Leo Landa

Optimizing the Knapsack Problem

1 Foreword

This report was written in December of 2002, as a term paper in professor T.C.Hu's "Interesting Algorithms" class (CSE 203A at UCSD, Fall quarter of 2002). It took some time and additional research to reignite the interest in the results in this report, which led to its being registered as a technical report with UCSD in April, 2004. However, the content presented is still the same as it was in 2002.

Since 2002, only the following changes were made:

- The author's name was changed from *Lev Landa* to *Leo Landa*, due to an official change of name;
- The author's e-mail was changed from *llanda@cs.ucsd.edu* to *leo@leolan.com*; both are valid, the latter should simply be considered the primary address;
- This foreword and acknowledgements section were written;
- The index page was updated to reflect the change in section numbers and page numbers due to the foreword and acknowledgements.

2 Acknowledgements

It is impossible to overestimate the contribution of UCSD professor T.C.Hu to this report's being written and published. Professor T.C.Hu's magnificent lectures in several courses have given me the knowledge, desire, inspiration, and consistency to perform enough research to culminate in this report. Furthermore, professor T.C.Hu's involvement after the report was written is the only reason for the report to be published – without his never-ending support and additional research (in collaboration with professor M.T.Shing), this report would never see the light of day.

3 Introduction

The one-dimension unlimited-supply integer knapsack problem can be formulated as follows:

> Given an unlimited number of items of several types, each one characterized by its integer weight and integer value, and a knapsack of limited total capacity for weight, find a combination of items whose total weight does not exceed the knapsack capacity while yields the maximum total value.

It is known from previous research that for large enough knapsack capacities (boundaries), a trend will become obvious: every solution to the problem of boundary b will be a combination of solution to $(b\text{-}w_i)$ and the item i, given that item i is characterized by highest "density", or v/w ratio. The proof of this fact is beyond the scope of the paper.

This essentially means that the table of knapsack solutions, mapping boundaries to maximum achievable weight will become predictable and "periodic" (the differences between integer solutions and continuous solutions will repeat with a period equal to the weight of the most valuable item). This makes it possible to skip solving the knapsack problem for big enough boundaries, but rather calculate the result directly (fast) using previous knowledge.

However, the problem arises – what boundary is big enough?

This paper will go hunting for the answer.

3.1 Note

There should be clear distinction between two types of goals that could be sought while hunting for the magic boundary.

One would be enough information to produce the solution to the given knapsack problem for *any* boundary. Let's call this a *general* knapsack problem.

Another would be enough information to produce the solution for *a particular* boundary. Let's call it a *particular* knapsack problem.

As we shall see later on, these two problems are related, but not the same. However, the hunt is on for both.

4 Intuition

4.1 Definitions and assumptions

For every knapsack problem examined, let us define the following variables and terms:

- N – the number of types of items available;
- w_i – the weight of item of type i;
- v_i – the value of item of type i;
- *Density* – a real number denoting the ratio between the value and the weight of a certain type of item;
- p_i – the density of items of type i;
- b – the boundary (knapsack capacity);

Let us also always distinguish the item with the highest density. To make notation easy, let us always label the types of items in such a way that the type with index 1 is the one with highest density. In case of ties between two types, we will always pick the type with the smallest weight. Further ties will mean exact same

types of items, so will consider them the same. We shall call that type “Most Valuable Type”. We shall call items of that type “Most Valuable Item”, or “MVI”.

- *theta* – the difference between a continuous solution (which is always $b\ v_1 / w_1$) and integer solution;
- *General stop point* – the smallest value of b, such that for any boundary larger than b the solution is a combination of the solution to $(b - w_1)$ and one MVI.

Let us also formulate the assumptions about the knapsack problem. These have been proven previously and will be considered axioms in this paper:

1. For any knapsack problem, if the optimal solution for certain b contains at least one item of type x, then if that item is removed from the solution, the remainder is the optimal solution for $(b - w_x)$. This stems from the dynamic programming nature of the knapsack problem.
2. For large enough boundaries b, the solution will consist of the solution to $(b - w_1)$ and one MVI.
3. Any particular knapsack problem can be solved using dynamic programming, utilizing a table of size N by b. Let us call such a table *general dynamic table*. The whole purpose of the paper is to provide solutions to knapsack problems without building such a table, because this table is too big to be built for certain large enough boundaries (it also takes too much time – the whole purpose of this research is to reduce the time needed to solve knapsack problems).

