Theory and Techniques for Synthesizing Efficient Breadth-First Search Algorithms

Srinivas Nedunuri[1], Douglas R. Smith[2], William R. Cook[1]

[1] University of Texas at Austin
[2] Kestrel Institute

Abstract. Although Breadth-First Search (BFS) has several advantages over Depth-First Search (DFS) its prohibitive space requirements have meant that algorithm designers often pass it over in favor of DFS. To address this shortcoming, we introduce a theory of efficient BFS (EBFS), along with a simple recursive program schema for carrying out the search. The theory is based on dominance relations, a long standing technique from the field of search algorithms. We also show that greedy and greedy-like algorithms form a very useful and important sub-category of EBFS. Finally, we show how the EBFS class can be used for semi-automated program synthesis by introducing some techniques for demonstrating that a given problem is solvable by EBFS. We illustrate our approach on several examples.

1 Introduction

Program synthesis is experiencing something of a resurgence [SGF10,SLTB$^+$06,GJTV11] [PBS11,VY08,VYY10] following negative perceptions of its scalability in the early 90s. Many of the current approaches aim for near-automated synthesis. In contrast, the approach we follow, we call *guided program synthesis*, also incorporates a high degree of automation but is more user-guided. The basic idea is to identify interesting classes of algorithms and capture as much *generic* algorithm design knowledge as possible in one place.The user instantiates that knowledge with problem-specific *domain* information. This step is often carried out with machine assistance. The approach has been applied to successfully derive scores of efficient algorithms for a wide range of practical problems including scheduling [SPW95], concurrent garbage collection [PPS10], and SAT solvers [SW08].

One significant class of algorithms that has been investigated is search algorithms. Many interesting problems can be solved by application of search. In such an approach, an initial search space is partitioned into subspaces, a process called *splitting,* which continues recursively until a *feasible* solution is found. A feasible solution is one that satisfies the given problem specification. Viewed as a search tree, spaces form nodes, and the subspaces after a split form the children of that node. The process has been formalized by Smith [Smi88,Smi10]. Problems which can be solved by global search are said to be in the Global Search (GS) class. The enhancements in GS over standard branch-and-bound include a number of techniques designed to improve the quality of the search by eliminating unpromising avenues. One such technique is referred to as *dominance relations.* Although they do not appear to have been widely used, the idea of dominance relations goes back to at least the 70s [Iba77]. Essentially, a dominance relation is a relation between two nodes in the search tree such that if one dominates the other, then the dominated node is guaranteed to lead to a worse solution than the dominating one, and can therefore be discarded. Establishing a dominance relation for a given problem is carried out by a user. However this process is not always obvious. There are also a variety of ways in which to carry out the search, for example Depth-First (DFS), Breadth-First (BFS), Best-First, etc. Although DFS is the most common, BFS actually has several advantages over DFS were it not for its exponential space requirement. The key to carrying out BFS space-efficiently is to limit the size of the frontier at any level. However, this has not been investigated in any systematic manner up to now.

This paper has two main contributions:

- We show how to limit the size of the frontier in search using dominance relations, thereby enabling space-efficient BFS. Additionally, we show that limiting the size of the undominated frontier to a constant results in a useful class of *greedy* algorithms.
- Even though our method is not automatic, we believe that the process should be straightforward to apply, without requiring Eureka steps. For this reason, we have devised techniques that address roadblocks in derivations, which are illustrated on some simple but illuminating examples. Further examples are in [NSC12]

2 Background To Guided Program Synthesis

2.1 Process

The basic steps in guided program synthesis are:

1. Start with a logical specification of the problem to be solved. A specification is a quadruple $\langle D, R, o, c\rangle$ where D is an input type, R an output or result type, $o : D \times R$ is a predicate relating correct or feasible outputs to inputs, and $c : D \times R \rightarrow Int$ is a cost function on solutions. An example specification is in Eg. 1 (This specification is explained in more detail below)
2. Pick an algorithm class from a library of algorithm classes (Global Search, Local Search, Divide and Conquer, Fixpoint Iteration, etc). An algorithm class comprises a *program schema* containing operators to be instantiated and an *axiomatic theory* of those operators (see [Ned12] for details). A schema is analogous to a template function in Java/C++ with the difference that both the template and template arguments are formally constrained.
3. Instantiate the operators of the program schema using information about the problem domain and in accordance with the axioms of the class theory. To ensure correctness, this step can be carried out with mechanical assistance. The result is an efficient algorithm for solving the given problem.
4. Apply low-level program transforms such as finite differencing, context-dependent simplification, and partial evaluation, followed by code generation. Many of these are automatically applied by Specware [S], a formal program development environment.

The result of Step 4 is an efficient program for solving the problem which is guaranteed correct by construction. The power of the approach stems from the fact that the common structure of many algorithms is contained in *one* reusable program schema and associated theory. Of course the program schema needs to be carefully designed, but that is done once by the library designer. The focus of this paper is the Global Search class, and specifically on how to methodically carry out Step 3 for a wide variety of problems. Details of the other algorithm classes and steps are available elsewhere [Kre98,Smi88,PPS10].

Example 1. Specification of the Shortest Path problem is shown in Fig. 2.1 (The $\mapsto$ reads as "instantiates to") The input D is a structure with 3 fields, namely a start node, end node and a set of edges. The result R is a sequence of edges ([] notation). A correct result is one that satisfies the predicate *path*? which checks that a path z must be a contiguous path from the start node to the end node (simple recursive definition not shown). Finally the cost of a solution is the sum of the costs of the edges in that solution. Note that fields of a structure are accessed using the '.' notation.

