Sage Algorithms for Knapsack Problem

Leo Landa
leo@leolan.com
Department of Computer Science and Engineering,
University of California,
San Diego

1 Abstract

We attack the unbounded integer knapsack problem, known to be NP-complete. The state-of-the-art algorithms for precise solutions are pseudo-polynomial with respect to n (the number of types of items available) and b (the capacity of the knapsack). With reasonable encoding, those algorithms are exponential with respect to the bits of encoding due to the presence of b. In practical settings, the boundary constraint b can be quite large with respect to the rest of the values in the problem statement, and becomes a dominating factor in both time and space complexity.

We present different algorithms for solving knapsack problems that are not dependent on the value of boundary in the original problem. These algorithms allow building useful pre-solution information about a particular instance of a knapsack problem (boundary excluded), which later be used in combination with *any* boundary to generate a final solution in polynomial time. Specifically, the Sage-2D algorithm shown provides pre-solution information for sufficiently large boundaries in $O(nw_1)$ time and space complexity (where w_1 is the weight of the best item), and the Sage-3D algorithm shown provides pre-solution information for any boundaries in $O(nw_1v_1)$ time and space complexity (where w_1 is the weight of the best item, and v_1 is the weight of the best item).

This type of approach has two advantages over the state-of-the-art algorithms. Firstly, even though the pre-solution algorithms are pseudo-polynomial with respect to the items' weights and values, they do not depend on the value of the boundary constraint b, which makes solving a large class of practical problems significantly easier (requiring less time and space). Secondly, this approach allows solving any number of knapsack problems with the same setup and varying boundary constraint in polynomial time, thus reducing the overhead of solving many knapsack-based practical problems.

We conjecture that there are many more NP-complete problems that can also be solved through a pseudo-polynomial pre-solution and polynomial final solution.

2 Introduction

In this section we define the problem formally, show known state-of-the-art properties of unbounded integer knapsack problems, and define a solution model for the rest of the paper.

2.1 Problem definition

The unbounded integer knapsack problem is define as follows. Given a set of (n) types of items with corresponding integer weights ($w_1, w_2, \ldots, w_n$) and values, maximize the combined value of a selected set of items (V), not exceeding the given knapsack carrying capacity (b):

$$\begin{array}{lll} \text{Maximize:} & V = & x_1v_1 + x_2v_2 + \ldots + x_nv_n \\ \text{Subject:} & & x_1w_1 + x_2w_2 + \ldots + x_nw_n \le b \\ & & x_j,\ w_j,\ v_j,\ b - \text{integers} \end{array}$$

The unbounded problem presumes that the solution values of x may be any non-negative integer number, i.e. there is an unlimited supply of each type of item. Note also that if the values of x, w, and v are integers, the value of b can also be considered effectively integer: since a linear combination of any integer values is integer, the resulting combination will be less than or equal to the whole part of boundary b.

2.2 Reasonable encoding

The specification of a problem includes specifying the weights, values, and boundary at a minimum. A reasonable encoding would specify all those values without explicitly specifying the value of n – the number of types. It is possible for the algorithm to calculate this number on its own from the encoding of the problem by counting the number of weights and values specified.

However, the number of items in a reasonable encoding is linear with respect to the encoding, since there cannot be more items specified than bits in the encoding of the problem.

Therefore, any complexity polynomial with respect to weights, values, or a boundary is pseudo-polynomial to the number of bits in a reasonable encoding, i.e. exponential, while any complexity polynomial with respect to the number of items is truly polynomial to the number of bits in a reasonable encoding.

2.3 Special cases: n=0, n=1

If n=0, the solution to any problem instance is trivial – the total value obtained is zero. The solution is obtained by a

If n=1, the solution to any problem instance is also trivial – it is comprised of the only available item; the x_1 is obtained by integer division of boundary b over the weight of the only item w_1. This is a polynomial operation with respect to the bits of encoding, and thus has no interest.

In the rest of the paper, we will concentrate on the problem instances where n is greater than 1.

2.4 Density

The art of solving integer knapsack problems recognizes the importance of the *density*, a descriptive property of each type of items. Density is the ratio of the items' value to weight, and represents the rate (efficiency) with which the item is increasing total value by using up the knapsack weight capacity.

Among the finite number of types of items, the highest density is considered *best*. The types of items with best density are considered *best types*, while types of items with non-best density are considered *non-best types*. The items corresponding to *best types* and *non-best types* are called *best items* and *non-best items* respectively.

2.5 Non-best items and their usage

It is known that optimal solutions to knapsack problems may contain no more that a certain finite number of non-best items. If a certain best type j has weight w_j, while a certain non-best type k has weight w_k, then any optimal solution contains no more than $w_j - 1$ items of type k. The reason is obvious: if the optimal solution contains at least w_j items of type k, then they occupy $w_j w_k$ of the total weight-carrying capacity. Exactly the same capacity can be occupied by w_k items of type j (i.e. by best items), and the total value produced by best items would be greater than the total value produced by non-best items (by definition of *best density*). Hence, the value of $w_j - 1$ is a cap for the number of

times an item of any non-best type may appear in a particular optimal solution[1]. For example, if a certain best type has value of 15 and weight of 5 (hence, density of 3), while some non-best type has value of 8 and weight of 4 (hence, density of 2), then optimal solutions cannot have 5 or more of non-best items: 5 non-best items take up 5*4=20 units of weight and produce 20*2=40 units of value, while the same units of weight can be taken by 4 items of best type, producing 20*3=60 units of value.

Therefore, optimal solutions to knapsack problems in question have a certain finite cap on the number of times each non-best item may appear in the solution, and do not have a cap on how many times a best item may appear in the solution.

2.6 Best items and their usage

Even though there is no cap on the number that a non-best item may appear in optimal solutions, in the face of several best types such a cap can be imposed on all but one of such types.

