Automating Branch-and-Bound for Dynamic Programs

Jakob Puchinger Peter J.Stuckey

NICTA Victoria Research Lab, Department of Comp. Sci. and Soft. Eng. University of Melbourne, Australia
{jakobp|pjs}@csse.unimelb.edu.au

Abstract

Dynamic programming is a powerful technique for solving optimization problems efficiently. We consider a dynamic program as simply a recursive program that is evaluated with memoization and lookup of answers. In this paper we examine how, given a function calculating a bound on the value of the dynamic program, we can optimize the compilation of the dynamic program function. We show how to automatically transform a dynamic program to a number of more efficient versions making use of the bounds function. We compare the different transformed versions on a number of example dynamic programs, and show the benefits in search space and time that can result.

Categories and Subject Descriptors I.2.2 [*Artificial Intelligence*]: Automatic Programming—Program transformation

General Terms Algorithms, Languages

Keywords Dynamic Programming, Branch and Bound, Automatic Transformation

1. Introduction

Dynamic programming (Bellman 1957) is a method of solving problems by decomposition into subproblems of the same form. Dynamic programming requires that the problem has the optimal substructure property, that is, one can create an optimal solution to a problem using only the optimal solutions of its subproblems. Dynamic programming is a highly successful approach to solving many optimization problems from bioinformatics to manufacturing, see e.g. (Hohwald et al. 2003; Yavuz and Tufekci 2006).

One reason dynamic programming is popular is dynamic programming solutions are typically easier to implement than other optimization approaches, since it simply requires functions and memoing. In this paper we therefore consider a dynamic program as simply being a recursive program that is evaluated with memoization and lookup of answers.

A classic example of a dynamic program is 0-1 knapsack.

EXAMPLE 1. *Consider the 0-1 knapsack problem. Given a set of items* $\{1,\ldots,n\}$ *of weight* $w_i, 1 \leqslant i \leqslant n$ *and profit* $p_i, 1 \leqslant i \leqslant n$*, and a constraint on total weight* W*. Choose the subset* $I \subseteq \{1,\ldots,n\}$ *such that* $\sum_{i\in I} w_i \leqslant W$ *and profit* $\sum_{i\in I} p_i$ *is maximized. The dynamic programming formulation solves knapsack problems* $\mathbf{k}(i,w)$ *returns the maximum profit for the knapsack problem which allows the use of items from* $\{1,\ldots,i\}$ *for a total weight of* w*. The relationship is represented by the pseudo-code*

$$\begin{aligned}\mathbf{k}(i,w) = {} & \mathsf{if}\ i = 0\ \mathsf{then}\ 0\\ & \mathsf{else\ if}\ w < w_i\ \mathsf{then}\ \mathbf{k}(i-1,w)\\ & \mathsf{else}\ \mathsf{max}(\mathbf{k}(i-1,w), \mathbf{k}(i-1,w-w_i)+p_i)\end{aligned}$$

Note that the function $\mathbf{k}$ above does not return the actual set of values I required to reach the optimal solution, but it can be determined by tracing back the calculations that lead to the optimal solution. This is standard for dynamic programming so we omit any further consideration of it in the paper, and concentrate solely on calculating the optimal solution.

In this paper we investigate how to optimize the compilation of this dynamic programming functions, given a function that determines an upper bound on the result.

EXAMPLE 2. . *An upper bound to the 0-1 knapsack problem (due to Dantzig (1957)) is the sum of* k *most profitable items where they fit in the knapsack, plus the proportion of the* $k+1th$ *most profitable item that fits in the remainder. If we assume that the items are ordered by profitability* p_i/w_i*, so* $p_i/w_i \geqslant p_j/w_j$ *for each* $1 \leqslant i < j \leqslant n$ *then it is easy to define an upper bound on each knapsack subproblem. Let* $P_i = \sum_{j=1}^{i} p_i$ *and* $W_i = \sum_{j=1}^{i} w_i$*.*

$$\mathbf{upper\_k}(i,w) = \begin{cases} P_i & \text{if } W_i \leqslant w \\ P_k + p_{k+1}(w - W_k)/w_{k+1} & \\ \qquad \text{if } \exists 0 \leqslant k < i . W_k \leqslant w \wedge W_{k+1} > w \end{cases}$$

We can use these bounds to improve the calculation of dynamic programs in a number of ways which we denote:

- *Local bounding* where we use already calculated alternatives to the dynamic program to give a local lower bound in calculating the remaining alternatives;
- *Ordering* where we use upper bounds to order the sub-problems examined in an order likely to find good solutions first; and
- *Argument bounding* where we calculate and pass lower bounds to each subproblem, and use upper bounds to prune the calculation of subproblems that are useless.

The use of bounds to improve dynamic programming dates at least back to the work of Morin and Martsen (1976). They consider a class of dynamic programs of the form maximization (or minimization) over a sum of incremental costs. Given a global lower bound L, they calculate the incremental costs required to reach a sub-problem. They check whether the sum c of the incremental costs to reach the sub-problem plus the upper bound on the sub-problem at least equals the global lower bound, before solving the sub-problem. This approach applied to 0-1 knapsack gives:

$$\begin{aligned}\mathbf{k}(i,w,c) = {} & \mathsf{if}\ c + \mathbf{upper\_k}(i,w) < L\ \mathsf{then}\ \mathbf{upper\_k}(i,w)\\ & \mathsf{else\ if}\ i = 0\ \mathsf{then}\ 0\\ & \mathsf{else\ if}\ w < w_i\ \mathsf{then}\ \mathbf{k}(i-1,w,c)\end{aligned}$$

PEPM'08, January 7–8, 2008, San Francisco, California, USA.

$$\mathsf{else}\ \mathsf{max}(\mathbf{k}(i-1,w,c),\ \mathbf{k}(i-1,w-w_i,c+p_i)+p_i)$$

The argument c records the profit made in the decisions taken to reach this sub-problem. It is increased, when we choose to add item i in the second argument of the max.

Later this approach was extended (Carraway and Schmidt 1991) to take advantage of lower bounds to improve the global lower bound. For example if we find that $c + \mathbf{lower\_k}(i, w) > L$, where $\mathbf{lower\_k}(i, w)$ is a lower bound on the knapsack subproblem then we can update L to be $c + \mathbf{lower\_k}(i, w)$. These approaches are only usable on dynamic programs defined as maximization (or minimization) over some associative commutative operator. They are quite related to the argument bounding approach. We note that the use of bounding for dynamic programming is a folklore technique, and there are many other kinds of bounding approaches that have been applied to specific problems (e.g. (Weingartner and Ness 1967; Spouge 1989; Hohwald et al. 2003; Yavuz and Tufekci 2006; Garcia de la Banda and Stuckey 2007)).

There are some works improving the performance of recursive programs by transforming them into dynamic programs via efficient incrementalization (Liu and Stoller 2003) or tabling (Somogyi and Sagonas 2006). In contrast to our approach those transformations do not use any kind of bounds information.

The contributions of this paper are:

- We show how one can automatically optimize dynamic program functions to make use of bounds, for a wide class of function.
- Our optimization creates much stronger bounded versions of dynamic programs than the only previous generic approaches we are aware of (Morin and Martsen 1976; Carraway and Schmidt 1991).
- We show that the automatically produced code competes with hand-specialized bounded dynamic programs.

