A Dynamic Programming Approach for Consistency and Propagation for Knapsack Constraints

Michael A. Trick (trick@cmu.edu)
GSIA - Carnegie Bosch Institute
Carnegie Mellon University
Pittsburgh, PA USA
15213

Abstract.
Knapsack constraints are a key modeling structure in constraint programming. These constraints are normally handled with simple bounding arguments. We propose a dynamic programming structure to represent these constraints. With this structure, we show how to achieve hyper-arc consistency, to determine infeasibility before all variables are set, to generate all solutions quickly, and to update the structure after domain reduction. Preliminary testing on a difficult set of multiple knapsack instances shows significant reduction in branching, though an effective implementation is needed in order to reduce computation time.

Keywords: Global Constraints, Dynamic Programming, Knapsack Constraints

1. Introduction

A (two sided) knapsack constraint is a linear constraint of the form $L \leq ax \leq U$ where L and U are scalars, $x = [x_1, x_2, ..., x_n]$ is an n-vector of variables, and $a = [a_1, a_2, ..., a_n]$ is an n-vector of coefficients. Each variable x_i is to take one value from a finite domain D_i (assumed to be a subset of the non-negative integers) while each coefficient a_i is a nonnegative integer.

Knapsack constraints form the backbone for many types of discrete formulations. Many standard integer programming formulations are nothing more than a series of knapsack constraints. Crowder, Johnson and Padberg (1983) used a thorough understanding of individual knapsacks to solve general integer programs. That paper revolutionized the constraint generation method for integer programming.

Despite their importance, knapsack constraints are generally handled in a straightforward way in most constraint programming systems. Domain reduction is generally limited to simple bounding arguments. For instance, if $d_i \in D_i$, and $a_i d_i > U$, then clearly d_i can be removed from D_i. Similarly, if every variable but one is set to the highest value in its domain, the remaining variable must be set high enough to make the total at least L.

This perhaps overly simple handling is done despite the fact that it only takes a small number of small knapsacks to turn an otherwise tractable formulation into a difficult instance. For instance, Cornujols and Dawande (1998) give a class of problems called the "Market Split Problem" where problems with as few as 5 knapsacks and 40 variables are resistant to solution by almost any technique.

Given the centrality of the knapsack as a modeling construct and the difficulty that even small knapsacks can cause, we propose a dynamic programming representation for handling knapsacks. This approach has the following characteristics:

1. It allows domain reduction in order to ensure hyper-arc consistency for a single knapsack. In other words, it will take the Di and remove any value for which there is no setting of the remaining variables to satisfy the knapsack.

2. It quickly identifies infeasibility even when the domains of most of the variables are not yet singletons.

3. The structure is easily updated after branching or variable instantiation, so the representation can be used throughout the constraint programming process.

4. The structure can be augmented to provide checks on the feasibility of a set of knapsack constraints.

5. The resulting structure can effectively generate all feasible solutions to the knapsack constraint.

This approach uses a carefully implemented standard dynamic programming formulation of the knapsack constraint in order to achieve the above objectives. It is important to note that traditional IP approaches, such as cutting planes or linear programming-based branch and bound, are often limited in their ability to achieve hyper-arc consistency or are expensive to update after domain reduction.

In Section 2, we give the basic dynamic programming structure, and show how it can be used to achieve hyper-arc consistency for the knapsack problem. We also discuss how to effectively generate all solutions to the constraint. Section 3 discusses updating the structure during the branching process. Section 4 addresses the problem of handling multiple knapsack constraints. In section 5, we give some computational results that show the effectiveness of this approach in handling hard multiple-knapsack problems. We conclude by outlining the advantages and disadvantages of this approach and by showing where the approach might be most useful.

2. Dynamic Programming and Knapsacks

Dynamic programming was one of the first proposed methods for solving knapsack optimization problems. Bellman (1957) first used the phrase "dynamic programming" in the context of solving knapsacks. Much follow-on work for this method occurred in the 1960s (see Nemhauser and Wolsey, 1988 for references). For the following exposition, we will outline the dynamic programming formulation for the 0-1 knapsack problem, where $D_i = \{0, 1\}$ for all i. Handling more general domains is a straightforward generalization of the notation.