Suppose there are two best types in a problem – j and k, with corresponding weights w_j and w_k and values v_j and v_k. Since both types are best, they both have the same density $\rho = v_j/w_j = v_k/w_k$.

In this case w_j items of type k use up w_jw_k of the weight-carrying capacity of the knapsack, producing a combined value of $v_kw_j = v_kw_jw_k/w_k = \rho w_jw_k$, or the density multiplied by the weight capacity occupied. The same combined value can be produced by w_k items of type j: they use up w_jw_k of the weight-carrying capacity of the knapsack, producing a combined value of $v_jw_k = v_jw_kw_j/w_j = \rho w_jw_k$, the same value.

Therefore, for two given best types of items, it is possible to cap the number of times the items of one time appear in a solution by preferring another best type. Furthermore, in the presence of several best types, it is possible to cap all best types but one, preferring that one to all other best types.

2.7 Solution model

All solutions discussed in this paper impose a cap on all the best types of items except for one. The one "uncapped" type is called *the best type*, and is chosen to have both the maximum density and the smallest weight. It is possible to select such a type in a preprocessing stage of an algorithm with n steps, which makes this step polynomial (section 2.2) and thus affordable.

For simplicity, we will assume that the preprocessing stage rearranges the indexes of items in such a way that *the best type* has index of 1, and the rest of the types have indexes 2 and higher. We will call this type *type 1*, or *best type*, and never use those terms to refer to other types of items with the density equal to the density of type 1.

2.8 Principle of optimality

Knapsack problem is a classic dynamic programming problem due to the inherent principle of optimality of solutions, which is the guiding principle for the DP-based algorithms. We will refer to this principle several times throughout the paper.

The principle of optimality states that any optimal solution is based on optimal sub-solutions. The general form of the principle means that if an optimal solution to a particular instance of a knapsack problem with boundary b can be split up into several sub-solutions (several sets of whole items, taking up portions of the boundary – b_1, b_2, …, b_k), then those sub-solutions are optimal solutions to their respective portions of boundaries.

[1] Note that this cap may be even smaller in case the greatest common divisor of the weights of the best item and non-best item in question is greater than one; however, this is not relevant at this point.

The validity of the principle of optimality is intuitive. The optimality of any solution by definition means that there does not exist any other solution yielding a better result. If a particular sub-solution to the problem is not optimal itself, i.e. there exists a better solution to sub-problem, then this better sub-solution can be combined with the rest of sub-solutions to the original problem, yielding a better result, which is contradiction.

Therefore, *all* sub-solutions of *any* optimal solution to *any* instance of the knapsack problem is an optimal solution itself, to the corresponding weight-carrying capacity occupied by it.

2.9 "Periodic" nature

With a cap present on all but one types of items, it becomes obvious that there exists a particular boundary b^{**} such that its optimal solution does not include any items of type 1, but also such that optimal solutions to any boundary larger than that *does* include them. Considering that w_1-1 is a sufficient cap for all $n-1$ capped types (sections 2.5 and 2.6), the weight-carrying capacity occupied by non-best items cannot exceed $(n-1)*(w_1-1)$, no solution for boundaries larger than this have more items of non-best types, which means that starting with boundary of $(n-1)*(w_1-1)+w_1$ and larger *all* solutions have at least one item of the best type. Even more, it means that solutions to *all* boundaries greater than b^{**} are combined from sub-solutions to b^{**} or less and items of the best type. This leads to a "periodic" nature of all solutions to boundaries b larger than b^{**}: each such solution is a combination of a best item and a previous solution to $b - w_1$.

The given boundary of $(n-1)*(w_1-1)+w_1$ as a possible b^{**} is an over-estimate for illustrative purposes. It is a sufficient, but not necessary condition, which can be quite larger than the actual values of b^{**}.

The significance of b^{**} is tremendous. If b^{**} is known, then a solution to *any* boundary larger than b^{**} can be found by combining an appropriate solution to b^{**} or less and the best items only. For example, if b^{**} is 200, w_1 is 10, and a particular problem has a boundary of 2000005, then the solution is obtained as a combination of solution to 195 and (2000005–195)/10= (1999810)/10 = 199981 best items (as opposed to running an algorithm of *O(2000005)* time and space complexity).

However, attempts to find b^{**} have not been very effective. Gilmore and Gomory [1] have found a sufficient but unnecessary condition for b^{**}:

$$b^{**} = v_1 / (\rho_1 - \rho_2)$$

In this formula ρ_1 refers to the density of the best type, and ρ_2 refers to the best density of the remaining types. This formula gives a great overestimation in the majority of cases, and gives infinity in case of two types with the best density.

2.10 Conclusions

The state of the art recognizes the periodic nature of the solutions to knapsack problems with fixed values and weights and varying boundary past some boundary b^{**}. However, there are no known conditions for b^{**} that are both necessary and sufficient. There are also no known algorithms to find such a boundary efficiently.

In this paper, we shall explore the properties of b^{**} in great detail, find necessary and sufficient conditions for it, and then use the properties found to make an algorithm that provides pre-solution information for boundaries both larger and smaller than b^{**} that allows constructing a solution to any boundary in linear time.

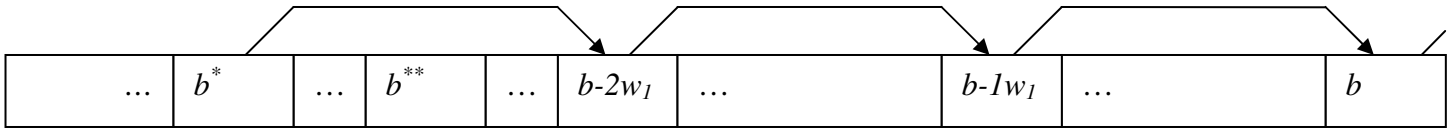
3 Analysis

In this section, we analyze some properties of optimal solutions to knapsack problems, extend the notion of b^{**}, define the concepts of threads and gains, and lay the foundation for Sage-2D and Sage-3D algorithms.

