Synthesis Of Greedy Algorithms Using Dominance Relations

Srinivas Nedunuri
Dept. of Computer Sciences
University of Texas at Austin
nedunuri@cs.utexas.edu

Douglas R. Smith
Kestrel Institute
smith@kestrel.edu

William R. Cook
Dept. of Computer Sciences
University of Texas at Austin
cook@cs.utexas.edu

Abstract

Greedy algorithms exploit problem structure and constraints to achieve linear-time performance. Yet there is still no completely satisfactory way of constructing greedy algorithms. For example, the Greedy Algorithm of Edmonds depends upon translating a problem into an algebraic structure called a matroid, but the existence of such a translation can be as hard to determine as the existence of a greedy algorithm itself. An alternative characterization of greedy algorithms is in terms of dominance relations, a well-known algorithmic technique used to prune search spaces. We demonstrate a process by which dominance relations can be methodically derived for a number of greedy algorithms, including activity selection, and prefix-free codes. By incorporating our approach into an existing framework for algorithm synthesis, we demonstrate that it could be the basis for an effective engineering method for greedy algorithms. We also compare our approach with other characterizations of greedy algorithms.

1 Introduction

A greedy algorithm repeatedly makes a locally optimal choice. For some problems this can efficiently lead to a globally optimal solution. Edmonds [Edm71] characterized greedy algorithms in terms of *matroids*. In 1981, Korte and Lovasz generalized matroids to define *greedoids* [KLS91]. The question of whether a greedy algorithm exists for a particular problem reduces to whether there exists a translation of the problem into a matroid/greedoid. However, the characterization does not provide any guidance on how to construct this translation. In addition, there are problems that have greedy solutions, such as Activity Selection and Prefix-free Codes, [CLRS01], that do not seem to fit within the matroid/greedoid model. A number of other attempts have been made to characterize greedy algorithms, [BM93, Cur03, Cha95, HMS93] but the challenge in all of these approaches is establishing the conditions required for a given problem to meet that characterization. Thus, there has been very little work in helping a developer actually construct greedy algorithms.

An alternative approach to constructing algorithms is to take a very general program schema and specialize it with problem-specific information. The result can be a very efficient algorithm for the given problem, [SPW95, SW08, NC09]. One such class of algorithms, Global Search with Optimality (GSO) [Smi88], operates by controlled search, where at each level in the search tree there are a number of choices to be explored. We have recently [NSC10] been working on axiomatically characterizing a class of algorithms, called Greedy Global Search (GGS), that specializes GSO, in which this collection of choices reduces to a single locally optimal choice, which is the essence of a greedy algorithm. Our characterization is based on dominance relations [BS74], a well-known technique for pruning search spaces. However, this still leaves open the issue of deriving a greedy dominance relation for a given problem, which is what we address in this paper. We start with a specialized form of the dominance relation in [NSC10] which is easier to work with. Our contribution is to introduce a tactic which enables the two forms of dominance to be combined and also show how the main greediness axiom of GGS theory can be constructively applied. We have used this approach to derive greedy solutions to a number of problems, a couple of which are shown in this paper.

Although our derivations are currently done by hand, we have expressed them calculationally as we hope to eventually provide mechanical assistance for carrying them out. In addition to providing a

Algorithm 1 Program Schema for GGS Theory

```
--given x:D satisfying i returns optimal (wrt. cost fn c) z:R satisfying o(x,z)
function solve :: D -> {R}
solve x =
    if Φ(x,r̂0(x) ∧ i(x)) then (gsolve x r̂0(x) {}) else {}

function gsolve :: D -> {R̂} -> {R} -> {R}
gsolve x space soln =
    let gsubs = {s | s∈subspaces x space ∧ ∀ss ∈ subspaces x space,s δx ss}
        soln' = opt c (soln ∪{z | χ(z,space) ∧ o(x,z)})
    in if gsubs = {} then soln'
       else let greedy = arbPick gsubs in gsolve x greedy soln'

function opt :: ((D,R) -> C) -> {R̂}-> {R̂}
opt c {s} = {s}
opt c {s,t} = if c(x,s)>c(x,t) then {s} else {t}
function subspaces :: D -> R̂-> {R̂}
subspaces x r̂ = {ŝ: ŝ≼x r̂∧Φ(x,ŝ)}
```

process for a developer to systematically construct greedy algorithms, we also believe that our approach has a potential pedagogical contribution. To that end, we show that, at least in the examples we consider, our derivations are not only more systematic but also more concise than is found in algorithms textbooks.

2 Background

2.1 Greedy Global Search (GGS) Theory

GSO, the parent class of GGS, is axiomatically characterized in [Smi88]. The definition contains a number of type and abstract operators, which must be instantiated with problem specific information. GGS [NSC10] specializes GSO with an additional operator, and axioms. The operators of GGS theory, which we informally describe here, are named $\hat{r}_0, \chi, \in, \preccurlyeq, \delta, \Phi$ along with an additional type $\widehat{R}$; they parametrize the program schema associated with the GGS class (Alg. 1). D, R, C, o and c come from the problem specification which is described in Section 2.2.

Given a *space* of candidate solutions (also called a *partial solution*) to a given problem (some of which may not be optimal or even correct), a GGS program partitions the space into *subspaces* (a process known as *splitting*) as determined by a subspace relation $\preccurlyeq_x$. Of those subspaces that pass a *filter* (a predicate Φ which is some weakened efficiently evaluable form of the correctness condition, o) one subspace is greedily chosen, as determined by a dominance relation δ_x, and recursively searched[1]. If a predicate χ on the space is satisfied, a solution is extracted from it. If that solution is correct it is compared with the best solution found so far, using the cost function c. The process terminates when no space can be further partitioned. The starting point is an initial space, computed by a function $\hat{r}_0$, known to contain all possible solutions to the given problem. The result, if any, is an optimal solution to the problem. Because spaces can be very large, even infinite, they are rarely represented extensionally, but instead by a *descriptor* of some type $\widehat{R}$. A relation $\in$ determines whether a given solution is contained in a space.