4.2 The goal

Consider a certain knapsack problem, in which the weight of MVI is 10. Consider that the problem at hand is to find a solution for a boundary of 10,000,000,000.

It is obvious that there is no need to analyze the items any further. It is clear that 10,000,000,000 divides 10, the weight of MVI, without remainder, which means that the solution will consist *only* of MVI. 1,000,000,000 of them to be exact.

This simple example illustrates for some particular knapsack problems it *not necessary* to know the general stop point. Some more insight into the problem should reveal some more details about the knapsack problem that would render the general stop point not necessary for certain types of problems.

However, the concept "general stop point is necessary to produce solution to any particular knapsack problem without general dynamic table" is already compromised.

Therefore, our quest for solution as stated should NOT be the quest for the general stop point, because that information is not sufficient. However, our solution should shed light on what a general stop point is. In other words, our goal is bigger.

5 Gain

In order to organize the workings of the periodic knapsack table, we need to define a concept that will help in analyzing it. The concept of *theta* does not seem to be the best way of quantifying the "quality" of a solution, so we will introduce a different one – gain.

5.1 The definition and meaning

The concept of *theta* is good, because it gives some insight into the periodic nature of the knapsack problem. The periodic repetition of *thetas* is a necessary (but not sufficient) condition for general stop point.

However, *thetas* are not very intuitive. They are real numbers, which are hard to use in an integer problem. So, instead of the concept of *theta*, we shall use the concept of *gain*:

For any solution of a knapsack problem and a boundary *b*, yielding total value of V, the gain is defined as follows:

$$gain = V - (b - b_{mod} w_1)\, v_1 / w_1$$

For example, if *b* is 24, *w1* is 10, *v1* is 7, and *V* is 17, then:

$$gain = 17 - (24 - 24_{mod}10)\ 7 / 10 = 17 - 14 = 3$$

The intuitive meaning of *gain* is as follows – it is the numerical difference between two solutions of the problem for a given boundary: the one with all the items, and the one with only MVI. Hence the term "gain" – it is an "improvement" over solutions consisting of MVI only.

It is clear that it is possible to derive the value of *gain* from the value of *theta*, and vice versa. They both are essentially the measure of the quality of the solution (the smaller the *theta*, the better the solution; the larger the *gain*, the better the solution). But from this point on we shall use *gain* only.

Let us also call the gain of any optimal solution for some boundary as *optimal gain*.

5.2 Gain improvements

From the definition of *gain* we can conclude the following:

Lemma 1. *For any two solution candidates of any knapsack problem for any boundary the better solution will always have higher gain.*

It is simple: the gain is essentially the difference between the solution and MVI-only solution, hence the if one gain is better than another we have:

$$Gain_1 > Gain_2$$
$$V_1 - V_{MVI} > V_2 - V_{MVI}$$
$$V_1 > V_2$$

Lemma 2. *For any knapsack boundary, the optimal solution is the one with the highest possible gain.*

This is essentially Lemma 1 as applied to all solutions.

Lemma 3. *If the optimal solution of knapsack problem for some boundary b has gain of G, then the optimal solution of knapsack problem for boundary ($b + w_1$) has gain of at least G.*

The proof is simple. If there is some solution for boundary *b*, then by adding one MVI to it we get a solution to ($b + w_1$) whose gain is exactly *G*:

$$G = S_1 - V_{MVI1} = S_1 - (b - b_{mod}w_1)\ v_1 / w_1$$
$$G_2 = S_2 - V_{MVI2} = (S_2) - (\ (b2 - (b2_{mod}w_1)\ )\ v_1 / w_1$$

$$= (S_1 + v_1) - ((b + w_1 - ((b + w_1)_{mod} w_1) v_1 / w_1$$
$$= (S_1 + v_1) - (b + w_1 - b_{mod} w_1) v_1 / w_1$$
$$= (S_1 + v_1) - ((b - b_{mod} w_1) v_1 / w_1 + v_1)$$
$$= S_1 - (b - b_{mod} w_1) v_1 / w_1$$
$$= G$$

Combining this result with Lemma 2, we conclude that Lemma 3 is correct.

Lemma 4. *For any boundary b, the optimal gain is bound by the real value* $(b_{mod}w_1)v_1/w_1$.

This comes from the concept of density. The density of any solution is the total achieved value over total weight. It is a linear combination of densities of the items present in a solution. Since the MVI has the highest density, the density of the solution cannot be higher than the density of MVI. That in turn means that the gain on a MVI-only solution has to no more dense than the gain of continuous solution, so the maximum achievable gain cannot be higher than the difference of boundaries (covered by integer solution and MVI-only solution) times the density of MVI.

