Synthesis of Fast Programs for Maximum Segment Sum Problems

Srinivas Nedunuri & William R. Cook
University of Texas at Austin

October 5, 2009
Motivation

- **Given**
  - Behavioral specifications
  - Pre/Post condition
- **Synthesize**
  - Efficient algorithms
- **Primary Tools**
  - Algorithm Theories
    - Global Search
    - Local Search
    - Divide and Conquer
  - “Calculation” (derivation) of program components
- **Global Search → Constraint Satisfaction**
What is Constraint Satisfaction?

Constraint Satisfaction

Given a set of variables, \( \{v\} \), assign a value, drawn from some domain \( D_v \), to each variable, in a manner that satisfies a given set of constraints.

- Many problems can be expressed as constraint satisfaction problems
  - Knapsack problems
  - Graph problems
  - Integer Programming

- We want to show that doing so leads to efficient algorithms
General versus Specific Constraint Solvers

- *Not* a generic constraint solver
- *Instead*...
- Synthesize algorithm for specific constraint-based problem
Example problem

Maximum Independent Segment Sum (MISS)

Maximize the sum of a selection of elements from a given array, with the restriction that no two adjacent elements can be selected.

The synthesis approach we follow starts with a formal specification of the problem.
Format of Specifications

**Structure of Specification**

- An input type, $D$
- A result type, $R$
- A cost type, $C$
- An output condition (postcondition), $o : D \times R \rightarrow \text{Boolean}$
- A benefit criterion, $\text{profit} : D \times R \rightarrow C$
Maximum Independent Segment Sum (MISS)

**Instantiation for MISS**

\[
D \mapsto \text{maxVar} : \text{Nat} \times \text{vals} : \{D_v\} \times \text{data} : [\text{Int}]
\]

\[
D_v = \{\text{False, True}\}
\]

\[
R \mapsto m : \text{Map(}\text{Nat} \rightarrow D_v\text{)} \times \text{cs} : \{D_v\}
\]

\[
C \mapsto \text{Int}
\]

\[
o \mapsto \lambda(x, z). \text{dom}(z.m) = \{1..(x.\text{maxVar})\} \land \text{nonAdj}(z)
\]

\[
\text{nonAdj} = \lambda z. \forall i. 1 \leq i < \#z.m. z_i \Rightarrow \neg z_{i+1}
\]

\[
\text{profit} \mapsto \lambda(x, z). \sum_{i=1}^{\#z}(z_i \rightarrow x_i | 0)
\]

**Example**

<table>
<thead>
<tr>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
</tr>
</thead>
<tbody>
<tr>
<td>T</td>
<td>F</td>
<td>F</td>
<td>T</td>
<td>F</td>
<td>T</td>
</tr>
</tbody>
</table>

1.. x.maxVars

<table>
<thead>
<tr>
<th>3</th>
<th>9</th>
<th>-2</th>
<th>-10</th>
<th>0</th>
<th>1</th>
</tr>
</thead>
</table>

x.data
Take the solution space (potentially infinite) and partition it. Each element of the partition is called a subspace, and is recursively partitioned until a singleton space is encountered, called a solution$^1$

**Partial Solution or Space ($\hat{z}$)**

An assignment to some of the variables. Can be extended into a (complete) solution by assigning to all the variables.

**Feasible Solution ($z$)**

A solution which satisfies the output condition

---

$^1$based on N. Agin, “Optimum Seeking with Branch and Bound”, Mgmt. Sci. 1966
Search Tree

zhat_0

subspaces

Solution space

Extract /= Nothing
Global Search with Optimization (GSO)

- An algorithm class that consists of a *program schema* (template) containing *operators* whose semantics is axiomatically defined.
- Operators must be instantiated by the user (developer). They are typically *calculated* (Dijkstra style).
- Two groups of operators: the basic space forming ones and more advanced ones which control the search.
### GSO Extension

