Make it Practical: A Generic Linear-Time Algorithm for Solving Maximum-Weightsum Problems

Isao Sasano   Zhenjiang Hu   Masato Takeichi
Department of Information Engineering
University of Tokyo
{sasano,hu,takeichi}@ipl.t.u-tokyo.ac.jp

Mizuhito Ogawa
NTT Communication Science Laboratories
mizuhito@theory.brl.ntt.co.jp

ABSTRACT

In this paper we propose a new method for deriving a practical linear-time algorithm from the specification of a maximum-weightsum problem: From the elements of a data structure x, find a subset which satisfies a certain property p and whose weightsum is maximum. Previously proposed methods for automatically generating linear-time algorithms are theoretically appealing, but the algorithms generated are hardly useful in practice due to a huge constant factor for space and time. The key points of our approach are to express the property p by a *recursive* boolean function over the structure x rather than a usual logical predicate and to apply program transformation techniques to reduce the constant factor. We present an *optimization theorem*, give a calculational strategy for applying the theorem, and demonstrate the effectiveness of our approach through several nontrivial examples which would be difficult to deal with when using the methods previously available.

Keywords: Maximum-weightsum problem, Linear-time algorithm, Program calculation, Mutumorphism, Fusion, Tupling

1. INTRODUCTION

Consider the following three optimization problems, the first of which appeared as an exercise in [12] and was discussed in [8].

> Professor McKenzie is consulting for the president of the A.-B Corporation, a company that has a hierarchical structure. That is, the supervisory relations form a tree rooted at the president. The personnel office has ranked each employee with a conviviality rating that is a positive or negative real number.
>
> **Party Planning Problem**
> The president wants to have a company party. To make the party fun for all attendees, the president does not want both an employee and his or her direct supervisor to attend.
>
> The problem is to design a linear algorithm making the guest list. The goal is to maximize the sum of the conviviality ratings of the guests.
>
> **Group Organizing Problem**
> The president wants to organize a project group in which the existing supervisory relations hold; that is, each member of a group (except the group leader) has in that group his or her direct or indirect supervisor.
>
> The problem is to design a linear algorithm making the member list. Again, the goal is to maximize the sum of the conviviality ratings of the group members.
>
> **Supervisor Chaining Problem**
> The president wants to award a chain of supervisors that has the largest sum of conviviality ratings.
>
> The problem is to design a linear algorithm making the list of supervisors.

These problems, though looking a bit different, can be formulated in the following uniform way: Given a tree t having nodes associated with weights, find a set of nodes vs that satisfies a certain property $p(t, vs)$ and has the maximum weightsum. That is,

$$spec\ t \quad = \quad \uparrow_{ws} / \ [vs \mid vs \leftarrow nodesets\ t,\ p\ (t, vs)]$$

where $ws\ vs$ computes the weightsum of vs and $nodesets$ generates all sets of nodes of a tree. The operator $\uparrow_f$ is called the selection operator [7] and is defined by

$$\begin{array}{rcll} a \uparrow_f b & = & a, & \textbf{if } f\ a > f\ b \\ & = & b, & \textbf{otherwise}. \end{array}$$

ICFP '00, Montreal, Canada.

The operator / is called the reduce operator [7] and is defined by

$$\oplus/[a_1, a_2, \ldots, a_n] = a_1 \oplus a_2 \oplus \cdots \oplus a_n.$$

The differences between the three problems are in the different definitions of the *property description* of p. The properties for the party planning problem, the group organizing problem, and the supervisor chaining problem are respectively denoted by p_{pp}, p_{go}, and p_{sc}, and are as follows.

- $p_{pp}(t, vs)$: for any v_1 and v_2 in vs, v_1 is not the parent or a child of v_2 in t.
- $p_{go}(t, vs)$: for any v_1 and v_2 in vs, either v_1 is an ancestor or a descendent of v_2 or there is in vs a common ancestor of v_1 and v_2.
- $p_{sc}(t, vs)$: for any v_1 and v_2 in vs, v_1 is an ancestor or a descendent of v_2, and vertices on the path between v_1 and v_2 are all included in vs.

We shall refer to these kinds of problems, the main subject of this paper, as *maximum-weightsum problems.* In general, a maximum-weightsum problem is to find, from the elements of a data structure, a subset which satisfies a certain property p and whose weightsum is maximum.

The maximum-weightsum problems are interesting in that they encompass a very large class of optimization problems [6, 11] (see Section 7 for some examples). The solving of maximum-weightsum problems, however, requires much insightful analysis. There are basically two kinds of approaches.

- *The Algebraic Approach.*

  Using the algebraic laws of programs [9, 8], one may try to calculate efficient solutions to the problems by program transformation. For instance, Bird derived a linear algorithm to solve the maximum segment sum problem [9], which is a maximum-weightsum problem on lists. Bird and de Moor [8] demonstrated the derivation of a greedy linear functional algorithm for the party planning problem.

  However, the success of derivation usually depends on a powerful calculation theorem and requires careful and insightful justification to meet the conditions of the theorem. For many cases, such justification is difficult for (even experienced) functional programmers to mimic to solve other similar problems.

- *The Construction-from-predicates Approach.*

  Though little known in functional programming community, it has been known for decades [1, 27, 6, 11] that if maximum-weightsum problems are specified by *regular predicates* [11], they are solvable in linear time on decomposable graphs and that linear-time algorithms for them can be derived automatically from the specifications of the graph problems.

  Though more systematic and constructive than the algebraic approach, this approach yields algorithms that suffer from a prohibitively large table (see Section 8 for details). The algorithm for solving the party planning problem, for instance, would need to construct a table with more than $2^{(2^{142})}$ entries [11]. The algorithms thus cannot be put to practical use.

In this paper we propose a new approach to deriving practical linear-time algorithms for maximum-weightsum problems over data structures such as lists, trees, and decomposable graphs. The key points of our approach are to express the property p by a *recursive* boolean function over the structure x rather than a usual logical predicate and to apply program transformation techniques to reduce the constant factor, thereby exploiting the advantages of the algebraic and construction-from-predicates approaches. Our main contributions can be summarized as follows.

- We propose an *optimization theorem* that gives a *generic* and *practical* linear-time algorithms for solving maximum-weightsum problems (Section 5). It provides a friendly interface for people deriving linear-time algorithms. Moreover, we give a calculational strategy for transforming the problem specification into the form to which the optimization theorem can be applied (Section 6). This is a transformation from a recursive boolean function into a property description in the form of mutumorphisms. This transformation utilizes tupling and fusion transformations.
- Our calculational approach is simple, general, and flexible. We demonstrate this in Section 7 by deriving linear-time algorithms for interesting and nontrivial maximum-weightsum problems. The Haskell codes for solving all the problems in this paper are available at http://www.ipl.t.u-tokyo.ac.jp/~sasano/mws.html.
- We are the first to successfully apply the *algebraic approach* to solving the huge-table problem appearing in the derivation of linear-time algorithms on decomposable graphs [1, 27, 6, 11], a problem not solved by table compression [6] or by dynamic table management [5]. This should be a significant step in making the theoretically appealing linear-time graph algorithms practically useful.

The organization of this paper is as follows. In Section 2 we describe our idea informally by using a list version of the party planning problem. After briefly reviewing the notational conventions and some basic concepts of program calculation in Section 3, we formally define maximum-weightsum problems in Section 4. We describe the optimization theorem in Section 5 and our calculation framework in Section 6. In Section 7 we highlight the features of our approach with several examples, and in Section 8 we discuss related work. We make our concluding remarks in Section 9.

2. A TOUR

Before addressing our approach formally, we briefly explain our idea by going through the list version of the party planning problem.

2.1 Specification

The list version of the party planning problem, which will be called the *maximum independent-sublist sum problem* (*mis* for short), is to compute vs, the set of elements from a non-empty list xs, such that no two elements in vs are adjacent in xs. Clearly, it is one of the maximum-weightsum problems on lists, which can be generally specified by

$$\begin{array}{lcl} mis & : & [\alpha] \to [\alpha] \\ mis\ xs & = & \uparrow_{ws} / \ [vs \mid vs \leftarrow subs\ xs,\ p_{mis}(xs, vs)] \end{array}$$

```haskell
mis :: [Elem] -> [MElem]
mis xs = let opts = mis' xs
         in getdata (foldr1 (bmax second)
                        [ (c,w,cand)
                        | (c,w,cand) <- opts,
                          c==2 || c==3])

mis' :: [Elem] -> [(Class,Weight,[MElem])]
mis' [x] = [(2,x,[(x,True)]), (3,0,[(x,False)])]
mis' (x:xs) =
  let opts = mis' xs
  in eachmax [ (table (marked mx) c,
                (if marked mx then weight mx else 0) + w,
                mx:cand)
             | mx <- [mark x, unmark x],
               (c,w,cand) <- opts]

bmax :: Ord w => (a -> w) -> a -> a -> a
bmax f a b = if f a > f b then a else b

eachmax :: (Eq c, Ord w) => [(c,w,a)] -> [(c,w,a)]
eachmax xs = foldl f [] xs
  where f [] (c,w,cand) = [(c,w,cand)]
        f ((c,w,cand) : opts) (c',w',cand') =
            if c==c' then
               if w>w' then (c,w,cand) : opts
               else opts ++ [(c',w',cand')]
            else (c,w,cand) : f opts (c',w',cand')
```

```haskell
type Weight = Int
type Elem = Weight
type MElem = (Elem,Bool)
type Class = Int

weight :: MElem -> Weight
weight (w,_) = w

marked :: MElem -> Bool
marked (_,m) = m

mark :: Elem -> MElem
mark x = (x,True)

unmark :: Elem -> MElem
unmark x = (x,False)

table :: Bool -> Class -> Class
table True 0 = 0
table True 1 = 0
table True 2 = 0
table True 3 = 2
table False 0 = 1
table False 1 = 1
table False 2 = 3
table False 3 = 3

second (_,x,_) = x
getdata (_,_,x) = x
```

Figure 1: A linear-time Haskell program for the mis problem.

where *subs* enumerates all sublists (not necessarily contiguous) of a list.