There are two important implications to this Lemma, which are discussed later (Implications 1 and 2).

Lemma 5. *For every infinite set of optimal solutions of the knapsack problem corresponding to the infinite set of all boundaries b such that* $b_{mod}w_1$ *is the same, the gain keeps improving until it reaches some maximum value, and that value remains constant for the rest of boundaries in the set.*

This comes directly from Lemmas 3 and 4: every gain of the set (except for the first one) is at least as high as the previous one (Lemma 3), and at the same time is upward-bound by a certain constant value (Lemma 4), which is the same for all boundaries in the set (by definition of the set).

So, from now on, we see that gains for certain solutions are somewhat grouped together. Every group (set of boundaries respective gains) has its own starting value of gain, which keeps improving as the boundary grows, until it reaches a certain maximum (note that the gain may stay the same for a while before it continues growing). Once that maximum is reached, gain stays the same for all remaining boundaries in the set.

Let us give a term to that value:

- *Thread stop point* – the minimal value of *b* in the thread that has the maximum possible gain of the thread.

Lemma 6. *For any general knapsack problem, the table of mapping of boundaries to optimal gains reflects a general increase in the values of gains, until gains for all groups stabilize; A general stop point is the point of last gain change; From that point on the gains are predictable: for any b the gain is the same as the gain of* $(b - w_1)$*.*

This is essentially Lemma 5 applied to all sets/groups. However, it should be obvious by now that the whole table of knapsack boundaries to solutions and gains is not only periodic from some point on, but that every period consists of one member of different groups. There are w_1 such groups corresponding to w1 possible values of $b_{\text{mod}}w_1$. Further analysis of the knapsack problem has to be based on those groups, which we shall formalize for ease of use.

6 Threads

So, having recognized the group nature of the knapsack solutions, and having observed the periodic nature of each of those groups (yielding the periodic nature of the whole table), we can try to formalize those groups to ease further analysis.

6.1 Definition

Let us introduce the concept of *thread*:

- *Thread* – an infinite set of all boundaries *b* for a given knapsack problem, such that $b_{\text{mod}}w_1$ is constant and the same for a thread.
- *Thread constant* – the value of $b_{\text{mod}}w_1$ for all values of *b* in a thread, which is the same for all *b* in the thread by construction of the thread.

So, every thread has its own constant, and that constant is different in all threads.

How many threads are there? Well, how many constants are there? Exactly w_1.

Do all constants belong to some thread? Yes.

Why don't we number the threads? Let's. Let us number the threads from 0 to $(w_1 - 1)$, and consider thread with index T to be the thread whose constant is T.

However, notice two important implications.

Implication 1. If the knapsack were not integer, but rather real, then this argument regarding the densities would be flawed. It means that real-numbered weights may yield a thread that never stabilizes – but rather keeps improving its gain. The upper gain bound would be unachievable, but the thread may try to get infinitely close to it. It may require an infinite supply of item types (to be addressed later).

Implication 2. If the number of types of items is not limited by some number, but the weights and values are integer, then any thread will *still* stabilize. The reason is simple – gain is an integer number that can improve discreetly by multiples of one. It also has a minimum value (0) and an upper bound that depends on MVI only. Therefore, the number of "steps" to an optimal gain within each thread is fixed. This means that optimal gain exists and is achievable, and the point of achieving that gain is the thread stop point.

6.2 Thread independence

Notice, that our first intuition is already explained: In case of MVI weight of 10 and boundary of 10,000,000,000 the thread that the boundary belongs to is thread 0, whose constant is zero, which means that no improvements in the gain are possible from the initial value of the gain. The initial value of the gain (first member of the thread is *b* of 0) is 0. Therefore, the solution to that boundary consists only of MVI, and the gain of that solution is also 0.

So far – so good.

We can also realize that the table will become fully periodic once every single thread has stabilized – i.e. the gains in each thread has reached the optimal value. Hence, the *general stop point* for any knapsack problem is the *maximum value of thread stop points for all threads of the problem.* This essentially shows the difference between a *particular* knapsack problem and *general* knapsack problem: the first one requires knowledge about one thread's stop point, while the second one requires all thread stop points.

