MATHEMATICAL ENGINEERING
TECHNICAL REPORTS

Calculus of Minimals: Deriving Dynamic-Programming Algorithms based on Preservation of Monotonicity

Akimasa MORIHATA and Kiminori MATSUZAKI and
Zhenjiang HU and Masato TAKEICHI

METR 2007–61 December 2007

Calculus of Minimals: Deriving Dynamic-Programming Algorithms based on Preservation of Monotonicity

Akimasa Morihata   Kiminori Matsuzaki   Zhenjiang Hu   Masato Takeichi

{morihata,kmatsu}@ipl.t.u-tokyo.ac.jp
{hu,takeichi}@mist.i.u-tokyo.ac.jp

The University of Tokyo

Abstract

Constructing efficient algorithms is difficult and often considered to be a privilege of a few specialists. *Program calculation* is a methodology for easy construction of efficient algorithms, where efficient algorithms are systematically derived from naive but obviously correct algorithms by calculational laws. This paper shows an ongoing effort to give a clear and effective methodology to deal with combinatorial optimization problems based on program calculation. First, we formalize greedy algorithms and dynamic programming algorithms in terms of minimals and the strictly monotone property. Next, we propose a new calculational law to derive dynamic programming algorithms, which builds on some calculational laws to construct orders satisfying monotone properties. Our law is not only applicable for a wide class of combinatorial optimization problems, but also suitable for automatic implementation. Finally, as a nontrivial application of our calculational laws, we show systematic derivations of algorithms for regular-language constrained shortest path problems.

1 Introduction

Constructing efficient algorithms is a laborious task. We need to investigate specific properties of the problem in detail, use complicated and tricky procedure to avoid unnecessary or inefficient computations, and accomplish probably painful proofs for its correctness with incomprehensible mathematics. These processes would be too difficult for nonspecialists.

Program calculation [BdM96] (or *calculational programming*) is a methodology for easy construction of efficient algorithms. The point of program calculation is to divide an algorithm construction process into two steps. We first give an obviously correct algorithm with no attention to its efficiency; after that, we improve its efficiency by applying *calculational laws*, which are mathematically correct program transformation rules. Program calculation has many advantages over the usual way. First, we do not need to worry about the correctness of the algorithm constructed. Because the calculational laws are correct, the algorithm constructed is proved to be correct by its construction. Second, we can understand the key insight of the algorithm from its construction. Since similar derivations for similar problems yield similar algorithms, we can characterize algorithms by their derivation processes. Third, program calculation is suitable for automatic implementation. Since what we manipulate are programs, we may achieve automatic improvement of programs by automating program transformations. Therefore, program calculation is helpful to construct efficient algorithms for everyone: not only specialists but also nonspecialists or even computers. Many researches about program calculation have been done, such as formalizing calculational laws [Bir84, BdM93a, BdM93b, dM95, Cur96, SHTO00, Cur03, MKHT06], deriving nontrivial algorithms [Bir89, Rav99b, dMG99, dMG00, Bir01, Bir06], and automating calculations [dMS01, SdM01, Yok06].

Our objective is to give a clear and effective methodology to deal with *combinatorial optimization problems* based on program calculation. Combinatorial optimization problems are optimization problems where a solution is constructed by a sequence of discrete decisions. Since combinatorial optimization problems have a great many applications, they are recognized to be one of the most important classes of problems in algorithm construction.

Although many endeavors have been made, it is still nontrivial to deal with combinatorial optimization problems calculationally. On one hand, several researches have been devoted to formalizing calculational laws for combinatorial optimization problems [BdM93b, BdM93a, dM95, BdM96, Cur96, Cur03]. Though they give a clear characterization for a wide class of combinatorial optimization problems, they are not suitable for automatic implementation. The calculational laws require an appropriate order that satisfies the monotonicity condition on candidates; however, such appropriate order may not be apparent from the specification of the problem. On the other hand, some classes of combinatorial optimization problems have been identified to be solvable automatically [ALS91, BPT92, SHTO00]. They are interesting, but maximum marking problems on a tree are the only things that they can cope with. To see the hardship, let us consider the *regular-language constrained shortest path problem* [Rom88, BJM00], which we will solve calculationally in Section 6. Given an edge-weighted edge-labeled graph and a regular language, a regular-language constrained shortest path problem is the problem to find the shortest path from the source to the destination such that the label of the path is in the regular language. It is difficult to solve regular-language constrained shortest path problems based on existing calculational laws. While the only order found in the specification is the order to compare weights of paths, it does not satisfy the monotonicity condition. Since regular-language constrained shortest path problems are maximum marking problems on a graph, not on a tree, the works given above cannot deal with them.

Our objective is to give a calculational framework that is easy to use and can cope with a large class of combinatorial optimization problems. Our main contributions are summarized as follows:

- We formalize a calculational framework for *minimals*, while existing calculational laws are based on minimums. Minimals satisfy good properties that minimums do not have. We show minimals are useful for calculation by giving calculational laws for them.
- We propose a new calculational law to derive dynamic programming algorithms for combinatorial optimization problems. Our law is applicable for a wide class of combinatorial optimization problems, including the regular-language shortest path problems. Moreover, our law is suitable for automatic implementation.
- We show systematic derivations of algorithms for regular-language constrained shortest path problems, which indicate the promise of our calculational laws.

The rest of this paper is organized as follows. In Section 2, we fix the basis of our formalization and show some basic results. In Section 3, we review existing works and discuss their strengths and drawbacks. In Section 4, we formalize a calculational framework for minimals. In Section 5, we propose our new calculational laws and discuss their properties. In Section 6, we explain how our calculational laws work throughout the derivations of efficient algorithms to solve regular-language constrained shortest path problems. We discuss related works in Section 7 and conclude this paper in Section 8.

2 Program Calculation

In this section, we explain the definitions and notations used throughout this paper. We basically follow Bird and de Moor [BdM96].

2.1 Basic Notations

Functions A function f that maps each element of a set A to an element of a set B is denoted by $f : A \to B$. The identity function on a set A is denoted by $id_A : A \to A$, i.e., $id_A(x) \stackrel{\text{def}}{=} x$ for all $x \in A$. The subscript may be omitted. The composition of functions is denoted by an operator $\circ$ and its definition is $(f \circ g)(x) \stackrel{\text{def}}{=} f(g(x))$. To denote repeated compositions of the same function, we borrow the power notation, i.e., $f^0 \stackrel{\text{def}}{=} id$ and $f^n \stackrel{\text{def}}{=} f \circ f^{n-1}$ if $1 \leq n \in \mathbb{N}$. Parentheses to denote function application may be omitted.

Tuples We use a pair of parentheses split by commas to denote a tuple, i.e., $(x, 0)$ means a tuple of a variable x and an integer 0. The projection of ith element is denoted by π_i, i.e., $\pi_i(a_1, \ldots, a_i, \ldots, a_k) \stackrel{\text{def}}{=} a_i$. A pair of angle brackets split by a comma, namely $\langle\ ,\ \rangle$, is used to construct pairs, and its definition is $\langle f, g\rangle(x) \stackrel{\text{def}}{=} (f(x), g(x))$.

Sequences We use a pair of brackets split by commas to denote a sequence. A set of sequences that consist in values of type A is denoted by A^*. The empty sequence is denoted by $[]$. The concatenation of two sequences is denoted by $+\!\!+$, i.e., $[x_0, \ldots, x_n] +\!\!+ [y_0, \ldots, y_m] = [x_0, \ldots, x_n, y_0, \ldots, y_m]$.

Sets We use a pair of curly brackets to denote a set. The empty set is denoted by $\emptyset$. A set of all subset of A is denoted by 2^A. Basic operators for sets, such as $\cup$, $\cap$, $\setminus$, and $\times$, are defined as usual. The size of a set S is denoted by $|S|$, or just S if it is not ambiguous. The operator $\uplus$ is used to merge the results of two functions, and its definition is $(f \uplus g)(x) \overset{\text{def}}{=} f(x) \cup g(x)$. For a predicate $p : A \to \mathit{Bool}$, the filtering function raised by p is denoted by $p\triangleleft : 2^A \to 2^A$, i.e., $p\triangleleft(X) \overset{\text{def}}{=} \{a \mid a \in X \wedge p(a)\}$.

Graphs We assume that readers know basic notions of graphs and basic algorithms for the shortest path problems. Refer textbooks such as [CSRL01, KT05] if necessary. A graph $G = (V, E)$ is a pair of a set of vertexes V and a set of edges E. A vertex is denoted by a pair of edges, thus $V \subseteq E \times E$. We express a path as a sequence of edges. Given a weight function $w : E \to \mathbb{R}$, an edge-weighted graph $N = (G, w)$ is called a *network*. Given an alphabet Σ, a graph $G = (V, E)$ is said to be labeled by Σ when a labeling function $l : E \to \Sigma$ is given. We use both labeling function l and weight function w to compute label and weight of a path respectively, as usual. We assume that there is no cycle whose weight is negative.

2.2 Functors

A category consists of a set of objects and a set of morphisms. Here we consider the category whose objects are sets and morphisms are total functions. A functor is a morphism of categories. For two categories $\mathcal{A}$ and $\mathcal{B}$, a functor $\mathsf{F} : \mathcal{A} \to \mathcal{B}$ maps each object $A \in \mathcal{A}$ to $\mathsf{F}A \in \mathcal{B}$, and each morphism $f \in \mathcal{A}$ to $\mathsf{F}f \in \mathcal{B}$, with satisfying the following properties.

$$\mathsf{F}(id_{\mathcal{A}}) = id_{\mathcal{B}}$$
$$\mathsf{F}(f \circ g) = \mathsf{F}f \circ \mathsf{F}g$$

An important functor is the power-set functor P, where $\mathsf{P}(A) \overset{\text{def}}{=} 2^A$ and $\mathsf{P}f(X) \overset{\text{def}}{=} \{f(a) \mid a \in X\}$.

Another important class of functors is polynomial functors. A functor is said to be polynomial if it is constructed by the combinations of the identity functor I, the constant functor $!\mathsf{B}$ where B is a parameter, the product bifunctor $\times$, and the coproduct bifunctor $+$. The definition is the following, in which F and G denote functors, A and B denote objects, and f denotes a morphism of appropriate type.

$$\begin{aligned}
\mathsf{I}A &\overset{\text{def}}{=} A \\
\mathsf{I}f &\overset{\text{def}}{=} f \\
!\mathsf{B}A &\overset{\text{def}}{=} B \\
!\mathsf{B}f &\overset{\text{def}}{=} id_B \\
(\mathsf{F} \times \mathsf{G})A &\overset{\text{def}}{=} (\mathsf{F}A \times \mathsf{G}A) \\
(\mathsf{F} \times \mathsf{G})f(x, y) &\overset{\text{def}}{=} (\mathsf{F}f(x), \mathsf{G}f(y)) \\
(\mathsf{F} + \mathsf{G})A &\overset{\text{def}}{=} (\{1\} \times \mathsf{F}A) \cup (\{2\} \times \mathsf{G}A) \\
(\mathsf{F} + \mathsf{G})f(x) &\overset{\text{def}}{=} \begin{cases} (1, \mathsf{F}f(a)) & \text{if } (1, a) = x \\ (2, \mathsf{G}f(b)) & \text{if } (2, b) = x \end{cases}
\end{aligned}$$

It is well-known that polynomial functors give a good characteristics of algebraic data types [MFP91, Fok92, Mei92].

2.3 Recursion Schema

In program calculation, combinatorial optimization problems have been formalized in two forms: the catamorphisms [BdM93b, BdM96, SHTO00] and the repetition operator [Cur96, Cur03]. First we introduce the notion of catamorphisms.

Definition 2.1 (algebra)**.** *For a functor* F*, an* F-algebra *is a pair* (A, ψ)*, where* A *is a set and* $\psi : \mathsf{F}A \to A$ *is a function.* □

Definition 2.2 (algebra morphism)**.** *For two* F*-algebras* $\mathcal{A} = (A, \psi)$ *and* $\mathcal{B} = (B, \phi)$*, an algebra morphism from* $\mathcal{A}$ *to* $\mathcal{B}$ *is a function* $h : A \to B$ *that satisfies the following equation.*

$$h \circ \psi = \phi \circ \mathsf{F}h \qquad \square$$

Definition 2.3 (initial algebra and catamorphism)**.** *An* F*-algebra* $(\mu\mathsf{F}, \mathsf{in}_\mathsf{F})$ *is said to be* initial *if for all* F*-algebra* (B, ϕ) *there exists a unique algebra morphism from* $(\mu\mathsf{F}, \mathsf{in}_\mathsf{F})$ *to* (B, ϕ)*. The unique morphism is called* catamorphism *and denoted by* $(\![\phi]\!)_\mathsf{F}$*.* $\square$

The catamorphism $(\![\phi]\!)_\mathsf{F}$ is well-defined because the initial F-algebra is unique up to isomorphism. We may omit the subscript for catamorphisms if it is clear from the context.

Catamorphisms have a fusion law.

Theorem 2.4 (cata fusion [MFP91])**.**

$$(f \circ \phi = \psi \circ \mathsf{F}f) \Rightarrow (f \circ (\![\phi]\!)_\mathsf{F} = (\![\psi]\!)_\mathsf{F}) \qquad \square$$

Catamorphisms are known to be a general recursion schema. Almost all functions that iterate over a tree-like structure can be recognized in terms of catamorphisms. However, catamorphisms hardly capture the functions that traverse on graphs. Thus, we introduce the notion of repetition, where a computation is repeated until no more computations are necessary. A comparison of catamorphisms and repetitions is found in [Cur96].

Definition 2.5 (repetition)**.** *For a function* $f : A \to A$*, the repetition of* f*, denoted by* f^**, is defined as follows.*

$$f^*(a) \overset{\text{def}}{=} f^n(a) \quad \text{if } f^n(a) = f^{n+1}(a) \qquad \square$$

Repetitions have a promotion law.

Theorem 2.6 (repetition promotion)**.**

$$(f \circ \phi = \psi \circ f) \Rightarrow (f \circ \phi^* = \psi^* \circ f)$$

Proof. From the definition of the repetition operator, it is sufficient to show $\forall n \in \mathbb{N} : f \circ \phi^n = \psi^n \circ f$ holds. The claim is proved by induction. The claim obviously holds for the base case, namely the case of $n = 0$. The step case is proved by the following calculation.

$$\begin{array}{ll}
f \circ \phi^{k+1} & = \quad \{ \text{ definition of the power notation } \} \\
& \quad\; f \circ \phi \circ \phi^k \\
& = \quad \{ \text{ assumption } \} \\
& \quad\; \psi \circ f \circ \phi^k \\
& = \quad \{ \text{ induction } \} \\
& \quad\; \psi \circ \psi^k \circ f \\
& = \quad \{ \text{ definition of the power notation } \} \\
& \quad\; \psi^{k+1} \circ f
\end{array} \qquad \square$$

2.4 Relational Calculus

Relations are useful to express optimization problems, because orders or nondeterministic choices are easily described in terms of relations. Here we introduce the notion of relations and relators with some useful operators. We give set-theoretic definitions with axiomatic characterizations for many operators. The former is easy to understand the meaning, while the latter is useful to give formal calculational proofs in abstract settings.

In this paper, a relation is a set of pairs. We basically distinguish relations from sets of pairs because we use them in different context. Functions are also relations, but again, we basically distinguish functions from relations. We denote a relation R between elements of set A and elements of set B, namely $R \subseteq A \times B$, as $R : B \leadsto A$. We may denote $a\,R\,b$ instead of $(a, b) \in R$.

We use the operator $\circ$ to denote the composition of relations, as the same as for functions, and it is defined as follows:

$$a\,(R \circ S)\,b \overset{\text{def}}{=} \exists c :\ a\,R\,c \land c\,S\,b$$

Note that this definition coincides with that of function compositions if both R and S are functions.

The converse of a relation R is denoted by R°, and its definition is $a\, R^\circ\, b \stackrel{\text{def}}{=} b\, R\, a$. It is known [BdM96] that relators respect converses, namely $\mathsf{F}(R^\circ) = (\mathsf{F}R)^\circ$.

To introduce relations to categorically settings, we will consider the category of relations, where an objects is a set and a morphism is a relation of appropriate type. Not all functors in the category are useful for calculations, and thus we introduce the notion of *relators*.

Definition 2.7 (relator)**.** *A functor* F *is said to be a* relator *if it respects inclusions, that is,* $R \subseteq S$ *implies* $\mathsf{F}R \subseteq \mathsf{F}S$ *for any relations* R *and* S.

From now on, we only consider relators as functors and use sanserif characters to denote them. Actually almost all useful functors in computer science are relators. For example, the powerset functor is a relator where relations are mapped as $X\ (\mathsf{P}R)\ Y \stackrel{\text{def}}{=} (\forall a \in X : (\exists b \in Y : a\, R\, b)) \wedge (\forall b \in Y : (\exists a \in X : a\, R\, b))$. Polynomial functors are also relators, and its mapping on relations is defined as follows.

$$\begin{aligned} \mathsf{I}R &\stackrel{\text{def}}{=} R \\ !\mathsf{A}R &\stackrel{\text{def}}{=} id_A \\ (u,w)\ ((\mathsf{F}\times\mathsf{G})R)\ (v,x) &\stackrel{\text{def}}{=} u\ (\mathsf{F}R)\ v \wedge w\ (\mathsf{G}R)\ x \\ u\ ((\mathsf{F}+\mathsf{G})R)\ v &\stackrel{\text{def}}{=} \begin{cases} u'\ (\mathsf{F}R)\ v' & \text{if } (1,u') = u \text{ and } (1,v') = v \\ u'\ (\mathsf{G}R)\ v' & \text{if } (2,u') = u \text{ and } (2,v') = v \end{cases} \end{aligned}$$

An operator Λ, called power transpose, takes a relation and returns the set-valued function that enumerates all element related. Its formal definition is the following.

$$\Lambda R(b) \stackrel{\text{def}}{=} \{a \mid a\, R\, b\}$$

The power operator is characterized by the following equation.

$$\Lambda S = T \Leftrightarrow S = \in \circ\, T$$

The following lemma shows the relationship between functors and power transpose.

Lemma 2.8. *For any relator* F *and relation* S, $\Lambda\mathsf{F}S = \Lambda\mathsf{F}{\in} \circ \mathsf{F}\Lambda S$ *holds.*

Proof.

$$\begin{aligned} \Lambda\mathsf{F}S = \Lambda\mathsf{F}{\in} \circ \mathsf{F}\Lambda S \quad &\Leftrightarrow \quad \{\text{ property of } \Lambda \text{ }\} \\ &\qquad \mathsf{F}S = \in \circ\, \Lambda\mathsf{F}{\in} \circ \mathsf{F}\Lambda S \\ &\Leftrightarrow \quad \{ \in \text{ cancels out } \Lambda \text{ }\} \\ &\qquad \mathsf{F}S = \mathsf{F}{\in} \circ \mathsf{F}\Lambda S \\ &\Leftrightarrow \quad \{\text{ } \mathsf{F} \text{ is functor }\} \\ &\qquad \mathsf{F}S = \mathsf{F}(\in \circ\, \Lambda S) \\ &\Leftrightarrow \quad \{ \in \text{ cancels out } \Lambda \text{ }\} \\ &\qquad \mathsf{F}S = \mathsf{F}S \end{aligned}$$

□

An operator $\cap$ is used to express "conjunction" of two relations. The following is the set-theoretic definition of $\cap$. Notice that the operator $\cap$ for relations is exactly the usual intersection operator of sets when we express relations as sets of pairs.

$$a\ (R \cap S)\ b \stackrel{\text{def}}{=} a\, R\, b \wedge a\, S\, b$$

The operator $\cap$ can be characterized by the following property.

$$(R \cap S) \supseteq X \Leftrightarrow (R \supseteq X) \wedge (S \supseteq X)$$

In general, relators do not distribute over $\cap$. For example, the powerset functor P does not distribute over $\cap$. However, polynomial functors distribute over $\cap$.