Define a function $f(i, b)$ equal to 1 if the variables $1 \ldots i$ can exactly fill a knapsack of size b, and 0 otherwise, where i ranges from 0 to n and b ranges from 0 to U. We define the dynamic programming recursion as follows

$$f(0, 0) = 1$$

$$f(i, b) = \max\{f(i-1, b), f(i-1, b-a_i)\}$$

We can visualize this as a network with one node for every (i, b). Edges go from $(i-1, b)$ to (i, b') between nodes with a 1 value for f. An edge corresponds to variable i taking on value 0 (an edge from $(i-1, b)$ to (i, b)) or value 1 (an edge from $(i-1, b-a_i)$ to (i, b)). For the knapsack $10 \leq 2x_1 + 3x_2 + 4x_3 + 5x_4 \leq 12$, the knapsack graph is shown in figure 1.

Rows in the above graph correspond to b values (0 through 12) while columns represent i values. A shaded node corresponds to a node with f value of 1.

Working forward from the node (0,0), we can determine whether we can reach one of (4,10), (4,11), or (4,12) in order to prove that there exists a feasible solution to the knapsack.

For constraint programming, we would like more than just determining feasibility. We can do domain reduction by working backwards from the "goal" nodes ((4,10), (4,11), and (4,12) in our example), determining which nodes can lead to a feasible knapsack. In doing so, we determine all feasible intermediate nodes. Any node that can not reach a goal node can be deleted from our graph. We call the result the reduced graph. This gives the graph in figure 2.

Note in this example, there is no longer any edge that corresponds to having x_4 take on value 0. We can therefore remove 0 from the feasible domain of x_4.

The reduced graph is formally created by defining a recursive function $g(i, b)$ equal to 1 if the variables $i+1 \ldots n$ can fill some knapsack

