

# Experiments with the "Oregon Trail Knapsack Problem"

**Jennifer J. Burg\***

**John Ainsworth<sup>1</sup>**

**Brian Casto**

Wake Forest University  
Winston-Salem, NC 27109  
burg@mthcsc.wfu.edu

**Sheau-Dong Lang**

University of Central Florida  
Orlando, Florida  
32816  
lang@cs.ucf.edu

## Abstract

This paper presents hybrid algorithms for a variation of the Bounded Knapsack Problem which we call the Oregon Trail Knapsack. Our problem entails imposing a cost as well as a weight limit, constraining the values of types of items by means of a variety of value functions, and allowing the value of one item type to be dependent on the presence or absence of another type in the knapsack. These modifications to the original problem make it more complex and require adaptations of known knapsack algorithms. To solve this problem, we combine constraint propagation techniques and domain pruning with classic branch and bound approaches that require a sorting of the items. Our experiments compare a constraint-language implementation with a simulation of the constraint-based system in a procedural language. Results indicate that the constraint-based solution is natural to the problem and efficient enough to solve large problem instances typical of the application.

## 1. Introduction

The knapsack problem has seen a number of variations over the years. In its basic form, the problem is to maximize the value of items placed in a knapsack without going over a weight limit. This is the *0/1 Knapsack Problem*, defined more formally as

$$\text{maximize } \sum_{j=1}^n v_j x_j$$

$$\text{subject to } \sum_{j=1}^n w_j x_j \leq W,$$

$$x_j = 0 \text{ or } 1, \quad 1 \leq j \leq n$$

where  $v_j$  is the value of item  $j$ ,  $w_j$  is item  $j$ 's weight,  $x_j$  is an integer, and  $W$  is the maximum weight allowed in the knapsack. If the item set is partitioned into subsets and only one item per subset can be placed in the knapsack, the problem is known as the *Multiple Knapsack Problem*. When more than one item of each type  $j$  is available, then each  $x_j$  is bounded by  $b_j$  (i.e.,  $0 \leq x_j \leq b_j$ ) and the problem becomes the *Bounded Knapsack Problem*. If  $b_j = +\infty$ , we have the *Unbounded Knapsack Problem*. When  $v_j = w_j$  for  $1 \leq j \leq n$ , the problem is to find a subset of weights whose sum is closest to but does not exceed the limit; this is the *Subset-Sum Problem*. When the number of items of each type is bounded, and the constraints are equality constraints, as in

$$\sum_{j=1}^n w_j x_j = W$$

the problem is equivalent to a cashier's making change and is called the *Change-Making Problem*. In the *0/1 Multiple Knapsack Problem*,  $m$  knapsacks, each with its own weight limit, are available. A variation of this, the *Generalized Assignment Problem*, gives a different value

---

\* This work was supported by National Science Foundation Grant CCR-9619523.

<sup>1</sup> John Ainsworth currently is employed by Elder Research Inc.

and weight to an item depending on the knapsack it is assigned to. All of the problems are NP-Hard [Martello and Toth].

We offer a variation of the *Bounded Knapsack Problem* which involves imposing a cost as well as a weight limit, defining the value of each item by a function that is not necessarily a constant, and allowing a value for an item type  $i$  to depend on the presence or absence of another item type  $j$  in the knapsack. The idea is taken from the Oregon Trail<sup>2</sup> computer game, where players are asked to imagine preparing for a trek across the Oregon Trail. In order to make it across country, the travelers need to get good value for the supplies they purchase. They have a given amount of money to spend, and the weight of their supplies cannot exceed the capacity of their wagon.

We model this game by imposing a cost limit as well as a weight limit and defining values by functions that are not necessarily constant. The value of an item type may be constant, or it may decrease as more than one item of that type is taken. Further, an item of one type may have no value at all if it is not accompanied by one of a certain other type, or an item of one type may be less valuable in the presence of another type. This provides a realistic problem: One does not need an infinite number of ropes, and thus they diminish in value as you take more of them; guns, on the other hand, are worthless without ammunition; and conversely, a bow and arrow is less valuable in the presence of a gun.