<table>
<thead>
<tr>
<th>Operator</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>extract</td>
<td>$D \times R \rightarrow R$</td>
<td>determines whether the given space corresponds to a leaf node, returns it if so, otherwise Nothing</td>
</tr>
<tr>
<td>subspaces</td>
<td>$D \times R \rightarrow {R}$</td>
<td>partitions the given space into subspaces</td>
</tr>
<tr>
<td>$\sqsubseteq$</td>
<td>${R \times R}$</td>
<td>if $r \sqsubseteq s$ then $s$ is a subspace of $r$ (any solution contained in $s$ is contained in $r$)</td>
</tr>
<tr>
<td>$\widehat{z}_0$</td>
<td>$D \rightarrow R$</td>
<td>forms the initial space (root node)</td>
</tr>
</tbody>
</table>

These can usually be written down by inspection of the problem.
The Search Control Operators

GSO Extension

<table>
<thead>
<tr>
<th>Operator</th>
<th>Called</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\Phi$</td>
<td>Necessary Filter</td>
<td>$D \times R \rightarrow \text{Boolean}$</td>
<td>Necessary condition for a space to contain feasible solutions</td>
</tr>
<tr>
<td>$\psi(\xi)$</td>
<td>Necessary (Consistent) Tightener</td>
<td>$D \times R \rightarrow R$</td>
<td>Tightens a given space to eliminate infeasible solutions. Preserves all (at least one) feasible solutions</td>
</tr>
<tr>
<td>$\text{ub}(ib)$</td>
<td>Upper (Initial) Bound</td>
<td>$D \times R \rightarrow C$</td>
<td>returns a upper(initial) bound on the profit of the best solution in the given space</td>
</tr>
</tbody>
</table>

These are usually derived from their specification by the application of domain knowledge.
Global Search Optimization: generic algorithm in Haskell

\[ f_{\text{gso}} :: D \times \{R\} \times \{R\} \rightarrow \{R\} \]
\[ f_{\text{gso}}(x, \text{active}, \text{soln}) = \]
\[ \text{if empty(active)} \]
\[ \text{then soln} \]
\[ \text{else let} \]
\[ (r, \text{rest}) = \text{arbsplit(active)} \]
\[ \text{soln'} = \text{opt(profit, soln} \cup \{z \mid \text{extract}(z, r) \land o(x,z)\}) \]
\[ \text{ok_subs} = \{\text{propagate}(x, s) : \]
\[ s \in \text{subspaces}(r) \]
\[ \land \text{propagate}(x, s) \neq \text{Nothing}\} \]
\[ \text{subs'} = \{s : s \in \text{ok_subs} \]
\[ \land \text{ub}(x, s) \geq \text{lb}(x, \text{soln'})\} \]
\[ \text{in f}_{\text{gso}}(x, \text{rest} \cup \text{subs}, \text{soln'}) \]
Global Search Optimization (cont.)

\[ \text{ub} :: D \times \{R\} \rightarrow C \]
\[ \text{ub}(x, \text{solns}) = \]
\[ \quad \text{if empty(solns) then \text{ib}(x) else profit}(x, \text{arb(solns)}) \]

\[ \text{propagate} \ x \ r = \]
\[ \quad \text{if \phi}(x, r) \text{ then (iterateToFixedPoint \ psi} \ x \ r) \text{ else Nothing} \]

\[ \text{iterateToFixedPoint} \ f \ x \ z = \]
\[ \quad \text{let \ fx} = f(x, z) \text{ in} \]
\[ \quad \text{if \ fx} = z \text{ then \ fx else iterateToFixedPoint} \ f \ x \ fx \]
Operator Instantiations for MISS

We already have $D, R, C, o,$ and $cost$ (from the specification). The space forming operators can be instantiated by inspection:

<table>
<thead>
<tr>
<th>Generic Instantiation (CSOT)</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\hat{z}_0$           $\mapsto$ $\lambda x. { m = \emptyset, cs = x.\text{vals}}$</td>
</tr>
<tr>
<td>$\text{subspaces}$     $\mapsto$ $\lambda (x, \hat{z}). { \hat{z}' : v = \text{chooseVar}({1..x.\text{maxVar}} - \text{dom}(\hat{z}.m)), \exists a \in \hat{z}.cs. \hat{z}'m = \hat{z}.m \oplus (v \mapsto a)}$</td>
</tr>
<tr>
<td>$\text{extract}$       $\mapsto$ $\lambda (x, \hat{z}). \text{dom}(\hat{z}.m) = {1..x.\text{maxVar}} \rightarrow \hat{z}</td>
</tr>
<tr>
<td>$\sqsubseteq$          $\mapsto$ ${(\hat{z}, \hat{z}')</td>
</tr>
<tr>
<td>$ib$                  $\mapsto$ $\text{maxBound}$</td>
</tr>
</tbody>
</table>