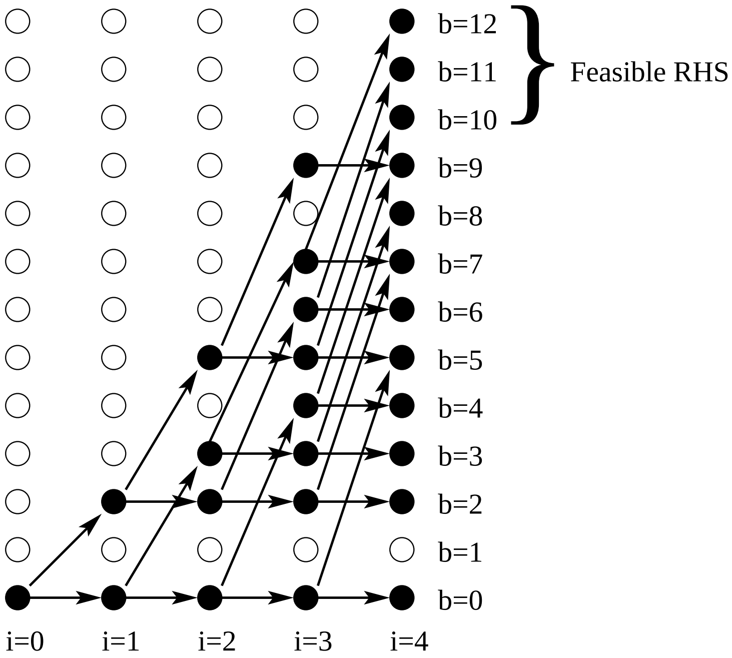

Figure 1. Knapsack Graph

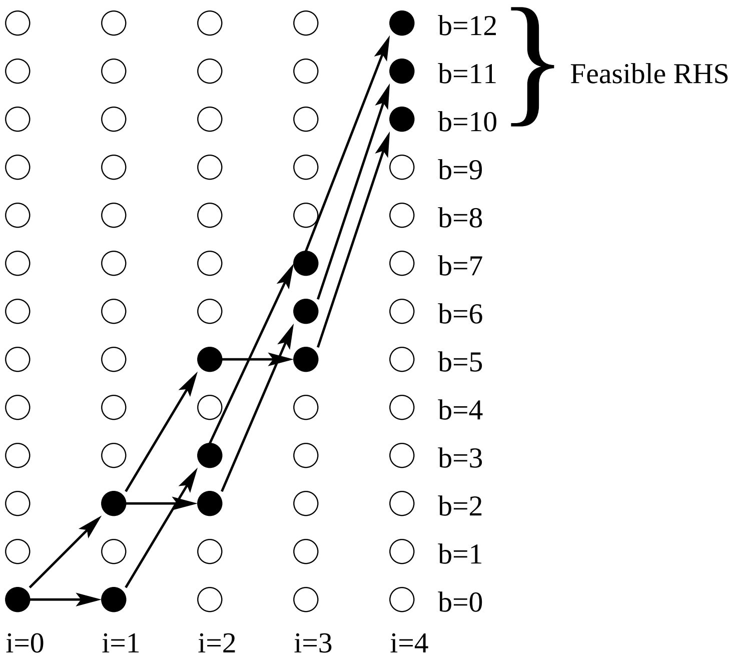

Figure 2. Reduced Graph

of size from L to U, given variables $1 \dots i$ have already filled exactly b. Again i ranges from 0 to n and b ranges from 0 to U.

This process is formalized in the algorithm Knapsack-hyper-arc-consistency:

From `Knapsack-hyper-arc-consistency`, we create the reduced graph with nodes (i, b) for i from 0 to n and b from 0 to U. There is an edge from $(i-1, b)$ to (i, b') if and only if $f(i-1, b) = 1$, $g(i, b') = 1$ and $b' - b \in D_i$, Such an edge is given label d_i.

The following lemmas are immediate from the definitions of f and g. We call the nodes (n, b) for $L \leq b \leq U$, goal nodes.

LEMMA 1. *Any path from* $(0, 0)$ *to a goal node in the reduced graph* G *from Knapsack-arc- consistency corresponds to a feasible solution to the knapsack constraint.*

LEMMA 2. *If variable* i *has no edge with label* d_i *in* G*, then* d_i *can be removed from* D_i *without affecting the set of feasible solutions to the knapsack constraint.*

We say that the knapsack constraint is hyper-arc consistent with domain D if for each x_i and $d_i \in D_i$, if x_i is assigned value d_i then there is an assignment of all other variables so that both the domain restrictions and the knapsack constraint is satisfied (see, for instance, Marriott and Stuckey, 1998).

LEMMA 3. *Once infeasible values have been removed from each domain, the resulting domains are hyper-arc consistent with the knapsack constraint.*

Example: For the knapsack is $80 \leq 27x_1 + 37x_2 + 45x_3 + 53x_4 \leq 82$ and the domains of each of the variables is $\{0, 1, 2, 3\}$, the three feasible solutions are $x_1 = 3$; $x_1 = 1, x_4 = 1$; $x_2 = 1, x_3 = 1$ (with the other variables at 0). The resulting domains are $\{0, 1, 3\}$, $\{0, 1\}$, $\{0, 1\}$, $\{0, 1\}$ respectively. Note that no simple bounds argument can reduce the domain of x_1 to remove the value 2.

Formally, this algorithm is only psuedo-polynomial: it's complexity is $O(nU^2)$. The space required is of the same complexity. This suggests that this algorithm should only be used in cases where U is not too large.

In practice, the reduced graph is often much smaller than this worst case value. An effective implementation of this graph would not create nodes for all (i, b) values. Instead, a linked list can be used that stores only nodes for which both the $f(i, b)$ and $g(i, b)$ values are 1.

```
Knapsack-hyper-arc-consistency (A,L,U,D)
/* A is a vector of knapsack coefficients a(1)..a(n),
   L is the lower bound,
   U is the upper bound,
   D is the set of feasible domains for each variable x(n) */

f(0,0) = 1
for (i=1;i<=n;i++) {
   for (b=0;b<=n;b++) {
      if (f(i-1,b)==1) {
         foreach (d in D(i)) {
            if (b+a(i)d <= U) {
               f(i,b+a(i)d) = 1;
            }
         }
      }
   }
}

if (f(n,b)=0 for all L<=b<=U) then return unsatisfiable;
g(n,b) = 1 for all L<=b<=U
for (i=n-1;i>=0;i--) {
   for (b=0;b<=n;b++) {
      if (g(i+1,b)==1) {
         foreach (d in D(i+1)) {
            if (b-a(i+1)d >= 0) {
               g(i,b-a(i+1)d) = 1;
            }
         }
      }
   }
}

for (i=1;i<=n;i++) {
   D(i)=D(i)i\{ d: there exists no b such that f(i-1,b)= 1 and
                                                    g(i,b+a(i)d) =1}
}
```