The process of algorithm development using this theory consists of the following steps:

1. Formally specify the problem. Instantiate the types of GGS theory.
2. Develop a domain theory (usually associativity and distributivity laws) for the problem.

[1] Such an approach is also the basis of branch-and-bound algorithms, common in AI

3. Instantiate the abstract search-control operators in the program schema. This is often done by a mechanically-assisted constructive theorem proving process called *calculation* that draws on the domain theory formulated in the previous step
4. Apply further refinements to the program such as finite differencing, context simplification, partial evaluation, and datatype refinement to arrive at an efficient (functional) program.

Our focus in this paper is on Step 3, in particular how to derive the dominance relation, but we also illustrate Steps 1 and 2. Step 4 is not the subject of this paper. Specware [S], a tool from Kestrel Institute, provides support for carrying out such correctness preserving program transformations. Details can be found in [Smi90].

2.2 Specifications

A *problem specification* (Step 1) is a 6-tuple $\langle D, R, C, i, o, c\rangle$, where D is an input type (the type of the problem instance data), R an output type (the type of the result), C a cost type, $i : D \rightarrow Boolean$ is pre-condition defining what constitutes a valid input, $o : D \times R \rightarrow Boolean$ is an *output* or *post condition* characterizing the relationship between valid inputs and valid outputs. The intent is that an algorithm for solving this problem will take any input $x : D$ that satisfies i and return a *solution* $z : R$ that satisfies o (making it a *feasible* solution) for the given x. Finally $c : D \times R \rightarrow C$ is a *cost criterion* that the result must minimize. When unspecified, C defaults to Nat and i to $true$. A *constraint satisfaction problem* is one in which D specifies a set of variables and a value set for each variable, R a finite map from values to variables, and o requires at least that each of the variables be assigned a value from its given value-set in a way that satisfies some constraint.

2.3 Dominance Relations

A dominance relation provides a way of comparing two spaces in order to show that one will always have a cheaper best solution than the second. The first one is said to *dominate* the second, and the second can be eliminated from the search. Dominance relations have a long history in operations research, [BS74, Iba77]. For our purposes, let $\widehat{z}$ be a partial solution in some type of partial solutions $\widehat{R}$, and let $\widehat{z} \oplus e$ be a partial solution obtained by "extending" the partial solution with some extension $e : t$ for some problem-specific type t using an operator $\oplus : \widehat{R} \times t \rightarrow \widehat{R}$. The operator $\oplus$ has some problem-specific definition satisfying $\forall z \cdot z \in \widehat{z} \oplus e \Rightarrow z \in \widehat{z}$. Lift o up to $\widehat{R}$ by defining $o(x, \widehat{z}) = \exists z \cdot \chi(z, \widehat{z}) \wedge o(x, z)$. Similarly, lift c by defining $c(x, \widehat{z}) = c(x, z)$ exactly when $\exists! z \cdot \chi(z, \widehat{z})$. Then

Definition 1. *Dominance* is a relation $\delta \subseteq D \times \widehat{R}^2$ such that:

$$\forall x, \widehat{z}, \widehat{z}' \cdot \delta(x, \widehat{z}, \widehat{z}') \Rightarrow (\forall e' \cdot o(x, \widehat{z}' \oplus e') \Rightarrow \exists e \cdot o(x, \widehat{z} \oplus e) \wedge c(x, \widehat{z} \oplus e) \leq c(x, \widehat{z}' \oplus e'))$$

Thus, $\delta(x, \widehat{z}, \widehat{z}')$ is sufficient to ensure that $\widehat{z}$ will always lead to at least one feasible solution cheaper than any feasible solution in $\widehat{z}'$. For readability, $\delta(x, \widehat{z}, \widehat{z}')$ is often written $\widehat{z}\,\delta_x\,\widehat{z}'$. Because dominance in its most general form is difficult to demonstrate, we have defined a stronger form of dominance which is easier to derive. This stronger form of dominance is based on two additional concepts: Semi-congruence and extension dominance, which are now defined.

Definition 2. *Semi-Congruence* is a relation $\rightsquigarrow \subseteq D \times \widehat{R}^2$ such that

$$\forall x, \forall e, \widehat{z}, \widehat{z}' \cdot \rightsquigarrow (x, \widehat{z}, \widehat{z}') \Rightarrow o(x, \widehat{z}' \oplus e) \Rightarrow o(x, \widehat{z} \oplus e)$$

That is, semi-congruence ensures that any feasible extension of $\widehat{z}'$ is also a feasible extension of $\widehat{z}$. For readability, $\rightsquigarrow (x, \widehat{z}, \widehat{z}')$ is written $\widehat{z} \rightsquigarrow_x \widehat{z}'$.

And

Definition 3. *Extension Dominance* is a relation $\widehat{\delta} \subseteq D \times \widehat{R}^2$ such that

$$\forall x,e,\widehat{z},\widehat{z}' \cdot \widehat{\delta}(x,\widehat{z},\widehat{z}')' \Rightarrow\ o(x,\widehat{z}\oplus e) \wedge o(x,\widehat{z}' \oplus e) \Rightarrow c(x,\widehat{z}\oplus e) \leq c(x,\widehat{z}' \oplus e)$$

That is, extension dominance ensures that one feasible completion of a partial solution is no more expensive than the same feasible completion of another partial solution. For readability, $\widehat{\delta}(x,\widehat{z},\widehat{z}')$ is written $\widehat{z}\,\widehat{\delta}_x\,\widehat{z}'$. Note that both $\rightsquigarrow_x$ and $\widehat{\delta}_x$ are pre-orders. The following theorem and proposition show how the two concepts are combined.