What is left is to define p_{mis}, the specific component of the problem. Because vs is a sublist of xs, we can specify p_{mis} in two steps: first marking the elements in xs which belong to vs, and then on the marked xs defining the property. Thus,

$$p_{mis}(xs, vs) = p\ (\mathit{marking}\ xs\ vs).$$

Here p checks that all pairs of marked elements are not adjacent in the marked xs.

$$\begin{array}{lcl} p & : & [\alpha] \to \mathit{Bool} \\ p\ [x] & = & \mathit{True} \\ p\ (x : xs) & = & \textbf{if}\ \mathit{marked}\ x \\ & & \textbf{then}\ \mathit{not}\ (\mathit{marked}\ (\mathit{hd}\ xs))\ \wedge\ p\ xs \\ & & \textbf{else}\ p\ xs \end{array}$$

$$\begin{array}{lcl} \mathit{hd} & : & [\alpha] \to \alpha \\ \mathit{hd}\ [x] & = & x \\ \mathit{hd}\ (x : xs) & = & x \end{array}$$

For later transformation, we define *hd* in the same way as p over the two cases singleton list and cons list. So much for our specification of the problem. It is worth noting that our specification is as natural as that in [11] using the monadic second-order logic (See Section 4.2). The critical difference is in specifying p by using a recursive function instead of a logical predicate. This enables us to use the functional program calculation for the later derivation.

2.2 Derivation

The derivation is based on our optimization theorem in Section 5, which says that if the property description p can be defined in mutumorphisms [15, 18, 17], then a linear-time algorithm solving the maximum-weightsum problem with respect to p can be derived automatically.

Therefore, the derivation of a linear-time algorithm for the mis problem reduces to be a derivation of mutumorphisms for p. More precisely, we hope to transform p to the following form:

$$\begin{array}{lcl} p\ [x] & = & \phi_1\ x \\ p\ (x : xs) & = & \phi_2\ x\ (p\ xs, p_1\ xs, \ldots, p_n\ xs) \end{array}$$

where ϕ_i's denote some functions and p_i's are auxiliary property descriptions defined in a fashion similar to that in which p is defined:

$$\begin{array}{lcl} p_i & : & [\alpha] \to \mathit{Bool} \\ p_i\ [x] & = & \phi_{i1}\ x \\ p_i\ (x : xs) & = & \phi_{i2}\ x\ (p\ xs, p_1\ xs, \ldots, p_n\ xs). \end{array}$$

Consider now the p for the mis problem. By introducing p_1 defined by

$$\begin{array}{lcl} p_1 & : & [\alpha] \to \mathit{Bool} \\ p_1\ [x] & = & \mathit{not}\ (\mathit{marked}\ x) \\ p_1\ (x : xs) & = & \mathit{not}\ (\mathit{marked}\ x) \end{array}$$

we can transform p to the following form.

$$\begin{array}{l} p\ [x] = \mathit{True} \\ p\ (x : xs) = \textbf{if}\ \mathit{marked}\ x\ \textbf{then}\ p_1\ xs\ \wedge\ p\ xs\ \textbf{else}\ p\ xs \end{array}$$

Applying our general optimization theorem for solving maximum-weightsum problems now soon yields a linear-time algorithm like that in Figure 1, where the algorithm is coded in Haskell. The function `bmax` in Figure 1 corresponds to the selection operator $\uparrow$. The main function `mis` takes a list as its argument and returns the input list with the selected elements marked with `True`. For example, `mis [1..4]` returns

`[(1,False),(2,True),(3,False),(4,True)].`

Note that our initial specification only returns `[2,4]`. The correctness of the algorithm follows from our optimization theorem (see Section 5). The linear property for `mis` comes from the following observation:

- The argument to `eachmax` has at most 8 elements, and so does the second argument of `f` used to define `eachmax`. So `eachmax` costs $O(1)$ time. Therefore, the auxiliary function `mis'` is a linear-time program.

- The `opts` in the body of `mis` has at most 4 elements, so it costs constant time to produce the final result after computing `mis' xs`.

The function `mis'` computes one optimal solution for each class, where classes correspond to elements of range of h defined as follows.

$$\begin{array}{lcl} h & : & [\alpha] \rightarrow (\mathit{Bool}, \mathit{Bool}) \\ h\ x & = & (p\ x,\ p_1\ x) \end{array}$$

Class `0` corresponds to $(\mathit{False}, \mathit{False})$, Class `1` to $(\mathit{False}, \mathit{True})$, Class `2` to $(\mathit{True}, \mathit{False})$, and Class `3` to $(\mathit{True}, \mathit{True})$. These classes can be interpreted as follows.

- Class `0` means that p does not hold and p_1 does not hold. This means that the head of the list is marked and the set of marked elements in the list is not independent.

- Class `1` means that p does not hold and p_1 holds. This means that the head of the list is not marked and the set of marked elements in the list is not independent.

- Class `2` means that p holds and p_1 does not hold. This means that the head of the list is marked and the set of marked elements in the list is independent.

- Class `3` means that p holds and p_1 holds. This means that the head of the list is not marked and the set of marked elements in the list is independent.

From the function h, we can automatically derive the definition of `table`. See Section 5 for details.

2.3 Remarks

Two remarks are worth making. First, the optimization theorem, which will be discussed in detail in Section 5, plays a significant role in our derivation. To apply this theorem, the only thing one have to do is to find the property description in mutumorphic form.

Second, the property description in mutumorphic form can be derived from a recursive property description by using the calculational strategy we present in Section 6. This derivation utilizes tupling and fusion transformations, which are nothing very special and for which a wealth of calculation techniques have been developed [18, 8]. Our derivation is thus surprisingly simple and powerful.

3. PRELIMINARIES

In this section we briefly review the notational conventions and some basic concepts of program calculation [7, 20, 8] used in this paper.

3.1 Recursive Data Types

To simplify the presentation and proof of the optimization theorem in Section 5, we restrict ourselves to considering polynomial data types. And to avoid categorical notations, we describe polynomial data types in the following form.

$$\begin{array}{lcl} D\ \alpha & = & C_1\ (\alpha, D_1, \ldots, D_{n_1}) \\ & | & C_2\ (\alpha, D_1, \ldots, D_{n_2}) \\ & | & \cdots \\ & | & C_k\ (\alpha, D_1, \ldots, D_{n_k}) \end{array}$$

Here D_i's denote $D\,\alpha$, and C_i's are called data constructors applying to an element of type α and a bounded number of recursive components. Though seemly restricted, these polynomial data types are powerful enough to cover our commonly used data types, such as lists, binary trees, rooted trees [6], and series-parallel graphs [23]. Moreover, other data types like the rose trees, a kind of regular datatype defined by

$$\mathit{RTree}\ \alpha = \mathit{Node}\ \alpha\ [\mathit{RTree}\ \alpha],$$

can be encoded into one of these polynomial data types. This will be demonstrated in Section 6.

For each data constructor C_i, we define F_i by

$$\mathsf{F}_i\ f\ (e, x_1, \ldots, x_{n_i}) = (e, f\ x_1, \ldots, f\ x_{n_i}).$$

3.2 Catamorphism

Catamorphisms, one of the most important concepts in program calculation [20, 22, 8], form a class of important recursive functions over a given data type. They are the functions that *promote through* the data constructors.

For example, for the type of lists, given e and $\oplus$, there exists a unique catamorphism *cata* satisfying the following equations.

$$\begin{array}{lcl} \mathit{cata}\ [\,] & = & e \\ \mathit{cata}\ (x : xs) & = & x \oplus (\mathit{cata}\ xs) \end{array}$$

In essence, this solution is a *relabeling*: it replaces every occurrence of $[\,]$ with e and every occurrence of $:$ with $\oplus$ in the cons list. Because of the uniqueness property of catamorphisms (i.e., for this example e and $\oplus$ uniquely determines a catamorphism over cons lists), we usually denote this catamorphism as $\mathit{cata} = (\![e \triangledown \oplus]\!)$.

DEFINITION 1 (CATAMORPHISM). *A catamorphism over a recursive data type D is characterized by*

$$f = (\![\phi_1, \ldots, \phi_k]\!)_D \ \equiv\ f \circ C_i = \phi_i \circ \mathsf{F}_i\, f \ \ (i = 1, \ldots, k)$$

If it is clear from the context, we usually omit the subscript D in $(\![\phi_1, \ldots, \phi_k]\!)_D$. □

Catamorphisms play an important role in program transformation (program calculation) because they satisfy a number of nice calculational properties in which the *fusion theorem* is of greatest importance:

THEOREM 1 (FUSION).

$$f \circ (\![\phi_1, \ldots, \phi_k]\!)_D = (\![\psi_1, \ldots, \psi_k]\!)_D$$

provided that for every i with $1 \leq i \leq k$

$$f \circ \phi_i = \psi_i \circ \mathsf{F}_i\, f.$$ □

The fusion theorem gives the condition that has to be satisfied in order to promote (fuse) a function into a catamorphism to obtain a new catamorphism. It actually provides a *constructive* but powerful mechanism for deriving a "bigger" catamorphism from a program in a compositional style, a typical style for functional programming. When applying the fusion theorem, we may do generalization, which is an operation substituting a new function for the target part of the fusion transformation.