More formally, our problem is to

$$\begin{aligned} & \text{maximize } \sum_{j=1}^n f_j(x_j, x_{d_j}) \\ & \text{subject to } \sum_{j=1}^n w_j x_j \leq W \quad \text{and} \\ & \quad \quad \quad \sum_{j=1}^n c_j x_j \leq C \end{aligned}$$

where  $0 \leq x_j \leq b_j$ ,  $1 \leq j \leq n$ ,  $x_j$  is an integer,  $W$  is the weight limit, and  $C$  is the cost limit.

$f_j(x_j, x_{d_j})$  is the value function for type  $j$ , in which  $d_j$  gives the index of the type upon which the value of  $x_j$  depends. If  $d_j = j$ , the value of the function  $f_j()$  depends only on  $x_j$ . Otherwise, if  $d_j \neq j$ , the value of function  $f_j()$  depends on  $x_j$  and  $x_{d_j}$ . An array of constant indices  $d_1, d_2, \dots, d_n$  provide the dependency information among the item types. (Each function  $f_j$  also has a value constant  $v_j$ , but for simplicity we do not include this in our notation.)

Specifically, there are three kinds of value functions, described as follows:

type 0:  $f_j(x_j, x_{d_j}) = x_j * v_j * [x_{d_j} > 0]$

where  $v_j$  is the constant value associated with each type  $j$  item. We use Iverson's notation, where  $[x_{d_j} > 0]$  gives a boolean value 1 or 0 depending on whether  $x_{d_j} > 0$  is true or false, respectively [Graham]. That is, items of type  $j$  have value only if at least one item of type  $d_j$  is present in the knapsack.

type 1:  $f_j(x_j, x_{d_j}) = [x_{d_j} > 0] * \sum_{i=0}^{x_j-1} r^i * v_j$

---

<sup>2</sup> The Oregon Trail computer game is a production of MECC. We might also have used the game Outpost®, by SierraOriginals, to model our problem. In this game, players must pack their space vehicle for space travel.

$$= [x_{d_j} > 0] * v_j * \frac{1 - r^{x_j}}{1 - r}$$

for  $0 < r < 1$ . This is the case where adding multiple items of type  $j$  results in diminishing values at a rate of  $r$ , i.e., the first item has a value of  $v_j$ , the second  $r * v_j$ , ..., the  $x_j^{th}$  has a value of  $r^{x_j-1} * v_j$ . We use  $r = \frac{1}{2}$  in our experiments. Again, this function can be multiplied by

$[x_{d_j} > 0]$ , giving type  $j$  no value when type  $d_j$  is not present.

type 2:  $f_j(x_j, x_{d_j}) = x_j * v_j - [x_{d_j} > 0] * t * x_j * v_j$

where  $0 < t \leq 1$  and  $d_j \neq j$ . That is, the value associated with  $x_j$  type  $j$  items is reduced by a factor of  $t$  when type  $x_{d_j}$  items are present in the solution. In particular, the value of type  $j$  items can be reduced to zero when  $t = 1$ . We will use  $t = \frac{1}{2}$  in our experiments.

These functions were chosen as reasonable ones to model the Oregon Trail problem. Others could easily be devised and incorporated into our constraint-based scheme, which is one of the advantages of the constraint-based approach.

Constraint-based solutions to many discrete optimization and operations research problems have proven to be natural, expressive, and efficient enough to be of practical value. (See [Simonis], [Wallace] and [Van Hentenryck] for surveys and examples.) We adopt a hybrid approach which begins in a manner similar to that taken for problems such as the Traveling Salesperson [Caseau and Laburthe][Pesant and Gendreau] but varies the method of computing the upper bound. Our experiments are based on two implementations of the *Oregon Trail Knapsack* problem, one in C and one in CLP(R) [Jaffar et al.], and variations of these. These experiments show the benefits of domain pruning, and also demonstrate the elegance of the constraint-based approach to the problem, which allows the value functions to be naturally expressed and absorbed into the constraint satisfaction system.