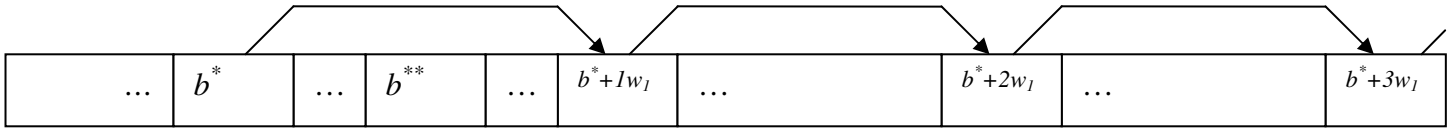
3.1 Periodic nature revisited: threads

In section 2.9 we have shown that solutions to all boundaries b above a particular b^{**} consist from at least one best item. In section 2.8 we have shown that the remaining sub-solution, taking up $b–w_1$ units of the weight-carrying capacity, is an optimal solution itself. In case this boundary is itself greater than b^{**}, the same argument can be applied, which means that the solution to $b–w_1$ consists of one best item and the solution to $b–2w_1$. The same argument can be applied to all boundaries of the form $b–kw_1$, as long as they are all greater than b^{**}.

Visually this relationship can be viewed on a number line as follows:

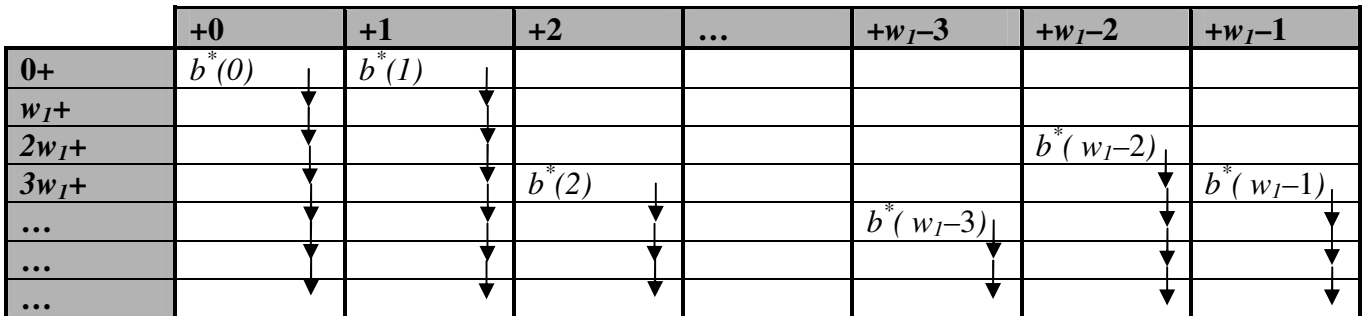


The boundaries are linked together because they have the same "nature": they are all a combination of a number of best items and a particular sub-solution to boundary b^* (which is equal to b^{**} or smaller). This sub-solution is common to the solutions of all boundaries linked. The solution to the boundary that is a starting point for the links yields a polynomial-time solution to any boundary in the chain, as long as the target boundary can be reached by adding the weight of the best item enough times. Hence, the solutions to problem instances for linked boundaries really revolve around the solution to boundary b^*:



Considering that all links have the same length – w_1 – we can see that there are w_1 paths. Within each path all boundaries have the same value modulo w_1. Each path has its own starting boundary b^*, which does not have any best items in its solution. Considering that there are w_1 such paths, there are w_1 values of b^*, one for each path.

Visually we can represent this structure among the solutions for large enough boundaries in a table form, rather than a linear form. The table simply wraps around the number line with w_1 entries in a row, so that boundaries that are a multiple of w_1 away from each other are lined up:



The arrows represent links between boundaries whose solutions differ by one best item only. In every column the boundaries are the same modulo w_1. The columns are labeled 0 through (w_1–1) corresponding to that modulo value common to all boundaries in the column. The solutions in every column become related by the best items starting at some point that we have earlier denoted as b^*. Every column has its own such b^*, and the largest one of them is nothing else but b^{**}. The values of b^* in some columns may be significantly smaller than b^{**}: for example, column zero (representing all boundaries divisible by the weight of the best item) has b^* of zero, meaning that solutions to *all* boundaries in the column are made up from the best item only.

The column view gives a very important insight into the structure of the solutions to knapsack problems with varying boundary.

We consider that all boundaries b such that $b \bmod w_1 = t$ to be making up a *thread* numbered t. There are w_1 such threads, numbered from 0 to w_1–1. Every thread is unique in a sense that it is based on a unique solution to its own b^*, which is the largest boundary in a thread whose optimal solution does not include the best item.

3.2 Comparing solutions: gain

In order to formalize the relationships between different candidate solutions for the same boundary, as well as solutions and their respective sub-solutions, we introduce the notion of *gain*.

Gain is a property of a solution candidate of a knapsack problem for a particular boundary. It is the difference between the total value of the solution and the maximum value obtained using the best type only. We will denote the value of gain with a letter g. Considering that best-type-only solutions involve best items only, as many as can fit in the carrying capacity b, the general formula for gain is as follows:

$$g(b) = V(b) - \lfloor b/w_1 \rfloor v_1$$

Gains have several important properties that explain why it was chosen as the basis for Sage algorithms.

Optimal gains cannot be less than zero. If any solution yields a result that is worse than the solution involving the best item only, it is by definition not optimal. All optimal solutions have gains of zero or more.

Gains cannot be more than v_1–1. A gain of $v1$ means that the solution has average density higher than the best density, which is impossible by definition of the best density.