Lemma 2.9. *For any relator* F *and relations* R *and* S, $\mathsf{F}(R \cap S) \subseteq \mathsf{F}R \cap \mathsf{F}S$.

Proof.

$$\begin{array}{rcl}
\mathsf{F}(R\cap S)\subseteq(\mathsf{F}R\cap\mathsf{F}S) & \Leftrightarrow & \{\text{ property of }\cap\ \}\\
 & & (\mathsf{F}(R\cap S)\subseteq\mathsf{F}R)\wedge(\mathsf{F}(R\cap S)\subseteq\mathsf{F}S)\\
 & \Leftarrow & \{\text{ relator }\}\\
 & & ((R\cap S)\subseteq R)\wedge((R\cap S)\subseteq S)\\
 & \Leftrightarrow & \{\text{ property of }\cap\ \}\\
 & & \text{True}
\end{array}$$

□

Lemma 2.10. *For any polynomial functor* F *and relations* R *and* S*,* $\mathsf{F}(R\cap S)=\mathsf{F}R\cap\mathsf{F}S$*.*

Proof. It is a direct consequence of Propositions 6.3.10 and 5.3.9 in [dM92]. □

Similar to $\cap$, we can define an operator $\cup$ as follows.

$$a\,(R\cup S)\,b\overset{\text{def}}{=}a\,R\,b\vee a\,S\,b$$

The operator $\cup$ is characterized by the following property.

$$(R\cup S)\subseteq X\Leftrightarrow(R\subseteq X)\wedge(S\subseteq X)$$

Different from $\cap$, relators do not distribute over $\cup$, even if they are polynomial.

Lemma 2.11. *For any relator* F *and relations* R *and* S*,* $\mathsf{F}R\cup\mathsf{F}S\subseteq\mathsf{F}(R\cup S)$ *holds.*

Proof.

$$\begin{array}{rcl}
(\mathsf{F}R\cup\mathsf{F}S)\subseteq\mathsf{F}(R\cup S) & \Leftrightarrow & \{\text{ property of }\cup\ \}\\
 & & (\mathsf{F}R\subseteq\mathsf{F}(R\cup S))\wedge(\mathsf{F}S\subseteq\mathsf{F}(R\cup S))\\
 & \Leftarrow & \{\text{ relator }\}\\
 & & (R\subseteq(R\cup S))\wedge(S\subseteq(R\cup S))\\
 & \Leftrightarrow & \{\text{ property of }\cup\ \}\\
 & & \text{True}
\end{array}$$

□

The operator $\Rightarrow$, corresponding to the logical implication, is defined as follows.

$$a\,(R\Rightarrow S)\,b\overset{\text{def}}{=}\neg(a\,R\,b)\vee a\,S\,b$$

Its characteristic property is the following.

$$(R\Rightarrow S)\supseteq X\Leftrightarrow S\supseteq(R\cap X)$$

Nothing interesting is known about relationship between relators and the $\Rightarrow$ operator. But, for polynomial functors, we have the following lemma.

Lemma 2.12. *For any polynomial functor* F *and relations* R *and* S*,* $\mathsf{F}(R\Rightarrow S)\subseteq\mathsf{F}R\Rightarrow\mathsf{F}S$*.*

Proof.

$$\begin{array}{rcl}
\mathsf{F}(R\Rightarrow S)\subseteq(\mathsf{F}R\Rightarrow\mathsf{F}S) & \Leftrightarrow & \{\text{ property of }\Rightarrow\ \}\\
 & & (\mathsf{F}(R\Rightarrow S)\cap\mathsf{F}R)\subseteq\mathsf{F}S\\
 & \Leftrightarrow & \{\text{ Lemma 2.10 }\}\\
 & & \mathsf{F}((R\Rightarrow S)\cap R)\subseteq\mathsf{F}S\\
 & \Leftarrow & \{\text{ relator }\}\\
 & & ((R\Rightarrow S)\cap R)\subseteq S\\
 & \Leftrightarrow & \{\text{ property of }\Rightarrow\ \}\\
 & & \text{True}
\end{array}$$

□

We use negation operator $\neg$, whose definition is $\neg R \overset{\text{def}}{=} R \Rightarrow \emptyset$. In our set theoretic setting, the operators introduced satisfy "usual" laws that logical operators satisfy, such as the double negation elimination law or the De Morgan law.

We will use the operator /, which is used to introduce "for all" quantification. Its definition is the following.

$$a\,(R/S)\,b \overset{\text{def}}{=} \forall c:\; b\,S\,c \Rightarrow a\,R\,c$$

Its axiomatic definition is the following.

$$R/S \supseteq X \overset{\text{def}}{=} R \supseteq (X \circ S)$$

From the definition, / operator is anti-monotonic to the right operand. Following lemmas show its relationship to relators.

Lemma 2.13. *For any relator* F *and relations* R *and* S, $\mathsf{F}(R/S) \subseteq \mathsf{F}R/\mathsf{F}S$.

Proof.

$$\begin{array}{rcl} \mathsf{F}(R/S) \subseteq \mathsf{F}R/\mathsf{F}S & \Leftrightarrow & \{\text{ property of / }\} \\ & & (\mathsf{F}(R/S) \circ \mathsf{F}S) \subseteq \mathsf{F}R \\ & \Leftarrow & \{\text{ relator }\} \\ & & ((R/S) \circ S) \subseteq R \\ & \Leftrightarrow & \{\text{ property of / }\} \\ & & \text{True} \end{array}$$

□

Lemma 2.14 (Lemma 8.3.1.2 in [dM92]). *For any polynomial functor* F *and relations* R *and* S, $\mathsf{F}((R/S) \cap S^\circ) = (\mathsf{F}R/\mathsf{F}S) \cap \mathsf{F}S^\circ$. □

For functions, some useful properties are known. We will use the following properties, where R and S are relations and f is a total function.

$$\Lambda(S \circ f) = \Lambda S \circ f \tag{1}$$

$$(f \circ R \subseteq S) \Leftrightarrow (R \subseteq f^\circ \circ S) \tag{2}$$

$$(R \circ f^\circ \subseteq S) \Leftrightarrow (R \subseteq S \circ f) \tag{3}$$

2.5 Orders and Minimization

We use *quasi-order* (also called *preorder*) to describe optimization problems.

Definition 2.15 (quasi-order). *A relation* $R : A \rightsquigarrow A$ *is called* quasi-order *if the following two properties are satisfied.*

$$\begin{array}{ll} \forall a \in A:\; a\,R\,a & \textit{(reflectivity)} \\ (a\,R\,b \wedge b\,R\,c) \Rightarrow a\,R\,c & \textit{(transitivity)} \end{array}$$

□

Definition 2.16 (totality). *A relation* $R : A \rightsquigarrow A$ *is said to be* total *if for all* $a \in A$ *and* $b \in A$, $a\,R\,b \vee b\,R\,a$ *is satisfied.* □

Equivalence relations and linear orders[1] are ones of the most well-known and important classes of quasi-orders.

Definition 2.17 (equivalence relation). *A quasi-order* $R : A \rightsquigarrow A$ *is called* equivalence relation *if the following property is satisfied.*

$$a\,R\,b \Rightarrow b\,R\,a \quad \textit{(symmetry)}$$

□

Definition 2.18 (linear order). *A total quasi-order* $R : A \rightsquigarrow A$ *is called* linear order *if the following property is satisfied.*

[1]Linear orders are also called *total orders*. We do not use the name to avoid the confusion between total orders and total quasi-orders.

$$(a\,R\,b \wedge b\,R\,a) \Rightarrow a = b \quad \textit{(antisymmetry)}$$ □

Here we would like to define some notations. We make a special use of $=$ and $<$. For a quasi-order R, $\overset{R}{=}$ denotes the equivalent part of R, i.e., $a \overset{R}{=} b \overset{\text{def}}{=} aRb \wedge bRa$. Similarly, for a quasi-order R, $\overset{R}{<}$ denotes the strict part of R, i.e., $a \overset{R}{<} b \overset{\text{def}}{=} a\,R\,b \wedge \neg(b\,R\,a)$. We use a function to produce an order from an order. For a function $g : A \to B$ and a quasi-order $R : A \rightsquigarrow A$, a quasi-order $R_g : B \rightsquigarrow B$ is defined by $a\;R_g\;b \overset{\text{def}}{=} g(a)\;R\;g(b)$ where both $g(a)$ and $g(b)$ must be defined. It is easy to confirm that these definitions exactly meet the requirement of quasi-orders or equivalent relations. The following equations show alternative definition of them.

$$\overset{R}{=} \overset{\text{def}}{=} R \cap R^\circ$$
$$\overset{R}{<} \overset{\text{def}}{=} R \cap \neg R^\circ$$
$$R_g \overset{\text{def}}{=} g^\circ \circ R \circ g$$

It is known that the sequential composition (also called lexicographic composition) of two quasi-orders is a quasi-order.

Definition 2.19 (sequential composition of two orders)**.** *For two relations $R : A \rightsquigarrow A$ and $S : A \rightsquigarrow A$, the* sequential composition of R and S, *denoted by $R\,;S$, is defined as follows.*

$$a\,(R\,;S)\,b \overset{\text{def}}{=} a\,S\,b \wedge (\neg(b\,S\,a) \vee a\,R\,b)$$ □

The sequential composition of two quasi-orders R and S, namely $R\,;S$, is a quasi-order, where the ordering is the same as S expect for equivalent elements in S, and equivalent elements in S are compared by R. The operator ; is associative. Note that the following definition is equivalent to the definition above.

$$R\,;S \overset{\text{def}}{=} S \cap (S^\circ \Rightarrow R)$$

We introduce the notion of *stronger order*, which has a deep relationship to *min* and *mnl*.

Definition 2.20 (stronger, strictly stronger, completely stronger)**.** *A quasi-order R is said to be* stronger *than a quasi-order S if aSb implies aRb for any a and b. A quasi-order R is said to be* strictly stronger *than a quasi-order S if $a \overset{S}{<} b$ implies $a \overset{R}{<} b$ for any a and b. A quasi-order R is said to be* completely stronger *than a quasi-order S if R is both stronger and strictly stronger than S.* □

Notice that the strictly stronger property does not imply the stronger property. It is because "R is strictly stronger than S" means that the strict part of R is stronger than that of S.

To extract minimum elements, we use an operator *min*. For a relation $R : A \rightsquigarrow A$, the relation $min_R : \mathsf{P}A \rightsquigarrow A$ is defined as follows.

$$(a, X) \in min_R \overset{\text{def}}{=} \forall b \in X : a\,R\,b$$

We can also give an equivalent definition in axiomatic style as follows.

$$min_R \overset{\text{def}}{=} \in \cap\, R/\ni$$

We use *mnl* to extract minimal elements, which is similar to *min* but different.

$$(a, X) \in mnl_R \overset{\text{def}}{=} \forall b \in X : b\,R\,a \Rightarrow a\,R\,b$$

The following definition is equivalent to the above.

$$mnl_R = min_{R^\circ \Rightarrow R}$$

If the relation R is a total quasi-order, $\Lambda min_R(X)$ computes all minimum elements in X based on the order R. However, min_R is useless when R is not total. For example, assume neither $a\,R\,b$ nor $b\,R\,a$ holds; then, $\Lambda min_R(X \cup \{a, b\}) = \emptyset$ for any X. Note that contrary to min_R, mnl_R works well even when R is not total.

The following lemmas are known important properties of *min*.

Lemma 2.21 (Equation 7.5 of [BdM96]).

$$min_R \circ \Lambda S = (S \cap R/S^\circ) \qquad \square$$

Lemma 2.22 (Exercise 7.10 of [BdM96]). *For any reflexive relations R and S, $R \subseteq S$ is equivalent to $min_R \subseteq min_S$.*

Proof.
($\Rightarrow$)

$$\begin{array}{rll}
min_R \subseteq min_S & \Leftrightarrow & \{ \text{ Definition of } min \ \} \\
& & (\in \cap R/\ni) \subseteq (\in \cap S/\ni) \\
& \Leftarrow & \{ \text{ trivial } \} \\
& & R/\ni \subseteq S/\ni \\
& \Leftrightarrow & \{ \text{ property of } / \ \} \\
& & (R/\ni \circ \ni) \subseteq S \\
& \Leftarrow & \{ \ (R/\ni \circ \ni) \subseteq R \text{ holds because of the property of } / \ \} \\
& & R \subseteq S \\
& \Leftrightarrow & \{ \text{ assumption } \} \\
& & \text{True}
\end{array}$$

($\Leftarrow$)

$$\begin{array}{rll}
min_R \subseteq min_S & \Leftrightarrow & \{ \text{ power transpose } \} \\
& & \forall X : \Lambda min_R(X) \subseteq \Lambda min_S(X) \\
& \Rightarrow & \{ \text{ Let } X = \{a,b\} \text{ where } a\,R\,b \ \} \\
& & \forall a,b : \ a\,R\,b \Rightarrow (\Lambda min_R(\{a,b\}) \subseteq \Lambda min_S(\{a,b\})) \\
& \Rightarrow & \{ \text{ from definition of } min \text{ and reflectivity of } R,\ \{a\} \subseteq \Lambda min_R(\{a,b\}) \ \} \\
& & \forall a,b : \ a\,R\,b \Rightarrow \{a\} \subseteq \Lambda min_S(\{a,b\}) \\
& \Rightarrow & \{ \text{ definition of } min \ \} \\
& & \forall a,b : \ a\,R\,b \Rightarrow a\,S\,b
\end{array} \qquad \square$$

3 Combinatorial Optimization Problems in Program Calculation

In this section, we review the results of existing works that give calculational laws for combinatorial optimization problems.

Optimization problems are problems where the objective is to find the best solution that is feasible. We can express specifications of optimization problems in terms of a function, a predicate, and a quasi-order; a function *enumerate* enumerates all solutions, a predicate *feasible* tests whether a solution is feasible or not, and an order R determines which solution is better.

$$\Lambda min_R \circ feasible\triangleleft \circ enumerate$$

In combinatorial optimization problems, a sequence of nondeterministic computation yields each solution. Let S be a relation that corresponds to the one step of nondeterministic computation. Then, we can express *enumerate* in terms of S.

$$enumerate = (\!|\Lambda(S \circ \mathsf{F}\!\in)|\!)$$

Or, alternatively, we can use the repetition style as follows.

$$enumerate = (\Lambda(S \circ \in))^*$$

Our objective is to give general calculational laws for problems described in the following forms.

$$\begin{array}{l}
\Lambda min_R \circ feasible\triangleleft \circ (\!|\Lambda(S \circ \mathsf{F}\!\in)|\!) \\
\Lambda min_R \circ feasible\triangleleft \circ (\Lambda(S \circ \in))^*
\end{array}$$

3.1 Greedy Theorem and Thinning Theorem

In this subsection, we review the existing results of Bird and de Moor [BdM93b, BdM93a, dM95, BdM96] and Curtis [Cur96, Cur03].

First, observe that *min* can do the computation of $\mathit{feasible}\lhd$. Let Q be the relation whose definition is $a\,Q\,b \overset{\text{def}}{=} \mathit{feasible}(a) \vee \neg\mathit{feasible}(b)$; then Q is a total quasi-order. Intuitively, Q is an ordering where elements for which *feasible* holds are smaller than those for which *feasible* does not hold. Now that the computation of $\mathit{feasible}\lhd$ is equivalent to min_Q whenever there is at least one feasible solution, the following expression is equivalent to the specification above.

$$\Lambda \mathit{min}_{R;Q} \circ \mathit{enumerate}$$

Thus it is sufficient for our objective to give calculational laws that enable us to solve problems written in the form above.

Alternatively, we can eliminate the $\mathit{feasible}\lhd$ part by fusing it with *enumerate*. Here we do not go this direction, while de Moor [dM95] gives a calculational law based on this observation.

In the existing works [BdM93b, BdM96, Cur96, Cur03], sufficient conditions to obtain efficient algorithms are shown.

Definition 3.1 (monotone)**.** *A relation $S : \mathsf{F}A \leadsto A$ is monotonic with respect to a quasi-order $R : A \leadsto A$ if the following property holds.*

$$\forall a \in \mathsf{F}A, b \in \mathsf{F}A, a' \in A : (a\, \mathsf{F}R\, b \wedge a'\, S\, a) \Rightarrow (\exists b' \in A : b'\, S\, b \wedge a'\, R\, b') \qquad \square$$

Note that the inequality above is equivalent to the following inequality.

$$R \circ S \supseteq S \circ \mathsf{F}R$$

For a relation $S : A \leadsto A$, the monotonicity condition above can be rephrased as follows by letting the relator F be I.

$$R \circ S \supseteq S \circ R$$

Now we introduce the greedy theorems.

Theorem 3.2 (greedy theorem for catamorphqisms [BdM93b, BdM96])**.** *If a relation $S : \mathsf{F}A \leadsto A$ is monotonic with respect to a quasi-order R°, then the following inequality holds.*

$$\mathit{min}_R \circ (\![\Lambda(S \circ \mathsf{F}\!\in)]\!) \supseteq \in \circ\, (\![\Lambda(\mathit{min}_R \circ (\Lambda S \circ \mathsf{F}\!\in))]\!) \qquad \square$$

Theorem 3.3 (greedy theorem for repetitions [Cur96, Cur03])**.** *If a relation $S : A \leadsto A$ is monotonic with respect to a quasi-order R°, then the following inequality holds.*

$$\mathit{min}_R \circ (\Lambda(S \circ \in))^{*} \supseteq \in \circ\, (\Lambda(\mathit{min}_R \circ (\Lambda S \circ \in)))^{*} \qquad \square$$

The statement of greedy theorems is natural. Since the monotone property implies that smaller elements are produced from smaller elements, we can discard nonminimum elements at each step to produce solutions. Note that "S is monotonic with respect to R" means "*larger* element yields *larger* element"; thus the theorems require monotonicity for R°.

