Dynamic Programming as a Software Component

Oege de Moor
Programming Tools Group, Computing Laboratory
Oxford University
Wolfson Building, Parks Road, Oxford OX1 3QD
United Kingdom

Abstract: - Dynamic programming is usually regarded as a design technique, where each application is designed as an individual program. This contrasts with other techniques such as linear programming, where there exists a single generic program that solves all instances. From a software engineering perspective, the lack of a generic solution to dynamic programming is somewhat unsatisfactory. It would be much preferable if dynamic programming could be understood as a software component, where the ideas common to all its applications are explicit in shared code. In this paper, we argue that such a component does indeed exist, at least for a large class of applications in which the decision process is a sequential scan of the input sequence. We also assess the suitability of C++ for expressing this type of generic program, and argue that the simplicity offered by lazy functional programming is preferable. In particular, functional programs can be manipulated as algebraic expressions. The paper does not present any novel results: it is an introduction to recent work on the formalisation of algorithmic paradigms in software engineering.

Key-Words: - Dynamic programming; sequential decision process; software component; functional programming; algebra of programming; program derivation.

1 Introduction

In introductory courses on algorithm design, dynamic programming is usually taught by example. Students are presented with a number of typical applications, and given an informal statement of the principle of optimality. By doing a number of similar examples in exercises, they then acquire the skill to apply dynamic programming to other problems.

There exist numerous formalisations of dynamic programming that state the principle of optimality as a monotonicity condition, for example [17,18,20,23,29]. These works do however stop short of presenting a single, generic program that solves all applications of their formalisation at one stroke. The construction of such a generic program, and the appropriate means for expressing it, is the subject of this paper.

The quest for generic software components is one of the main goals of software engineering. It is now commonplace for programmers to use collections of such generic components (usually called *application frameworks*) in the construction of *e.g.* user interfaces, data bases, and messaging systems. In this paper, we would like to suggest that it might be possible to construct an application framework for the solution of optimisation problems as well. Such an application framework would harness much of the expertise now shared by a limited number of experts, and make their results available to a wider audience of programmers.

The structure of the paper is as follows. We start by presenting the class of problems to which our generic program applies – this includes many of the sequential decision processes as formalised by Karp and Held [20], but not all typical examples of dynamic programming. We then proceed to sketch the generic algorithm for solving that class of problems.

It may seem natural to code this type of generic algorithm as a C++ template, and we examine briefly how this may be achieved. There are a number of reasons, however, why C++ is not an appropriate medium for this type of generic program, and we argue that a lazy functional programming language such as Haskell is better suited to the task. Haskell programs can be regarded as algebraic expressions that can be manipulated to improve code quality.

Indeed, the generic program presented in this paper was discovered using an algebra of programs that generalises Haskell, by manipulating relations instead of functions. This generalisation allows one to deal with the nondeterminacy inherent in optimisation problems. Without going into detail, we sketch the characteristics of such a calculus for program derivation. Finally, we discuss related work, both in combinatorial optimisation and in software engineering. Some of this related work suggests further improvements to our software component.

This paper does not present any novel results. The generic algorithm was first presented in [24], and its formal development is a chapter in [4]. The purpose of this paper is to present these known results to researchers with an interest in dynamic programming, who are perhaps not so familiar with the terminology and notation of modern software engineering.

2 A Generic Problem

Our generic problem can be viewed as an instance of the generate-and-test paradigm: first generate all candidate solutions, then test for feasibility, and finally select a feasible solution that is optimal. In algorithm design, this is called the *exhaustive search* paradigm.

In most applications of dynamic programming, the generation of all candidate solutions is a decision process: at each step of the generation, one can make a choice between two (or more) alternatives. To formalise this, let us assume that the input is presented as a sequence of values

$$[a_1, a_2, ..., a_n] \ .$$

Generation of a candidate solution starts with a *seed* value, say e. At each step, there is a choice on how to add in the next item a_i. The first possibility is to use the binary operator $(\oplus)$, and the alternative is to use $(\otimes)$. A typical candidate solution will thus consist of a number of applications of $(\oplus)$, interleaved with applications of $(\otimes)$:

$$(((((e \ \oplus a_1) \oplus a_2\ ) \otimes a_3) \oplus a_4) \ ... \ ) \otimes a_n \ .$$

The set of all candidate solutions is obtained by interleaving the two operators in all possible ways. It is obvious how this could be generalised to more than two choices, but for simplicity we shall confine ourselves to just two. It follows that the generation of candidate solutions is characterised by three quantities: the seed value e, and the two binary operators $(\oplus)$ and $(\otimes)$.

To also specify the feasibility test, we shall need a feasibility predicate, which we shall call p.

Finally, we need to define the selection of an optimal element. Usually this is specified through an objective function (which is to be minimised or maximised), but here we shall specify it through a comparison relation R that is a preorder. (A *preorder* is a binary relation that is reflexive and transitive.) This formulation will make it easier to pin down the applicability conditions of our algorithm. An optimal element from the set of feasible solutions is one that is minimal in the preorder R.

Summing up, our generic problem is given by a generator that is a decision process (e, $(\oplus)$ and $(\otimes)$), a feasiblity predicate p, and a preorder R. Provided these five quantities satisfy certain conditions, which amount to the principle of optimality, our generic algorithm will provide an efficient method of computing an optimal feasible solution.

Before proceeding to that generic algorithm, we first consider some specific instances of the generic problem.