Furthermore, since threads may stabilize independently of each other (i.e. stop points are in different places of the table), it is not necessarily necessary to build up an *entire* table of mappings of boundaries to solutions for boundaries from 0 to general stop point. Once some particular thread stop point has been reached, the rest of the values for that thread can be filled out without calculation. This means that the calculation of the general stop point's solution does not necessarily mean calculating *all* solutions to the general stop point. It may be less work.

So, our problems reduces to finding thread stop points for all threads.

6.3 Dynamic nature of threads

It is important that we observe a certain fact.

The problem of finding thread stop points is itself a dynamic programming problem. We may consider the solution to the problem stated for *N* items to be

based on the solution to the problem stated for (N – 1) items, with the last item attempting to improve optimal gains in each thread. If it does improve an optimal gain in some thread, then that thread's stop point may change in either direction. If it does not improve the gain, the thread's stop point may still change (decrease). However, the thread's optimal gain will never be reduced in quality by the new item.

Right away we can almost envision a table of threads versus items, and try to fit all information in it. However, it is not exactly clear how to do that yet.

6.4 Threads and MVI

Notice that for an easy case of one type of item available each thread becomes very clear. The gain for all the threads is zero, since MVI cannot improve gain on itself. This may as well be a starting point for building a table.

6.5 Thread 0

Also notice that thread 0, having a gain of 0, cannot be improved. That thread has an upper bound on the gain (which is 0), and that 0 is achieved right away for b of 0. Therefore, no addition of any item will ever change the make-up of the solutions to boundaries in thread 0 – every item other than MVI will be used exactly 0 times.

6.6 Thread $(0 + w_x) \bmod w_1$

Now consider thread $((0 + w_x) \bmod w_1)$. How many items of type x may each solution to that thread have?

May it have none? Yes, it may. Items may be non-used if their densities are bad enough.

May have one? Yes, it may. It will essentially mean that the solution will consist of the solution for some boundary in thread 0 (which would be MVI only) and one item of type x.

May it have two? No, it may not. Because if it had two items, then following Axiom 1 the subsolution of that problem with one of items x removed would contain item x. However, it is thread 0, which is made up of MVI only. So, this thread would never, ever, never-never-ever have two items of type x.

6.7 Thread $(T + w_x) \bmod w_1$

So, from this on we can make the same argument for thread $((0 + 2\ w_x)_{\ mod}\ w_1)$ – it will never have more than two items. And so forth, using mathematical induction – thread $((T + w_x)_{\ mod}\ w_1)$ may have one more item x than thread T, but no more than that.

6.8 Thread basics: linking

Essentially, this gives rise to "linking" threads with respect to a particular item:

1. We always know that thread $((T + w_x)_{\ mod}\ w_1)$ may have one more item x than thread T, but no more than that;
2. We always know that thread 0 does not have any non-MVI items.

6.9 Calculating gain in threads

Suppose we try to probe the gain for a solution for some boundary that contains some item x (solution 2, for some thread *t2*). It means that the solution would be based on the subsolution (solution 1, for some thread *t1*). If we know that subsolution's parameters, we can find out the new gain. Observe:

$$gain_2 = S_2 - (b_2 - (b_2)_{mod} w_1)\ v_1 / w_1$$

Define b_0 and S_0:

$$b_0 := b_1 - (b_1)_{mod} w_1$$
$$S_0 := b_0\ v_1 / w_1$$

Considering:

$(b_0)_{mod} w_1$	$= 0$	(by definition of b_0)
t_1	$= (b_1)_{mod} w_1$	(by definition of threads)
t_2	$= (b_2)_{mod} w_1$	(by definition of threads)
b_1	$= (b_0 + (b_1)_{mod} w_1) = b_0 + t_1$	(by definition of b_0)
b_2	$= b_1 + w_x = b_0 + t_1 + w_x$	(by construction of b_2)
S_1	$= S_0 + gain_1$	(by definitions of gain and S_0)
S_2	$= S_1 + v_x = S_0 + gain_1 + v_x$	(by construction of S_2)

Hence:

$$\begin{aligned} gain_2 &= S_2 - (b_2 - (b_2)_{mod} w_1)\ v_1 / w_1 \\ &= (S_0 + gain_1 + v_x) - ((b_0 + t_1 + w_x) - (b_0 + t_1 + w_x)_{mod} w_1)\ v_1 / w_1 \end{aligned}$$

$$= (S_0 + gain_1 + v_x) - ((b_0 + t_1 + w_x) - (t_1 + w_x)_{mod} w_1)\ v_1 / w_1$$
$$= (S_0 + gain_1 + v_x) - (b_0)\ v_1 / w_1 + ((t_1 + w_x) - (t_1 + w_x)_{mod} w_1)\ v_1 / w_1$$
$$= (S_0 + gain_1 + v_x) - S_0 - ((t_1 + w_x) - (t_1 + w_x)_{mod} w_1)\ v_1 / w_1$$
$$= (gain_1 + v_x) - ((t_1 + w_x) - (t_1 + w_x)_{mod} w_1)\ v_1 / w_1$$