$\oplus$ denotes adding a pair to a map and is defined as

$$m \oplus (x \mapsto a) \triangleq m - \{(x, a')\} \cup \{(x, a)\}$$

The search control operators $\Phi, \psi, ub$ are given default definitions (not shown). We now have a working implementation of an algorithm for MISS.
With this instantiation, the abstract program is correctly instantiated into a working solver. But it has exponential complexity! (The search space grows exponentially). Even with good definitions for the search control operator it still grows exponentially.

So we incorporate a concept that has been used in operations research for several decades: dominance relations.
With this instantiation, the abstract program is correctly instantiated into a working solver. But it has exponential complexity! (The search space grows exponentially). Even with good definitions for the search control operator it still grows exponentially.

So we incorporate a concept that has been used in operations research for several decades: **dominance relations**.
What are dominance relations?

- Enables the comparison of one partial solution with another to determine if one of them can be discarded.
- Given \( \hat{z} \) and \( \hat{z}' \) if the best possible solution in \( \hat{z} \) is better than the best possible solution in \( \hat{z}' \) then \( \hat{z}' \) can be discarded.
What are dominance relations?

- Enables the comparison of one partial solution with another to determine if one of them can be discarded.
- Given \( \hat{Z} \) and \( \hat{Z}' \) if the best possible solution in \( \hat{Z} \) is better than the best possible solution in \( \hat{Z}' \) then \( \hat{Z}' \) can be discarded.
One way to derive dominance is to focus on a restricted case: dominance relative to equivalent extensions.

- Let $\hat{Z} \oplus e$ denote combining a partial solution $\hat{Z}$ with an extension $e$.
- When $\hat{Z} \oplus e$ is a (feasible) complete solution, $e$ is called the (feasible) completion of $\hat{Z}$.

A special case of dominance arises when all feasible completions of a space are also feasible completions for another space, and the first solution is always better than the second solution.
Definitions

**Definition: Semi-Congruence**

is a relation $\sim \subseteq R^2$ such that

$$\forall e, \hat{z}, \hat{z}' \in R : \hat{z} \sim \hat{z}' \Rightarrow o(\hat{z}' \oplus e) \Rightarrow o(\hat{z} \oplus e)$$

Then we need to say something about when one space is “better” than another. We call this weak dominance. If $\hat{z}$ weakly dominates $\hat{z}'$, then any feasible completion of $\hat{z}$ is at least as beneficial as the same feasible completion of $\hat{z}'$

**Definition: Weak Dominance**

is a relation $\delta \subseteq R^2$ such that

$$\forall e, \hat{z}, \hat{z}' \in R : \hat{z} \delta \hat{z}' \Rightarrow o(\hat{z} \oplus e) \land o(\hat{z}' \oplus e) \Rightarrow p(\hat{z} \oplus e) \geq p(\hat{z}' \oplus e)$$
Definitions

Definition: Semi-Congruence

is a relation $\sim \subseteq R^2$ such that

$$\forall e, \hat{z}, \hat{z}' \in R : \hat{z} \sim \hat{z}' \Rightarrow o(\hat{z}' \oplus e) \Rightarrow o(\hat{z} \oplus e)$$

Then we need to say something about when one space is “better” than another. We call this weak dominance. If $\hat{z}$ weakly dominates $\hat{z}'$, then any feasible completion of $\hat{z}$ is at least as beneficial as the same feasible completion of $\hat{z}'$.