Figure 3. Knapsack-hyper-arc-consistency

It should be noted that many more complicated dynamic programming formulations have been proposed for the knapsack problem. These methods typically are used when the goal is to optimize a linear function relative to a knapsack constraint. In these dynamic programming formulations, clever reduction techniques are developed based on characterizations of optimal solutions. In our case, we are concerned not with optimality but with characterizations of feasibility. For this, the more complicated formulations are not applicable.

In some applications, the set of feasible values for the knapsack is not a simple range $[L, U]$. For instance, parity considerations may require an even value, or the range may have infeasible values embedded within it. It is straightforward to modify this approach to handle such requirements simply by appropriately defining the "goal nodes" of the recursion.

Finally, we can generate all solutions to the knapsack constraint by finding all paths in G. Since G is acyclic, this can be done easily via standard methods. For instance, simply exploring G in a depth-first method will generate all paths in time linear in the length of the output. It is also straightforward to determine the number of feasible solutions without generating them by a recursion similar to the recursion given above.

3. Updating the Knapsack Graph

In the course of constraint programming algorithms, the domains of the variables will generally be reduced, either due to constraint propagation or due to branching on domain values for a variable. Rather than just resolving the knapsack with each change in domain, we can update the Knapsack Graph directly.

Given a reduced graph and a domain value d_i for x_i, we would like to create the reduced graph where d_i is no longer a feasible value for x_i. First note that the reduced graph is completely determined by the f and g function values. Second, clearly no $f(i', b)$ value is affected for $i' < i$, nor are $g(i'b)$ values affected for $i' > i$.

How can we recalculate $f(i', b)$ for $i' > i$ (the calculation for $g(i', b)$ for $i' < i$ is similar)? The simplest manner is to keep a list of nodes whose f value might have changed. Initially, this set of "possibles" contains all nodes $(i + 1, b)$ where $f(i, b - d_i) = 1$. Given a "possible" node (i', b), to determine if $f(i', b)$ still equals 1, it is necessary to check if $f(i' - 1, d'_i - 1) = 1$ for any $d'_i - 1 \in D'_i - 1$. If so, then $f(i', b)$ remains 1 and (i', b) can be removed from the possible set. If not, then $f(i', b)$

gets set to 0, and $(i'+1, b+d'_i)$ is added to the possible set for all $d'_i \in Di'$.

This approach is no worse than recreating the reduced graph and may be much better. In particular, in many situations, the set of possibles contains only a small fraction of the reduced graph, so only a small portion needs to be updated.

4. Handling Multiple Constraints

Multiple knapsack constraints occur either directly in a formulation or in a maximization problem where the objective function is linear. The results given for a single knapsack constraint can be generalized to multiple knapsacks in two different ways: combining multiple knapsacks into one or using the reduced graph to deduce infeasible knapsacks.

4.1. Combining into One Knapsack

A standard approach to handling multiple knapsacks is to combine them into a single knapsack. For example, given two knapsacks:

$$L_1 \leq ax \leq U_1$$
$$L_2 \leq a'x \leq U_2$$

we can multiply the second knapsack by some value α and add the two constraints:

$$L_1 + \alpha L_2 \leq (a + \alpha a')x \leq U_1 + \alpha U_2$$

A third constraint would be multiplied by α^2, a fourth by α^3 and so on.

As long as α is large enough, this modification does not create any additional solutions. Unfortunately, "large enough" can be quite large: if a_{max} and d_{max} are the largest coefficient value and the largest domain value respectively, then setting α to $na_{max}d_{max}$ is sufficient. With even a small number of constraints, the size of the knapsack and the commensurate memory requirements can get overwhelming.

If we choose α large enough, this knapsack can be handled exactly as given in the previous section. In this case, we have a strong test for infeasibility, a hyper-arc consistent domain reduction algorithm, and a method for updating when the domain of a variable changes.