## 2. Approaches to the 0/1 Knapsack and the Bounded Knapsack Problems

Approaches to the *0/1 Knapsack Problem* have evolved from the first dynamic programming algorithms, to algorithms based on computing an upper bound through either a linear programming relaxation or a Lagrangian relaxation of the problem [Dantzig] [Maculén][Fisher], to branch and bound methods based upon a sorting of the variables [Horowitz and Sahni][Martello and Toth], and on to algorithms which avoid sorting by identifying a "core problem" [Balas and Zemel][Martello and Toth]. These methods have successfully solved problems with  $n > 200,000$  within seconds.

A *Bounded Knapsack Problem* can be transformed into a *0/1 Knapsack Problem* in polynomial time, and generally the best way to solve the *Bounded Knapsack Problem* is to transform it and use the efficient *0/1 Knapsack* algorithms.

Both *0/1 Knapsack* and *Bounded Knapsack* are NP-Hard problems for which pseudo-polynomial dynamic programming algorithms exist. It would be possible to adapt the dynamic programming method to our variant of the knapsack problem by first doing a topological sort of the item types based on the dependencies in the value functions. If we order the item types such that type  $i$  always precedes type  $j$  if its value depends on  $j$ , then, assuming there are no cycles in the dependencies, we could compute maximum values in the bottom-up fashion required for

this approach. However, we don't wish to discard the possibility of cyclical dependencies, and furthermore dynamic programming has the disadvantage of not allowing any pruning of the search space. In fact, while the dynamic programming solution to the *Bounded Knapsack*

*Problem* has a complexity of  $O(W \sum_{j=1}^n b_j)$  (assuming all the weight parameters are integers), it is

in practice generally less efficient than branch and bound.

Our method for solving the *Oregon Trail Knapsack Problem* combines domain pruning -- an essential constraint-based approach -- with a classic branch and bound method that compares known upper bounds to lower bounds. These combined methods allow for a natural problem representation, easy addition of constraints, and enough efficiency to solve reasonably large problem instances.

### 3. Domain Pruning and Branch and Bound in the Oregon Trail Knapsack

#### 3.1. Domain Pruning

From a constraint-based perspective, the problem is to give values to the variables  $x_1$  through  $x_n$ , where the domain of  $x_j$ ,  $1 \leq j \leq n$ , is  $\{0, \dots, b_j\}$ . For our application, a vector of variables,  $y_1$  through  $y_n$ , is maintained to represent Iverson's notation  $[x_i > 0]$  for  $1 \leq i \leq n$ . That is, whenever  $x_i > 0$ ,  $y_i = 1$ , and otherwise  $y_i = 0$ . Given in the problem are vector  $\mathbf{d}$ , the value functions for each item type, the bounds (in  $\mathbf{b}$ ), the weights (in  $\mathbf{w}$ ), the costs (in  $\mathbf{c}$ ), the weight limit ( $W$ ), and the cost limit ( $C$ ). The search proceeds in a depth-first manner. At each level  $j$  in the search tree, the children of  $j$  represent the different values possible for variable  $x_j$ .

At level  $j$  of the search tree, domain-pruning is based upon the redundant constraints

$$x_k w_k + \sum_{i=1}^j w_i x_i \leq W, \quad \text{for all } k, j+1 \leq k \leq n \text{ and}$$

$$x_k c_k + \sum_{i=1}^j c_i x_i \leq C, \quad \text{for all } k, j+1 \leq k \leq n$$

That is, after  $j$  items have been placed in the knapsack, for every item type  $k$  not in the knapsack, if  $m$  items of type  $k$  would put us over the weight or cost limit, then  $\{m, \dots, b_k\}$  are pruned from  $x_k$ 's domain.

