FORMAL ALGORITHM DESIGN APPROACHES FOR DYNAMIC PROGRAMMING AND GREEDY ALGORITHMS

LEILA MOFARAH-FATHI

Formal Algorithm Design Approaches for Dynamic Programming and Greedy Algorithms

by

© Leila Mofarah-Fathi

A thesis submitted to the

School of Graduate Studies

in partial fulfilment of the

requirements for the degree of

Master of Engineering

Faculty of Engineering and Applied Science

Memorial University

May 2010

St. John's Newfoundland

Abstract

In this thesis we present a formal study of greedy algorithms and dynamic programming in a predicative framework. A simple approach is presented based on specialization of an abstract algorithm representing an algorithm design approach. This provides not only reuse of the algorithm, but also reuse of its proof. Moreover, the simplicity and applicability of the design techniques are not sacrificed. For each method, a problem is parameterized to create a specification, which is then transformed to a concrete algorithm following the proposed process.

Acknowledgment

I am sincerely thankful to my supervisor, Dr. Theodore S. Norvell, for his very helpful suggestions, discussions, support and guidance through all phases of this research work. It is a pleasure to thank Engineering Faculty and staff who made this thesis possible with providing an excellent environment.

I am grateful to friends who made their moral support always available in the course of my studies, in particular for the coffee breaks.

I owe my deepest gratitude to my parents, Fatemeh and Hossein, for always believing in me and their unconditional support.

Finally, I am grateful to the love of my life, Reza, for his encouragements, care and support in numerous ways throughout the course of this Master work.

Contents

Chapter 1

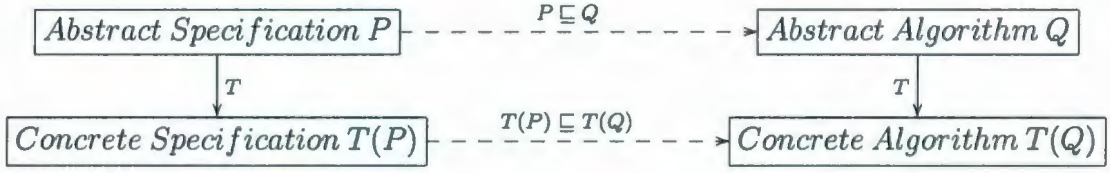
Introduction

Algorithm generally refers to a *concrete algorithm*. However, there is a broader perspective to algorithms known as *abstract algorithms*. To form an abstract algorithm, a deeper understanding of the abstract input, and output is required. The abstract algorithm excludes the optimization details of an algorithm, or data structure and datatypes for implementation purposes. To develop an abstract algorithm, the main concept is the knowledge to design an algorithm.

Moreover, algorithm design approaches, such as *greedy algorithms*, *dynamic programming*, *divide-and-conquer*, and binary search, are generally taught and understood as informal ideas. Can we capture each algorithmic approach formally?

We are investigating how *abstract specification*s can be proved to be implemented by *abstract algorithms*. For this study we consider algorithm design techniques including dynamic programming and greedy algorithms. Therefore, by applying a transformation that maps the abstract specification into a concrete specification, we can derive a concrete algorithm from the abstract algorithm. With the derived method

comes along a *formal proof* of abstract algorithm correctness. This allows the abstract algorithm to be reused, along with its proof, to solve multiple *concrete problems.* The approach is summarized as follows. Suppose we know that an abstract specification P is refined by an abstract algorithm Q, then if we need an algorithm for a problem $R = T(P)$, where T is a data transform, we can refine R with the same transformation $T(Q)$.



One of the design approaches presented in this thesis is dynamic programming which is a recursive approach to solving instances of problems by creating subinstances and using the obtained solutions to the subinstances to create a solution to the original instance. It is mostly applied to optimization problems, however, it also covers some non-optimization problems. The idea of dynamic programming is been taught in many books on algorithms such as [27, 13]. Its history goes back to 1955 when R. Bellman systematically studied this approach [1]. There have been other studies by other authors on its applications, and computational complexity. However, the main purpose of this thesis is neither studying concrete algorithms of dynamic programming independently nor the study of their complexity. It is rather on how to derive concrete dynamic programming algorithms from abstract algorithms.

This approach has been studied by researchers such as Bird and de Moor who have published papers such as [4, 3] and a book [2] on how to to calculate programs. In these studies algorithm design techniques such as dynamic programming, greedy algorithms, exhaustive search and divide-and-conquer are approached in an algebraic

form of programming. The algebraic approach can be applied to derive individual programs and is also a tool for studying general principles of programming, in particular those concerning optimization problems. Bird and de Moor theories make use of the categorical calculus of relations as a mathematical framework, providing the possibility of abstracting away the datatypes. This framework helps in formulating theorems and proofs. The calculus consists of two methods of reasoning, *pointfree* and *pointwise*. The former is reasoning based on functions and relations, and effectively avoids the use of quantifiers. The latter, however, is reasoning based on a formalism such as predicate calculus, and is well suited for automated development of programs. The main concept used in the studies of Bird and de Moor in solving problems is *catamorphism*. Catamorphisms in Bird and de Moor's theory is an operator similar to the fold operator in many functional languages. The results, mostly in a recursive format, are then translated into a functional programming style. The style of reasoning with functions and relations is pointfree.

A recent study by Lew and Mauch [27] represents dynamic programming problems in categories such as group routing problems, optimal binary tree problems, and some non-optimization problems. It is also a very good source of examples and applications of dynamic programming. A study by Ward [39] presents a unified model of algorithm design for design techniques such as dynamic programming.

Greedy algorithms are the other design approach studied in this thesis. Such algorithms provide a considerable benefit of simplicity and efficiency, if applicable to a problem. Greedy Algorithms are used for solving optimization problems, sequentially making locally optimal choices to make a globally optimal solution. There have been many studies for conditions under which greedy algorithms can be applicable to

problems, which are not considered in this thesis and can be studied in the following suggested sources. For a greedy algorithm to be applicable to a problem, the greedy structure should be satisfied [6]. *Matroids* and *greedoids* are structures that meet certain conditions guaranteeing a greedy solution. Although, greedy algorithms can be applied to other problems not fitting in this categories. Matroids, which first appeared in the combinatorial optimization study by Edmonds [12], exhibit the property of optimal substructure in a problem. Optimal substructure indicates that to create an optimal solution to a problem instance, optimal solutions to the subinstances are used [6]. There is also the concept of *graphic matroids*, which, for instance, covers the structure of the minimum spanning tree problem. Greedoid theory is studied by Korte and Lovasz in [24, 25, 26]. There are other studies on optimization problems and particularly greedy algorithms known as *priority algorithms* [5]. In the study of priority algorithms [5], greedy algorithms are known to satisfy the property of *incremental priority* (fixed priority). Therefore, problem subinstances are evaluated based on their priority, high to low, and removed from the input list. The priority is defined with respect to the *objective* (cost) function. This algorithm framework can be viewed as a generalization of *online algorithms* in [34, 23]. In addition, the authors of [5] derive lower bounds with respect to priority algorithms.

Sharon Curtis has studied optimization problems and contributed to the study of dynamic programming and greedy algorithms by approaching them in a *relational* context [9, 10, 8, 7]. This study introduces more relationship between dynamic programming and greedy algorithm and approaches them as combinatorial optimization problems. Curtis claims using a loop operator in *imperative programming* style gives an extra degree of freedom in generating feasible solutions.

Formal proof and *algorithm correctness* have been studied by J. McCarthy [28], C. A. R. Hoare [20, 22], R. Floyd [14], and E. Dijkstra [11]. There are also recent studies done by D. Smith [32, 35, 36, 38, 37]. Smith, in Kestrel Interactive Development System (KIDS), has approached correct and efficient algorithm development in two parts, techniques of automation process and the concept theory of algorithm design. Smith explains his studies are not about the effectiveness of tactics such as dynamic programming, but rather about how to apply the tactics to several type of problems. To derive algorithms, first the problem is formally described with *specifications*, then the theory is applied to derive the algorithm. Following datatype refinements, a correct and efficient program also known as concrete specification is developed from an abstract specification. The work is mainly on software development by refinement techniques and mechanizing the development of software. All of the KIDS transformations preserve correctness and are automatic, with the exception of choosing the algorithm design technique.

The main contribution of this thesis is deriving abstract dynamic programming and greedy algorithms and the transformation process to derive concrete algorithms from concrete problem specifications. The mathematical framework used to provide these methods is Predicative Programming. Predicative Programming is presented by E. Hehner in [18, 16, 17]. Pointwise reasoning is applied in defining functions. The proposed methods are suggested for the SIMPLE environment developed by T. Norvell [31, 29]. SIMPLE is an integrated development environment to develop and edit proofs and program development proofs. This environment assists its user in checking the steps required for proof and algorithm development. SIMPLE can extend to an environment to include proof support for programming and specification languages.

This thesis tries to provide more applicable methods for computer scientists resolving some inadequacies of the other introduced methods.

The rest of this thesis is structured as follows. Chapter 2 presents *category theory* as the mathematical framework of theories and algorithms studied in the literature and related work. Related work includes definitions and theories of dynamic programming by Bird and de Moor, and an application of their theory, optimal bracketing. It also includes the idea of a simple recursive loop by Curtis to generate *feasible* solutions, and an application of greedy algorithm which is Kruskal's minimum spanning tree. In addition, it presents Smith's approach on algorithm development in KIDS project including algorithm development steps, basic concepts, and software development by refinement. This chapter ends with an algorithm of job scheduling process presented to describe refinement and development concepts on algorithm development.

Chapter 3 presents the proposed generic dynamic programming algorithm. Like the divide-and-conquer method, it works by finding solutions to subinstances and combining solutions to the subinstances. Unlike divide-and-conquer, dynamic programming saves the solutions to the subinstances. There are two approaches to implementing dynamic programming: top-down and bottom-up. In this chapter we will discuss an approach to solving problems based on concretization of top-down and bottom-up abstract dynamic programming algorithms. Along the way, we also formalize the closely related divide-and-conquer approach. Matrix multiplication, otherwise known as optimal bracketing, and maximum black square on an image are the applications of dynamic programming considered in this chapter.

Chapter 4 presents the proposed greedy algorithm. In this chapter we are investigating how abstract specifications can be proved to be implemented by abstract

greedy algorithms. In this chapter, a formal structure for greedy algorithms in a predicative style is presented. Considering there can be many possible greedy choices in each step which all lead to a *completed* solution, the proposed algorithm makes an arbitrary selection amongst all possible greedy choices. Applications of greedy algorithms presented in this chapter are Kruskal's algorithm, and Huffman codes.

Finally, chapter 5 presents the summary and conclusion of this thesis and introduces possible future works. All chapters on this thesis can be studied independently.

Chapter 2

Background and Related Work

Studies are conducted by several researchers on algorithm design techniques such as dynamic programming, greedy algorithms, exhaustive search and divide-and-conquer in an algebraic form of programming. The algebraic approach can be applied to derive individual programs and is also a tool for studying general principles of programming, in particular those concerning optimization problems.

In this chapter we present category theory as the mathematical framework of theories and algorithms studied in the literature and related work. Related work section includes definitions and theories of dynamic programming by Bird and de Moor, and an application of their theory, optimal bracketing. It also includes the idea of a simple recursive loop by Curtis to generate feasible solutions, and an application of greedy algorithm which is Kruskal's minimum spanning tree. In addition, it presents Smith's approach on algorithm development in KIDS project including algorithm development steps, basic concepts, and software development by refinement. This chapter ends with an algorithm of job scheduling process presented to describe refinement

and development concepts on algorithm development.

2.1 Mathematical and Programming Frameworks

2.1.1 Functional Programming

Some algorithms which are studied in the related work are written in a functional programming style. Therefore, this section is an introduction to functional programming elements and features.

2.1.1.1 Datatypes and Functions

An example of a simple datatype is *boolean* defined as follows [2]:

$$Bool ::= false \mid true$$

which may also be used to define a new datatype such as:

$$Either ::= bool\ Bool \mid char\ Char.$$

Functions can be written in either of the following styles

$$and : (Bool \times Bool) \rightarrow Bool$$

$$and\,(false, b) = false$$

$$and\,(true, b) = b$$

$$cand : Bool \rightarrow (Bool \rightarrow Bool)$$

$$cand\ false\ b = false$$

$$cand\ true\ b = b$$

In the above definitions of a function, *cand* is called a curried function and *and* is called a non-curried function.

2.1.1.2 Recursive Definitions

Datatypes and functions may also be defined recursively [2]. For instance, natural numbers can be defined as:

$$Nat \ ::= \ zero \mid succ\, Nat$$

and *plus* function can be defined as:

$$plus\,(m,\ zero) \ = \ m$$

$$plus\,(m,\ succ\, n) \ = \ succ\,(plus\,(m, n))$$

Some function definitions can be written using numbers and some arithmetic operations. Factorial function *fact* that uses $n + 1$ instead of *succ* is an example of this case

$$fact\, 0 \ = \ 1$$

$$fact\,(n + 1) \ = \ (n + 1) \times fact\, n.$$

2.1.1.3 Lists

A *list* is a datatype commonly used in functional programming with two basic definitions [2]:

$$listr\ A \ ::= \ nil \mid cons\,(A, listr\ A) \tag{2.1}$$

$$listl\ A \ ::= \ nil \mid snoc\,(list\ A,\ A) \tag{2.2}$$

Cons-list defined in (2.1) builds a list in which new elements are added to the front; snoc-list defined in (2.2) builds a list in which new elements are added to the rear.

An example of functions on lists is *map*, which applies a function to every member of a list

$$f \ : \ B \rightarrow C$$

$$map\, f \ : \ list\, B \rightarrow list\, C$$

$$map\, f\, [a_1, a_2, \ldots, a_n] = [f a_1, f a_2, \ldots, f a_n]\,.$$

2.1.1.4 Trees

One form of a *tree* can be described as a datatype whose values are either tips containing data or pairs of trees [2]:

$$tree\, A \ ::= \ tip\, A \,|\, bin\, (tree\, A,\ tree\, A).$$

An example of this definition is *tree Char* where its elements are characters $'A', 'B'$, and $'C'$:

$$bin\, (tip\, 'A',\ bin\, (tip\, 'B',\ tip\, 'C')).$$

2.1.2 Category Theory

Category theory is an algebraic structure that is useful for developing relations between specifications, designs, correctness proofs, and programming languages. It provides a framework for correctness and data validity of programming and algorithm development [21]. Some examples of categories include Set –the category of all sets– and Rel –the category of all relations. Set is first described as an example of categorical concepts. There is also a short description of Rel that is introduced by Bird and

de Moor [2]. The elements of category theory presented in this chapter are necessary to understand the related work on algorithm development. To study these concepts in more details, an example is provided in (2.1.3).

2.1.2.1 Categories

By definition a *category*, denoted by C, is a collection of *objects* and *arrows* (*morphisms*) where each arrow $f : B \to A$ has a source and a target object [33, 2]. On each category, *composition arrows* and *identity arrows* are defined. For any two arrows $f : B \to A$ and $g : C \to B$, the composition arrow is

$$f \cdot g : C \to A.$$

For arrows $f : B \to A$, $g : C \to B$, and $h : D \to C$ the following associative law is defined:

$$f \cdot (g \cdot h) = (f \cdot g) \cdot h.$$

For each object A and arrow $f : B \to A$, an identity arrow $id_A : A \to A$ satisfies the following identity laws:

$$f \cdot id_B = f, \text{ and}$$

$$id_A \cdot f = f.$$

For any category C, the opposite category C^{op} is defined such that it has the same objects but reversed arrows, by inversion of source and target.