So, the formula is:

$$gain_2 = (gain_1 + v_x) - ((t_1 + w_x) - (t_1 + w_x)_{mod} w_1)\ v_1 / w_1$$

What's the significance? The significance is that this formula *does not rely on b_0, but rather on thread information only.*
It means that we can accumulate knowledge about threads without actually considering the boundaries *at all.*

6.10 Algorithm, take one

What is the minimal information we want to know? The optimal gain values for each thread.

What else do we want to know? What is the minimum value of boundary *b* for which it happens.

What are we waiting for? Nothing.

Let's define the table *F* of items versus threads. Let us build some tables in such a way that cell F*(x,t)* tells us all the information we need to know about thread *t*, if items 1 through *x* are allowed to be used.

What information are we going to store in the tables?

First of all, the gain of that thread. Let's denote that value as *F(x,t)*;

Second of all, the minimum boundary value *b*, for which that gain happens. Let's denote that value as *B(x,t)*;

What do we know right away?

1. For any *t* the value *F(0,t)* is equal to 0;
2. For any *t* the value *B(0,t)* is equal to *t;*

What happens when we try add a new item, x?

First of all, adding it cannot spoil the gains of the solution for (x – 1), so we can fill that row out with values from row (x – 1).

Second of all, adding it will not change the gain for thread 0, so we may leave that alone.

Third of all, we can start from thread 0, make thread steps of w_x, and improve the table as follows:

1. let $t := 0$
2. let $old\_t := t$
3. let $t := (t + w_x) mod\ w_1$
4. *If* $(t = 0)$
   4.1. stop
5. let $g := (F(x,old\_t) + v_x) - ((old\_t + w_x) - (old\_t + w_x)_{mod} w_1)\ v_1 / w_1$
6. if($g < F(x,t)$)
   6.1. goto 2
7. if($g > F(x,t)$)
   7.1. let $F(x,t) := g$
   7.2. let $B(x,t) := B(x,old\_t) + w_x$
   7.3. goto 2
8. if($B(x,t) + w_x < B(x,t)$) // item improves min. boundary
   8.1. let $B(x,t) := B(x,old\_t) + w_x$
9. goto 2

This algorithm will fill out one row of the matrix – figuring out how item *x* affects all threads. It is *almost* what we want.

One – we have to add the processing of MVI – first row;
Two – we have to loop existing algorithm over all remaining items;
Three – we have to add a step that copies the previous row to the current one before we process the current one;
Four – we have to figure out what happens in case of a miss.

7 Chains

It seems that in all the logic above we have missed an interesting fact – that while analyzing the effect of a particular item on a knapsack we just might miss some threads. This chapter addresses this issue.

7.1 The reason for misses

Why are misses possible in the first place?

Consider the case of an item of weight 4, while there are 8 threads. If we analyze that item, we would start with thread 0, then move on to thread 4, then to thread 0 again, and so forth. Meanwhile, the solution to thread 5 may just consist of that item and the solution to thread 1!

So, why does this happen, in general? Turns out the explanation is quite simple.

From group theory (or whatever the appropriate math section it is), we know, that if two numbers X and Y are mutually prime (that is, their GCD is 1), then the set of values of { $X_{mod}Y$, $2X_{mod}Y$, …, $YX_{mod}Y$ } will consist of all the members of the set { 0, 1, …, Y-1 }. Both sets are of the same size, so the first set will consist of non-repeating numbers.

Essentially, our first draft of the algorithm does exactly that – goes through the first set of numbers. But in case when the greatest common divisor of the number of threads (which is the weight of MVI) and the weight of the item in question, then it will cover only a part of the threads – the ones whose value mod that GCD is zero.

7.2 Definition of chains

Essentially, if you think about it, if we start from thread 1 and continue with the process, we will get the same number of items as if we start with thread 0. Those sets of numbers are non-intersecting. We need a term for such a set of numbers, and that term will be: *chain*.

A Chain for item with weight w and t threads is a set of all numbers X between 0 and (t – 1), such that for all X in the set the value of $X_{mod}t$ is the same. That number will distinguish the chains – so, “chain 2” will mean that the number is 2.