3.3 Mutumorphisms

Mutumorphisms, generalizations of catamorphisms to mutually defined functions, are defined as follows [15, 16, 17].

DEFINITION 2 (MUTUMORPHISMS). *Functions f_1, f_2, ..., f_n are said to be mutumorphisms on a recursive data type $D\ \alpha$ if each function f_i is defined mutually by*

$$f_i \circ C_j = \phi_{ij} \circ \mathsf{F}_j\,(f_1 \vartriangle f_2 \vartriangle \ldots \vartriangle f_n)$$

for $j \in \{1, 2, \ldots, k\}$. □

Note that $f_1 \vartriangle f_2 \vartriangle \ldots \vartriangle f_n$ represents a function defined as follows.

$$(f_1 \vartriangle f_2 \vartriangle \ldots \vartriangle f_n)\ x = (f_1\ x, f_2\ x, \ldots, f_n\ x)$$

It is known that mutumorphisms can be turned into a single catamorphism by the tupling transformation [15, 16, 17].

THEOREM 2 (MUTU TUPLING). *If $f_1, f_2, \ldots, f_n$ are mutumorphisms like those in Definition 2, then*

$$f_1 \vartriangle f_2 \vartriangle \ldots \vartriangle f_n = (\!|\phi_1, \phi_2, \ldots, \phi_k|\!)_D$$

where $\phi_i = \phi_{1i} \vartriangle \ldots \vartriangle \phi_{ni}$ for $i = 1, \ldots, k$. □

4. MAXIMUM-WEIGHTSUM PROBLEMS

Before addressing our approach, we should be more precise about the maximum-weightsum problems and should clarify the limitations of the existing solutions.

4.1 A Formal Definition

A maximum-weightsum problem can be rephrased as follows. Given a data structure x, the task is to find a way to mark some elements in x such that the marked data structure x, say x^*, satisfies a certain property, say p, and the weightsum of the marked elements is maximum. Here the "maximum" means that no other marking of x satisfying p can produce a larger weight sum. A straightforward solution, whose complexity is exponential in the number of elements in x, is as follows.

$$\begin{array}{lcl} mws & : & (D\ \alpha^* \to Bool) \to D\ \alpha \to D\ \alpha^* \\ mws\ p\ x & = & \uparrow_{wsum} / \ [x^* | x^* \leftarrow gen\ x,\ p\ x^*] \end{array}$$

We use $gen\ x$ to generate all possible markings of input data x, and from those which satisfy the property p we use $\uparrow_{wsum} /$ to select one whose weightsum of marked elements is maximum.

Before defining gen and $wsum$, we explain some of our notation for marking. For a data type of α, we use α^* to extend α with marking information. It can be defined more concretely by

$$\alpha^* = (\alpha, Bool)$$

where a boolean value indicates whether or not the element of type α is marked. Accordingly, we use a^*, $b^*, \ldots, x^*$ to denote variables of the type α^* or the type $D\ \alpha^*$ (i.e., $D\ (\alpha^*)$).

The function gen, exhaustively enumerating all possible ways of marking or unmarking every element, can be recursively defined by

$$\begin{array}{l} gen\ :\ D\ \alpha \to [D\ \alpha^*] \\ gen = (\!|\eta_1, \ldots, \eta_k|\!) \\ \quad \textbf{where} \\ \qquad \eta_i\ (e, xs_1^*, \ldots, xs_{n_i}^*) = \\ \qquad\quad [\,C_i\ (e^*, x_1^*, \ldots, x_{n_i}^*)\ | \\ \qquad\qquad e^* \leftarrow [mark\ e, unmark\ e], \\ \qquad\qquad x_1^* \leftarrow xs_1^*, \\ \qquad\qquad x_2^* \leftarrow xs_2^*, \\ \qquad\qquad\quad \vdots \\ \qquad\qquad x_{n_i}^* \leftarrow xs_{n_i}^*] \quad (i = 1 \ldots k) \end{array}$$

where $mark$ and $unmark$ are functions for marking and unmarking:

$$\begin{array}{lcl} mark\ x & = & (x, True) \\ unmark\ x & = & (x, False). \end{array}$$

The function $wsum$ computes the sum of weights of marked elements in a data structure:

$$\begin{array}{l} wsum : D\ \alpha^* \to Weight \\ wsum = (\!|\phi_1, \ldots, \phi_k|\!) \\ \quad \textbf{where} \\ \qquad \phi_i\ (e^*, w_1, \ldots, w_{n_i}) = \\ \qquad\quad (\textbf{if}\ marked\ e^*\ \textbf{then}\ weight\ e^*\ \textbf{else}\ 0) + \\ \qquad\quad w_1 + \cdots + w_{n_i} \end{array}$$

where $marked$ is a function which takes as its argument an element e^* and checks whether e^* is marked.

$$\begin{array}{lcl} marked & : & D\ \alpha^* \to Bool \\ marked\ (e, m) & = & m. \end{array}$$

So much for the specification of mws, with which we can define various maximum-weightsum problems. For example, the mis problem in Section 2 can be specified by

$$mis = mws\ p$$

where p is the description of the "independent" property specified in Section 2.

4.2 Limitation of Borie's Approach

The maximum-weightsum problems have many instances in graph algorithms. Bern et al. showed that many graph problems — such as the minimum vertex cover problem, the maximum independent set problem, the maximum matching problem, and the traveling salesman problem — can be described by mws, and, more interestingly, that they can be solved in linear time on decomposable graphs [6]. Basing their work on Bern et al.'s and on Courcelle's [14], Borie et al. proposed a method for automatically constructing linear-time algorithms that solve the maximum-weightsum problems [11].

The main idea is to restrict the property p to be described in terms of a small canonical set of primitive predicates (such as the incident predicate $Inc\ (v, e)$) or a combination of them by logical operators ($\wedge$, $\vee$, and $\neg$) and (either first-order or second-order) quantifiers ($\forall$ and $\exists$). For example, the property p for the maximum independent set problem is described by

$$\begin{array}{lcl} p & = & \forall v_1 \ \forall v_2 \ \neg \ Adj\ (v_1, v_2) \\ Adj\ (v_1, v_2) & = & \exists e_1 \ (Inc\ (v_1, e_1) \ \wedge \ Inc\ (v_2, e_1)) \\ & & \wedge \ \neg \ (v_1 = v_2). \end{array}$$

With this restriction, it is possible to automatically construct a linear-time algorithm for *mws p* from the predicative structure of p.

Though attractive in theory, a linear-time algorithm needs to create a huge table, and this prevents these algorithms from actually being used [6, 11, 5]. The table created for the maximum independent set problem, for instance, contains more than $2^{(2^{142})}$ entries. The functional approach we propose can reduce this number to 8 (See Figure 1).

5. THE OPTIMIZATION THEOREM

This section focuses on a formal study of our functional approach to solving the maximum-weightsum problems, in which approach our main theorem, the *optimization theorem*, plays a significant role. It not only clarifies a sufficient condition for the existence of linear-time algorithms, but also gives a calculation rule for the construction of such algorithms. As will be seen later, the key points of our calculation are the functional (rather than predicative) structure of property descriptions, and the good use made of program transformation such as fusion and tupling [17, 16, 15]. We begin here by giving in Lemma 3 a sufficient condition under which *efficient* linear-time algorithms are guaranteed to exist. Then we use mutumorphisms to formalize in Lemma 4 the class of problems we can solve in linear time. Finally, we summarize the two lemmas in our optimization theorem.

5.1 A Sufficient Condition

It is obvious that not all optimization problems specified by

$$\begin{array}{lcl} spec & : & D\ \alpha \to D\ \alpha^* \\ spec & = & mws\ p \end{array}$$

can be solved in linear time. Our first result gives a sufficient condition for p in the form of the composition of a function and a catamorphism.