If we choose a smaller α, the constraint is still valid but there may be solutions that satisfy the constraint that do not satisfy the original problem. In this case, we get weaker forms of our results. If the aggregate knapsack is infeasible, then the original set of knapsacks was infeasible also, though the converse might not be true. Similarly, if we do domain reduction based on the aggregate knapsack, the result is still valid, but we no longer have hyper-arc consistency: we may have a domain value for which there is no completing solution to the original knapsacks. The update remains easy, and would be faster than that for larger α due to the smaller reduced graph.

4.2. Deducing Bounds for One Knapsack Based on Another

Given the computational expense involved in creating the reduced graph for a knapsack problem, it would be advantageous to use that structure to generate better bounds on other values. The simple structure of the reduced graph makes it easy to calculate bounds on possible values for other knapsack constraints.

Suppose we are given two knapsack constraints

$$L_1 \le ax \le U_1$$

$$L_2 \le a'x \le U_2$$

and we have created the reduced graph G for the first knapsack. We can calculate the maximum value of $a'x$ over all feasible solutions in G as follows:

define $u(i,b)$ be the maximum value of $a'x$ for each node (i,b) in G using just the variables $1 \ldots i$. We can calculate this as follows:

$u(0,0) = 0$
$u(i,b) = \max\{u(i-1, b-d_i)$: there is an edge from $(i-1, b-d_i)$ to (i,b) in $G\}$.

We can similarly define $l(i,b)$ as the minimum value of $a'x$ by replacing the "max" with a "min" in the recursion.

In order for there to be a solution in G that satisfies $L_2 \le a'x \le U_2$, there must be a goal node (n,b) in G such that the intersections of the ranges $[l(n,b), u(n,b)]$ and $[L2, U2]$ is nonempty. If it is empty for all goal nodes, then the pair of knapsacks is not simultaneously satisfiable.

These values can be used to further decrease the size of the reduced graph. Let $u'(i,b)$ be the maximum value of $a'x$ using just variables $i+1 \ldots n$ in G, and $l'(i,b)$ be the minimum value of $a'x$ using just variables $i+1 \ldots n$ in G. These can be calculated in a similar manner to how we calculated u and l. Then $u^*(i,b) = u(i,b) + u'(i,b)$ and $l^*(i,b) = l(i,b) + l'(i,b)$ give the maximum and minimum value for for $a'x$ for any node (i,b). If the range between $l^*(i,b)$ and $u^*(i,b)$ does not intersect $[L_2, U_2]$, then (i, b) and all its incident arcs can be deleted from G, with the possibility for domain reduction.

Of course, these calculations can be done for multiple knapsacks, and each knapsack can take the role of defining the reduced graph.

Handling more general constraints is simply a matter of appropriately defining l and u values. The simple structure of the reduced graph makes it straightforward to find bounds relative to the feasible solutions to a knapsack constraint.

5. Computational Results

To test the usefulness of this dynamic programming approach to knapsack constraints, we developed a system to solve the feasibility version of the Market Split problem of Cornujols and Dawande (1998). Their description of this problem comes from Williams: A large company has two divisions D1 and D2. The company supplies retailers with several products. The goal is to allocate each retailer to either division D1 or division D2 so that D1 controls 40% of the company's market for each product and D2 the remaining 60%. This can be formulated asking whether the system of equations

$$\sum_j a_{ij} x_j = b_i \text{ for all } i = 1 \ldots m$$

$$x_j \in \{0, 1\} \text{ for all } j = 1 \ldots n$$

where n is the number of retailers, m is the number of products, a_{ij} is the demand of retailer i for product j, and the right hand side b_i is given by the desired market split.

In Cornuejols and Dawinde, they generated random instances where a_{ij} was generated as a uniform integer between 0 and 99, $n = 10(m-1)$, and b_i was set to $\lfloor 1/2 \sum_j a_{ij} \rfloor$. These instances are almost always infeasible for under 6 constraints.

These instances were shown to be extremely difficult to solve by standard integer programming methods. Only recently have sophisticated approaches (Aardal et. al, 1999, seems to be the best approach) been developed to solve the 6 constraint, 50 variable instances. Even the 4 constraint, 30 variable instances are extremely difficult for standard methods. Working with five instances of this size problem, OPL Studio 3.1 (Ilog, 2000) gives the results in Table 1.