How many chains will there be for every item? Exactly the GCD of the item’s weight and the number of threads.

7.3 Max usage number for items

It is clear that the algorithm will traverse each of the chains completely, provided that there is a good starting point (first thread to analyze).

But this fact of chaining brings up an interesting fact. We know that any item of weight w can be used no more than w_1–1, since if it is used w_1 times, they take the volume of w_1w, which, when occupied by w items of the MVI, will yield a better value, due to the maximum density of the MVI.

But the concept of chaining can restrict that number further. Using the Greatest Common Divisor of w_1 and w (let’s denote it as g), we can say

that the number of times the item is used is bound by $w_1w/g - 1$. Because if the

Lemma 7. *The number of times a non-MVI item of weight w is used is bound by $w_1/g - 1$, where g is the greatest common divisor of w and w_1.*

Consider the opposite case – when there are some w_1/g items of weight w there; they occupy the volume of $w_1 w/g$, which, when occupied by w/g items of the MVI, will yield a better value, due to the maximum density of the MVI.

7.4 Chain analysis

For every item, the number of chains is different.

While considering each item, we shall have to determine the number of chains, and analyze each chain separately.

To analyze a chain, it is enough to find a good starting point – a point before any potential $w_1/g - 1$ points that may use the item.

7.5 Chain 0

Chain 0 is the one that starts with thread 0. It is quite simple, really – all the analysis of it has been done before. We know that no non-MVI item will ever be used for thread 0, so we go through the rest of the threads in the chain in the right order (there just happens to be $w_1/g - 1$ of them).

7.6 Chain X

For any non-zero chain life gets trickier.

We don't have a nice starting point – a thread that will never ever use the item. It does exist for every scenario, but I just failed to provide a general way of finding it.

And we DO need it, because a solution utilizing the item may be based on a good solution to the previous considered thread that also uses that item; so if that previous thread is not updated yet, the one in question may just fail to be updated at all.

So, in order to analyze arbitrary threads, we are going to cheat. We shall make a double pass through the items in a chain. Then, if we start with some thread that should be updated based on the previous thread, which is not updated yet, then during the second pass that "previous" one will be ready.

Two passes are enough – because, essentially, at some point we shall hit the good "starting" thread, then the rest of the threads. Of course, about half of the work is going to be extra, but we just don't know ahead of time which parts it is going to be.

Plus, a double pass is still linear time with respect to w_1.

In order to be *completely* specific, we are going to consider $2w_1/g - 1$ threads, starting with any one! Why do we not need all $2w_1/g$ threads? Because we don't really need a "double" pass, we just need our sequence to have a subsequence starting with some unknown thread and containing all threads in the chain – so one thread may just be the last of the first pass and the first of the second pass.

In other words, combining the analysis of chain 0 and chain X, we conclude that we know how many threads we have to consider in each chain in order to completely reflect the effect of a particular item. For chain 0 it is $w_1/g - 1$ threads, for all other chains it is $2w_1/g - 1$.

8 Possibility analysis

Possibility analysis deals with hunting for an answer whether an optimal solution for a particular thread has anything to do with that particular thread.

8.1 Examples of an impossibility

Consider a knapsack problem involving one type of item only, whose weight is 10. Obviously, there are ten threads, and each thread's gain is zero. However, the solution for thread 5 (which, in this case, deals with boundaries of 5, 15, 25, etc.) will always occupy smaller volumes (in this case, 0, 10, 20, etc.), which means that the actual volumes in the thread are not "achievable".

8.2 Possibility tracking

It is quite easy to track it.

First of all, we know that zero is achievable.

Second of all, any solution based on a subsolution is as achievable as the subsolution.

That is enough.

9 Neighbors

Consider a certain solution for thread t, which is non-zero. Suppose that its gain is worse than the gain for its smaller neighbor, thread t–1. What does it mean? It means that using the solution from t–1 as a solution for t is better – just not achievable for t.

Should we track for it? Not really.

There are two cases to explore.

Case one is when the temporary neighbor solution is used, but some other item type's analysis changes it later on to an achievable solution. Well, if some solution is clearly better than another, it does not matter whether the one kicked out was achievable or not, and what was it based on. The new solution is achievable – which means there is a clear way to hit the volumes specified by the thread.

Case two is when the neighbor solution is actually kept. Well, in that case the neighbor solution is made up of some items, which were "kept" at the appropriate stages in the algorithm. However, that same algorithm is going to try those items for the thread in question as well – it just will be marked as unachievable, because the very first contributing thread will be other than zero.