In the remainder of the paper we first define the class of dynamic programs we will optimize, and discuss memoization. In Section 3 we show how to build an evaluation of the upper bound of an expression. In Section 4 we introduce the first two transformations: automatically adding local bounding information, and then extend this with ordering. In Section 5 we introduce the argument bounding approach which passes bounds information into the dynamic program. In Section 6 we discuss the extension of our approach to expressions involving loops. We give experimental results in Section 7 and then conclude.

2. Dynamic Programs as Functions

A dynamic program is most easily understood from a computer science perspective as simply a recursive function. We assume our dynamic programs use the following language of expressions. $\mathbf{dp}(\bar{o})$ is the dynamic programming function. We will only be interested in the part of the body of the function that appears above a call to **dp** since the transformations will not rely on the remaining parts. We use o to represent *other* parts of the expression, and $\bar{o}$ to represent a sequence of other parts.

$$\begin{array}{lrll} \text{Expressions} & e & ::= & o \mid x \mid \mathbf{dp}(\bar{o}) \mid e + e \mid e \times e \\ & & & \mathsf{if}\ o\ \mathsf{then}\ e\ \mathsf{else}\ e \mid \\ & & & \mathsf{min}(e,e) \mid \mathsf{max}(e,e) \mid \mathsf{let}\ x = e\ \mathsf{in}\ e \mid \end{array}$$

We assume x are variables, o are other expressions, which may not include **dp** expressions or make use of let-defined variables that (transitively) depend on **dp** expressions. We assume that $\times$ is restricted to expressions which are non-negative, this is not a strong restriction since dynamic programs that make use of multiplication (of **dp** expressions) typically satisfy it. We also assume for simplicity each let expression uses a different unique variable name.

The introduction gives the definition of the 0-1 knapsack dynamic program using this notation.

Call-based evaluation of a dynamic program, can be considered simply as evaluation of the recursive function, with the proviso that if a recursive call is made that has previously been made, rather than recomputing the result, the previously calculated answer is retrieved. Thus we are interested in *memoizing* answers (Michie 1968).

2.1 Memoization

In order to make a function into a dynamic program we must memoize the results of the function calls. We assume the following memoization functions: $getdp(\bar{o})$ returns a pair p of status of the memoization of $\mathbf{dp}(\bar{o})$ and a value. $optimal(p)$ is true if the optimal value is memoized, $value(p)$ returns the value memoized, $setdp(\bar{o}, x)$ memoizes the value x as the optimal value for $\mathbf{dp}(\bar{o})$ and returns x. The memoized version of a function simply wraps the expression e of the function body by $\mathsf{let}\ p = getdp(\bar{o})\ \mathsf{in}\ \mathsf{if}\ optimal(p)\ \mathsf{then}\ value(p)\ \mathsf{else}\ setdp(\bar{o}, e)$.

EXAMPLE 3. *The memoizing version of knapsack program from Example 1 is*

$$\begin{array}{l} \mathbf{k}(i,w) = \mathsf{let}\ p = getdp(\bar{o}) \\ \quad \mathsf{in}\ \mathsf{if}\ optimal(p)\ \mathsf{then}\ value(p) \\ \quad\quad \mathsf{else}\ setdp(i,w, \\ \quad\quad\quad \mathsf{if}\ i = 0\ \mathsf{then}\ 0 \\ \quad\quad\quad \mathsf{else\ if}\ w < w_i\ \mathsf{then}\ \mathbf{k}(i-1,w) \\ \quad\quad\quad \mathsf{else}\ \mathsf{max}(\ \mathbf{k}(i-1,w), \\ \quad\quad\quad\quad \mathbf{k}(i-1,w-w_i)+p_i)) \end{array}$$

3. Building Bounds Expressions for Dynamic Programs

In this paper we shall concentrate on dynamic programs for maximization problems. The optimization of minimization programs is analogous. The assumption of this paper is that the author of the dynamic program for $\mathbf{dp}(\bar{o})$ has also defined an upper bounding function $\mathbf{upper\_dp}(\bar{o})$ such that $\mathbf{dp}(\bar{o}) \leqslant \mathbf{upper\_dp}(\bar{o})$. We assume that after some amount of initialization work (that occurs before the dynamic program is invoked), the bounds function can be determined in (amortized) constant time.

In order to create bounded dynamic programs we will wish to create two types of expression from the definition of the dynamic program. The first expression $\text{UB}(e, Let)$ builds an expression to evaluate the upper bound of expression e under the assumption that equations $x = e$ in Let hold (the let definitions in scope). It is defined in Figure 1. The definition is straightforward, we simply replace calls to $\mathbf{dp}(\bar{o})$ by $\mathbf{u\_dp}(\bar{o})$. We define $\mathbf{u\_dp}(\bar{o}) = \mathbf{upper\_dp}(\bar{o})$. The reason for having two names for the upper bound, is that later we will replace this definition by a smarter version.

We now build expressions that evaluate a lower bounded version of a given expression e. The expression $\text{LBED}(e, l, Let)$ returns, under the assumption that the equations in Let hold, the value of e if it is greater than the given lower bound l, or if this is not the case, $\text{LBED}(e, l, Let)$ returns an upper bound on the value of e. This upper bound is guaranteed to be less than or equal to l. The expression is defined in Figure 2.

We assume that any newly introduced variables x are fresh. The interesting cases are for: $\mathbf{dp}(\bar{o})$ where we call a function $\mathbf{lbed\_dp}(\bar{o}, l)$ which will return the lower bounded version $\mathbf{dp}(\bar{o})$ (more on this later); for $+$ we use the upper bound of the second summand to give a correct lower bound for evaluating the first summand, while we use the actual value of the first summand to lower bound the second summand; similarly for $\times$; min where if

$$
\begin{array}{lcl}
\textsc{ub}(o, Let) & := & o \\
\textsc{ub}(x, \{x = e\} \cup Let) & := & \textsc{ub}(e, Let) \\
\textsc{ub}(\mathbf{dp}(\bar{o}), Let) & := & \mathbf{u\_dp}(\bar{o}) \\
\textsc{ub}(e_1 + e_2, Let) & := & \textsc{ub}(e_1, Let) + \textsc{ub}(e_2, Let) \\
\textsc{ub}(e_1 \times e_2, Let) & := & \textsc{ub}(e_1, Let) \times \textsc{ub}(e_2, Let) \\
\textsc{ub}(\mathsf{if}\ o\ \mathsf{then}\ e_1\ \mathsf{else}\ e_2, Let) & := & \mathsf{if}\ o\ \mathsf{then}\ \textsc{ub}(e_1, Let)\ \mathsf{else}\ \textsc{ub}(e_2, Let) \\
\textsc{ub}(\mathsf{min}(e_1, e_2), Let) & := & \mathsf{min}(\textsc{ub}(e_1, Let), \textsc{ub}(e_2, Let)) \\
\textsc{ub}(\mathsf{max}(e_1, e_2), Let) & := & \mathsf{max}(\textsc{ub}(e_1, Let), \textsc{ub}(e_2, Let)) \\
\textsc{ub}(\mathsf{let}\ x = e_1\ \mathsf{in}\ e_2, Let) & := & \textsc{ub}(e_2, \{x = e1\} \cup Let)
\end{array}
$$