Lemma 3 (Optimization Lemma). *Let spec be defined by*

$$\begin{array}{lcl} spec & : & D\ \alpha \to D\ \alpha^* \\ spec & = & mws\ (accept \circ (\![\phi_1, \ldots, \phi_k]\!)). \end{array}$$

If the domain of the predicate accept is finite, then spec can be solved in linear time. □

To prove the lemma, we define an optimization function *opt* in Figure 2. We will show that *opt* solves *spec* correctly and that is computable in linear time. In Figure 2, *getdata* is a function which takes as its argument a triple and returns the third element.

$$getdata\ (c, w, r^*) = r^*$$

Correctness

Here we show how the right-hand side of

$$spec\ x = opt\ accept\ \phi_1\ \ldots\ \phi_k\ x$$

is transformed into the left-hand side. In the transformation rules we use the auxiliary functions ψ_i' $(i = 1, \ldots, k)$ defined by

$$\begin{array}{l} \psi_i'\ (e, cand_1, \ldots, cand_{n_i}) = \\ \quad [\,(\phi_i\ (e^*, c_1, \ldots, c_{n_i}), \\ \qquad (\textbf{if}\ marked\ e^*\ \textbf{then}\ weight\ e^*\ \textbf{else}\ 0) \\ \qquad + w_1 + \ldots + w_{n_i}, \\ \qquad C_i\ (e^*, r_1^*, \ldots, r_{n_i}^*))\ | \\ \qquad\qquad e^* \leftarrow [mark\ e, unmark\ e], \\ \qquad\qquad (c_1, w_1, r_1^*) \leftarrow cand_1, \cdots, \\ \qquad\qquad (c_{n_i}, w_{n_i}, r_{n_i}^*) \leftarrow cand_{n_i}]. \end{array}$$

$$\begin{array}{cl} & opt\ accept\ \phi_1 \ldots \phi_k\ x \\ = & \quad \{\ \text{unfold}\ opt\ \} \\ & getdata\ (\uparrow_{snd} /\ [\,(c, w, r^*) \\ & \qquad\qquad |\ (c, w, r^*) \leftarrow (\![\psi_1, \ldots, \psi_k]\!)\ x, \\ & \qquad\qquad accept\ c]\,) \\ = & \quad \{\ (\![\psi_1, \ldots, \psi_k]\!) = eachmax \circ (\![\psi_1', \ldots, \psi_k']\!)\ \} \\ & getdata\ (\uparrow_{snd} / \\ & \quad [\,(c, w, r^*)\,|\,(c, w, r^*) \leftarrow eachmax\ ((\![\psi_1', \ldots, \psi_k']\!)\ x), \\ & \qquad\qquad accept\ c]\,) \\ = & \quad \{\ filter\ (accept \circ fst) \circ eachmax = \\ & \qquad\qquad eachmax \circ filter\ (accept \circ fst)\ \} \\ & getdata\ (\uparrow_{snd} /\ (eachmax \\ & \qquad [\,(c, w, r^*)\,|\,(c, w, r^*) \leftarrow (\![\psi_1', \ldots, \psi_k']\!)\ x, \\ & \qquad\qquad accept\ c]\,)) \\ = & \quad \{\ \uparrow_{snd} / \circ\ eachmax = \uparrow_{snd} /\ \} \\ & getdata\ (\uparrow_{snd} /\ (eachmax \\ & \qquad [\,(c, w, r^*)\,|\,(c, w, r^*) \leftarrow (\![\psi_1', \ldots, \psi_k']\!)\ x, \\ & \qquad\qquad accept\ c]\,)) \\ = & \quad \{\ c = (\![\phi_1, \ldots, \phi_k]\!)\ r^*\ \} \\ & getdata\ (\uparrow_{snd} / \\ & \qquad [\,(c, w, r^*)\,|\,(c, w, r^*) \leftarrow (\![\psi_1', \ldots, \psi_k']\!)\ x, \\ & \qquad\qquad accept\ ((\![\phi_1, \ldots, \phi_k]\!)\ r^*)\,]\,) \\ = & \quad \{\ getdata \circ \uparrow_{snd} / = \uparrow_{wsum} / \circ\ map\ getdata\ \} \\ & \uparrow_{wsum} /\ [r^*\,|\,r^* \leftarrow (map\ getdata \circ (\![\psi_1', \ldots, \psi_k']\!))\ x, \\ & \qquad\qquad accept\ ((\![\phi_1, \ldots, \phi_k]\!)\ r^*)\,] \\ = & \quad \{\ (\![\eta_1, \ldots, \eta_k]\!) = map\ getdata \circ (\![\psi_1', \ldots, \psi_k']\!)\ \} \\ & \uparrow_{wsum} /\ [r^*\,|\,r^* \leftarrow (\![\eta_1, \ldots, \eta_k]\!)\ x, \\ & \qquad\qquad accept\ ((\![\phi_1, \ldots, \phi_k]\!)\ r^*)\,] \\ = & \quad \{\ \text{fold}\ mws\ \} \\ & mws\ (accept \circ (\![\phi_1, \ldots, \phi_k]\!))\ x \\ = & \quad \{\ \text{fold}\ spec\ \} \\ & spec\ x \end{array}$$

The first transformation rule is simply an unfolding of *opt*. The second transformation rule is

$$(\![\psi_1, \ldots, \psi_k]\!) = eachmax \circ (\![\psi_1', \ldots, \psi_k']\!).$$

The relation between ψ_i and ψ_i' is as follows.

$$\psi_i = eachmax \circ \psi_i'$$

The function $(\![\psi_1, \ldots, \psi_k]\!)$ applies the function *eachmax* to the list of candidates at each stage. The function $eachmax \circ (\![\psi_1', \ldots, \psi_k']\!)$, however, first creates all the markings of input data and corresponding weights and classes. Then it applies the function *eachmax*. These two functions compute the same candidate solutions because the function *eachmax* takes a list which contains candidate solutions and returns a list which consists of the rightmost optimal solution for each class in the input list, preserving the order, and the function *eachmax* is idempotent (i.e., $eachmax \circ eachmax = eachmax$).

$$
\begin{array}{l}
\mathit{opt}\ \mathit{accept}\ \phi_1 \ldots \phi_k\ x = \mathit{getdata}\ (\uparrow_{snd} /\ [\,(c, w, r^*) \mid (c, w, r^*) \leftarrow (\![\psi_1, \ldots, \psi_k]\!)_D\ x,\ \mathit{accept}\ c]\,) \\
\quad \mathbf{where}\ \psi_i\ (e, \mathit{cand}_1, \ldots, \mathit{cand}_{n_i}) = \\
\quad\quad \mathit{eachmax}\ [\,(\phi_i\ (e^*, c_1, \ldots, c_{n_i}), \\
\quad\quad\quad\quad (\mathbf{if}\ \mathit{marked}\ e^*\ \mathbf{then}\ \mathit{weight}\ e^*\ \mathbf{else}\ 0) + w_1 + \ldots + w_{n_i}, \\
\quad\quad\quad\quad C_i\ (e^*, r_1^*, \ldots, r_{n_i}^*)) \mid \\
\quad\quad\quad\quad\quad e^* \leftarrow [\mathit{mark}\ e, \mathit{unmark}\ e], \\
\quad\quad\quad\quad\quad (c_1, w_1, r_1^*) \leftarrow \mathit{cand}_1, \cdots, (c_{n_i}, w_{n_i}, r_{n_i}^*) \leftarrow \mathit{cand}_{n_i}] \qquad (i = 1, \ldots, k)
\end{array}
$$

Figure 2: Optimization function *opt*.

The third transformation rule

$$\mathit{filter}\ (\mathit{accept} \circ \mathit{fst}) \circ \mathit{eachmax} = \mathit{eachmax} \circ \mathit{filter}\ (\mathit{accept} \circ \mathit{fst})$$

means the commutativity between *filter* and *eachmax* functions, where *filter* is the abbreviation of

$$\lambda p.\ \lambda xs.\ [\,x \mid x \leftarrow xs,\ p\ x].$$

The function $\mathit{filter}\ (\mathit{accept} \circ \mathit{fst}) \circ \mathit{eachmax}$ first applies the function *eachmax* and then filters. The function $\mathit{eachmax} \circ \mathit{filter}\ (\mathit{accept} \circ \mathit{fst})$ first filters and then applies the function *eachmax*. These two functions compute the same result because the predicate $(\mathit{accept} \circ \mathit{fst})$ is concerned only with the classes and because the functions *filter* and *eachmax* preserve the order.

The fourth transformation rule is

$$\uparrow_{snd} / \circ\ \mathit{eachmax} = \uparrow_{snd} /.$$

This equation holds because $\uparrow_{snd} /$ returns the rightmost optimal solution, and *eachmax* returns a list which consists of the rightmost optimal solution for each class in the input list, preserving the order.

The fifth transformation rule is

$$c = (\![\phi_1, \ldots, \phi_k]\!)\ r^*.$$

This means that the class to which r^* belongs is c, which can be shown by induction.

The sixth transformation rule is

$$\mathit{getdata} \circ \uparrow_{snd} / = \uparrow_{wsum} / \circ\ \mathit{map}\ \mathit{getdata}.$$

This holds because the second element is the weightsum of the third element, which can be shown by induction.

The seventh transformation rule

$$(\![\eta_1, \ldots, \eta_k]\!) = \mathit{map}\ \mathit{getdata} \circ (\![\psi_1', \ldots, \psi_k']\!)$$

follows from Theorem 1.

The eighth and ninth transformation rules are simply the foldings of *mws* and *spec*. □

Linearity

Here we show that the function *opt* is linear.

$$
\begin{array}{l}
\mathit{opt}\ \mathit{accept}\ \phi_1 \ldots \phi_k\ x = \\
\quad \mathit{getdata}\ (\uparrow_{snd} / \\
\quad\quad [\,(c, w, r^*) \mid (c, w, r^*) \leftarrow (\![\psi_1, \ldots, \psi_k]\!)_D\ x,\ \mathit{accept}\ c]\,)
\end{array}
$$

The important observation is that the number of elements in the list $(\![\psi_1, \ldots, \psi_k]\!)_D\ x$ is bounded by the number of classes[1] (i.e., the number of elements in the domain of *accept* or the range of the catamorphism) because *eachmax* returns a list whose length is bounded by the number of classes. Therefore, if $(\![\psi_1, \ldots, \psi_k]\!)_D\ x$ can be computed in linear time, so can *opt*.