2.1.2.2 Functors

A *Functor* is a category of categories, its objects are categories and its arrows are maps between categories denoted by F [33, 2]. A mapping of a category to category

includes mapping of two elements of a category: a mapping of objects to objects and a mapping of arrows to arrows. Let C and D be categories, a functor $\mathsf{F} : D \to C$ maps each object of D such as B to an object of C such as $\mathsf{F}(B)$, and each arrow of D such as $f : B \to A$ to an arrow of C such as $\mathsf{F}(f) : \mathsf{F}(B) \to \mathsf{F}(A)$ where

$$\mathsf{F}(id_A) = id_{\mathsf{F}(A)}, \text{ and}$$
$$\mathsf{F}(f \cdot g) = \mathsf{F}f \cdot \mathsf{F}g.$$

2.1.2.3 Products

A *product* of two objects A and B in a category C is an object denoted by $A \times B$ together with two arrows $outl : A \times B \to A$ and $outr : A \times B \to B$ with the following property [33, 2]: for any object C and arrows

$$f : C \to A, \text{ and}$$
$$g : C \to B,$$

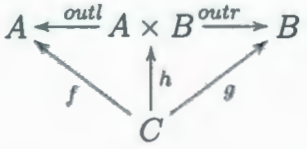
there exists a unique arrow

$$h : C \to A \times B$$

such that

$$h = \langle f, g \rangle \quad \equiv \quad outl \cdot h = f \text{ and } outr \cdot h = g$$

where $\langle f, g \rangle$ is pronounced "pair f and g". The following diagram summarizes the type information



2.1.2.4 Coproducts

A *coproduct* of two objects A and B in a category C is an object denoted by $A + B$ together with two arrows $inl : A \to A + B$ and $inr : B \to A + B$ with the following property [33, 2]: for any object C and arrows

$$f : A \to C, \text{ and}$$

$$g : B \to C,$$

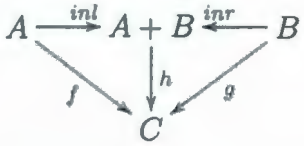
there exists a unique arrow

$$h : A + B \to C$$

such that

$$h = [f, g] \quad \equiv \quad h \cdot inl = f \text{ and } h \cdot inr = g$$

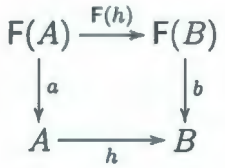
where $[f, g]$ is pronounced "case f or g". The coproduct of A and B in C is the product of A and B in C^{op}. The following diagram summarizes the type information



2.1.2.5 F-Algebras and F-Homomorphisms

Let K be a category and $\mathsf{F} : \mathsf{K} \to \mathsf{K}$ a functor, an F-*algebra* is a pair (A, α) of an object A of K and an arrow $\alpha : \mathsf{F}(A) \to A$ of K [33].

An F-*homomorphism* from an algebra (A, a) to an algebra (B, b) is an arrow $h : A \to B$ of K such that the following diagram commutes [33, 2]:



A simple example is the algebra $(Nat, +)$ of the natural numbers and addition which is an algebra of the functor:

$$\begin{cases} \mathsf{F}(A) = A \times A \\ \mathsf{F}(h) = h \times h \end{cases}$$

2.1.2.6 Initial Algebras and Catamorphisms

An F-algebra (T, α) is an initial F-algebra if and only if for every F-algebra (A, f), there is a unique homomorphism $h : T \to A$ which is called *catamorphism* $(\!|f|\!)$[33, 2]. The following diagram summarizes the type information

$$\begin{array}{ccc} \mathsf{F}(T) & \xrightarrow{\mathsf{F}(|\!|f|\!|)} & \mathsf{F}(A) \\ {\scriptstyle \alpha}\downarrow & & \downarrow{\scriptstyle f} \\ T & \xrightarrow[|\!|f|\!|]{} & A \end{array}$$

For all arrows $h : T \to A$ there is a universal property

$$h = (\!|f|\!) \equiv h \cdot \alpha = f \cdot \mathsf{F}(h)$$

Example 1. Catamorphisms on Strings

A *string* is a list of chars which in functional programming can be presented as a datatype defined as follows [2]

$$String ::= nil \,|\, cons(Char, String).$$

To create a catamorphism, the definition of string declares that the initial algebra is built as $[nil, cons] : \mathsf{F}\, String \to String$ of the functor F:

$$\mathsf{F}\, A = 1 + (Char \times A)$$

$$\mathsf{F}\, f = id + (id \times f)$$

Here $nil : 1 \to String$ is a constant function. Every algebra of the functor on string is $[c, f]$ where $c : 1 \to A$ is a constant and $f : Char \times A \to A$ is a function.

In order to form $h \cdot \alpha = f \cdot \mathsf{F}(h)$ to create a catamorphism, h is defined as $(\!|c, f|\!)$.

2.1.2.7 Hylomorphism

The composition of a catamorphism with the converse of a catamorphism is called a *hylomorphism*, such as $(\!|R|\!).(\!|S|\!)^\circ$ [2]. Let $R : \mathsf{F}\,A \to A$, $S : \mathsf{F}\,B \to B$, $(\!|R|\!) : T \to A$, and $(\!|S|\!) : B \to T$, where T is the initial type of F, then $(\!|R|\!) \cdot (\!|S|\!)^\circ : B \to A$. Hylomorphisms can be characterized as least fixed points.

Theorem 2. *Suppose that* $R : \mathsf{F}A \to A$ *and* $S : \mathsf{F}B \to B$ *are two* F-algebras, *then* $(\!|R|\!) \cdot (\!|S|\!)^\circ : B \to A$ *is given by*

$$(\!|R|\!) \cdot (\!|S|\!)^\circ = (\mu X \ : \ R \cdot \mathsf{F}\,X \cdot S^\circ)$$

2.1.3 Set Theory and Category Set

This section is a brief survey of set theory. It shows that sets and functions between sets have the structure of categories.

2.1.3.1 Sets

One view of sets is that a *set* is a collection of elements that share a common property such as P [21], and is defined as follows

$$S = \{x \mid Px\}\,.$$

Each element in a set is called a *member*, $x \in S$. An element outside the set is not a member $n \notin S$.

$$x \in S,\ y \in S, \ldots,\ z \in S \text{ is also written as } x, y, \ldots, z \in S$$

A *subset* A of a set B is a set such that all of its members are also members of set B, formally defined as (2.3). In addition, two sets are equal if the subset relation is mutually definable, formally defined as (2.4).

$$A \subseteq B \equiv (x \in A \text{ implies } x \in B, \text{ for all } x) \tag{2.3}$$

$$A = B \equiv (A \subseteq B \text{ and } B \subseteq A) \tag{2.4}$$

Some examples of sets are the *empty set*, and *powerset*. The empty set $\{\}$ is a set with no members. Sets can have none, one, or more members. Powerset is a set containing all subsets of a set, including an empty set and the set itself.

If $x, y \in S$, *pair* (x, y) can be defined as an ordered pair. Two pairs are equal if their ordered members are equal.

$$(x, y) = (x', y') \equiv x = x' \text{ and } y = y'$$

The cartesian product of $A \times B$ contains all pairs (x, y) as defined

$$A \times B = \{(x, y) \mid x \in A \text{ and } y \in B\}.$$

Operations such as $\cup, \cap, \bigcup, \bigcap$, and $\neg$ are defined on sets. For instance, $A \cup B$ is the union of A and B defined as follows:

$$A \cup B = \{x \mid x \in A \text{ or } x \in B\}.$$

If S is a set of sets, its union $\bigcup S$ is the set containing all members of any of its members defined as follows:

$$\bigcup S = \{x \mid (\exists y \in S \cdot x \in y)\}$$

2.1.3.2 Functions on Sets

A *function* consists of two sets, the source and the target, and a mapping between them [21]. In concept, a function is very close to functor; it is written in form of $(f : S \to T)$, where

- S is a source set,
- T is a target set,
- f is a total mapping which maps members of S to members of T.

Two mappings could be equal, however, form different functions by having different source and target sets. Thus,

$$(f : S \to T) = (f' : S' \to T') \text{ iff } \begin{cases} S = S', \\ T = T', \text{ and} \\ fx = f'x, \text{ for all } x \in S \end{cases}$$

For simplicity, function $f : S \to T$ is abbreviated to f and a mapping is denoted using λ notation. If x is a variable in S and $\cdots x \cdots$ is an expression in T when x, then $(\lambda x \cdots x \cdots)$ is a λ expression. The corresponding function $((\lambda x \cdots x \cdots) : S \to T)$ includes the presented mapping, source set S, and target sets T. The composition of functions $f : S \to T$ and $g : T \to U$ is also a function

$$f; g = ((\lambda x \cdot g(fx)) : S \to U).$$

2.1.3.3 Isomorphisms

Let f and g be functions defined as $f : S \to T$ and $g : T \to S$, if there is a unique g for f where they have the following relation

$$f; g = I_s$$

then g is called the *inverse* of f. A function that has an inverse is called an *isomorphism*; S and T are said to be *isomorphic* ($S \simeq T$) [21]. In category Set isomorphism is also referred to as *bijection* which has the property of *surjection* and *injection*. If the image of a function $f : S \to T$, which is a subset of T which can be obtained by applying f to some members of S, contains all members of the target set T, then f is said to be a surjection. Conversely, for any subset S of T, there is a functor that maps each member of S to itself in T which is said to be the injection.

2.1.3.4 Sets and Categories

There are two type of categories: *concrete category* and *abstract category* [21]. F is called a concrete category if it is a family of functions that has the two following properties:

- if $(f : s \to t) \in \mathsf{F}$ then $I_s, I_t \in \mathsf{F}$
- if $(f : s \to t), (g : t \to u) \in \mathsf{F}$ then $((f; g) : s \to u) \in \mathsf{F}$.

The objects of the category defined as a family of sets denoted by $\| \mathsf{F} \|$. The arrows of the category are the functions denoted by $| \mathsf{F} |$.

Set is the category that has all sets as its objects and all functions as its arrows. Set is an example of a concrete category; all concrete categories are subcategories of

Set.

Formally, an abstract category C is defined as ($\| \mathsf{C} \|$, $| \mathsf{C} |$, I, ;, $\leftarrow$, $\rightarrow$) where [21]

- $\| \mathsf{C} \|$ is a family of objects
- $| \mathsf{C} |$ is a family of arrows
- I identity is a function from $\| \mathsf{C} \|$ to $| \mathsf{C} |$
- ; composition is a partial binary operator on $| \mathsf{C} |$
- $\leftarrow$ source and $\rightarrow$ target are total functions from $| \mathsf{C} |$ to $\| \mathsf{C} \|$

Moreover, the following properties must be satisfied in a category [21]:

- $\overrightarrow{I_s} = s = \overleftarrow{I_s}$,
- $f;g$ is defined just when $\overrightarrow{f} = \overleftarrow{g}$,
- $\overleftarrow{f;g} = \overleftarrow{f}$, and $\overrightarrow{f;g} = \overrightarrow{f}$,
- $f;I_{\overrightarrow{f}} = f = I_{\overleftarrow{f}};f$,
- $(f;g);h = f;(g;h)$

2.1.4 Relations and Category Rel

2.1.4.1 Rel

Another example of a category is Rel which is defined on sets and relations [2]. In Rel objects are sets and each arrow is a relation $R : A \rightarrow B$ which is a subset of the

Cartesian product $A \times B$. In relations $a\,R\,b$ indicates $(a, b) \in R$. The identity arrow is:

$$id_A = \{(a, a) \mid a \in A\}.$$

Composition arrow of $R : B \to A$ and $S : C \to B$ is defined as:

$$a\,T\,c \equiv (\exists b \cdot a\,R\,b \wedge b\,S\,c)$$

2.1.4.2 Powerset Functor

A functor $\mathsf{P} : \mathsf{Fun} \to \mathsf{Fun}$ – where Fun is the category of sets and total functions – is a mapping from a set A to the *powerset* $\mathsf{P}A$ defined as follows [2]

$$\mathsf{P}\,A = \{x \mid x \subseteq A\}.$$

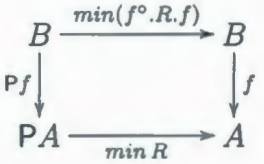
It also applies f to a set which then applies to every element of it, by mapping a function f to the function $\mathsf{P}f$.

2.1.4.3 Min and Max

For any relation $R : A \to A$, the relation $min\,R : \mathsf{P}A \to A$ relates x to a if a is an element of x and a lower bound of x with respect to the relation R [2]. For all $X : A \to A$ we can also define $max\,R = min\,R^\circ$, so a maximum element with respect to R is a minimum element with respect to R°. The minimum element applies to a function f as follows

$$min\,R \cdot \mathsf{P}f = f \cdot min(f^\circ \cdot R \cdot f)$$

The following diagram summarizes the type information



2.1.4.4 Monotonic Algebras

By definition, an F-algebra $S : \mathsf{F}(A) \to A$ is *monotonic* on a relation $R : A \to A$ if [2]

$$S \cdot \mathsf{F}(R) \subseteq R \cdot S.$$

The following diagram summarizes the type information

Furthermore, if S is monotonic on a preorder R°, then

$$(\!|min\, R \cdot \Lambda S|\!) \subseteq min\, R \cdot \Lambda((\!|S|\!)).$$

To illustrate, consider the following example when F-algebra S is addition of natural numbers, $plus : Nat \times Nat \to Nat$, S is monotonic on relation R "Less than or Equal", leq, if

$$plus \cdot (leq \times leq) \subseteq leq \cdot plus.$$

To show the correctness we have:

$$c = a + b \; and \; a \leq a' \; and \; b \leq b' \;\; \Rightarrow \;\; c \leq a' + b'$$

2.1.4.5 Power Transpose (Lambda)

A *power transpose* also known as *lambda* operator Λ is an operation on relations which converts a relation to the corresponding function [9]

$$(\Lambda R)x = \{y \mid y\, R\, x\}.$$

2.1.4.6 Coreflexives

A *domain* of a relation is a relation defined as [9]

$$dom\, R = \{(y, y) \mid \exists x \cdot (x, y) \in R\}.$$

A *notdomain* of a relation is a relation defined as

$$notdom\, R = \{(y, y) \mid \neg\exists x \cdot (x, y) \in R\}.$$

As shown in the above definitions, in the studies by Curtis sets of pairs are reversed from their usual order.

2.1.5 Algorithm Development and Category Specs

2.1.5.1 Specification

A *Specification* is a presentation of a theory describing objects, operations, and properties and axioms that constrain the meaning of the symbols [35, 32].

Example 3. Specification of Partial Order

Specification *PreOrder* is

sort E

op *-le-:E,E* $\rightarrow$ *Boolean*

axiom *reflexivity* is $x\ le\ x$

axiom *transitivity* is $x\ le\ y\ \wedge\ y\ le\ z \Longrightarrow x\ le\ z$

end-spec

2.1.5.2 Morphisms

A *morphism* is a translator of the language of one specification into language of another specification preserving the correctness of the theorems of the source specification in the destination specification; a morphism translates theorems and interprets symbols of a theorem to expressions [35, 32].

Example 4. *Partial-Order* specification using *PreOrder* specification

spec *Partial-Order*

import *PreOrder*

axiom *antisymmetry* is $x\ le\ y\ \wedge\ y\ le\ x \Longrightarrow x = y$

end-spec

Example 5. A specification morphism from Partial-Order to Integer

morphism *Partial-Order-to-Integer* is:

$\{E \rightarrow$ Integer, $le \rightarrow \leq\}$

An *import (extension) morphism* is used to create a new theorem using existing theorems. For example, specification of preorder which has the axioms of reflexivity and transitivity could be imported to build up a new specification Partial-Order which has those axioms and an additional axiom of antisymmetry.

2.1.5.3 Category Specs

Category of *Specs* is a category which has specifications as its objects and specification morphism as its arrows [35, 32]. The definition of *colimit*(coproduct) is also defined in this category. For example, $A \xrightarrow{j} C$ and $A \xrightarrow{i} B$ has a colimit $B \xleftarrow{i} A \xrightarrow{j} C$ is computed as follows:

- form the disjoint union of all sort and operator symbols of A, B, and C
- define $s \approx t$ iff $(i(s) = t \vee i(t) = s \vee j(s) = t \vee j(t) = s)$ where $i : A \to B$ and $j : A \to C$
- use morphism to define the axioms of colimit from axioms of A, B, and C

The colimit is the collection of all the equivalent relations ($\approx$) and sort, symbols, and axioms.

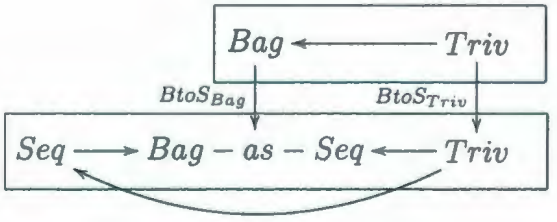
Example 6. The theory of Binary Relation presented as a colimit of *Antisymmetry* and *PreOrder*

In this example morphisms are $\{E \rightarrow E,\ le \rightarrow le\}$ from *BinaryRelation* to *Antisymmetry* and *Preorder*. Colimits can be used to construct a large specification from a diagram of specs and morphisms.