Figure 1. UB expressions

$$
\begin{array}{lcl}
\textsc{lbed}(o, l, Let) & := & o \\
\textsc{lbed}(x, l, \{x = e\} \cup Let) & := & \textsc{lbed}(e, l, Let) \\
\textsc{lbed}(\mathbf{dp}(\bar{o}), l, Let) & := & \mathbf{lbed\_dp}(\bar{o}, l) \\
\textsc{lbed}(e_1 + e_2, l, Let) & := & \mathsf{let}\ x = \textsc{lbed}(e_1, l - \textsc{ub}(e_2, Let), Let)\ \mathsf{in} \\
 & & \quad x + \textsc{lbed}(e_2, l - x, Let) \\
\textsc{lbed}(e_1 \times e_2, l, Let) & := & \mathsf{let}\ x = \textsc{lbed}(e_1, l \div \textsc{ub}(e_2, Let), Let)\ \mathsf{in} \\
 & & \quad x \times \textsc{lbed}(e_2, l \div x, Let) \\
\textsc{lbed}(\mathsf{if}\ o\ \mathsf{then}\ e_1\ \mathsf{else}\ e_2, l, Let) & := & \mathsf{if}\ o\ \mathsf{then}\ \textsc{lbed}(e_1, l, Let) \\
 & & \mathsf{else}\ \textsc{lbed}(e_2, l, Let) \\
\textsc{lbed}(\mathsf{min}(e_1, e_2), l, Let) & := & \mathsf{let}\ x = \textsc{lbed}(e_1, l, Let)\ \mathsf{in} \\
 & & \quad \mathsf{if}\ x \leqslant l\ \mathsf{then}\ x \\
 & & \quad \mathsf{else}\ \mathsf{min}(x, \textsc{lbed}(e_2, l, Let)) \\
\textsc{lbed}(\mathsf{max}(e_1, e_2), l, Let) & := & \mathsf{let}\ x = \textsc{lbed}(e_1, l, Let)\ \mathsf{in} \\
 & & \quad \mathsf{max}(x, \textsc{lbed}(e_2, \mathsf{max}(l, x), Let)) \\
\textsc{lbed}(\mathsf{let}\ x = e_1\ \mathsf{in}\ e_2, l, Let) & := & \textsc{lbed}(e_2, l, \{x = e_1\} \cup Let)
\end{array}
$$

Figure 2. LBED expressions

the first expression fails to surpass the lower bound, we do not need to look at the second expression; and for max where the first expression can improve the lower bound for the second expression.

The definition of $\mathbf{lbed\_dp}(\bar{o}, l)$ is

$$\mathsf{let}\ x = \mathbf{u\_dp}(\bar{o})\ \mathsf{in}\ \mathsf{if}\ x \leqslant l\ \mathsf{then}\ x\ \mathsf{else}\ \mathbf{dp}(\bar{o})$$

which checks if the upper bound of the $\mathbf{dp}(\bar{o})$ cannot reach the required lower bound. If so we simply return the upper bound, otherwise we solve the subproblem.

Note that the definition of LBED above is careful to introduce let expressions for any result that is reused. This is important to avoid unnecessary repeated computation.

THEOREM 1. *Let C be a context assigning values to all (free) variables in e. Let $[\![e']\!]_C$ be the value of e' obtained by evaluating expression e' in context C. Suppose $[\![x]\!]_C = [\![e_x]\!]_C$ for each $x = e_x \in Let$. Then $u = [\![\textsc{lbed}(e, l, Let)]\!]_C$ is such that either $u > l$ and $u = [\![e]\!]_C$ or $u \leqslant l$ and $u \geqslant [\![e]\!]_C$.*

4. Local Bounding

With the transformations of the previous section we are ready to introduce local bounding transformations. For the local bounding transformation, once we have determined the first part of a max expression, we will use the bounds to determine if we should evaluate the second part. Figure 3 defines the local bounding transformation: we replace an expression $\mathbf{dp}(\bar{o}) = e$ by $\mathbf{dp}(\bar{o}) = \textsc{local}(e, \{\})$.

EXAMPLE 4. *Consider the knapsack program defined in Example 1 the result of the local bounding transformation (omitting the memoization wrapper and after inlining the definition of* $\mathbf{lbed\_k}(i-1, w - w_i, x_1 - p_i)$*) is*

$$
\begin{array}{l}
\mathbf{k}(i, w) = \mathsf{if}\ i = 0\ \mathsf{then}\ 0 \\
\qquad \mathsf{else}\ \ \mathsf{if}\ w < w_i\ \mathsf{then}\ \mathbf{k}(i-1, w) \\
\qquad \mathsf{else}\ \ \mathsf{let}\ x_1 = \mathbf{k}(i-1, w)\ \mathsf{in} \\
\qquad\quad \mathsf{max}(\ x_1, \\
\qquad\qquad \mathsf{let}\ x_2 = \mathsf{let}\ x_3 = \mathbf{u\_k}(i-1, w - w_i)\ \mathsf{in} \\
\qquad\qquad\qquad\quad \mathsf{if}\ x_3 \leqslant x_1 - p_i\ \mathsf{then}\ x_3 \\
\qquad\qquad\qquad\quad \mathsf{else}\ \mathbf{k}(i-1, w - w_i) \\
\qquad\qquad \mathsf{in}\ x_2 + p_i)
\end{array}
$$

One can see that if the upper bound on the second call $\mathbf{k}(i-1, w - w_i)$ is not good enough to increase the max then the recursive call will not be made. Clearly the resulting code could be improved further, for example, we can substitute for x_2 since it occurs only once. Also if x_2 takes the value $\mathbf{u\_k}(i-1, w - w_i)$ we are guaranteed that it will not create a maximum, but fixing this is more difficult, and the overhead is low.

The local bounding technique ensures that before any call to $\mathbf{dp}(\bar{o})$ if we have a known lower bound, then we check whether the upper bound can surpass the lower bound. The number of calls to **dp** to solve a problem using local bounds cannot increase.

THEOREM 2. *The set of calls C of the form $\mathbf{dp}(\bar{o}')$ made in order to solve a given problem $\mathbf{dp}(\bar{o})$ using the local bounding transformed version is a (possibly non-strict) subset of the set of calls C' used to solve $\mathbf{dp}(\bar{o})$ using the standard form.*

Hence the local bounding technique is likely to be beneficial as long as the overhead of computing $\mathbf{u\_dp}(\bar{o}')$ is small enough.