#### 3.2. Bounding

Domain pruning effectively gives us a new bound  $b_k'$  on the maximum number of type  $k$  items that can be taken. The upper bound of the solution at level  $j$  is given by

$$V = \sum_{i=1}^j f_i(x_i, x_{d_i}) + \sum_{i=j+1}^n f_i(x_i = b_i', x_{d_i})$$

Note that where  $d_i > j$ ,  $x_{d_i}$  is not yet bound, and thus we do not yet know the value of  $[x_{d_i} > 0]$  (i.e., the value of  $y_{d_i}$ ). However, when the problem is implemented in a constraint language such as CLP(R), the constraints  $y_{d_i} \geq 0$ ,  $y_{d_i} \leq 1$ , and  $y_{d_i} \leq x_{d_i}$  are sufficient to put bounds on  $V$ .

Whenever domain pruning imposes a constraint  $x_i \leq b_i'$  where  $b_i' = 0$ ,  $y_i$  is forced to 0, and since  $V$  depends on in part on  $x_i$ , the maximum value of  $V$  is adjusted accordingly. That is, implicit in  $V$  is an upper bound, and we can use  $V$  both for optimization and for pruning the search space. Clearly, the upper bound for the solution under consideration must always be

greater than any known lower bound  $L$  or there is no point in continuing down this solution path, as expressed by the constraint  $V > L$ . Most naively,  $L$  can be given an initial value of 0. At the bottom of a search path,  $V$  gives the actual value of the solution, and the lower bound is reset to this new best answer so far. We will show that it is possible to produce better lower bounds during the search.

In the presence of our value functions where the value of one item type may be dependent on the presence of another, domain pruning combined with the upper bound constraint is effective in reducing the search space. Consider the following example:

$$\begin{aligned} \mathbf{w} &= [20,100,10,15], \quad \mathbf{b} = [3,3,3,3] \\ f_1 &= [x_4 > 0] * x_1 * 20 \quad f_2 = 100 * x_2 \\ f_3 &= 10 * x_3 \quad f_4 = 15 * x_4 \\ W &= 350 \end{aligned}$$

Say that we already have found the solution  $x_1 = 3, x_2 = 2, x_3 = 3, x_4 = 3$ , with a value of 335 (and we arbitrarily choose to explore branches from  $x_i = b_i$  down to  $x_i = 0$ ). When we backtrack to  $x_1 = 2$  and  $x_2 = 3$  and prune the domains of  $x_3$  and  $x_4$ , we get bounds of  $b_3' = 1$  and  $b_4' = 0$ . That is,  $x_4 \leq 0$ , which means  $y_4 = 0$  as well (i.e., no value is given for item type 1), so the upper bound is now 310 and further search from this point is unnecessary.

### 3.3. Computing Multiple Lower and Upper Bounds in a Hybrid Approach

#### 3.3.1. Lower Bounds without Sorting

In the classic 0/1 Knapsack Problem, a straightforward way to get a lower bound is to look ahead down the current solution path and take, in order, the items encountered until the weight limit is exceeded. The last item is then removed from the knapsack and the value of the items still in is computed. Since this is indeed a possible solution, it constitutes a lower bound for the current solution path, though clearly a better one on that path could exist.

In a hybrid approach, this lower bound computation can be combined with the local propagation and domain pruning of the constraint-based algorithm. That is, while retaining the basic depth-first search strategy, at each level of the search tree we can generate a lower bound for each sibling node at that level using the simple lookahead procedure just described. As before, the upper bound is calculated on the basis of domain pruning. Then the upper bound for a given node can be compared to the maximum of all the lower bounds at that level, and if the upper bound is less than the maximum, a solution need not be pursued from this node.