To prove that $(\![\psi_1, \ldots, \psi_k]\!)_D$ is a linear-time algorithm, it is suffice to show that $\psi_i\ e\ \mathit{cand}_1 \ldots \mathit{cand}_{n_i}$ can be computed in constant time. Taking the fact that ϕ_i can be computed in $O(1)$ time because the size of its argument is independent of the input, we can see that the construction of each triple (c, w, r^*) (i.e., each element consumed later by *eachmax*) can be computed in $O(1)$ time because c can be obtained in $O(1)$ time by ϕ_i, w by performing the $+$ operation n_i times, and r^* simply by combining $e^*, r_1^*, \ldots, r_{n_i}^*$. Here n_i is bounded by

$$N \;=\; \max\ \{n_i \mid 1 \le i \le k\}.$$

Moreover, exactly $(2 \times |\mathit{cand}_1| \times \ldots \times |\mathit{cand}_{n_i}|)$ triples of (c, w, r) are generated, and this number is bounded by the constant $2C^N$, where C is the number of classes. Therefore, the complexity of $\psi_i\ e\ \mathit{cand}_1 \ldots \mathit{cand}_{n_i}$ is $O(1)$. Though the size of the bound increases exponentially with N, in many well-used data types, such as lists and binary trees, N is small (perhaps one or two). □

Remarks

This lemma was inspired by Bern et al.'s work on *decomposable graphs* [6], where a similar condition on decomposable graphs was given. We generalize the idea from decomposable graphs[2] to generic recursive data types. Moreover, we give a concrete linear time algorithm using the optimization function *opt* in Figure 2.

In the optimization lemma, we give the optimization function *opt* to compute an *optimal* solution. It is worth noting that by slightly changing the definition of the optimization function *opt*, we can also do *recognition* and *enumeration* in linear time. Recognition means recognizing whether or not a solution satisfying the property description exists, and enumeration means counting the number of solutions satisfying the property description [11].

5.2 Decomposition Lemma

To apply the optimization lemma, we need to represent the property description p as $\mathit{accept} \circ (\![\phi_1, \ldots, \phi_k]\!)$, where the domain of *accept* is finite. The following lemma gives a method for transforming the property description p into the above form when p is defined using mutumorphisms.

[1]From the condition that the domain of *accept* is finite, we know the range of the catamorphism $(\![\phi_1, \ldots, \phi_k]\!)$ is finite. As seen in Section 2, this catamorphism projects the input to this finite range, whose elements are called *classes* [6].

[2]Decomposable graphs do not allow the addition of a new element when gluing smaller graphs.

LEMMA 4 (DECOMPOSITION LEMMA). *If the property description $p_0 : D\ \alpha^* \to Bool$ can be defined as mutumorphisms with other property descriptions $p_i : D\ \alpha^* \to Bool$ for $i = 1, \ldots, n$, then p_0 can be decomposed into*

$$p_0 = accept \circ (\![\phi_1, \ldots, \phi_k]\!),$$

where the domain of accept is finite. □

PROOF. To prove the lemma, we suppose that mutumorphisms p_0, $p_1, \ldots, p_n$ are defined as follows. For each $i \in \{0, 1, \ldots, n\}$ and each $j \in \{0, 1, \ldots, k\}$,

$$p_i \circ C_j = \phi_{ij} \circ \mathsf{F}_j\ (p_0 \vartriangle p_1 \vartriangle \cdots \vartriangle p_n).$$

By the mutu tupling theorem, we have

$$p_0 \vartriangle p_1 \vartriangle \ldots \vartriangle p_n = (\![\phi_1, \ldots, \phi_k]\!)_D \qquad (1)$$

where

$$\phi_i = \phi_{0i} \vartriangle \phi_{1i} \vartriangle \ldots \vartriangle \phi_{ni} \qquad (i = 1, \ldots, k).$$

By defining

$$accept\ (x_0, x_1, \ldots, x_n) = x_0,$$

we obtain

$$p_0 = accept \circ (\![\phi_1, \ldots, \phi_k]\!)_D.$$

Next we show that the domain of *accept*, *i.e.*, the range of $(\![\phi_1, \ldots, \phi_k]\!)_D$, is finite. Functions $p_0, p_1, \ldots, p_n$ have ranges which consist of two elements, *True* and *False*. So, from equation (1), the number of elements in the range of $(\![\phi_1, \ldots, \phi_k]\!)_D$ is 2^{n+1}, which is a finite number. This means that the domain of *accept* is finite. □

5.3 The Main Theorem

Combining the above two lemmas, we obtain the optimization theorem.

THEOREM 5 (OPTIMIZATION THEOREM). *The maximum-weightsum problem specified by*

$$\begin{array}{lcl} spec & : & D\ \alpha \to D\ \alpha^* \\ spec & = & mws\ p_0 \end{array}$$

can be solved in linear time if the property description $p_0 : D\ \alpha^* \to Bool$ can be defined as mutumorphisms with other property descriptions $p_i : D\ \alpha^* \to Bool$ for $i = 1, \ldots, n$. □

This theorem provides a friendly interface for the derivation of efficient linear-time algorithms solving the maximum-weightsum problems. When the property description is defined as mutumorphisms which consist of n property descriptions, a derived catamorphism will have range which has 2^n elements. In many cases, property description p can be easily described as mutumorphisms which consist of a small number of property descriptions (two to four for examples in this paper). For example, the party planning problem is described with mutumorphisms which consist of only two property descriptions. This means that the linear-time algorithms obtained are practical.

To see how the theorem works, recall the mis problem in Section 2. The property description p is defined with p_1 in the following mutumorphic form:

$$\begin{array}{l} p\ [x] = True \\ p\ (x : xs) = \textbf{if}\ marked\ x\ \textbf{then}\ p_1\ xs\ \wedge\ p\ xs\ \textbf{else}\ p\ xs \\ \\ p_1\ [x] = not\ (marked\ x) \\ p_1\ (x : xs) = not\ (marked\ x) \end{array}$$

It follows from the optimization theorem that an efficient linear-time algorithm solving the mis problem can be derived automatically.

It is easy to extend lemma 4 and theorem 5 to allow the property description p_0 to be defined as mutumorphisms with other functions whose ranges are finite.

6. CALCULATION STRATEGY

In this section we show how to derive practical linear-time algorithms that solve the maximum-weightsum problems by applying the optimization theorem.

Specification Specify the property description for a given maximum-weightsum problem using a recursive function p on a data structure R.

Step 1 If R is not a polynomial data type, find a polynomial data structure D into which R can be encoded, and then transform the property description p on R into p' on D. If the data structure R is a polynomial data structure, then do nothing. That is, let $p' = p$ and $D = R$.

Step 2 Derive p' in terms of mutumorphisms which consist of several property descriptions $p'_0, p'_1, \ldots, p'_n$ on D by generalizing some part in p' and fusing it. If necessary, do tupling transformation to get a catamorphism, since a catamorphism must be obtained when applying the fusion theorem.

Step 3 Apply the optimization theorem to get a linear-time algorithm that solves the maximum-weightsum problem.

We demonstrate these steps on the party planning problem.

The Party Planning Problem

Specification The inputs in the party planning problem are trees, so we can use the following recursive (regular) data structure.

$$Org\ \alpha ::= Leader\ \alpha\ [Org\ \alpha]$$

The specification of the problem is

$$\begin{array}{lcl} pp & : & Org\ \alpha \to Org\ \alpha^* \\ pp & = & mws\ p \end{array}$$

where the property description p of the party planning problem can be written as follows.

$$\begin{array}{l} p : Org\ \alpha^* \to Bool \\ p\ (Leader\ v\ [\,]) = True \\ p\ (Leader\ v\ (t : ts)) = \\ \qquad not\ (bothmarked\ v\ (getLeader\ t))\ \wedge \\ \qquad p\ t\ \wedge\ p\ (Leader\ v\ ts) \end{array}$$

If the tree contains only a single node, then the property is satisfied. Otherwise we check its root v_1 with its children one by one to make certain that both v_1 and its child are not marked at the same time, and then we check other parts of the tree recursively. Here *bothmarked* v_1 v_2 is used to check whether both v_1 and v_2 are marked, and *getLeader* returns the root of the organization tree:

$$\begin{array}{l} bothmarked : \alpha^* \to \alpha^* \to Bool \\ bothmarked\ v_1\ v_2 = marked\ v_1\ \wedge\ marked\ v_2 \end{array}$$

$$\begin{array}{l} getLeader : Org\ \alpha \to \alpha \\ getLeader\ (Leader\ v\ ts) = v. \end{array}$$

Notice that our initial specification is rather straightforward.

Step 1 The data type $Org\ \alpha$ is regular but nonpolynomial data type whose nodes can have arbitrarily many children. So we transform this data type $Org\ \alpha$ into a polynomial data type. In fact, we can represent the type $Org\ \alpha$ by the following binary tree structure, called a rooted tree [6].

$$\begin{array}{lcl} RTree\ \alpha & ::= & Root\ \alpha \\ & | & Join\ (RTree\ \alpha)\ (RTree\ \alpha) \end{array}$$

The relation between these two data types can be captured by the following functions.

$$\begin{array}{l} r2o : RTree\ \alpha \to Org\ \alpha \\ r2o\ (Root\ v) = Leader\ v\ [\,] \\ r2o\ (Join\ t_1\ t_2) = \textbf{let}\ Leader\ v\ ts = r2o\ t_2 \\ \qquad\qquad\qquad\quad \textbf{in}\ Leader\ v\ ((r2o\ t_1) : ts) \end{array}$$

$$\begin{array}{l} o2r : Org\ \alpha \to RTree\ \alpha \\ o2r\ (Leader\ v\ [\,]) = Root\ v \\ o2r\ (Leader\ v\ (t : ts)) = \\ \qquad Join\ (o2r\ t)\ (o2r\ (Leader\ v\ ts)) \end{array}$$

These two functions convert data types in linear time.