2.1 Example: knapsack

In the 0/1 knapsack problem, the sequence of input values consists of (value,weight) pairs, which we shall call *items*. A solution is a selection of items, which we also represent by a (value, weight) pair. The seed solution is the empty selection, whose value and weight are both 0:

$$e = (0,0) \ .$$

For each item (v, w), we have the choice of including it in a candidate solution (tv, tw), or to leave it out. Our two binary operators are therefore defined as follows:

$$(tv, tw) \oplus (v, w) = (tv+v, tw+w)$$
$$(tv, tw) \otimes (v, w) = (tv, tw) \ .$$

The feasibility predicate is the test whether the total weight has not exceeded a given capacity c:

$$p\ (tv,tw) = (tw \leq c\ )$$

Finally, one solution is better than another if its total value is greater, so the preorder R is given by:

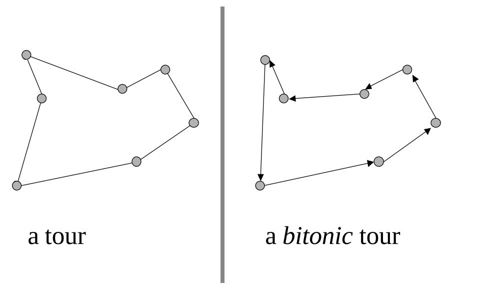
$$(tv_1,tw_1)\ R\ (tv_2,tw_2) \quad = \quad tv_1 \geq tv_2 \ .$$

2.2 Example: bitonic tours

The bitonic tours problem, due to J. L. Bentley, is a variation on the well-known Euclidean travelling salesman problem. Given is a set of vertices in the plane, and it is required to find a cyclic path that visits each point exactly once. Such a path is called a *tour*. The general problem of computing a shortest tour is NP-complete, so we simplify the problem by restricting our attention to *bitonic* tours. A tour is bitonic if it starts at the leftmost point, goes strictly left to right to the rightmost point, and it goes strictly right to left back to the starting point. The problem is to compute a shortest bitonic tour; we may assume that no two vertices have the same x-coordinate. This formulation of the problem is adapted from [11].

To illustrate, Fig. 1 shows a set of points with an shortest tour on the left, and a shortest bitonic tour on the right.

Fig. 1: bitonic tours



We assume that the points $v_1, v_2, ..., v_n$ are sorted by x-coordinate, so the generation of candidate solutions will proceed by a left-to-right plane sweep.

A bitonic tour can be represented as a pair of two disjoint sequences, one for each uni-directional part:

$$([a_1, a_2, ..., a_n], [b_1, b_2, ...,b_m]) \ .$$

Of course we do not wish to generate the same bitonic tour twice, so we stipulate that the last vertex a_n of the first component is the rightmost vertex. When this rule is applied throughout generation of tours, each bitonic tour has only one representation.

The smallest bitonic tour consists of just two elements, namely the two leftmost vertices. It follows that the seed is defined by

$$e = ([v_2], [v_1]) \ .$$

For each vertex v_i with $i > 2$, we have the choice between including it on either part of the tour:

$$([a_1, a_2, ..., a_n], [b_1, b_2, ...,b_m]) \oplus v_i$$
$$=$$
$$(([a_1, a_2, ..., a_n, v_i], [b_1, b_2, ...,b_m]) \ ,$$

and

$$([a_1, a_2, ..., a_n], [b_1, b_2, ...,b_m]) \otimes v_i$$
$$=$$
$$([b_1, b_2, ...,b_m, v_i], [a_1, a_2, ..., a_n]) \ .$$

Note the reversal of the components in the definition of ($\otimes$): this is to enforce the requirement that the rightmost vertex is always on the first component of a tour.

Any bitonic tour is feasible, so the predicate p always returns true:

$$p\ (x,y) = \text{true} \ .$$

We could have chosen the generator and the feasibility predicate so that all tours get generated, and then pick out those that are bitonic. That model of the problem would however not lead to an efficient solution.

An optimal bitonic tour is one whose length is as small as possible. The *length* of a bitonic tour is the sum of the path lengths of its two components, plus the length of the two edges that connect their leftmost and rightmost vertices. We can thus define the preorder R by

$$(x_1, y_1)\ R\ (x_2, y_2)$$
$$=$$
$$length(x_1, y_1) \leq length(x_2,y_2) \ .$$

It is worthwhile to reflect for a moment on the difference between the models of the bitonic tours problem and of the knapsack problem. In bitonic tours, we decided to model the problem of finding an optimal tour itself, rather than its length. This is just a matter of convenience: in order to compute the length, we need at least the last elements of both components of a tour, so it is easier to explain the model for finding tours rather than just lengths. As we shall see, the generic algorithm is not significantly more complicated when we wish to find the optimal object itself.

2.3 Example: bus stop placement

Our final example is an exercise from one of the standard texts on dynamic programming [14]: “A long one-way street consists of m blocks of equal length. A bus runs uptown from one end of the street to the other. A fixed number n of bus stops are to be located so as to minimise the total distance walked by the population. Assume that each person taking an uptown bus trip walks to the nearest bus stop, gets on the bus, rides, gets off at the stop nearest his destination, and walks the rest of the way. During the day, exactly B_j people from block j start uptown bus trips, and C_j complete uptown bus trips at block j. Write a program that finds an optimal location of bus stops.”

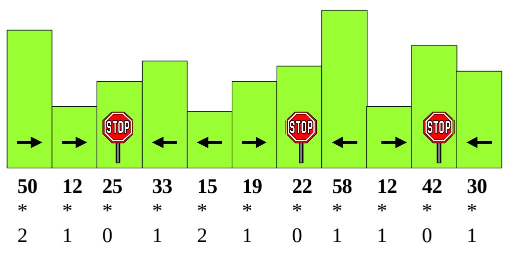
The input can be represented as a sequence

$$[B_1 + C_1, B_2 + C_2, ..., B_m + C_m]$$

Each item $B_j + C_j$ represents the number of people passing through block j.