Although these theorems give a clear formalization of greedy algorithms, they are difficult to use for nonspecialist. The most significant hardship is the step to find an appropriate order. The theorems require the monotone property, which is not easy to check because of its "for all something, exists something" style definition. To see the hardship, recall that a general specification of combinatorial optimization problems:

$$\Lambda \mathit{min}_R \circ \mathit{feasible}\lhd \circ (\![\Lambda(S \circ \mathsf{F} \in)]\!)$$

Defining a new relation Q such that $a\,Q\,b \overset{\text{def}}{=} \mathit{feasible}(a) \vee \neg\mathit{feasible}(b)$, we have derived the following program, for which the greedy theorem is applicable.

$$\Lambda \mathit{min}_{R;Q} \circ (\![\Lambda(S \circ \mathsf{F} \in)]\!)$$

But, it is not easy to confirm S is monotonic with respect to the complicated order $R\,;Q$. Moreover, the relation S may not be monotonic with respect to the order $R\,;Q$, that is, $R\,;Q$ is too strong and too many useful elements are discarded. Now, we would like to introduce a new weaker order T (namely, $R\,;Q$ is stronger than T), so that we could save appropriate amount of elements. However, since T is weaker than $R\,;Q$, T will not total. and then the greedy theorems may become useless, because min_T results in an empty set for many useful inputs if T is not total.

Bird and de Moor proposed another operator *thin* to deal with non-total quasi orders. For a quasi-order $R : A \leadsto A$, $\mathit{thin}_R : \mathsf{P}A \leadsto \mathsf{P}A$ is a relation satisfying the following property.

$$(Y, X) \in \mathit{thin}_R \stackrel{\text{def}}{=} (Y \subseteq X \land \forall b \in X, \exists a \in Y : a\,R\,b)$$

Intuitively, *thin* discards a part of elements that are larger than another element. Notice that *thin* can work well even if R is not total, which is one of the most important difference between *min* and *thin*. Now let us introduce another theorem, called thinning theorem.

Theorem 3.4 (thinning theorem [BdM96, Bir01])**.** *For any relations R and S, and an quasi-order Q, the following inequality holds provided that S is monotonic with respect to Q° and $Q \subseteq R$.*

$$\mathit{min}_R \circ (\![\Lambda(S \circ \mathsf{F} \in)]\!) \supseteq \mathit{min}_R \circ (\![\mathit{thin}_Q \circ \Lambda(S \circ \mathsf{F} \in)]\!) \qquad \Box$$

As similar to the greedy theorems, the thinning theorem matches our intuition. We can discard elements that never yield minimums, even when min_R discards elements that may yield minimums. The disposal of useless elements are performed by thin_Q, and the monotonicity condition of Q guarantees correctness of the disposal.

It is worth noting that one must give an implementation of thin_Q. Since the efficiency of derived algorithms highly depends on the implementation of thin_Q, the choice of its implementation is one of the most substantial issue. However, it is difficult to give appropriate implementation for thin_Q in general, because definition of *thin* is quite vague and there are too many choices. Too many functions, and even the identity function, satisfy the characterization.

Another important issue is how we obtain the order Q. The order Q does not appear in the specification, and thus one need to find Q from nothing; besides, Q° must satisfy the monotone property. Therefore, obtaining an appropriate order Q is not easy in practice.

Lastly, let us mention the inequality in the statements of the greedy theorems and the thinning theorem. The inequality implies that resulted programs may produce nothing even if there exist minimum elements. Although such outcomes are hardly raised for practical examples, those theorems guarantees nothing about it.

3.2 Automatic Derivation of Dynamic Programming Algorithms

Some researches [ALS91, BPT92, SHTO00] show that a class of combinatorial optimization problems, called *maximum marking problems*, is solvable automatically. Maximum marking problems are the problems where a solution is a marking of an underlying structure and the objective is to obtain the solution whose marked elements has the maximum sum. Marking should satisfy some constraint, which makes the problem hard. Bird [Bir01] clarified the relationship between the result about maximum marking problems and the thinning theorem. We review the result in this subsection.

First, we formalize the maximum marking problem. Let S_A be the type of the structure S that consists in values of type A. Let M_A be the type expressing markings for a value of type A, and its definition is $M_A = \{M(a) \mid a \in A\} \cup \{N(a) \mid a \in A\}$ where M and N are constructors. Let $\mathit{allMarking}_S : S_{\mathbb{R}} \to 2^{S_{M_{\mathbb{R}}}}$ be a function that stands for enumeration of all marking on the structure S. Let $\mathit{wsum} : S_{M_{\mathbb{R}}} \to \mathbb{R}$ be the function that sums up all numbers marked by M in a structure S. Now, given a constraint $\mathit{constraint} : S_{M_{\mathbb{R}}} \to \mathit{Bool}$, a maximum marking problem is a problem to compute the following expression.

$$\Lambda \mathit{min}_{\geq_{\mathit{wsum}}} \circ \mathit{constraint}\triangleleft \circ \mathit{allMarking}_S$$

Sasano et al. [SHTO00] showed that a class of maximum marking problems is solvable automatically. While they give the theorem that can cope with all algebraic data structures, here we only consider maximum marking problems on sequences for the simplicity of presentation. The followings are concrete definitions of functions *allMarking* $: \mathbb{R}^* \to 2^{(M^*_{\mathbb{R}})}$ and *wsum* $: M^*_{\mathbb{R}} \to \mathbb{R}$ for the case of sequences.

$$
\begin{array}{lcl}
\mathit{allMarking}([]) & \stackrel{\text{def}}{=} & [] \\
\mathit{allMarking}([a] +\!\!+\, x) & \stackrel{\text{def}}{=} & \{[a'] +\!\!+\, y \mid a' \in \{M(a), N(a)\} \wedge y \in \mathit{allMarking}(x)\} \\
\mathit{wsum}([]) & \stackrel{\text{def}}{=} & 0 \\
\mathit{wsum}([M(a)] +\!\!+\, x) & \stackrel{\text{def}}{=} & a + \mathit{wsum}(x) \\
\mathit{wsum}([N(a)] +\!\!+\, x) & \stackrel{\text{def}}{=} & \mathit{wsum}(x)
\end{array}
$$

In addition, we use an operator *fold* defined as follows.

$$
\begin{array}{lcl}
\mathit{fold}_{(f,e)}([]) & = & e \\
\mathit{fold}_{(f,e)}([a] +\!\!+\, x) & = & f(a, \mathit{fold}_{(f,e)}(x))
\end{array}
$$

Theorem 3.5 (optimization theorem on lists [SHTO00])**.** *Let MMP be a maximum marking problem defined as follows.*

$$\mathit{MMP} \stackrel{\text{def}}{=} \Lambda \mathit{min}_{\geq_{wsum}} \circ (\mathit{accept} \circ \mathit{fold}_{(\phi,e)}) \triangleleft \circ\ \mathit{allMarking}$$

Then, the following program computes an optimal solution in time linear to the size of the input, whenever the range of $\mathit{fold}_{(\phi,e)}$ is finite, functions ϕ and accept are computed in constant time, and all results of wsum and $\mathit{fold}_{(\phi,e)}$ are memoized.

$$
\begin{array}{l}
\mathit{MMP} = \Lambda \mathit{min}_{\geq_{wsum}} \circ (\mathit{accept} \circ \mathit{fold}_{(\phi,e)}) \triangleleft \circ\ \mathit{reducedMarking} \\
\quad \mathit{reducedMarking}([]) \stackrel{\text{def}}{=} [] \\
\quad \mathit{reducedMarking}([a] +\!\!+\, x) \stackrel{\text{def}}{=} \Lambda \mathit{mnl}_R(\{[a'] +\!\!+\, y \mid a' \in \{M(a), N(a)\} \wedge y \in \mathit{reducedMarking}(x)\}) \\
\quad a\, R\, b \stackrel{\text{def}}{=} a \geq_{wsum} b \wedge \mathit{fold}_{(\phi,e)}(a) = \mathit{fold}_{(\phi,e)}(b)
\end{array}
$$

□

Notice that the function *reducedMarking* generates only a small set of markings, because majority of markings are discarded by mnl_R in each step of recursion. Therefore, the programs resulted are efficient. The strong point of Theorem 3.5 is that we can obtain efficient program immediately once the specification of the problem is written in the required form.

As a concrete example, let us solve the *maximum segment sum* [Ben86, Bir89] problem by Theorem 3.5. The problem is to find the segment, namely consecutive sublist including an empty sequence, which has the maximum sum. The specification of the maximum segment sum problem *MSS* can be given in the maximum marking problem style as follows.

$$
\begin{array}{l}
\mathit{MSS} = \Lambda \mathit{min}_{\geq_{wsum}} \circ (\mathit{accept} \circ \mathit{fold}_{(\phi,e)}) \triangleleft \circ\ \mathit{allMarking} \\
\quad \mathit{accept}(p,q,r) \stackrel{\text{def}}{=} p \\
\quad e \stackrel{\text{def}}{=} (\mathit{True}, \mathit{True}, \mathit{True}) \\
\quad \phi(M(a), (p,q,r)) \stackrel{\text{def}}{=} (q, q, \mathit{False}) \\
\quad \phi(N(a), (p,q,r)) \stackrel{\text{def}}{=} (p, r, r)
\end{array}
$$

In the specification above, the function ϕ computes three boolean values, where the first records whether the marking corresponds to a segment, the second records whether the marking corresponds to an initial-segment, and the third records whether there is no marked element. Now that the specification of the maximum segment sum problem is certainly written in the required form, Theorem 3.5 immediately gives a linear-time program. Though the derived program is a bit inefficient, it exactly corresponds to the efficient program introduced by Bentley [Ben86], when we remove its inefficiency by the optimization proposed by Matsuzaki [Mat07].

Arnborg et al. [ALS91] and Borie et al. [BPT92] independently showed results similar to that of Sasano et al. Any maximum marking problem is solvable in time linear to the size of the underlying structure, whenever the underlying structure is a tree-decomposable graph and constraint is expressed by a monadic second order logic formula. The point is that we can translate these problems into equivalent maximum marking problems, where the underlying structure is a tree and the constraint can be checked by a tree automaton. Noticing that the operator *fold* does the same iteration to the string automaton, we can understand Theorem 3.5 as a special case of their results, where underlying structures are monadic trees. Their general results also corresponds to the general result of Sasano et al.: Sasano et al. use catamorphisms as a a generalization of *fold* to express constraints, and a tree automaton is essentially equivalent to a catamorphism of finite range.

Though they are interesting and useful, their drawback is the lack of generality. Only the maximum marking problems on a tree can be dealt with. They cannot work for other problems, for example problems on graphs.

Bird [Bir01] showed a correspondence between these results and the thinning theorem. Notice that the function *allMarking* can be expressed by a catamorphism, that is, $allMarking = (\!|\Lambda(marking \circ \mathsf{F}\in)|\!)$ where *marking* is an appropriate relation. The key observation is that the function *marking* is always monotonic with respect to the order $R \overset{\text{def}}{=} \geq_{wsum} \cap =_{fold_{(\phi,e)}}$. Moreover, Λmnl_R certainly satisfies the requirement of $thin_R$ in this case. Now we can see that the thinning theorem yields Theorem 3.5. This result sounds helpful to extend Theorem 3.5 so that it can cope with more general class of problems. However, Bird showed nothing about the extension of Theorem 3.5, and it is still unclear when the R above satisfies the monotonicity condition and when Λmnl_R satisfies the requirement of $thin_R$.

4 Calculus of Minimals

The objective of this section is to give an solution of the following four questions, which are not solved in the existing works.

- When can we obtain all minimums based on greedy algorithms?
- Can we check the premise of the greedy theorems more easily?
- Can we give a more concrete alternative of *thin*?
- Can we unify the greedy theorems and the thinning theorem into one framework?

We first introduce the notion of *proper thinnings*, which characterizes an appropriate disposal of surely unnecessary elements. We develop a calculus of minimals by identifying mnl_R as a proper thinning. In summary, we propose our greedy theorems for *mnl*, which give a solution of the questions above. Necessary lemmas are shown at the last of this section.

4.1 Proper thinning

First of all, let us introduce the notion of *proper thinnings*.

Definition 4.1. *A function $f : \mathsf{P}A \to \mathsf{P}A$ such that $f(X) \subseteq X$ for any $X \subseteq A$ is said to be a proper thinning if the following equation holds.*

$$f(f(X) \cup f(Y)) = f(X \cup Y) \qquad \Box$$

The notion of proper thinnings corresponds to discarding surely unnecessary elements. Consider $f(X \cup Y)$, discarding unnecessary elements in $X \cup Y$. The requirement states that we can locally discard surely unnecessary elements beforehand. In other words, the discarding must be conservative in the sense that it must not discard elements that may affect global disposal.

The following two lemmas shows important properties of proper thinnings.

Lemma 4.2. *For any proper thinning f, $f(X) = f(f(X))$ holds.*

Proof.

$$\begin{array}{rl} f(f(X) \cup f(Y)) = f(X \cup Y) & \Rightarrow \quad \{ \text{ Let } Y = \emptyset \ \} \\ & \quad f(f(X) \cup f(\emptyset)) = f(X \cup \emptyset) \\ & \Leftrightarrow \quad \{ \ f(\emptyset) = \emptyset \text{ because } Y \supseteq f(Y) \ \} \\ & \quad f(f(X)) = f(X) \end{array} \qquad \Box$$

Lemma 4.3. *For any proper thinning f and sets X and Y such that $X \supseteq Y$, $f(X) = f(Y)$ holds if and only if $Y \supseteq f(X)$ holds.*

Proof. The "only if" part is proved by the contraposition. If $Y \supseteq f(X)$ does not holds, $f(X) = f(Y)$ never holds because $Y \supseteq f(Y)$. The following calculation proves the "if" part.

$$\begin{array}{rcl}
(X \supseteq Y) \wedge (Y \supseteq f(X)) & \Rightarrow & \{ \ f(X) \cup Y = Y \ \} \\
& & (X \supseteq Y) \wedge (f(f(X) \cup Y) = f(Y)) \\
& \Leftrightarrow & \{ \ f \text{ is a proper thinning, Lemma 4.20} \ \} \\
& & (X \supseteq Y) \wedge (f(X \cup Y) = f(Y)) \\
& \Rightarrow & \{ \ X \cup Y = X \ \} \\
& & f(X) = f(Y)
\end{array}$$

□

The notion of proper thinning is general. Most of enumeration of minimums, enumeration of minimals, and filtering are proper thinnings. It is easy to see that a filtering function $p\triangleleft$ raised by a predicate p is a proper thinning.

Lemma 4.4. *For any predicate p, the filtering function $p\triangleleft$ is a proper thinning.*

Proof. It is fairly easy to see $p\triangleleft$ satisfies the requirement. □

However, neither Λmin_R nor Λmnl_R is a proper thinning in general. For example, consider a quasi-order R, where neither $a\,R\,b$ nor $b\,R\,a$ holds for elements a and b. Let $X = \{a, b\}$, $Y = \{b\}$. Then, although $\Lambda min_R(\Lambda min_R(X) \cup \Lambda min_R(Y)) = \Lambda min_R(\emptyset \cup \{b\}) = \{b\}$, $\Lambda min_R(X \cup Y) = \emptyset$ and the requirement is not satisfied. But this fact is not awkward. Notice that $\Lambda min_R(X)$ is not appropriate use of *min* in the sense it attempt to obtain the minimum elements from a set in which we cannot define the minimum. We would like to exclude such peculiar cases, and actually Bird and de Moor [dM92, BdM92, BdM96] introduced notions to determine proper use of *min* or *mnl*.

Definition 4.5 (well-bounded)**.** *A relation $R : A \rightsquigarrow A$ is said to be well-bounded if, for any nonempty set $X \subseteq A$, the following property is satisfied.*

$$\forall b \in X : \exists a \in \Lambda min_R(X) : a\,R\,b$$

□

The notion of well-boundedness is expressed by the following inequality.

$$\in \subseteq R^\circ \circ min_R$$

Definition 4.6 (well-supported)**.** *A relation $R : A \rightsquigarrow A$ is said to be well-supported if, for any nonempty set $X \subseteq A$, the following property is satisfied.*

$$\forall b \in X : \exists a \in \Lambda mnl_R(X) : a\,R\,b$$

□

Similar to well-boundedness, the notion of well-supportedness is expressed by the following inequality.

$$\in \subseteq R^\circ \circ mnl_R$$

Now we show that both Λmin_R and Λmnl_R are proper thinnings if we exclude peculiar cases.

Lemma 4.7. *The function Λmin_R is a proper thinning if R is well-bounded and transitive.*

Proof. From Lemma 4.22, $X \cup Y \supseteq \Lambda min_R(X) \cup \Lambda min_R(Y) \supseteq \Lambda min_R(X \cup Y)$ holds. Thus the requirement holds from Lemma 4.21, well-boundedness, and transitivity. □

Lemma 4.8. *The function Λmnl_R is a proper thinning if R is well-supported and transitive.*

Proof. From Lemma 4.22, $X \cup Y \supseteq \Lambda mnl_R(X) \cup \Lambda mnl_R(Y) \supseteq \Lambda min_R(X \cup Y)$ holds. Thus the requirement holds from Lemma 4.21 and well-boundedness, because $(R^\circ \Rightarrow R) \circ R \subseteq (R^\circ \Rightarrow R)$ holds as the following calculation shows.

$$\begin{array}{rcl}
c\,(R^\circ \Rightarrow R)\,b \wedge b\,R\,a & \Leftrightarrow & \{ \text{ definition of } \Rightarrow \} \\
& & (\neg(b\,R\,c) \vee c\,R\,b) \wedge b\,R\,a \\
& \Leftrightarrow & \{ \text{ distributing } \wedge \text{ over } \vee \} \\
& & (\neg(b\,R\,c) \wedge b\,R\,a) \vee (c\,R\,b \wedge b\,R\,a) \\
& \Rightarrow & \{ \text{ transitivity } \} \\
& & \neg(a\,R\,c) \vee (c\,R\,a) \\
& \Leftrightarrow & \{ \text{ definition of } \Rightarrow \} \\
& & c\,(R^\circ \Rightarrow R)\,a
\end{array}$$

□

As shown, many known computations are proper thinning. Moreover, we can construct proper thinnings.

Lemma 4.9. *For any proper thinning f, $f \circ p\triangleleft$ is a proper thinning.*

Proof.

$$\begin{array}{rcl}
(f \circ p\triangleleft)(X \cup Y) & = & \{ \text{ trivial (filter) } \} \\
& & f(p\triangleleft(X) \cup p\triangleleft(Y)) \\
& = & \{ \; f \text{ is a proper thinning } \} \\
& & f((f \circ p\triangleleft)(X) \cup (f \circ p\triangleleft)(Y)) \\
& = & \{ \text{ Since } f(X) \subseteq X,\ p\triangleleft \circ f \circ p\triangleleft = f \circ p\triangleleft \ \} \\
& & f((p\triangleleft \circ f \circ p\triangleleft)(X) \cup (p\triangleleft \circ f \circ p\triangleleft)(Y)) \\
& = & \{ \text{ trivial (filter) } \} \\
& & (f \circ p\triangleleft)((f \circ p\triangleleft)(X) \cup (f \circ p\triangleleft)(Y))
\end{array}$$

□

Now we would like to introduce our theorem, which shows when a proper thinning enables us to obtain an efficient algorithm. Recall that combinatorial optimization problems are formalized as follows.

$$\Lambda \mathit{min}_R \circ \mathit{feasible}\triangleleft \circ \mathit{enumerate}$$

In the previous section, we expressed *enumerate* in terms of a sequence of nondeterministic choices. For example, in repetition style, we describe *enumerate* as $(\Lambda S \circ \in)^*$. Here we would like to give a bit more detailed observation. A step of nondeterministic computation is expressed in terms of a union of deterministic choices. That is, there are appropriate functions f_i with an index set I such that $S = \bigcup_{i \in I} f_i$, in which each function f_i denotes each choice. This decomposition is natural in combinatorial optimization problems, because each choice is usually apparent from the problem.