Theorem 2.1. *Let c^* denote the cost of the best feasible solution in a space. If $\rightsquigarrow$ is a semi-congruence relation, and $\widehat{\delta}$ is an extension dominance relation, then*

$$\forall x,\widehat{z},\widehat{z}' \cdot \widehat{z}\widehat{\delta}_x\widehat{z}' \wedge \widehat{z} \rightsquigarrow_x \widehat{z}' \Rightarrow c^*(x,\widehat{z}) \leq c^*(x,\widehat{z}')$$

Proof. See Appendix □

It is not difficult to see that $\widehat{\delta}_x \cap \rightsquigarrow_x$ is a dominance relation.The following proposition allows us to quickly get an extension dominance relation for many problems. We assume we can apply the cost function to partial solutions.

Proposition 1. *If the cost domain C is a numeric domain (such as Integer or Real) and $c(x,\widehat{z}\oplus e)$ can be expressed as $\widehat{c}(x,\widehat{z}) + k(x,e)$ for some functions $\widehat{c}$ and k then $\widehat{\delta}_x$ where $\widehat{z}\,\widehat{\delta}_x\,\widehat{z}' = \widehat{c}(x,\widehat{z}) \leq \widehat{c}(x,\widehat{z}')$ is an extension dominance relation.*

Proof. See Appendix □

In addition to the dominance requirement from Theorem 2.1, there is an additional condition on δ, [NSC10]:

$$i(x) \wedge (\exists z \in \widehat{r} \cdot o(x,z)) \Rightarrow \\ (\exists z^* \cdot e(z^*,\widehat{r}) \wedge o(x,z^*) \wedge c(x,z^*) = c^*(\widehat{r})) \vee \exists \widehat{s}^* \lessdot_x \widehat{r}, \forall \widehat{s} \lessdot_x \widehat{r} \cdot \widehat{s}^*\ \delta_x\ \widehat{s} \quad (2.1)$$

This states that, assuming a valid input x, an optimal feasible solution z^* in a space $\widehat{r}$ that contains feasible solutions must be immediately extractable or a subspace $\widehat{s}^*$ of $\widehat{r}$ must dominate all the subspaces of $\widehat{r}$.

2.4 Notation

The following notation is used throughout the paper: $\mapsto$ is to be read as "instantiates to". A type declaration of the form $\{a : T, b : U, \cdots\}$ where T and U are types denotes a product type in which the fields are accessed by $a, b, \ldots$ using a "dot" notation $o.a, o.b$, etc. An instance of this type can be constructed by $\{a = v, b = w, \ldots\}$ where $v, w, \cdots$ are values of type $T, U, \cdots$ resp. The notation $o\{a_i = v, a_j = w, \ldots\}$ denotes the object identical to o except field a_i has the value v, a_j has w, etc. $[T]$ is the type of lists of elements of type T, as_i accesses the ith element of a list as, $[a]$ constructs a singleton list with the element a, $[a \mid as]$ creates a list in which the element a is prefixed onto the list as, and $as{+}{+}bs$ is the concatenation of lists as and bs, $as - bs$ is the list resulting from removing from as all elements that occur in bs. *first* and *last* are defined so that for a non-empty list $as = first(as){+}{+}[last(as)]$. Similarly, $\{T\}$ is the type of sets of elements of type T. $T \rightarrowtail U$ is the type of finite maps from T to U.

3 A Process For Deriving Greedy Algorithms

We first illustrate the process of calculating $\rightsquigarrow$ and $\widehat{\delta}$ by reasoning backwards from their definitions on a simple example.

Example 1. Activity Selection Problem [CLRS01]

> Suppose we have a set $S = \{a_i, a_2, \ldots, a_n\}$ of n proposed activities that wish to use a resource, such as a lecture hall, which can be used by only one activity at a time. Each activity a_i has a start time s_i and finish time f_i where $0 \leq s_i < f_i < \infty$. If selected, activity a_i takes place in the half-open time interval $[s_i, f_i)$. Activities a_i and a_j are compatible if the intervals $[s_i, f_i)$ and $[s_j, f_j)$ do not overlap. The activity selection problem is to select a maximum-size subset of mutually compatible activities.

Step 1 is to formally specify the problem. The input is a set of activities and a solution is subset of that set. Every activity is uniquely identified by an *id* and a start time (s) and finish time (f). The output condition requires that activities must be chosen from the input set, and that no two activities overlap. For convenience we define a precedence operator $\preceq$:

$$\begin{array}{rcl} D & \mapsto & \{Activity\} \\ & & Activity = \{id : Nat, s : Nat, f : Nat\} \\ R & \mapsto & \{Activity\} \\ o & \mapsto & \lambda(x,z) \cdot noOvp(x,z) \wedge z \subseteq x \\ & & noOvp(x,z) \doteq \forall i, j \in z \cdot i \neq j \Rightarrow i \preceq j \vee j \preceq i \\ & & i \preceq j = i.f \leq j.s \\ c & \mapsto & \lambda(x,z) \cdot \|z\| \end{array}$$

In order to devise a greedy algorithm, the question is what should the basis be for an optimal local choice? Should we always pick the activity that starts first? Or the activity that starts last? Or the activity that overlaps the least number of other activities? Or something else? We now show how to systematically arrive at the answer.