We assume that bus stops are to be placed in front of blocks (and not at block boundaries). Furthermore, if there is a bus stop in front of a block, people passing through that block do not need to walk at all. As an example, consider the situation depicted in Fig. 2.

Fig. 2: bus stops



The blocks are represented by vertical bars, and the number (in bold face) under each bar is the number of people passing through that block. Three bus stops have been placed, and arrows indicate the direction people walk to get to a bus stop. For example, to get from the first block to the bus stop, 50 people have to walk two blocks. The bottom row of numbers indicates the distance to the nearest bus stop. By taking the dot product of the two rows of numbers, we obtain the total cost of the arrangement shown.

We can represent a placement of bus stops as a partition of the input sequence, with a bus stop at the beginning of each partition component except the first. For this representation to be correct, we need to assume the existence of a fictitious block at the beginning of the street, with 0 people passing through it. The arrangement in Fig. 2 is thus represented by:

$$\begin{array}{l} [\ [0, 50, 12], \\ \ \ [25, 33, 15, 19], \\ \ \ [22, 58, 12], \\ \ \ [42, 30]\ ]\ . \end{array}$$

The role of 0 at the beginning is to allow the placement of a bus stop at the first block in the street.

Readers who are not accustomed to the data structuring facilities of modern programming languages may wish to contemplate this representation somewhat further. A partition is a sequence, whose components are lists themselves. It would of course be possible to code this representation purely in terms of indices, but the intended meaning would be much less perspicuous.

How do we generate all possible placements by a decision process? The starting point is the seed value e, representing a fictitious building at the beginning of the street:

$$e = [\ [0]\ ]\ .$$

Note that e is a list with one element (namely [0]), and that element itself is a list containing 0.

Given a block (that is the number of people passing through it), we can decide either to place a bus stop, or not. Placing a bus stop means starting a new partition component, whereas leaving out a bus stop corresponds to gluing the new block at the end of an existing partition component. This leads us to the following definitions of the two choice operators:

$$\begin{array}{ll} & [\ [0, b_1, b_2, ..., b_{i1}], \\ & \ \ [b_{i1+1}, ..., b_{i2}], \\ & \ \ ... , \\ & \ \ [b_{ik+1}, ..., b_{ij}]\ ]\ \oplus b_{ij+1} \\ = & \\ & [\ [0, b_1, b_2, ..., b_{i1}], \\ & \ \ [b_{i1+1}, ..., b_{i2}], \\ & \ \ ... , \\ & \ \ [b_{ik+1}, ..., b_{ij}], \\ & \ \ [b_{ij+1}]\ ] \end{array}$$

and

$$\begin{array}{ll} & [\ [0, b_1, b_2, ..., b_{i1}], \\ & \ \ [b_{i1+1}, ..., b_{i2}], \\ & \ \ ... , \\ & \ \ [b_{ik+1}, ..., b_{ij}]\ ]\ \otimes b_{ij+1} \\ = & \\ & [\ [0, b_1, b_2, ..., b_{i1}], \\ & \ \ [b_{i1+1}, ..., b_{i2}], \\ & \ \ ... , \\ & \ \ [b_{ik+1}, ..., b_{ij}, b_{ij+1}]\ ]\ . \end{array}$$

Recall that we should place exactly n bus stops. Obviously taking more bus stops would decrease the total amount of walking, so the problem is unaltered if we only require that there are *at most* n bus stops. Accordingly we define the feasibility predicate by

$$p\ ([x_1, x_2, ..., x_k]) = (k \leq n{+}1)\ .$$

That is, there should be at most n+1 components in a partition, because each component x_i represents a bus stop, except x_1. The advantage of this definition over one where ($\leq$) is replaced by (=) will become clear later. Briefly, we require that feasible solutions to the whole problem are composed of feasible solutions to sub-problems. That property would not be satisfied if ($\leq$) were replaced by (=).

We omit a formal definition of the function *walk* that returns the total walking cost of a particular placement; its definition should be clear from the example, and the details are not illuminating. One placement *xs* is preferable over another placement *ys* if its total walking cost is less. The preorder R in our generic problem statement is therefore defined:

$$xs\ R\ ys \ = \ (walk(xs) \leq walk(ys))\ .$$

3 A Generic Solution

An instance of our generic problem is specified by a decision process (the seed e and two operators $(\oplus)$ and $(\otimes)$), a feasibility predicate p, and a preorder R. Our aim is to give an algorithm that solves the generic problem, in terms of these five parameters, as well as some others whose existence is implied by the principle of optimality.

Of course such a generic solution should satisfy certain criteria in order to be acceptable. First, for each instance, its asymptotic time complexity should be as good as a specialised program derived by traditional methods. Second, all the interesting processing should happen in the statement of the algorithm, and not in its parameters. In particular, we shall require that all parameters are (amortised) constant time operations.

It would not be too difficult to give generic code for dynamic programming that takes the functional equations, derived by the programmer, as a parameter, and only performs the task of tabulating intermediate solutions. We do not consider that an acceptable solution, for the derivation of functional equations is in itself a non-trivial task. Thus the use of our generic algorithm does not require the programmer to record the functional equations explicitly: all she needs to do is to verify a number of monotonicity conditions. Below we first present these conditions, and then proceed to describe how they are exploited in the algorithm.