Gains are always integer. Since gains are defined as a difference between two integer numbers, they are also always integer.

The gain of a solution that contains an item of non-best type *j* can be expressed through the gain of the sub-solution without it and the thread of the sub-solution. Suppose the sub-solution for boundary b_s in thread s has a gain of g_s, and the solution for boundary $b_b = b_s+w_1$ consists of the same items with an addition of item j. Then the gain of the latter solution, g_b, can be found as follows:

$$
\begin{aligned}
s &= b_s \bmod w_1 \\
g_b &= V_b - \lfloor b_b/w_1 \rfloor v_1 \\
&= (V_s + v_j) - \lfloor (b_s+w_j)/w_1 \rfloor v_1 \\
&= (V_s + v_j) - \lfloor (\lfloor b_s/w_1 \rfloor w_1 + b_s \bmod w_1 + w_j)/w_1 \rfloor v_1 \\
&= (V_s + v_j) - \lfloor (\lfloor b_s/w_1 \rfloor w_1 + s + w_j)/w_1 \rfloor v_1 \\
&= (V_s + v_j) - (\lfloor b_s/w_1 \rfloor + \lfloor (s + w_j)/w_1 \rfloor) v_1 \\
&= (V_s + v_j) - \lfloor b_s/w_1 \rfloor v_1 - \lfloor (s + w_j)/w_1 \rfloor v_1 \\
&= (V_s - \lfloor b_s/w_1 \rfloor v_1) + v_j - \lfloor (s + w_j)/w_1 \rfloor v_1 \\
&= g_s + v_j - \lfloor (s + w_j)/w_1 \rfloor v_1
\end{aligned}
$$

Note that the final formula for g_s does not depend on the actual boundaries involved, but rather *thread* numbers. This formula essentially allows to find gain of a solution that consists of a combination of a sub-solution (gain and thread information) and a particular non-best item.

There are exactly v_1 distinct values that optimal gains may have. This is follows from the previous three points.

Solutions obtained by adding or removing the best items have the same gain as the original solution. To see why this is true, suppose that the sub-solution (smaller) for boundary b_s has a gain of g_s, and the boundary for the larger problem is $b_b=b_s+w_1$, while the value of the solution to the larger problem is $v_b=v_s+v_1$. By definition of gain we have:

$$
\begin{aligned}
g_s &= V_s - \lfloor b_s/w_1 \rfloor v_1 \\
g_b &= V_b - \lfloor b_b/w_1 \rfloor v_1 \\
&= (V_s+v_1) - \lfloor (b_s+w_1)/w_1 \rfloor v_1 \\
&= (V_s+v_1) - (\lfloor b_s/w_1 \rfloor+1) v_1 \\
&= (V_s+v_1) - (\lfloor b_s/w_1 \rfloor v_1 + v_1) \\
&= V_s+v_1 - \lfloor b_s/w_1 \rfloor v_1 - v_1 \\
&= V_s - \lfloor b_s/w_1 \rfloor v_1 \\
&= g_s
\end{aligned}
$$

An important implication of this is the fact that for large enough boundaries the gains of the solutions in a single thread do not change, since they are all obtained from each other by adding or removing the best item.

Better solutions for the same boundary have higher gain. Suppose there are two candidate solutions for some boundary b, yielding values of v_b (gain g_b) and v_s (gain g_s), such that $v_s < v_b$. Then, by definition of gain, we have:

$$
\begin{aligned}
&v_s < v_b \\
&g_s + \lfloor b/w_1 \rfloor v_1 < g_b + \lfloor b/w_1 \rfloor v_1 \\
&g_s < g_b
\end{aligned}
$$

The same mechanism can be used to prove that any relationship between two solutions of a problem for the same boundary holds for corresponding gains.

In each thread, the values of optimal gain are non-decreasing as boundary increases, and reach a certain maximum at b^*. Since the solution for any boundary in the thread (with an exception of the smallest one) can be achieved by combining the solution to the previous boundary in thread and the best item, and such combination leads to the same gain as the previous solution, any solution that has a smaller gain loses to this combination. In other words, once a certain gain is achieved in a thread, all the boundaries greater than the one that whose solution achieves it can also have the same gain. This effectively means that the quality of solutions to the boundaries in a thread cannot get worse – it will at least be as good as the previous ones. Considering that there is a finite number of values that gain can have, even if it improves for certain boundaries, it cannot keep improving forever – at some boundary it reaches the highest possible value for the thread. The boundary where that happens is, in fact, the b^* for the thread – the largest boundary whose optimal solution does not have the best item (if it had the best item, its gain would be the same as the gain of the previous boundary in the thread, as shown above).

4 Algorithm Sage-2D: pre-solution for sufficiently large boundaries

The Sage-2D algorithm builds pre-solution information that consists of the precise values of b^* for every thread of the problem, as well as the solutions for those boundaries. This information is enough to solve the given knapsack problem for any boundary that is no less than the value of b^* it its thread.

4.1 Approach

The problem of finding b^{**} has evolved into finding a set of b^* – one value for each thread – and their corresponding solutions. The proposed algorithm starts off by finding the required information for a knapsack setup consisting of the best type only, and then updates the information by processing the rest of the available types one by one.

The procedure involves comparing existing optimal solutions for each thread to new candidates, selecting the best one fitting the selection criteria. The selection criterion is as follows:

- The solution with higher gain is preferred over the solution with lower gain;
- If gains are the same, the solution with smaller boundary is preferred[2];

4.2 Principle of optimality: relationship between different b^*

Suppose there exists information about the values of b^* in all threads, including their optimal solutions and gains.

Thread zero always consists of boundaries that are divisible by w_1, their preferred solutions under the solution model discussed in section 2.7 consist of items of the best type only, which means that the gains of such solutions are zeroes. All of the boundaries are linked, and the b^* of the thread is zero.