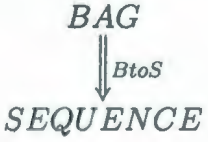
2.1.5.4 Refinement and Diagrams

Specification morphisms can be used to structure and refine a specification [32, 35]. A morphism between a specification domain and codomain is a refinement which reduces the number of possible implementations. Thus, refinement represents a particular design decision or property that corresponds to the implementation of a domain specification and is also applicable to codomain specification. A *diagram morphism* from diagram D to diagram E is a set of specification morphisms, from a specification in D to a specification in E, which maps diagram D to diagram E, represented as $D \Longrightarrow E$. The following diagram summarizes the refinement structure:

Example 7. A diagram morphism as data type refinement that maps bags to sequences



is represented by diagram morphism as:



2.1.5.5 Global Search

The definition of global search by Smith [38] is as follows:

> The basic idea of global search is to represent and manipulate sets of candidate solutions. The principal operations are to *extract* solutions from a set and to *split* a set into subsets. Derived operations include various *filters* which are used to eliminate sets containing no feasible or optimal solutions... Thus global search algorithms are based on an abstract data type of informational representations called *space descriptors*. In addition to the extraction and splitting operations mentioned above, the type also includes a predicate *satisfies* that determines when a candidate solution is in the set denoted by a descriptor.

2.2 Related Work in Dynamic Programming

Bird and de Moor's theories [2, 4, 3] make use of categorical calculus of relations as a mathematical framework, providing the possibility of abstracting away the datatypes and enhancing the capabilities of functional calculus. This framework helps in formulating theorems to calculate programs in form of an abstract solution. In addition, it is a framework to conduct proofs . Theorems are based on structural similarities of the specification of the problems. The main concept used in solving problems is catamorphism. The results, mostly in a recursive format, are then translated into a functional programming language, using fold and unfold operations. The style of reasoning with functions and relations is pointfree.

In order to solve optimization problems, Bird and de Moor formulated a problem M in the following equation [2]

$$M = min\, R \cdot \Lambda\left(\llparenthesis h \rrparenthesis \cdot \llparenthesis T \rrparenthesis^{\circ}\right) \tag{2.5}$$

where h is a function. An algorithm to solve the problem M is presented as

$$(\mu X \ : \ min\, R \cdot \mathsf{P}\,(h \cdot \mathsf{F}X) \cdot \Lambda T^{\circ}) \subseteq M \tag{2.6}$$

Using dynamic programming for this algorithm optimal solutions to subinstances of problems are composed to make an optimal solution. This method is presented in two theorems of dynamic programming from [2]. In theorem 8 below, the problem instance is decomposed in all possible ways into subinstances. Then, recursively solved subinstances are composed to make an optimal solution. In theorem 9 below, the decompositions that will not lead to an optimal solution will be removed from the feasible solutions.

Theorem 8. (Bird and de Moor) *Let* $M = min\, R \cdot \Lambda\,(\!|h|\!) \cdot (\!|T|\!)^{\circ})$. *If* h *is monotonic on* R, *then*

$$(\mu X \ : \ min\, R \cdot \mathsf{P}\,(h \cdot \mathsf{F}X) \cdot \Lambda T^{\circ}) \subseteq M.$$

This theorem has a recursive scheme in which the input is decomposed in all possible ways. There are some problems that clearly declare that some decompositions will not lead us to a result better than others. Thus, this idea is added to the theorem 8 to eliminate unprofitable decompositions. The result is the theorem 9.

Theorem 9. (Bird and de Moor) *Let* $M = min R \cdot \Lambda\,(\!|h|\!) \cdot (\!|T|\!)^{\circ})$. *If* h *is monotonic on* R *and* Q *is a preorder satisfying* $h \cdot \mathsf{F}\,(\!|h|\!) \cdot (\!|T|\!)^{\circ}) \cdot Q^{\circ} \subseteq R^{\circ} \cdot h \cdot \mathsf{F}\,(\!|h|\!) \cdot (\!|T|\!)^{\circ})$ *then*

$$(\mu X \ : \ min\, R \cdot \mathsf{P}\,(h \cdot \mathsf{F}\,X) \cdot thin\, Q \cdot \Lambda T^{\circ}) \subseteq M.$$

Hylomorphisms are important because they capture the idea of using an intermediate data structure in a solution of a problem. As mentioned in theorem 2,

hylomorphisms can be characterized as least fixed points. In both theories, optimal solutions can be computed as a least fixed point. If ΛT° returns finite non-empty sets and R is a connected preorder (Relation R is connected if $R \cdot R^{\circ} = \prod$) then the unique solution is entire.

Example 10. Optimal Bracketing

Let's study the theorem by a standard application of dynamic programming that is the problem of building a minimum cost binary tree [2]. The problem is often formulated as one of bracketing an expression $a_1 \oplus a_2 \oplus \cdots \oplus a_n$ in the best possible way. It is assumed that $\oplus$ is an associative operation, so the way in which the expression is bracketed does not affect its value. However, different bracketings may have different costs, and the objective is to find a bracketing of minimum cost. The cost of each tree is compared by relation R.

A binary tree represents a datatype for bracketing with values in the tips:

$$tree\ A ::= tip\ A \mid bin(tree\ A, tree\ A).$$

For example, the bracketing $(a_1 \oplus a_2) \oplus (a_3 \oplus a_4)$ is represented by the tree:

$$bin(bin(tip\ a_1, tip\ a_2), bin(tip\ a_3, tip\ a_4)).$$

The general method (2.5) is used to present optimization problems

$$min\ R.\Lambda\left((\!|h|\!) \cdot (\!|T|\!)^{\circ}\right).$$

Introducing function $flatten : tree\ A \rightarrow list^{+} A$ that flattens a tree to a list:

$$flatten = (\!|wrap, cat|\!)$$

where *wrap* turns its argument into a singleton list, $cat : (list^{+}A)^2 \rightarrow list^{+}A$ takes two non-empty lists of tip values to a non-empty list from left to right, and the following substitutions:

$$\mathsf{F}\, f = id + f \times f$$

$$R = R$$

$$h := [tip, bin]$$

$$(\!|h|\!) = id$$

$$T = [wrap, cat]$$

$$\therefore (\!|T|\!) = flatten.$$

$\Lambda flatten^{\circ}$ creates all possible trees. This optimization problem is represented as:

$$min\, R \cdot \Lambda(\!|wrap, cat|\!)^{\circ}.$$

Thus, the problem to find the minimum cost tree is formulated as follows:

$$mct \subseteq min\, R \cdot \Lambda(\!|wrap,\ cat|\!)^{\circ}.$$

In words, a list of expressions forms a tree that is being made in all possible ways, making a set of results. Then according to the cost of building a tree determined by relation R, $min\, R$ chooses the minimum cost trees. When

$$R = cost^{\circ} \cdot leq \cdot cost,$$

$$t_0\, R\, t_1 \equiv cost\, t_0 \leq cost\, t_1.$$

The cost of building a single tip is zero, while the cost of building a node is some function of sizes of the expressions associated with the two subtrees, plus the cost of building the two subtrees; furthermore, $\langle cost, size \rangle$ form a catamorphism.

Since the condition of monotonicity applies

$$cost \cdot [tip, bin] = g \cdot (id + \langle cost, flatten \rangle^2)$$

$$g \cdot (id + (leq \times id)^2) \subseteq leq \cdot g,$$

where

$$g = [zero, outl \cdot opb \cdot (id \times sz)^2],$$

the dynamic programming theorem 8 is, therefore, applicable. Thus, we can compute a minimum cost tree using the least fixed point of the recursion equation.

$$\mu X : min\, R \cdot [tip, bin \cdot (X \times X)] \cdot \Lambda\, [wrap, cat]^\circ \subseteq mct$$

$$X = (single \rightarrow tip \cdot wrap^\circ, min\, R \cdot \mathsf{P}(bin \cdot (X \times X) \cdot \Lambda cat^\circ)$$

where

$$(p \rightarrow f, g)\, a = \begin{cases} f\, a, & \text{if } p\, a \\ g\, a, & \text{otherwise} \end{cases}$$

and Λcat° can be represented as a function

$$split = zip \cdot \langle inits^+, tails^+ \rangle,$$

where *zip* is a function that transforms pairs of lists to lists of pairs, $inits^+$ and $tails^+$ return the list of proper initial and tail segments of a list.

To make it easier to follow consider this explanation:

$$X\, ([a]) = tip\, a$$

Let's have s as every way that we can split $l = \Lambda cat^{\circ}$:

$$
\begin{aligned}
X\,(l) = & \\
& \text{let } s \;=\; \{(l_0, l_1) \mid l = cat\,(l_0, l_1)\} \text{ in} \\
& \text{let } t = \{(l_0, l_1) \in s \cdot bin\,(Xl_0)\,, (Xl_1)\} \text{ in} \\
& \text{let } u = min\; t \text{ in} \\
& u.
\end{aligned}
$$

In words, produce all the possible pairs, get each pair, form a tree, and apply X to it, recurs on both of them and use previously calculated optimal trees, then make the tree and finally pick the best one.

To compute the optimum bracketing for the main problem, we need to compute the smaller recursively defined subinstances of the problem instance. We will represent an array as a list of rows, however we also consider columns.

$$
\begin{aligned}
array &= list\; row \cdot inits \\
row &= list\; mct \cdot tails \\
col &= list\; mct \cdot inits
\end{aligned}
$$

inits returns the list of non-empty initial segment in increasing order of length, and *tail* the tail segments in decreasing order. First step is to express *mct* in terms of *row* and *col*.

$$
\begin{aligned}
mct \;&= min\, R \cdot list\,(bin \cdot (mct \times mct)) \cdot zip \cdot \langle inits^{+}, tails^{+} \rangle \\
&= mix \cdot \langle col \cdot init, row \cdot tail \rangle
\end{aligned}
$$

where $zip \cdot \langle inits^{+}, tails^{+} \rangle$ creates all possible break points, and $mix = min\, list\, R \cdot list\, bin \cdot zip$ zip the pairs, change it to tree, and picks the best. Next, *col* is represented

in terms of row and col.

$$col = next \cdot \langle col \cdot init, row \cdot tail \rangle.$$

Therefore,

$$col = (single \rightarrow wrap \cdot tip \cdot head \cdot, next \cdot \langle col \cdot init, row \cdot tail \rangle)$$

$$next = snoc \cdot \langle outl, mix \rangle.$$

Finally, row is expressed in terms of mct and col,

$$row = cons \cdot \langle mct, row \cdot tail \rangle.$$

Therefore,

$$row = (single \rightarrow wrap \cdot tip \cdot head, cons \cdot \langle mct, row \cdot tail \rangle).$$

Array is computed as a catamorphism on cons-lists, building columns from right to left and then using the column entries to extend each row. Therefore, we have array as

$$array = (\!|fstcol, addcol|\!),$$

where

$$fstcol = wrap \cdot wrap \cdot tip,$$

$$addcol = cons \cdot \langle wrap \cdot tip \cdot outl, step \rangle,$$

$$step = list\ cons \cdot zip \cdot \langle tail \cdot process, outr \rangle.$$

The program to calculate optimal bracketing as a result of the calculations is as follows:

```
> data Tree a = Tip a | Bin(Int, a) (Tree a, Tree a)
> mct = head · last · array
> array = cata1list(fstcol, addcol)
> fstcol = wrap · wrap · tip
> addcol = cons · pair · (wrap · tip · outl, step)
> step = list cons · zip · pair(tail · process, outr)
> process = loop next · cross(wrap · tip, id)
> next = snoc · pair(outl, minlist r · list bin · zip)
>         where r = leq · cross(cost, cost)
> cost(Tip a) = 0
> cost(Bin(c, s)ts) = c
> size(Tip a) = a
> size(Bin(c, c)ts) = s
> tip = Tip
> bin(x, y) = Bin(c, c)(x, y)
>                 where c = cb(size x, size y) + cost x + cost y
>                       s = sb(size x, size y)
```

The program produced by Bird and de Moor theorem is very concise and well developed. However, the process summarized here is about eight pages of a book. Furthermore, some of the conclusions made in one line, however, would be better written in a few lines of equations to provide more clarity for the reader.

2.3 Related Work in Greedy Algorithms

Sharon Curtis made a contribution to the study of dynamic programming and greedy algorithms by approaching them in a relational context [9]. This study introduces more relationships between dynamic programming and greedy algorithms and approaches them as combinatorial optimization problems. It also uses a simple loop operator to generate feasible solutions.

Sharon Curtis also claims that although the theories by Bird and de Moor are useful to express considerable number of problems, it is not easily applicable to all optimization problems and not applicable to some exceptional cases at all. She takes Huffman Coding as an example, and concludes that catamorphism and anamorphism methods are Top-Down methods while examples such as Huffman Coding need Bottom-Up methods.

2.3.1 General Specification

The general specification in a relational context is [9]

$$min\, R \cdot \Lambda Gen$$

where Gen is a relation that produces a feasible solution, and ΛGen produces the set of all feasible solutions to the problem. The loop operator lim expresses the specification as

$$min\, R \cdot \Lambda\, lim\, T$$

where T is a relation which refers to each step of constructing a feasible solution. In the algorithm $lim\, T$, the construction step is repeated in a loop and stops when it

can generate no more new steps.

The Greedy algorithms are presented in more detail in this section. A feasible solution that satisfies a global optimal condition is made in steps, where each step extends a partial solution. To make a decision in each step, a local optimality condition is used to compare created *partial* solutions. The Greedy algorithm $lim\, G$ performs a sequence of steps which takes a partial solution and results in a feasible solution that is globally optimal. This sequence stops when a new step does not produce a new extension. The greedy step G considers extending a partial solution in all possible ways when making the locally optimal choice

$$G = min\, S \cdot \Lambda T.$$

In each step G, T constructs a piece, and ΛT returns the set containing all extended partial solutions.

2.3.2 The Algorithm

The algorithm is in form of a relational model of a loop called *limit* operator [9]. Sharon Curtis proves in her theorems that limits are a generalization of catamorphism and anamorphisms. This section is about the theory of converting catamorphisms to the *lim* operator. Let

$$(\!|P|\!) : A \rightarrow B,$$

where A is the carrier set of the initial F-algebra α, and the problem under construction is

$$min\, R \cdot \Lambda(\!|P|\!).$$

On the other hand, in $lim\,T$, $T : C \to C$ for some type C. Thus, to get to C from A and to get to B from C two additional functions such as $start : A \to C$ and $finish : C \to B$ are required. Now, cata P is represented based on lim operator

$$(\!|P|\!) = finish \cdot lim\,T \cdot start.$$

For this purpose, some constraints on the functions *start* and *finish* are required, presented in a theorem.

Theorem 11. *If start is a function, finish is simple and also the converse of a function, and*

$$dom\,\text{finish} = notdom\,T$$

$$Q = \text{finish} \cdot lim\,T \cdot start$$

$$R' = \text{finish}^{\circ} \cdot R\,\text{finish},$$

then

$$min\,R \cdot \Lambda Q = \text{finish} \cdot min\,R' \cdot \Lambda lim\,T \cdot start.$$

This paragraph helps to determine the type of C. A new F'-algebra α' is created that is of the form of a join $\alpha' = [Pen, Fin]$ which relates to type B and type A, where Fin marks a finished portion that is computed and Pen for a pending portion. To convert the previously introduced F-structure to the recently introduced F'-algebra α', *start* is a function that changes the structures by making "Pending" labeling, then *finish* is a reverse of a function and removes the Fin label from a finished computation.

$$start = (\!|Pen|\!)_{\mathsf{F}}$$

$$finish = Fin^{\circ}.$$

The following definition presents P' that executes a P-step:

$$P' = [Fin \cdot P \cdot \mathsf{F}\, Fin^{\circ}, 0],$$

where 0 represents that it reaches an unfinished portion, it does not perform. $\mathsf{F}\, Fin^{\circ}$ check s that all the needed results so far have been finished, and removes the Fin labels on them. According to P', the construction step known as T is determined as

$$T = (\!|P' \cup \alpha'|\!)_{\mathsf{F}} \cdot notdom\, Fin^{\circ},$$

where, $(\!|P' \cup \alpha'|\!)_{\mathsf{F}'}$ presents that either step P occurs or nothing, and the rest of the expression represents that the loop terminates when the computation has finished.