4.1 Implementing the bounds function

We are given an implementation of the upper bounds function $\mathbf{upper\_dp}(\bar{o})$ which can be used for $\mathbf{u\_dp}(\bar{o})$, but we should also

$$
\begin{array}{lcl}
\textsc{local}(o, Let) & := & o \\
\textsc{local}(x, Let) & := & x \\
\textsc{local}(\mathbf{dp}(\bar{o}), Let) & := & \mathbf{dp}(\bar{o}) \\
\textsc{local}(e_1 + e_2, Let) & := & \textsc{local}(e_1, Let) + \textsc{local}(e_2, Let) \\
\textsc{local}(e_1 \times e_2, Let) & := & \textsc{local}(e_1, Let) \times \textsc{local}(e_2, Let) \\
\textsc{local}(\mathsf{if}\ o\ \mathsf{then}\ e_1\ \mathsf{else}\ e_2, Let) & := & \mathsf{if}\ o\ \mathsf{then}\ \textsc{local}(e_1, Let) \\
 & & \mathsf{else}\ \textsc{local}(e_2, Let) \\
\textsc{local}(\mathsf{min}(e_1, e_2), Let) & := & \mathsf{min}(\textsc{local}(e_1, Let), \textsc{local}(e_2, Let)) \\
\textsc{local}(\mathsf{max}(e_1, e_2), Let) & := & \mathsf{let}\ x = \textsc{local}(e_1, Let)\mathsf{in} \\
 & & \quad \mathsf{max}(x, \textsc{lbed}(e_2, x, Let)) \\
\textsc{local}(\mathsf{let}\ x = e_1\ \mathsf{in}\ e_2, Let) & := & \textsc{local}(e_2, \{x = e_1\} \cup Let)
\end{array}
$$

Figure 3. LOCAL expressions

be aware of the possibility that when determining $\mathbf{u\_dp}(\bar{o})$ we may already have determined $\mathbf{dp}(\bar{o})$. In this case, there is no point in using the inaccurate upper bound function $\mathbf{upper\_dp}(\bar{o})$, when we have the accurate answer memorized. Hence we will extend the memoization of $\mathbf{dp}(\bar{o})$ so that bounds can be memorized as well, as follows:

$$
\begin{array}{l}
\mathbf{u\_dp}(\bar{o}) = \mathsf{let}\ p = getdp(\bar{o}) \\
\qquad \mathsf{in\ if}\ known(p)\ \mathsf{then}\ bound(p) \\
\qquad\quad \mathsf{else}\ bsetdp(\bar{o}, \mathbf{upper\_dp}(\bar{o}))
\end{array}
$$

where we extend the memoization functions with: $known(p)$ is true if either a bound or the optimal value is memoized, $bound(p)$ returns the bound or optimal value recorded in p, and $bsetdp(\bar{o}, x)$ memoizes the value x as a bound for $\mathbf{dp}(\bar{o})$ and returns x.

4.2 Ordering

Once we are not necessarily going to evaluate all of the subproblems that make up a dynamic program, then the order in which we try to evaluate them can make a difference. We can use the bounds as an estimate of where the better solution lies, and try those values first. The quicker a good solution is found, the more pruning the bounded dynamic program can achieve. This technique is known as best first search in branch and bound type algorithms.

The most obvious place to use ordering is in the max expressions that set up the bounds. Modifying the local bounding transformation to also order evaluations is straightforward, we simply evaluate the bounds of both expressions in a max before choosing which order to evaluate them. We change the following rules for the local transformation to implement ordering, see Figure 4.

EXAMPLE 5. *The local ordered version of knapsack (where again we have omitted the memoization wrapper and inlined calls to* **lbed_k***) is:*

$$
\begin{array}{l}
\mathbf{k}(i, w) = \mathsf{if}\ i = 0\ \mathsf{then}\ 0 \\
\qquad \mathsf{else\ if}\ w < w_i\ \mathsf{then}\ \mathbf{k}(i-1, w) \\
\qquad \mathsf{else\ let}\ x_1 = \mathbf{u\_k}(i-1, w)\ \mathsf{in} \\
\qquad\quad \mathsf{let}\ x_2 = \mathbf{u\_k}(i-1, w - w_i) + p_i\ \mathsf{in} \\
\qquad\quad \mathsf{if}\ x_1 \geqslant x_2 \\
\qquad\quad \mathsf{then\ let}\ x_3 = \mathbf{k}(i-1, w)\ \mathsf{in} \\
\qquad\qquad \mathsf{max}(\ x_3, \\
\qquad\qquad\quad \mathsf{let}\ x_4 = \\
\qquad\qquad\qquad \mathsf{let}\ x_5 = \mathbf{u\_k}(i-1, w - w_i)\ \mathsf{in} \\
\qquad\qquad\qquad \mathsf{if}\ x_5 \leqslant x_1 - p_i\ \mathsf{then}\ x_5 \\
\qquad\qquad\qquad \mathsf{else}\ \mathbf{k}(i-1, w - w_i) \\
\qquad\qquad\quad \mathsf{in}\ x_4 + p_i) \\
\qquad\quad \mathsf{else\ let}\ x_6 = \mathbf{k}(i-1, w - w_i) + p_i\ \mathsf{in} \\
\qquad\qquad \mathsf{max}(\ x_6, \\
\qquad\qquad\quad \mathsf{let}\ x_7 = \mathbf{u\_k}(i-1, w)\ \mathsf{in} \\
\qquad\qquad\quad \mathsf{if}\ x_7 \leqslant x_6\ \mathsf{then}\ x_7 \\
\qquad\qquad\quad \mathsf{else}\ \mathbf{k}(i-1, w))
\end{array}
$$

One can begin to see the reason why a programmer may want these transformations automated, given the relative size of this code, to the starting code of Example 1. Clearly we can improve this code by noticing that e.g. $x_7 = x_1$.

As defined the ordering is only applied to a top-most min or max expression. We could define an ordered version of LBED which also orders any subexpressions. Since the ordering code adds significant overhead, we are likely not to want to use it at every level of the expression. In order to experiment effectively with ordering we extend the expression language with ordered versions of the operations oplus (+), otimes ($\times$), omin (min) and omax (max). We can then extend the bounded expression evaluation to handled these new expressions using ordering. See Figure 5.

Theorem 2 extends also to the ordering version, but the overheads of the ordering version are larger compared to the simple local version.

5. Argument Bounding

The weakness of the local bounding approaches is that in each **dp** call we need to calculate a possible answer before we can make use of the bounding approach to prune. Given we have already calculated bounds in the function that calls $\mathbf{dp}(\bar{o})$ we should make use of this in the calculation of $\mathbf{dp}(\bar{o})$. This leads to the *argument bounding* approach where we add an extra argument to the **dp** function, to communicate the previously calculated lower bound. In effect we simply replace $\mathbf{dp}(\bar{o}) = e$ by $\mathbf{dp}(\bar{o}, l) = \textsc{lbed}(e, l, \{\})$.

On the face of it argument bounding could be disastrous. By adding a new argument to the **dp** function we extend the number of calls that need to be memoized. We will avoid this by carefully reusing the same memoization for different lower bounds l.

Argument bounding has other advantages. We now can move the handling of bounds calculations into the start of the **dp** function, instead of the call sites. This leads to cleaner and faster code.

The argument bounded function is created as shown below. The bounding calculations and memoization lookup and storage are folded into the expression.

$$
\begin{array}{l}
\mathbf{dp}(\bar{o}, l) = \mathsf{let}\ p = getdp(\bar{o}) \\
\qquad \mathsf{in\ if}\ optimal(p)\ \mathsf{then}\ value(p) \\
\qquad\quad \mathsf{else\ let}\ u = \mathsf{if}\ known(p)\ \mathsf{then}\ bound(p) \\
\qquad\qquad\qquad\qquad \mathsf{else}\ \mathbf{upper\_dp}(\bar{o}) \\
\qquad\qquad\quad \mathsf{in\ if}\ u \leqslant l\ \mathsf{then}\ u \\
\qquad\qquad\qquad \mathsf{else\ let}\ r = \textsc{lbed}(e, l, \{\})\ \mathsf{in} \\
\qquad\qquad\qquad\quad \mathsf{if}\ r > l\ \mathsf{then}\ setdp(\bar{o}, r) \\
\qquad\qquad\qquad\quad \mathsf{else}\ bsetdp(\bar{o}, r)
\end{array}
$$