Definition: Weak Dominance

is a relation $\hat{\delta} \subseteq R^2$ such that

$$\forall e, \hat{z}, \hat{z}' \in R : \hat{z} \hat{\delta}\hat{z}' \Rightarrow o(\hat{z} \oplus e) \land o(\hat{z}' \oplus e) \Rightarrow p(\hat{z} \oplus e) \geq p(\hat{z}' \oplus e)$$
Definitions

**Definition: Semi-Congruence**

is a relation $\sim \subseteq R^2$ such that

$$\forall e, \hat{z}, \hat{z}' \in R : \hat{z} \sim \hat{z}' \Rightarrow o(\hat{z}' \oplus e) \Rightarrow o(\hat{z} \oplus e)$$

Then we need to say something about when one space is “better” than another. We call this weak dominance. If $\hat{z}$ weakly dominates $\hat{z}'$, then any feasible completion of $\hat{z}$ is at least as beneficial as the same feasible completion of $\hat{z}'$

**Definition: Weak Dominance**

is a relation $\delta \subseteq R^2$ such that

$$\forall e, \hat{z}, \hat{z}' \in R : \hat{z} \delta \hat{z}' \Rightarrow o(\hat{z} \oplus e) \land o(\hat{z}' \oplus e) \Rightarrow p(\hat{z} \oplus e) \geq p(\hat{z}' \oplus e)$$
Dominance Relations (contd.)

To get a dominance test, combine the two

**Theorem (Dominance)**

\[ \forall \hat{z}, \hat{z}' \in R : \hat{z} \delta \hat{z}' \land \hat{z} \leadsto \hat{z}' \Rightarrow \text{profit}^*(\hat{z}) \geq \text{profit}^*(\hat{z}') \]

ie., if \( \hat{z} \) is semi-congruent with \( \hat{z}' \) and \( \hat{z} \) weakly dominates \( \hat{z}' \) then the cost of the best solution in \( \hat{z} \) at least as beneficial as the best solution in \( \hat{z}' \)

When \( \text{profit}^*(\hat{z}) \geq \text{profit}^*(\hat{z}') \) we say \( \hat{z} \) dominates \( \hat{z}' \), written \( \hat{z} \delta \hat{z}' \)

How does this fit into CSOT? Following is a cheap way to get a weak-dominance condition:

**Theorem (Profit Distribution)**

If \( \text{profit} \) distributes over \( \oplus \) and \( \text{profit}(\hat{z}) \geq \text{profit}(\hat{z}') \) then \( \hat{z} \delta \hat{z}' \)
To get a dominance test, combine the two

**Theorem (Dominance)**

\[ \forall \hat{z}, \hat{z}' \in R : \hat{z} \hat{\delta} \hat{z}' \land \hat{z} \rightsquigarrow \hat{z}' \Rightarrow \text{profit}^*(\hat{z}) \geq \text{profit}^*(\hat{z}') \]

i.e., if \( \hat{z} \) is semi-congruent with \( \hat{z}' \) and \( \hat{z} \) weakly dominates \( \hat{z}' \) then the cost of the best solution in \( \hat{z} \) at least as beneficial as the best solution in \( \hat{z}' \)

When \( \text{profit}^*(\hat{z}) \geq \text{profit}^*(\hat{z}') \) we say \( \hat{z} \) *dominates* \( \hat{z}' \), written \( \hat{z} \hat{\delta} \hat{z}' \)

How does this fit into CSOT? Following is a cheap way to get a weak-dominance condition:

**Theorem (Profit Distribution)**

If \( \text{profit} \) distributes over \( \oplus \) and \( \text{profit}(\hat{z}) \geq \text{profit}(\hat{z}') \) then \( \hat{z} \hat{\delta} \hat{z}' \)
First calculate the semi-congruence condition \( \sim \) between \( \hat{z} \) and \( \hat{z}' \). Working backwards from the conclusion of the definition of semi-congruence:

\[
\begin{align*}
o(\hat{z} \oplus e) &= \{ \text{unfold defn, let } L = \#\hat{z}, L' = \#\hat{z}' \}
dom(\hat{z}.m) + \dom(e.m) &= [1..(x\.maxVar)] \\
&\land \text{nonAdj}(\hat{z}) \land \text{nonAdj}(e) \land (\hat{z}_L \Rightarrow \neg e_1) \\
&\Leftarrow \{ \text{nonAdj}(\hat{z}') \land \text{nonAdj}(e) \land (\hat{z}'_L \Rightarrow \neg e_1) \text{, from } o(\hat{z}' \oplus e) \}
dom(\hat{z}.m) + \dom(e.m) &= [1..(x\.maxVar)] \\
&\land \text{nonAdj}(\hat{z}) \land ((\hat{z}'_L \Rightarrow \neg e_1) \Rightarrow (\hat{z}_L \Rightarrow \neg e_1)) \\
&= \{ \text{anti-monotonicity of } (k \Leftarrow) \}
dom(\hat{z}.m) + \dom(e.m) &= [1..(x\.maxVar)] \\
&\land \text{nonAdj}(\hat{z}) \land (\neg \hat{z}'_L \Rightarrow \neg \hat{z}_L) \\
&= \{ \text{vars assigned consecutively } \land \dom(\hat{z}' .m) + \dom(e.m) = [1..(x\.maxVar)] \}
L = L' \land \text{nonAdj}(\hat{z}) \land (\neg \hat{z}'_L \Rightarrow \neg \hat{z}_L) \\
&= \{ \text{simplification} \}
L = L' \land \text{nonAdj}(\hat{z}) \land (\hat{z}_L \Rightarrow \hat{z}'_L)
\end{align*}
\]
Since *profit* is a distributive profit function, the definition for $\delta$ follows immediately: $\hat{z} \sim \hat{z}' \land profit(\hat{z}) \geq profit(\hat{z}')$

This dominance test reduces the complexity of the MISS algorithm from exponential to polynomial. This is good but we can do better.
Apply a “Neighborhood” tactic to calculate a tightener for a space: If a segment is selected, then the next segment must not be selected.
An upper bound

- An upper bound on a partial solution is the value of the best possible solution obtainable from that partial solution.
- Combine the profit of the partial solution with the best possible profit obtainable from the remaining variables.

\[
upperBound(x, \hat{z}) = p(x, \hat{z}) + \sum_{i=\#\hat{z}}^{\#x.sqnce} \max(x.sqnce(i), 0)
\]
What is the cumulative effect of all the operators?

For input $x = [1 \ldots 10]$:

<table>
<thead>
<tr>
<th>Operator Added</th>
<th># of Nodes in Search Tree</th>
</tr>
</thead>
<tbody>
<tr>
<td>None</td>
<td>2047</td>
</tr>
<tr>
<td>+ dominates</td>
<td>486</td>
</tr>
<tr>
<td>+tighten</td>
<td>12</td>
</tr>
<tr>
<td>+upperBound</td>
<td>12</td>
</tr>
</tbody>
</table>

- Dominance and Tightening are very significant in eliminating large swathes of the search space.
- But the algorithm is still not linear time.
Finite Differencing (Page & Koenig, 1982)

Incrementally update an expensive computation rather than computing it each time in the loop. Requires introducing accumulating arguments into the main search loop. Tedious, but not difficult.
Final Algorithm

Theorem

Algorithm MISS runs in linear time

Following table shows the results of running on sequences of randomly generated numbers of varying length

<table>
<thead>
<tr>
<th>Input Length</th>
<th>NC (s)</th>
<th>Sasano (s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1000</td>
<td>0.00</td>
<td>0.00</td>
</tr>
<tr>
<td>10,000</td>
<td>0.12</td>
<td>0.14</td>
</tr>
<tr>
<td>20,000</td>
<td>0.22</td>
<td>0.28</td>
</tr>
<tr>
<td>40,000</td>
<td>0.43</td>
<td>0.72</td>
</tr>
<tr>
<td>80,000</td>
<td>0.75</td>
<td>1.8</td>
</tr>
<tr>
<td>100,000</td>
<td>1.1</td>
<td>2.8</td>
</tr>
<tr>
<td>200,000</td>
<td>2.2</td>
<td>8.9</td>
</tr>
<tr>
<td>400,000</td>
<td>4.6</td>
<td>stack overflow</td>
</tr>
</tbody>
</table>
Using the same approach, and with several small changes to the derivation, we have synthesized efficient linear-time algorithms for variations on the problem, specifically Maximum Multi-Marking and Maximum Alternating Segment Sum (see the paper for the details).

In all cases we outperform the code produced by Sasano et al. using program transformation.
Summary & Conclusions

- We have shown how the addition of dominance relations can significantly improve the complexity of an algorithm.
- We have applied the ideas of program synthesis to some useful and well-known problems.
- Program synthesis is an effective way of generating effective and efficient code.
- The methodology we have applied can be used to generate algorithms for a family of related programs, with sharing of derivations. In contrast, program transformation requires a completely new transformation for each variation.