For a simple example of how the upper bound can be less than a neighboring lower bound, consider the following:

$$\begin{aligned} \mathbf{w} &= [10,10], \quad \mathbf{b} = [2,2] \\ f_1 &= [x_2 > 0] * x_1 * 10 \\ f_2 &= 20 * x_2 \\ W &= 20 \end{aligned}$$

Say that we first put 2 of item 1 in the knapsack. At that point, our upper bound is 0, since domain pruning shows that we cannot take any of item type 2, and thus  $f_1$  dictates that we get no value for item type 1. The lower bound for the neighboring node, however, (where we try taking only 1 of item type 1) is 30. This is based on a lookahead where we see that after taking just one of item type 1, we can take 1 of item type 2 without going over the weight limit.

### 3.3.2. Lower and Upper Bounds with Sorting

Similar to the lower bound, an upper bound for a solution path can be computed by looking ahead down the path and taking, in sorted order, the items encountered until the weight limit is exceeded. Then the last item is removed and the value of the items is computed, with an additional fraction of the offending item added back in, that fraction being precisely the fraction of the weight of the item that would fit in the knapsack. This method works under the assumption that items are sorted in decreasing order according to the value/weight ratio [Horowitz, Sahni, and Rajasekaran].

Our problem is complicated by the presence of a cost limit and non-constant value functions. Sorting items is more difficult because not all items of type  $i$  necessarily have the same value. As a result of type 1 value functions, it is possible that the first instance of type  $i$  placed in the knapsack is more valuable than the first instance of item type  $j$ , but then the first instance of item type  $j$  might at the same time have greater value than the second instance of item type  $i$ . Thus, we must sort the items individually rather than grouping them by type. Moreover, we sort twice, once based on the value/weight ratio and once based on the value/cost ratio. In each of these sorts, inter-type dependencies among the values of types is an additional complication, since the value of an item may not be known until the end of the search path, making it impossible to sort the items statically. Our solution to this difficulty is to sort the items individually and without regard to dependencies, and then to use the dependencies in an optimistic manner to compute the upper bound. That is, at level  $j$  of the search tree, for each item type  $i$  defined by a type 0 or type 1 function (resp. type 2 function) and dependent on another type  $d_i$  where  $d_i > j$ , we assume  $x_{d_i} > 0$  (resp.  $x_{d_i} = 0$ ) in computing  $f_i$ . During our lookahead as an upper bound is computed, we tentatively place individual items in the knapsack in sorted order, noting which have been chosen as we do so, and using the weight limit (resp. cost limit) as the limiting factor along with the list sorted by value/weight (resp. value/cost). We then compute the value for these items so that our total value is greater than or equal to the actual value we would get for these items, ensuring a legitimate upper bound.

It should be clear that both sorts, one based on value/weight and one based on value/cost, yield upper bounds during the lookahead. This is true because the solution space to the problem with both a weight and a cost limit is a subset of the solution space of the problem with a weight limit alone or a cost limit alone.

Two lower bounds can be computed from the same two sorted lists. The difference, however, is that during the lookaheads, *both* the weight limit and the cost limit must be used to bound the items placed in the knapsack. This is clearly necessary, since the lower bound must represent the value of a feasible solution. Once we have two upper bounds and two lower bounds, we use the smaller of the upper bounds and the greater of the lower bounds for the upper bound to lower bound comparison.

We have applied this approach in our C implementation of the *Oregon Trail Knapsack*, combining it with the domain-pruning described above. While looking ahead and computing the bound sacrifices some of the declarative nature of a constraint-based implementation, it has the advantage of providing tighter bounds in many cases. For all nodes at a level, if any node has an upper bound lower than the maximum known lower bound, then we do not need to pursue a search path from that node. This gives us another way to bound the search for significant pruning of the search space.

## 4. Implementation

We have implemented two basic versions of this problem, one in CLP(R) [Jaffar et al.], and the other a simulation of the constraint-based system in C. Variations of these involve using different types of lower and upper bounds, based either on sorting, domain pruning, or both. Table 1 summarizes the features of the problem variants. *ks\_all* and its variants were implemented only in C because of the procedural nature of the sort and lookahead.