The memoized answer is recovered, if it already records the optimal value this is returned. Otherwise if a previous upper bound has been recorded it puts it in u, otherwise the upper bound u is calculated using **upper_dp**. Now if the upper bound u is no greater

$$
\begin{array}{lcl}
\textsc{order}(\mathsf{max}(e_1,e_2),Let) & := & \mathsf{let}\ x_1 = \textsc{ub}(e_1,Let)\ \mathsf{in}\ \mathsf{let}\ x_2 = \textsc{ub}(e_2,Let)\ \mathsf{in} \\
& & \mathsf{if}\ x_1 \geqslant x_2 \\
& & \mathsf{then}\ \mathsf{let}\ x_3 = \textsc{order}(e_1,Let)\ \mathsf{in} \\
& & \quad \mathsf{max}(x_3, \textsc{lbed}(e_2,x_3,Let)) \\
& & \mathsf{else}\ \mathsf{let}\ x_4 = \textsc{order}(e_2,Let)\ \mathsf{in} \\
& & \quad \mathsf{max}(x_4, \textsc{lbed}(e_1,x_4,Let))
\end{array}
$$

Figure 4. ORDER expressions

$$
\begin{array}{lcl}
\textsc{lbed}(\mathsf{oplus}(e_1,e_2),l,Let) & := & \mathsf{let}\ x_1 = \textsc{ub}(e_1,Let)\ \mathsf{in}\ \mathsf{let}\ x_2 = \textsc{ub}(e_2,Let)\ \mathsf{in} \\
& & \mathsf{if}\ x_1 + x_2 \leqslant l\ \mathsf{then}\ x_1 + x_2 \\
& & \mathsf{else\ if}\ x_1 > x_2 \\
& & \quad \mathsf{then}\ \mathsf{let}\ x_3 = \textsc{lbed}(e_1, l - x_2, Let)\ \mathsf{in} \\
& & \quad\quad x_3 + \textsc{lbed}(e_2, l - x_3, Let) \\
& & \quad \mathsf{else}\ \mathsf{let}\ x_4 = \textsc{lbed}(e_2, l - x_1, Let)\ \mathsf{in} \\
& & \quad\quad x_4 + \textsc{lbed}(e_2, l - x_4, Let) \\
\textsc{lbed}(\mathsf{otimes}(e_1,e_2),l,Let) & := & \mathsf{let}\ x_1 = \textsc{ub}(e_1,Let)\ \mathsf{in}\ \mathsf{let}\ x_2 = \textsc{ub}(e_2,Let)\ \mathsf{in} \\
& & \mathsf{if}\ x_1 \times x_2 \leqslant l\ \mathsf{then}\ x_1 \times x_2 \\
& & \mathsf{else\ if}\ x_1 > x_2 \\
& & \quad \mathsf{then}\ \mathsf{let}\ x_3 = \textsc{lbed}(e_1, l \div x_2, Let)\ \mathsf{in} \\
& & \quad\quad x_3 \times \textsc{lbed}(e_2, l \div x_3, Let) \\
& & \quad \mathsf{else}\ \mathsf{let}\ x_4 = \textsc{lbed}(e_2, l \div x_1, Let)\ \mathsf{in} \\
& & \quad\quad x_4 \times \textsc{lbed}(e_2, l \div x_4, Let) \\
\textsc{lbed}(\mathsf{omin}(e_1,e_2),l,Let) & := & \mathsf{let}\ x_1 = \textsc{ub}(e_1,Let)\ \mathsf{in}\ \mathsf{let}\ x_2 = \textsc{ub}(e_2,Let)\ \mathsf{in} \\
& & \mathsf{if}\ \mathsf{min}(x_1,x_2) \leqslant l\ \mathsf{then}\ \mathsf{min}(x_1,x_2) \\
& & \mathsf{else\ if}\ x_1 \leqslant x_2 \\
& & \quad \mathsf{then}\ \mathsf{let}\ x_3 = \textsc{lbed}(e_1,l,Let)\ \mathsf{in} \\
& & \quad\quad \mathsf{if}\ x_3 \leqslant l\ \mathsf{then}\ x_3 \\
& & \quad\quad \mathsf{else}\ \mathsf{min}(x_3, \textsc{lbed}(e_2,l,Let)) \\
& & \quad \mathsf{else}\ \mathsf{let}\ x_4 = \textsc{lbed}(e_2,l,Let)\ \mathsf{in} \\
& & \quad\quad \mathsf{if}\ x_4 \leqslant l\ \mathsf{then}\ x_4 \\
& & \quad\quad \mathsf{else}\ \mathsf{min}(x_4, \textsc{lbed}(e_1,l,Let)) \\
\textsc{lbed}(\mathsf{omax}(e_1,e_2),Let) & := & \mathsf{let}\ x_1 = \textsc{ub}(e_1,Let)\ \mathsf{in}\ \mathsf{let}\ x_2 = \textsc{ub}(e_2,Let)\ \mathsf{in} \\
& & \mathsf{if}\ \mathsf{max}(x_1,x_2) \leqslant l\ \mathsf{then}\ \mathsf{max}(x_1,x_2) \\
& & \mathsf{else\ if}\ x_1 \geqslant x_2 \\
& & \quad \mathsf{then}\ \mathsf{let}\ x_3 = \textsc{lbed}(e_1,l,Let)\ \mathsf{in} \\
& & \quad\quad \mathsf{max}(x_3, \textsc{lbed}(e_2, \mathsf{max}(l,x_3), Let)) \\
& & \quad \mathsf{else}\ \mathsf{let}\ x_4 = \textsc{lbed}(e_2,l,Let)\ \mathsf{in} \\
& & \quad\quad \mathsf{max}(x_4, \textsc{lbed}(e_1, \mathsf{max}(l,x_4), Let))
\end{array}
$$

Figure 5. Operators with ordering

than the calling lower bound l we simply return the upper bound, otherwise we evaluate the body using the given bound l and store the result in r. If the result r surpasses the lower bound l we store it as optimal, otherwise we store it as a bound.

The only other change is required in expression $\textsc{lbed}(e,l,\{\})$, the definition of **lbed_dp**$(\bar{o},l')$ is changed to **dp**$(\bar{o},l')$.

EXAMPLE 6. *The argument bounded version of knapsack is:*

$$
\begin{array}{l}
\mathbf{k}(i,w,l) = \\
\quad \mathsf{let}\ p = getdp(\bar{o}) \\
\quad \mathsf{in\ if}\ optimal(p)\ \mathsf{then}\ value(p) \\
\quad\quad \mathsf{else\ let}\ u = \mathsf{if}\ known(p)\ \mathsf{then}\ bound(p) \\
\quad\quad\quad\quad\quad \mathsf{else}\ \mathbf{upper\_k}(i,w) \\
\quad\quad\quad \mathsf{in\ if}\ u \leqslant l\ \mathsf{then}\ u \\
\quad\quad\quad\quad \mathsf{else\ let}\ r = \mathsf{if}\ i = 0\ \mathsf{then}\ 0 \\
\quad\quad\quad\quad\quad\quad\quad \mathsf{else\ if}\ w < w_i\ \mathsf{then}\ \mathbf{k}(i-1,w,l) \\
\quad\quad\quad\quad\quad\quad\quad\quad \mathsf{else\ let}\ x_1 = \mathbf{k}(i-1,w,l)\ \mathsf{in} \\
\quad\quad\quad\quad\quad\quad\quad\quad\quad \mathsf{max}(\ x_1, \mathbf{k}(i-1, w-w_i, \\
\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad \mathsf{max}(l,x_1) - p_i) \\
\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad + p_i) \\
\quad\quad\quad\quad\quad\quad \mathsf{in\ if}\ r > l\ \mathsf{then}\ setdp(\bar{o},r)\ \mathsf{else}\ bsetdp(\bar{o},r)
\end{array}
$$

Note that as opposed to the local transformation, we only require one lookup of the memo table per function call.

