaluation demonstrates that $DPSolver$ outperforms current tools with a speedup of several dozens or even hundreds.

Our investigation demonstrates the effectiveness of accelerating constraint solving via dynamic programming. $DPSolver$ currently handles problems containing array variables. In the future, we will improve $DPSolver$ to also handle problems having variables of other data structures, such as trees. Given a problem description written in natural languages, we also plan to automatically create a DP-oriented model by extracting arguments, variables, constraints, and objective functions directly from the description. In this way, users will learn about which problem is efficiently solvable and what is the corresponding MiniZinc model.

Acknowledgments

We thank anonymous reviewers for their valuable feedback. This work is partially supported by the National Natural Science Foundation of China NSFC under Grant No. 91646202 as well as Beijing Municipal Commission of Science and Technology under Grant No. Z181100008918005.

References

[Baatar *et al.*, 2007] Davaatseren Baatar, Natashia Boland, Sebastian Brand, and Peter J. Stuckey. Minimum cardinality matrix decomposition into consecutive-ones matrices: CP and IP approaches. In *International Conference on Integration of Artificial Intelligence (AI) and Operations Research (OR) Techniques in Constraint Programming*, pages 1–15. Springer, 2007.

[Bertsekas, 2000] Dimitri P. Bertsekas. *Dynamic Programming and Optimal Control*. Athena Scientific, 2nd edition, 2000.

[Chu *et al.*, 2012] Geoffrey Chu, Maria Garcia de la Banda, and Peter J. Stuckey. Exploiting subproblem dominance in constraint programming. *Constraints*, 17(1):1–38, 2012.

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

[de Uña *et al.*, 2019] Diego de Uña, Graeme Gange, Peter Schachte, and Peter J. Stuckey. Compiling CP subprob-

lems to MDDs and d-DNNFs. *Constraints*, 24(1):56–93, 2019.

[Dekker *et al.*, 2017] Jip J. Dekker, Gustav Björdal, Mats Carlsson, Pierre Flener, and Jean-Noël Monette. Auto-tabling for subproblem presolving in MiniZinc. *Constraints*, 22(4):512–529, 2017.

[Gent *et al.*, 2006] Ian P. Gent, Karen E. Petrie, and Jean-François Puget. Chapter 10 - Symmetry in Constraint Programming. In Francesca Rossi, Peter van Beek, and Toby Walsh, editors, *Handbook of Constraint Programming*, volume 2 of *Foundations of Artificial Intelligence*, pages 329 – 376. Elsevier, 2006.

[Kitching and Bacchus, 2007] Matthew Kitching and Fahiem Bacchus. Symmetric component caching. In *IJCAI*, pages 118–124, 2007.

[Marinescu and Dechter, 2005] Radu Marinescu and Rina Dechter. AND/OR branch-and-bound for graphical models. In *IJCAI*, pages 224–229, 2005.

[Moor, 1994] Oege De Moor. Categories, relations and dynamic programming. *Mathematical Structures in Computer Science*, 4(1):33–69, 1994.

[Morihata *et al.*, 2014] Akimasa Morihata, Masato Koishi, and Atsushi Ohori. *Dynamic Programming via Thinning and Incrementalization*. Springer International Publishing, 2014.

[Nethercote *et al.*, 2007] Nicholas Nethercote, Peter J. Stuckey, Ralph Becket, Sebastian Brand, Gregory J. Duck, and Guido Tack. MiniZinc: Towards a standard CP modelling language. *Principles and Practice of Constraint Programming–CP 2007*, pages 529–543, 2007.

[Ohrimenko *et al.*, 2009] Olga Ohrimenko, Peter J. Stuckey, and Michael Codish. Propagation via lazy clause generation. *Constraints*, 14(3):357–391, 2009.

[Prestwich *et al.*, 2018] Steven Prestwich, Roberto Rossi, S. Armagan Tarim, and Andrea Visentin. Towards a closer integration of dynamic programming and constraint programming. *EPiC Series in Computing*, 55:202–214, 2018.

[Sauthoff *et al.*, 2011] Georg Sauthoff, Stefan Janssen, and Robert Giegerich. Bellman's GAP: a declarative language for dynamic programming. In *International ACM Sigplan Symposium on Principles and Practices of Declarative Programming*, pages 29–40, 2011.

[Schulte *et al.*, 2006] Christian Schulte, Mikael Lagerkvist, and Guido Tack. Gecode. *Software download and online material at the website: http://www.gecode.org*, pages 11–13, 2006.

[Smith, 2005] Barbara M. Smith. Caching search states in permutation problems. In *International Conference on Principles and Practice of Constraint Programming*, pages 637–651. Springer, 2005.

[Zhou *et al.*, 2015] Neng-Fa Zhou, Håkan Kjellerstrand, and Jonathan Fruhman. *Constraint solving and planning with Picat*. Springer, 2015.