[2] In practice, sometimes both gain and boundary match, even though the solutions are made up from different sets of items. There could be more conditions added to the selection criteria, for example *reachability*. Sometimes one solution may consist of items whose combined weight matches the specified boundary precisely (i.e. *reaches* it), while another solution's combined weight is less than the boundary (i.e. *does not reach* it). An algorithm may take this or any other criteria into consideration, but these are all secondary to the primary criteria of knapsack solutions, which is to maximize combined value for given boundaries. Therefore, these criteria are omitted from discussion.

Suppose we choose some other thread with a non-zero gain at its corresponding b^*. Suppose that solution for its b^* contains at least one item of some type j (since this is a b^*, this is type is not the first type). The solution is based on two components – the type j and the rest of the items that use up $b^*–w_j$ units of weight carrying capacity. From the principle of optimality described in section 2.8, this sub-solution must also be optimal for its corresponding thread. Following the logic of the principle of optimality, if that sub-solution was not optimal, then we could combine the optimal solution for that thread with the same item j to achieve a better solution for the b^* in question.

Effectively, the principle of optimality claims that all sub-solutions of the optimal solutions to b^* in every thread are also b^* for the corresponding threads.

4.3 Adding a new type to existing solution: chains

Suppose there exists information about the values of b^* in all threads, including their optimal solutions and gains, corresponding to a certain set of items. Suppose we were to extend the knapsack problem with a new type j of weight w_j *and value* v_j.

For any thread t, the item j is either part of the optimal solution in the thread, or it is not.

If it is, in fact, part of the solution, then it is used either once or more than once. If it is used once, then this solution is a combination of a sub-solution (optimal for its thread) and does not use the item. If it is used more than once, then it this solution is a combination of a sub-solution (optimal for its thread) and also uses this item at least once.

Therefore, in order to find out whether an item is part of a new optimal solution for some thread t (a solution which is better than the solution in the absence of this item), it is sufficient to know the optimal solution for the thread $(t – w_j)$ *mod* w_1. In turn, to find whether the optimal solution for that thread has item j it is sufficient to know the optimal solution for the thread $(t – 2w_j)$ *mod* w_1. Therefore, every thread has a "parent" thread with respect to item j, such that it is necessary to know the optimal solution for the "parent" thread to find the optimal solution for the "child" thread. Checking solutions must go from parents to children, provided that the very first parent's information is reliable, until all threads are checked.

This procedure involves going "forward" from some start thread t: first checking thread t, then thread $(t + w_j)$ *mod* w_1, then $(t + 2w_j)$ *mod* w_1, then $(t + 3w_j)$ *mod* w_1, and so on. Such a procedure encounters exactly $w_1/gcd(w_1,w_j)$ jumps through distinct threads before a thread is repeated (a full circle is made), where $gcd(w_1,w_j)$ stands for the greatest common divisor of w_1 and w_j. The definition of greatest common divisor both necessary and sufficient for the claim:

- **sufficient**, since $w_1/gcd(w_1,w_j)$ jumps by w_j each cover a total distance of $w_1w_j/gcd(w_1,w_j)$ units, which when divided by the number of threads (w_1) yields $w_j/gcd(w_1,w_j)$, which is an integer number – and hence the end thread is the same as the start thread;
- **necessary**, since if it took less than $w_1/gcd(w_1,w_j)$ jumps to come back to the same thread (say, k jumps, $k<w_1/gcd(w_1,w_j)$), it would mean that the value of kw_j evenly divides both w_1 and w_j, making the value of $(w_1w_j)/(kw_j) = w_1/k$ a divisor of both w_1 and w_j; however, this value is greater than $w_1/gcd(w_1,w_j)$, which is a contradiction by definition of greatest common divisor.

Hence, the technique of jumping forward involves making a full loop of $w_1/gcd(w_1,w_j)$ threads. If the greatest common divisor is not equal to 1, this loop would not visit all threads, but only a fraction of them. The algorithm would have to visit several sequences of threads (precisely, $gcd(w_1,w_j)$ of them, independently of each other). We call such sequences *chains*, and index the chains with numbers from 0 to $gcd(w_1,w_j)–1$ based on the smallest thread number in the chain.

4.4 Starting points in chains

In order to visit all threads in a chain for the purpose of evaluating the usefulness of an additional item, a proper starting thread has to be chosen – a thread whose optimal solution would not include item j, so that its child thread could have no more than one item j in its optimal solution, the next one – no more than two, etc.

Every chain corresponding to any non-best type (even a type with the best density) has such a thread that its corresponding optimal solution does not contain the item j. Suppose it wasn't so, and every single thread in the chain had an optimal solution did in fact have at least one item j. From the principle of optimality defined for b^* as discussed in section 4.2 we conclude that if a thread's optimal solution involves a non-zero number of items of type j (non-best), then the sub-solution missing one such item must also be optimal for its thread. Therefore, if an item is used by any thread at all, there is at least one thread that uses it once less, and so on recursively, until a thread with 1 item is encountered, whose parent thread's optimal solution does not require this item at all. This proves the existence of such a thread.

Since there always exists at least one thread whose optimal solution does not involve the item of the new type, any one of such threads can be used as the first thread to start scanning the threads forward. Finding such a thread directly is not an easy task.

However, such a thread can be recognized during traversal of the chain – it would be the thread whose optimal solution does not use the solution to the parent thread as a sub-solution. Due to the existence of such a thread, it is guaranteed to be encountered on the first pass. Therefore, a double-pass through the chain (or looping through the threads twice) is sufficient to find all optimal solutions that may be based on the new item – the first pass encounters the good starting point, and the second pass ensures that all thread's solutions are based on optimal solutions parents.