These are the applicability conditions of our algorithm: there exist a predicate q, a preorder S and a total preorder T such that the following are satisfied

1. $p(e)$ holds, and $p\ (x \oplus a) = q\ (x \oplus a) \wedge p(x)$,
2. $x\ R\ y$ and $x\ S\ y$ and $q\ (y \oplus a)$ implies
   $(x \oplus a)\ R\ (y \oplus a)$ and
   $(x \oplus a)\ S\ (y \oplus a)$ and $q(x \oplus a)$,
3. $x\ T\ y$ implies $(x \oplus a)\ T\ (y \oplus a)$,
4. conditions 1-3 also hold when operator $(\oplus)$ is replaced by $(\otimes)$.

The first condition says that feasible solutions are composed of feasible solutions to sub-problems. Clearly if this condition is satisfied at all, we could take q to be p itself. However, it often happens that we can chose a test for q that is computationally less expensive than p.

Condition (2) is a *dominance criterion*. It says that if x is better than y (that is $x\ R\ y$ and $x\ S\ y$), then the extension of x is better than the extension of y; furthermore, the extension of x is a feasible solution if the extension of y is. The use of dominance criteria is common in formalisations of dynamic programming, and they are also the basis of a well-known strategy for speeding up naive dynamic programming algorithms [15,16,30].

The third condition states that the operators are monotonic on T. The role of T will be to order intermediate solutions in a table; this condition implies that we do not need to re-order existing elements after they have been updated.

Our algorithm keeps a collection of feasible solutions in a list L. The list $L = [x_1, x_2, ..., x_k]$ is ordered so that consecutive elements are related by the preorder T:

$$x_1\ T\ x_2\ \ T\ ...\ T\ \ x_k\ \ .$$

Initially the list L contains only one element, namely the seed $[e]$. The first applicability condition states that e is feasible.

We then scan the sequence of input items from left to right. For each item a, we compute two new lists:

$$L_1 = [x_1 \oplus a, x_2 \oplus a, ..., x_k \oplus a]$$

and

$$L_2 = [x_1 \otimes a, x_2 \otimes a, ..., x_k \otimes a]\ .$$

Because of the third condition, both of these lists are ordered with respect to T. Merging L_1 and L_2 gives a new list, still ordered with respect to T. The merging operation is well-defined because T is assumed to be total. The merged list may contain a number of infeasible elements. The first condition, together with the fact that all elements of L were feasible, tells us that we can test elements of the merged list for feasibility using q. Removing the infeasible elements gives a new ordered list of feasible solutions, which we shall name L'.

In principle we could continue by considering the next item on the input, but doing so would be wasteful: the dominance criterion gives the opportunity of substantially reducing the size of L'. To do so, we compare adjacent elements in L', and discard any y for which there exists a neighbour x such that both $x\ R\ y$ and $x\ S\ y$. We call this step *squeezing*. Finally, after squeezing, we obtain our new list L, and we can consider the next item in the input sequence.

Upon termination of the input processing, we return an element of L that is minimal in the preorder R.

It is worthwhile noting that the choice of T has a big impact on the effectiveness of squeezing, because we only compare adjacent elements. Although none of the conditions relates T to R and

S, it is usually a simple combination of these two preorders. The idea of implementing dynamic programming through merging and squeezing is due to Ahrens and Finke [1], who applied it to the knapsack problem.

3.1 Example: knapsack

Let us now examine appropriate choices for q, S and T in the knapsack problem. The feasibility predicate p is

$$p\ (tv,tw) = (tw \leq c)\ .$$

This is cheap to evaluate, so we can take q to be p itself. The first condition is satisfied, because the capacity c is non-negative, and the weights of individual items are also non-negative.

To satisfy the dominance criterion, we need a preorder S so that

$$(tv_1, tw_1)\ S\ (tv_2, tw_2) \quad \text{and} \quad tw_2 + w \leq c$$

implies

$$tw_1 + w \leq c\ .$$

The obvious choice for S is therefore

$$(tv_1, tw_1)\ S\ (tv_2, tw_2) \ = \ (tw_1 \leq tw_2)\ .$$

It remains to choose T so that squeezing can eliminate dominated solutions. One such choice is

$$(tv_1, tw_1)\ T\ (tv_2, tw_2) \ = \ (tv_1 \geq tv_2)\ .$$

This guarantees that the list of intermediate solutions is strictly decreasing both in value and in weight. If the weights are integers, it follows that the time complexity of the resulting algorithm is $O(cn)$, where n is the number of items in the input sequence. If the weights are floating point numbers, the running time may be exponential. Admittedly it is possible to obtain algorithms that are much faster in practice, through the combined use of dynamic programming and branch-and-bound [22].

3.2 Example: bitonic tours

In the bitonic tours problem, the feasibility predicate is constant true, so the first condition is trivially satisfied.

To also satisfy the dominance criterion, we need a preorder S such that for any two tours t_1 and t_2: if

$$t_1\ S\ t_2 \text{ and } length(t_1) \leq length(t_2),$$

we have both

$$length(t_1 \oplus v) \leq length(t_2 \oplus v)$$

and

$$length(t_1 \otimes v) \leq length(t_2 \otimes v)\ .$$

That is, when we add a new vertex to either unidirectional part of t_1 and t_2, tour t_1 is still shorter than t_2. Clearly this condition can only be satisfied if the endpoints of the unidirectional parts coincide. We therefore define:

$$\begin{array}{l} ([a_1, a_2, ..., a_n], [b_1, b_2, ...,b_m]) \\ S \\ ([c_1, c_2, ..., c_k], [d_1, d_2, ..., d_l]) \\ = \\ a_n = c_k \quad \text{and } b_m = d_l\ . \end{array}$$

We define the preorder T to be trivially true, so that the merge step degenerates into a simple list concatenation. Because all tours obtained by application of ($\otimes$) have the same endpoints, squeezing results in only one solution being added to the list of candidates at each step. In this instance the time complexity of the algorithm is therefore quadratic.