Most of the types and operators of GGS theory are straightforward to instantiate. We will just set $\widehat{R}$ to be the same as R. The initial space is just the empty set. The subspace relation $\lessdot$ splits a space by selecting an unchosen activity if one exists and adding it to the existing partial solution. The extract predicate χ can extract a solution at any time:

$$\begin{array}{rcl} \widehat{R} & \mapsto & R \\ \widehat{r_0} & \mapsto & \lambda x \cdot \emptyset \\ \lessdot & \mapsto & \lambda(x, \widehat{z}, \widehat{z'}) \cdot \exists a \in x - \widehat{z} \cdot \widehat{z'} = \widehat{z} \cup \{a\} \\ \chi & \mapsto & \lambda(z, \widehat{z}) \cdot z = \widehat{z} \\ \Phi & \mapsto & ? \\ \delta & \mapsto & ? \end{array}$$

The tricky bit is finding bindings for δ and Φ to complete the instantiation, which we will do in step 3. First, in step 2 we explore the problem and try and formulate a domain theory. The composition operator $\oplus$ is just $\cup$. The $\preceq$ relation can be lifted up to sets of activities by defining the start and finish times of a set of activities, namely $(u \oplus v).f = \max\{u.f, v.f\}$ and $(u \oplus v).s = \min\{u.s, v.s\}$. The following theorem will come in handy:

Theorem 3.1. $noOvp(s) \Leftrightarrow \exists s_1, \ldots s_n \subseteq s \cdot s = \bigcup_{1 \leq i \leq n} s_i \wedge (\forall i \cdot 1 \leq i < n \Rightarrow s_i \preceq s_{i+1} \wedge noOvp(s_i)) \wedge noOvp(s_n)$

$o(x, \widehat{y} \oplus \{a\} \oplus e)$
$= \{\text{defn}\}$
$noOvp(\widehat{y} \cup \{a\} \cup e) \wedge \widehat{y} \cup \{a\} \cup e \subseteq x$
$\Leftarrow \{\text{Theorem 3.1 above}\}$
$\widehat{y} \preceq \{a\} \preceq e \wedge noOvp(\widehat{y}) \wedge noOvp(e) \wedge \widehat{y} \cup \{a\} \cup e \subseteq x$
$\Leftarrow \{noOvp(\widehat{y}) \wedge noOvp(e) \wedge \widehat{y} \cup e \subseteq x \text{ by assumption}\}$
$\widehat{y} \preceq \{a\} \preceq e \wedge a \in x$
$\Leftarrow \{\widehat{y} \preceq \{a'\} \preceq e, \text{ apply transitivity}\}$
$\widehat{y} \preceq \{a\} \wedge a.f \leq a'.f \wedge a \in x$

Figure 3.1: Derivation of semi-congruence relation for Activity Selection

This says that any set of non-overlapping activities can be partitioned into internally non-overlapping subsets that follow each other serially.

For Step 3, we instantiate Definition 2 and reason backwards from its consequent, while assuming the antecedent. Each step of the derivation is accompanied by a hint (in $\{\}$) that justifies the step. Additional assumptions made along the way form the required semi-congruence condition. First note that a solution $z' \neq \emptyset$ can be expressed as $\widehat{y} \cup \{a'\} \cup e$ or alternatively $\widehat{y} \oplus \{a'\} \oplus e$, for some $\widehat{y}, a', e$, such that, by Theorem 3.1, $\widehat{y} \preceq \{a'\} \preceq e$. Now consider the feasibility of a solution $\widehat{y} \oplus \{a\} \oplus e$, obtained by switching out a' for some a, assuming $o(x, \widehat{y} \oplus \{a'\} \oplus e)$ as shown in Fig. 3.1

That is, $\widehat{y} \oplus a$ can be feasibly extended with the same feasible set of activities as $\widehat{y} \oplus a'$ provided $\widehat{y}$ finishes before a starts and a finishes before a' finishes and a is legal. By letting $\widehat{y} \oplus a$ and $\widehat{y} \oplus a'$ be subspaces following a split of $\widehat{y}$ this forms a semi-congruence condition between $\widehat{y} \oplus a$ and $\widehat{y} \oplus a'$. Since c is a distributive cost function, and all subspaces of a given space are the same size, the dominance relation equals the semi-congruence relation, by Proposition 1. Next, instantiating condition 2.1, we need to show that if $\widehat{y}$ contains feasible solutions, then in the case that any solution immediately extracted from $\widehat{y}$ is not optimal, (ie. the optimal lies in a subspace of $\widehat{y}$) there is *always* a subspace $\widehat{y} \oplus a$ that dominates every extension of $\widehat{y}$. Unfortunately, the dominance condition derived is too strong to be able to establish the instantiation of 2.1. Logically, what we established is a *sufficient* dominance test. That is $\widehat{y} \preceq \{a\} \wedge a.f \leq a'.f \wedge a \in x \Rightarrow \widehat{y} \oplus \{a\} \; \delta_x \; \widehat{y} \oplus \{a'\}$. How can we weaken it? This is where the filter Φ comes in. The following theorem shows how to construct a simple dominance relation from a filter:

Theorem 3.2. *Given a filter* Φ *satisfying* $\forall \widehat{z}' \cdot (\exists z \in \widehat{z}' \cdot o(x,z)) \Rightarrow \Phi(x, \widehat{z}')$, $\neg\Phi(x, \widehat{z}') \Rightarrow \forall \widehat{z} \cdot \widehat{z} \; \delta_x \; \widehat{z}'$.

The theorem says that a space that does not pass the necessary filter is dominated by any space.On a subspace $\widehat{y} \oplus a'$, one such filter (that can be mechanically derived by a tool such as KIDS [Smi90]) is $\widehat{y} \preceq \{a'\}$. Now, we can combine both dominance tests with the 1st order variant of the rule $(p \Rightarrow r \wedge q \Rightarrow r) \Rightarrow (p \vee q \Rightarrow r)$, reasoning backwards as we did above as shown in Fig. 3.2.