More specifically, it is needed to have $w_1/gcd(w_1,w_j)–1$ jumps forward from the starting point to visit all the rest of the threads, and it is needed to have $w_1/gcd(w_1,w_j)$ jumps forward to consider all possible threads as a starting point. Therefore, in general case, it is needed to have $2w_1/gcd(w_1,w_j)–1$ jumps forward from an arbitrary thread in the chain.

The only special case is chain 0, which includes thread 0. It is known ahead of time that thread 0 does not included any items that are not of the best type, and therefore this thread can be used as a starting point. In this special case, only $w_1/gcd(w_1,w_j)–1$ jumps forward are required to visit each thread.

4.5 Sage-2D: the algorithm

The algorithm builds a two-dimensional (hence the name) table T, whose first dimension corresponds to the types of items analyzed ($1…n$), and the second dimension corresponds to the threads ($0…w_1–1$). Each cell T[x,y] contains information about the optimal solution for thread y, if item types 1 through x are used. The cell contains the gain of the optimal solution ($T[x,y].g$) and the smallest boundary where that gain can be reached ($T[x,y].b$).

The algorithm assumes that there is at least one type of items (otherwise the solution is trivial, as discussed in section 2.3), and that the item of highest density and smallest weight has index of 1.

Algorithm Sage-2D
FOR *thread* = 0 TO w_1–1
    T[1,*thread*].g = 0
    T[1,*thread*].b = t
FOR *item* = *2* TO n
    FOR *thread* = 0 TO w_1–1
        T[*item*,*thread*] = T[*item*–1,*thread*]
    number_of_chains = $gcd(w_1,w_{item})$
    FOR *chain* = 0 TO *number_of_chains*
        IF *chain* = *0*
            number_of_jumps = w_1/*number_of_chains*–1
        ELSE
            number_of_jumps = 2w_1/*number_of_chains*–1
        thread = *chain*
        jumps_so_far = 0
        WHILE *jumps_so_far* < *number_of_jumps*
            next_thread = (*thread* + w_j) mod w_1
            cand_gain = T[*item*,*thread*].g + v_{item} – ((*thread*+w_j) *mod* w_1)v_1
            cand_boundary = T[*item*,*thread*].b + w_j
            IF *cand_gain* > T[*item*,*next_thread*].g
                T[*item*,*next_thread*].g = *cand_gain*
                T[*item*,*next_thread*].b = *cand_boundary*
            ELSE IF *cand_gain* = T[*item*,*next_thread*].g
                    AND *cand_boundary* < T[*item*,*next_thread*].b
                T[*item*,*next_thread*].b = *cand_boundary*
            thread = *next_thread*
            jumps_so_far = *jumps_so_far*+1

By the end of algorithm's execution, cells $T[n,0]$ through $T[n,w_1–1]$ contain maximum gains achievable in respective threads (by using all available types of items), and smallest possible boundaries where such gains occur (b^*).

The algorithm runs in $O(nw_1)$ space, since it requires a table of n-by-w_1 cells with constant-sized cells. It runs in $O(nw_1)$ time, since it visits each cell at most twice, spending constant time processing a cell.

This information can be used to solve the knapsack problem for any sufficiently large boundary b in linear time – by calculating the thread to which the boundary belongs, checking against the appropriate b^*, and using the gain supplied to calculate precise solution for the given boundary.

Note also that since any meaningful w_1 is smaller than b, this algorithms runs faster than the regular DP-based algorithm, which makes it possible to run Sage-2D before the DP-based algorithm by at worst not even doubling the execution time, and at best spending as much less time as b/w_1, which for many practical problems is a huge factor.

4.6 Algorithm extensions

The Sage-2D algorithm as specified in section 4.5 builds information about the highest achievable gain in every thread, as well the information about the smallest boundaries for which those

gains are achieved. This information can be useful if the problem at hand involves maximizing the total knapsack value for sufficiently large boundaries.

In practical tasks, more fine-tuned selection can be required. As specified in a footnote in section 4.1, more selection criteria such as reachability can be added to the algorithm or tracked by it. As long as those extra criteria are applicable only in case of ties on the values and boundaries, all of them can be safely added, since the algorithm considers *all* possible candidates with tying gains and boundaries for every thread.

One of the most important practical additional requirements to the algorithm would be the ability to reconstruct the actual item count for each type that comprises the solutions found by the algorithm. Such recovery procedure can be added to the algorithm by adding extra information to the cells (constant size, nevertheless), and tracking that information (also constant time per cell). This additional information would include the last type of item added and the number of items of the type used by the solution. The important detail is that such tracking allows solution recovery in $O(n)$ steps, since there are at most n type information entries to recover. It is not sufficient to simply track the last *item* used, since the solution may consist of the number of items of order $O(nw_1)$, and the recovery would thus become exponential.

5 Algorithm Sage-3D: pre-solution for all boundaries

The Sage-3D algorithm builds pre-solution information that consists of all gains achievable in every thread (not just the optimal gain) and their corresponding smallest boundaries. This information is enough to solve the given knapsack problem for any boundary with no restrictions.

5.1 Approach

At the top level, the Sage-3D algorithm is similar to the Sage-2D algorithm. It assumes the types have been preprocessed so that the first type is the best type. It assumes that there exists at least one type of items; otherwise the solutions become trivial (section 2.3). It initializes by processing the best type in a special way – knowing without any processing all the properties of solutions based on the best type only. It then proceeds to process all types in the setup one-by-one, essentially solving several sub-problems (involving two types, then three types, etc.). While processing every type, it loops through all the chains, and all the threads in each chain, to update the thread information.

What is different, however, is the amount of information kept regarding each particular thread. While Sage-2D always preserves information about one solution per thread (the one yielding the best gain), the Sage-3D preserves information about several solutions per thread. This "extended" information contains not only the best solution for the thread (including best achievable *gain* and corresponding b^*), but rather *all* achievable gains that make sense for the thread, and their corresponding smallest boundaries for which those gains are achieved. Smaller boundaries correspond to smaller gains. If a certain gain is achievable at some boundary, but a larger gain is achievable at smaller boundary, then the information about the smaller gain is discarded as not making sense.