Table 2 summarizes the different types of input programs run in our initial experiments. We began with small inputs ( $n = 30$  and  $b_j = 3$  to 5) and a variety of combinations of function types and dependencies, looking for trends. type *a* input data consists of only type 0 functions and no dependencies, while type *d* input data consists of 40% dependencies, 50% type 0 functions, 30% type 1 functions, and 20% type 2 functions.

Table 3 gives our initial results in terms of number of nodes visited. Table 4 gives the corresponding execution times. *ksnaive* is the least intelligent implementation, offered only as a baseline. We did not run it on large input sizes because of the obvious inefficiency.

**Table 1. Program Variants**

Name	Upper Bound from Domain Pruning (Section 3.1)	Lower Bound from "Best Answer So Far" (Section 3.2)	Lower Bound from Lookahead Without Sort (Section 3.3.1)	Upper Bounds from Lookahead With Sort (Section 3.3.2)	Lower Bounds from Lookahead With Sort (Section 3.3.2)
<i>ksnaive</i>	no	no	no	no	no
<i>ks</i>	yes	yes	no	no	no
<i>ks2</i>	yes	yes	yes	no	no
<i>ks_all*</i>	yes	yes	no	yes	yes
<i>ks_all2</i>	yes	yes	no	yes	no
<i>ks_all3</i>	no	yes	no	yes	yes
<i>ks_all4</i>	yes	yes	no	yes	yes

\*Note that *ks\_all4* differs from *ks\_all* only in variable ordering.

Both the C and CLP(R) implementations are based on a recursive depth-first search, and thus the number of nodes traversed in the search trees is the same for each with each of the variants listed, with the exception of the *ks\_all* variants, which we implemented only in C due to their procedural approach to sorting and lookahead. The execution times are from the C programs. Optimization is simulated in CLP(R) by writing new solution values to a file and reading the most recent value when the lower bound constraint is checked.

**Table 2. Types of Input**

Input Type	% Dependencies	% Type 0 Functions	% Type 1 Functions	% Type 2 Functions
type <i>a</i>	0	100	0	0
type <i>b</i>	80	50	50	0
type <i>c</i>	80	30	20	50
type <i>d</i>	40	50	30	20

**Table 3. Experimental Results (in nodes) for Programs**

Input Type	ksnaive	ks	ks2	ks_all	ks_all2	ks_all3	ks_all4
10a	15957	868	861	20	30	23	105
10b	20788	880	868	46	46	54	396
10c	26499	970	956	99	99	235	411
10d	17728	411	397	19	22	24	178
20a	14398307	234611	243609	242	245	283	3754
20b	14398307	98274	98257	33517	33521	460924	53521
20c	14398307	47800	50743	9451	9451	50235	22271
20d	14398307	128067	128080	1266	1266	2849	12155
30a	–	1553576	1553533	5240	5243	5934	39917
30b	–	452485	452430	120928	120978	4389400	537992
30c	–	1483741	1483706	59783	59783	143286	107175
30d	–	820865	820797	2363	2002	2363	9711

**Table 4. Experimental Results (in seconds) for Programs**

Input Type	ksnaive	ks	ks2	ks_all	ks_all2	ks_all3	ks_all4
10a	.31	.05	.10	.00	.00	.00	.01
10b	.90	.06	.14	.01	.01	.01	.04
10c	.99	.07	.17	.01	.00	.02	.04
10d	.42	.03	.07	.00	.00	.00	.02
20a	410.05	21.69	36.43	.04	.03	.05	.51
20b	680.43	12.37	20.48	5.46	3.82	49.41	9.16
20c	662.01	6.74	11.87	1.55	1.02	5.32	3.49
20d	612.54	16.14	26.66	.17	.12	.29	1.74
30a	–	261.18	441.46	.97	.66	.90	6.81
30b	–	85.69	189.52	25.92	18.96	622.09	114.54
30c	–	383.31	577.32	11.81	8.29	19.97	20.54
30d	–	168.57	255.29	.43	.29	.41	1.82