This idea is a bit non-intuitive, but clears up a bit when you exercise the algorithm.

10 The algorithm

So, having done all the research, we come up with the following algorithm. For any given knapsack problem involving N types of items of weights w_1, w_2, …, w_N and values v_1, v_2, …, v_N, do the following:

1. Identify MVI – the item with the greatest ratio of *v/w* and the smallest *w*; Swap item 1 with that item, so that index 1 refers to MVI;
2. let *F(1,0)* := 0
3. let *B(1,0)* := 0
4. let *P(1,0)* := true
5. for *t* := 1 .. $w_1 - 1$
   5.1. let *F(1,t) := 0*
   5.2. let *B(1,t)* := t
   5.3. let *P(1,t)* := false
6. for *x* := 2 .. N
   6.1. for *t* := 0 .. $w_1 - 1$
      6.1.1. let *F(x,t)* := *F(x*–1,*t*)
      6.1.2. let *B(x,t)* := *F(x*–1,*t*)
      6.1.3. let *P(x,t)* := *F(x*–1,*t*)
   6.2. let *number_of_chains* := gcd(w_1, w_x)
   6.3. for *chain* := 0 .. *number_of_chains*–1
      6.3.1. if *chain* = 0
         6.3.1.1. let *number_of_tries* := w_1 / *number_of_chains* – 1
      6.3.2. else
         6.3.2.1. let *number_of_tries* := 2w_1 / *number_of_chains* – 1
      6.3.3. let *last_thread* := *chain*
      6.3.4. let *thread* := (*last_thread* + w_x) mod w_1
      6.3.5. for *try* := 1 .. *number_of_tries*
         6.3.5.1. let *g* := (*F(x,last_thread)* + v_x) –
                  ((*last_thread* + w_x) – (*last_thread* + w_x)$_{mod}w_1$) v_1 / w_1
         6.3.5.2. if *g* < *F(x,thread)*
            6.3.5.2.1. *go to step 6.3.5.5*
         6.3.5.3. if *g* > *F(x,thread)*
            6.3.5.3.1. let *F(x,thread)* := g
            6.3.5.3.2. let *B(x,thread)* := *B(x,last_thread)* + w_x
            6.3.5.3.3. let *P(x,thread)* := *P(x,last_thread)*
            6.3.5.3.4. *go to step 6.3.5.5*
         6.3.5.4. if B*(x,thread)* + *wx* < B*(x,thread)*
            6.3.5.4.1. let *B(x,thread)* := *B(x,last_thread)* + w_x
            6.3.5.4.2. let *P(x,thread)* := *P(x,last_thread)*
         6.3.5.5. let *last_thread* := thread
         6.3.5.6. let *thread* := (*last_thread* + w_x) mod w_1

At the end of this algorithm, the values of *B(N,0), B(N,1), …, B(N,w_1)* will contain the stop conditions for threads 0, 1, …, w_1–1 respectively. The largest of them will be the GENERAL STOP CONDITION. The values of *F(N,0), F(N,0), …, F(N,w_1)* will contain the gains of the optimal solutions for those threads. The values of *P(N,0), P(N,1), …, P(N,w_1)* will contain the possibility of those solutions – i.e. whether the optimal solutions for those threads actually hits the prescribed boundaries (in case of true) or is less than them (in case of false).

11 Examples

It is quite tiresome to trace the whole algorithm for example cases, so I shall demonstrate the algorithm's results for some particular examples.

11.1 Case 1

Let there be three types of items of weights 7, 5, 1 and values 10, 7, 1 respectively. The algorithm produces the following values:

B(3,0)=0	*F(3,0)=0*	*P(3,0)=true*
B(3,1)=1	*F(3,1)=1*	*P(3,1)=true*
B(3,2)=2	*F(3,2)=2*	*P(3,2)=true*
B(3,3)=10	*F(3,3)=4*	*P(3,3)=true*
B(3,4)=11	*F(3,4)=5*	*P(3,4)=true*
B(3,5)=5	*F(3,5)=7*	*P(3,5)=true*
B(3,6)=6	*F(3,6)=8*	*P(3,6)=true*

Hence, general stop point is 11.

Among other things, we can read from these results that for every large enough B such that $B_{mod}7$ is 3 the value of gain will be 4 (i.e. the solution value will be $(10(B - B_{mod}7) / 7) + 4$, and the minimum boundary for which it will happen is 10.