Just as the principle of optimality lies at the heart of the Sage-2D algorithm and dictates that sub-solutions to optimal solutions are optimal themselves, thus allowing the gradual build-up of relevant information, the same principle lies at the heart of Sage-3D. The notion of *optimal* somewhat changes, but the principle still holds and prescribes that any solution worthy enough to be preserved is based on sub-solutions that are also as worthy.

5.2 Principle of optimality: relationship between different solutions not involving the best items

The Sage-3D algorithm has to track all relevant per thread – all useful gains and their corresponding boundaries.

It has already been shown in section 3.2 that a solution involving the best item has the same gain as the sub-solution without it. Therefore, if a certain gain is achieved in a thread, the larger boundaries have gains no worse than this one. Hence, optimal solutions to all boundaries in a thread exhibit non-decreasing gains as boundary increases. For certain boundaries the gain is greater than the gain for the previous boundary in the thread. For all others the gain is the same as for the previous boundary.

All smallest boundaries that exhibit changes in gain as compared to the previous boundary in the thread have no best item as part of the optimal solution. If they did, their gain would not be different from the previous boundary in the thread.

Therefore, the key boundaries that contain enough information about solutions to all boundaries in the thread have two properties:

- Their solutions do not contain the best items;
- Their gain is greater than any gain corresponding to smaller boundaries in a thread;

As shown in section 3.2, there are exactly v_1 distinct values that a gain have – integers from 0 to v_1–1. Therefore the maximum information about a particular thread involves information about v_1 distinct gains – the flag indicating whether the particular gain appears in optimal solutions at all, and the minimum boundary where such a gain is achievable in case it does appear in optimal solutions.

Information about each such gain refers to a particular solution that is comprised of several items. The sub-solutions of those must also be optimal under the conditions described above. If it wasn't so, it would be possible to replace the sub-solution with a better one (bigger gain, or smaller boundary, or both), and thus provide a better solution for the thread in question (bigger gain, or smaller boundary, or both).

Thus the principle of optimality holds true for every sub-solution of any solution that satisfies the criteria above.

5.3 Adding a new type to existing solution: chains

The same logic about chain traversal as for the Sage-2D (described in section 4.2) applies for Sage-3D. The only difference is that analyzing a pair of threads (parent and child) involves checking for more than one solution, but rather for all solutions that parent thread has – since each one of them may potentially be a sub-solution for the child thread.

5.4 Starting point in chains

In case of chain 0, just as in Sage-2D, the traversal of the chain can start from thread 0, since no solution in thread 0 has any non-best items.

In case of chains 1 and higher, the traversal still requires a double loop technique in order to ensure that all optimal solutions are found. Even though the starting boundary may belong to different threads in case of different solutions (different gains), a double loop is still enough for all the same reasons.

5.5 Clean-up stage

Tracking described in previous sections is guaranteed to track all the optimal solutions as defined in section 5.2. However, it may also pollute the information table with unnecessary information.

The pollution happens when a certain gain becomes feasible for a thread whose optimal solutions do not have such a gain. Since the thread does not have this gain in any of the optimal solutions, it means that the particular gain is achievable for a boundary that is larger than an existing boundary for a higher gain – making the newly found solution sub-optimal.

It is possible to check every new gain achieved for a thread to comply with the compatibility rules at the time of processing the gain, but this would involve checking multiple additional cells while processing just one cell, and thus making the order of the running time significantly worse.

Instead, Sage-3D utilizes delayed clean-up. The table is checked after processing each item. Processing each item involves checking at most $w_1 v_1$ cells, and clean-up involves checking exactly as many cells, after all optimal solutions are guaranteed to have been filled in. Since it is essentially one more sweep of the table, the time complexity of the algorithm is not hurt.

In realistic environments, it might not be necessary to clean up the whole table by cleaning up every portion corresponding to every item. If intermediate solutions (corresponding to setups with a smaller number of types) that are produces "automatically" on the way are not important, the clean-up may be invoked only on the final slice of the table – corresponding to the whole setup only.

For the sake of clarity, considering that the clean-up does not really worsen the tight bound of the algorithm running time, we present the clean-up stage after processing every type, and leave further optimizations that do not have academic interest outside the scope of the paper.

Finally, the actual usage of the table once it is built requires speedy lookup of actual boundaries. The table is already pre-sorted in terms of boundaries, but the useful information is sparse. Therefore, in order to assist the binary search of boundaries, the clean-up stage also performs the initialization of all boundary values of the non-feasible gains to the value of the closest of the larger feasible boundaries.

5.6 Sage-3D: the algorithm

The algorithm build a three-dimensional (hence the name) table T, whose first dimension corresponds to the types of items analyzed ($1 \ldots n$), the second dimension corresponds to the threads ($0 \ldots w_1 - 1$), and the third dimension corresponds to various values of gains that might be achieved by optimal solutions ($0 \ldots v_1 - 1$). Each cell $T[x,y,z]$ contains information about the optimal solution achieving gain z, in thread y, if items types 1 through x are used. The cell contains a boolean flag indicating if the gain is feasible (i.e. if the gain is achievable by some optimal solution) – $T[x,y,z].f$, and the smallest boundary whose optimal solution does achieve the gain in question – $T[x,y,z].b$.

The algorithm assumes that there is at least one type of items (otherwise the solution is trivial, as discussed in section 2.3), and that the item of highest density and smallest weight has index of 1.