2.2 Global Search

Before delving into a program schema for Global Search, it helps to understand the structures over which the program schema operates. In [Smi88], a *search space* is represented by a descriptor of some type $\widehat{R}$, which is an abstraction of the result type R. The initial or starting space is denoted $\bot$. There are also two predicates *split*: $D \times \widehat{R} \times \widehat{R}$, written $\pitchfork$, and *extract*: $\widehat{R} \times R$, written χ. Split defines when a space is a subspace of another space, and extract captures when a solution is extractable from a space. We say a solution z is *contained* in a space y (written $z \in y$) if it can be extracted after a finite number of splits. A feasible space is one that contains feasible solutions. We often write $\pitchfork\ (x, y, y')$ as $y \pitchfork_x y'$ for readability, and even drop the subscript when there is no confusion. *Global Search theory (GS-theory)* [Smi88] axiomatically characterizes the relation between the predicates $\bot$, $\pitchfork$ and χ, as well as ensuring that the associated program schema computes a result that satisfies the specification. In the sequel, the symbols $\widehat{R}, \bot, \pitchfork, \chi, \oplus$ are all assumed to be drawn from GS-theory.

$$\begin{aligned}
D &\mapsto \langle start : Node, end : Node, edges : \{Edge\}\rangle \\
&\quad Edge = \langle f : Node, t : Node, cost : Nat\rangle \\
R &\mapsto [Edge] \\
o &\mapsto \lambda(x, z) \cdot path?(z, x.start, x.end) \\
&\quad path?(p, s, f) = ... \\
c &\mapsto \lambda(x, z)\ \cdot \textstyle\sum_{edge \in z} edge.cost
\end{aligned}$$

Fig. 2.1. Specification of Shortest Path problem

Example 2. Instantiating GS-theory for the Shortest Path problem requires instantiating the free terms in the theory. The type of solution spaces $\widehat{R}$ is the same as the result type R. However, there is a covariant relationship between an element of $\widehat{R}$ and of R. For example, the initial space, corresponding to all possible paths, is the empty list. A space is split by adding an edge to the current path - that is the subspaces are the different paths that result from adding an edge to the parent path. Finally a solution can be trivially extracted from any space by setting the result z to the space p. This is summarized in Fig. 2.2 ([] denotes the empty list, and ++ denotes list concatenation).

2.3 Dominance Relations

$$\begin{aligned}
\widehat{R} &\mapsto R \\
\bot &\mapsto \lambda x \cdot [] \\
\pitchfork &\mapsto \lambda(x, p, pe) \cdot \exists e \in x.edges \cdot pe = p{+\!+}[e] \\
\chi &\mapsto \lambda(z, p) \cdot p = z
\end{aligned}$$

Fig. 2.2. GS instantiation for Shortest Path

As mentioned in the introduction, a dominance relation provides a way of comparing two subspaces in order to show that one will always contain at least as good a solution as the other. (Goodness in this case is measured by some cost function on solutions). The first space is said to *dominate* ($\rhd$) the second, which can then be eliminated from the search. Letting c^* denote the cost of an optimal solution in a space, this can be formalized as (all free variables are assumed to be universally quantified):

$$y \rhd y' \Rightarrow c^*(x, y) \leq c^*(x, y') \tag{2.1}$$

Another way of expressing the consequent of (2.1) is

$$\forall z' \in y' \cdot o(x, z') \Rightarrow \exists z \in y \cdot o(x, z) \wedge c(x, z) \leq c(x, z') \tag{2.2}$$

To derive dominance relations, it is often useful to first derive a semi-congruence relation [Smi88]. A semi-congruence between two partial solutions y and y', written $y \rightsquigarrow y'$, ensures that any way of extending y' into a feasible solution can also be used to extend y into a feasible solution. Like $\pitchfork$, $\rightsquigarrow$ is a ternary relation over $D \times \widehat{R} \times \widehat{R}$ but as we have done with $\pitchfork$ and many other such relations in this work, we drop the input argument when there is no confusion and write it as a binary relation for readability. Before defining semi-congruence, we introduce two concepts. One is the idea of *useability* of a space. A space y is is useable, written $o^*(x, y)$, if $\exists z.\, \chi(y, z) \wedge o(x, z)$, meaning a feasible solution can be extracted from the space. The second is the notion of incorporating sufficient information into a space to make it useable. This is defined by an operator $\oplus : \widehat{R} \times t \rightarrow \widehat{R}$ that takes a space and some additional information of type t and returns a more defined space. The type t depends on $\widehat{R}$. For example if $\widehat{R}$ is the type of lists, then t might also be the same type. Now the formal definition of semi-congruence is:

$$y \rightsquigarrow y' \Rightarrow o^*(x, y' \oplus e) \Rightarrow o^*(x, y \oplus e)$$

That is, $y \rightsquigarrow y'$ is a sufficient condition for ensuring that if y' can be extended into a feasible solution than so can y *with the same extension*. Now if c is compositional (that is, $c(s \oplus t) = c(s) + c(t)$) then it can be shown [Ned12] that if $y \rightsquigarrow y'$ and y is cheaper than y', then y ***dominates*** y' (written $y \rhd y'$). Formally:

$$y \rightsquigarrow y' \wedge c(x, y) \leq c(x, y') \Rightarrow y \rhd y' \tag{2.3}$$

Dominance relations are a part of GS-theory [Smi88].

Example 3. Shortest Path between two given nodes in a graph. If there are two paths p and p' leading from the start node, if p and p' both terminate in the same node then $p \rightsquigarrow p'$. The reason is that any path extension e (of type $t = [Edge]$) of p' that leads to the target node is also a valid path extension for p. Additionally if p is shorter than p' then p dominates p', which can be discarded. Note that this does not imply that p leads to the target node, simply that no optimal solutions are lost in discarding p'. This dominance relation is formally derived in Eg. 5

Example 4. 0-1 Knapsack

The 0-1 Knapsack problem is, given a set of items each of which has a weight and utility and a knapsack that has some maximum weight capacity, to pack the knapsack with a subset of items that maximizes utility and does not exceed the knapsack capacity. Given combinations k, k', if k and k' have both examined the same set of items and k weighs less than k' then any additional items e that can be feasibly added to k' can also be added to k, and therefore $k \rightsquigarrow k'$. Additionally if k has at least as much utility as k' then $k \rhd k'$.

The remaining sections cover the original contributions of this paper .