The binding for m above shows that 2.1 is satisfied by picking an activity in $x - \widehat{y}$ with the earliest finish time, after overlapping activities have been filtered out. Note how in verifying 2.1 we have extracted a witness which is the greedy choice. This pattern of witness finding is common across many examples. The program schema in Alg. 1 can now be instantiated into a greedy solution to the Activity Selection Problem.

In contrast to our derivation, the solution presented in [CLRS01] starts off by assuming the tasks are sorted in order of finishing time. Only after reading the pseudocode and a proof is the reason for this clear (though how to have thought of it a priori is still not!). For us, the condition falls out of the process of investigating a possible dominance relation. Note that had we partitioned the solution $\widehat{y} \oplus \{a'\} \oplus e$

$\exists a \in x - \widehat{y}, \forall a' \in x - \widehat{y} \cdot \widehat{y} \oplus a \, \delta_x \widehat{y} \oplus a'$
$\Leftarrow \{\widehat{y} \preceq \{a\} \wedge a.f \leq a'.f \wedge a \in x \Rightarrow \widehat{y} \oplus \{a\} \, \delta_x \widehat{y} \oplus \{a'\} \; \& \widehat{y} \not\preceq \{a'\} \Rightarrow \widehat{y} \oplus \{a\} \, \delta_x \widehat{y} \oplus \{a'\}\}$
$\exists a \in x - \widehat{y}, \forall a' \in x - \widehat{y} \cdot \widehat{y} \not\preceq \{a'\} \vee (\widehat{y} \preceq \{a\} \wedge a.f \leq a'.f)$
$= \{\text{logic}\}$
$\exists a \in x - \widehat{y} \cdot \widehat{y} \preceq \{a\} \wedge \forall a' \in x - \widehat{y} \cdot \widehat{y} \preceq \{a'\} \Rightarrow a.f \leq a'.f$
$= \{\text{logic}\}$
$\exists a \in x - \widehat{y} \cdot \widehat{y} \preceq \{a\} \wedge \forall a' \in x - \widehat{y} \cap \{b \mid \widehat{y} \preceq \{b\}\} \cdot a.f \leq a'.f$
$= \{\text{define } a \leq b = a.f \leq b.f\}$
$\exists a \in x - \widehat{y} \cdot \widehat{y} \preceq \{a\} \wedge \forall a' \in x - \widehat{y} \cap \{b \mid \widehat{y} \preceq \{b\}\} \cdot m \leq a' \Rightarrow a.f \leq a'.f$
where $m = \min_{\leq} x - \widehat{y} \cap \{b \mid \widehat{y} \preceq \{b\}\}$
$= \{\text{law for monotone } p : (\forall x \in S \cdot m \leq x \Rightarrow p(x)) \equiv p(m)\}$
$\exists a \in x - \widehat{y} \cdot \widehat{y} \preceq \{a\} \wedge a.f \leq m.f$ where $m = \min_{\leq} x - \widehat{y} \cap \{b \mid \widehat{y} \preceq \{b\}\}$
$= \{\text{law for anti-monotone } p : (\exists x \in S \cdot m \leq x \wedge p(x)) \equiv p(m)\}$
$m.f \leq m.f$ where $m = \min_{\leq} x - \widehat{y} \cap \{b \mid \widehat{y} \preceq \{b\}\}$
$=$
$true$

Figure 3.2:

differently as $e \preceq \{a'\} \preceq \widehat{y}$, we would have arrived at an another algorithm that grows the result going backwards rather than forwards, which is an alternative to the solution described in [CLRS01].

Next we apply our process to the derivation of a solution to a fairly non-trivial problem, that of determining optimum prefix-free codes.

Example 2. Prefix-Free Codes

> Devise an encoding, as a binary string, for each of the characters in a given text file so as to minimize the overall size of the file. For ease of decoding, the code is required to be *prefix-free*, that is no encoding of a character is the prefix of the encoding of another character (e.g. assigning "0" to 'a' and "01" to 'b' would not be allowed).

D.A. Huffman devised an efficient greedy algorithm for this in 1952. We show how it can be systematically derived. Step 1 is to specify the problem. The input is a table of character frequencies, and the result is a table of bit strings, one for each character in the input, satisfying the prefix free property.

$$\begin{array}{rcl} D & \mapsto & Char \rightarrowtail Frequency \\ & & Char = Frequency = Nat \\ R & \mapsto & Char \rightarrowtail [Boolean] \\ o & \mapsto & \lambda x, z.\ dom(z) = dom(x) \wedge \forall c \neq c' \in dom(z) \cdot \neg prefixOf(z(c), z(c')) \\ & & prefixOf(s,t) = \exists u \cdot t = s +\!\!+ u \vee s = t +\!\!+ u \\ c & \mapsto & \lambda x, z.\ \sum_{c \in dom(z)} \|z(c)\| \times x(c) \end{array}$$

Often a good way of ensuring a condition is to fold it into a data structure. Then the rules for constructing that data structure ensure that the condition is automatically satisfied. It turns out that a binary tree in which the leaves are the letters, and the path from the root to the leaf provides the code for that letter, ensures that the resulting codes are automatically prefix-free[2]. One obvious way to construct

[2]It is possible to systematically derive this datatype by starting with an obvious datatype such as a set of binary strings and folding in a required property such as prefix-freeness. However, we do not pursue that here