Algorithm Sage-3D

```
FOR thread = 0 TO w_1–1
        T[1,thread,0].f = TRUE
        T[1,thread,0].b = t
        FOR gain = 1 to v_1–1
                T[1,thread,gain].f = FALSE
                T[1,thread,gain].b = 0
FOR item = 2 to n
        FOR thread = 0 to w_1–1
                FOR gain = 0 TO v_1–1
                        T[item,thread,gain] = T[item-1,thread,gain]
        number_of_chains = gcd(w_1,w_item)
        FOR chain = 0 TO number_of_chains
                IF chain = 0
                        number_of_jumps = w_1/number_of_chains–1
                ELSE
                        number_of_jumps = 2w_1/number_of_chains–1
                thread = chain
                jumps_so_far = 0
                WHILE jumps_so_far < number_of_jumps
                        next_thread = (thread + w_j) mod w_1
                        FOR gain = 0 TO v_1–1
                                IF T[item,thread,gain].f = FALSE
                                        CONTINUE
                                cand_gain = gain + v_item – ((thread+w_j)) mod w_1)v_1
                                IF cand_gain < 0
                                        CONTINUE
                                cand_boundary = T[item,thread,gain].b + w_j
                                IF T[item,next_thread,cand_gain].f = FALSE
                                                OR cand_boundary < T[item,next_thread,cand_gain].b
                                        T[item,next_thread,cand_gain].f = TRUE
                                        T[item,next_thread,cand_gain].b = cand_boundary
                        thread = next_thread
                        number_of_jumps = number_of_jumps+1
        FOR thread = 0 TO w_1
                FOR last_gain = v_1–1 TO 1 STEP –1
                        IF T[item,thread,last_gain].f = TRUE
                                BREAK
                        T[item,thread,last_gain].b = –1
                FOR gain = last_gain–1 TO 0 STEP –1
                        IF T[item,thread,gain].f = FALSE
                                T[item,thread,gain].b = T[item,thread,last_gain].b
                                CONTINUE
                        IF T[item,thread,gain].b ≥ T[item,thread,gain].b
                                T[item,thread,gain].f = FALSE
                                T[item,thread,gain].b = T[item,thread,last_gain].b
                                CONTINUE
                        last_gain = gain
```

By the end of algorithm's execution, cells $T[n,t,0]$ through $T[n,t,w_1-1]$ contain information about optimal gains achievable in thread t, as well as the smallest possible boundaries corresponding to gains achievable.

The algorithm runs in $O(nw_1v_1)$ space, since it requires a table of n-by-w_1-by-v_1 cells with constant-sized cells. It runs in $O(nw_1v_1)$ time, since it visits each cell at most four times (once for initialization, at most twice for processing, once for cleanup), spending constant time processing a cell.

This information can be used to solve the knapsack problem for any boundary b in linear time – by calculating the thread to which the boundary belongs, then checking the cells corresponding to the appropriate cell to find the closest feasible boundary not exceeding the boundary from the problem statement, and using the gain corresponding to the found boundary to calculate precise solution for the given problem. Note that in order to find such an entry in the table in polynomial time, the v_1 cells corresponding to the thread must be searched using a binary search (since they are sorted with respect to boundaries), resulting in $O(log(v_1))$ search time, or linear with the bits of encoding. A brute-force search of the table would result in $O(v_1)$ search time, or exponential with the bits of encoding.

5.7 Algorithm extensions

The Sage-3D algorithm as specified in section 5.6 builds information about all optimal achievable gains in every thread, as well the information about the smallest boundaries for which those gains are achieved. This information can be useful if the problem at hand involves maximizing the total knapsack value for any given boundaries.

In practical tasks, more fine-tuned selection can be required. As specified in a footnote in section 4.1, more selection criteria such as reachability can be added to the algorithm or tracked by it. Just as for the Sage-2D algorithm, as long as those extra criteria are applicable only in case of ties on the values and boundaries, all of them can be safely added, since the algorithm considers *all* possible candidates with tying gains and boundaries for every thread.

One of the most important practical additional requirements to the algorithm would be the ability to reconstruct the actual item count for each type that comprises the solutions found by the algorithm. Such recovery procedure can be added to the algorithm by adding extra information to the cells in the same fashion as for Sage-2D, which is discussed in section 4.6.

6 Conjectures

This paper presents two algorithms, Sage-2D and Sage-3D, that provide a qualitatively different approach of solving knapsack problems. Instead of attacking a particular problem instance with a particular boundary, the algorithms build pre-solution information that can be used to solve particular instances of the given problem for *any* boundary (with certain restrictions in case of Sage-2D algorithm).

Even though state-of-the-art DP-based precise solutions to multiple instances of unbounded integer knapsack problems may involve only one execution – for the largest boundary of all sets, the running time of such algorithms heavily depends on the largest boundary, which can make the actual time performance in realistic environment unfeasible. The Sage algorithms do not make such a distinction between boundaries, and provide sufficient information for all of them, regardless of size of the boundary constraints.

Since knapsack problems are directly related to a number of other integer programming problems, other problems may also be solved by algorithms directly resembling Sage-2D and Sage-3D.

The fact that Sage-2D runs in relatively smaller time complexity than Sage-3D and even DP-based algorithms, suggests that the larger instances of the problem that are commonly feared to represent the worst cases are in fact best cases. Solving relatively larger instances is conceptually easier than solving smaller instance. This, in conjunction with the concept of a common pre-solution algorithm and a polynomial final-solution algorithm, suggests that a finer categorization of NP-complete problems may in fact be necessary to better analyze the practical complexity of those problems.

7 Resources

An online demo of the Sage-2D and Sage-3D algorithms is available on the web at http://www.leolan.com/research/knapsack/sage/ – the interactive script runs entered setups through Sage-2D or Sage-3D algorithms, providing the resulting thread view of the solutions, as well as traces of the algorithms' execution.

8 References

[1] P.C. Gilmore and R.E. Gomory, "The Theory of Computation of Knapsack Functions", *J. ORSA*, pp. 1045-1074, 1966.