3.3 Example: placing bus stops

In the bus stops example, the feasibility predicate is the requirement that we place no more than n stops. If we keep the number of allocated stops as part of the generated solution, this is an easy test, so we can take the predicate q in the generic algorithm to be the feasibility predicate p itself.

Recall that in this problem, the operator ($\oplus$) places a bus stop (by starting a new list partition component), and the operator ($\otimes$) skips a block (by gluing it to an existing list partition component). These two operators only preserve the order of walking cost if two partitions have the same last component. To wit, if

$$\begin{array}{ll} & walk([x_1, x_2, ..., x_k]) \leq walk([y_1, y_2, ..., y_i]) \\ \text{and} & k \leq i \\ \text{and} & x_k = y_i \\ \text{and} & i + 1 \leq n + 1 \end{array}$$

then

$$\begin{array}{ll} & walk([x_1, x_2, ..., x_k, [a]]) \\ & \quad \leq walk([y_1, y_2, ..., y_i, [a]]) \\ \text{and} & k+1 \leq i+1 \\ \text{and} & [a] = [a] \\ \text{and} & k+1 \leq n + 1\ . \end{array}$$

This is the relevant property for ($\oplus$); the same holds for ($\otimes$). It follows that we can instantiate the preorder S in the generic algorithm to

$$\begin{array}{l} [x_1, x_2, ..., x_k]\ S\ [y_1, y_2, ..., y_i] \\ = \\ k \leq i \quad \text{and} \quad x_k = y_i\ . \end{array}$$

It may appear that this is a computationally expensive comparison, as x_k and y_i are both sequences. However, recall that $[x_1, x_2, ..., x_k]$ and $[y_1, y_2, ..., y_i]$ partition the same list of blocks, so we can make the test $x_k = y_i$ by comparing the lengths of x_k and y_i.

It remains to define a total preorder T for ordering partial solutions. As remarked above, making an appropriate choice can greatly benefit the effectiveness of squeezing. A strategy that often succeeds is to take a relaxation of S. Instead of the conjunction (as in the definition of S), we take a lexicographical composition:

$$[x_1, x_2, ..., x_k]\ T\ [y_1, y_2, ..., y_i] = k < i \quad \text{or} \quad (k = i \text{ and } |x_k| \leq |y_i|)\ .$$

(Here $|z|$ denotes the number of elements in the sequence z.) Clearly this preorder is total, and furthermore the operators in our decision process are monotonic on this preorder.

The resulting program for the bus stops problem has time complexity $O(nm^2)$, where n is the number of bus stops, and m the number of blocks.

3 Programming the solution

The generic algorithm has been presented in some detail, but further work is required to actually program the algorithm, once and for all, as a software component. Such generic implementations are the topic of much current work in specialised branches of algorithm design, for instance in computational geometry [25]. There, standard algorithms are coded in C++, using features such as abstract classes and templates.

In this section, we sketch how our generic algorithm can be programmed as a C++ template, and we assess the suitability of C++ for exploring generic algorithms. It turns out that C++ forces the programmer to obfuscate the logical structure of the algorithm, but in return it offers a high degree of efficiency.

We contrast the lack of expressiveness in C++ with that of the functional programming language Haskell. Haskell allows us to concisely express the algorithm as presented above, keeping its logical structure, without severely compromising efficiency.

It is assumed that the reader has some familiarity with the basic concepts of C++, but not necessarily with those of Haskell.

4.1 A program in C++

We model our algorithm through a template that takes four parameters: a class of input items (called *Item*), a class of solutions (named *Sol*), an integer bound on the number of partial solutions, and an integer bound on the number of input values. The seven parameters to the algorithm are given through pure virtual functions, with prototypes:

seed e:	*seed*(*Sol*&)
operator ($\oplus$):	*void op1*(*Sol*,*Item*, &)
operator ($\otimes$):	*void op2*(*Sol*,*Item*,*Sol*&)
predicate q:	*int feasible*(*Sol*)
preorder R:	*int cheaper*(*Sol*,*Sol*)
preorder S:	*int lighter*(*Sol*,*Sol*)
preorder T:	*int precedes*(*Sol*,*Sol*)

To use the algorithm, one instantiates the template, and then derives a concrete class from that instance. For example, in a solution to the knapsack problem, we would define appropriate classes named *Item* and *Selection*, and then

class *KpSdp* : **public** *Sdp<Item,Selection*,50,100>
{ ... };

with methods to override the virtual functions that are parameters to the algorithm. To use the algorithm, one calls the constructor of *KpSdp*, which takes an array of input items and its size as parameters. Next, one calls the method *sdp* (defined in the *Sdp* template) to carry out the calculation, and a final call to *report*(&*ostream*) prints a solution. One can then re-initialise with a different input set, and repeat the processing.

Using the C++ template *Sdp* is straightforward, if somewhat cumbersome. The instantiation for the knapsack problem, for example, takes about 70 lines of trivial C++.

The main difficulty in the generic program itself lies in the updating of the list of intermediate solutions. In our description of the algorithm, that update consists of four separate phases. First two new lists of candidate solutions are constructed, using the methods *op1* and *op2*. Next, these two lists are merged. Third, the infeasible solutions are removed from the merged list. Finally, we squeeze the list, removing any element that is dominated by one of its neighbours.

The obvious way of implementing the table of intermediate solutions is through an array of fixed size. Admittedly this places a burden on the programmer to determine appropriate bounds, but it leads to very fast code, and it avoids the memory management problems that would arise from pointer-based structures. Such problems become especially nettlesome because partial solutions refer to each other, so the complete solution can be recovered from the table by *report*.