3 A Theory Of Space-Efficient Breadth-First Search (EBFS)

While search can in principle solve any computable function, it still leaves open the question of how to carry it out effectively. Various search strategies have been investigated over the years; two of the most common being Breadth-First Search (BFS) and Depth-First Search (DFS). It is well known that BFS offers several advantages over DFS. Unlike DFS which can get trapped in infinite paths[3], BFS will always find a solution if one exists. Secondly, BFS does not require backtracking. Third, for deeper trees, BFS will generally find a solution at the earliest possible opportunity. However, the major drawback of BFS is its space requirement which grows exponentially. For this reason, DFS is usually preferred over BFS.

Our first contribution in this paper is to refine GS-theory to identify the conditions under which a BFS algorithm can operate space-efficiently. The key is to show how the size of the undominated frontier of the search tree can be polynomially bounded. Dominance relations are the basis for this.

In [Smi88], the relation $\pitchfork^l$ for $l \geq 0$ is recursively defined as follows:

$$\begin{aligned} y \pitchfork^0 y' &= (y = y') \\ y \pitchfork^{l+1} y' &= \exists y'' \cdot y \pitchfork y'' \wedge y'' \pitchfork^l y' \end{aligned}$$

From this the next step is to define those spaces at a given frontier level that are not dominated. However, this requires some care because dominance is a pre-order, that is it satisfies the reflexivity and transitivity axioms as a partial order does, but not the anti-symmetry axiom. That is, it is quite possible for y to dominate y' and y' to dominate y but y and y' need not be equal. An example in Shortest Path is two paths of the same length from the start node that end at the same node. Each path dominates the other. To eliminate such cyclic dominances, define the relation $y \approx y'$ as $y \rhd y' \wedge y' \rhd y$. It is not difficult to show that $\approx$ is an equivalence relation. Now let the *quotient frontier* at level l be the quotient set $\mathit{frontier}_l / \approx$. For type consistency, let the *representative* frontier $\mathit{rfrontier}_l$ be the quotient frontier in which each equivalence class is replaced by some arbitrary member of that class. The representative frontier is the frontier in which cyclic dominances have been removed. Finally then the *undominated* frontier undom_l is $\mathit{rfrontier}_l - \{y \mid \exists y' \in \mathit{rfrontier}_l \cdot y' \rhd y\}$.

Now given a problem in the GS class, if it can be shown that $\|\mathit{undom}_l\|$ for any l is polynomially bounded in the size of the input, a number of benefits accrue: (1) BFS can be used to tractably carry out the search, as implemented in the raw program schema of Alg. 1, (2) The raw schema of Alg. 1 can be transformed into an efficient tail recursive form, in which the entire frontier is passed down and (3) If additionally the tree depth can be polynomially bounded (which typically occurs for example in *constraint satisfaction problems* or CSPs [Dec03]) then, under some reasonable assumptions about the work being done at each node, the result is a polynomial-time algorithm for the problem.

3.1 Program Theory

A program theory for EBFS defines a recursive function which given a space y, computes a non-trivial subset $F_x(y)$ of the optimal solutions contained in y, where

$$F_x(y) = \mathit{opt}_c\{z \mid z \in y \wedge o(x, z)\}$$

[3] resolvable in DFS with additional programming effort

Algorithm 1 pseudo-Haskell Program Schema for EBFS (schema parameters underlined)

```
solve :: D -> {R}
solve(x) = bfs x {initial(x)}

bfs :: D -> {RHat}-> {R}
bfs x frontier =
  let localsof y = let z = extract x y
                   in if z!={} && o(x,z) then z else {}
      locals = (flatten.map) localsof frontier
      allsubs = (flatten.map) (subspaces x) frontier
      undom = {yy : yy∈allsubs &&
                    (yy'∈subs && yy' ‘dominates‘ yy ⇒ yy==yy')}
      subsolns = bfs x undom
  in opt(locals ∪ subsolns)

subspaces :: D -> RHat -> {RHat}
subspaces x y = {yy: split(x,y,yy))

opt :: {R} -> {R}
opt zs = min {c x z | z ∈zs}
```

opt_c is a subset of its argument that is the optimal set of solutions (w.r.t. the cost function c), defined as follows:

$$opt_c S = \{z \mid z \in S \wedge (\forall z' \in S \cdot c(z) \leq c(z'))\}$$

Also let $undom(y)$ be $undom_{l(y)+1} \cap \{yy \mid y \pitchfork yy\}$ where $l(y)$ is the level of y in the tree. The following theorem defines a recurrence that serves as the basis for computing $F_x(y)$:

Theorem 3.1. *Let $\pitchfork$ be a well-founded relation of GS-theory and $G_x(y) = opt_c(\{z \mid \chi(y,z) \wedge o(x,z)\} \cup \bigcup_{yy \in undom(y)} G_x(yy)\})$ be a recurrence. Then $G_x(y) \subseteq F_x(y)$.*

The theorem states that if the feasible solutions immediately extractable from a space y are combined with the solutions obtained from G_x of each undominated subspace yy, and the optimal ones of those retained, the result is a subset of $F_x(y)$. The next theorem demonstrate non-triviality[4] of the recurrence by showing that if a feasible solution exists in a space, then one will be found.

Theorem 3.2. *Let $\pitchfork$ be a well-founded relation of GS-Theory and G_x be defined as above. Then*

$$F_x(y) \neq \emptyset \Rightarrow (\{z \mid \chi(y,z) \wedge o(x,z)\} \cup \bigcup_{yy \in undom(y)} G_x(yy)\}) \neq \emptyset$$

Proofs of both theorems are in [NSC12]. From the characteristic recurrence we can straightforwardly derive a simple recursive function `bfs` to compute a non-trivial subset of F_x for a given y, shown in Alg. 1

The final program schema that is included in the Specware library is the result of incorporating a number of other features of GS such as necessary filters, bounds tests, and propagation, which are not shown here. Details of these and other techniques are in [Smi88].

[4] Non-triviality is similar but not identical to completeness. Completeness requires that every optimal solution is found by the recurrence, which we do not guarantee.

3.2 A class of strictly greedy algorithms (SG)

A greedy algorithm [CLRS01] is one which repeatedly makes a locally optimal choice. For some classes of problems this leads to a globally optimum choice. We can get a characterization of optimally greedy algorithms within EBFS by restricting the size of $undom_l$ for any l to 1. If $undom_l \neq \emptyset$ then the singleton member y^* of $undom_l$ is called the *greedy* choice.