Theorem 4.10 (proper thinning law for catamorphisms)**.** *If a function g is a proper thinning and $g \circ \mathsf{P}f_i \circ \Lambda\mathsf{F}\!\in \; \subseteq \mathsf{P}f_i \circ \Lambda\mathsf{F}\!\in \circ \mathsf{F}g$ holds for all f_i, then the following equation holds.*

$$g \circ (\!|(\biguplus_{i \in I} \mathsf{P}f_i) \circ \Lambda\mathsf{F}\!\in|\!)_{\mathsf{F}} = (\!|g \circ (\biguplus_{i \in I} \mathsf{P}f_i) \circ \Lambda\mathsf{F}\!\in|\!)_{\mathsf{F}}$$

Proof. From Theorem 2.4, it is sufficient to show that $g \circ (\biguplus_{i \in I} \mathsf{P}f_i) \circ \Lambda\mathsf{F}\!\in \; = g \circ (\biguplus_{i \in I} \mathsf{P}f_i) \circ \Lambda\mathsf{F}\!\in \circ \mathsf{F}g$.

$$\begin{array}{rcl}
g \circ (\biguplus_{i \in I} \mathsf{P}f_i) \circ \Lambda\mathsf{F}\!\in & = & \{ \; g \text{ is a proper thinning } \} \\
& & g \circ (\biguplus_{i \in I}(g \circ \mathsf{P}f_i \circ \Lambda\mathsf{F}\!\in)) \\
& = & \{ \text{ premise of the theorem and Lemma 4.3 } \} \\
& & g \circ (\biguplus_{i \in I}(g \circ \mathsf{P}f_i \circ \Lambda\mathsf{F}\!\in \circ \mathsf{F}g)) \\
& = & \{ \; g \text{ is a proper thinning } \} \\
& & g \circ (\biguplus_{i \in I} \mathsf{P}f_i) \circ \Lambda\mathsf{F}\!\in \circ \mathsf{F}g
\end{array}$$

□

Theorem 4.11 (proper thinning law for repetition)**.** *If a function g is a proper thinning and $g \circ \mathsf{P}f_i \subseteq \mathsf{P}f_i \circ g$ holds for all f_i, then the following equation holds.*

$$g \circ (\biguplus_{i \in I} \mathsf{P}f_i)^* = (g \circ (\biguplus_{i \in I} \mathsf{P}f_i))^* \circ g$$

Proof. From Theorem 2.6, it is suffice to show that $g \circ (\biguplus_{i \in I} \mathsf{P}f_i) = g \circ (\biguplus_{i \in I} \mathsf{P}f_i) \circ g$.

$$\begin{array}{rcl}
g \circ (\biguplus_{i \in I} \mathsf{P}f_i) & = & \{ \; g \text{ is a proper thinning } \} \\
& & g \circ (\biguplus_{i \in I}(g \circ \mathsf{P}f_i)) \\
& = & \{ \text{ premise of the theorem and Lemma 4.3 } \} \\
& & g \circ (\biguplus_{i \in I}(g \circ \mathsf{P}f_i \circ g)) \\
& = & \{ \; g \text{ is a proper thinning } \} \\
& & g \circ (\biguplus_{i \in I} \mathsf{P}f_i) \circ g
\end{array}$$

□

Our greedy theorems above show when one can discard unnecessary elements based on a proper thinning. One important characteristics of our theorems is the guarantee that the derived programs can enumerate all optimal solutions. Peculiar cases, such as a generated program results in an empty set for many useful input, are excluded. It is one of our objective.

Another important characteristics is the use of functions. In our theorems, enumeration of candidates is described by an union of functions, instead of a relation. The key observation used here is that proper thinnings guarantees that the disposal is conservative; thus we can decompose a relation into union of functions. Though it may look a small change, it is quite effective. It is sufficient to consider teach choice independently for confirming the premise for our theorems. Moreover, we can enjoy many good properties of functions. In the next subsection, we will propose our greedy theorems, whose premises are easy to confirm thanks for the use of functions. In the next section, we will make intensive use of properties of functions and introduce useful calculational laws.

4.2 Calculus of Minimals

In this section, we focus on a specific proper thinning, mnl_R with well-supported quasi-order R, and develop a calculus of *mnl*. We show greedy theorems for *mnl*, which are similar to those for *min*. In the next section, we will build a useful calculational law on the greedy theorems for *mnl*.

One may wonder why we focus on Λmnl_R, instead of Λmin_R. First of all, nothing is lost from use of mnl_R instead of min_R.

Lemma 4.12. *Well-boundedness implies well-supportedness.*

Proof. It is obvious because each minimum element is a minimal element. □

Lemma 4.13. *For any well-bounded quasi-order R, min_R is equivalent to mnl_R.*

Proof. First, notice that well-boundedness implies totality. It is because, if R is not total, there exist elements a and b such that neither $a\,R\,b$ nor $b\,R\,a$ holds, and then $\{a, b\}$ has no minimum element. Therefore, from Lemma 2.22, it is sufficient to show $R = (R^\circ \Rightarrow R)$ if R is total.

$$\begin{array}{rcl} a\,(R^\circ \Rightarrow R)\,b & \Leftrightarrow & \{ \text{ definition of } \Rightarrow \} \\ & & \neg(b\,R\,a) \vee a\,R\,b \\ & \Leftrightarrow & \{ \text{ trivial } (\vee) \} \\ & & (\neg(b\,R\,a) \wedge \neg(a\,R\,b)) \vee a\,R\,b \\ & \Leftrightarrow & \{ R \text{ is total } \} \\ & & a\,R\,b \end{array}$$
□

Moreover, the use of *mnl* is beneficial. Well-supportedness is much easier to obtain than well-boundedness, and we can enjoy many properties when we consider to construct appropriate orders. One important difference is about equivalence relations. Any equivalence relation is well-supported, while it is not well-bounded unless it has exactly one equivalence class. Furthermore, different from well-boundedness, well-supportedness is remarkably stable under usual construction of quasi-orders.

Lemma 4.14. *Any equivalence relation is well-supported.*

Proof. From definition, $\Lambda mnl_R = id$ if R is an equivalence relation. Thus R is well-supported. □

Lemma 4.15 (Lemma 8 in [BdM92])**.** *For any polynomial functor F, function f, and well-supported relations R and S, R_f, $R \cap S$, $R\,;S$, and $\mathsf{F}R$ are well-supported.* □

Well, let us see the properties of *mnl* itself. The lemma bellow shows that the strictly stronger property characterizes the computation of mnl_R.

Lemma 4.16. *A quasi-order R is strictly stronger than a quasi-order S if and only if $mnl_R \subseteq mnl_S$ holds.*

Proof. Notice that relations $(R^\circ \Rightarrow R)$ and $(S^\circ \Rightarrow S)$ are reflective. Thus, from Lemma 2.22, it is sufficient to show that R is strictly stronger than S if and only if $(R^\circ \Rightarrow R) \subseteq (S^\circ \Rightarrow S)$ holds.

$$
\begin{array}{rcl}
\overset{R}{<} \supseteq \overset{S}{<} & \Leftrightarrow & \{ \text{ definition of } \overset{R}{<} \text{ and } \overset{S}{<} \ \} \\
 & & (\neg R^{\circ} \cap R) \supseteq (\neg S^{\circ} \cap S) \\
 & \Leftrightarrow & \{ \text{ introducing negations } \} \\
 & & \neg(\neg R^{\circ} \cap R) \subseteq \neg(\neg S^{\circ} \cap S) \\
 & \Leftrightarrow & \{ \text{ De Morgan law } \} \\
 & & (R^{\circ} \cup \neg R) \subseteq (S^{\circ} \cup \neg S) \\
 & \Leftrightarrow & \{ \text{ introducing implications } \} \\
 & & (R \Rightarrow R^{\circ}) \subseteq (S \Rightarrow S^{\circ}) \\
 & \Leftrightarrow & \{ \text{ introducing converse } \} \\
 & & (R \Rightarrow R^{\circ})^{\circ} \subseteq (S \Rightarrow S^{\circ})^{\circ} \\
 & \Leftrightarrow & \{ \text{ Lemma 4.23 } \} \\
 & & (R^{\circ} \Rightarrow R) \subseteq (S^{\circ} \Rightarrow S)
\end{array}
\qquad \square
$$

We would like to consider the greedy theorem for *mnl*. Since we have formalized enumeration of candidates in terms of functions instead of a relation, the notion of monotonicity becomes simpler.

Definition 4.17 (monotone, strictly monotone, completely monotone). *For a function $f : \mathsf{F}A \to A$ and a quasi-order $R : A \leadsto A$, f is* monotonic *(strictly monotonic, completely monotonic) with respect to R if R_f is stronger (strictly stronger, completely stronger, respectively) than $\mathsf{F}R$.* □

The definition of monotonicity above exactly corresponds to the definition of previous section. Notice that since $\mathsf{I}A$ is equivalent to A, a function $f : A \to A$ is monotonic (strictly monotonic, completely monotonic) with respect to R if R_f is stronger (strictly stronger, completely stronger, respectively) than R. Now we give greedy theorems for *mnl*.

Theorem 4.18 (minimal-based greedy theorem for catamorphqisms). *Given a polynomial functor F and well-supported relation R, the following equation holds provided that each total function $f_i : \mathsf{F}A \to A$ is strictly monotonic with respect to R.*

$$\Lambda mnl_R \circ \llparenthesis (\biguplus_{i \in I} \mathsf{P}f_i) \circ \Lambda\mathsf{F}\!\in \rrparenthesis = \llparenthesis \Lambda mnl_R \circ (\biguplus_{i \in I} \mathsf{P}f_i) \circ \Lambda\mathsf{F}\!\in \rrparenthesis$$

Proof. Since Λmnl_R is a proper thinning because of the well-supportedness of R, it suffices to show $\Lambda mnl_R \circ \mathsf{P}f_i \circ \Lambda\mathsf{F}\!\in\, = \Lambda mnl_R \circ \mathsf{P}f_i \circ \mathsf{F}\Lambda mnl_R$ from Theorem 4.10.

$$
\begin{array}{rcl}
\Lambda mnl_R \circ \mathsf{P}f_i \circ \Lambda\mathsf{F}\!\in & = & \{ \text{ trivial } (mnl) \ \} \\
 & & \mathsf{P}f_i \circ \Lambda mnl_{R_{f_i}} \circ \Lambda\mathsf{F}\!\in \\
 & = & \{ \ R_{f_i} \supseteq \mathsf{F}R, \text{ and Lemmas 4.3 and 4.16 } \} \\
 & & \mathsf{P}f_i \circ \Lambda mnl_{R_{f_i}} \circ \Lambda mnl_{\mathsf{F}R} \circ \Lambda\mathsf{F}\!\in \\
 & = & \{ \text{ Lemma 4.24 } \} \\
 & & \mathsf{P}f_i \circ \Lambda mnl_{R_{f_i}} \circ \Lambda\mathsf{F}\!\in \circ\, \mathsf{F}\Lambda mnl_R \\
 & = & \{ \text{ trivial } (mnl) \ \} \\
 & & \Lambda mnl_R \circ \mathsf{P}f_i \circ \Lambda\mathsf{F}\!\in \circ\, \mathsf{F}\Lambda mnl_R
\end{array}
\qquad \square
$$

Theorem 4.19 (minimal-based greedy theorem for repetitions). *If each total function f_i is strictly monotonic with respect to a well-supported relation R, then the following equation holds.*

$$\Lambda mnl_R \circ (\biguplus_{i \in I} \mathsf{P}f_i)^* = (\Lambda mnl_R \circ \biguplus_{i \in I} \mathsf{P}f_i)^* \circ \Lambda mnl_R$$

Proof. Since Λmnl_R is a proper thinning because of the well-supportedness of R, it is suffice to show $\Lambda mnl_R \circ \mathsf{P}f_i = \Lambda mnl_R \circ \mathsf{P}f_i \circ \Lambda mnl_R$ from Theorem 4.10.

$$
\begin{array}{rcl}
\Lambda mnl_R \circ \mathsf{P}f_i & = & \{ \text{ trivial } (mnl) \ \} \\
 & & \mathsf{P}f_i \circ \Lambda mnl_{R_{f_i}} \\
 & = & \{ \ R_{f_i} \supseteq R, \text{ and Lemmas 4.3 and 4.16 } \} \\
 & & \mathsf{P}f_i \circ \Lambda mnl_{R_{f_i}} \circ \Lambda mnl_R \\
 & = & \{ \text{ trivial } (mnl) \ \} \\
 & & \Lambda mnl_R \circ \mathsf{P}f_i \circ \Lambda mnl_R
\end{array}
\qquad \square
$$

The theorems above show sufficient conditions to solve combinatorial optimization problems efficiently. When R is total, we can obtain optimal solutions by considering only minimum candidates in each step. This case corresponds to greedy algorithms. Even if R is not total, we can obtain optimal solutions by computing minimal candidates in each step. This case corresponds to dynamic programming algorithms. Therefore, our theorems can derive both greedy and dynamic programming algorithms. It is worth noting that thanks for the use of functions, our theorems are much easier to confirm the premise than existing greedy theorems seen in the previous section. For our theorems, checking "for all something, something holds" style condition is sufficient, while one must check "for all something, exists something such that something holds" style condition for the existing theorems.

4.3 Remarks

Here we would like to mention relationship between existing greedy theorems and our greedy theorems.

As explained in Section 3.1, Bird, de Moor, and Curtis formalized greedy theorems [BdM93b, BdM93a, BdM96, Cur96, Cur03]. Our motivation is to give greedy theorems that are easier to use. We formalized *mnl* instead of *min* so that we will not be in trouble with totality. We use union of functions instead of relations to enumerate solutions so that we can easily check the monotonicity condition. We seek for the condition in which equality between specification and derived program exactly holds, so that we need not distinguish enumerating of all optimal solutions from obtaining an optimal solution. Actually, as the price of easiness, our theorems are a bit more restrictive than exiting ones. However, our theorems can cope with most of cases that existing theorems can cope with.

We use mnl_R instead of min_R. Even though mnl_R is equivalent to $min_{R^\circ \Rightarrow R}$, the use of mnl_R is a challenging task, because $R^\circ \Rightarrow R$ is not always a quasi-order even if R is a quasi-order. Thus theories prepared for *min* is not directly applicable for *mnl*. More precisely, $R^\circ \Rightarrow R$ is always reflective and total, but not transitive in general. If $R^\circ \Rightarrow R$ is transitive, our results agree with the existing result, because the monotonicity condition for $R^\circ \Rightarrow R$ is exactly the strictly monotonicity condition of R, as shown by Lemma 4.16.

To deal with non-total quasi-orders, Bird and de Moor introduced the thinning theorem [BdM96]. Although the thinning theorem is effective, the definition of *thin* is vague and thus use of *thin* is difficult. We focus on *mnl* and avoid introducing it. Certainly, Λmnl_R satisfies the requirement of *thin* if R is well-supported. The definition of mnl_R is not vague. Moreover, our greedy theorem can cope with most cases that the thinning theorem can cope with.

4.4 Lemmas

Lemma 4.20. *A function $f : A \to A$ such that $X \supseteq f(X)$ is a proper thinning if and only if $f(f(X) \cup Y) = f(X \cup Y)$ holds for any sets $X \subseteq A$ and $Y \subseteq A$.*

Proof.
($\Rightarrow$)

$$\begin{array}{rcl} f(f(X) \cup Y) & = & \{\ f \text{ is a proper thinning }\} \\ & & f(f(f(X)) \cup f(Y)) \\ & = & \{\ \text{Lemma 4.2 }\} \\ & & f(f(X) \cup f(Y)) \\ & = & \{\ f \text{ is a proper thinning }\} \\ & & f(X \cup Y) \end{array}$$

($\Leftarrow$)

$$\begin{array}{rcl} f(X \cup Y) & = & \{\ \text{assumption }\} \\ & & f(f(X) \cup Y) \\ & = & \{\ \cup \text{ is commutative }\} \\ & & f(Y \cup f(X)) \\ & = & \{\ \text{assumption }\} \\ & & f(f(Y) \cup f(X)) \\ & = & \{\ \cup \text{ is commutative }\} \\ & & f(f(X) \cup f(Y)) \end{array}$$

□

Lemma 4.21. *For relations S, T, P, and Q, assume that $P \circ Q \subseteq P$, and $\in \subseteq Q^\circ \circ min_P$ hold. Then, $min_P \circ \Lambda T \subseteq S \subseteq T$ implies $min_P \circ \Lambda S \subseteq min_P \circ \Lambda T$.*

Proof.

$$\begin{array}{rcl}
min_P \circ \Lambda S \subseteq min_P \circ \Lambda T & \Leftrightarrow & \{ \text{ Lemma 2.21 } \} \\
& & (S \cap P/S^\circ) \subseteq (T \cap P/T^\circ) \\
& \Leftrightarrow & \{ \text{ property of } \cap \text{ } \} \\
& & ((S \cap P/S^\circ) \subseteq T) \wedge ((S \cap P/S^\circ) \subseteq P/T^\circ) \\
& \Leftrightarrow & \{ \ S \subseteq T \ \} \\
& & (S \cap P/S^\circ) \subseteq P/T^\circ \\
& \Leftarrow & \{ \text{ trivial } (\cap) \text{ } \} \\
& & P/S^\circ \subseteq P/T^\circ \\
& \Leftrightarrow & \{ \text{ property of } / \text{ } \} \\
& & P/S^\circ \circ T^\circ \subseteq P \\
& \Leftarrow & \{ \ T = \in \circ \Lambda T \subseteq Q^\circ \circ min_P \circ \Lambda T \subseteq Q^\circ \circ S \ \} \\
& & P/S^\circ \circ S^\circ \circ Q \subseteq P \\
& \Leftarrow & \{ \text{ property of } / \text{ } \} \\
& & P \circ Q \subseteq P \\
& \Leftrightarrow & \{ \text{ assumption } \} \\
& & \text{True}
\end{array}$$

□

Lemma 4.22. *For any relations R, S, and T, $min_R \circ \Lambda(S \cup T) \subseteq (min_R \circ \Lambda S) \cup (min_R \circ \Lambda T)$ holds.*

Proof.

$$\begin{array}{ll}
\multicolumn{2}{l}{min_R \circ \Lambda(S \cup T) \subseteq (min_R \circ \Lambda S) \cup (min_R \circ \Lambda T)} \\
\Leftrightarrow & \{ \text{ Lemma 2.21 } \} \\
& ((S \cup T) \cap R/(S^\circ \cup T^\circ)) \subseteq ((S \cap R/S^\circ) \cup (T \cap R/T^\circ)) \\
\Leftrightarrow & \{ \text{ distribute } \cap \text{ over } \cup \text{ } \} \\
& ((S \cap R/(S^\circ \cup T^\circ)) \cup (T \cap R/(S^\circ \cup T^\circ))) \subseteq ((S \cap R/S^\circ) \cup (T \cap R/T^\circ)) \\
\Leftarrow & \{ \text{ trivial } \} \\
& (R/(S^\circ \cup T^\circ) \subseteq R/S^\circ) \wedge (R/(S^\circ \cup T^\circ) \subseteq R/T^\circ) \\
\Leftarrow & \{ \ / \text{ is anti-monotonic to its right operand } \} \\
& (S^\circ \cup T^\circ) \supseteq S^\circ \wedge (S^\circ \cup T^\circ) \supseteq T^\circ \\
\Leftrightarrow & \{ \text{ trivial } (\cup) \text{ } \} \\
& \text{True}
\end{array}$$