Unfortunately, however, the separation Sol into stages (of the list update) does not make sense for a table of fixed size. The intermediate lists obtained by merging, filtering and squeezing may easily grow too large. The C++ program therefore has to abandon this neat separation, and carry out all four phases in an interleaved fashion. The resulting loop has 4 levels of nested *if* statements. Some of this complexity can be hidden through appropriate procedural abstraction, but even then the code is hardly conducive to experimentation. Our C++ implementation of the list update is about 50 lines, so directly including it in this paper was quite out of the question.

4.2 A program in Haskell

Haskell is a functional programming language in the tradition of Lisp [5]. It differs from Lisp in a number of respects, however. For example, Haskell is a strongly typed language, and it does not allow any side-effects whatsoever. Haskell programs are evaluated lazily, so that an expression is only evaluated if it is needed to produce the final result.

Before we return to a discussion of the list update step in our generic algorithm, it may be helpful to first get an impression of what programs in Haskell look like. In Fig. 4, the full text of our generic algorithm is displayed; absolutely no detail has been left out. It may seem odd that there is no mention of types at all, and yet we claim that Haskell is a strongly typed language. This is because in Haskell, the compiler can automatically work out the types by itself. One thus gets the advantages of type safety without any of the burdens.

Fig. 4 Haskell program

```
listmin r = foldr1 choose
            where choose a b = if r a b then a else b

meet r s a b = r a b && s a b

squeeze r []      = []
squeeze r [a]     = [a]
squeeze r (a:b:x)  | r a b        = squeeze r (a:x)
                   | r b a        = squeeze r (b:x)
                   | otherwise  = a:squeeze r (b:x)

merge r [] y        = y
merge r x []        = x
merge r (a:x) (b:y) = if r a b
                        then a:merge r x (b:y)
                        else b:merge r (a:x) y

purge v q t x y = squeeze v (filter q (merge t x y))

sdp r s t q o1 o2 c   = listmin r .
                        foldr (step (meet s r) t q o1 o2) [c]

step v t q o1 o2 a xs = purge v q t
                         [o1 a x | x <- xs]
                         [o2 a x | x <- xs]
```

Instantiating the generic program in Haskell is an equally snappy affair. Fig. 5 shows the code for the knapsack problem. This instantiation does not only compute the optimal value, it also returns a selection that realises that value. The Haskell code is so much shorter than the C++ equivalent because we do not need to declare an *Item* class and a *Selection* class: these are just tuples of values. Furthermore, in lieu of the rather cumbersome use of an abstract base class, we can simply pass the parameters as function arguments. The brevity is thus partly due to the fact that tuples, lists and functions are all manipulated with the same ease as values of primitive type (such as integers).

Fig. 5 Instance for knapsack

```
kp c = the . sdp gv lw gv ok [cons,rhs] empty
       where   value (x,v,w) = v
               weight (x,v,w) =w
               the (x,v,w) = x
               gv s t = value s >= value t
               lw s t = weight s <= weight t
               ok s = weight s <= c
               cons (a,b) (x,v,w) = ((a,b):x, a+v, b+w)
               rhs (a,b) (x,v,w) = (x,v,w)
               empty = ([],0,0)
```

Let us now examine how the Haskell program implements the list update step. Recall that we described this step as having four phases:

1. Construction of two new candidate lists, using the binary operators (⊕) and (⊗).
2. Merging of these two lists with preorder *T*.
3. Removal of infeasible solutions with predicate *q*.
4. Squeezing with the dominance criterion (*R* and *S*).

All four phases are implemented in the function called *step*. It takes 7 arguments: *v* is the dominance criterion (*R* and *S*), *t* is the preorder *T* (capital names have a special purpose in Haskell), *q* is the feasibility test, *o1* and *o2* are the operators (⊕) and (⊗). The penultimate argument *a* is an input item, and *xs* is the list of intermediate solutions. The result of *step* is the new list of intermediate solutions.

The first phase of the computation is the construction of two new lists:

[*o1 a x* | *x* <- *xs*] and [*o2 a x* | *x* <- *xs*] .

This feature of Haskell, which allows us to use a set-like notation for lists, is a useful shorthand. These two lists are passed as arguments to the function *purge*.

The function *purge* performs the next three phases of the computation of the new list of intermediate solutions. Its definition shows these phases as successive function applications:

squeeze v (*filter q* (*merge t x y*)) .

The function (*merge t x y*) merges the two candidate lists, (*filter q*) removes all elements that do not satisfy *q*, and finally (*squeeze v*) removes dominated neighbours.

It now appears as if our Haskell program suffers from precisely the problem that we sought to avoid in C++, namely the creation of huge intermediate results. This is however not the case, because all expressions are evaluated lazily. This implies that the four phases of *step* execute in tandem: the elements of the two candidate lists from the first phase are generated on demand by *merge.* The result of merge is generated on demand by *filter*, and the computation of each element from *filter* is triggered by the computation of *squeeze*. The four phases thus interact in much the same way as pipes do in the Unix operating system. It follows that, despite the nice compositional description in the program, we produce the same entangled computation that had to be explicitly coded in C++.

Interested readers may wish to glance at [24], which describes the same generic algorithm as this paper, for an audience of programming language researchers. In particular, it contains the full Haskell code for the bitonic tours and bus stops problems. As in the case of knapsack, the instantiations are extremely compact.

5 An Algebra of Programs

In his Turing Award lecture, John Backus suggested that functional programs (written in languages such Haskell) form an algebra, and that programs can be algebraically manipulated to improve their time and space behaviour [2]. The example of the previous section gives some credence to the possibility. Haskell programs are so short that their algebraic manipulation is practically feasible.