One extra requirement of the argument bounded version is that the initial call must have a suitable initial bound. If we are maximizing we need a lower bound, note that this is the opposite bound to the function we require for the optimizing transformation. The usual approach is to use a heuristic solution to the problem to get a good bound.

We straightforwardly combine the argument bounding approach with ordering of subexpressions using the ordered rewriting for LBED of Section 4.2.

A result analogous to Theorem 2 does not hold for the argument bounding transformation. The body of **dp**$(\bar{o},l)$ can be executed multiple times for different values of l if they occur in a decreasing sequence, and are still greater than the actual optimal value. Our experiments will show that in fact any repeated computation for the same value $\bar{o}$ is very rare. Note that a result analogous to Theorem 2 does hold for the method of (Morin and Martsen 1976), but this also results in its inherent weakness. If the heuristic lower bound to the

problem is poor, there will not be as much pruning as in argument bounding, which updates this bound as computation proceeds.

6. Extending the Expression Language

Many dynamic programs use some form of set or list comprehension (loops) to define the recursive function. The transformations defined above straightforwardly extend to handle expressions of the form

$$\mathsf{min}\{e[x] \mid x \in o\} \mid \mathsf{max}\{e[x] \mid x \in o\} \mid \sum\{e[x] \mid x \in o\} \mid \Pi\{e[x] \mid x \in o\}$$

assuming the set o being looped over does not depend on the dynamic programming function. The only complexity is in describing the looping structure.

For example we can define the lower bounded evaluation of a max set expression as

$$\text{LBED}(\mathsf{max}\{e[x] \mid x \in o\}, l, Let) = \mathsf{foldl}(\lambda y.\lambda x.\mathsf{max}(y, \text{LBED}(e[x], y, Let)), l, o)$$

The function $\lambda y.\lambda x.\mathsf{max}(y, \text{LBED}(e[x], y, Let))$ takes the current lower bound as first argument y and the value for x and computes the maximum of the lower bounded evaluation of the expression with value x inserted, using y as the lower bound. This creates the new lower bound (and answer to the maximum). This function is folded over the set of values o for x, starting with initial lower bound l.

In order to create looping versions of $\sum$ (similarly for Π) we need to fold functions that pass a pair of (current sum, current lower bound).

Again we can extend the ordering approach to lower bounded evaluation of these loop expressions straightforwardly. Essentially the upper bounds of $e[x]$ are calculated for each $x \in o$ and then the values of o are sorted by this value to give list o'. We then fold over the list o'. For example

$$\begin{array}{l}\text{LBED}(\mathsf{omax}\{e[x] \mid x \in o\}, l, Let) = \\ \quad \mathsf{let}\ o' = [\mathsf{snd}(p) \mid p \in \mathsf{sort}([(-\text{UB}(e[x]), x) \mid x \in o])] \\ \quad \mathsf{in}\ \mathsf{foldl}(\lambda y.\lambda x.\mathsf{max}(y, \text{LBED}(e[x], y, Let)), l, o')\end{array}$$

The values x in o are paired with the negation of their upper bound $\text{UB}(e[x])$, and then sorted, and then the ordered x values are extracted into o'. This is then the order the loop is executed.

7. Experiments

We created a prototype optimizing compiler for dynamic program expressions in Prolog. It manipulates a term representing the dynamic program to create the term representing the bound optimized version. After transformation, some optimization of the term is performed, removing repeated let definitions, and substituting for let defined variables that occur at most once. Finally it outputs a C procedure for evaluating the dynamic program. The memoization functions and "other code" o used by the dynamic program are assumed to exist already in the C code.

The transformer does not fully support the looping expressions of Section 6, and in particular does not support the ordering on loop expressions.[1] For the one example (Open Stacks) where this is used, the code was added by hand.

We examine 3 problems: 0-1 knapsack, shortest paths and open stacks. The first two are well-studied and there are specific better algorithms than the dynamic programming approaches but they serve to illustrate the benefits of our approach. For the third problem, a dynamic programming solution is the state of the art.

[1] There is no intrinsic reason except the messiness of converting higher order terms into C code.

type	time	count	lookup	prune	resolve
Uncorrelated					
dp	154.60	2542848	2490438		
dpl	47.76	712591	690062		
dpo	66.40	712591	690062		
dpa	0.08	728	17	704	0
dpao	0.04	716	169	506	0
dpm	1.68	18568	3458	15050	
dpme	0.20	2663	363	2300	
Weakly correlated					
dp	142.84	2328037	2275126		
dpl	36.84	588732	565892		
dpo	51.92	588732	565892		
dpa	0.12	1000	27	962	0
dpao	0.20	861	129	666	0
dpm	5.44	64428	15622	48733	
dpme	1.56	14230	2292	11938	
Strongly correlated					
dp	309.00	7561692	2026466		
dpl	10.00	162775	88842		
dpo	13.96	162775	88842		
dpa	3.04	47149	0	30000	0
dpao	4.32	47597	3	26447	0
dpm	14.28	231792	121651	37439	
dpme	10.96	139123	76941	35712	
Inverse strongly correlated					
dp	169.60	2948450	2894723		
dpl	75.40	1106735	1075550		
dpo	105.48	1106735	1075550		
dpa	0.88	11392	763	10590	0
dpao	0.84	8347	81	6626	0
dpm	4.92	68184	46401	21728	
dpme	2.92	34501	20309	14192	

Table 1. 0-1 Knapsack on 4 classes: uncorrelated, weakly corr, strongly corr, inverse str. corr.

7.1 Knapsack

We have seen the knapsack example throughout the paper. The upper bounding function is discussed in the introduction. While there are many better algorithms for solving knapsack, see e.g. (Kellerer et al. 2004), the standard one above remains very good for medium sized problems.

We compare the knapsack code of: the original dynamic program (**dp**), locally bounded optimization (**dpl**), locally bounded ordering (ordering the max) (**dpo**), argument bounded optimization (**dpa**), and argument bounded ordering (**dpao**), as well as the approach of (Morin and Martsen 1976) (**dpm**) and its extension (Carraway and Schmidt 1991) (**dpme**).

We use the generator `gen2.c` (see (Martello et al. 1999)) available at `www.diku.dk/~pisinger` to create knapsack benchmarks with 500 items. We created 100 examples each of uncorrelated, weakly correlated, strongly correlated, inverse strongly correlated knapsack problems.

Table 1 shows the average results of each approach in terms of *time* (milliseconds), the *count* of the number of times the function body was executed (after lookup and pruning), the number of *lookups* of previous optimal solutions, the number of calls *pruned* by argument bounds, and the number of *resolves*, where a call $\mathbf{dp}(\bar{o})$ for which the function body has previously executed again executes the function body. We leave the column blank where the count is not applicable. Note that $count - resolves$ also gives the space required for memoization.