□

Lemma 4.23.

$$(R \Rightarrow S)^\circ = (R^\circ \Rightarrow S^\circ)$$

Proof. It is sufficient to show $((R \Rightarrow S)^\circ \supseteq X) \Leftrightarrow ((R^\circ \Rightarrow S^\circ) \supseteq X)$ for any X.

$$\begin{array}{rcl}
(R \Rightarrow S)^\circ \supseteq X & \Leftrightarrow & \{ \text{ introducing converse } \} \\
& & (R \Rightarrow S) \supseteq X^\circ \\
& \Leftrightarrow & \{ \text{ property of } \Rightarrow \text{ } \} \\
& & S \supseteq (R \cap X^\circ) \\
& \Leftrightarrow & \{ \text{ introducing converse } \} \\
& & S^\circ \supseteq (R^\circ \cap X) \\
& \Leftrightarrow & \{ \text{ property of } \Rightarrow \text{ } \} \\
& & (R^\circ \Rightarrow S^\circ) \supseteq X
\end{array}$$

□

Lemma 4.24. *For any polynomial functor F and relation R, $\Lambda mnl_{\mathsf{F}R} \circ \Lambda\mathsf{F}{\in} = \Lambda\mathsf{F}{\in} \circ \mathsf{F}\Lambda mnl_R$ holds.*

Proof.

$$
\begin{array}{lcl}
\Lambda mnl_{\mathsf{F}R} \circ \Lambda\mathsf{F}\!\in & = & \{\ \Lambda\mathsf{F}\!\in \text{ is a function, and the property of functions (1) } \} \\
 & & \Lambda(mnl_{\mathsf{F}R} \circ \Lambda\mathsf{F}\!\in) \\
 & = & \{\ \text{Lemma 2.21 } \} \\
 & & \Lambda(\mathsf{F}\!\in \cap\, (\mathsf{F}R^\circ \Rightarrow \mathsf{F}R)/\mathsf{F}\!\in^\circ) \\
 & = & \{\ \mathsf{F} \text{ is polynomial and Lemma 2.12 } \} \\
 & & \Lambda(\mathsf{F}\!\in \cap\, \mathsf{F}(R^\circ \Rightarrow R)/\mathsf{F}\!\in^\circ) \\
 & = & \{\ \mathsf{F} \text{ is polynomial and Lemma 2.14 } \} \\
 & & \Lambda(\mathsf{F}(\in \cap\, (R^\circ \Rightarrow R)/\ni)) \\
 & = & \{\ \text{definition of } mnl \ \} \\
 & & \Lambda\mathsf{F} mnl_R \\
 & = & \{\ \text{Lemma 2.8 } \} \\
 & & \Lambda\mathsf{F}\!\in \circ\, \mathsf{F}\Lambda mnl_R
\end{array}
$$
□

5 Preservation of Monotonicity

We explained the monotonicity properties are the key issue to construct efficient algorithms, though appropriate orders are hard to obtain in practice. In this section, we show several calculational laws to solve combinatorial optimization problems. First we introduce some calculational laws to obtain appropriate orders. We adopt the constructive approach, in which we construct appropriate complicated orders by combining simple orders. After that, we propose a theorem that enables us to solve combinatorial optimization problems with ease, as the outcome of the proposed laws. Necessary lemmas are shown in Section 5.3 with their proofs.

5.1 Preservation of Monotonicity

Recall the drawback of the greedy theorems. Although the monotonicity properties are necessary for the greedy theorems, it is hard to find such an appropriate order. What we need is calculational laws to derive an appropriate order constructively.

First we show that one of the simplest orders, namely equivalence relations, satisfies good properties.

Lemma 5.1. *Any function is strictly monotonic with respect to any equivalence relation.*

Proof. Any equivalence relation has no strict part; thus any function is strictly monotonic with respect to them. □

Lemma 5.2. *Any function is completely monotonic with respect to the equivalence relation* $=$.

Proof. For any function f, $a = b$ implies $f(a) = f(b)$ and thus f is monotonic with respect to $=$. From Lemma 5.1, any function is strictly monotonic with respect to $=$. In summary, any function is completely monotonic with respect to $=$. □

Although the statement of Lemmas 5.1 and 5.2 might sound trivial, they are important. Recall that any equivalence relation is well-supported, as shown in Lemma 4.14. These properties imply that equivalence relations are useful to construct appropriate orders.

Next, we show the way to construct orders that satisfy monotonicity properties. The following lemma gives a way to check whether R_f satisfies monotonicity properties.

Lemma 5.3. *For relation $R : A \leadsto A$ and functions $f : B \to A$ and $g : \mathsf{F}B \to B$, assume that there exists a (possibly partial) function $g' : \mathsf{F}A \to A$ such that $f \circ g = g' \circ \mathsf{F}f$ holds. Then, g is monotonic (strictly monotonic, completely monotonic) with respect to R_f if and only if g' is monotonic (strictly monotonic, completely monotonic, respectively) with respect to R on the range of $\mathsf{F}f$.*

Proof. We give a proof for the monotonic case. Others are similar.

$$\begin{array}{rcl} {(R_f)}_g \supseteq \mathsf{F}R_f & \Leftrightarrow & \{ \text{ definition of } {(R_f)}_g \ \} \\ & & g^\circ \circ f^\circ \circ R \circ f \circ g \supseteq \mathsf{F}(f^\circ \circ R \circ f) \\ & \Leftrightarrow & \{ \text{ property of converses } \} \\ & & (f \circ g)^\circ \circ R \circ (f \circ g) \supseteq \mathsf{F}(f^\circ \circ R \circ f) \\ & \Leftrightarrow & \{ \text{ assumption } \} \\ & & (g' \circ \mathsf{F}f)^\circ \circ R \circ (g' \circ \mathsf{F}f) \supseteq \mathsf{F}(f^\circ \circ R \circ f) \\ & \Leftrightarrow & \{ \text{ property of converses and and relators } \} \\ & & \mathsf{F}f^\circ \circ g'^\circ \circ R \circ g' \circ \mathsf{F}f \supseteq \mathsf{F}f^\circ \circ \mathsf{F}R \circ \mathsf{F}f \\ & \Leftrightarrow & \{ \text{ definition of } R_{g'} \ \} \\ & & \mathsf{F}f^\circ \circ R_{g'} \circ \mathsf{F}f \supseteq \mathsf{F}f^\circ \circ \mathsf{F}R \circ \mathsf{F}f \end{array} \qquad \square$$

Corollary 5.4. *For any functions f and g, g is completely monotonic with respect to $=_f$ if and only if there exists a (possibly partial) function g' such that $f \circ g = g' \circ \mathsf{F}f$ holds.*

Proof. From Lemma 5.3, it is sufficient that g' is completely monotonic with respect to $=$, and it is always satisfied because of Lemma 5.2. □

Lemma 5.3 and Corollary 5.4 show that if we can successfully obtain a function g' such that $f \circ g = g' \circ \mathsf{F}f$ holds, then we can check the monotonicity properties much easier. Note that obtaining such g' is a sufficient condition of fusion transformation, as shown in Theorem 2.4. Fusion transformation is well researched, and many researches have done for automatic implementation of Theorem 2.4 [SdM01, YHT05, Yok06]. We can automatically guarantee the existence of g' by borrowing results of such researches.

Effectiveness of use of functions, instead of relations, can be clearly seen in Lemma 5.3 and Corollary 5.4. It is very difficult to find similar lemmas if f and g are relations. The use of functions enables us to obtain monotonicity conditions constructively.

Monotonicity properties are closed under intersections, which is useful to construct a weaker relation.

Lemma 5.5. *For any relations R and S, a function $f : \mathsf{F}A \to A$ is monotonic (strictly monotonic, completely monotonic) with respect to $R \cap S$, provided that f is monotonic (strictly monotonic, completely monotonic, respectively) with respect to both R and S.*

Proof. We give a proof for the monotonic case. Others are similar.

$$\begin{array}{rcl} (R \cap S)_f \supseteq \mathsf{F}(R \cap S) & \Leftrightarrow & \{ \ (R \cap S)_f = R_f \cap S_f \ \} \\ & & R_f \cap S_f \supseteq \mathsf{F}(R \cap S) \\ & \Rightarrow & \{ \text{ assumption } \} \\ & & \mathsf{F}R \cap \mathsf{F}S \supseteq \mathsf{F}(R \cap S) \\ & \Leftrightarrow & \{ \text{ Lemma 2.9 } \} \\ & & \text{True} \end{array} \qquad \square$$

Next is about sequential composition of two orders, namely $R\,;S$.

Lemma 5.6. *For any polynomial functor F and relations R and S, a function $f : \mathsf{F}A \to A$ is monotonic (strictly monotonic, completely monotonic) with respect to $R\,;S$, provided that f is completely monotonic with respect to S and monotonic (strictly monotonic, completely monotonic, respectively) with respect to R.*

Proof. The monotonic case is proved in Lemma 5.15. The strictly monotonic case is proved by combining Lemmas 5.11 and 5.15. The completely monotonic case is the combination of both result. □

Lemma 5.6 is useful in practice. An order $R\,;S$ is used to solve multi-objective optimization problems, in which one would like to find R-minimum elements in the S-minimum elements.

Let us consider a concrete example borrowed from [MPRS99]. For a network, the length and the capacity of a path are respectively the weight sum and the maximum weight of edges in the path. Consider problems to find a minimum-length path or a maximum-capacity path. Since we may construct solutions by extending paths step by step, we would like to know whether an extension of a path satisfies the monotonicity conditions. For the minimum-length paths problem, an extension of a path is completely monotonic; for two paths p_1 and p_2 of the same destination, where the length of p_1 is strictly less than (equal to) the length of p_2, the length of $p_1 +\!\!+ [e]$ is certainly strictly less than (equal to) the length of $p_2 +\!\!+ [e]$. For the maximum-capacity paths problem, an extension of a path is monotonic but not strictly monotonic; for two paths p_1 and p_2 of the same destination, where the capacity of p_1 is strictly larger than the capacity of p_2, the capacity of

$p_1 ++ [e]$ is larger or equal than the capacity of $p_2 ++ [e]$. From the observation above, we can conclude that we can efficiently compute the maximum-capacity path in the shortest paths, because of Lemma 5.6; however, it may be hard to compute the shortest path in the maximum-capacity paths, because Lemma 5.6 is not applicable.

As seen in the example above, Lemma 5.6 enables us to solve complicated problems having additional constraints in a step-by-step manner by decomposing a problem into small and simple subproblems.

So far, we have explained how appropriate orders are obtained constructively. It is worth noting that the proposed constructions also preserve the well-supportedness, as shown in Lemma 4.15. Thus the constructed orders are applicable for *mnl* without caring well-supportedness.

5.2 Derivation of Dynamic Programming

In the previous section, we introduced some ways to obtain appropriate order. While they may be insightful in some extent, they have no direct connection to concrete problems. They do not give any solutions straightforwardly for a class of combinatorial optimization problems. In this section, we propose a useful theorem that enables us to solve a class of combinatorial optimization problems directly.

Before proposing the theorem, we would like to introduce a notion.

Definition 5.7 (follow)**.** *For functions $f : A \to B$ and $g : A \to C$, we say that f follows g if $g(a_1) = g(a_2)$ implies $f(a_1) = f(a_2)$ for any elements $a_1 \in A$ and $a_2 \in A$.* □

Lemma 5.8. *For functions f and g, the following statements are equivalent.*

1. *f follows g.*
2. *$=_f \supseteq =_g$.*
3. *There exists a (possibly partial) function f' such that $f = f' \circ g$.*

Proof. The second statement is just a rephrase of the first one. It is easy to see the third statement implies the second one: $g(a_1) = g(a_2)$ implies $f(a_1) = f'(g(a_1)) = f'(g(a_2)) = f(a_2)$. In the following, we show that the second statement implies the third. Let $f' = f \circ g^\circ$. Then $f \subseteq f' \circ g$ holds from the property of functions (3). The following calculation shows $f \supseteq f' \circ g$ also holds.

$$\begin{array}{cl} f & \\ = & \{\ h = h \circ h^\circ \circ h \text{ holds for any function } h\ \} \\ & f \circ f^\circ \circ f \\ \supseteq & \{\ =_f \supseteq =_g \text{ and } =_f = f^\circ \circ f\ \} \\ & f \circ g^\circ \circ g \\ = & \{\ \text{definition of } f'\ \} \\ & f' \circ g \end{array}$$

Thus $f' \circ g = f$ holds. Now it is sufficient to show f' is simple, that is $(f \circ g^\circ) \circ (f \circ g^\circ)^\circ \subseteq id$.

$$\begin{array}{rcl} (f \circ g^\circ) \circ (f \circ g^\circ)^\circ \subseteq id & \Leftrightarrow & \{\ \text{distributing a converse over the composition}\ \} \\ & & f \circ g^\circ \circ g \circ f^\circ \subseteq id \\ & \Leftrightarrow & \{\ f \text{ is a function, and the properties of functions (2) and (3)}\ \} \\ & & g^\circ \circ g \subseteq f^\circ \circ f \\ & \Leftrightarrow & \{\ (f^\circ \circ f) \Leftrightarrow =_f\ \} \\ & & =_g \subseteq =_f \end{array}$$

□

Now we would like to introduce our theorem.

Theorem 5.9 (derivation of dynamic programming)**.** *An index set I, functions $p : A \to Bool$, $q_i : \mathsf{F}A \to Bool$, $q'_i : A \to Bool$, $f_i : \mathsf{F}A \to A$, where $i \in I$ and a well-supported quasi-order $R : A \leadsto A$ are given. Assume that each function f_i is strictly monotonic with respect to R. Then, the following equations hold, where R' is defined as $R' \stackrel{\text{def}}{=} R \cap =_{g_1} \cap =_{g_2} \cap =_{g_3}$, provided that there exist functions g_j where $j \in \{1, 2, 3\}$*

such that p, each q_i, and each q'_i follow functions g_1, g_2, and g_3, respectively, and $\exists f'_{ij} : g_j \circ f_i = f'_{ij} \circ \mathsf{F} g_j$ holds for each f_i.

$$\Lambda mnl_R \circ p\triangleleft \circ (\![\biguplus_{i\in I}(q'_i\triangleleft \circ \mathsf{P} f_i \circ q_i\triangleleft) \circ \Lambda\mathsf{F}\!\in]\!) = \Lambda mnl_R \circ p\triangleleft \circ (\![\Lambda mnl_{R'} \circ \biguplus_{i\in I}(q'_i\triangleleft \circ \mathsf{P} f_i \circ q_i\triangleleft) \circ \Lambda\mathsf{F}\!\in]\!)$$
$$\Lambda mnl_R \circ p\triangleleft \circ (\biguplus_{i\in I}(q'_i\triangleleft \circ \mathsf{P} f_i \circ q_i\triangleleft))^* = \Lambda mnl_R \circ p\triangleleft \circ (\Lambda mnl_{R'} \circ \biguplus_{i\in I}(q'_i\triangleleft \circ \mathsf{P} f_i \circ q_i\triangleleft))^* \circ \Lambda mnl_{R'}$$

Proof. Since p follows g_1, $\Lambda mnl_R \circ p\triangleleft = \Lambda mnl_R \circ p\triangleleft \circ \Lambda mnl_{R\cap =_g}$ holds from Lemma 5.17. From Lemmas 4.3, 4.16, and 5.14, $\Lambda mnl_R \circ p\triangleleft \circ \Lambda mnl_{R\cap =_g} = \Lambda mnl_R \circ p\triangleleft \circ \Lambda mnl_{R'}$. R' is well-supported from Lemmas 4.14 and 4.15, and f_i is strictly monotonic with respect to R' from Lemma 5.5. Therefore, thanks for Lemma 5.18, the minimal-based greedy theorems (Theorems 4.18 or 4.19) are applicable and yield the equations. □

Theorem 5.9 is general, but a bit complicated and hard to understand. The intricacy comes from the filters $q_i\triangleleft$ and $q'_i\triangleleft$, which expresses restrictions of domain and range of f_i. In other words, Theorem 5.9 deal with problems where each choice itself has constraint, for example, each choice must not be chosen twice. The following corollary is simpler, which corresponds to the case where each choice has no constraint.

Corollary 5.10. *Assume that a function p follows a function g and each function f_i is strictly monotonic with respect with a quasi-order R. Then the following equations hold, provided that there exist functions f'_i such that $g \circ f_i = f'_i \circ \mathsf{F} g$ for all functions f_i.*

$$\Lambda mnl_R \circ p\triangleleft \circ (\![(\biguplus_{i\in I} \mathsf{P} f_i) \circ \Lambda\mathsf{F}\!\in]\!) = \Lambda mnl_R \circ p\triangleleft \circ (\![\Lambda mnl_{R\cap =_g} \circ (\biguplus_{i\in I} \mathsf{P} f_i) \circ \Lambda\mathsf{F}\!\in]\!)$$
$$\Lambda mnl_R \circ p\triangleleft \circ (\biguplus_{i\in I} \mathsf{P} f_i)^* = \Lambda mnl_R \circ p\triangleleft \circ (\Lambda mnl_{R\cap =_g} \circ (\biguplus_{i\in I} \mathsf{P} f_i))^* \circ \Lambda mnl_{R\cap =_g}$$

Proof. It is a direct consequence of Theorem 5.9; let each function q_i and q'_i be the constant function that always returns *True*. □

Theorem 5.9 and Corollary 5.10 give a way to efficiently solve combinatorial optimization problems. Assume that we can solve a simple problem to get R-minimal elements. Then we can solve a complicated problem having additional constraint, whenever the constraint satisfies the requirement. They are applicable for a quite wide class of combinatorial optimization problems because they does not require any specific structure of problems.

Corollary 5.10 states that considering $(R \cap =_g)$-minimal solutions are enough to get optimal solutions if the premises are satisfied. Recall that $(R\cap =_g)$ is the order where two candidates a and b are compared by R if and only if both elements belong to the same equivalent class of $=_g$, i.e., $g(a) = g(b)$; thus Corollary 5.10 can be understood as a derivation of dynamic programming algorithm. At each step of recursion, fill the table whose keys are equivalent classes of $=_g$ and the values are the R-minimal solutions of each classes. Theorem 5.9 requires much larger table whose keys are equivalent classes of $=_{g_1} \cap =_{g_2} \cap =_{g_3}$.