A perhaps surprising result is that our characterization of greedy algorithms is broader than a well-known characterization of greedy solutions, namely the Greedy Algorithm over algebraic structures called *greedoids* [BZ92], which are themselves more general than *matroids.* We demonstrated this in earlier work [NSC10] although at the time we were not able to characterize the greedy class as a special case of EBFS.

Another interesting result is that even if $\|undom_l\|$, for any l, cannot be limited to one but can be shown to be some constant value, the resulting algorithm, we call *Hardly Strictly Greedy*[5] *(HSG),* still has the same complexity as a strictly greedy one. A number of interesting problems have the HSG property, and these are discussed later.Note that for problems in the SG class, there is no longer any "search" in the conventional sense.

4 Methodology

We strongly believe that every formal approach should be accompanied by a methodology by which it can be used by a competent developer, without needing great insights. Guided program synthesis already goes a long way towards meeting this requirement by capturing design knowledge in a reusable form. The remainder of the work to be done by a developer consists of instantiating the various parameters of the program schema. The second main contribution of this paper is to describe some techniques, illustrated with examples, that greatly simplify the instantiation process. We wish to reiterate that once the dominance relation and other operators in the schema have been instantiated, *the result is a complete solution to the given problem.* We focus on dominance relations because they are arguably the most challenging of the operators to design. The remaining parameters can usually be written down by visual inspection.

The simplest form of derivation is to reason backwards from the conclusion of $y \rightsquigarrow y' \Rightarrow o^*(x, y' \oplus e) \Rightarrow o^*(x, y \oplus e)$, while assuming $o^*(x, y' \oplus e)$. The additional assumptions that are made along the way form the required semi-congruence condition. The following example illustrates the approach.

Example 5. Derivation of the semi-congruence relation for Shortest Path in Eg. 1 is fairly straightforward calculation as shown in Fig 4.1. It relies on the specification of Shortest Path given in Eg. 1 and the GS-theory in Eg. 2.

The calculation shows that a path y is semi-congruent to y' if y and y' both end at the same node and additionally y is itself a valid path from the start node to its last node. Since the cost function is compositional, this immediately produces a dominance relation $y \rhd y' = last(y) = last(y') \wedge path?(y, x.start, n) \wedge \sum_{edge \in y} edge.cost \leq \sum_{edge' \in y'} edge'.cost$. Note the use of the distributive law for *path*? in step 4. Such laws are usually formulated as part of a domain theory during a domain discovery process, or even as part of the process of trying to carry out a derivation such as the one just shown. Given an appropriate constructive prover (such as the one in KIDS [Smi90]) such a derivation could in fact be automated. Other examples that have been derived using this approach are Activity Selection

[5] This name inspired by that of the *Hardly Strictly Bluegrass* festival held annually in San Francisco

$o^*(x, y \oplus e)$
$= \{\text{defn of } o^*\}$
$\exists z \cdot \chi(y \oplus e, z) \wedge o(x, z)$
$= \{\text{defn of } \chi\}$
$o(x, y \oplus e)$
$= \{\text{defn of } o\}$
$path?(y \oplus e, x.start, x.end)$
$= \{\text{distributive law for } path?\}$
$\exists n \cdot path?(y, x.start, n) \wedge path?(e, n, x.end)$
$\Leftarrow \{o^*(x, y' \oplus e), \text{ ie.} \exists m \cdot path?(y', x.start, m) \wedge path?(e, m, x.end). \text{ Let } m \text{ be a witness for } n\}$
$path?(y, x.start, m) \wedge path?(e, m, x.end)$
$= \{m = last(y).t, \text{ (where } last \text{ returns the last element of a sequence)}\}$
$last(y).t = last(y').t \wedge path?(y, x.start, n)$

Fig. 4.1. Derivation of semi-congruence relation for Shortest Path

[NSC10], Integer Linear Programming [Smi88], and variations on the Maximum Segment Sum problem [NC09]. The next two sections deal with situations in which the derivation is not so straightforward.

4.1 Technique 1: An exchange tactic

In the example just considered, and many such others, the derivation process was free of *rabbits* (Dijkstra's term for magic steps that appear seemingly out of nowhere). However, some cases are a little more challenging. As an example consider the following problem:

Example 6. One-Machine Scheduling. This is the problem of scheduling a number of jobs on a machine so as to minimize the sum of the completion times of the jobs (because dividing the sum of the completion times by the number of jobs gives the average amount of time that a job waits before being processed). A schedule is a permutation of the set of input jobs $\{J_1, J_2, \ldots J_n\}$. The input to the problem is a set of tasks, where a task consists of a pair of an *id* and duration, p. The result is a sequence of tasks. The output condition o requires that every task (and only those tasks) in the input be scheduled, ie placed at a unique position in the output sequence. Finally the cost of a solution, as stated above, is the sum of the completion times of the tasks. The problem specification is therefore:

$$\begin{aligned} D &\mapsto \{Task\} \\ R &\mapsto [Task] \\ &\quad Task = \langle id : Id, p : Time\rangle \\ o &\mapsto \lambda(x, z) \cdot asBag(z) = x \\ c &\mapsto \lambda(x, z) \cdot \textstyle\sum_{i=1}^{n} ct(z, i) \\ &\quad ct(z, i) = \textstyle\sum_{j=1}^{i} z_j.p \end{aligned}$$

The instantiation of terms in GS-theory is similar to that of Shortest Path:

$$\begin{aligned} &\widehat{R} \mapsto R \\ &\bot \mapsto \lambda x \cdot [] \\ &\pitchfork \mapsto \lambda(x, s, ss) \cdot \exists t \in x.\, ss = s +\!\!+ [t] \\ &\chi \mapsto \lambda(z, p) \cdot p = z \\ &\rhd \mapsto ? \end{aligned}$$

However, attempting to derive a semi-congruence relation in the same manner as we did for the Shortest Path problem by comparing two schedules αa and αb will not work. This is because every task must be scheduled, so any extension ω that extends say αa must contain b but as each task can be scheduled only once, such an extension will not be feasible for αb. Such situations are very common in scheduling and planning problems[6]. For such problems, note that when $\widehat{R}$ is a sequence type, every possible way a (called a *choice*) of extending some sequence α ie. α++$[a]$, written αa for conciseness, forms a subspace of α. A simple example is the problem of generating all bit strings. If the current space is some bit string say [1,0,0,1] then the two subspaces are [1,0,0,1]++[0] and [1,0,0,1]++[1] , written 10010 and 10011 resp. Another example occurs in CSP. If α is the sequence of assignments to the first i variables, then αv for every v in $\mathcal{D}_{\mathrm{i+1}}$ is a subspace of α. The tactic to try in such situations is to compare two partial solutions that are permutations of each other. This idea is backed up by the following theorem.