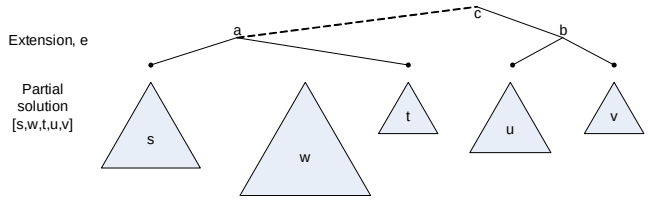

Figure 3.3: An extension applied to a partial solution with 5 trees

such a tree is by merging together smaller trees. This will be the basis of the $\lessdot$ relation: the different subspaces arise from the $t(t-1)/2$ choices of pairs of trees to merge, where t is the number of trees at any point in the search. The starting point $\widehat{r}_0$ is the ordered collection of leaves representing the letters. The extract predicate χ checks there is only one tree left and generates the path to each leaf (character) of the tree, which becomes the code for that character. With this information, we can instantiate the remaining terms in GGS, except for δ ($\langle\rangle$ is the non-elementary*BinTree* constructor)

$$\begin{array}{lcl}
\widehat{R} & \mapsto & [BinTree] \\
\widehat{r}_0 & \mapsto & \lambda x \cdot asList(dom(x)) \\
\lessdot & \mapsto & \lambda(x,\widehat{z},\widehat{z}') \cdot \exists s,t \in \widehat{z}.\ \widehat{z}' = [\langle s,t\rangle \mid (\widehat{z}-s-t)] \\
\chi & \mapsto & \lambda(z,\widehat{z}) \cdot \|\widehat{z}\| = 1 \wedge \forall p \in paths(\widehat{z}) \cdot z(last(p)) = first(p) \\
 & & paths(\langle s,t\rangle) = (map\,prefix0\,paths(s)) +\!\!+ (map\,prefix1\,paths(t)) \\
 & & paths(l) = [l] \\
 & & prefix0(p) = [0 \mid p],\ prefix1(p) = [1 \mid p] \\
\Phi & \mapsto & true \\
\delta & \mapsto & ?
\end{array}$$

One interesting difference between this problem and the Activity Selection problem is that every subspace leads to a feasible solution. For that reason, Φ is just *true*. It is not obvious what the criterion should be for a greedy choice. Should it be to combine the trees with the least number of leaves, or those with the least height, or something else? To proceed with our process and apply Defs. 2 and 3, we will define $\oplus$ as a left-associative binary operator whose net effect is to merge two *BinTree*s from its left argument together into another *BinTree*. The right argument specifies which trees are to be merged. That is, $\widehat{z} \oplus (i,j) = [\langle \widehat{z}_i, \widehat{z}_j \rangle \mid \widehat{z} - \widehat{z}_i - \widehat{z}_j]$. For example, Fig 3.3 shows the merger of trees s and t and the merger of trees u and v in a partial solution $\widehat{z}$ to form two subtrees with roots a and b. This is described by the expression $(\widehat{z} \oplus (1,3)) \oplus (3,4) = \widehat{z} \oplus (1,3) \oplus (3,4)$. The extension in this case is $(1,3) \oplus (3,4)$. A semi-congruence condition is shown in Fig. 3.4. Assuming $o(x,\widehat{z}' \oplus e)$ and a definition of $lvs(\widehat{z}) = map\,last\,(paths(\widehat{z}))$. (Note we omit the remainder of the output condition since it is implied by the Binary Tree construction) This says that any two partial solutions of the same size are extensible with the same feasible extension. This semi-congruence condition is trivially satisfied by any two subspaces of a split. For the extension dominance condition, it is easy to show that $c(x,\widehat{z} \oplus e)$ can be expressed as $\widehat{c}(x,\widehat{z}) + k(x,e)$ for some k where $\widehat{c}(x,\widehat{z}) = \sum_{i=1}^{|lvs(\widehat{z})|} d(\widehat{z})(i) \cdot x(lvs(\widehat{z})_i)$ where $d(\widehat{z})$ is a function returning the depth of its argument leaf within the tree $\widehat{z}$, and therefore by Prop. 1, it is sufficient to determine conditions under which $\widehat{c}(x,\widehat{z}) \leq \widehat{c}(x,\widehat{z}')$. However, if we try to calculate such a condition as we have done for semi-congruence we will end up with an expression that involves the depths of individual leaves in the trees. Is there a simpler form? We have been investigating ways to provide a developer with hints about how to proceed. We call these *tactics*. In earlier work [NSC09] we introduced tactics for the derivation of operators for non-optimization problems. We now introduce a tactic for dominance relations. We have used this tactic to derive greedy solutions for a number of problems, including Machine Scheduling, several variations on the Maximum Segment Sum Problem,[NC09], Minimum Spanning

$o(x, \widehat{z} \oplus e)$
$= \{\text{defn of } o \text{ on } \widehat{R}\}$
$\exists z \cdot \chi(z, \widehat{z} \oplus e) \wedge o(x,z)$
$= \{\text{defn of } \chi, o\}$
$\exists z \cdot \|\widehat{z} \oplus e\| = 1 \wedge \forall p \in paths(\widehat{z} \oplus e) \cdot z(last(p)) = first(p) \wedge dom(z) = x \wedge \cdots$
$= \{\text{intro defn}\}$
$\|\widehat{z} \oplus e\| = 1 \wedge dom(z) = x \wedge \cdots$
where $z = \{last(p) \mapsto first(p) \mid p \in paths(\widehat{z} \oplus e)\}$
$\Leftarrow \{o(x, \widehat{z}' \oplus e) \Rightarrow dom(z') = x \text{ where } z = \{last(p) \mapsto first(p) \mid p \in paths(\widehat{z}' \oplus e)\}\}$
$\|\widehat{z} \oplus e\| = 1 \wedge asSet(lvs(\widehat{z})) = asSet(lvs(\widehat{z}'))$
$= \{\text{split does not alter set of leaves}\}$
$\|\widehat{z} \oplus e\| = 1$
$= \{\|\widehat{z} \oplus e\| = \|\widehat{z}\| - \|e\|, \|\widehat{z}' \oplus e\| = 1\}$
$\|\widehat{z}\| = \|\widehat{z}'\|$