Let us examine the time complexity of the resulted programs of Corollary 5.10. First, assume R is a linear order. In this case, it is sufficient for each recursion step to keep only one element for each equivalent class, which is raised from g. Let k denote the size of the range of g; then k candidates are considered in each step. Thus their time complexity is $\mathrm{O}(kT_p + knI(T_R + T_g + T_f))$ where T_R, T_g, T_p, and T_f are the costs to compute R, g, p, and each f_i, respectively, n is the number of recursion, which is the size of the input structure for the case of catamorphisms, and I is the index set used in the corollary, which is the set of choices in each step. Notice that parameters except k are fixed when program inputted is fixed. Therefore the choice of g determines the efficiency of the derived program. If R is not a linear order, then the corollary might not be effective. For example, if R is an equivalence relation mnl_R cannot discard any candidates and the corollary can do nothing. But the corollary is effective for most of cases in practice. Similar observation can be done for Theorem 5.9: the time complexity of resulted program is proportional to the size of range of g_1 times that of g_2 times that of g_3.

The resulted algorithm may have a local inefficiency caused by the recursive recomputations of R and g_j. Memoizing the result of R and g_j may improve efficiency, especially the solutions are large structures and each choice corresponds to a construction of a structure.

What Theorem 5.9 requires is existence of functions g_j and f'_{ij}. To find an appropriate function g_j, one of the easiest ways is decomposition, as Lemma 5.8 suggests. For example, assume that p is decomposed into

$p_1 \circ p_2$; then, since p follows p_2 from Lemma 5.8, p_2 is a candidate of g_1. Actually, the constraints are defined in terms of a composition of functions in the most of cases. Next, we should confirm the existence of functions f'_{ij} such that $g_j \circ f_i = f'_{ij} \circ \mathsf{F} g_j$. As explained, this is a well-known premise of fusion transformations and we can automate this step. It may be interesting fact that fusability implies derivation of dynamic programming.

Corollary 5.10 is a generalization of Theorem 3.5. In Theorem 3.5, constraints are expressed by $(\mathit{accept} \circ \mathit{fold}_{(f,e)})\triangleleft$. Thus the constraint, namely $\mathit{accept} \circ \mathit{fold}_{(f,e)}$, follows $\mathit{fold}_{(f,e)}$ from Lemma 5.8. Moreover, $\mathit{fold}_{(f,e)}$ is always fusable to *allMarking*, which is fairly easy to confirm. In summary, since *allMarking* is a catamorphism on lists, Corollary 5.10 is always applicable for the maximum marking problems considered here, and yields linear-time algorithms if the range of $\mathit{fold}_{(f,e)}$ is finite. The same result also holds for the general result of Sasano et al. [SHTO00], namely the results for maximum marking problems on trees.

Finally, we would like to discuss a small extension of Theorem 5.9. Consider optimal solutions should satisfy two constraints p_1 and p_2. Assume that there exist functions h_1 and h_2 such that p_1 follows h_1, p_2 follows h_2, and both h_1 and h_2 satisfies the premises of Theorem 5.9. From the requirement of h_1 and Lemma 5.5, $R \cap =_{h_1}$ satisfies the strictly monotone condition. Thus, from the requirement of h_2 and Lemma 5.5, $R \cap =_{h_1} \cap =_{h_2}$ satisfies the strictly monotone condition. Now we can see that our theorem easily cope with logical conjunctions of constraints. Moreover, since logical negations do not affect to check the premise of Theorem 5.9, the theorem can cope with the logical conjunctions, logical disjunctions, and logical negations of constraints. The important thing is that it is sufficient to consider conditions p_1 and p_2 independently. This fact much eases difficulty to obtain efficient algorithms when one want to deal with complicated constraints.

5.3 Lemmas

Lemma 5.11. *For any relations R and S, $\overset{R\,;\,S}{<}$ is equivalent to $\overset{R}{<}\,;S$.*

Proof.

$$\begin{array}{ll}
\overset{R\,;\,S}{<} & \Leftrightarrow \quad \{ \text{ definition of } \overset{R\,;\,S}{<} \ \} \\
& \quad (S \cap (\neg S^\circ \cup R)) \cap (\neg S^\circ \cup (S \cap \neg R^\circ)) \\
& \Leftrightarrow \quad \{ \text{ distributivity } \} \\
& \quad (S \cap (\neg S^\circ \cup R) \cap \neg S^\circ) \cup (S \cap (\neg S^\circ \cup R) \cap (S \cap \neg R^\circ)) \\
& \Leftrightarrow \quad \{ \text{ simplification } \} \\
& \quad (S \cap \neg S^\circ) \cup (S \cap (\neg S^\circ \cup R) \cap \neg R^\circ) \\
& \Leftrightarrow \quad \{ \text{ distributivity } \} \\
& \quad (S \cap \neg S^\circ) \cup (S \cap \neg S^\circ \cap \neg R^\circ) \cup (S \cap R \cap \neg R^\circ) \\
& \Leftrightarrow \quad \{ \text{ simplification } \} \\
& \quad (S \cap \neg S^\circ) \cup (S \cap R \cap \neg R^\circ) \\
& \Leftrightarrow \quad \{ \text{ distributivity } \} \\
& \quad S \cap (\neg S^\circ \cup (R \cap \neg R^\circ)) \\
& \Leftrightarrow \quad \{ \text{ definition of } \overset{R}{<} \text{ and } \Rightarrow \} \\
& \quad S \cap (S^\circ \Rightarrow \overset{R}{<}) \\
& \Leftrightarrow \quad \{ \text{ definition of } \overset{R}{<}\,;S \ \} \\
& \quad \overset{R}{<}\,;S
\end{array}$$

□

Lemma 5.12. *For any relation R and equivalence relation S, $R \cap S$ is equivalent to $R\,;S$.*

Proof.

$$\begin{array}{ll}
R\,;S & = \quad \{ \text{ definition of } R\,;S \ \} \\
& \quad S \cap (\neg S^\circ \cup R) \\
& = \quad \{ \text{ distributivity } \} \\
& \quad (S \cap \neg S^\circ) \cup (S \cap R) \\
& = \quad \{ \text{ Since } S \text{ is equivalence relation, } S \cap \neg S^\circ = \emptyset \ \} \\
& \quad S \cap R
\end{array}$$

□

Lemma 5.13. *For any quasi-orders R and S, $R\,;S$ is strictly stronger than S.*

Proof.

$$\begin{array}{lll}
\overset{R\,;\,S}{<} \supseteq \overset{S}{<} & \Leftrightarrow & \{\ \text{Lemma 5.11}\ \} \\
& & \overset{R}{<}\,;S \supseteq \overset{S}{<} \\
& \Leftrightarrow & \{\ \text{definition of } \overset{R}{<}\,;S\ \} \\
& & (S \cap (\neg S^{\circ} \cup \overset{R}{<})) \supseteq \overset{S}{<} \\
& \Leftrightarrow & \{\ \text{distributivity}\ \} \\
& & ((S \cap \neg S^{\circ}) \cup (S \cap \overset{R}{<})) \supseteq \overset{S}{<} \\
& \Leftrightarrow & \{\ \text{definition of } \overset{S}{<}\ \} \\
& & (\overset{S}{<} \cup (S \cap \overset{R}{<})) \supseteq \overset{S}{<} \\
& \Leftrightarrow & \{\ \text{trivial } (\cup)\ \} \\
& & \text{True}
\end{array}$$

□

Lemma 5.14. *For any relation R and equivalence relation S, R is completely stronger than $R \cap S$.*

Proof. It is obvious that R is stronger than $R \cap S$. The following calculation shows R is strictly stronger than $R \cap S$.

$$\begin{array}{lll}
\overset{R\,\cap\,S}{<} & = & \{\ \text{Lemma 5.12}\ \} \\
& & \overset{R\,;\,S}{<} \\
& = & \{\ \text{Lemma 5.11}\ \} \\
& & \overset{R}{<}\,;S \\
& = & \{\ \text{Lemma 5.12}\ \} \\
& & \overset{R}{<} \cap S \\
& \subseteq & \{\ \text{trivial } (\cap)\ \} \\
& & \overset{R}{<}
\end{array}$$

□

Lemma 5.15. *For any polynomial functor F and relations R and S, a function $f : \mathsf{F}A \to A$ is monotonic with respect to $R\,;S$, provided that f is monotonic with respect to R and completely monotonic with respect to S.*

Proof.

$$\begin{array}{lll}
(R\,;S)_f \supseteq \mathsf{F}(R\,;S) & \Leftrightarrow & \{\ (R\,;S)_f = R_f\,;S_f\ \} \\
& & R_f\,;S_f \supseteq \mathsf{F}(R\,;S) \\
& \Leftrightarrow & \{\ \text{definition of } R\,;S\ \} \\
& & (S_f \cap ({S_f}^{\circ} \Rightarrow R_f)) \supseteq \mathsf{F}(S \cap (S^{\circ} \Rightarrow R)) \\
& \Leftarrow & \{\ \mathsf{F} \text{ is polynomial, and Lemmas 2.9 and 2.12}\ \} \\
& & (S_f \cap ({S_f}^{\circ} \Rightarrow R_f)) \supseteq (\mathsf{F}S \cap (\mathsf{F}S^{\circ} \Rightarrow \mathsf{F}R)) \\
& \Leftrightarrow & \{\ \text{property of } \cap\ \} \\
& & (S_f \supseteq (\mathsf{F}S \cap (\mathsf{F}S^{\circ} \Rightarrow \mathsf{F}R))) \wedge (({S_f}^{\circ} \Rightarrow R_f) \supseteq (\mathsf{F}S \cap (\mathsf{F}S^{\circ} \Rightarrow \mathsf{F}R))) \\
& \Leftrightarrow & \{\ S_f \supseteq \mathsf{F}S \text{ holds from assumption}\ \} \\
& & ({S_f}^{\circ} \Rightarrow R_f) \supseteq (\mathsf{F}S \cap (\mathsf{F}S^{\circ} \Rightarrow \mathsf{F}R)) \\
& \Leftrightarrow & \{\ \text{property of } \Rightarrow\ \} \\
& & \neg {S_f}^{\circ} \cup R_f \supseteq (\mathsf{F}S \cap (\neg \mathsf{F}S^{\circ} \cup \mathsf{F}R)) \\
& \Leftrightarrow & \{\ \text{distributivity}\ \} \\
& & \neg {S_f}^{\circ} \cup R_f \supseteq ((\mathsf{F}S \cap \neg \mathsf{F}S^{\circ}) \cup (\mathsf{F}S \cap \mathsf{F}R)) \\
& \Leftarrow & \{\ \text{trivial } (\cup)\ \} \\
& & (\neg {S_f}^{\circ} \supseteq (\mathsf{F}S \cap \neg \mathsf{F}S^{\circ})) \wedge (R_f \supseteq (\mathsf{F}S \cap \mathsf{F}R)) \\
& \Leftrightarrow & \{\ R_f \supseteq \mathsf{F}R \text{ holds from assumption}\ \} \\
& & \neg {S_f}^{\circ} \supseteq (\mathsf{F}S \cap \neg \mathsf{F}S^{\circ}) \\
& \Leftrightarrow & \{\ S_f \supseteq \mathsf{F}S \text{ holds from assumption}\ \} \\
& & S_f \cap \neg {S_f}^{\circ} \supseteq (\mathsf{F}S \cap \neg \mathsf{F}S^{\circ}) \\
& \Leftrightarrow & \{\ \text{definition of } \overset{\mathsf{F}S}{<} \text{ and } \overset{S_f}{<}\ \} \\
& & \overset{S_f}{<} \supseteq \overset{\mathsf{F}S}{<} \\
& \Leftrightarrow & \{\ \text{assumption}\ \} \\
& & \text{True}
\end{array}$$

□

Lemma 5.16. *Let $\preceq$ be a linear order in which* True *is strictly smaller than* False. *Then, for any quasi-order R and predicate q, the following inequality holds.*

$$(\Lambda mnl_R \circ q\lhd)(X) \subseteq \Lambda mnl_{R;\preceq_q}(X)$$

Proof. If $q\lhd(X) = \emptyset$, then $(\Lambda mnl_R \circ q\lhd)(X) = \emptyset \subseteq \Lambda mnl_{R;\preceq_q}(X)$. Now assume $q\lhd(X) = Y \neq \emptyset$.

$$\begin{aligned}
(\Lambda mnl_R \circ q\lhd)(X) \subseteq \Lambda mnl_{R;\preceq_q}(X) \quad \Leftrightarrow \quad & \{\text{ definition of } q\lhd \text{ }\} \\
& \Lambda mnl_R(Y) \subseteq \Lambda mnl_{R;\preceq_q}(X) \\
\Leftrightarrow \quad & \{\text{ Since } \forall b_1, b_2 \in Y : b_1 \preceq_p b_2,\ \Lambda mnl_R(Y) = \Lambda mnl_{R;\preceq_q}(Y) \text{ }\} \\
& \Lambda mnl_{R;\preceq_q}(Y) \subseteq \Lambda mnl_{R;\preceq_q}(X) \\
\Leftarrow \quad & \{\ \Lambda mnl \text{ is a proper thinning, } X \supseteq Y\text{, and Lemma 4.3 }\} \\
& \Lambda mnl_{R;\preceq_q}(X) \subseteq Y \\
\Leftrightarrow \quad & \{\ \Lambda mnl_{\preceq_q}(X) = Y \ \} \\
& \Lambda mnl_{R;\preceq_q}(X) \subseteq \Lambda mnl_{\preceq_q}(X) \\
\Leftrightarrow \quad & \{\text{ Lemmas 4.16 and 5.13 }\} \\
& \text{True}
\end{aligned}$$

□

Lemma 5.17. *For any well-supported quasi-order R, predicate p, and function g, the following equation holds if p follows g.*

$$\Lambda mnl_R \circ p\lhd = \Lambda mnl_R \circ p\lhd \circ \Lambda mnl_{R\cap =_g}$$

Proof. From Lemmas 4.3 and 4.9, it is sufficient to show $\Lambda mnl_{R\cap =_g}(X) \supseteq (\Lambda mnl_R \circ p\lhd)(X)$. Let $\preceq$ be a linear order in which True is strictly smaller than False.

$$\begin{aligned}
\Lambda mnl_{R\cap =_g}(X) \quad \supseteq \quad & \{\text{ Claim: } R\,;\preceq_p \text{ is strictly stronger than } R \cap =_g\text{, and Lemma 4.16 }\} \\
& \Lambda mnl_{R;\preceq_p}(X) \\
\supseteq \quad & \{\text{ Lemma 5.16 }\} \\
& (\Lambda mnl_R \circ p\lhd)(X)
\end{aligned}$$

Now we prove the claim.

$$\begin{aligned}
\overset{R\,\cap\,=_g}{<} \quad = \quad & \{\text{ Lemma 5.12 }\} \\
& \overset{R\,;\,=_g}{<} \\
= \quad & \{\text{ Lemma 5.11 }\} \\
& \overset{R}{<}\,;=_g \\
= \quad & \{\text{ Lemma 5.12 }\} \\
& \overset{R}{<} \cap =_g \\
\subseteq \quad & \{\ p \text{ follows } g \text{ and Lemma 5.8 }\} \\
& \overset{R}{<} \cap =_p \\
= \quad & \{\text{ definition of } \preceq_p \text{ }\} \\
& \overset{R}{<} \cap \preceq_p \cap \preceq_p{}^{\circ} \\
\subseteq \quad & \{\ (A \cap B) \subseteq ((A \cap B) \cup \neg A) = A \Rightarrow B \ \} \\
& (\preceq_p{}^{\circ} \Rightarrow \overset{R}{<}) \cap \preceq_p \\
= \quad & \{\text{ definition of } \overset{R}{<}\,;\preceq_p \text{ }\} \\
& \overset{R}{<}\,;\preceq_p \\
= \quad & \{\text{ Lemma 5.11 }\} \\
& \overset{R\,;\,\preceq_p}{<}
\end{aligned}$$

□

Lemma 5.18. *For functions f, g, p, and q, let f' be a partial function such that $\mathsf{P}f' = p\lhd \circ \mathsf{P}f \circ q\lhd$. Such f' always exists, and f' is monotonic (strictly monotonic, completely monotonic) with respect to a quasi-order $R \cap =_{g_1} \cap =_{g_2}$, provided that p and q follow g_1 and g_2 respectively, and f is monotonic (strictly monotonic, completely monotonic, respectively) with respect to $R \cap =_{g_1} \cap =_{g_2}$.*

Proof. First, we give the definition of f'.

$$f'(a) \overset{\text{def}}{=} f(a) \quad \text{if } q(a) \wedge p(f(a))$$

It is obvious that f' satisfies the requirement. Now we check the monotone property. We prove the monotonic case, and others are similar.

$$\begin{array}{lll}
a\,(R \cap =_{g_1} \cap =_{g_2})\,b & \Leftrightarrow & \{ \text{ definition of } R \cap =_{g_1} \cap =_{g_2} \} \\
& & a\,(R \cap =_{g_1} \cap =_{g_2})\,b \wedge g_2(a) = g_2(b) \\
& \Leftrightarrow & \{ \ q \text{ follows } g_2 \ \} \\
& & a\,(R \cap =_{g_1} \cap =_{g_2})\,b \wedge q(a) = q(b) \\
& \Rightarrow & \{ \ \text{assumption} \ \} \\
& & f(a)\,(R \cap =_{g_1} \cap =_{g_2})\,f(b) \wedge q(a) = q(b) \\
& \Rightarrow & \{ \ \text{definition of } R \cap =_{g_1} \cap =_{g_2} \ \} \\
& & f(a)\,R\,f(b) \wedge g_1(f(a)) = g_1(f(b)) \wedge q(a) = q(b) \\
& \Rightarrow & \{ \ p \text{ follows } g_1 \ \} \\
& & f(a)\,R\,f(b) \wedge p(f(a)) = p(f(b)) \wedge q(a) = q(b)
\end{array}$$

From the calculation above, $a\,(R \cap =_g)\,b$ implies the following two: (i) $f(a)\,R\,f(b)$ (ii) $f'(a)$ is undefined if and only if $f'(b)$ is undefined. Therefore, f' is monotonic with respect to $R \cap =_{g_1} \cap =_{g_2}$. □

6 Deriving Algorithms for Regular-Language Constrained Shortest Path Problems

In this section, we consider the regular-language constrained shortest path problems [Rom88, BJM00]. We derive algorithms to solve the problems as applications of Theorem 5.9. Note that since we assume there is no negatively weighted cycle, there is a lower bound of distance of two vertexes. In other words, each order used in this section is well-supported.

6.1 Shortest Path Problems

Before discussing the regular-language shortest path problems, we would like to review calculational aspects of the shortest path problems. Given a network $((V, E), w)$, a source $s \in V$, and a destination $t \in V$, a shortest path problem is the problem to compute the shortest paths from s to t. The following is the specification of the shortest path problems in a calculational style.

$$\begin{array}{l}
SP \overset{\text{def}}{=} (\pi_1 \circ \Lambda mnl_W \circ endWith_t\lhd \circ (\biguplus_{e \in E} \mathsf{P}extend_e \uplus \mathsf{P}id)^*)\ \{([], s)\} \\
\quad endWith_v(p, v') \overset{\text{def}}{=} v = v' \\
\quad extend_e(p, v) \overset{\text{def}}{=} (p +\!\!+ [e], v') \quad \text{if } e = (v, v') \\
\quad (p_1, v_1)\,W\,(p_2, v_2) \overset{\text{def}}{=} w(p_1) \leq w(p_2)
\end{array}$$

Throughout the computation, we manage a pair as a candidate, whose first value records a path and second value records the terminal vertex of the path. All paths are enumerated by recursive application of $extend_e$, where subscript e stands for the edge concatenated to paths. We extract all paths terminating with the vertex t, and search the path of the minimum weight.