## 5. Conclusions

The CLP(R) implementation proved to be more natural and easier to adapt to new value functions, with the limitation that it cannot handle non-linear functions. More precisely, non-linear equations are delayed until variable bindings make them linear. In the presence of type 1 value functions, this limits the amount of pruning achieved by the  $V > L$  constraint. For example, say that at some point in execution, we have chosen zero of item type 1, zero of item type 2, and from domain pruning we know that  $x_3 \leq 1$ . The value functions are as follows:

$$f_1 = x_1 * [x_2 > 0] * 10$$

$$f_2 = 50 * \left(1 - \left(\frac{1}{2}\right)^{x_2}\right)$$

$$f_3 = 30 * \left(1 - \left(\frac{1}{2}\right)^{x_3}\right)$$

At this point,  $f_1 = 0$  and  $f_2 = 0$ . Given that we are using a monotonically increasing type 1 value function for  $f_3$  and we know that  $0 \leq x_3 \leq 1$ , we can see by inspection that  $f_3 \leq 15$ . The

constraint system, however, has no specific value for  $x_3$ , and thus it delays evaluation of  $f_3$ , resulting in a failure to recognize the upper bound of 15. A solution to this problem is to impose new constraints,  $V' = f_1(x_1, x_{d1}) + f_2(x_2, x_{d2}) + f_3(b_3', x_{d3})$  and  $V \leq V'$ , where  $b_3'$  is a new upper bound on  $x_3$  found through domain pruning.

The restriction to non-linear constraints aside, we found the CLP(R) implementation much easier to program, maintain, and modify with new constraints and value functions. To simulate the constraint system in a procedural language, we had to "manually" add to or subtract from the bounds whenever it was recognized that a new bound on some  $x_i$  had been established through domain pruning, or a decision to take or not take a certain item type was made. In a constraint-based language, the value functions can be stated as constraints in the query at the outset of the problem, and the upper bound is in a sense self-maintaining as increasingly tighter bounds are placed on the  $x_i$ 's on which  $V$  depends. Also, each type of value function in the C implementation must be handled as a special case when bounds and total values are computed, making the introduction of new functions difficult.

From our experiments, we observe that the two types of upper bounds used -- one based on sorting and one based on domain pruning -- do not necessarily prune the same branches of the search space. The sorted upper bound is generally tighter, since domain pruning is localized to a one-step lookahead. That is, from domain pruning we may know that  $x_i \leq b_i'$  for  $1 \leq i \leq n$ , so we essentially sum the value functions where  $x_i = b_i'$  to get the upper bound. But while  $b_i'$  items of type  $i$  may still fit in the knapsack, it is unlikely that  $b_i'$  items of all item types  $i$  for  $1 \leq i \leq n$  will fit together in the knapsack, and thus the upper bound is in many cases not tight. Where there are few dependencies in the functions (i.e., the value of one item type does not depend on the presence of another), the sorted upper bound is generally better, since it considers putting into the knapsack only relatively valuable items one at a time until the weight limit is exceeded. In the presence of dependencies, however, the pruned upper bound can be better, as in the simple example below.

$$\begin{aligned} \mathbf{w} &= [20, 10, 20], \\ f_1 &= [x_3 > 0] * x_1 * 100 \\ f_2 &= 50 * x_2 \\ f_3 &= 25 * x_3 \\ W &= 30 \end{aligned}$$

Say that we have just set  $x_1 = 1$ . An upper bound after pruning the search space would be 50. An upper bound based solely on a sort of the value/weight ratio would be 150. Here the upper bound based on pruning is better. Our experiments bear out the fact that the two kinds of upper bounds can work in different ways to prune the search space, as evidenced by the fact that while *ks* and *ks\_all3* are both superior to *ksnaive*, the two kinds of upper bounds combined in *ks\_all* and *ks\_all2* result in the fewest nodes visited in the search tree.