Theorem 12. *Given the definitions,*

$$(\!|P|\!)_{\mathsf{F}} = \text{finish} \cdot lim\, T \cdot \text{start}.$$

2.3.3 The Greedy Algorithms

The Greedy Algorithms are an approach to solve optimization problems, making a sequence of locally optimal choices to create a globally optimal solution [9]. The Greedy algorithms, however, are not applicable to problems where the greedy structure is not satisfied. Let S be the local optimality criterion, and R be the global optimality criterion, the greedy algorithm is presented in the following theorem.

Let

$$M = min\, R \cdot \Lambda lim\, T \tag{2.7}$$

$$G = min\, S \cdot \Lambda T, \tag{2.8}$$

where R is a preorder on the set of completed solutions represented by $notdom\, T$. If the following conditions are satisfied:

$$dom\, G = dom\, T \tag{2.9}$$

$$G \cdot (lim\, T)^\circ \subseteq (lim\, T)^\circ \cdot R \tag{2.10}$$

then

$$lim\, G \subseteq M.$$

In each greedy step G, min selects the best with respect to S that is not necessarily the same relation as R. The algorithm is $lim\, G$ that is the repeat of G until it is finished. The first condition in the theorem ensures that G can be performed while we have some unfinished solution. The second condition ensures that G makes the correct choice. It means that continuing the algorithm will lead us to a completed result at least as good as R. Thus, the algorithm $lim\, G$ could be implemented as a simple loop with body f and guard $dom\, T$, where a variant checks the termination of the loop.

2.3.4 Application of Greedy Algorithms: Kruskal's method

Kruskal's algorithm is a well-known greedy algorithm that finds a spanning tree of minimum cost in an undirected (connected) graph $G = (V, E)$ with edge costs $w : E \to \mathbb{R}$ [10].

This algorithm builds a set of edges that do not create a cycle. Starting by an empty set and adding an edge with a minimum cost in each step, the set is formed. Each created set is a partial solution, and the edges sets forming spanning trees are completed solutions. G in this example adds an acyclic edge – an edge which does

not create a cycle – to the set, S is a relation of minimum weight edge, and R is a relation comparing the sums of weights of the sets of edges.

One of the conditions for the greedy algorithm to be applicable is (2.9) which is satisfied since a new edge can be easily added to the set of edges. The specification of the Kruskal's algorithm is, therefore, presented as the format of the problem specification (2.7)

$$min\, R \cdot \Lambda\, SpanningTree$$

where

$$R = \sum_{e \in t} edgecost\, e,$$
$$SpanningTree = lim\, S.$$

The local cost relation S on sets of edges is defined as:

$$es \cup \{e\}\ S\ es, \quad \text{if } e \in \{E - es\} \ \wedge\ acyclic(es \cup \{e\})$$

where es set is a subset of the set of edges and is acyclic.

To prove that the condition (2.10) is satisfied, providing the set of edges created in this way is of a minimum cost, two sets ds and ds' is created as

$$ds = \{e_1 \ldots e_n\} - \{e'_1 \ldots e'_n\},$$

and

$$ds' = \{e'_1 \ldots e'_n\} - \{e_1 \ldots e_n\}.$$

Let i be the lowest integer such that $e_i \in ds$. If adding it to the set of edges creates a cycle, remove another edge from the set which is in ds' and forms a cycle. If not, remove any other edge that is in ds'. Since the greedy step selected the lowest

cost edge, this replacement will not reduce the cost of spanning tree. Therefore, the created tree is of minimum weight.

2.4 Related Work in Automated Algorithm Development

Smith with his Kestrel Interactive Development System (KIDS) [38] has approached correct and efficient algorithm development in two parts, techniques of automation process and the concept theory of algorithm design. Algorithm design tactics, such as dynamic programming and greedy algorithms, play the most important role when solving a problem. Smith [35] explains that his studies are not about how effective the tactics are, rather, it is about how to apply the tactics to several types of problems. First, the problem is formally described with specifications, then the theory is applied to derive the algorithm. When making a theory for a problem, all of the definitions, laws, and inference rules of a problem instance should be considered. Smith has worked on software development by refinement techniques [32] and mechanizing the development of software [35]. Following datatype refinements, a correct and efficient program also known as concrete specification is developed from an abstract specification.

All of the KIDS transformations preserve correctness and are automatic, with the exception of choosing algorithm design technique [38]. From the user's point of view the system allows the user to make high-level design decisions like, "design a divide-and-conquer algorithm for that specification" or "simplify that expression in

context". Software development differs from algorithm development in elements of software design such as data structure, optimization techniques, and runtime environment. However, they are similar in algorithm design elements such as the application domain, system requirement, and category of specifications. Furthermore, rules of morphisms and colimits are introduced and applied to refine them.

During development, the user views a partially implemented specification annotated with input assumptions, invariants, and output conditions. Algorithm development by KIDS typically consist of following steps:

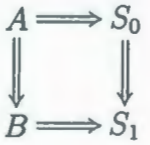
- Develop a domain theory
- Create a specification
- Apply a design tactic
- Apply optimizations
- Apply data type refinements
- Compile

2.4.1 Software Development by Refinement

Software development is applying a sequence of refinements to a specification S_0 [35]. The following diagram illustrates this formal process where S_i, $i = 0, 1, \ldots, n$, are structured specifications and arrows $\Downarrow$ are refinements. In each refinement step a design decision is made from S_i to S_{i+1} which cuts down the number of possible implementations. This refinement process turns a high-level specification S_0 to a

low-level specification S_n. Then, a morphism translates specification S_n to a code in a programming language. Thus, constructing specifications and refinements are the main parts to software development by refinement.

Two libraries are created for this purpose, a library of specifications including common datatypes and common mathematical structures, and a library of refinements such as algorithm design, datatype refinement, and expression optimization. Therefore, the use of abstract design knowledge which is expressed as reusable refinements has an important role is the development process. An application of this refinement is illustrated in the following diagram:



where a refinement $A \Longrightarrow B$ from a library is selected, then a classification arrow $A \Longrightarrow S_0$ is constructed, and the colimit of $B \Longleftarrow A \Longrightarrow S_0$ is computed. This process results in a refinement arrow of $S_0 \Longrightarrow S_1$.

2.4.2 Application

The scheduling of a set of jobs on a processor is subject to a precedence relation that constraints the order in which jobs can run [38]. Suppose that each job completes in unit time, that each job has a deadline, and that we wish to minimize the number of jobs that fail to complete before their deadlines. If we define a schedule to be an ordering of a given set of jobs that is consistent with a given precedence relation, the optimization problem is to minimize the cost function which is the number of jobs in a schedule that fail to complete before their deadline.

2.4.2.1 Development of a Domain Theory

The first and the most difficult step is formalizing the domain theory [38]. A theory presentation is comprised of sets of imported theories, type definitions, function specifications with optional operational definitions, laws, and rules of inference. The following is a domain theory for the job scheduling problem. A schedule is a linear arrangement of a set of jobs which can be expressed in terms of a bijection.

$$Injective(M : seq(integer), S : set(integer)) : boolean$$

$$= range(M) \subseteq S$$

$$\wedge \forall(i, j)(i \in domain(M) \wedge j \in domain(M) \wedge i \neq j \implies M(i) \neq M(j))$$

$$Bijective(M : seq(integer), S : set(integer)) : boolean$$

$$= Injective(M, S) \wedge range(M) = S$$

Now, the concept that a schedule must be consistent with the given precedence

relation is captured in the following definition and associated laws:

$$Consistent(S: \ seq(JOB), P: \ binrel(JOB,JOB)) : boolean$$
$$= \forall(i,j)(i \in range(S) \ \wedge \ j \in range(S) \ \wedge \ \langle i,j \rangle \in P$$
$$\Longrightarrow Index(i,S) < Index(j,S)),$$

where $Index(i,S)$ returns the index of element i in sequence S.

$$\forall(p)(Consistent([],P) = true),$$
$$\forall(a,\ p)(Consistent([a],P) = true)$$

and

$$\forall(S1,S2,P)(Consistent(concat(S1,S2),P)$$
$$= (Consistent(S1,P) \wedge Consistent(S2,P)$$
$$\wedge \text{Cross-Consistent}(range(S1), range(S2), P)))$$

where

$$Cross\text{-}Consistent(R1 : set(JOB), R2 : set(JOB), P : binrel\langle JOB, JOB\rangle) : boolean$$
$$= \forall(I,J)(I \in R1 \ \wedge \ J \in R2 \Longrightarrow \langle J,I\rangle \notin P)$$

2.4.2.2 Create a Specification

A specification can be presented as a quadruple $F = \langle D,R,I,O\rangle$ where D is the input type which should satisfy $I : D \rightarrow boolean$, the input condition [38]. The output type is R and the output condition that should be satisfied by a feasible solution is $O : D \times R \rightarrow boolean$. Having defined a specification in this format, the

following program format can be derived:

$$\text{Function } F(x : D)$$

$$\text{where } I(x)$$

$$\text{Returns } \{z : R \mid O(x, z)\}$$

$$= Body$$

The user enters a specification stated in terms of the underlying domain theory. Formally the problem of enumerating schedules can be specified as follows.

$$Schedules(Jobs : set(JOB), Precedes : binrel(JOB), (JOB))$$

$$\text{where } Irreflexive(Precedes, Jobs)$$

$$\text{returns } \{S : seq(JOB) \mid Bijective(S, Jobs) \wedge Consistent(S, Precedes)\}$$

2.4.2.3 Apply a Design Tactic

The user selects an algorithm design tactic from a menu and applies it to a specification [38]. For this example global search design tactic has been selected.

Theorem 13. *Let G be a global search theory. If Φ is a necessary filter then the following program specification is consistent.*

$$Function\ F(x : D) : set(R)$$

$$where\ I(x)$$

$$returns\ \{z \mid O(x, z)\}$$

$$= If\ \Phi(x, s_0(x))$$

$$Then\ F\_gs(x, s_0(x))$$

$$Else\ \{\}$$

where

$$
\begin{aligned}
&\textit{function } F\_gs(x : D,\ s : S) :\ set(R) \\
&\textit{where } I(x)\ \&\ J(x,s)\ \&\ \Phi(x,s) \\
&\textit{returns } \{z \mid \textit{Satisfies } (z,s)\ \&\ O(x,z)\} \\
&= \{z \mid \textit{Extract } (z,s)\ \&\ O(x,z)\} \\
&\qquad \cup\ \textit{reduce}(\cup, \{F\_gs(x,t)\ \mid \textit{Split}(x,s,t)\ \&\ \Phi(x,t)\}).
\end{aligned}
$$

On input x the program F calls F_gs with $s_0(x)$ and unions together all solutions that can be directly extracted from the space and the union of all solutions found recursively by splitting and surviving the filter.

The algorithm development process summarized in the above steps creates an algorithm for job scheduling problem using global Search technique. Further details on the process and the programming syntax can be obtained from [38, 32, 35, 36].

function *SCHEDULES-GS*

(JOBS:set(JOB), Precedes: binrel(JOB,JOB), ps:seq(integer))

where *Irreflexive(Precedes,Jobs)* $\wedge$ *range(ps)* $\subseteq$ *Jobs*

$\wedge$*Consistent(ps,Precedes)* $\wedge$ *Injective(ps,Jobs)*

$\wedge$ *@iCross-Consistent@(range(ps),Jobs|range(ps),Precedes)*

returns $\{$*SCHED* | *Extends(SCHED,ps)* $\wedge$ *Consistent(SCHED,Precedes)*

$\wedge$ *Bijective(SCHED,Jobs)*$\}$

$=$ $\{$*SCHED* | *Consistent(SCHED,Precedes)* $\wedge$ *Bijective(SCHED,Jobs)*

$\wedge$*SCHED=ps*$\}$

$\cup$ *reduce*($\cup$, {*SCHEDULES-GS*(*Jobs*,*Precedes*,*@iNew-pas@*)

/ *Consistent*(*@iNew-ps@*, *Precedes*)

$\wedge$ *Injective*(*@iNew − ps@*, *Jobs*)

$\wedge$ *@iCross-Consistent*

@(*range*(*@iNew-ps@*),*Jobs*\|*range*(*@iNew-ps@*),*Precedes*)

$\wedge \exists(I)$ (*@iNew-ps@* = *append*(*ps*,*I*) $\wedge$ $I \in$ *Jobs*)})

function *@iSCHEDULES@*(*Jobs* :*set*(*JOB*),*Precedes* :*binrel*(*@iJOB@*,*@iJOB@*))

where *Irreflexive*(*Precedes*, *Jobs*)

returns {*SCHED* | *Bijective*(*SCHED*, *Jobs*) $\wedge$ *Consistent*(*SCHED*, *Precedes*)}

= if *@iCross-Consistent@*(*range*([]), *Jobs**range*([]), *Precedes*)

then *SCHEDULES-GS*(*Jobs*, *Precedes*, [])

else {}

Chapter 3

Formal Dynamic Programming

Dynamic programming is a recursive approach to solving optimization problems by dividing a problem instance into subinstances, then combining obtained solutions to the subinstances to create a solution to the original instance. It is used typically for optimization problems, but has a broader usage to create algorithms for non-optimization problems. By storing solutions to solved subinstances of a problem, dynamic programming algorithms can be particularly efficient.

This chapter includes a formal approach for divide-and-conquer algorithm that is used to define dynamic programming algorithms, top-down and bottom-up. The approaches to solving problems are based on the specialization of an abstract dynamic programming algorithm. This provides not only the reusing of the algorithm, but also reusing of its proof. Finally, applications of dynamic programming including matrix chain multiplication and largest black square are presented.

3.1 Notations

Primed and unprimed variables in specifications

A assignments such as $x := E$ changes the value of the variable x. In specifications, the initial value of x is denoted by the unprimed variable x, and the final value of x is denoted by the primed variable x'.

Refinement $\sqsubseteq$

Let P and Q be specifications, P is refined by Q if every behavior accepted by Q is accepted by P

$$P \sqsubseteq Q.$$

For example, let

$$\begin{aligned} f &= \langle x' > x \rangle \\ g &= \langle x' = x + 1 \rangle \end{aligned}$$

since every behavior of g is accepted by f, we can say g refines f

$$f \sqsubseteq g.$$

Set notations

Filtering set builder

If x is a variable, S is a set and P is some boolean expression describing x then

$$\{x \in S \mid P\}$$

represents the subset of S such that contains exactly elements that fit the description P. For example

$$\{n \in \mathbb{Z} \mid n < 0\}$$

is the set of negative integers.

Mapping set builder

If x is a variable, S is a set and E is an expression then

$$\{x \in S \cdot E\}$$

is the set of all values of E where x is a value of S. For example

$$\{y \in \mathbb{N} \cdot 2n + 1\}$$

is the set of positive odd numbers.

Filtering and mapping set builder

The full set builder notation combines filtering with mapping. It first filters and then maps. The set

$$\{x \in S \mid P \cdot E\}$$

is the set of all values of the expression E where x is a value of the set S such that the boolean expression P is true. For example

$$\{i \in \mathbb{Z} \mid -10 < i < 10 \cdot 2i\}$$

is the set of even numbers between -20 and 20

$$\{-18, -16, ..., 16, 18\}.$$

Maximum and minimum set builder

Set builder can be used for describing maxima and minima. The set

$$\mathsf{max}\,\{x \in S \mid P \cdot E\}$$

is the maximum over the set of all values of the expression E where x is a value of the set S such that P is true. For example

$$\mathsf{min}\{n \in \mathbb{N} \mid n \text{ is prime} \cdot n^2\}$$

is the minimum over the set of all squares of prime numbers, which is 4.

Let declaration

In algorithms the value of variable x can be restricted using the let declaration

$$\mathsf{let}\, x \mid P \cdot S$$

where P is true. For example

$$\mathsf{let}\, n \mid q = p(n)$$

assigns the value of n to be some i where $q = p(i)$.

Sets of consecutive integers

The finite set of consecutive integers

$$\{0, ..k\}$$

is the set of all integers i such that $0 \leq i < k$.

And, the finite set of consecutive integers

$$\{0, .., k\}$$

is the set of all integers i such that $0 \leq i \leq k$.

Through

In writing algorithms, the for loop is denoted by

$$\textsf{for } i : 0 \textsf{ through } n \cdot S$$

which represents the following loop that is executed with the integer values of i ascending

$$S[i : 0];\ S[i : 1];\ \ldots ; S[i : n].$$

3.2 Introduction

Algorithm design approaches, such as greedy algorithms, dynamic programming, divide-and-conquer, and binary search, are generally taught and understood as informal ideas. Can we capture each algorithmic approach formally?