Figure 3.4: Derivation of extension dominance relation for Huffman problem

$c(\widehat{z}) \leq c(\widehat{z}')$
$=$ $\{\text{unfold defn of } c\}$
$\sum_{i=1}^{|lvs(s)|}(d(s)(i)+h) \cdot x(lvs(s)_i) + \sum_{i=1}^{|lvs(t)|}(d(t)(i)+h) \cdot x(lvs(t)_i)$
$+ \sum_{i=1}^{|lvs(u)|}(d(u)(i)+2) \cdot x(lvs(u)_i) + \sum_{i=1}^{|lvs(v)|}(d(v)(i)+2) \cdot x(lvs(v)_i)$
$\leq$
$\sum_{i=1}^{|lvs(u)|}(d(u)(i)+h) \cdot x(lvs(u)_i) + \sum_{i=1}^{|lvs(v)|}(d(v)(i)+h) \cdot x(lvs(v)_i)$
$+ \sum_{i=1}^{|lvs(s)|}(d(s)(i)+2) \cdot x(lvs(s)_i) + \sum_{i=1}^{|lvs(t)|}(d(t)(i)+2) \cdot x(lvs(t)_i)$
$=$ $\{\text{algebra}\}$
$(h-2) \cdot \sum_{i=1}^{|lvs(s)|} x(lvs(s)_i) + (h-2) \cdot \sum_{i=1}^{|lvs(t)|} x(lvs(t)_i)$
$\leq (h-2) \cdot \sum_{i=1}^{|lvs(u)|} x(lvs(u)_i) + (h-2) \cdot \sum_{i=1}^{|lvs(v)|} x(lvs(v)_i)$
$\Leftarrow$ $\{\text{algebra}\}$
$\sum_{i=1}^{|lvs(s)|} x(lvs(s)_i) + \sum_{i=1}^{|lvs(t)|} x(lvs(t)_i) \leq \sum_{i=1}^{|lvs(u)|} x(lvs(u)) + \sum_{i=1}^{|lvs(v)|} x(lvs(v)_i) \wedge h > 2$

Figure 3.5:

Tree, and Professor Midas' Driving Problem.

Exchange Tactic: Try to derive a dominance relation by comparing a partial solution $\widehat{y} \oplus a \oplus \alpha \oplus b$ (assuming some appropriate parenthesization of the expression) to a variant obtained by exchanging a pair of terms, that is, $\widehat{y} \oplus b \oplus \alpha \oplus a$, with the same parenthesization

Given a partial solution $\widehat{y}$, suppose trees s and t are merged first, and at some later point the tree containing s and t is merged with a tree formed from merging u and v (tree *satcubv* in Fig. 3.3), forming a partial solution $\widehat{z}$. Applying the exchange tactic, when is this better than a partial solution $\widehat{z}'$ resulting from swapping the mergers in $\widehat{z}$, ie merging u and v first and then s and t? Let $d(T)_i$ be the depth of leaf i in a tree T, and let h be the depth of s (resp. u) from the grandparent of u (resp. s) in $\widehat{z}$ (resp $\widehat{z}'$), (the distance from c to the root of s in Fig. 3.3). The derivation of the extension dominance relation is shown in Fig. 3.5.

That is, if the sum of the frequencies of the leaves of s and t is no greater than the sum of the frequencies of leaves of u and v then s and t should be merged before u and v. (The condition $h > 2$

simply means that no dominance relation holds when $\langle u,v\rangle$ is immediately merged with $\langle s,t\rangle$. It is clear in that case that the tree is balanced). How does this help us derive a dominance relation between two subspaces after a split? The following theorem shows that the above condition serves as a dominance relation between the two subspaces $[\langle s,t\rangle \mid (\widehat{y}-s-t)]$ and $[\langle u,v\rangle \mid (\widehat{y}-u-v)]$:

Theorem 3.3. *Given a GGS theory for a constraint satisfaction problem,* $(\exists\alpha \cdot (\widehat{y}\oplus a\oplus\alpha\oplus b)\,\delta_x\,(\widehat{y}\oplus b\oplus\alpha\oplus a)) \Rightarrow \widehat{y}\oplus a\;\delta_x\;\widehat{y}\oplus b$

By using a witness finding technique to verify condition 2.1 as we did for Activity Selection, we will find that the greedy choice is just the pair of trees whose sums of letter frequencies is the least. This is the same criterion used by Huffman's algorithm. Of course, for efficiency, in the standard algorithm, a stronger dominance test is used: $\sum_{i=1}^{|lvs(s)|} x(lvs(s)_i) \leq \sum_{i=1}^{|lvs(u)|} x(lvs(u)_i) \wedge \sum_{i=1}^{|lvs(t)|} x(lvs(t)_i) \leq \sum_{i=1}^{|lvs(v)|} x(lvs(v)_i)$ and the sums are maintained at the roots of the trees as the algorithm progresses. We would automatically arrive at a similar procedure after applying finite differencing transforms, [Smi90, NC09]. In contrast to our stepwise derivation, in most presentations of Huffman's algorithm, (e.g. [CLRS01]) the solution is presented first, followed by an explanation of the pseudocode, and then several pages of lemmas and theorems justifying the correctness of the algorithm. The drawback of the conventional approach is that the insights that went into the original algorithm development are lost, and have to be reconstructed when variants of the problem arise. A process for greedy algorithm development, such the one we have proposed here, is intended to remedy that problem.