Our overall conclusions are these: We generally get our best performance on *ks\_all2*, which includes two kinds of upper bounds but only one kind of lower bound. While *ks\_all* traverses fewer nodes than *ks\_all2* in some cases, the time spent by *ks\_all* computing the lower bound based on sorting does not result in a relative time savings. (We need a better method of sorting in the presence of functions and dependencies to get a better lower bound.) The fact that *ks\_all2* (which has domain pruning) generally performs better than *ks\_all3* indicates that domain pruning is worth the effort. *ks\_all* visits fewer nodes than either *ks\_all3* (which has only the sorted upper bound) and *ks* (which has only the "pruned" upper bound), showing that the two

upper bounds can cut different branches of the search tree. *ks\_all* and *ks\_all2* perform better where there are relatively fewer dependencies or special value functions. However, this is not the type of problem we would generally expect in the Oregon Trail Knapsack, where the value of many types of items will depend on whether other types of items are taken. In our estimation, types *b* and *c* input represent a more realistic problem instance. On this type of input, *ks*, which uses only domain pruning to get the upper bound, performs quite well enough to be of practical value. However, *ks2*, which looks for a lower bound across sibling nodes, was not found to be more efficient than *ks*. It eliminates very few nodes as opposed to *ks*, and at a cost of almost doubling the execution time in some cases. We conclude that while *ks\_all* and *ks\_all2* can visit fewer nodes in the search tree, *ks* -- the purest constraint-based approach -- has the advantage over the other two of being more declarative and adaptable, allowing new value functions and constraints to be added easily.

Our on-going work includes continuing with the analysis of empirical results, varying more systematically the weight and/or cost limits, cost constants, weight constants, value constants, and percentages of dependencies and function types. We also are working on improving the upper bound in the *ks\_all* variants by integrating the cost and weight limits into one sort. Another idea for improving the upper bound is to do a topological sort of item types based upon their dependencies. Variables would then be ordered in the search tree according to this sort. Finally, we would like to implement our algorithms in other constraint languages to determine their efficiency and the extent to which these languages can accommodate the hybrid algorithms with their existing global constraints, optimization and domain pruning features.

## References

- E. Balas and E. Zemel. An algorithm for large zero-one knapsack problems. *Operations Research* 28, 1130-1154, 1980.
- Y. Caseau and F. Laburthe. Solving small TSPs with constraints. *Logic Programming: Proceedings of the Fourteenth International Conference on Logic Programming*. Lee Naish, ed. MIT Press, 1997.
- G. B. Dantzig. Discrete variable extremum problems. *Operations Research* 5, 266-277, 1957.
- M. L. Fisher. The Lagrangian relaxation method for solving integer programming problems. *Management Science* 27, 1-18, 1981.
- R. L. Graham, D. E. Knuth, and O. Patashnik. *Concrete Mathematics*, 2nd ed., Addison-Wesley, 1994.
- E. Horowitz, S. Sahni, and S. Rajasekaran. *Computer Algorithms: C++*. Computer Science Press, 1997.
- J. Jaffar, S. Michaylov, P. J. Stuckey, and R. Yap. The CLP(R) Language and System. *ACM Transactions on Programming Languages and Systems* 14, 3 (July 1992), 339-395.
- N. Maculan. Relaxation Lagrangienne: le probleme du knapsack 0-2. *INFOR (Canadian Journal of Operational Research and Information Processing)* 21, 315-327, 1983.
- S. Martello, P. Toth. *Knapsack Problems: Algorithms and Computer Implementations*. John Wiley & Sons, 1990.
- G. Pesant and M. Gendreau. A view of local search in constraint programming. *Principles and Practice of Constraint Programming -- CP96*, 353-366, 1996.

- H. Simonis. "A problem classification scheme for finite domain constraint solving."  
*Proceedings of the CP'96 Workshop on Constraint Programming Applications*, 1-25,  
1996.
- P. Van Hentenryck. *Constraint Satisfaction in Logic Programming*. MIT Press, 1989.
- M. Wallace. *Survey: Practical applications of constraint programming*. Technical Report, IC  
Parc, September 1995.