Theorem 4.1. *Suppose it can be shown that any feasible extension of αa must eventually be followed by some choice b. That is, any feasible solution contained in αa must be contained in $\alpha a\beta b$ for some β. Let $\alpha b\beta a$ be the partial solution obtained by exchanging a and b. If $R(\alpha, a, b)$ is an expression for the semi-congruence relation $\alpha b\beta a \rightsquigarrow \alpha a\beta b$ and $C(\alpha,a,b)$ is an expression for $c(\alpha b\beta a\gamma) \leq c(\alpha a\beta b\gamma)$, for any α, β, then $R(\alpha, a, b) \wedge C(\alpha, a, b)$ is a dominance relation $\alpha b \rhd \alpha a$.*

Proof. See [Ned12]

□

Example 6 Revisited. We now show how to derive a dominance relation for this problem. The tactic above suggests the following: Suppose some partial schedule is extended by picking task a to assign in the next position and this is followed subsequently by some task b. When is this better than picking b for the next position and a subsequently? Let $y = \alpha a\beta b$ and $y' = \alpha b\beta a$. It is not difficult to show that y and y' are unconditionally semi-congruent. To apply Theorem 4.1 it is necessary to derive an expression for $c(\alpha b\beta a\gamma) \leq c(\alpha a\beta b\gamma)$. Let $z = y\gamma$ and $z' = y'\gamma$ and let i be the position of a (b) in y (resp. y') and j be the position of b (a) in y (resp. y'). As shown in Fig. 4.2, the calculation is simple enough to be automated. The derivation shows that for any feasible solution $\alpha b\beta a\omega$ extending αb there is a cheaper feasible solution $\alpha a\beta b\omega$ that extends αa provided $a.p \leq b.p$. By Theorem 4.1, this constitutes the dominance relation $\alpha a \rhd \alpha b$. Finally, as $\leq$ is total order, there must be a choice that dominates all other choices, namely the task with the least processing time. Therefore the problem is in the SG class. Following this greedy choice at every step therefore leads to the optimum solution. Instantiating the library schema derived from Alg. 1 with such a dominance relation (along with the other parameters) immediately results in a greedy algorithm for this problem. The result corresponds to the Shortest Processing Time (SPT) rule, discovered by W.E. Smith in 1956. We have shown how it can be systematically derived.

We have applied the tactic above to derive other scheduling algorithms, for example an algorithm for the scheduling problem $1//L_m$ in which the goal is to minimize the maximum lateness of any job (amount by which it misses its due date), as well as variant of it to derive dominance relations for planning problems [Ned12].

[6] In planning, actions that must occur after another action to achieve a feasible plan are called *action landmarks*

$c(z) \le c(z')$
$= \{\text{unfold defn of } c\}$
$c(\alpha) + ct(z,i) + c(\beta) + ct(z,j) + c(\gamma) \le c(\alpha) + ct(z,j) + c(\beta) + ct(z,i) + c(\gamma)$
$= \{\text{unfold defn of } ct\text{. Realize that } c(\alpha) = \sum_{i=1}^{\|\alpha\|} \sum_{j=1}^{i} \alpha_j.p \text{ and let } pt(\alpha) = \sum_{j=1}^{\|\alpha\|} \alpha_j.p\}$
$c(\alpha) + pt(\alpha) + a.p + c(\beta) + pt(\alpha) + a.p + pt(\beta) + b.p$
$\le$
$ct(\alpha) + pt(\alpha) + b.p + c(\beta) + pt(\alpha) + b.p + pt(\beta) + a.p$
$= \{\text{algebra}\}$
$2(a.p) + b.p \le 2(b.p) + a.p$
$= \{\text{algebra}\}$
$a.p \le b.p$

Fig. 4.2. Calculation of cost comparison relation for 1 mach. scheduling

4.2 Technique 2: General Dominance

There are situations in which the above tactic will fail. Consider the following problem from [CLRS01] and [Cur03]:

Example 7. Professor Midas's Driving Problem

> Professor Midas wishes to plan a car journey along a fixed route. There are a given number of gas stations along the route, and the professor's gas tank when full can cover a given number of miles. Derive an algorithm that minimizes the number of refueling stops the professor must make.

The input data is assumed to be a sequence of cumulative distances of gas stations from the starting point (*cds*) along with the car's tank capacity (*cap*, measured in terms of distance). The variables will represent the gas stations along the route, that is variable i will be the ith gas station. A stop at a gas station is indicated in the result by assigning the corresponding variable *true*, and *false* otherwise. The start and finish are considered mandatory stops (that is z_1 and z_n are required to be *true*). Finally, the cost of a solution is a simple count of the number of variables assigned *true*. An obvious requirement on the input is that the distance between any two stations not exceed the tank capacity of the car. These ideas are captured in the following specification (in the cost function *false* is interpreted as 0 and *true* as 1). Note that a type $\langle \ldots \mid P\rangle$ denotes a predicate subtype in which the type members must satisfy the predicate P.