In joint work with Richard Bird [4], I have been constructing just such an algebra, for the purpose of classifying common solution methods in combinatorial optimisation, such as dynamic programming, greedy algorithms and branch-and-bound. One starts with a generic problem, like that described in Section 2, expressed as a program itself. The aim is then to find conditions under which the inefficient but clear program can be rewritten (through a series of algebraic manipulations) to an efficient program. Such an approach to program derivation and classification was pioneered by Darlington [8,9,13], who illustrated its use in the area of sorting algorithms.

There is a problem, however, in applying this strategy to optimisation problems. More often than not, the specification is indeterminate, because there may be several solutions that realise the same optimal value. It follows that any functional program embodies a decision as to which of these equally acceptable solutions is returned. That very early design decision might then preclude the synthesis of an efficient algorithm. To keep our options open, the initial problem statement should specify a relation between input and output, say *P*. The programmer's task is then to find a total function *f* so that

$$f \subseteq P\ .$$

Because total functions are just a special kind of relation, it follows that in our algebraic manipulations we wish to manipulate relational expressions, as a generalisation of functional ones.

The idea of reasoning about programs in terms of relations is an old one ([4] describes its history, and cites references). However, we cannot immediately use those works to realise Backus' vision, because we also wish to profit by the benefits of functional programming, in particular the treatment of functions as first-class values, that can be passed as arguments to other programs. Fortunately, the view of relations offered by category theory allows one to generalise all the important operations from functional programming to relations in a canonical manner.

These are the twin ingredients of the algebra of programming that Richard Bird and myself have constructed: the calculus of relations, and the use of category theory to generalise important operations from functional programming to relations. We have used it to identify and formalise a number of algorithmic paradigms, in particular two views of dynamic programming. One of these was informally explained in this paper; an alternate view of dynamic programming is somewhat more general, but more difficult to implement directly as a software component.

It would go beyond the scope of this paper to give a full introduction to all the necessary terminology of this algebraic approach. Instead, we shall attempt to give the reader an impression of its nature by phrasing the generic problem that is the subject of this paper in terms of the calculus.

A relation is a set of pairs. General relations are written with capital letters, and functions have lower-case identifiers. We shall often read relations as non-deterministic mappings, which take an input on the right and produce an output on the left. For example,

$$a\ P\ b$$

says that a is one possible result when P is applied to b. Relational composition is written as an infix dot, and it is defined by

$$a\ (P \cdot Q)\ c \\ = \\ \exists\, b : a\, P\, b \wedge b\, Q\, c\ .$$

In the special case that P and Q are functions, this coincides with the usual definition of function composition.

We have chosen to view relations as sets of pairs, but one could also view them as set-valued functions. That is, given a relation R, the corresponding set-valued function is written ΛR:

$$\Lambda R\ (b) = \{\ a \mid a\, R\, b\ \}\ .$$

Using the notation introduced above, our generic problem can be formulated as follows:

$$Sdp = min\ R \cdot filter\ p \cdot gen\ op_1\ op_2\ .$$

Here R is a preorder, p is a function returning a boolean result, and op_1 and op_2 are binary functions. The relation *min R* relates a set to its minimum elements:

$$a\ (min\ R)\ x \\ = \\ a \in x\ \wedge (\forall\, b \in x : a\, R\, b\ )\ .$$

The function (*filter p*) removes all infeasible elements from a set:

$$filter\ p\ x = \{\ a \mid a \in x\ \wedge p\ a\ \}\ .$$

It remains to specify the decision process that takes a list of input items, and returns a set of candidate solutions. We shall do so through the function *foldl*, which is very commonly used in functional programming. It takes a function of two arguments, a seed value and a list, and it returns the left-to-right sum of the list, as shown in

$$foldl\ (\oplus)\ e\ [a_1, a_2, ..., a_n] \\ = \\ ((e \oplus a_1) \oplus a_2\ ) .... \oplus a_n\ .$$

This is where the category theory comes in: it tells us that there is a canonical generalisation of *foldl* where $(\oplus)$ is a general relation and not just a function. Furthermore, that canonical generalisation satisfies the same algebraic properties as its functional counterpart. In this particular example, it is easy to see what the canonical generalisation is. At each step, we apply the nondeterministic mapping $(\oplus)$, effectively making a choice at each step of the iteration. This suggests how we can define our decision process:

$$gen\ op_1\ op_2 \\ = \\ \Lambda\ (foldl\ (op_1 \cup op_2)\ e)\ .$$

The programmer's task is to turn this specification into a program, using the algebraic properties of the operators involved. The details of this calculation can be found in Chapter 8 of [4].

6 Related work

The generic algorithm presented in this paper owes much to the pioneering efforts of others. In particular, it was inspired by the seminal paper on sequential decision processes by Karp and Held [20]. Our view of dynamic programming is also heavily influenced by the work of Helman [18,19]. Helman actually states a program scheme that is even more general than that considered here; it is however not clear to us how that scheme can be efficiently implemented as a software component.

Many specific algorithms have elements in common with our generic program. We already noted that the idea of merging and squeezing was borrowed from the work of Ahrens and Finke [1]. The use of dominance criteria to speed up naive dynamic programming algorithms has been investigated in the algorithm design community [15,16,30]. Some of these efforts have focussed on a generic problem that is less general than the one in this paper, namely the *least weight subsequence* problem [19].

In programming methodology, our work is very much akin to that of Smith [27,28]. Smith's notion of problem reduction generators is quite similar to the generic algorithm presented here. However, Smith does not state a generic algorithm: instead, he seeks to express his result as a meta-program, that given a specification, will produce an efficient implementation of a dynamic programming algorithm.