We investigate how abstract specifications can be proved to be implemented by abstract algorithms. By applying a transformation that maps the abstract specification into a concrete specification, we can derive a concrete algorithm from the abstract algorithm. This allows the abstract algorithm to be reused, along with its proof, to solve multiple concrete problems. The approach is summarized as follows. Suppose we know that an abstract specification P is implemented by an abstract algorithm Q,

then if we need an algorithm for a problem $R = T(P)$, where T is a data transform, we can refine R with $T(Q)$.

One of the design approaches presented in this thesis is dynamic programming, which is a recursive approach to solving optimization and other problems [27, 13]. Like the divide-and-conquer method, it works by finding solutions to subinstances of a problem and combining obtained solutions to the subinstances. Unlike divide-and-conquer, dynamic programming saves the solutions to subinstances for possible later use. There are two approaches to implementing dynamic programming: top-down and bottom-up.

In this part we will discuss an approach to solving problems based on concretization of top-down and bottom up abstract dynamic-programming algorithms. Along the way, we also formalize the closely related divide-and-conquer approach.

First, let's consider two concrete problems to which we can apply our techniques.

3.3 Introduction to Applications

3.3.1 The Matrix Chain Multiplication

Matrix Chain Multiplication is the problem of finding the minimum cost of calculating the product of a sequence of matrices $A_1 A_2 \ldots A_n$ [6]. Each matrix A_i has dimensions d_{i-1} by d_i .

The cost of multiplying one single matrix is zero, and the cost of multiplying two matrices $A_i A_{i+1}$is $d_{i-1} \times d_i \times d_{i+1}$. The cost of any matrix chain multiplication, consisting more than two matrices, depends on how the chain is split and how

the two subchains are multiplied. Consider the following matrix chain example of four: $A_1A_2A_3A_4$. A feasible solution is the parenthesization $((A_1A_2)(A_3A_4))$. Its corresponding cost is the sum of the following three parts:

(a) the cost of first subproduct (A_1A_2), $d_0 \times d_1 \times d_2$

(b) the cost of the second subproduct (A_3A_4), $d_2 \times d_3 \times d_4$

(c) the cost of multiplying the two matrices resulted from the subproducts $A_{1..2}$ and $A_{3..4}$, $d_0 \times d_2 \times d_4$.

Thus, the optimal cost of the product $A_iA_{i+1} \ldots A_k \ldots A_j$, where $i < j$, is the minimum, over all k such that $i < k < j$, of the sum of

(a) the optimal cost of calculating $A_{i..k}$,

(b) the cost of calculating $A_{k+1..j}$, and

(c) $d_i \times d_k \times d_j$.

For the matrix chain problem $A_{1..n}$, the problem instance space only includes the dimensions of the matrices to be multiplied. Each problem instance is a number $n \geqslant 2$ and a set of chains of indices: $\mathbb{N}^n$. The problem asks for the minimum cost to do the multiplication.

3.3.2 The Largest Black Square

The problem is to find the size of the largest black square in a black and white image. We will represent the image with a constant Boolean array $M \in X \times X \to Y$ where $X = \{0, ..N\}$, $Y = \{0, .., N\}$ and $N \in \mathbb{N}$ is a constant. Each black pixel is represented by *true* and each white pixel by *false*. Let $Y = X \cup \{N\}$. Pixels are indexed by X

while corner points are indexed by Y. The problem is to find

$$\max\{(p,q) \in Y \times Y \cdot lsea(p,q)\}$$

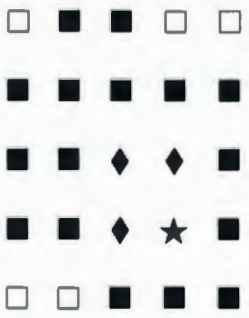
where *lsea* stands for 'largest black square ending at' and is defined for each corner by

$$lsea(p,q) = \max\{r \in \{0,..\min\{p,q\}\} \mid square(p,q,r)\}$$

where

$$square(p,q,r) = (\forall i \in \{p-r,..p\}, j \in \{q-r,..q\} \cdot M(i,j))\,.$$

As illustrated in the following figure, we can find the largest square ending at a corner point (p,q) (marked as ★ in the figure), if we know the sizes of the largest squares ending at each of its three neighbors to the north, west, and north west (marked as ♦ in the figure).



In this chapter we apply dynamic programming to the Matrix Chain Multiplication and the Largest Black Square problems, to further discuss the presented approach.

3.4 Divide-and-Conquer

3.4.1 The idea of Divide-and-Conquer

Divide-and-conquer is a recursive approach to solving problems [6, 13]. This method divides the problem instance to number of subinstances in order to determine a solution by combining the solutions to the subinstances. The divide-and-conquer method proceeds in three steps: *divide*, *conquer*, and *combine*.

Initially, the instance is divided to subinstances. Each of the subinstances are solved yielding a set of solutions to the subinstances. The resulted solutions will be combined to create a solution to the original instance.

3.4.2 Formal Divide-and-Conquer

Consider a space of problem instances P, a space of solutions S, and a function $f : P \rightarrow S$, mapping problem instances to solutions. Formally, given a problem instance p we need to compute $f(p)$. The specification of a problem can be written in the following definition in SIMPLE [31, 30].

Definition $EvaluateFunction(p)$ ::=

slot P : *set*

slot S : *set*

slot $f : P \rightarrow S$

require $p \in P$

ensure $s' = f(p)$

In order for the divide-and-conquer strategy to be applicable, we need to define the following entities:

$$PB \text{ and } PL \text{ are sets such that } PL \cup PB = P \tag{3.1}$$

$$divide : PB \rightarrow 2^P \text{ is a function} \tag{3.2}$$

$$combine : PB \times 2^{(P \times S)} \rightarrow S \text{ is a function} \tag{3.3}$$

We assume that leaf instances PL are easy to solve and branch problems PB will be solved recursively. For all $p \in PB$ and $A \subseteq P \times S$ we require:

$$f(p) = combine(p, A) \text{ provided for all } q \in divide(p),\ (q, f(q)) \in A \tag{3.4}$$

and that *divide* induces a well-founded order on P. In a well-founded order, there is no infinite descending chain of members. A totally ordered set $(A, \leq)$ is said to have a well-founded order if and only if every nonempty subset of A has a least element [19]. In another way:

$$q\ pred\ p \equiv q \in divide(p)$$

$$(q \leq p) \equiv q\ pred^+\ p.$$

The abstract divide-and-conquer algorithm can be written formally as a functional program DC that refines f.

```
Function DC(p)
  if p ∈ PL then return f(p)
  else let D = divide(p)
       let A = {q ∈ D · (q, DC(q))}
       return combine(p, A)
```

Theorem. under conditions (3.2), (3.3), $s := f(p) \sqsubseteq DC(p)$

Proof. This theorem can be proved by the method of induction, on p. An input p of function DC is either a leaf or a branch problem instance. The base case of the induction is when p is a leaf problem instance. According to the assumption made by the proposed theory its solution can be easily calculated, therefore, $DC(p) = f(p)$. The inductive hypothesis assumes there exists a function *divide* which creates all possible subinstances of p, and all subinstances are solved.

$$\text{for all } q \in \mathit{divide}(p) \cdot DC(q) = f(q)$$

Now, it is required to show that $f(p)$ for branch problems can be calculated. According to equation (3.4), $f(p)$ can be calculated by function *combine* which joins required solutions to the subinstances where for all $q \in \mathit{divide}(p) \cdot DC(q) = f(q)$. If *combine* can be defined, $f(p)$ is computed and provides $DC(p) = f(p)$. From this inductive proof we can conclude $f(p) \sqsubseteq DC(p)$, if the assumptions are met. Therefore, if there exist functions *divide* and *combine* to satisfy the requirements, the theorem is proved by the method of induction. □

3.5 Dynamic Programming

3.5.1 The Idea of Dynamic Programming

Dynamic programming is very close to divide and conquer in the concept of dividing the problem instance into subinstances, then using the solutions to the subinstances to

create a solution. Therefore, in this section, formalization of the closely related divide-and-conquer approach is used as a basis for dynamic programming algorithms. Since subinstances of problems share the same structure as the problem, the solution to a general problem instance provides a solution to the subinstances within it. Therefore, dynamic programming provides a recursive approach to finding a solution to a problem instance by recursively finding solutions to subinstances. It also reuses solutions to shared subinstances in order to reduce the run-time of the program.

There are two methods of dynamic programming, top-down and bottom-up, which are discussed later in this section with more details. In the top-down method the calculated solutions are saved and reused several times in a program, which is called *memoization.* For bottom-up, tables are used to save the solutions to the subinstances in order of dependency for later use. This is called *tabulation* [6, 2].

3.5.2 Formal Dynamic Programming

Formal dynamic programming uses divide-and-conquer with a modification described later in the top-down and bottom-up sections. The function divide should meet the main concepts of dynamic programming and how to create subinstances of problems that meet the principle of optimality. Dynamic programming is applicable to a problem when there exist a function *divide* and a function *combine* which create PB and PL according to formulas 3.1, 3.2, and 3.3. A variable A stores the subinstances of a problem and solutions to those subinstances.

At any given time during running the algorithm, its initiation, and its termination there should be no false information in variable A. That is considered as an invariant

for variable A. The postcondition is that A contains all the subinstances of a problem instance p within itself. Furthermore, space A is expanded when new information is entered and no information is lost or removed from it.

3.5.3 Top-Down Dynamic Programming

One of the approaches to implementing dynamic programming is the top-down approach [6]. The proposed formal divide-and-conquer definition is used as the basis for a top-down dynamic programming algorithm. We regard the top-down dynamic programming approach to be a special case of divide-and-conquer combined with memoization, that is, the storing of solutions to the subinstances.

To apply memoization to the existing divide-and-conquer definition, a variable A is used as a table to store calculated results. It stores a set of pairs (p, s) that satisfies $s = f(p)$. We write $A(p)$ to mean the solution that is paired with p in A. As an invariant A represents a partial function. We can get a top-down algorithm using the refinement of function $DC(p)$.

Definition $DynamicTD(p)$ ::=

inv $\forall (q,t) \in A \cdot t = f(q)$

var $A : P \rightarrow S := \emptyset$

proc $Solve(p : P)$

post $(p, f(p)) \in A' \ \wedge \ A' \supseteq A$

if $\exists s \cdot \ (p,s) \in A$ then $A(p)$

else if $p \in PL$ then (

```
            let s := f(p)
            A := A ∪ {(p, s)})
        else (
            let D := divide(p)
            for q ∈ D · Solve(q)
            assert ∀q ∈ D · (q, f(q)) ∈ A
            let s = combine(p, A)
            A := A ∪ {(p, s)})
    end proc Solve
    Solve(p)
    s := A(p)
}
```

That this algorithm refines the dynamic programming problem is expressed in SIMPLE as a theorem [31, 30].

```
Theorem EvaluateFunction(p) ⊑ Solve(p)
        where EvaluateFunction(p) =
                (∀(q, t) ∈ A · t = f(q)) ⇒
                (∀(q, t) ∈ A' · t = f(q)) ∧
                    (∃s · (p, s) ∈ A')
```

Then, we have that:

Theorem $s := f(p) \sqsubseteq DynamicTD(p)$

3.6 Bottom-up Dynamic Programming

The other approach to implement dynamic programming is the bottom-up approach [6]. The same proposed formal divide-and-conquer definition is used as a basis for the bottom-up algorithm for dynamic programming. The technique which applies to the bottom-up approach is tabulation, which is slightly different than memoization. In this method, there could be some subinstances that are solved but never used, which does not happen in top-down approach. All possible subinstances are solved, stored, and combined to build a solution to the main problem. The bottom-up approach avoids the memory and time overhead of recursive calls. There is no need to implement the divide function because it has been considered in the structure of the bottom-up approach and creates a new level of subinstances in each step. Instead, to get a bottom-up algorithm, we need, for each problem p, a sequence of problems $r(i)$ so that $p = r(i)$, for some i and so that, for each i, either $r(i)$ is a leaf or all problems in $divide(r(i))$ are in $\{r(0), r(1), \ldots, r(i-1)\}$.

We can get a bottom-up algorithm using the refinement of function $DC(p)$.

Definition $DynamicBU(p)$ ::=

inv $\forall(q,t) \in A \cdot t = f(q)$

var $A : P \rightarrow S := \emptyset$

proc $Solve(p : P)$

```
        let n | p = r(n)
        post (p, f(p)) ∈ A' ∧ A' ⊇ A
        for i : 0 through n (
          if r(i) ∈ PL then (
            let s := f(r(i))
            A := A ∪ {(r(i), s)})
          else (
            assert ∀q ∈ divide(r(i)) · (q, f(q)) ∈ A
            let s = combine (r (i) , A)
            A := A ∪ {(r(i), s)}))
      end proc Solve
      Solve(p)
      s := A(p)
    }
```

That this algorithm refines the dynamic programming problem is expressed in SIMPLE as a theorem.

```
Theorem EvaluateFunction(p) ⊑ Solve(p)
        where EvaluateFunction(p) =
                (∀(q, t) ∈ A · t = f(q)) ⇒
                (∀(q, t) ∈ A' · t = f(q)) ∧
                  (∃s · (p, s) ∈ A')
```

And therefore:

Theorem $s := f(p) \sqsubseteq DynamicBU(p)$

3.7 Application of Dynamic Programming

3.7.1 Matrix chain Multiplication

To understand the Matrix Chain Multiplication problem as an instance of the general *EvaluateFunction* specification, we need to fill in the three slots of the specification.

- Define P to be $\mathbb{N}^+$, the set of all finite sequences of natural numbers with length at least two, $d_{0..n} ::= (d_0, d_1, \ldots, d_n)$.
- Define S to be $\mathbb{N}$, the set of natural numbers.
- Define f to be the function that maps sequences of natural numbers to a natural number that is the minimum cost of multiplying the corresponding matrix sequence: We define f recursively as

$$\begin{aligned} f(d_{0..1}) &= 0 \\ f(d_{0..n}) &= \min_{k \in \{1,..n\}} f(d_{0..k}) + f(d_{k..n}) + d_0 \times d_k \times d_n, \text{ if } n > 1 \end{aligned}$$

Filling the three slots S, P, and f, with these definitions adapts the problem.

To adapt the top-down dynamic solution we need to determine the $PL, PB, divide$, and *combine* slots.

- Define PL to be the set of all sequences of natural numbers with length two.

$$PL = \mathbb{N}^2$$

- Define PB to be the set of all finite sequences of natural numbers with length greater than two,

$$PB = \bigcup_{n \in \mathbb{N} | n > 2} \mathbb{N}^n$$

- Define *divide* to be the function that generates all the subsequences of the sequence $(d_0, d_1, \ldots, d_n)$ that are required in the process of producing the result,

$$\begin{aligned} divide(d_{0..n}) &= \{k \mid 0 < k < n \cdot d_{0..k}\} \\ &\quad \cup \{k \mid 0 < k < n \cdot d_{k..n}\}. \end{aligned}$$

- Define *combine* to be the function that calculates the cost of a problem instance using the solved subinstances stored in variable A.

$$combine(d_{0..n}, A) = \min_{k \in \{1,..n\}} \begin{pmatrix} A(d_{0..k}) + A(d_{k..n}) \\ + \quad d_0 \times d_k \times d_n \end{pmatrix}$$

- The variable A that stores the pair of problem instances and their corresponding cost that is used by the combine function.

Filling these slots adapts both top-down and bottom-up algorithms, first we derive the top-down algorithm of this example in the following algorithm.

Definition $MCMTD(p)$ ::=

inv $\forall (q, t) \in A \cdot t = f(q)$

```
var A : ℕ⁺ → ℕ := ∅
proc Solve(d_{0..n} : ℕ⁺)
  post (d_{0..n}, f(d_{0..n})) ∈ A' ∧ A' ⊇ A
  var s : ℕ
  if ∃s · (d_{0..n}, s) ∈ A then A(d_{0..n})
  else if n = 2 then (
        s := 0
        A := A ∪ {(d_{0..n}, 0)})
      else (
        for k : 1 through n − 1 (
          Solve(d_{0..k})
          Solve(d_{k..n}))
        assert ∀q ∈ divide(d_{0..n}) · (q, f(q)) ∈ A
        s := ∞
        for k : 1 through n − 1
          s := s min (A(d_{0..k}) + A(d_{k..n}) + d_0 × d_k × d_n)
        A := A ∪ {(d_{0..n}, s)})
end proc Solve
Solve(d_{0..n})
s := A(d_{0..n})
}
```

The optimization problem of matrix chain multiplication returns the minimum cost. In order to get the process of doing the multiplication we can store the break-points k of each sequence longer than two.