The results in Table 1 clearly show the benefits of bounded evaluation. The argument bounding approach is clearly superior to other approaches including **dpm** and **dpme**. Ordering is better for all cases except strongly correlated examples , and with poor initial lower bounds it can be massively better. Note **dpa** and **dpao** substantially improve on **dpm** and **dpme**.

type	time	count	lookup	prune	resolve
dp	6590.12	41726638	82703503		
dpl	205.77	978535	4584		
dpo	109.95	376304	4608		
dpa	101.58	395346	43899	735172	4938
dpao	59.16	198962	14268	374486	0
dp	6720.35	41748030	82746060		
dpl	364.30	1692531	9980		
dpo	173.81	589276	10029		
dpa	117.48	469791	93281	827654	15611
dpao	69.93	242022	33742	434612	0
dp	6603.70	41265211	81789087		
dpl	290.22	1383633	8445		
dpo	100.70	341407	8461		
dpa	127.85	505092	64475	930808	9404
dpao	74.98	255488	24127	475364	0

Table 2. USA-road-d.NY non-Euclidean bounds.

type	time	count	lookup	prune	resolve
dp	6963.84	41726638	82703503		
dpl	61.58	272404	1038		
dpo	38.42	119714	1052		
dpa	51.93	215811	29995	398375	191
dpao	7.26	19427	364	37690	0
dp	6755.91	41748030	82746060		
dpl	167.94	751044	3686		
dpo	99.19	313666	3751		
dpa	62.19	256666	60248	448779	473
dpao	11.04	28897	709	55737	0
dp	6856.39	41265211	81789087		
dpl	126.65	573112	2960		
dpo	55.78	169317	2973		
dpa	71.45	288697	41169	531596	747
dpao	14.68	39093	822	76152	0

Table 3. USA-road-d.NY Euclidean bounds.

7.2 Shortest Path

Finding shortest paths in a directed graph with nodes numbered 1 to n is another classic dynamic program: the Floyd-Warshall algorithm. The function $\mathbf{s}(i,j,k)$ returns the length of the shortest path from node i to node j using at most $1,\ldots,k$ as intermediate nodes. The recursive function is defined as follows.

$$\begin{aligned}\mathbf{s}(i,j,k) = {} & \mathsf{if}\ i=j\ \mathsf{then}\ 0\\ & \mathsf{else\ if}\ k=0\ \mathsf{then}\ d_{ij}\\ & \mathsf{else}\ \mathsf{min}(\mathbf{s}(i,j,k-1),\\ & \qquad \mathbf{s}(i,k,k-1)+\mathbf{s}(k,j,k-1))\end{aligned}$$

where d_{ij} is the (directed) edge length from i to j. Again while there are many algorithms for shortest path, this $O(n^3)$ algorithm is reasonable for medium sized graphs with negative edge lengths, particularly when multiple questions need to be answered, since it can reuse previously stored results.

We use two lower bounding functions in the experiments. The first simply makes use of connectivity. Let $p_{min} = \sum\{d'_i \mid 1 \leqslant i \leqslant n, d'_i = \min_j d_{ij}, d'_i < 0\}$ be the shortest possible node distinct path. Note if edge lengths are non-negative then $p_{min} = 0$.

We use a union-find algorithm applied to all edges incident to $1,\ldots,k$ (in other words all (i,j) where $\{i,j\} \cap \{1,\ldots,k\} \neq \varnothing$ and $d_{ij} < +\infty$) to define a representative $r(i,k)$ for each node i of the connected cluster it belongs to. If two nodes i and j have different representatives $r(i,k)$ and $r(j,k)$ then they cannot be connected through $1,\ldots,k$ and only the direct arc is possible. Otherwise we use p_{min} as the bound.

$$\mathbf{l\_s}(i,j,k) = \mathsf{if}\ r(i,k) \neq r(j,k)\ \mathsf{then}\ d_{ij}\ \mathsf{else}\ p_{min}$$

If each node i represents a position (x_i, y_i) in 2D space and the edge lengths d_{ij} are guaranteed to be greater than or equal to the Euclidean distance $\sqrt{(x_i-x_j)^2+(y_i-y_j)^2}$, then we can improve the above bound by replacing p_{min} by the Euclidean distance $\sqrt{(x_i-x_j)^2+(y_i-y_j)^2}$.

We compare the same set of procedures as for knapsack, except for **dpm** and **dpme** which are not applicable. The ordering is on the min. The instances we used for evaluating our approach were derived from instances of the 9th DIMACS Implementation Challenge - Shortest Paths (`www.dis.uniroma1.it/~challenge9/`). We used three 500 node slices of the New York City dataset, with up to 100 feasible shortest path queries per instance. We solve them without (Table 2), or with the Euclidean distance bounds (Table 3).

The results are shown in Tables 2 and 3. For **dpa** and **dpao** we use a heuristic upper bound greater than the sum of maximum arc lengths. For these examples the difference between local and argument bounding is less than previously, and the advantage of ordering is clear. This example also illustrates how a more accurate bounding function (Euclidean) can substantially improve the results.

7.3 Open Stacks

The minimization of maximum number of open stacks problem is defined as follows: Given a set of products P for each product $p \in P$ we have a set of customers wanting the product $cust(p)$. A stack is open for a customer c from the time the first product p is created where $c \in cust(p)$, to the time the last product p is created with $c \in cust(p)$. The aim is to order the set of products to minimize the maximum number of open stacks. Then $a(p', S) = |(\cup_{p \in S \cup \{p'\}} cust(p)) \cap (\cup_{p \in P-S} cust(p))|$ is the number of stacks open when scheduling p' after $P-S-\{p'\}$ and before S. The recursive function definition $\mathbf{o}(S)$ which give the number of open stacks for scheduling S assuming $P-S$ where scheduled previously is:

$$\begin{aligned}\mathbf{o}(S) = {} & \mathsf{if}\ S=\{\}\ \mathsf{then}\ 0\\ & \mathsf{else}\ \mathsf{min}\{\mathsf{max}(a(p, S-\{p\}), \mathbf{o}(S-\{p\})) \mid p \in S\}\end{aligned}$$

We use the lower bounding function discussed in (Garcia de la Banda and Stuckey 2007).

We compare on a few of the more difficult small benchmarks (20 products, 20 customers, each a suite of 100 instances) from the Constraint Modelling Challenge 2005 (Constraint Modelling Challenge). We compare the same set of codes as in knapsack, as well as the dynamic programming solution (**dpc**) to this problem that won the Constraint Modelling Challenge 2005, beating other solutions by at least two orders of magnitude.

The results are shown in Table 4. For **dpa**, **dpao**, **dpm**, **dpme** and **dpc** we use the best heuristic upper bound of 6 heuristics (for details see (Garcia de la Banda and Stuckey 2007)). For these examples this is almost always the optimal answer. Hence **dpm** is very similar to **dpa** in performance. **dpme** is slower because it needs to find many heuristic solutions for the sub-problems.

To show the sensitivity to the upper bound, we also use simply the number of customers. The poor upper bound illustrates the weakness of the approach of (Morin and Martsen 1976). Since it cannot use solutions it has already generated to improve the upper bounding. Table 5 shows the results using this upper bound on the methods that make use of it. We can see that **dpa**, **dpao**, and **dpme** are almost unchanged, but **dpm** performs uniformly worse than **dpl**.