6.1.1 Deriving the Bellman-Ford Algorithm

Now we would like to derive an efficient algorithm from the specification. Notice that we can decompose $extend_e$ into a function $addEdge_e$ with a predicate $endWith_v$.

$$\begin{array}{c}
\mathsf{P}extend_e = \mathsf{P}addEdge_e \circ endWith_{\pi_1(e)}\lhd \\
addEdge_e(p, v) \overset{\text{def}}{=} (p +\!\!+ [e], \pi_2(e))
\end{array}$$

Since $addEdge_e$ is a total function, while $extend_e$ is a partial function, it is easy to confirm the monotonicity condition. Actually it is obvious that id and $addEdge_e$ is strictly monotonic with respect to W. Now observe that $endWith_v$ uses only its second input, which tells us $endWith_v$ follows π_2. Therefore, we use π_2 to derive a dynamic programming algorithm based on Theorem 5.9.

Let us calculate as follows.

$$\begin{array}{ll}
\multicolumn{2}{l}{SP} \\
= & \{ \text{ definition of } SP \ \} \\
 & (\pi_1 \circ \Lambda mnl_W \circ endWith_t\lhd \circ (\biguplus_{e \in E} \mathsf{P} extend_e \uplus \mathsf{P} id)^*) \ \{([], s)\} \\
= & \{ \text{ decompose } extend_e \ \} \\
 & (\pi_1 \circ \Lambda mnl_W \circ endWith_t\lhd \circ (\biguplus_{e \in E} (\mathsf{P} addEdge_e \circ endWith_{\pi_1(e)}\lhd) \uplus \mathsf{P} id)^*) \ \{([], s)\} \\
= & \{ \text{ Theorem 5.9 (claim: } \pi_2 \circ addEdge_e = f'_e \circ \pi_2\text{; notice that } \pi_2 \circ id = id \circ \pi_2 \text{ holds.) } \} \\
 & (\pi_1 \circ \Lambda mnl_W \circ endWith_t\lhd \circ (\Lambda mnl_{W \cap =_{\pi_2}} \circ (\biguplus_{e \in E} (\mathsf{P} addEdge_e \circ endWith_{\pi_1(e)}\lhd) \uplus \mathsf{P} id))^*) \ \{([], s)\} \\
= & \{ \text{ folding } extend_e \ \} \\
 & (\pi_1 \circ \Lambda mnl_W \circ endWith_t\lhd \circ (\Lambda mnl_{W \cap =_{\pi_2}} \circ (\biguplus_{e \in E} \mathsf{P} extend_e \uplus \mathsf{P} id))^*) \ \{([], s)\}
\end{array}$$

The claim is confirmed as follows.

$$\begin{array}{lll}
(\pi_2 \circ addEdge_e) \ (p, v) & = & \{ \text{ definition of } addEdge_e \ \} \\
 & & \pi_2(e) \\
 & = & \{ \text{ Let } f'_e(v) \overset{\text{def}}{=} \pi_2(e) \ \} \\
 & & (f'_e \circ \pi_2) \ (p, v)
\end{array}$$

We derived a program based on Theorem 5.9. Recall that the quasi-order $W \cap =_{\pi_2}$ compares the paths of the same destination. Therefore, this program recursively computes candidates of shortest paths from s to each vertex, until we cannot find better paths any more. This algorithm is exactly the Bellman-Ford algorithm. The time complexity of the derived algorithm is $\mathrm{O}(VE)$.

6.1.2 Deriving the Dijkstra Algorithm

If all weights are positive, the Dijkstra algorithm improves efficiency. The idea of the Dijkstra algorithm is to delay finding a path until it becomes the shortest path not searched yet. To express such delay, we give a counter that counts the number of recursive calls accomplished.

$$\begin{array}{l}
SP = (\pi_1 \circ \Lambda mnl_{W'} \circ endWith'_t\lhd \circ (\biguplus_{e \in E} \mathsf{P} extend'_e \uplus \mathsf{P} next)^*) \ \{([], s, 0)\} \\
\quad endWith'_v(p, v', k) \overset{\text{def}}{=} v = v' \\
\quad extend'_e(p, v, k) \overset{\text{def}}{=} (p +\!\!+ [e], v', k+1) \quad \text{if } e = (v, v') \\
\quad next(p, v, k) \overset{\text{def}}{=} (p, v, k+1) \\
\quad (p_1, v_1, k) \, W' \, (p_2, v_2, k) \overset{\text{def}}{=} w(p_1) \le w(p_2)
\end{array}$$

We introduce new functions $extendD_e$ that delay constructing paths to a vertex v until $\chi(\pi_2(e))$-th recursive call, according to a function $\chi : V \to \mathbb{N}$.

$$extendD_e(p, v, k) \overset{\text{def}}{=} extend'_e(p, v, k) \quad \text{if } e = (v, v') \wedge k \ge \chi(v')$$

Notice that $extendD_e$ does nothing harmful: it does nothing in the early step, and it will be equivalent to $extend'_e$ after some proper number of recursive calls. Therefore, $(\biguplus_{e \in E} \mathsf{P} extendD_e \uplus \mathsf{P} next)^*$ is equivalent to $(\biguplus_{e \in E} \mathsf{P} extend'_e \uplus \mathsf{P} next)^*$.

Now we do similar process to the previous derivation. First, we decompose the $extendD_e$ into a total function with a filtering.

$$\begin{array}{l}
\mathsf{P} extendD_e = \mathsf{P} addEdge'_e \circ properD_e\lhd \\
\quad addEdge'_e(p, v, k) \overset{\text{def}}{=} (p +\!\!+ [e], \pi_2(e), k+1) \\
\quad properD_e(p, v, k) \overset{\text{def}}{=} e = (v, v') \wedge \chi(v') \le k
\end{array}$$

Observe that $properD_e$ uses only its second and third input; thus $properD_e$ follows $\langle \pi_2, \pi_3 \rangle$. $endWith'_t$ also follows $\langle \pi_2, \pi_3 \rangle$, since $\mathit{endWith'}_t$ uses only its second input. Now we calculate as follows.

$$
\begin{array}{ll}
& SP \\
= & \{ \text{ delayed variant of } SP \ \} \\
& (\pi_1 \circ \Lambda mnl_{W'} \circ \mathit{endWith'}_t \lhd \circ (\biguplus_{e \in E} \mathsf{P}\mathit{extendD}_e \uplus \mathsf{P}\mathit{next})^*) \ \{([], s, 0)\} \\
= & \{ \text{ decompose } \mathit{extendD}_e \ \} \\
& (\pi_1 \circ \Lambda mnl_{W'} \circ \mathit{endWith'}_t \lhd \circ (\biguplus_{e \in E} (\mathsf{P}\mathit{addEdge'}_e \circ \mathit{properD}_e \lhd) \uplus \mathsf{P}\mathit{next})^*) \ \{([], s, 0)\} \\
= & \{ \text{ Theorem 5.9 (premises are confirmed bellow), and let } W'_{23} = W' \cap =_{\langle \pi_2, \pi_3 \rangle} \ \} \\
& (\pi_1 \circ \Lambda mnl_{W'} \circ \mathit{endWith'}_t \lhd \circ (\Lambda mnl_{W'_{23}} \circ (\biguplus_{e \in E} (\mathsf{P}\mathit{addEdge'}_e \circ \mathit{properD}_e \lhd) \uplus \mathsf{P}\mathit{next}))^*) \ \{([], s, 0)\} \\
= & \{ \text{ folding } \mathit{extendD}_e \ \ \} \\
& (\pi_1 \circ \Lambda mnl_{W'} \circ \mathit{endWith'}_v \lhd \circ (\Lambda mnl_{W'_{23}} \circ (\biguplus_{e \in E} \mathsf{P}\mathit{extendD}_e \uplus \mathsf{P}\mathit{next}))^*) \ \{([], s)\}
\end{array}
$$

The premises of Theorem 5.9 is easy to confirm, as the following calculations show.

$$
\begin{array}{lll}
(\langle \pi_2, \pi_3 \rangle \circ \mathit{next}) \ (p, v, k) & = & \{ \text{ definition of } \mathit{next} \ \} \\
& & (v, k+1) \\
& = & \{ \text{ Let } g(v,k) \overset{\text{def}}{=} (v, k+1) \ \} \\
& & (g \circ \langle \pi_2, \pi_3 \rangle) \ (p, v, k) \\
(\langle \pi_2, \pi_3 \rangle \circ \mathit{addEdge'}_e) \ (p, v, k) & = & \{ \text{ definition of } \mathit{addEdge}_e \ \} \\
& & (\pi_2(e), k+1) \\
& = & \{ \text{ Let } f''_e(v,k) \overset{\text{def}}{=} (\pi_2(e), k+1) \ \} \\
& & (f''_e \circ \langle \pi_2, \pi_3 \rangle) \ (p, v, k)
\end{array}
$$

We have derived an efficient algorithm for delayed variant of SP. In the calculation, we do not use any specific property of χ. This fact tells us that the correctness of the derived algorithms does not depend on the choice of χ. However, the efficiency of derived algorithms does depend the choice of χ. In the Dijkstra algorithm, χ results in the nearness ranking from the source vertex. The exact value of χ is not known in advance, but revealed as the computation goes. Since all weights of edges are positive, the nearest vertex that has not been visited yet is certainly the next nearest vertex in each step of computation. The time complexity of the Dijkstra algorithm is $\mathrm{O}(V \log V + E)$, if we implement it efficiently using a Fibonacci heap. Similarly, we can use A^* search algorithms, which is also delayed variant but uses another definition of χ.

6.2 Regular-Language Constrained Shortest Path Problems

In previous subsection, we saw that our theorem enables us to derive and prove the well-known shortest path algorithms. In this subsection, we show that our theorem also works well for more difficult problems, namely regular-language constrained shortest path problems. Regular-language constrained shortest path problems are the problems to compute the shortest path such that the label of the path should be in a regular language. Here we give the formal definition of the regular-language shortest path problems.

Definition 6.1 (regular-language constrained shortest path problem). *Given a network $N = ((V, E), w)$ with a labeling function l, a source $s \in V$, a destination $t \in V$, and a regular language L, a* regular language constrained shortest path problem *is the problem to find the shortest path p from s to t such that $l(p) \in L$.* □

Romeuf [Rom88] might be the first one who introduced the problems. Barret et al. [BJM00] generalized the problems as the formal-language constrained path problems. Regular-language constrained shortest path problems contain many problems, such as shortest odd/even path problems, shortest path problems with forbidden paths [VD05], and the traveling salesman problem; besides, regular-language constrained shortest path problems are important in many areas, such as transportation networks [BBJ+02, BBJ+07] and queries [FFG06].

A regular-language constrained shortest path problem have an constraint, namely the label of each path should be in a regular language. We use deterministic finite state automata to determine whether a string is in a regular language or not.

Definition 6.2 (deterministic finite state automata). *A deterministic finite state automaton $\mathcal{A}$ is a tuple $(Q, \Sigma, \delta, q_0, Q_F)$, where Q is a finite set of states, Σ is an alphabet, $\delta : (Q \times \Sigma) \to Q$ is a function to compute transition, $q_0 \in Q$ is the initial state, and $Q_F \subseteq Q$ is a set of final states.*

The run of an automaton $\mathcal{A} = (Q, \Sigma, \delta, q_0, Q_F)$ is denoted a function $accept_{\mathcal{A}} : \Sigma^* \to Bool$, which is defined using a function $state_{\mathcal{A}} : \Sigma^* \to Q$ as follows.

$$\begin{aligned}
accept_{\mathcal{A}} &\stackrel{\text{def}}{=} (\in Q_F) \circ state_{\mathcal{A}} \\
state_{\mathcal{A}}([]) &\stackrel{\text{def}}{=} q_0 \\
state_{\mathcal{A}}(x +\!\!+ [\sigma]) &\stackrel{\text{def}}{=} \delta(state(x), \sigma)
\end{aligned}$$

We assume $l(e) = e$ for the simplicity of the presentation; thus, the alphabet of automata is E. Now we give the specification of the problem in a calculational style.

$$RLSP \stackrel{\text{def}}{=} (\pi_1 \circ \Lambda mnl_W \circ (accept_{\mathcal{A}} \circ \pi_1)\lhd \circ endWith_t \lhd \circ (\textstyle\biguplus_{e \in E} \mathsf{P} extend_e \uplus \mathsf{P} id)^*)\ \{([], s)\}$$

The specification is the same as the usual shortest path problem, except for the additional constraint $(accept_{\mathcal{A}} \circ \pi_1)\lhd$.

6.2.1 Basic Algorithm

Romeuf [Rom88] proposed the basic algorithm to solve the regular-language constrained shortest path problem. In this subsection, we derive the algorithm from Theorem 5.9. The process of the derivation is almost the same as the case of the usual shortest path problems. Recall that $extend_e$ can be decomposed into $addEdge_e$ with $endWith_{\pi(e)}$. From the definition of $accept_{\mathcal{A}}$ and Lemma 5.8, $accept_{\mathcal{A}} \circ \pi_1$ follows $state_{\mathcal{A}} \circ \pi_1$. The following calculation confirm the fusable requirement.

$$\begin{aligned}
(state_{\mathcal{A}} \circ \pi_1 \circ addEdge_e)\ (p, v) &= \quad \{ \text{ definition of } addEdge \text{ } \} \\
&\quad\ state_{\mathcal{A}}(p +\!\!+ [e]) \\
&= \quad \{ \text{ definition of } state_{\mathcal{A}}, \text{ where } \mathcal{A} = (Q, \Sigma, \delta, q_0, Q_F) \text{ } \} \\
&\quad\ \delta(state_{\mathcal{A}}(p), e) \\
&= \quad \{ \text{ let } g_e(q) \stackrel{\text{def}}{=} \delta(q, e) \text{ } \} \\
&\quad\ (g_e \circ state_{\mathcal{A}} \circ \pi_1)\ (p, v)
\end{aligned}$$

Recall that $endWith_v$ follows π_2. Thus Theorem 5.9 derives the following algorithm, where an order $W_{\mathcal{A}} \stackrel{\text{def}}{=} W \cap =_{state_{\mathcal{A}} \circ \pi_1} \cap =_{\pi_2}$ is used to discard unnecessary candidates.

$$RLSP = (\pi_1 \circ \Lambda mnl_W \circ (accept_{\mathcal{A}} \circ \pi_1)\lhd \circ endWith_t \lhd \circ (\Lambda mnl_{W_{\mathcal{A}}} \circ (\biguplus_{e \in E} \mathsf{P} extend_e \uplus \mathsf{P} id))^*)\ \{([], s)\}$$

This program works as follows: compute candidates of shortest paths from the initial state of the source vertex to each state of each vertex recursively, until we cannot find better paths any more. We can recognize the program as the Bellman-Ford algorithm on the larger graph where each vertex and each edge are duplicated according to the states of the automaton. This is exactly the idea of the algorithm proposed by Romeuf [Rom88]. Romeuf showed that the regular-language constrained shortest paths are the shortest paths on the larger graph that corresponds to the product of the underlying graph and the automaton. The time complexity of the resulting program is $O(VQ^2E)$, where Q is the set of states of the automaton. This result is a bit inefficient. We need to construct the larger graph explicitly to compute the result, while many vertexes and edges are probably unnecessary to find the optimal path to the destination.

6.2.2 Case 1: all weights are positive

When all weights are positive, we can easily remove the inefficiency by the Dijkstra algorithm. The improvement was proposed by Barrett et al. [BBJ+02, BBJ+07].

We do the similar calculation. Again, we introduce $extendD_e$ to introduce the Dijkstra algorithm. $extendD_e$ can be decomposed into $addEdge'_e$ and $properD_e$. $state_{\mathcal{A}} \circ \pi_1$ satisfies the fusable condition, which is confirmed almost the same calculation as the previous one. It is because the computation for first component in $addEdge'_e$ is the same as that in $addEdge_e$ and $state_{\mathcal{A}} \circ \pi_1$ is only sensitive to first components. Now, since $properD$ follows $\langle \pi_2, \pi_3 \rangle$, we obtain the following algorithm, where $W'_{\mathcal{A}} \stackrel{\text{def}}{=} W' \cap =_{state_{\mathcal{A}} \circ \pi_1} \cap =_{\langle \pi_2, \pi_3 \rangle}$.

$$RLSP = (\pi_1 \circ \Lambda mnl_{W'} \circ (accept_{\mathcal{A}} \circ \pi_1)\lhd \circ endWith'_t \lhd \circ (\Lambda mnl_{W'_{\mathcal{A}}} \circ (\biguplus_{e \in E} \mathsf{P} extendD_e \uplus \mathsf{P} next))^*)\ \{([], s, 0)\}$$

Recall that a function χ manages the delay of $extendD_e$. Choosing an appropriate χ, we obtain the Dijkstra-like algorithm to solve regular-language constrained shortest path problems. The time complexity of the algorithm is $\mathrm{O}(VQ\log(VQ) + EQ)$, if we implement it efficiently using a Fibonacci heap, where Q is the set of states of the automaton. Since the feasible solution found firstly is the optimal solution, we may avoid constructing and traversing whole of the large graph. Similarly, we can use other algorithms, such as A^* algorithms.

The most important lesson is that the additional constraint, namely $(accept_{\mathcal{A}} \circ \pi_1)\triangleleft$, does not affect from the delay. The only thing required is $state_{\mathcal{A}}$ satisfies an appropriate property to the construction of first component in $addEdge_e$. In other words, from any procedure to construct paths, we can derive a dynamic-programming based algorithm for regular-language constrained shortest path problems, whenever construction of a path is expressed in terms of one-step extensions of a path, that is $p \mathbin{+\!\!+} [e]$.

6.2.3 Case 2: general case

As discussed, the Dijkstra algorithm improves efficiency very much. Now we propose a new efficient algorithm that works even if the graph has some negatively weighed edges.

We have derived the algorithms based on Theorem 5.9. These derivations suggest that we can derive an efficient algorithm even if we change weights. If we can redefine weights so that there are no negatively-weighted edges, then we can apply the Dijkstra algorithm. Though redefining weights may change shortest paths, we can do it without changing shortest paths, as shown by Johnson [Joh77]. One of the most important observations is the following lemma.

Lemma 6.3. *For a graph $G = (V, E)$, two weight functions w and w', assume that there exists a function $h : V \to \mathbb{R}$ such that $w'(u,v) \overset{\text{def}}{=} w(u,v) + h(u) - h(v)$ holds for all $(u,v) \in E$. Then shortest paths in the network (G, w) is the same as those in the network (G, w').* □

Thus, redefining the weight function according to h does not change the shortest paths. Furthermore, we can construct an appropriate h such that $w(u,v) + h(u) - h(v) \geq 0$ holds for all $(u,v) \in E$, by solving a shortest path problem. Consider a graph $G' = (V \cup \{v^*\}, E \cup \{(v^*, v) \mid v \in V\})$ and a weight function $w''(e) = \mathsf{if}\ e \in E\ \mathsf{then}\ w(e)\ \mathsf{else}\ 0$. Solve the single-source shortest path problem on the network (G', w'') from v^*, and let $h(v)$ be the optimal cost from v^* to v. Then, $w(u,v) + h(u) - h(v) \geq 0$ holds, because the optimality of $h(v)$ implies $w(u,v) + h(u) \geq h(v)$.