$$\begin{aligned}
D &\mapsto \langle cds : [Nat], cap : Nat \mid \forall x \in D \cdot \forall i < \|x.cds\| \cdot x.cds[i+1] - x.cds[i] \le x.cap\rangle \\
R &\mapsto [Boolean] \\
o &\mapsto \|z\| = \|x.cds\| \wedge fsok(x,z) \\
&\quad fsok(x,z) = \forall i,j \cdot i \le j \cdot didntStop(z,i,j) \Rightarrow span(x,i,j) \le x.cap \\
&\quad didntStop(z,a,b) = \forall i \cdot a \le i \le b \cdot \neg z_i \\
&\quad span(x,i,j) = x.cds[j+1] - x.cds[i-1] \\
c &\mapsto \lambda x, z \cdot \textstyle\sum_{i=1}^{\|z\|} z_i
\end{aligned}$$

The instantiation of GS-theory, with the exception of $\rhd$, is as it was for the machine scheduline example (Eg. 6). Attempting to apply the Exchange tactic described above and derive a semi-congruence relation between $\alpha T \beta F$ and $\alpha F \beta T$ (T is *true* and F is *false*) that does not depend on β will fail. The counter-example of Fig 4.3 shows why (boxes represent variables, shading means the variable was set true): it is possible that there is some extension e to αT which delays a stop but which is too long a span for αF . In such situations, we have found it useful to try to establish *general dominance* (Def. 2.2).

As before, it is useful to identify any distributive laws. In this case, the combination of partial solutions r and s satisfies *fsok* provided each partial solution independently satisfies *fsok* and where they abut satisfies *fsok*. Expressing the law formally requires broadening the definition of *fsok* somewhat to take into account the offset t of a particular sequence from the start, that is: $fsok(x, z, t) = \forall i, j \cdot i \leq j \wedge didntStop(z, i, j) \Rightarrow span(x, i{+}t, j{+}t) \leq x.cap$. Then:

$$fsok(x, y \oplus e, 0) = fsok(x, y, 0) \wedge fsok(x, e, \|y\|) \wedge fs2ok(x, y, e)$$

where *fs2ok* deals with the boundary between y and e and can be shown to be

$$fs2ok(x, y, e) = fsok(x, lfs(y){+\!\!+}ffs(e), \|y - lfs(y)\|)$$

where *ffs (*resp. *lfs)* denotes the initial (resp. last) false span of a segment, if any.

Now consider the two possible solutions after a split again, namely αT and αF. To demonstrate $o(x, \alpha F e)$ for some e, the usual backwards inference procedure can be applied, assuming $\alpha T e'$ for some e' (for brevity, the input x to *fsok* has been dropped)

$o(x, \alpha F e)$
$= \{\text{defn }\}$
$fsok(\alpha F e, 0)$
$= \{\text{defn }\}$
$fsok(\alpha, 0) \wedge fsok(F, \|\alpha\|) \wedge fs2ok(\alpha, F) \wedge fsok(e, \|\alpha\| + 1) \wedge fs2ok(\alpha F, e)$
$= \{fsok(\alpha, 0) \text{ because } o(x, \alpha T e'),\ fsok(F, -) \text{ because of restriction on } D\}$
$fs2ok(\alpha, F)\ \wedge\ fsok(e, \|\alpha\| + 1))\ \wedge\ fs2ok(\alpha F, e)$
$= \{\text{see below}\}$
$fs2ok(\alpha, F)$

To demonstrate both $fsok(e, \|\alpha\| + 1)$ and $fs2ok(\alpha F, e)$, let $e = e'[1 = T]$ (e' with the first variable assigned *true*). Clearly $fsok(e, \|\alpha\| + 1)$ if $fsok(e', \|\alpha\| + 1)$ and $fs2ok(\alpha F, e)$ if $fs2ok(\alpha, F)$ because $ffs(e)$ is empty. As αF has one stop less than αT and e has at most one extra, it follows that $c(x, \alpha F e) \leq c(x, \alpha T e')$. Therefore αF dominates αT provided there is sufficient fuel to make it to the next stop. As there are only two branches following a split, the greedy choice is clear. Informally this rule is to travel as far as possible without stopping.

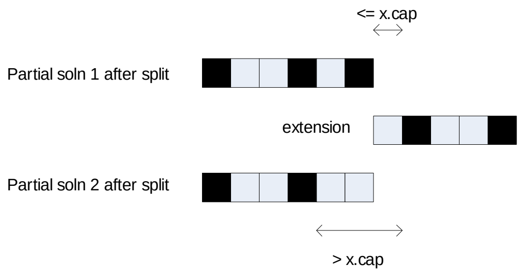

Fig. 4.3. Counter-example: extension works for the 1st partial soln but not for the 2nd

Other algorithms we have derived by applying general dominance have been a SG algorithm for Shortest Path similar to Dijkstra's algorithm, and SG algorithms similar to Prim and Kruskal for Minimum Spanning Trees [NSC12].

4.3 Technique 3: Feasibility Problems

Finally, we show that the notion of greediness applies not only to optimality problems, but also feasibility problems. By letting the "cost" of a solution be its correctness and using the standard ordering on Booleans, namely that $\mathit{false} < \mathit{true}$, we can derive a feasibility dominance criterion for $y \rhd_F y'$, namely $o(x, y') \Rightarrow o(x, y)$ [Ned12]. One way to use this constraint is derive conditions under which $o(x, y')$ is false, ensuring y' is dominated. An example of this follows.

Example 8. Searching for a key in an ordered sequence. A combined problem specification and GS-theory instantiation is:

$$\begin{aligned}
D &\mapsto \langle \mathit{seq} : [\mathit{Int}], \mathit{key} : \mathit{Int} \mid \mathit{unique}(\mathit{key}, \mathit{seq}) \wedge \mathit{ordered}(\mathit{seq}) \rangle \\
R &\mapsto \mathit{Nat} \\
o &\mapsto \lambda(x, z) \cdot x.\mathit{seq}[z] = x.\mathit{key} \\
\widehat{R} &\mapsto (\mathit{Nat}, \mathit{Nat}) \\
\pitchfork &\mapsto \lambda(x, (i, j), (k, l)) \cdot (k = i \wedge l = (i + j)\ \mathit{div}\ 2)) \vee \\
&\qquad\qquad\qquad\qquad\quad (k = (i + j)\ \mathit{div}\ 2) + 1 \wedge l = j) \\
\chi &\mapsto \lambda(y, z) \cdot z = y
\end{aligned}$$