Next we transform the property description p on $Org\ \alpha$ to p' on $RTree\ \alpha$. Let p' and $getLeader'$ be the functions on $RTree\ \alpha$ which correspond to p and $getLeader$ on $Org\ \alpha$. These functions should satisfy the following equations.

$$\begin{array}{lcl} p' & : & RTree\ \alpha^* \to Bool \\ p'\ t & = & p\ (r2o\ t) \\ \\ getLeader' & : & RTree\ \alpha \to Bool \\ getLeader'\ t & = & getLeader\ (r2o\ t) \end{array}$$

A simple fusion calculation yields

$$\begin{array}{l} p'\ (Root\ v) = True \\ p'\ (Join\ t_1\ t_2) = \\ \quad not\ (marked\ (getLeader'\ t_1)\ \wedge \\ \qquad marked\ (getLeader'\ t_2))\ \wedge \\ \quad p'\ t_1\ \wedge\ p'\ t_2 \\ \\ getLeader'\ (Root\ v) = v \\ getLeader'\ (Join\ t_1\ t_2) = getLeader'\ t_2. \end{array}$$

Step 2 To represent p' using mutumorphisms which consist of only property descriptions, we generalize the part $marked \circ getLeader'$ and let it be lm'.

$$\begin{array}{lcl} lm' & : & RTree\ \alpha^* \to Bool \\ lm' & = & marked \circ getLeader' \end{array}$$

By a simple fusion calculation we can get

$$\begin{array}{lcl} lm'\ (Root\ v) & = & marked\ v \\ lm'\ (Join\ t_1\ t_2) & = & lm'\ t_2. \end{array}$$

Now we can represent p' using mutumorphisms which consist of the following two property descriptions.

$$\begin{array}{lcl} p'\ (Root\ v) & = & True \\ p'\ (Join\ t_1\ t_2) & = & not\ (lm'\ t_1\ \wedge\ lm'\ t_2)\ \wedge \\ & & p'\ t_1\ \wedge\ p'\ t_2 \\ lm'\ (Root\ v) & = & marked\ v \\ lm'\ (Join\ t_1\ t_2) & = & lm'\ t_2 \end{array}$$

Step 3 By applying our optimization theorem, we get the following linear-time algorithm.

$$\begin{array}{l} pp = r2o \circ (opt\ accept\ \phi_1\ \phi_2) \circ o2r \\ \quad \textbf{where} \\ \qquad accept\ (x_0, x_1) = x_0 \\ \qquad \phi_1\ v = (True, marked\ v) \\ \qquad \phi_2\ (a_1, b_1)\ (a_2, b_2) = \\ \qquad\qquad (not\ (b_1 \wedge b_2) \wedge a_1 \wedge a_2,\ b_2) \end{array}$$

So much for the derivation. Note that we can go further to compute ϕ_i statically and store it in a table, though this is not necessary. To do so, define four classes by

$$\begin{array}{l} c_0 = (False, False) \\ c_1 = (False, True) \\ c_2 = (True, False) \\ c_3 = (True, True) \end{array}$$

and simplify the functions $accept$, ϕ_1, and ϕ_2. We write ϕ_2 as a table.

$$\begin{array}{lcl} accept\ c & = & (c == c_2)\ \vee\ (c == c_3) \\ \phi_1\ v & = & \textbf{if}\ marked\ v\ \textbf{then}\ c_3\ \textbf{else}\ c_2 \end{array}$$

ϕ_2	c_0	c_1	c_2	c_3
c_0	c_0	c_1	c_0	c_1
c_1	c_0	c_1	c_0	c_1
c_2	c_0	c_1	c_2	c_3
c_3	c_0	c_1	c_2	c_1

7. FEATURES

In this section we highlight several important features of our calculational approach for solving the maximum-weightsum problems: its simplicity, generality, and flexibility.

7.1 Simplicity

First, as seen in the derivation of a linear-time algorithm for solving the party planning problem as well as seen in Section 2, our calculation is surprisingly simple. With the optimization theorem, all we need to do is derive a mutumorphic form of the property description. Fortunately, there are a lot of handy calculation strategies for deriving mutumorphisms [17], such as generalization of subexpressions to functions and fusion transformation [18, 8].

In the following we shall further illustrate this simplicity by solving the other two problems mentioned in the Introduction, whose derivation of linear algorithms are indeed not so straightforward for even an experienced functional programmer.

Group Organizing Problem

The inputs in the group organizing problem are trees, and we use the same data structure $Org\ \alpha$ that we used for the party planning problem. The property p on the marked tree for this problem is that one of the ancestor nodes of any two marked nodes must be marked. This can be described recursively by

$$
\begin{array}{l}
p : Org\ \alpha^* \rightarrow Bool \\
p\ (Leader\ v\ [\,]) = True \\
p\ (Leader\ v\ (t : ts)) = \\
\qquad \textbf{if}\ marked\ v\ \textbf{then}\ True \\
\qquad \textbf{else if}\ nm\ t\ \textbf{then}\ p\ (Leader\ v\ ts) \\
\qquad\qquad \textbf{else}\ p\ t\ \wedge\ nm\ (Leader\ v\ ts).
\end{array}
$$

The first equation means that a single-node tree always satisfies p. The second equation means that for a tree with the root v, if the root is marked, then any way of marking the nodes of its subtrees is acceptable. Otherwise, we have to check each of the subtrees to make sure that at most a single subtree has marked nodes. The function $nm\ t$ checks that the tree t has no marked nodes; it is defined as follows.

$$
\begin{array}{lcl}
nm\ (Leader\ v\ [\,]) & = & not\ (marked\ v) \\
nm\ (Leader\ v\ (t : ts)) & = & nm\ t\ \wedge\ nm\ (Leader\ v\ ts)
\end{array}
$$

Using the calculational strategy described in Section 6, we can turn these functions into the following mutumorphisms on $Rtree\ \alpha$, which consist of only property descriptions, and thus can obtain a linear-time algorithm for the group organizing problem.

$$
\begin{array}{lcl}
p' & : & RTree\ \alpha^* \rightarrow Bool \\
p'\ (Root\ v) & = & True \\
p'\ (Join\ t_1\ t_2) & = & \textbf{if}\ lm'\ t_2\ \textbf{then}\ True \\
 & & \textbf{else if}\ nm'\ t_1\ \textbf{then}\ p'\ t_2 \\
 & & \qquad \textbf{else}\ p'\ t_1\ \wedge\ nm'\ t_2 \\
 & & \\
nm'\ (Root\ v) & = & not\ (marked\ v) \\
nm'\ (Join\ t_1\ t_2) & = & nm'\ t_1\ \wedge\ nm'\ t_2
\end{array}
$$

Supervisor Chaining Problem

Similarly, for the supervisor chaining problem we can specify the property that the marked nodes form a path chain as follows.

$$
\begin{array}{l}
p\ : Org\ \alpha^* \rightarrow Bool \\
p\ (Leader\ v\ [\,]) = True \\
p\ (Leader\ v\ (t : ts)) = \\
\quad \textbf{if}\ marked\ v\ \textbf{then} \\
\qquad \textbf{if}\ nm\ t\ \textbf{then}\ p\ (Leader\ v\ ts) \\
\qquad \textbf{else}\ marked\ (getLeader\ t)\ \wedge\ p\ t\ \wedge \\
\qquad\qquad ol\ (Leader\ v\ ts) \\
\quad \textbf{else if}\ nm\ t\ \textbf{then}\ p\ (Leader\ v\ ts) \\
\qquad\quad \textbf{else}\ p\ t\ \wedge\ nm\ (Leader\ v\ ts)
\end{array}
$$

The function $ol\ t$ checks that the leader of the tree t is marked and the other nodes of t are not marked.

$$
\begin{array}{lcl}
ol\ (Leader\ v\ [\,]) & = & marked\ v \\
ol\ (Leader\ v\ (t : ts)) & = & nm\ t\ \wedge\ ol\ (Leader\ v\ ts)
\end{array}
$$

This property p can be expressed using the following mutumorphisms which consist of only property descriptions.

$$
\begin{array}{lcl}
p' & : & RTree\ \alpha^* \rightarrow Bool \\
p'\ (Root\ v) & = & True \\
p'\ (Join\ t_1\ t_2) & = & \textbf{if}\ lm'\ t_2\ \textbf{then} \\
 & & \quad \textbf{if}\ nm'\ t_1\ \textbf{then}\ p'\ t_2 \\
 & & \quad \textbf{else}\ lm'\ t_1\ \wedge\ p'\ t_1\ \wedge\ ol'\ t_2 \\
 & & \textbf{else if}\ nm'\ t_1\ \textbf{then}\ p'\ t_2 \\
 & & \qquad \textbf{else}\ p'\ t_1\ \wedge\ nm'\ t_2 \\
 & & \\
ol'\ (Root\ v) & = & marked\ v \\
ol'\ (Join\ t1\ t2) & = & nm'\ t_1\ \wedge\ ol'\ t2
\end{array}
$$

Now we can use our optimization theorem, as we did for the partying planning problem, to obtain a linear-time algorithm that solves the supervisor chaining problem.

7.2 Generality

Our approach is general (polytypic) enough to deal with maximum-weightsum problems on data structures that are not lists or trees. To illustrate this, we derive a linear-time algorithm solving the maximum two disjoint paths problem on series-parallel graphs [23].