Next, is the bottom-up algorithm derived below for this example. In this algorithm, the instance sequence r is the segments of d of increasing length, starting with length 2 as follows

$$d_{0..1},\ d_{1..2}, \ldots,\ d_{n-1..n},$$

$$d_{0..2},\ d_{1..3}, \ldots,\ d_{n-2..n},$$

$$\vdots$$

$$d_{0..n}$$

```
Definition MCMBU(d_{0..n} : ℕ⁺) ::=
  inv ∀(q,t) ∈ A · t = f(q)
  var A : ℕ⁺ → ℕ := ∅
  proc Solve(d_{0..n} : ℕ⁺)
    post (d_{0..n}, f(d_{0..n})) ∈ A' ∧ A' ⊇ A
    for l : 2 through n + 1
    for i : 0 through n − l + 1 (
      let j = i + l − 1
      if l = 2 then A := A ∪ {(d_{i..j}, 0)}
      else (
        assert ∀q ∈ divide(d_{i..j}) · (q, f(q)) ∈ A
        var s := ∞
```

```
            for k : i + 1 through j − 1
                s := s min (A(d_{i..k}) + A(d_{k..j}) + d_i × d_k × d_j)
            A := A ∪ {(d_{i..j}, s)}))
    end proc Solve
    Solve(d_{0..n})
    s := A(d_{0..n})
}
```

More examples such as Radix-Code can be implemented in a very similar approach as Matrix chain multiplication.

3.7.2 Largest Black Square

To understand the Largest Black square problem as an instance of the general *EvaluateFunction* specification, we need to fill in the three slots of the specification.

- Define P to be the set of all corner points $(p, q) \in Y \times Y$, where $Y = \{0, .., N\}$
- Define S to be the set of numbers r such that $0 \leq r \leq N$
- Define f to be

$$f(p, q) = lsea(p, q)$$

where

$$lsea(p, q) = max\{r \in \{0, ..min\{p, q\}\} \mid square(p, q, r)\},$$

and

$$square(p, q, r) \quad = \quad (\forall i \in \{p - r, ..p\}, j \in \{q - r, ..q\} \cdot M(i, j)).$$

Filling the three slots S, P, and f, adapts the problem.

To adapt the bottom-up dynamic programming algorithm we need to determine PL, PB, *divide*, and *combine* slots.

- Define PL to be the set of all pairs (p, q) that are located on the most top row or the most left column,

$$PL = \{(p, q) \in Y \times Y \mid p = 0 \vee q = 0\}.$$

- Define PB to be all other points

$$PB = \{(p, q) \in Y \times Y \mid p \neq 0 \wedge q \neq 0\}.$$

- Define *divide* to be the function that generates the set of three neighbors of a point (p, q) to the north, west, and north west,

$$divide(p, q) = \{(p-1, q), (p-1, q-1), (p, q-1)\}.$$

  Note that these three neighbors are lexicographically prior to (p, q).

- Define *combine* to be the function that finds the size of the largest black square using the solved neighbor points stored in space A. The idea is that if a square is black, the largest square at (p, q) can not be larger than 1 plus the largest square ending at any of the neighbors generated by *divide*. On the other hand, there is a square ending at (p, q) that is of size 1 plus the minimum of the squares ending at the three neighbors.

$$\begin{aligned} & combine((p, q), A) \\ = \ & \textsf{if } \neg M(p-1, q-1) \textsf{ then } 0 \\ & \textsf{else } 1 + min\{A(p-1, q), A(p, q-1), A(p-1, q-1)\} \end{aligned}$$

These slots adapt both top-down and bottom-up algorithms. According to top-down algorithm approach and the introduced *divide*() and *combine*() functions, the following algorithm is derived for 'largest black square ending at' (p, q). The r sequence for this example is

$$(0,0),\ (0,1),\ldots,\ (0,n),$$
$$(1,0),\ (1,1),\ldots,\ (1,n),$$
$$\vdots$$
$$(n,0),\ (n,1),\ldots,\ (n,n)$$

Definition $LESATD$ ::=

inv $\forall((p,q),t) \in A \cdot t = lsea(p,q)$

var $A : Y \times Y \rightarrow \mathbb{N} := \emptyset$

proc $Solve(p,q)$

post $((p,q), lsea(p,q)) \in A' \ \wedge \ A' \supseteq A$

var $s : \mathbb{N}$

if $\exists s.\ ((p,q),s) \in A$ then $A(p,q)$

else if $p = 0 \ \vee \ q = 0$ then(

$s := 0$

$A := A \cup \{((p,q),0)\}$)

else (

$Solve(p-1,q)$

$Solve(p-1,q-1)$

$Solve(p,q-1)$

```
        assert ∀(p, q) ∈ divide(p, q) · ((p, q), lsea(p, q)) ∈ A
        s := 1 + min {A(p − 1, q), A(p − 1, q − 1), A(p, q − 1)}
        A := A ∪ {((p, q), s)})
    end proc Solve
    Solve(N, N)
    s := A(N, N)
}
```

For the bottom-up algorithm the other decision that needs to be made is the ordering of the instances so that subinstances are solved before super-instances. For this problem, instances can be ordered lexicographically. According to the same *divide*() and *combine*() functions and the bottom-up algorithm, the following algorithm is derived for Largest Black Square ending at (p, q). In this algorithm, to create all the subinstances from the first level to the desired one, cross points are considered in one loop. This loop starts from the point $(0, 0)$ to (p, q) in one loop starting from 0 to $p \times N + q$, where the row and column number are derived by div and mod.

```
Definition LEASBU ::=
  inv ∀((p, q), t) ∈ A · t = lsea(p, q)
  var A : Y × Y → ℕ := ∅
  proc Solve(p, q)
    post ((p, q), lsea(p, q)) ∈ A' ∧ A' ⊇ A
    for i : 0 through p × N + q (
```

```
        let a = i div (N + 1)
        let b = i mod (N + 1)
        var s : ℕ
        if a = 0 ∨ b = 0 then (
          s := 0
          A := A ∪ {((a, b), 0)})
        else (
          if ¬M(a − 1, b − 1)
            s := 0
          else (
            assert ∀(p, q) ∈ divide(a, b) · ((p, q), lsea(p, q)) ∈ A
            s := 1 + min {A(a − 1, b), A(a, b − 1), A(a − 1, b − 1)})
          A := A ∪ {((a, b), s)}))
    end proc Solve
    Solve(p, q)
    s := A(p, q)
}
```

This algorithm serves to calculate the *lsea* function for each intersection point and store the result in the A table. To find the largest square is now just a matter of looking for the largest value in the table.

Chapter 4

Formal Greedy Algorithms

Greedy Algorithms solve optimization problems, sequentially making locally optimum choices to make a globally optimum solution. In this chapter, we are investigating how abstract specifications can be proved to be implemented by abstract greedy algorithms. We provide a formal structure to greedy algorithm in a predicative style. Applications of greedy algorithms include Kruskal's algorithm, and Huffman codes.

4.1 The Ideas of Greedy Algorithm

Greedy Algorithm is an optimization problem solving approach, sequentially making locally optimum choices to make a globally optimum solution. Applying the greedy algorithm won't provide an optimal solution to all problems. However, for some problems, making the best choice in each step leads to an optimum solution. For a greedy algorithm to be applicable to a problem, the greedy structure should be satisfied in a problem [6].

Some parameters are required to create a greedy algorithm, including a global

cost function and a local cost function. The former function measures the degree of optimality in a feasible solution, and the latter function is used in each step of the algorithm for comparing partial solutions. There is a relation between local and global cost functions, in most cases being identical. However, it is not required for greedy algorithm to be defined having equal local and global cost functions.

4.2 Formal Greedy Algorithm

4.2.1 Defining Parameters and Transformations

The problem is specified by the following parameters of a problem space:

- Let S be a search space of partial solutions
- $i \in S$ is an initial partial solution
- $C \subseteq S$ is a set of completed solutions
- $g : S \rightarrow 2^S$ is a function that creates partial solutions from partial solutions where $g(x) = \emptyset$ when $x \in C$
- $cost : S \rightarrow \mathbb{N}$ is a function that calculates the cost of a partial solution
- $\leq_{global}$: $C \rightarrow C$ is a cost comparison relation, and if $x, y \in C$ then $x \leq_{global} y$ provides that $cost\ x \leq cost\ y$ with respect to relation *global*

To specify the greedy algorithm there is a list of entities derived from the above parameters for solution space:

- $\leq_{local}$: $S \rightarrow S$ is a cost comparison relation, and if $x, y \in S$ then $x \leq_{local} y$ provides that *cost* $x \leq$ *cost* y with respect to relation *local*

- $g : 2^S \rightarrow 2^S$ is the extension of g to sets

  - $g(\emptyset)=\emptyset$
  - $g(\{x\}) = g(x)$
  - $g(X \cup Y) = g(X) \cup g(Y)$.

- g^* is the transitive and reflexive closure of g

$$g^*(X) = X \cup g(X) \cup g(g(X)) \cup \ldots \tag{4.1}$$

  Note that

$$x \in C \text{ implies } g^*(x) = g^*(\{x\}) = \{x\} \tag{4.2}$$

- $\min\limits_{\leq} X$ is the set of all minima of X with respect to a relation $\leq$

$$\min_{\leq} X = \{x \in X \mid (\forall y \in X \cdot x \leq y)\}$$

The goal set G is defined as the globally optimal set of completed solutions reachable from the starting point i by zero or more applications of the generating function, g. That is

$$G = \min_{\leq global} (C \cap g^*(\{i\})) \tag{4.3}$$

In many cases, all completed solutions can be generated from i, in which case

$$G = \min_{\leq global} C$$

A problem can be described by the following definition of *Search*, ensuring that there is a result in goal set G for it.

Definition *Search* ::=

ensure $x' \subseteq G$

The solution based on the greedy algorithm is presented in the following definition of *GreedySearch*. In this definition, $x :\in S$ is an operator that assigns an arbitrary element of S to variable x.

Definition *GreedySearch* ::=

var $x := i$

while $x \notin C$ inv $reaches(x, G)$

$x :\in \min_{\leq local} g(x)$

The invariant says that some member of the goal is reachable from x.

$$\text{inv: } reaches(x, G) = (g^*(\{x\}) \cap G) \neq \emptyset \tag{4.4}$$

Therefore, if the invariant holds *Search* is refined by *GreedySearch*.

Theorem 14. under conditions (4.6), (4.7) and (4.8)

Search $\sqsubseteq$ GreedySearch

Proof. For the greedy algorithm to be correct, we must check that the body of the while loop preserves the invariant, $reaches(x, G)$. There are also two cases to be considered for the relation between the local and global optimality conditions: *a*) when they are identical, *b*) when not identical global condition can be derived from

the local condition, $x \leq_{global} y \Longrightarrow x \leq_{local} y$. The correctness of the algorithm is verified in the following three phases:

1. Prior to initiating the while loop, when variable x is initialized:

   $reaches(x, G)$ is a post condition of $x := i$ which becomes $reaches(i, G)$

   $$\begin{aligned} & reaches(i, G) \\ = \ & \text{by def (4.4)} \\ & (g^*(i) \cap G) \neq \emptyset \\ = \ & \text{by the property } (A \subseteq B \Longrightarrow B \cap A = A) \\ & G \neq \emptyset \end{aligned}$$

   The resulting equation of $G \neq \emptyset$ is true if the problem has been defined correctly, i.e., in a way that there exists a solution.

2. We need to show that the invariant is preserved by $x :\in \min_{\leq local} g(x)$

   The while loop is running, meaning that the loop condition is true, $x \notin C$. There is also the assumption that the local optimality condition is either identical to the global optimality condition, or can be derived from it. We need to show that assuming $reaches(x, G)$ is true, then $reaches(x', G)$ will be true after the body of the loop is executed and produced $x' \in \min_{\leq local} g(x)$.

$$
\begin{aligned}
& reaches(x, G) \\
, \quad & x \notin C \\
& \text{Let } x' \in \min_{\leq local} g(x) \\
\stackrel{?}{\Longrightarrow} \quad & reaches(x', G)
\end{aligned}
$$

Consider H a set including all the cases that lead to G.

$$H = \{x \,|\, reaches(x, G)\}$$

If $x \in H$, *and* $x \notin C$ then at least one successor of x is also in H. Therefore,

$$\text{Let } y \in min(g(x) \cap H) \tag{4.5}$$

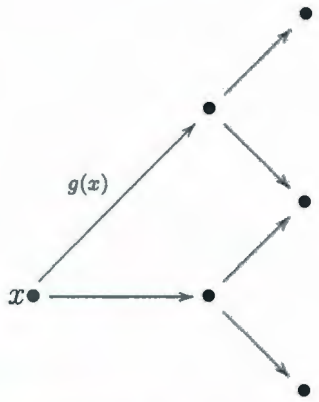
Let's get back to some features of greedy algorithm. First, the generation function $g(x)$ should imply that x is in C. This prevents the algorithm from running into a dead end. Otherwise the following could happen. Suppose variable x has a value of x_0 and we pick $x_1 \in \min g(x)$, now in the next iteration $g(x)$ is empty and the algorithm is stuck. Therefore,

$$g(x) = \emptyset \quad \text{implies that } x \in C \tag{4.6}$$

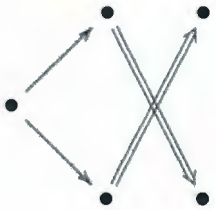
As stated in the introduction, studies have shown that there are several cases for greedy algorithm to be applicable. However, cases considered here are those with the monotonicity condition that is the second condition required for the generation function $g(x)$. Therefore,

$$a \leq_{local} b \Longrightarrow a' \leq_{local} b', \qquad \forall a, b, a' \in g(a), b' \in g(b) \tag{4.7}$$

This means when a partial solution is better in one step, the continuation of it will result in a better partial solution in the next step. As illustrated in the following diagram, taking generation function will not reverse the order of optimality.



Thus, the following condition marked by " $\Rightarrow$ " won't happen:



The studied case is known as Better-Local condition in [9].

Third, the generation function $g(x)$ should also generate a better partial solution than x with respect to the local cost relation. Otherwise, the algorithm could follow an ever descending path that never reaches C. Therefore,

$$g(a) \geq_{local} a, \qquad \forall a \tag{4.8}$$

These conditions also implies that when a completed solution is derived as a result of a better partial solution, the continuation of the worse partial solution

will not lead to a better completed solution.

$$a \leq b \Longrightarrow a \leq b', \qquad \forall a, b, b' \cdot a \in C \wedge b' \in C \wedge b' \in g(b) \tag{4.9}$$

Considering the above properties (4.6), (4.7) and (4.8) of greedy algorithms, H includes the set of generated successors of x. This concept helps rewriting the equation of (4.5) as

$$\textsf{Let}\ y \in min(g(x))$$

On the other hand, the body of the program considers

$$x :\in \min\ g(x)$$

Therefore, all the properties of y are properties of x'. So, $x' \in H$ meaning $reaches(x', G)$ is true.

We can summarize this as

$$\begin{aligned}
& reaches(x, G) \\
= \ & g^*(x) \cap G \neq \emptyset \\
= \ & \text{since } x \notin C \\
& g^*(g(x)) \cap G \neq \emptyset \\
= \ & \text{Distributivity} \\
& \exists y \in g(x) \cdot g^*(y) \cap G \neq \emptyset \\
\Rightarrow \ & \text{under conditions (4.6), (4.7) and (4.8)} \\
& \forall y \in \min\ g(x) \cdot g^*(y) \cap G \neq \emptyset \\
\Rightarrow \ & \text{since } x' \in \min\ g(x) \\
& reaches(x', G)
\end{aligned}$$

3. Following loop termination:

The while loop terminates when the loop condition is false, so $x \in C$.

$$
\begin{aligned}
& \mathit{reaches}(x, G) \\
= \quad & \text{by def (4.4)} \\
& (g^*(\{x\}) \cap G) \neq \emptyset \\
= \quad & \text{by def (4.2) and } x \in C \\
& (\{x\} \cap G) \neq \emptyset \\
\Rightarrow \quad & x \in G
\end{aligned}
$$

□

4.2.2 Optimization

The proposed method for greedy algorithms generally makes a greedy step in each state by applying the generation function $g(x)$ to every $x \in S$. Then, it selects the best $x :\in \min g(x)$ with respect to the cost function. However, there can be a more efficient implementation to the generation function. One optimization approach commonly applied, is to assess the cost function in order to determine a total ordering of the input. Revising the input as suggested, provides a more efficient implementation of the algorithm. It provides efficiency by making the highest priority item be evaluated earlier and removed from the input list. In order for this process not to affect the correctness of the algorithm, an optimized approach should be a refinement of the general method of greedy algorithms.