The input D provides the sequence and the key, requiring that the sequence be ordered and the key occur uniquely in the sequence. The result is the index of the desired key. The two subspaces after a split are the sequence from the start i of the parent sequence to the midway point and from some point immediately after the midway to the end j of the parent sequence. (This split relation is derived in [Smi10]). In general, there could be an n-way split, or a split at any chosen point in the range but for simplicity, only the binary midpoint case is illustrated. There are only two subspaces after a split denoted L and R. Fig 4.4 derives the condition under which $o(x, \alpha L)$ holds. Negating this condition, ie. $x.\mathit{key} > x.\mathit{seq}[(i+j)\ \mathit{div}\ 2]$ determines when $o(x, \alpha L)$ is false and αL is dominated, leaving at most one undominated child, αR. Completing the instantiation of GS-theory with this dominance condition provides the bindings for the parameters of the program schema of Alg. 1. Since the depth of the search is $O(\log n)$, the result is an $O(\log n)$ greedy algorithm that implements Binary Search.

4.4 HSG problems

$$\begin{aligned}
&o(x, \alpha L) \\
&= \{\text{defn. of } o\} \\
&\exists z \in \alpha L \cdot o(x, z) \\
&= \{\text{defn. of } o\} \\
&\textstyle\bigvee_{p=i}^{(i+j)/2} x.\mathit{seq}[p] = x.\mathit{key} \\
&\Rightarrow \{\text{ordered elements}\} \\
&x.\mathit{key} \leq x.\mathit{seq}[(i + j)\ \mathit{div}\ 2]
\end{aligned}$$

Fig. 4.4. Derivation of greedy dominance relation for binary search

The problems illustrated so far have all been Strictly Greedy (SG). This was intentional. For one thing, many problems have a greedy solution (or a greedy approximation). Additionally, as one moves down an algorithm hierarchy, the narrower class generally has a more efficient algorithm. The price to be paid is that it is usually more difficult to establish the conditions necessary for membership in a tighter class. The techniques we have demonstrated for establishing membership in SG apply equally well to the broader category of HSG and indeed the catch-all one of EBFS. Although problems in the broader categories are seemingly sparser, we have demonstrated several problems are in HSG ; for example, 2-SAT (Boolean satisfiability in which there are at most 2 variables per clause) [Ned12] as well as a family of Segment Sum problems [NC09]. Noteworthy is that the run-time performance of the solutions we derived for the Segment Sum problems consistently exceeded those obtained by program transformation [SHT00,SHT01,SOH05]. Genetic algorithms in which the descendant population is maintained at a constant level are another example of HSG algorithms.

5 Related Work

Gulwani et al. [SGF10,GJTV11] describe a powerful program synthesis approach called *template-based synthesis*. A user supplies a template or outline of the intended program structure, and the tool fills in the details. A number of interesting programs have been synthesized using this approach, including Bresenham's line drawing algorithm and various bit vector manipulation routines. A related method is inductive synthesis [IGIS10] in which the tool synthesizes a program from examples. The latter has been used for inferring spreadsheet formulae from examples. All the tools rely on powerful SMT solvers. The Sketching approach of Solar-Lezama et al [PBS11] also relies on inductive synthesis. A *sketch*, similar in intent to a template, is supplied by the user and the tool fills in such aspects as loop bounds and array indexing. Sketching relies on efficient SAT solvers. To quote Gulwani et al. the benefit of the template approach is that "the programmer only need write the structure of the code and the tool fills out the details" [SGF10].Rather than the programmer supplying an arbitrary template, though, we suggest the use of a program schema from the appropriate algorithm class (refer to Step 2 of the process in Sec. 2.1). We believe that the advantage of such an approach is that, based on a sound theory, much can already be inferred at the abstract level and this is captured in the theory associated with the algorithm class. Furthermore, knowledge of properties at the abstract level allows specialization of the program schema with information that would otherwise have to either be guessed at by the programmer devising a template or inferred automatically by the tool (e.g. tail recursive implementation or efficient implementation of dominance testing with hashing). We believe this will allow semi-automated synthesis to scale up to larger problems such as constraint solvers (SAT, CSP, LP, MIP, etc.), planning and scheduling, and O/S level programs such as garbage collectors [PPS10].

Program verification is another field that shares common goals with program synthesis - namely a correct efficient program. The difference lies in approach - we prefer to construct the program in a way that is guaranteed to be correct, as opposed to verifying its correctness after the fact. Certainly some recent tools such as Dafny [Lei10] provide very useful feedback in an IDE during program construction. But even such tools requires significant program annotations in the form of invariants to be able to automatically verify non-trivial examples such as the Schorr-Waite algorithm [Lei10]. Nevertheless, we do not see verification and synthesis as being necessarily opposed. For example, ensuring the correctness of the instantiation of several of the operators in the program schema which is usually done by inspection is a verification task, as is ensuring correctness of the schema that goes in the class library. We also feel that recent advances in verification via SMT solvers will also help guided synthesis by increasing the degree of automation.

Refinement is generally viewed as an alternative to synthesis. A specification is gradually refined into an efficient executable program. Refinement methods such as Z and B have proved to be very popular. In contrast to refinement, guided program synthesis already has the program structure in place, and the main body of work consists of instantiating the schema parameters followed by various program transformations many of which can be mechanically applied. Both refinement and synthesis rely extensively on tool support, particularly in the form of provers. We expect that advances in both synthesis and refinement will benefit the other field.

Curtis [Cur03] presents a classification scheme for greedy algorithms. Each class has some conditions that must be met for a given algorithm to belong to that class. The greedy algorithm is then automatically correct and optimal. Unlike Curtis, our results extend beyond strictly greedy algorithms. We also rely extensively on calculational proofs for problem instances.

Another approach has been taken by Bird and de Moor [BM93] who show that under certain conditions a dynamic programming algorithm simplifies into a greedy algorithm. Our characterization in can be considered an analogous specialization of (a form of) branch-and-bound. The difference is that we do not require calculation of the entire program, but specific operators, which is a less onerous task.