The series-parallel graph is defined as follows.

$$
\begin{array}{lcl}
SPG & ::= & Base\ (Vert, Vert, Edge) \\
 & | & Series\ SPG\ SPG \\
 & | & Parallel\ SPG\ SPG
\end{array}
$$

Here $Vert$ represents the type of vertices and $Edge$ represents the type of edges. Every graph should have a single source and a single sink. The two data constructors are $Series$ and $Parallel$. $Series\ g_1\ g_2$ makes sense only when the sink of g_1 is the source of g_2, and $Parallel\ g_1\ g_2$ makes sense only when g_1 and g_2 share a source and a sink.

Figure 3 shows the meaning of the constructors. For example, the middle graph in Figure 3 is represented by

$$
g = Series\ (Base\ (u, w, e1))\ (Base\ (w, v, e2)).
$$

The maximum two disjoint paths problem is defined as follows: when given two vertices s and t, find two disjoint paths between the two vertices s, t such that the sum of the edge weights is maximum. Since the two disjoint paths between the vertices s and t can be seen as a cycle containing the vertices s and t, we can specify the property p of the problem by

$$
\begin{array}{lcl}
p & : & Vert \rightarrow Vert \rightarrow SPG \rightarrow Bool \\
p\ s\ t\ g & = & cycle\ g\ \wedge\ th\ s\ g\ \wedge\ th\ t\ g.
\end{array}
$$

Here $cycle$ judges whether the marked edges in the graph form a cycle, and $th\ v$ judges whether there is a marked edge incident to the vertex v. These two functions can be defined as follows.

$$
\begin{array}{lcl}
cycle\ (Base\ e) & = & not\ (marked\ e) \\
cycle\ (Series\ g_1\ g_2) & = & (cycle\ g_1\ \wedge\ nm\ g_2) \vee \\
 & & (nm\ g_1\ \wedge\ cycle\ g_2) \\
cycle\ (Parallel\ g_1\ g_2) & = & (cycle\ g_1\ \wedge\ nm\ g_2) \vee \\
 & & (nm\ g_1\ \wedge\ cycle\ g_2) \vee \\
 & & (span\ g_1\ \wedge\ span\ g_2) \\
 & & \\
th\ v\ g & = & anyMarked\ (inc\ v\ g)
\end{array}
$$

The function $span$ judges whether the marked edges in the graph span between the source and sink of the graph, nm

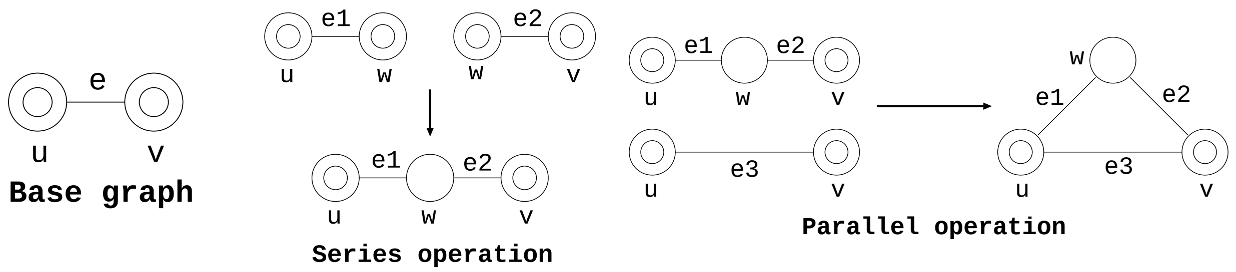

Figure 3: Operations of series-parallel graphs.

judges whether or not there are marked edges in the graph, *anyMarked* takes as its argument a list of edges es and judges whether or not there is a marked edge in es, and $inc\ v\ g$ gathers all the edges of g which are incident to the vertex v.

$$
\begin{array}{lcl}
span\ (Base\ e) & = & marked\ e \\
span\ (Series\ g_1\ g_2) & = & span\ g_1 \wedge span\ g_2 \\
span\ (Parallel\ g_1\ g_2) & = & (span\ g_1 \wedge nm\ g_2) \vee \\
 & & (nm\ g_1 \wedge span\ g_2) \\
 & & \\
nm\ (Base\ e) & = & not\ (marked\ e) \\
nm\ (Series\ g_1\ g_2) & = & nm\ g_1 \wedge nm\ g_2 \\
nm\ (Parallel\ g_1\ g_2) & = & nm\ g_1 \wedge nm\ g_2 \\
 & & \\
inc\ v\ (Base\ e@(v_1, v_2, \_)) & = & \textbf{if}\ v == v_1\ \vee\ v == v_2 \\
 & & \textbf{then}\ [e]\ \textbf{else}\ [\,] \\
inc\ v\ (Series\ g_1\ g_2) & = & inc\ v\ g_1 +\!\!+\, inc\ v\ g_2 \\
inc\ v\ (Parallel\ g_1\ g_2) & = & inc\ v\ g_1 +\!\!+\, inc\ v\ g_2
\end{array}
$$

By fusion calculation, we can get the following efficient recursive definition for th.

$$
\begin{array}{lcl}
th\ v\ (Base\ e@(v_1, v_2, \_)) & = & \textbf{if}\ v == v_1\ \vee\ v == v_2 \\
 & & \textbf{then}\ marked\ e\ \textbf{else}\ False \\
th\ v\ (Series\ g_1\ g_2) & = & th\ v\ g_1\ \vee\ th\ v\ g_2 \\
th\ v\ (Parallel\ g_1\ g_2) & = & th\ v\ g_1\ \vee\ th\ v\ g_2
\end{array}
$$

Now the property description $p\ s\ t$ is represented as mutumorphisms with property descriptions *cycle*, *span*, *nm*, $th\ s$, and $th\ t$. It follows from our optimization theorem that a practical linear-time algorithm can be obtained.

7.3 Flexibility

Here we explain flexibility of our derivation in coping with modification of the specifications of problems. We choose as our example the maximum segment sum problem, which is to compute the maximum of the sums of all segments (contiguous sublist) of a list. This is a quite well-known problem in the program calculation community [9]. In the following we will not only give a new solution to it but will also demonstrate that we can straightforwardly solve a set of related problems that would not be easily solved by using the previously available approaches.

The maximum segment sum problem is actually a maximum-weightsum problem where the property is that all marked elements in a list should be adjacent (connected). This property can be specified as follows.

$$
\begin{array}{lcl}
conn\ [x] & = & True \\
conn\ (x : xs) & = & \textbf{if}\ marked\ x\ \textbf{then} \\
 & & \quad nm\ xs \vee \\
 & & \quad (\underline{marked\ (hd\ xs)}\ \wedge conn\ xs) \\
 & & \textbf{else}\ conn\ xs
\end{array}
$$

If the list contains only a single element, then the property is satisfied. Otherwise, if the head is marked, then either none of the other elements are marked or all marked elements are connected to the head. If the head is not marked, the remaining list is checked recursively. As before, the function nm judges whether or not any of the elements are marked.

$$
\begin{array}{lcl}
nm\ [x] & = & not\ (marked\ x) \\
nm\ (x : xs) & = & not\ (marked\ x)\ \wedge\ nm\ xs
\end{array}
$$

To transform *conn* into mutumorphisms which consist only of property descriptions, we generalize the part *marked* $(hd\ xs)$ and let it be $mh\ xs$.

$$mh\ xs \;=\; marked\ (hd\ xs)$$

By a simple fusion calculation, we can easily get the following definition of mh.

$$
\begin{array}{lcl}
mh\ [x] & = & marked\ x \\
mh\ (x : xs) & = & marked\ x
\end{array}
$$

So we have obtained the following mutumorphisms which consist only of property descriptions.

$$
\begin{array}{lcl}
conn\ [x] & = & True \\
conn\ (x : xs) & = & \textbf{if}\ marked\ x\ \textbf{then} \\
 & & \quad nm\ xs\ \vee\ (mh\ xs\ \wedge conn\ xs) \\
 & & \textbf{else}\ conn\ xs
\end{array}
$$

Application of the optimization theorem gives us a linear-time algorithm. Although our linear algorithm may use a few more operations than that described by Bird [9], it is much easier to derive.

Now consider an extension of the maximum segment sum problem where we are interested only in those segments containing only even numbers. The property for this extended problem is

$$p\ xs = conn\ xs\ \wedge\ evens\ xs$$

where *evens* can be defined by

$$
\begin{array}{lcl}
evens\ [x] & = & \textbf{if}\ marked\ x\ \textbf{then}\ even\ (weight\ x) \\
 & & \textbf{else}\ True \\
evens\ (x : xs) & = & \textbf{if}\ marked\ x\ \textbf{then} \\
 & & \quad even\ (weight\ x) \wedge evens\ xs \\
 & & \textbf{else}\ evens\ xs
\end{array}
$$

It is easy to see that p can be defined as mutumorphisms with property descriptions *conn*, *evens*, *nm*, and *mh*, and thus that we obtain a linear algorithm for solving this extended problem.

Following this line, we can consider many similar segment problems. We can, for example, consider maximum sums of segments that have even numbers of elements by defining the following property

$$p\ xs = conn\ xs\ \wedge\ even\ (mnum\ xs),$$

where $mnum\, xs$ is the number of marked elements in xs. To get mutumorphisms which consist only of property descriptions, we generalize the part $even\ (mnum\ xs)$ and let it be $em\ xs$.

$$em\ xs = even\ (mnum\ xs)$$

After deriving a recursive form for em by using calculations similar to those described above, we can apply the optimization theorem to get a linear-time algorithm. It should be noted that deriving a linear-time algorithm for this problem requires some creativity when using the approach described in [9, 18] because the property p is not prefix-closed. This makes filter promotion difficult.