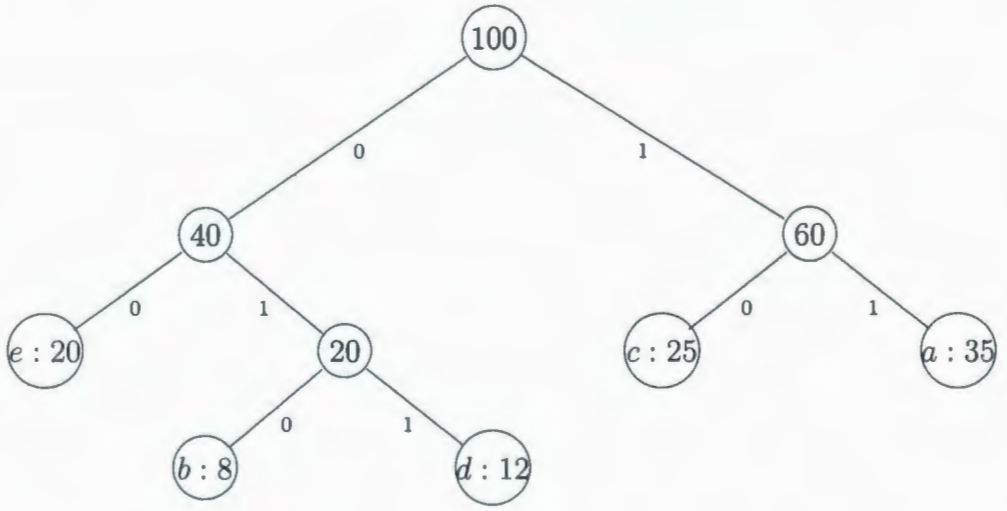
4.3 Application

In this section, application of greedy algorithms is studied using Kruskul's algorithm and Huffman Coding. Kruskal's algorithm is an example of minimum spanning tree algorithm. Huffman coding is a high efficiency algorithm that is been broadly used in compressing data.

4.3.1 Huffman Coding

Huffman coding is a method for compressing data considering the occurrence frequency of each character [6]. This method is highly efficient and is widely used to compact data. In order to encode the data, each character is given a code with a variable length. This can be more efficient than a fixed length code. The higher the occurrence frequency of a character, the shorter the code length assigned to it. Huffman code is a prefix code, there is no codeword constructed which is a prefix to another codeword. The data is represented by the set Ch, each character in it is $c \in Ch$, and the function $f : Ch \to \mathbb{N}$ provides the frequency number. In order to encode the data, a tree is created in the following way: in each step two characters (or sets) with the lowest frequency are chosen and encoded 0 for the left child, and 1 for the right child. The following tree illustrates the process of encoding characters

$a - e$, with the frequency of $f(a) = 35$, $f(b) = 8$, $f(c) = 25$, $f(d) = 12$, $f(e) = 20$.



4.3.1.1 Defining Slots

To specify the problem, it is required to fill the slots of the problem space:

- Define each element of S to be a set of binary trees. In order to define S, define BT as the set containing

$$\begin{cases} \langle c \rangle & \text{where } c \in Ch, \text{ or} \\ \langle l, r \rangle & \text{where } l, r \in BT \end{cases}$$

  Each element of S is a subset of BT. For any $x \in S$ and $c \in Ch$, the leaf $\langle c \rangle$ must appear in at least one member of x.

- Define i as a set of trees, each including one leaf for each element of Ch

$$i = \{\langle c \rangle \mid c \in Ch\}$$

- Define each member of C as a set of size one, $|x| = 1$, whose sole member is a binary tree including every $c \in Ch$

- Define g as a function merging two trees, by making subtrees as the left and right child in a new combined tree, and removing the original subtrees

$$g(x) = \{\{\langle t, s\rangle\} \cup x - \{t\} - \{s\} \mid t, s \in x \wedge t \neq s\}$$

- Define $cost_{global}$ of tree t as the sum of the cost of retrieving each char $c \in Ch$, which is depth of a char in a tree d times its frequency

$$\begin{aligned} cost(d, \langle c\rangle) &= f(c) \times d, && \text{for all } c \in Ch \\ cost(d, \langle s, t\rangle) &= cost(d+1, s) + cost(d+1, t) && \text{for } t, s \in Tree \\ cost(t) &= cost(0, t), && \text{for } t \in Tree \end{aligned}$$

- Define $\leq_{global}$ as a cost comparison relation between trees

$$\{t\} \leq_{global} \{s\} \equiv cost(t) \leq cost(s), \quad \text{for } \{t\}, \{s\} \in C$$

To adapt the greedy solution, following parameters are specified:

- Define $cost_{local}$ similar to the global cost function, noticing that the global cost applies to a completed solution which is a tree and the local cost applies to a partial solution which is a forest

$$cost_{local}(x) = \sum_{t \in x} cost(t), \qquad \text{for } t \in Tree,\ x \in S$$

- Define $\leq_{local}$ as a local comparison relation between two forests

$$x \leq_{local} y \equiv \sum_{t \in x} cost(t) \leq \sum_{s \in y} cost(s)$$

4.3.1.2 Implementation Process

In order to implement the concrete Huffman algorithm based on the abstract definition of greedy algorithm defined by Definition *GreedySearch* in 4.2.1, we need to fill in the slots using the parameters and definitions defined in 4.3.1.1. These slots include

- The input parameter is Ch, a set of chars and a frequency function $f(c) : Ch \rightarrow \mathbb{N}$

- Initialization: $\mathsf{var}\ x := i$

  Definition of i can be implemented using a loop

  $$x := \emptyset$$

  $$\mathsf{for}\ c \in Ch$$

  $$x := x \cup \langle c \rangle$$

- Defining the while guard: $\mathsf{while}\ x \notin C$

  To implement this part, consider

  - initialization of x creates sets with length $|Ch|$, $|x| = |Ch|$
  - the generation function $x' \in g(x)$ in each iteration joins two trees together reducing the length of a set by one , $|x'| = |x| - 1$
  - iteration ends when $x \in C$, which implies $|x| = 1$

  $$[|x| = |Ch|,\ |x'| = |Ch| - 1] \sqsubseteq x := x - 1$$

  This assignment is repeated $|Ch| - 1$ times until $|x| = 1$. Therefore this sequence can be implemented by a for loop starting from 1 to $|Ch| - 1$

  $$\mathsf{for}\ k := 1\ \mathsf{through}\ |Ch| - 1$$

- Generation step in $x :\in \min\ g(x)$

  According to proposed greedy algorithm in section 4.2.1, generation function produces all the possible combination of merging two trees together, from where the lowest cost forest is selected. However, Huffman method suggests an optimization case where trees are sorted according to their frequencies and generation function selects two subtrees of minimum frequency, merges them to create a new tree with subtrees at its children, and finally deletes the original subtrees from the set. Therefore, it is required to show how this method implies the local cost of the new value of x is optimal.

  Let's consider the properties of the cost function in more detail

$$cost(d, \langle c \rangle) = f(c) \times d, \text{ for all } c \in Ch \tag{4.10}$$

$$cost(d, \langle s, t \rangle) = cost(d+1, s) + cost(d+1, t) \text{ for } t, s \in Tree \tag{4.11}$$

$$cost(t) = cost(0, t), \text{ for } t \in Tree \tag{4.12}$$

$$cost(x) = \sum_{t \in x} cost(t), \text{ for } t \in Tree, \text{ and } x \in S \tag{4.13}$$

  Equation (4.10) represents the cost of a single char in a tree being its depth times its frequency. The next equation (4.11) describes the relation between a tree and its subtrees. A single tree can also be considered as a zero depth tree as represented in (4.12). The cost function is defined for a tree, however, it can be applied to a forest as represented in equation (4.13).

  Applying the proposed greedy algorithm in 4.2.1 constructs an algorithm for this application. Generally, the greedy approach makes a greedy step in each state by applying the generation function $g(x)$ to every $x \in S$. For this application,

greedy step produces all the possible cases of creating a new tree joining two subtrees. Then, it selects the lowest cost tree $x :\in \min g(x)$. However, Huffman method suggests an optimization technique to implement the encoding system. Using Huffman coding, this process is implemented by choosing two subtrees with the lowest frequency, placing subtrees as left and right child of a new tree, and finally removing the original subtrees. In this section, we show how this optimized technique satisfies the minimum cost of creating encoding trees.

Frequency function can be extended to trees, calculated according to frequency of each char

$$\begin{aligned} f(\langle c \rangle) &= f(c), && \text{for all } c \in Ch \\ f(\langle t, s \rangle) &= f(t) + f(s), && \text{otherwise} \\ \Longrightarrow \qquad f(t) &= \sum_{c \in t} f(c), && \text{for } t \in Tree \end{aligned} \tag{4.14}$$

Concluding easily from equation (4.10) and (4.12), cost of a tree can be expressed in terms of $d_{c,s}$, depth of a char c in tree t, times $f(c)$, frequency of char c.

$$cost(s) = cost(0, s) = \sum_{c \in s} d_{c,s} \times f(c), \quad \text{for all } s \in BT \tag{4.15}$$

Now, the cost of the same tree as a subtree s in depth 1 is calculated in a similar

way by adding 1 to the depth of each char in subtree.

$$\begin{aligned} cost(1,s) &= \sum_{c \in s}((d_{c,s}+1) \times f(c)) \\ &= \sum_{c \in s}(d_{c,s} \times f(c)) + \sum_{c \in s} f(c) \\ &= \text{by def (4.15)} \\ & \quad cost(0,s) + \sum_{c \in s} f(c) \\ &= \text{by def (4.14)} \\ & \quad cost(0,s) + f(s) \end{aligned}$$

The result implies the relation between the cost of a subtree s in depth 0 and its cost in depth 1.

$$cost(1,s) = cost(0,s) + f(s) \tag{4.16}$$

Therefore, def (4.11) for $d = 0$ can be rewritten as

$$\begin{aligned} cost(0,\langle s,t\rangle) &= cost(1,s) + cost(1,t) \\ &= \text{by (4.16)} \\ & \quad cost(0,s) + f(s) + cost(0,t) + f(t) \end{aligned}$$

This can be more simplified as

$$cost(\langle s,t\rangle) = cost(s) + f(s) + cost(t) + f(t) \tag{4.17}$$

Now, using the derived details we reconsider the proposed greedy algorithm for this application. In a state that $x \notin C$, every x represents a forest $x = \{t_0, t_1, \ldots t_n\}$, where each t_i is a tree. Let's consider the generation function according to the proposed general method of greedy algorithm, which in each

state produces a new forest by joining two subtrees together, and removing the original subtrees from the forest. For an arbitrary $y \in g(x)$ we have

$$y = x \cup \{\langle t_i, t_j \rangle\} - \{t_i, t_j\}$$

And therefore

$$\begin{aligned} cost(y) &= cost(x) + cost(\langle t_i, t_j \rangle) - cost(t_i) - cost(t_j) \\ &= \text{by (4.17)} \\ &\quad cost(x) + cost(t_i) + f(t_i) + cost(t_j) + f(t_j) - cost(t_i) - cost(t_j) \\ &= cost(x) + f(t_i) + f(t_j) \end{aligned} \tag{4.18}$$

The result in (4.18) explains the cost of every $y \in g(x)$ simply depends on $f(t_i) + f(t_j)$ because $cost(x)$ is a fixed amount. Therefore, to minimize the $cost(y)$ we only need to minimize the value of $f(t_i)+f(t_j)$ which can be satisfied by choosing two subtrees t_i and t_j with the lowest frequencies. Finally, the optimization technique of Huffman method, by using the two trees with the lowest cost, implements the greedy algorithm very efficiently. It considerably reduces the complexity of the algorithm.

4.3.1.3 Greedy Algorithm for Huffman Coding

Definition *HuffmanSearch* ::=

var $x := \emptyset$

for $c \in Ch$

$x := x \cup \langle c \rangle$

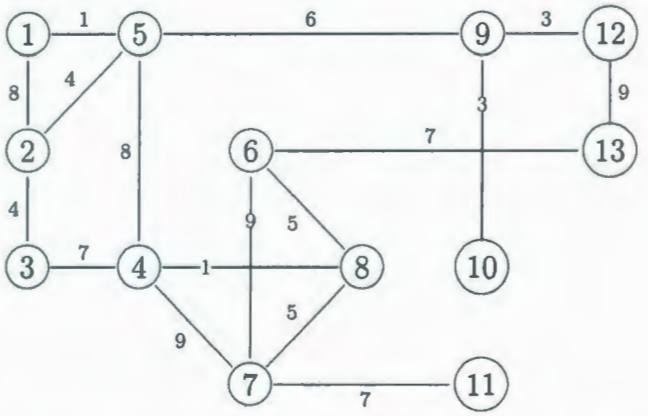
```
for k := 1 through |Ch| − 1   inv reaches(x, G) {
    var s, t
    s, t :=  the 2 trees in x of lowest f
    x := x ∪ {⟨s, t⟩} − {s, t}
}
```

4.3.2 Kruskal's Algorithm

For a connected edge weighted graph, a *minimum spanning tree* is a tree which has a minimum weight and spans the graph by connecting all the vertices. Kruskal's algorithm finds a minimum spanning tree in a connected undirected graph $G = (V, E)$ with the weight function $w : E \to \mathbb{R}$. For some connected graphs, minimum spanning trees are not unique and there are several qualifying feasible solutions which can be formed.

Kruskal's algorithm is a greedy algorithm which adds the best choice at a time to the partial solution. First, the edges are sorted in non-decreasing order according to their weight. Then, from the list of sorted edges, the minimum weight edge is selected and added to partial solution, if it does not form a cycle. In order to verify whether a new edge $e = \{u, v\} \in E$ can be added without creating a cycle, the two vertices $u \in V$ and $v \in V$ are verified not to be connected in the existing partial solution.

For the following connected graph:

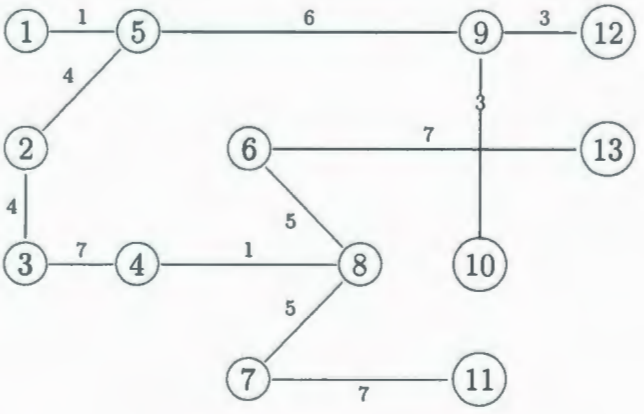


Kruskal's algorithm may create a minimum spanning tree by choosing edges in the following order:

$$(1,5) \rightarrow 1,\ (4,8) \rightarrow 1,\ (9,12) \rightarrow 3,\ (9,10) \rightarrow 3,\ (2,3) \rightarrow 4,\ (2,5) \rightarrow 4,$$

$$(6,8) \rightarrow 5,\ (7,8) \rightarrow 5,\ (5,8) \rightarrow 6,\ (3,4) \rightarrow 7,\ (6,13) \rightarrow 7,\ (7,11) \rightarrow 7.$$

The sequence of decisions results in the following minimum spanning tree:



4.3.2.1 Defining Parameters of the Problem Space and Greedy Solution

To specify the problem, it is required to define the elements of search problem:

- Define each member of S as a set of edges $e \in E$ of graph $G = (V, E)$ with no

cycles, $S \subseteq 2^E$

$$S = \{F \mid F \subseteq E \ \wedge \ F \text{ is } acyclic\}$$

Where

$$F \text{ is } acyclic \equiv \forall u, v \mid \{u, v\} \in F \ \cdot \ \neg(F - \{\{u, v\}\} \ connects \ (u, v)) \quad (4.19)$$

$$F \ connects \ (u, v) \equiv \exists k \mid \{u, k\} \in F \ \cdot \ k = v \ \vee \ F \ connects \ (k, v)$$

- Define i as the empty set, $i = \emptyset$
- Define each member of C as a set of edges that connect all vertices $v \in V$ with no cycles

$$C \ = \ \{F \in S \mid \forall u, v \in V \ \cdot \ F \ connects \ (u, v)\}$$

- Define g as a function which adds a new edge to a partial solution, without creating a cycle

$$g(x) \ = \ \{x \cup \{e\} \mid e \in E \ \cdot \ x \cup \{e\} \text{ is } acyclic\} \quad (4.20)$$

To verify whether a new edge $e = \{u, v\}$ creates a cycle, it is required to check if there is any connection between vertices u and v by any edge (set of edges) in x.