6 Summary and Future Work

We have shown how Breadth-First Search can be carried out efficiently by relying on dominance relations. This is an important result as Breadth-First Search has several advantages over Depth-First Search. Secondly, we demonstrated some techniques by which dominance relations can be derived and illustrated them on several problems. We hope to identify and collect more techniques over time and catalogue then in the style of design patterns [GHJV95].

Nearly all the derivations shown in this paper have been carried out by hand. However, they are simple enough to be automated. We plan on building a prover that incorporates the ideas mentioned in here. We are encouraged by the success of a similar prover that was part of KIDS, a predecessor to Specware.

We are currently applying some of these ideas to the problem of synthesizing fast planners that produce good quality plans. We hope to report on this work in the near future.

References

[BM93] R. S. Bird and O. De Moor. From dynamic programming to greedy algorithms. In *Formal Program Development, volume 755 of Lecture Notes in Computer Science*, pages 43–61. Springer-Verlag, 1993.

[BZ92] A. Björner and G. M. Ziegler. Introduction to greedoids. In Neil White, editor, *Matroid Applications*. Cambridge University Press, 1992.

[CLRS01] T. Cormen, C. Leiserson, R. Rivest, and C. Stein. *Introduction to Algorithms*. MIT Press, 2nd edition, 2001.

[Cur03] S. A. Curtis. The classification of greedy algorithms. *Sci. Comput. Program.*, 49(1-3):125–157, 2003.

[Dec03] R Dechter. *Constraint Processing*. Morgan Kauffman, 2003.

[GHJV95] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design patterns: elements of reusable object-oriented software*. Addison-Wesley Professional, 1995.

[GJTV11] S. Gulwani, S. Jha, A. Tiwari, and R. Venkatesan. Synthesis of loop-free programs. In *PLDI*, pages 62–73, 2011.

[Iba77] T. Ibaraki. The power of dominance relations in branch-and-bound algorithms. *J. ACM*, 24(2):264–279, 1977.

[IGIS10] S. Itzhaky, S. Gulwani, N. Immerman, and M. Sagiv. A simple inductive synthesis methodology and its applications. In *OOPSLA*, pages 36–46, 2010.

[Kre98] C. Kreitz. Program synthesis. In W. Bibel and P. Schmitt, editors, *Automated Deduction – A Basis for Applications*, volume III, chapter III.2.5, pages 105–134. Kluwer, 1998.

[Lei10] K. R. M. Leino. Dafny: an automatic program verifier for functional correctness. In *Proc. 16th intl. conf. on Logic for Prog., AI, & Reasoning*, LPAR, pages 348–370, 2010.

[NC09] S. Nedunuri and W.R. Cook. Synthesis of fast programs for maximum segment sum problems. In *Intl. Conf. on Generative Prog. and Component Engineering (GPCE)*, Oct. 2009.

[Ned12] S. Nedunuri. *Theory and Techniques for Synthesizing Efficient Breadth-First Search Algorithms.* PhD thesis, Univ. of Texas at Austin, 2012.

[NSC10] S. Nedunuri, D. R. Smith, and W. R. Cook. A class of greedy algorithms and its relation to greedoids. *Intl. Colloq. on Theoretical Aspects of Computing (ICTAC)*, 2010.

[NSC12] S. Nedunuri, D. R. Smith, and W. R. Cook. Theory and techniques for synthesizing graph algorithms using breadth-first search. In *1st Workshop on Synthesis (SYNT) in Computer Aided Verification (CAV)*, 2012.

[PBS11] Y. Pu, R. Bodík, and S. Srivastava. Synthesis of first-order dynamic programming algorithms. In *OOPSLA*, pages 83–98, 2011.

[PPS10] D. Pavlovic, P. Pepper, and D. R. Smith. Formal derivation of concurrent garbage collectors. In *Math. of Program Constr. (MPC)*, 2010.

[S] Specware. http://www.specware.org.

[SGF10] S. Srivastava, S. Gulwani, and J. S. Foster. From program verification to program synthesis. In *POPL*, pages 313–326, 2010.

[SHT00] I. Sasano, Z. Hu, and M. Takeichi. Make it practical: A generic linear-time algorithm for solving maximum-weightsum problems. In *Intl. Conf. Functional Prog.(ICFP)*, 2000.

[SHT01] Isao Sasano, Zhenjiang Hu, and Masato Takeichi. Generation of efficient programs for solving maximum multi-marking problems. In *Proc. 2nd Intl. SAIG Workshop*, 2001.

[SLTB$^+$06] A. Solar-Lezama, L. Tancau, R. Bodik, S. Seshia, and V. Saraswat. Combinatorial sketching for finite programs. In *Proc. of the 12th intl. conf. on architectural support for prog. lang. and operating systems (ASPLOS)*, pages 404–415, 2006.

[Smi88] D. R. Smith. Structure and design of global search algorithms. Tech. Rep. Kes.U.87.12, Kestrel Institute, 1988.

[Smi90] D. R. Smith. Kids: A semi-automatic program development system. *IEEE Trans. on Soft. Eng., Spec. Issue on Formal Methods*, 16(9):1024–1043, September 1990.

[Smi10] D. R. Smith. Global search theory revisited. *Unpublished*, December 2010.

[SOH05] Isao Sasano, Mizuhito Ogawa, and Zhenjiang Hu. Maximum marking problems with accumulative weight functions. In *Proc. ICTAC*. Springer-Verlag, 2005.

[SPW95] D. R. Smith, E. A. Parra, and S. J. Westfold. Synthesis of high-performance transportation schedulers. Technical report, Kestrel Institute, 1995.

[SW08] D. R. Smith and S. Westfold. Synthesis of propositional satisfiability solvers. Final proj. report, Kestrel Institute, 2008.

[VY08] M. Vechev and E. Yahav. Deriving linearizable fine-grained concurrent objects. PLDI '08, pages 125–135, 2008.

[VYY10] M. Vechev, E. Yahav, and G. Yorsh. Abstraction-guided synthesis of synchronization. POPL '10, pages 327–338, 2010.