There is a wealth of work in programming methods and language research on efficiently implementing recursion equations [10,26]. This requires the programmer to first express a solution as a recursive program (the 'functional equations' of dynamic programming), which gets subsequently optimised (through some form of tabulation) to efficient iterative code. The most recent example of such work is that of Liu [21], who has achieved a high degree of automation. As explained before, we wish to model the whole process of dynamic programming, and not merely the tabulation phase.

6 Acknowledgements

I would like to thank S.A. Belbas for the invitation to present this work at CSCC '99. Richard Bird suggested several improvements over an early draft.

References:

[1] J.H. Ahrens and G. Finke, Merging and sorting applied to the 0-1 knapsack problem, *Operations Research*, Vol. 23, No. 6, 1975, pp. 1099-1109.

[2] J. Backus, Can programming be liberated from the Von Neumann style? A functional style and its algebra of programs, *Communications of the ACM*, Vol. 21, 1981, pp. 613-641.

[3] R.S. Bird, Tabulation techniques for recursive programs, *Computing Surveys*, Vol. 12, No. 4, 1980, pp. 403-417.

[4] R.S. Bird and O. de Moor, *Algebra of programming*, Prentice Hall, 1997.

[5] R.S. Bird, *Introduction to Functional Programming in Haskell*, Prentice Hall, 1998.

[6] P. Bonzon, Necessary and sufficient conditions for dynamic programming of combinatorial type, *Journal of the ACM*, Vol. 17, No. 4, 1970, pp. 675-682.

[7] E.A. Boiten, Improving recursive functions by inverting the order of evaluation, *Science of Computer Programming*, Vol. 18, No. 2, 1992, pp. 675-682.

[8] R.M. Burstall and J. Darlington, A transformation system for developing recursive programs, *Journal of the ACM*, Vol. 24, No. 1, 1977, pp. 675-682.

[9] K.L. Clark and J. Darlington, Algorithm classification through synthesis, *Computer Journal*, Vol. 23, No. 1, 1980, pp. 61-65.

[10] N.H. Cohen, Characterization and elimination of redundancy in recursive programs, In: *Procs. Principles of Programming Languages*, 1979, pp. 143-157.

[11] T.H. Cormen, C.E. Leiserson and R.L. Rivest, *Introduction to Algorithms*, MIT Press, 1990.

[12] S. Curtis, *A relational approach to optimization problems*, D.Phil. thesis, Computing Laboratory, Oxford, UK, 1996.

[13] J. Darlington, A synthesis of several sorting algorithms. *Acta Informatica*, Vol. 11, No. 1, 1978, pp. 1-30.

[14] E.V. Denardo, *Dynamic Programming: Models and Applications*, Prentice-Hall, 1982.

[15] D. Eppstein, Z. Galil, R. Giancarlo, and G.F. Italiano, Sparse dynamic programming II: Convex and concave cost functions. *Journal of the ACM*, Vol. 39, No. 3, 1992, pp. 546-567.

[16] Z. Galil and R. Giancarlo, Speeding up dynamic programming with applications to molecular biology, *Theoretical Computer Science*, Vol. 64, 1989, pp. 107-118.

[17] P. Helman and A. Rosenthal, A comprehensive model of dynamic programming, *SIAM Journal on Algebraic and Discrete Methods*, Vol. 6, No. 2, 1985, pp. 319-334.

[18] P. Helman, A common schema for dynamic programming and branch-and-bound algorithms, *Journal of the ACM*, Vol. 36, No. 1, 1989, pp. 97-128.

[19] D.S. Hirschberg and L.L. Larmore, The least weight subsequence problem, *SIAM Journal of Computing*, Vol. 16, No. 4, 1987, pp. 628-638.

[20] R.M. Karp and M. Held, Finite-state processes and dynamic programming, *SIAM Journal on Applied Mathematics*, Vol. 15, No. 3, 1967, pp. 693-718.

[21] Y. A. Liu and S. D. Stoller. Dynamic programming via static incrementalization. In *Proceedings of the 8th European Symposium on Programming*, Amsterdam, The Netherlands, March 1999. Springer-Verlag.

[22] S. Martello and P. Toth, *Knapsack Problems: Algorithms and Computer Implementations*, Wiley 1990.

[23] L.G. Mitten, Composition principles for synthesis of optimal multistage processes, *Operations Research*, Vol. 12, 1964, pp. 610-619.

[24] O. de Moor. A generic program for sequential decision processes. In PLILP'95, LNCS 982, 1-23.

[25] J. Nievergelt. An introduction to geometric computing: from algorithms to software. Ch 1, in M. van Kreveld et al. (eds.), *Algorithmic Foundations of Geographic Information Systems*, LNCS 1340, Springer, 1997.

[26] A. Pettorossi, Methodologies for transformations and memoing in applicative languages, PhD. Thesis CST-29-84, University of Edinburgh, Scotland, 1984.

[27] D.R. Smith and M.R. Lowry, Algorithm theories and design tactics, *Science of Computer Programming*, Vol. 14, Nos. 2-3, 1990, pp. 305-321.

[28] D.R. Smith, Structure and design of problem reduction generators, in: B. Moeller (ed.), *Constructing Programs from Specifications*, North-Holland 1991, pp. 91-124.

[29] M. Sniedovich, A new look at Bellman's principle of optimality, *Journal of Optimization Theory and Applications*, Vol. 49, No. 1, 1986, pp. 161-176.

[30] F.F. Yao, Speed-up in dynamic programming, *SIAM Journal on Algebraic and Discrete Methods*, Vol. 3, No. 4, 1982, pp. 532-540.