- Define $cost_{global}$ as the sum of the weights of a set of connected edges

$$cost_{global}(x) = cost(x) = \sum_{e \in x} w(e), \quad \text{where } x \in S$$

- Define $\leq_{global}$ as a cost comparison relation for two sets of connected edges

$$x \leq_{global} y \ \equiv \ \sum_{e \in x} w(e) \leq \sum_{e \in y} w(e), \quad \text{where } x, y \in S$$

To adapt the greedy solution, following parameters are specified:

- Define $cost_{local}$ as the sum of the weights of a set of edges

$$cost_{local}(x) = cost(x) = \sum_{e \in x} w(e), \quad \text{where } x \in S \tag{4.21}$$

- Define $\leq_{local}$ as a cost comparison relation for two sets of edges

$$x \leq_{local} y \equiv \sum_{e \in x} w(e) \leq \sum_{e \in y} w(e), \quad \text{where } x, y \in S$$

The definition of local and global cost for this example are identical and apply to a set of edges.

4.3.2.2 Implementation Process

- Initialization is $x = \emptyset$

- Defining the while guard: **while** $x \notin C$

To implement this part, consider the definition of C, which implies every $v \in V$ should be included in a completed solution. In addition, every possible pair of edges should be connected. Therefore, the partial solution x connects every possible pair of vertices in V.

$$\forall u, v \in V \cdot \ x \ connects \ (u, v)$$

Therefore, **while** $x \notin C$ is implemented as **while** $\exists u, v \in V \ \cdot \ \neg x \ connects \ (u, v)$.

- Generation function in $x :\in \min g(x)$

Generally, as defined above in (4.20), generation function creates a set of all possible edges which can join to the existing partial solution x, $x \cup \{e\}$, without

creating a cycle. Then, cost function (4.21) is applied to the set of created partial solutions. Finally, a minimum cost partial solution is selected.

However, Kruskal's algorithm suggests how this method can be optimized by adding an edge with a minimum weight to the partial solution if it does not create a cycle. In order to easily access an edge with a minimum weight, the set of edges can be sorted according to the weight. To prove this optimization method refines the general generation method, let's consider local cost function (4.21) for $y \in g(x)$ according to definition of (4.20).

$$\begin{aligned} \textsf{Let} \quad y &= x \cup \{e_{new}\} \in g(x) \\ cost(y) &= \sum_{e \in y} w(e) \\ &= \text{by def (4.20) and } y = x \cup \{e_{new}\} \\ &\quad \sum_{e \in x} w(e) + w(e_{new}) \\ &= w(x) + w(e_{new}) \end{aligned}$$

Which clearly proves that the cost of a new partial solution $y \in g(x)$ directly depends on the weight of a new edge added to the old partial solution x. Therefore, the optimization method of Kruskal implements the general generation function.

Considering Kruskal's greedy algorithm $x :\in \min g(x)$ is implemented by $x := x \cup \{e\}$ where e is a minimum weight edge such that $x \cup \{e\}$ is acyclic. The implementation of the optimized generation function selects an edge with the lowest weight and joins it to the existing partial solution, if it does not create a cycle. To verify whether it creates a cycle, the existing partial solution is

examined not to already connect the vertices of the new edge. This assures joining the new edge to the partial solution will not create a cycle. This implies to implement this section we are only required to verify whether partial solution x connects the vertices of a new edge, summarized as follows.

$$\begin{aligned} & \text{if } x \text{ is acyclic} \\ & \text{then } x \cup \{e\} \text{ is acyclic} \\ \equiv\ & \neg x \textit{ connects } e \end{aligned}$$

The implementation of generation function in $x :\in \min\ g(x)$ is summarized as follows:

$$\begin{aligned} & \text{let } e_{new} := \text{ an edge with the lowest } w \text{ such that } \neg x \textit{ connects } e_{new} \\ & x := x \cup \{e_{new}\} \end{aligned}$$

4.3.2.3 Greedy Algorithm for Kruskal's Method

Definition *KruskalSearch* ::=

var $x := \emptyset$

while $(\exists u, v \in V \cdot \neg x \textit{ connects } (u, v))$ inv $\textit{reaches}(x, G)\{$

let $e_{new} :=$ an edge with the lowest w such that$\neg x \textit{ connects } e_{new}$

$x := x \cup \{e_{new}\}$

}

Chapter 5

Conclusion

We investigated how abstract specifications can be proved to be implemented by abstract algorithms. For this study we considered algorithm design techniques such as dynamic programming and greedy algorithms. By applying a transformation that maps the abstract specification into a concrete specification, we showed how to derive a concrete algorithm from the abstract algorithm. With the derived method came along a formal proof of abstract algorithm correctness. This allows the abstract algorithm to be reused, along with its proof, to implement multiple concrete problems. The approach can be summarized as follows. Suppose we know that an abstract specification P is implemented by an abstract algorithm Q, then if we need an algorithm for a problem $R = T(P)$, where T is a data transform, we can implement R with $T(Q)$.

The study by Bird and de Moor uses categorical calculus as the mathematical framework. This framework helps formulating theorems and proofs. Generally, the framework and theories does not match up how computer programmers typically view

problems; therefore, the preliminary concepts were also presented. The style of reasoning with functions and relations is pointfree, which has the advantage of avoiding to formulate bound variables used by quantifiers and is described in terms of functional decomposition. However, we have experienced that the understanding of the theorems and proofs are always impossible without sketching and creating a supporting pointwise reasoning, which describes a function by its application to arguments. Optimal bracketing also known as matrix chain multiplication was described as an application of this case in chapter 2, the same example was later studied under our proposed dynamic programming algorithm in chapter 3. This provided the possibility of comparing two methods by considering the level of difficulty and complexity of the methods.

Sharon Curtis claims that although the theories by Bird and de Moor are useful to express considerable number of problems, it is not easily applicable to all optimization problems and not applicable to some exceptional cases at all. She takes Huffman Coding as an example, and concludes that catamorphism and anamorphism methods are top-down methods while examples such as Huffman Coding need bottom-up methods. The method developed by Curtis uses a limit operator $lim\ T$, a simple loop where lim recursively applies T to the input until it can not be reapplied. This method has a lower complexity comparing with catamorphisms and anamorphisms as presented by Bird and de Moor. However, Curtis uses the same framework of pointfree reasoning and her proposed algorithm for greedy algorithm, $lim\ T$, is yet complicated. Kruskal's method was described as the application of this method in chapter 2. Kruskal's method fits into the category of graphic matroids and fixed priority algorithms described in the introduction and is also later studied with our

proposed method in chapter 4 providing an easy comparison of complexity and efficiency of algorithm.

One of the algorithm design approaches studied in this thesis is dynamic programming, which is presented in Chapter 3. An abstract algorithm for dynamic programming has been formally developed for an abstract specification. This specification includes slots for the problem space, solution space, and a function mapping them. This abstract algorithm is presented in top-down and bottom-up approaches. Principle elements of dynamic programming in the proposed method includes leaf problems, branch problems, divide function, and conquer function. Leaf problems are easy to solve and branch problems are solved by using functions divide and conquer. If these two functions can be defined for a problem, dynamic programming can be applicable. The main theory of dynamic programming is proved to be correct by method of induction. Application of dynamic programming such as Matrix chain multiplication and Largest black square represents how this abstract algorithm can be implemented in concrete algorithms.

The other algorithm design approach studied in this thesis is greedy algorithm, which is presented in chapter 4. The parameters required to create a greedy algorithm includes definition of completed solution, a global cost function, a local cost function, and a generation function. The global cost function measures the degree of optimality in a feasible solution, and the local cost function is used in each step of the algorithm for comparing partial solutions. There is a relation between local and global cost functions, in most cases being identical. However, it is not required for greedy algorithm to be defined having equal local and global cost functions. The generation function creates all the possible next greedy steps and the best is selected

with respect to its cost and adds it to the partial solution. If for any problem an optimization method is used, it is required to show how it will refine the general proposed method. As mentioned earlier in the introduction, there are many problems with different structures which can benefit from greedy algorithm. However, in this thesis one case has been proved to qualify for this method. This case includes problems with the monotonicity condition, referred as better-local by Curtis. The main theory of greedy algorithms is proved under certain conditions, such as the property of monotonicity and the assumption that continuing the greedy step after a complete solution is created will not result in a better completed solution, since the generation function stops running when a completed solution is created.

In addition to independent examples, choosing the same examples in the literature and main body of the thesis aims to describe how the proposed methods provide more applicable methods for computer scientists by resolving some inadequacies of the other introduced methods such as complexity.

Finally, future work can include implementing the proposed techniques in SIMPLE to refine abstract specification by abstract dynamic programming algorithm, and abstract greedy algorithms. It can also provide more details on the conditions under which the proposed methods are applicable to particular problems. It can further define conditions required on divide and conquer functions, and proving the correctness of greedy algorithm in other possible cases. The proposed methods can be studied on more examples on different categories of problems such as hard to solve problems. Future work can also include complexity study of algorithms and methods of further optimizations.

Bibliography

[1] R. Bellman. *Dynamic Programming.* Princeton University Press, Princeton, 1957.

[2] R. Bird and O. de Moor. *Algebra of Programming.* Prentice Hall, London, 1997.

[3] R. S. Bird and O. de Moor. From dynamic programming to greedy algorithms. In B. Moller, H. Partsch, and S. Schuman, editors, *Formal Program Development, Volume 755 of Lecture Notes in Computer Science*, pages 43–61, Berlin, 1993. Springer-Verlag.

[4] R. S. Bird and O. de Moor. Solving optimisation problems with catamorphisms. In R. S. Bird, C. C. Morgan, and J. C. P. Woodcock, editors, *2nd International Conference on the Mathematics of Program Construction, Volume 669 of Lecture Notes in Computer Science*, pages 45–66. Springer-Verlag, 1993.

[5] A. Borodin, M. N. Nielsen, and C. Rackoff. (Incremental) priority algorithms. In *Proceedings of the Thirteenth ACM-SIAM Symposium on Discrete Algorithms*, page 1996, 2002.

[6] T. H. Cormen, C. E. Leiserson, and R. L. Rivest. *Introduction to Algorithms.* McGraw-Hill, New York, 2nd edition, 2001.

[7] S. Curtis. Dynamic programming: A different perspective. In R. Bird and L. Meertens, editors, *Algorithmic Languages and Calculi*, pages 1–23, London, 1997. Chapman & Hall.

[8] S. Curtis. Use of relational operators for algorithm development. cms.brookes.ac.uk/staff/SharonCurtis/publications/lim.ps.gz, 1999.

[9] S. A. Curtis. *A Relational Approach to Optimization Problems*. PhD thesis, University of Oxford, Oxford, U.K., April 1996.

[10] S. A. Curtis. The classification of greedy algorithms. *Science of Computer Programming*, 49:125–157, 2003.

[11] E. W. Dijkstra. Guarded commands, nondeterminancy and formal derivation of programs. *Communications of ACM*, 18:453–457, 1975.

[12] J. Edmonds. Matroids and greedy algorithms. *Mathematical Programming*, 1:126–136, 1971.

[13] J. Edmonds. How to think about algorithms. loop invariants and recursion. http://www.cse.yorku.ca/~jeff/notes/3101/TheNotes.pdf, Version 0.12, January 2007.

[14] R. W. Floyd. Assigning meanings to programs. In *Proceeding of Symposium in Applied Mathematics and Mathematical Aspect of Computer Science*, pages 19–32, 1967.

[15] D. Gries. *The Science of Programming*. Springer-Verlag, New York, 1981.

[16] E. C. R. Hehner. Predicative programming part I. *Communications of the ACM*, 27:134–143, 1984.

[17] E. C. R. Hehner. Predicative programming part II. *Communications of the ACM*, 27:144–151, 1984.

[18] E. C. R. Hehner. *A Practical Theory of Programming*. Springer-Verlag, New York, 1993.

[19] J. L. Hein. *Discrete Structures, Logic, and Computability*. Jones and Bartlett Publishers, Boston, 2nd edition, 2002.

[20] C. A. R. Hoare. An axiomatic basis for computer programming. In C. A. R. Hoare and C. B. Jones, editors, *Essays in Computing Science*, pages 45–58, New York, 1989. Prentice Hall.

[21] C. A. R. Hoare. Notes on an approach to category theory for computer scientists. In M. Broy, editor, *Constructive Methods in Computing Science, NATO Advanced Science Institute Series (Series F: Computer and System Sciences)*, volume 55, pages 245–305. Springer Verlag, 1989.

[22] C. A. R. Hoare, I. J. Hayes, H. Jifeng, C. C. Morgan, A. W. Roscoe, J. W. Sanders, I. H. Sorensen, J. M. Spivey, and B. A. Sufrin. Laws of programming. *Communications of ACM*, 30:672–686, 1987.

[23] D. S. Hochbaum. *Approximation Algorithms for NP-hard Problems*. PWS Publishing Company, 1995.

[24] B. Korte and L. Lovasz. Mathematical structures underlying greedy algorithms. In *Fundamentals of Computation Theory, Volume 117 of Lecture Notes in Computer Science*, pages 205–209, Berlin, 1981. Spring-Verlag.

[25] B. Korte and L. Lovasz. Greedoids and linear objective functions. *SIAM Journal on Algebraic and Discrete Methods*, 5:229–238, 1984.

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

[27] A. Lew and H. Mauch. *Dynamic Programming: A computational Tool*. Springer, Berlin, 2007.

[28] J. McCarthy. Towards mathematical science of computation. www-formal.stanford.edu/jmc/towards.ps, 1996.

[29] T. S. Norvell. The SIMPLE Report. Draft, Memorial University, St. John's, NL, Canada, 2004 (Unpublished).

[30] T. S. Norvell. Faster search by elimination. In *Newfoundland Electrical and Computer Engineering Conference*, November 2005.

[31] T. S. Norvell and Z. Ding. An environment for proving and programming. In *Newfoundland Electrical and Computer Engineering Conference*, October 1999.

[32] D. Pavlovic and D. R. Smith. Software development by refinement. In B. K. Aichernig and T. S. E. Maibaum, editors, *UNU/IIST 10th Anniversary Colloquium, Formal Methods at the Crossroads: From Panaea to Foundational Support, Volume 2757 of Lecture Notes in Computer Science*, pages 267–286. Springer, 2003.

[33] B. C. Pierce. *Basic Category Theory for Computer Scientists*. The MIT Press, Cambridge, Mass, 1991.

[34] J. Sgall. On-line scheduling. In A. Fiat and G. J. Woeginger, editors, *Online Algorithms: The State of the Art*, pages 196–231. Springer, 1998.

[35] D. Smith. Mechanizing the development of software. In M. Broy and R. Steinbrueggen, editors, *Calculational System Design, Proceedings of the NATO Advanced Study Institute*, pages 251–292. IOS Press, 1999.

[36] D. R. Smith. KIDS: A semi-automatic program development system. *IEEE transactions on Software Engineering Special Issue on Formal Methods*, 16:1024–1043, 1990.

[37] D. R. Smith. KIDS: A knowledge-based software development system. In M. R. Lowry and R. D. McCartney, editors, *Automating Software Design*, pages 483–514, Menlo Park, California, 1991. AAAI Press.

[38] D. R. Smith. Automating the design of algorithms. *Lecture Notes In Computer Science*, 755:324–354, 1993.

[39] J. Ward. A unified model of algorithm design. Master's thesis, Dept. of Computer Science, University of Toronto, Toronto, ON, Canada, 2007.