11.2 Case 2

Let there be two types of items of weights 5, 499999 and values 7, 699998 respectively. The algorithm produces the following results:

B(2,0)=0	*F(2,0)=0*	*P(2,0)=true*
B(2,1)=1	*F(2,1)=0*	*P(2,1)=false*
B(2,2)=1499997	*F(2,2)=1*	*P(2,2)=true*
B(2,3)=999998	*F(2,3)=3*	*P(2,3)=true*
B(2,4)=499999	*F(2,4)=5*	*P(2,4)=true*

Hence, the general stop point is 1499997. Mind you, this number was found using the table of size 2 by 5, with constant time of filling out every cell – which can easily be done by hand.

This particular example shows the usefulness of the algorithm – in just about 10 steps it saved the building of a table of about a million cells, and told us that for every large enough B such that $B_{mod}5$ is 1 the value of gain will be 0 (i.e. the solution value will be $(10(B - B_{mod}5) / 5) + 0$, and the minimum boundary for which it will happen is 1. It also told us that those solution's weights will be smaller than the actual boundaries – since the flag P(2,1) is set to false.

11.3 Case 3

Let there be four types of items of weights 8, 6, 4, 12 and values 20, 12, 9, 30 respectively. The algorithm produces the following results:

B(4,0)=0	*F(4,0)=0*	*P(4,0)=true*
B(4,1)=1	*F(4,1)=0*	*P(4,1)=false*
B(4,2)=18	*F(4,2)=2*	*P(4,2)=true*
B(4,3)=19	*F(4,3)=2*	*P(4,3)=false*
B(4,4)=12	*F(4,4)=10*	*P(4,4)=true*
B(4,5)=13	*F(4,5)=10*	*P(4,5)=false*
B(4,6)=6	*F(4,6)=12*	*P(4,6)=true*
B(4,7)=7	*F(4,7)=12*	*P(4,7)=false*

Hence, the general stop point is 19.

This particular data is very interesting for a number of reasons. First, the even weights provide clear cases when boundaries are not achievable (it is not possible to hit odd boundaries), and the algorithm picks it up correctly. Secondly, any odd boundary will essentially be a solution to the preceding even one, and the algorithm picks that up as well. Third, the test data contains two items of the maximum density (20 to 8 is the same as 30 to 12), and the algorithm still works. Fourth, it figures out the solution for thread 2 (boundaries of 2, 10, 18, etc.) as a combination of the second and third item:

For boundary of 10 we use one weight of 6 and one weight of 4, producing value of 12+9 = 21, which has a gain of 1 to the MVI solution for weight 8 (8/8 = 1, 1x20 = 20, 21-20 = 1). For boundary of 18 we use one weight of 6 and one weight of 12, producing value of 42, which has a gain of 2 to the MVI solution for weight 16 (16/8 = 2, 2x20 = 40, 42 – 40 = 2). For the rest of this thread (26, 34, 42, etc) the gains don't get any better.

12 Conclusions

12.1 Algorithm characteristics

The presented algorithm runs in $O(N\ w_1)$ time, which can be quite smaller than the actual general stop points (as illustrated in example 2). Besides being quite fast, it also gives a lot of information about the knapsack table, which may render even the general stop point unnecessary – if the hunt is for a particular solution, which belongs to a thread that has a significantly lower stop condition.

12.2 Further improvements

While I doubt that it is possible to beat the upper-bound on the algorithm time, it is certainly possible to improve certain characteristics of the algorithm by recognizing an interesting fact.

The algorithm is essentially based on two assumptions:

1. The period of the table is w_1, if w_1 is the weight of MVI;
2. Thread zero consists of values 0, P, 2P, 3P, where P is the period of the table (which, in turn, is w_1);

And while it is certainly true that the table will be periodic with a period of w_1 from some point on (provided the definition of MVI), the reality may be even more interesting: the period just may be a factor of w_1.

Consider a setup containing two items with the largest density, whose respective weights are 10 and 15. According to the algorithm, 10 will be a period of the table, which is true. But 5 will also be a period of the table that will set in at some point – since starting from 10 any multiple of 5 can be picked using 10 and 15.

So, essentially, the smallest predictable period of the table is actually the greatest common divisor of the weights of all items whose density is maximum. This can cut down the tables that the algorithm builds by a factor of that GCD.

However, the algorithm itself will have to be significantly modified to reflect the change in the two assumptions that it makes.

But it is still very much possible! If I had more time in this semester, I'd even do it! J