4 Related Work

Curtis [Cur03] has a classification scheme for greedy algorithms. Each class has a 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, we are not attempting a classification scheme. Our goal is to simplify the process of creating greedy algorithms. For that reason, we present derivations in a calculational style whenever the exposition is clear. In contrast, Curtis derives the "meta-level" proofs, namely that the conditions attached to a given algorithm class in the hierarchy are indeed correct, calculationally but the "object-level" proofs, namely those showing a given problem formulation does indeed meet those conditions, are done informally. We believe that this should be the other way around. The meta-level proofs are (hopefully) carried out only a few times and are checked by many, but the object level proofs are carried out by individual developers, and are therefore the ones which ought to be done calculationally, not only to keep the developer from making mistakes but also with a view to providing mechanical assistance (as was done in KIDS, a predecessor of Specware). Another difference between our work and Curtis is that while Curtis's work is targeted specifically at greedy algorithms, for us greedy algorithms are just a special case of a more general problem of deriving effective global search algorithms. In the case that the dominance relation really does not lead to a singleton choice at each split, it can still prove to be highly effective. This was recently demonstrated on some Segment Sum problems we looked at, [NC09]. Although the dominance relation we derived for those problem did not reduce to a greedy choice, it was nonetheless key to reducing the complexity of the search (the width of the search tree was kept constant) and led to a very efficient breadth-first solution that was much faster than comparable solutions derived by program transformation.

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 [NSC10] 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.

Helman [Hel89] devised a framework that unified branch-and-bound and dynamic programming. The framework also incorporated dominance relations. However, Helman's goal was the unification of the two paradigms, and not the process by which algorithms can be calculated. In fact the unification, though providing a very important insight that the two paradigms are related at a higher level, arguably makes the derivation of particular algorithms harder.

Charlier [Cha95], also building on Smith's work, proposed a new algorithm class for greedy algorithms that embodied the matroid axioms. Using this class, he was able to synthesize Kruskal's MST algorithm and a solution to the $1/1/\sum T_i$ scheduling problem. However he reported difficulty with the equivalent of the Augmentation (also called Exchange) axiom. The difficulty with a new algorithm class is often the lack of a repeatable process for synthesizing algorithms in that class, and this would appear to be what Charlier ran up against. In contrast, we build on top of the GSO class, adding only what is necessary for our purposes. As a result we can handle a wider class of algorithms than would belong in Charlier's Greedy class, such as Prim's and Huffman's.

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.

[BS74] K.R. Baker and Z-S. Su. Sequencing with due-dates and early start times to minimize maximum tardiness. *Naval Research Logistics*, 21(1):171–176, 1974.

[Cha95] B. Charlier. The greedy algorithms class: formalization, synthesis and generalization. Technical report, 1995.

[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.

[Edm71] J. Edmonds. Matroids and the greedy algorithm. *Math. Programming*, 1(1):127–136, 1971.

[Hel89] P. Helman. A common schema for dynamic programming and branch and bound algorithms. *J. ACM*, 36(1):97–128, 1989.

[HMS93] P. Helman, B. M. E. Moret, and H. D. Shapiro. An exact characterization of greedy structures. *SIAM J. on Discrete Math.*, 6:274–283, 1993.

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

[KLS91] B. Korte, L. Lovasz, and R. Schrader. *Greedoids*. Springer-Verlag, 1991.

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

[NSC09] S. Nedunuri, D. R. Smith, and W. R. Cook. Tactical synthesis of efficient global search algorithms. In *Proc. NASA Symposium on Formal Methods*, April 2009.

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

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

[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.

[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.

5 Appendix: Proofs of Theorems

5.1 Proofs of Theorem 2.1 and Proposition 1:

Theorem 2.1: If $\rightsquigarrow$ is a semi-congruence relation, and $\widehat{\delta}$ is a extension dominance relation, then

$$\forall x, \forall \widehat{z}, \widehat{z}' \cdot \widehat{z} \widehat{\delta}_x \widehat{z}' \wedge \widehat{z} \rightsquigarrow_x \widehat{z}' \Rightarrow c^*(x, \widehat{z}) \leq c^*(x, \widehat{z}')$$

Proof. By contradiction. (input argument x dropped for readability). Suppose that $\widehat{z} \widehat{\delta}_x \widehat{z}' \wedge \widehat{z} \rightsquigarrow_x \widehat{z}'$ but $\exists z'^* \in \widehat{z}', O(z'^*) \wedge c(z'^*) < c^*(\widehat{z})$, that is $c(z'^*) < c(z)$ for any feasible $z \in \widehat{z}$. We can write z'^* as $\widehat{z}' \oplus e$ for some e. Since z'^* is cheaper than any feasible $z \in \widehat{z}$, specifically it is cheaper than $z = \widehat{z} \oplus e$, which by the semi-congruence assumption and Definition 2, is feasible. But by the extension dominance assumption, and Definition 3, this means $c(z) \leq c(z'^*)$, contradicting the initial assumption. □

Proposition 1: If the cost domain C is a numeric domain (such as *Integer* or *Real*) and $c(x, \widehat{z} \oplus e)$ can be expressed as $\widehat{c}(x, \widehat{z}) + k(x, e)$ for some functions $\widehat{c}$ and k then $\widehat{\delta}_x$ where $\widehat{z} \widehat{\delta}_x \widehat{z}' = \widehat{c}(x, \widehat{z}) \leq \widehat{c}(x, \widehat{z}')$ is an extension dominance relation

Proof. By showing that Definition 3 is satisfied. $c(\widehat{z} \oplus e) \leq c(\widehat{z}' \oplus e) = \widehat{c}(\widehat{z}) + k(e) \leq \widehat{c}(\widehat{z}') + k(e)$ by distributivity of c which is just $c(\widehat{z}) \leq c(\widehat{z}')$ after arithmetic. □