Now the calculation in Section 6.2.2 certainly shows that this redefinition does not affect the correctness of the derived algorithm. In summary, we can efficiently compute the regular-language constrained shortest path problems in the general setting by the following procedure. First, compute the special single-source shortest path problem, described above, to obtain the appropriate weights. After that, compute the regular-language constrained shortest path problems on the positively weighted graph, which is solved by the Dijkstra algorithm. The overall evaluation requires $\mathrm{O}(VE + VQ\log(VQ) + EQ)$.

It is worth noting that it is unnecessary to recompute special single-source shortest path problem, if we have already solved it. It is probable that we query many times using different regular languages. In such cases, precomputing the h improves efficiency.

6.3 Summary

We have derived several algorithms based on Theorem 5.9. Our theorem enables us to derive algorithms quite easily. Furthermore, our theorems clarify the correspondence between algorithms for usual shortest path problems and those for regular-language constrained shortest path problems.

The knowledge about usual shortest path problems enables us to solve regular-language constrained shortest path problems efficiently: the Dijkstra algorithm improves efficiency, and Johnson algorithm enables us to use the Dijkstra algorithm. The most important intuition from Theorem 5.9 is summarized as the following sentence. The good relationship between automata and $addEdge_e$ enables us to solve regular-language shortest path problems based on the dynamic programming technique, and the fact never depend the algorithm to find shortest path, whenever we construct a path by repetition of one-step extension. So we can consider the efficiency, namely the part to compute shortest paths, separately from the regular-language constraint part.

7 Related Works

In this paper, we formalized greedy algorithms and dynamic programming algorithms in terms of minimals and the strictly monotone property, and proposed new calculational laws that enable us to solve a large class of combinatorial optimization problems. Since dynamic programming is one of the most important techniques for constructing efficient algorithms, there are many researches to formalize it [Bel57, Iba73, Mor82, Hel89, dM92, BdM93a, dM95, BdM96, Cur96, Cur97, KV06] and automatically obtain it [ALS91, BPT92, SHTO00, GS02, GM02, LS03, SOH05]. Many researchers mention the importance of the strictly monotone property. If we have the strictly monotone property, then naive memoization enables us to produce a dynamic programming algorithm. Though obtaining the strictly monotone property is one of the most important parts to construct dynamic programming algorithms, most of the researches treat the strictly monotone property as an assumption. We concentrated to construct the strictly monotone property.

In Section 6 we derived algorithms for the regular-language constrained shortest path problems. Although many researches devoted themselves to deriving graph algorithms, for example [BdM93b, BvKW98, Rav99a, Rav99b] and especially shortest path algorithms [MR93, BvdEvG94, Dur02], it is known to be hard to derive graph algorithms systematically. Existing works were tried to derive basic algorithms, such as the Dijkstra algorithm. We have gone another way. We proposed the theorem that enables us to derive complicated dynamic programming algorithms from simple and basic algorithms.

We introduced Theorem 5.9 to derive dynamic programming algorithms. The programs derived from Theorem 5.9 may not be efficient. To improve their efficiency, we are considering three approaches. The first approach is to improve base greedy algorithms, as shown in Section 6.2. Theorem 5.9 derives a dynamic programming algorithm from a greedy algorithm, and to refine the greedy algorithm will improve the derived algorithm. The second one is to reduce the size of the table. The technique to reduce and minimize automata may be useful to reduce the size of the table. Matsuzaki [Mat07] also proposed a procedure to reduce the size of the table, which is applicable for maximum marking problems. The third one is to derive more efficient algorithms, such as greedy algorithms or parallel algorithms. Ibaraki [Iba78] showed that a class of dynamic programming problems corresponds to a class of problems that are solvable by a branch-and bound procedure. Matsuzaki [Mat07] showed that all maximum marking problems on binary trees are parallelizable.

8 Conclusion

As an ongoing effort to establish a methodology to construct efficient algorithms for combinatorial optimization problems, we showed the following three in this paper. First, we gave a framework to calculate combinatorial optimization problems using functions, minimals, and the strictly monotone property. Second, we introduced lemmas to construct orders satisfying the monotone properties, and proposed Theorem 5.9 that enables us to derive complicated dynamic programming algorithms from simple algorithms. Finally, we showed derivations of algorithms to solve regular-language constrained shortest path problems, as an application of Theorem 5.9.

Now we are considering two directions of further research.

For theory, we would like to give a general formalization that contains both catamorphisms and repetitions. In this paper, we independently showed the results about catamorphisms and repetitions, and these results are, hopefully, unified into a general framework. Furthermore, we would like to give a methodology to derive greedy algorithms. The strictly monotone property is also important for greedy algorithms. We hope that our result is helpful to derive greedy algorithms.

For application, we would like to construct a system to compute optimal path queries. In Section 6, we showed regular-language constrained shortest path problems are solved efficiently. There are other variants of shortest path problems, for example resource constrained shortest path problems [AAN83, Jaf84] and shortest path problems of time-dependent networks [OR90, IGSD98], and their combinations [BJ04, SJH06]. Such variants of shortest path problems, called optimal path problems, are important because they have a lot of practical applications. We believe Theorem 5.9 can cope with a wide class of them.

Acknowledgements

We are very grateful to Shin-Cheng Mu for his helpful comment about the relationship between our work and existing calculational works. His comment helped us very much to improve the whole of this paper.

References

[AAN83] Yash. P. Aneja, V. Aggarwal, and Kunhiraman Nair. Shortest chain subject to side constraints. *Networks*, 13(2):295–392, 1983.

[ALS91] Stefan Arnborg, Jens Lagergren, and Detlef Seese. Easy problems for tree-decomposable graphs. *Journal of Algorithms*, 12(2):308–340, 1991.

[BBJ$^+$02] Christopher L. Barrett, Keith Bisset, Riko Jacob, Goran Konjevod, and Madhav V. Marathe. Classical and contemporary shortest path problems in road networks: Implementation and experimental analysis of the TRANSIMS router. In *Algorithms - ESA 2002, 10th Annual European Symposium, Rome, Italy, September 17-21, 2002, Proceedings*, volume 2461 of *Lecture Notes in Computer Science*, pages 126–138. Springer, 2002.

[BBJ$^+$07] Christopher L. Barrett, Keith Bisset, Riko Jacob, Goran Konjevod, Madhav V. Marathe, and Dorothea Wagner. Label constrained shortest path algorithms: An experimental evaluation using transportation networks. Technical report, Virginia Tech (USA), Arizona State University (USA), and Karlsruhe University (Germany), 2007.

[BdM92] Richard S. Bird and Oege de Moor. Between dynamic programming and greedy: Data compression. Programming Research Group, 11 Keble Road, Oxford OX1 3QD, England, 1992.

[BdM93a] Richard S. Bird and Oege de Moor. From dynamic programming to greedy algorithms. In *Formal Program Development - IFIP TC2/WG 2.1 State-of-the-Art Report*, volume 755 of *Lecture Notes in Computer Science*, pages 43–61. Springer, 1993.

[BdM93b] Richard S. Bird and Oege de Moor. Solving optimisation problems with catamorphisms. In *Mathematics of Program Construction, 2nd International Conference, Oxford, U.K., June 29 - July 3, MPC 1992, Proceedings*, volume 669 of *Lecture Notes in Computer Science*, pages 45–69, 1993.

[BdM96] Richard S. Bird and Oege de Moor. *Algebra of Programming.* Prentice Hall, 1996.

[Bel57] Richard Bellman. *Dynamic Programming.* Princeton University Press, 1957.

[Ben86] Jon Bentley. *Programming Pearls.* ACM, New York, NY, USA, 1986.

[Bir84] Richard S. Bird. The promotion and accumulation strategies in transformational programming. *ACM Transactions on Programming Languages and Systems*, 6(4):487–504, 1984.

[Bir89] Richard S. Bird. Algebraic identities for program calculation. *Computer Journal*, 32(2):122–126, 1989.

[Bir01] Richard S. Bird. Maximum marking problems. *Journal of Functional Programming*, 11(4):411–424, 2001.

[Bir06] Richard S. Bird. Loopless functional algorithms. In *Mathematics of Program Construction, 8th International Conference, MPC 2006, Kuressaare, Estonia, July 3-5, 2006, Proceedings*, volume 4014 of *Lecture Notes in Computer Science*, pages 90–114. Springer, 2006.

[BJ04] Gerth Stølting Brodal and Riko Jacob. Time-dependent networks as models to achieve fast exact time-table queries. *Electronic Notes in Theoretical Computer Science*, 92:3–15, 2004.

[BJM00] Christopher L. Barrett, Riko Jacob, and Madhav V. Marathe. Formal-language-constrained path problems. *SIAM Journal on Computing*, 30(3):809–837, 2000.

[BPT92] Richard B. Borie, R. Gary Parker, and Craig A. Tovey. Automatic generation of linear-time algorithms from predicate calculus descriptions of problems on recursively constructed graph families. *Algorithmica*, 7(5&6):555–581, 1992.

[BvdEvG94] Roland Carl Backhouse, J. P. H. W. van den Eijnde, and A. J. M. van Gasteren. Calculating path algorithms. *Science of Computer Programming*, 22(1–2):3–19, 1994.

[BvKW98] Rudolf Berghammer, Burghard von Karger, and Andreas Wolf. Relation-algebraic derivation of spanning tree algorithms. In *Mathematics of Program Construction, MPC 1998, Marstrand, Sweden, June 15-17, 1998, Proceedings*, pages 23–43, London, UK, 1998. Springer-Verlag.

[CSRL01] Thomas H. Cormen, Clifford Stein, Ronald L. Rivest, and Charles E. Leiserson. *Introduction to algorithms, Second edition.* MIT Press, Cambridge, MA, USA, 2001.

[Cur96] Sharon A. Curtis. *A Relational Approach to Optimization Problems.* PhD thesis, Oxford University Computing Laboratory, 1996.

[Cur97] Sharon A. Curtis. Dynamic programming: a different perspective. In *Algorithmic Languages and Calculi, IFIP TC2 WG2.1 International Workshop on Algorithmic Languages and Calculi, 17-22 February 1997, Alsace, France*, volume 95 of *IFIP Conference Proceedings*, pages 1–23. Chapman & Hall, 1997.

[Cur03] Sharon A. Curtis. The classification of greedy algorithms. *Science of Computer Program*, 49(1–3):125–157, 2003.

[dM92] Oege de Moor. *Categories, Relations and Dynamic Programming.* PhD thesis, Oxford University Computing Laboratory, 1992. Technical Monograph PRG-98.

[dM95] Oege de Moor. A generic program for sequential decision processes. In *Programming Languages: Implementations, Logics and Programs, 7th International Symposium, PLILP'95, Utrecht, The Netherlands, September 20-22, 1995, Proceedings*, volume 982 of *Lecture Notes in Computer Science*, pages 1–23. Springer, 1995.

[dMG99] Oege de Moor and Jeremy Gibbons. Bridging the algorithm gap: A linear-time functional program for paragraph formatting. *Science of Computer Programming*, 35(1):3–27, 1999.

[dMG00] Oege de Moor and Jeremy Gibbons. Invited talk: Pointwise relational programming. In *Algebraic Methodology and Software Technology. 8th International Conference, AMAST 2000, Iowa City, Iowa, USA, May 20-27, 2000, Proceedings*, volume 1816 of *Lecture Notes in Computer Science*, pages 371–390. Springer, 2000.

[dMS01] Oege de Moor and Ganesh Sittampalam. Higher-order matching for program transformation. *Theoretical Computer Science*, 269(1–2):135–162, October 2001.

[Dur02] Juan Eduardo Durán. Transformational derivation of greedy network algorithms from descriptive specifications. In *Mathematics of Program Construction, 6th International Conference, MPC 2002, Dagstuhl Castle, Germany, July 8-10, 2002, Proceedings*, volume 2386 of *Lecture Notes in Computer Science*, pages 40–67. Springer, 2002.

[FFG06] Sergio Flesca, Filippo Furfaro, and Sergio Greco. Weighted path queries on semistructured databases. *Information and Computation*, 204(5):679–696, 2006.

[Fok92] Maarten M. Fokkinga. *Law and Order in Algorithmics.* PhD thesis, University of Twente, Dept INF, Enschede, The Netherlands, 1992.

[GM02] Robert Giegerich and Carsten Meyer. Algebraic dynamic programming. In *Algebraic Methodology and Software Technology, 9th International Conference, AMAST 2002, Saint-Gilles-les-Bains, Reunion Island, France, September 9-13, 2002, Proceedings*, volume 2422 of *Lecture Notes in Computer Science*, pages 349–364. Springer, 2002.

[GS02] Robert Giegerich and Peter Steffen. Implementing algebraic dynamic programming in the functional and the imperative programming paradigm. In *Mathematics of Program Construction, 6th International Conference, MPC 2002, Dagstuhl Castle, Germany, July 8-10, 2002, Proceedings*, volume 2386 of *Lecture Notes in Computer Science*, pages 1–20. Springer, 2002.

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

[Iba73] Toshihide Ibaraki. Solvable classes of discrete dynamic programming. *Journal of mathematical analysis and applications*, 43(3):642–693, 1973.

[Iba78] Toshihide Ibaraki. Branch-and-bound procedure and state-space representation of combinatorial optimization problems. *Information and Control*, 36(1):1–27, 1978.

[IGSD98] Irina Ioachim, Sylvie Gélinas, François Soumis, and Jacques Desrosiers. A dynamic programming algorithm for the shortest path problem with time windows and linear node costs. *Networks*, 31(3):193–204, 1998.

[Jaf84] Jeffery M. Jaffe. Algorithms for finding paths with multiple constraints. *Networks*, 14(1):95–116, 1984.

[Joh77] Donald B. Johnson. Efficient algorithms for shortest paths in sparse networks. *Journal of the ACM*, 24(1):1–13, 1977.

[KT05] Jon Kleinberg and Eva Tardos. *Algorithm Design*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2005.

[KV06] Jevgeni Kabanov and Varmo Vene. Recursion schemes for dynamic programming. In *Mathematics of Program Construction, 8th International Conference, MPC 2006, Kuressaare, Estonia, July 3-5, 2006, Proceedings*, volume 4014 of *Lecture Notes in Computer Science*, pages 235–252. Springer, 2006.

[LS03] Yanhong A. Liu and Scott D. Stoller. Dynamic programming via static incrementalization. *Higher-Order and Symbolic Computation*, 16(1–2):37–62, 2003.

[Mat07] Kiminori Matsuzaki. *Parallel Programming with Tree Skeletons*. PhD thesis, Graduate School of Information Science and Technology, The University of Tokyo, 2007.

[Mei92] Erik Meijer. *Calculating Compilers*. PhD thesis, Nijmegen University, 1992.

[MFP91] Erik Meijer, Maarten M. Fokkinga, and Ross Paterson. Functional programming with bananas, lenses, envelopes and barbed wire. In *Functional Programming Languages and Computer Architecture, 5th ACM Conference, Cambridge, MA, USA, August 26-30, 1991, Proceedings*, volume 523 of *Lecture Notes in Computer Science*, pages 124–144. Springer, 1991.

[MKHT06] Akimasa Morihata, Kazuhiko Kakehi, Zhenjiang Hu, and Masato Takeichi. Swapping arguments and results of recursive functions. In *Mathematics of Program Construction, 8th International Conference, MPC 2006, Kuressaare, Estonia, July 3-5, 2006, Proceedings*, volume 4014 of *Lecture Notes in Computer Science*, pages 379–396. Springer, 2006.

[Mor82] Thomas L. Morin. Monotonicity and the principle of optimality. *Journal of Mathematical Analysis and Applications*, 86:665–674, 1982.

[MPRS99] Ernesto Q. Vieira Martins, Marta Margarida B. Pascoal, Deolinda Maria L. Dias Rasteiro, and Jose Luis E. Santos. The optimal path problem. *Investigação Operacional*, 19:43–69, 1999.

[MR93] Bernhard Möller and Martin Russling. Shorter paths to graph algorithms. In *Mathematics of Program Construction, Second International Conference, Oxford, U.K., June 29 - July 3, 1992, Proceedings*, volume 669 of *Lecture Notes in Computer Science*, pages 250–268. Springer, 1993.

[OR90] Ariel Orda and Raphael Rom. Shortest-path and minimum-delay algorithms in networks with time-dependent edge-length. *Journal of the ACM*, 37(3):607–625, 1990.

[Rav99a] Jesús N. Ravelo. Relations, graphs and programs. Technical report, Programming Research Group Technical Monograph PRG-125, Oxford University Computing Laboratory, 1999.

[Rav99b] Jesús N. Ravelo. Two graph algorithms derived. *Acta Informatica*, 36(6):489–510, 1999.

[Rom88] Jean-François Romeuf. Shortest path under rational constraint. *Information Processing Letters*, 28(5):245–248, 1988.

[SdM01] Ganesh Sittampalam and Oege de Moor. Higher-order pattern matching for automatically applying fusion transformations. In *Proceedings of the Second Symposium of Programs as Data Objects, PADO'01*, volume 2053 of *Lecture Notes in Computer Science*, pages 218–237. Springer, 2001.

[SHTO00] Isao Sasano, Zhenjiang Hu, Masato Takeichi, and Mizuhito Ogawa. Make it practical: a generic linear-time algorithm for solving maximum-weightsum problems. In *Proceedings of the 5th ACM SIGPLAN International Conference on Functional Programming, ICFP'00*, pages 137–149, New York, NY, USA, 2000. ACM Press.

[SJH06] Hanif D. Sherali, Chawalit Jeenanunta, and Antoine G. Hobeika. The approach-dependent, time-dependent, label-constrained shortest path problem. *Networks*, 48(2):57–67, 2006.

[SOH05] Isao Sasano, Mizuhito Ogawa, and Zhenjiang Hu. Maximum marking problems with accumulative weight functions. In *Theoretical Aspects of Computing - ICTAC 2005, Second International Colloquium, Hanoi, Vietnam, October 17-21, 2005, Proceedings*, pages 562–578. Springer, 2005.

[VD05] Daniel Villeneuve and Guy Desaulniers. The shortest path problem with forbidden paths. *European Journal of Operational Research*, 165(1):97–107, 2005.

[YHT05] Tetsuo Yokoyama, Zhenjiang Hu, and Masato Takeichi. Calculation rules for warming-up in fusion transformation. In *the 2005 Symposium on Trends in Functional Programming, TFP 2005, Tallinn, Estonia, 23-24 September 2005*, pages 399–412, 2005.

[Yok06] Tetsuo Yokoyama. *Deterministic Higher-order Matching for Program Transformation.* PhD thesis, Graduate School of Information Science and Technology, The University of Tokyo, March 2006.