The dynamic programming formulation reduces the number of subproblems significantly only when domain reduction and/or multiple-knapsack capacity bounds are included.

Table I. Solving 4 constraint 30 variable instances

Method	Branches
Solver (Constraint Based feasibility testing)	1,715,993
CPLEX (Integer Programming Branch and Bound)	1,010,421
Dynamic Program without domain reduction	2,236,433
Dynamic Program with domain reduction	531,661
Dynamic Program with domain reduction and Multiple-knapsack capacity bounds	241,443

Key to making this a useful method is the efficient implementation of the technique. A straightforward implementation of the dynamic program with domain reduction gives a computation time of approximately 14,000 seconds, compared with 600 seconds for Solver and 204 seconds for CPLEX on average (times are on a 650 Mz PC running Windows 2000). Note that the approach of Aardal et al. (1999) can solve these problems in approximately one second.

Taking advantage of the sparcity of the reduced graph reduces the dynamic programming time by a factor of 6, putting the average time to 2157 seconds. While this time is still too long, further savings are possible with a more careful implementation.

The method of combining knapsacks outlined in section 4.1 can solve these problems very quickly by a "generate and test" approach: First we generate all solutions to the combined knapsack then we test each to determine feasibility to the individual knapsacks. Using a α value of 5 gave a knapsack formulation that had 19,560 solutions on average. These could be generated and checked for feasibility to the individual knapsacks in under 3 seconds. Since all solutions to the aggregate knapsack are generated, this approach is complete: it finds a solution to the collection of knapsacks if and only if at least one exists.

In addition to the branching improvements, and possible time improvements via "generate and test", there are other advantages to the dynamic programming approach. In particular, the ease at which all solutions to a knapsack can be generated cannot be mimicked by the other approaches. The knapsack approach can also quickly updates after domain reduction, which suggests applications in the likely situation where there are constraints other than the knapsack constraints. Like many other global constraints, this strong approach to handling knapsack constraints may show its strength only in hard problems that mix many types of constraints. And finally, the 0-1 knapsack problem may not be the best problem set to show off the strength of this approach:

CPLEX embeds much is what is known about 0-1 knapsacks, only some of which generalizes to variables with more complicated domains.

The dynamic programming approach to knapsack constraints is certainly not always appropriate. If the right hand side of the knapsack is very large, then the time and space requirements may be prohibitive. The updates required, even with the dynamic updating suggestion, are much more expensive than simpler "bounds" approaches. In models where the knapsacks are not the critical feature, it is clear that the expense of the update is unlikely to be worthwhile. But in cases where the problem is difficult, and where knapsacks are causing that difficulty, the dynamic programming approach can greatly improve the computational characteristics of the problem.

References

Aardal, K, R.E. Bixby, A.J. Hurkens, A.K. Lenstra, and J.W. Smeltink: 1999, "Market Split and Basis Reduction: Towards a Solution of the Cornujols-Dawande Instances" in G. Cornujols, R.E. Burkard, G.J. Woeginger (eds) *Integer Programming and Combinatorial Optimization, 7th International IPCO Conference*, Lecture Notes in Computer Science 1610, Springer-Verlag Berlin, pp 1-16

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

Cornujols, G. and M. Dawande: 1998, "A Class of Hard Small 0-1 Programs" in R.E. Bixby, E.A. Boyd, R.Z. Rios-Mercado (eds) *Integer Programming and Combinatorial Optimization, 6th International IPCO Conference.* Lecture Notes in Computer Science 1412, Springer-Verlag, Berlin, pp 284-293.

Crowder, H.P, E.L. Johnson, and M.W. Padberg: 1983, "Solving Large-Scale Zero-One Linear Programming Problems", *Operations Research* **31**, 803-834.

ILOG, Inc (2000), *OPL 3.1 Users Guide*

Marriott, K. and P.J. Stuckey: 1998, *Programming with Constraints: An Introduction*, MIT Press.

Martello, S. and P. Toth: 1990, *Knapsack Problems: Algorithms and Computer Implementations.* Wiley, Chichester, U.K.

Nemhauser, G.L. and L.A. Wolsey: 1988, *Integer and Combinatorial Optimization*, Wiley.

Williams, H.P: 1978, *Model Building in Mathematical Programming*, Wiley.