Ordering is distinctly worse for this example, because so many solutions are equivalent in terms of objective, and it forces more to be explored. Note that **dpa** is only around 4 times slower and never requires more than 25% more space than the best known solution **dpc** which incorporates many other kinds of optimizations, as well

type	time	call	look	prune	resolve
problem_20_20					
dp	1359.49	483773	4186989		
dpl	59.18	21920	18167		
dpo	101.16	14179	5697		
dpa	10.82	3145	5	12659	0
dpao	109.00	15241	26636	16916	972
dpm	11.02	2869	11881	23948	
dpme	148.04	2781	11563	23354	
dpc	2.87	1970	2816		
wbo_20_20					
dp	3074.13	1003447	9014094		
dpl	195.87	66439	58442		
dpo	293.29	36113	14139		
dpa	18.27	4735	9	16987	0
dpao	313.56	38642	57453	33913	2237
dpm	20.53	4973	18526	49729	
dpme	241.87	4353	16190	44697	
dpc	4.58	3284	4672		
wbop_20_20					
dp	3237.96	1048595	9437166		
dpl	184.36	61233	54879		
dpo	269.02	32260	13238		
dpa	16.40	4156	8	14516	0
dpao	285.82	34386	49194	31554	1948
dpm	20.84	4698	17642	47747	
dpme	204.53	3760	13501	40049	
dpc	5.02	3465	6464		
wbp_20_20					
dp	1342.36	471523	4097099		
dpl	91.29	33323	27930		
dpo	157.60	21489	8938		
dpa	13.64	4012	8	15566	0
dpao	169.20	23059	40236	26884	1462
dpm	16.62	4584	18745	38453	
dpme	183.60	3451	13647	31248	
dpc	4.36	3009	5147		

Table 4. Openstacks: problem_20_20, wbo_20_20, wbop_20_20, wbp_20_20

type	time	call	look	prune	resolve
problem_20_20					
dpa	10.24	3214	13	12810	0
dpao	108.69	15241	26636	16916	972
dpm	339.45	122301	917271	136591	0
dpme	147.98	2788	11580	23407	0
wbo_20_20					
dpa	18.04	4878	33	17247	0
dpao	312.09	38642	57453	33913	2237
dpm	2028.98	653654	5582559	301957	0
dpme	241.78	4382	16254	45012	0
wbop_20_20					
dpa	15.38	4255	36	14599	0
dpao	285.91	34386	49194	31554	1948
dpm	1552.44	494477	4086391	396711	0
dpme	205.33	3777	13550	40215	0
wbp_20_20					
dpa	13.56	4064	21	15614	0
dpao	168.09	23059	40236	26884	1462
dpm	307.69	110895	785350	170869	0
dpme	184.40	3481	13760	31452	0

Table 5. Openstacks: problem_20_20, wbo_20_20, wbop_20_20, wbp_20_20 with weak upper bounds

its own special bounding approach. We think this is impressive for an approach derived directly from the naive recursive equation.

8. Conclusion

Branch and bound for dynamic programs is folklore, with little formally published description, hence it is less frequently used than it could be. Adding branch and bound by hand also compromises the simplicity of dynamic programming, and hence is perhaps seen as unattractive, but it can massively improve performance. By allowing automatic bounding of dynamic programs programmers gain the advantages of bounding without the complexity. Of the approaches we investigate argument bounding, with or without ordering is best, and even though it is not guaranteed to resolve the same problem, clearly this happens only rarely in our examples. Clearly the effectiveness of the approach depends upon the tightness of the bounding function supplied, but the simple bounds example for shortest paths illustrates that even with a relatively crude bounding function we can still get significant improvements.

An interesting direction for future work would be to integrate our ideas in one of the available dynamic programing frameworks (Giegerich and Meyer 2002; Eisner et al. 2005), mainly used in bioinformatics applications. This may lead to performance improvements by orders of magnitude for existing and future applications, as it did for our examples.

We can extend the approach to a greater class of expressions (negation, subtraction, multiplication by non-positive numbers) if we also have a a separate lower bounding function. There are other optimizations/transformations that we could imagine including: checking whether we have already computed the optimal answer $\mathbf{dp}(\bar{o})$ before worrying about bounding. and optimistically improving bounds in the hope of finding good solutions faster.

Acknowledgments

We would like to thank Moshe Sniedovich for helpful discussions on this subject, and Ting Chen for his work on an earlier prototype transformer, as well as the referees whose comments helped improve the presentation.

References

R. Bellman. *Dynamic Programming*. Princeton University Press, 1957.

R. Carraway and R. Schmidt. An improved discrete dynamic programming algorithm for allocating resources among interdependent projects. *Management Science*, 37(9):1195–1200, 1991.

Constraint Modelling Challenge. Constraint Modelling Challenge 2005. `http://www.dcs.st-and.ac.uk/~ipg/challenge/`, 2005.

G. Dantzig. Discrete variable extremum problems. *Operations Research*, 5:266–277, 1957.

J. Eisner, E. Goldlust, and N.A. Smith. Compiling Comp Ling: Weighted dynamic programming and the Dyna language. In *Proceedings of Human Language Technology Conference and Conference on Empirical Methods in Natural Language Processing (HLT-EMNLP)*, pages 281–290, Vancouver, 2005. Association for Computational Linguistics.

M. Garcia de la Banda and P.J. Stuckey. Dynamic programming to minimize the maximum number of open stacks. *INFORMS Journal of Computing*, 19(4):607–617, 2007. See (Constraint Modelling Challenge) for a shorter version.

R. Giegerich and C. Meyer. Algebraic dynamic programming. In *AMAST '02: Proceedings of the 9th International Conference on Algebraic Methodology and Software Technology*, volume 2422 of *LNCS*, pages 349–364. Springer, 2002.

H. Hohwald, I. Thayer, and R.E. Korf. Comparing best-first search and dynamic programming for optimal multiple sequence alignment. In Georg Gottlob and Toby Walsh, editors, *IJCAI-03, Proceedings of the Eighteenth International Joint Conference on Artificial Intelligence*, pages 1239–1245. Morgan Kaufmann, 2003.

H. Kellerer, U. Pferschy, and D. Pisinger. *Knapsack Problems*. Springer, 2004.

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

S. Martello, D. Pisinger, and P. Toth. Dynamic programming and strong bounds for the 0-1 knapsack problem. *Management Science*, 45(3):414–424, 1999.

D. Michie. “memo” functions and machine learning. *Nature*, 218:19–22, 1968.

T.L. Morin and R.E. Martsen. Branch-and-bound strategies for dynamic programming. *Operations Research*, 24(4):611–627, 1976.

Z. Somogyi and K. F. Sagonas. Tabling in Mercury: Design and implementation. In Pascal Van Hentenryck, editor, *Practical Aspects of Declarative Languages, 8th International Symposium, PADL 2006*, volume 3819 of *LNCS*, pages 150–167. Springer, 2006.

J.L. Spouge. Speeding up dynamic programming algorithms for finding optimal lattice paths. *SIAM Journal on Applied Mathematics*, 49(5): 1552–1566, 1989.

H.M. Weingartner and D.N. Ness. Methods for the solution of the multi-dimensional 0/1 knapsack problems. *Operations Research*, 15:83–103, 1967.

M. Yavuz and S. Tufekci. A bounded dynamic programming solution to the batching problem in mixed-model just-in-time manufacturing systems. *International Journal of Production Economics*, 103(2), 2006.