Finally, we mention that maximum segment sum problem can be generalized to trees: a segment in a list is simply generalized to a set of connected nodes in a tree. It should be easy for reader to derive a linear-time algorithm to solve this generalized problem.

8. RELATED WORK

Since the mid-1980s there have been several powerful approaches by dynamic programming, and many NP-complete problems have thus been reduced to linear-time problems for families of recursively constructed graphs. The pioneering work in this field was on series-parallel graphs [23].

The backbone concept is a class of graphs with bounded *tree-width* [21], which was independently developed as *partial k-tree* [2] and is also discussed in terms of *separators* [24] and *cliques* [19]. The set of graphs with tree-width at most k is equal to the set of k-terminal graphs constructed by algebraic composition rules [4]. NP-complete problems on graphs, such as the *Hamilton path problem*, are often linear in the size and beyond exponential in the tree width by the *divide-and-conquer* strategy according to this algebraic construction.

Much work has been done [10, 3], but most of it is on individual graph problems. Courcelle showed that whether a (hyper) graph with bounded tree width satisfies a closed monadic second order (MSOL, for short) formula (of graphs) is solvable in linear time [13]. Borie et al. further showed that the recognition, enumeration, and optimization problems specified by an extension of MSOL formula can be solved in linear time [11]. This graph variant of MSOL uses $Inc\ (v, e)$, which means a vertex v is an incident of an edge e, instead of the use of a successor function in ordinary MSOL [25].

Although appealing in theory, these methods are hardly useful in practice due to a huge constant factor for space and time. This arises from the manipulation of huge tables:

- The construction of tables reflects the decomposition of the property description into primitive ones; previous methods adapt fixed basic predicates as primitives, whereas our method freely adapts new primitives if they are in the form of mutumorphisms.

- Their construction of tables causes the exponential blow-up at each occurrence of quantifiers. Our method replaces the occurrences of quantifiers with recursions, which are computationally more efficient.

The comparison of description power between mutumorphisms and MSOL is not studied yet, which is a future work.

Bird calculated a linear-time algorithm for solving the maximum segment sum problem on lists [9], which is a kind of maximum-weightsum problem. Bird and de Moor studied optimization problems, which include maximum-weightsum problems, in a more general way that used relation calculus [8]. For instance, a greedy linear functional program for the party planning problem can be derived using relation calculus. Using relation calculus, they developed a very general framework to treat optimization problems. It is called *thinning theory*, and the *thinning theorem* plays the central role. But when applying the thinning theorem, one has to find two preorders which meet prerequisites of the thinning theorem, and this is sometimes difficult for non-experts. We instead focus on a useful class of optimization problems, maximum-weightsum problems, and propose a very simple way to derive efficient linear algorithms. When the target problem is a maximum-weightsum problem, the two preorders for the thinning theorem can be derived automatically by applying the optimization theorem. To apply our optimization theorem, the only thing one has to do is describe the property in the form of mutumorphisms. One significant reason for achieving this simplicity in derivation is that we have recognized the importance of the finiteness of the range of a catamorphism as in the optimization lemma, which received little attention in [8].

9. CONCLUSIONS

In this paper we present a new method for deriving practical linear-time algorithms for maximum weightsum problems on recursive data structures. From a specification represented as a functional program, our method obtains practical linear-time algorithms by deriving a catamorphism whose range size is small. Though our method does not guarantee that the derived catamorphism has a range with the smallest number of elements, in many cases it has range consisting of about 10 or 20 elements because the properties of many problems can be described in mutumorphisms which consist of three or four property descriptions. This means our derived linear-time algorithms are practical in many cases. Our method also enables linear time algorithms to be derived in a simple, systematic, and natural way, and it can flexibly cope with modification of the specification.

Though our current focus is on well-used recursive data structures such as lists and trees, our method should be able to deal with the graphs with bounded tree width because a class of the graphs with bounded tree width can be represented as a recursive data structure like that described in Section 3. Our next target problems are linear-time control flow analyses for structured procedural programs, for which the control flow graphs are known to have bounded tree width [26]. Though we currently restrict data types to polynomial data types, we think our optimization theorem can be extended to regular data types.

Our method of course has limitations, and the most fundamental is that tree width is bounded. The two-dimensional maximum segment sum problem, for example, cannot be dealt with by our method because a two-dimensional matrix corresponds to a grid graph, which will have unbounded tree width.

10. ACKNOWLEDGMENTS

This paper owes much to the thoughtful and helpful discussion comments made by Akihiko Takano, Hideya Iwasaki, and other Tokyo CACA members. Thanks to Jeremy Gibbons for his valuable comments on an earlier version of this paper, to Oege de Moor for providing us other interesting optimization problems, and to Richard Bird for kindly sending us his draft about solving maximum-weightsum problems using the thinning theorem. Thanks also to the anonymous referees for useful comments on an earlier version of this paper.

11. REFERENCES

[1] A. Arnborg and A. Proskurowski. Linear time algorithms for NP-hard problems on graphs embedded in k-trees. Technical Report TRITA-NA-8404, Royal Institute of Technology, Sweden, 1984.

[2] S. Arnborg. Complexity of finding embeddings in a k-tree. *SIAM Journal Algebraic Discrete Mathematics*, 8:277–287, 1987.

[3] S. Arnborg. Decomposable structures, Boolean function, representations, and optimization. In J. Wiedermann and P. Hájek, editors, *Mathematical Foundations of Computer Science 1995*, pages 21–36, 1995. Lecture Notes in Computer Science, Vol. 969, Springer-Verlag.

[4] S. Arnborg, B. Courcelle, A. Proskurowski, and D. Seese. An algebraic theory of graph reduction. *Journal of the Association for Computing Machinery*, 40(5):1134–1164, 1993.

[5] B. Aspvall, A. Proskurowski, and J. A. Telle. Memory requirements for table computations in partial k-tree algorithms. *Algorithmica*, 27(3):382–394, 2000. previously presented at SWAT'98.

[6] M. W. Bern, E. L. Lawler, and A. L. Wong. Linear-time computation of optimal subgraphs of decomposable graphs. *Journal of Algorithms*, 8:216–235, 1987.

[7] R. Bird. An introduction to the theory of lists. In M. Broy, editor, *Logic of Programming and Calculi of Discrete Design*, NATO ASI Series 36, pages 5–42. Springer-Verlag, 1987.

[8] R. Bird and O. de Moor. *Algebra of Programming.* Prentice Hall, 1996.

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

[10] H. Bodlaender. A tourist guide through treewidth. *Acta Cybernetica*, 11:11–21, 1993.

[11] R. B. Borie, R. G. Parker, and C. A. Tovey. Automatic generation of linear-time algorithms from predicate calculus descriptions of problems on recursively constructed graph families. *Algorithmica*, 7:555–581, 1992.

[12] T. H. Cormen, C. E. Leiserson, and R. L. Rivest. *Introduction to Algorithms.* The MIT Electrical Engineering and Computer Science Series. MIT Press, 1990.

[13] B. Courcelle. Graph rewriting: An algebraic and logic approach. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science*, volume B, chapter 5, pages 194–242. Elsevier Science Publishers, 1990.

[14] B. Courcelle. The monadic second-order logic of graphs. I. Recognizable sets of finite graphs. *Information and Computation*, 85(1):12–75, Mar. 1990.

[15] M. Fokkinga. Tupling and mutumorphisms. *Squiggolist*, 1(4), 1989.

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

[17] Z. Hu, H. Iwasaki, M. Takeichi, and A. Takano. Tupling calculation eliminates multiple data traversals. In *Proceedings of the 1997 ACM SIGPLAN International Conference on Functional Programming*, pages 164–175, Amsterdam, The Netherlands, 9–11 June 1997.

[18] J. Jeuring. *Theories for Algorithm Calculation.* Ph.D thesis, Faculty of Science, Utrecht University, 1993.

[19] I. Kříž and R. Thomas. Clique-sums, tree-decompositions and compactness. *Discrete Mathematics*, 81:177–185, 1990.

[20] E. Meijer, M. Fokkinga, and R. Paterson. Functional programming with bananas, lenses, envelopes and barbed wire. In *Proc. Conference on Functional Programming Languages and Computer Architecture (LNCS 523)*, pages 124–144, Cambridge, Massachusetts, Aug. 1991.

[21] N. Robertson and P. D. Seymour. Graph minors. II. Algorithmic aspects of tree-width. *Journal of Algorithms*, 7(3):309–322, Sept. 1986.

[22] T. Sheard and L. Fegaras. A fold for all seasons. In *Conference on Functional Programming Languages and Computer Architecture*, pages 233–242, 1993.

[23] K. Takamizawa, T. Nishizeki, and N. Saito. Linear-time computability of combinatorial problems on series-parallel graphs. *Journal of the Association for Computing Machinery*, 29:623–641, 1982.

[24] R. Thomas. A Menger-like property of tree-width: The finite case. *Journal of Combinatorial Theory, Series B*, 48:67–76, 1990.

[25] W. Thomas. Automata on infinite objects. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science*, volume B, chapter 4, pages 133–192. Elsevier Science Publishers, 1990.

[26] M. Thorup. All structured programs have small tree width and good register allocation. *Information and Computation*, 142:159–181, 1998.

[27] T. V. Wimer. *Linear Algorithms on k-terminal Graphs.* PhD thesis, Clemson University, 1987. Report No. URI-030.