Exact Algorithms and Useful Tricks for the Knapsack Problem

Author: Shibiao JIANG
Professor: Guochuan ZHANG
Design and Analysis of Algorithms
Zhejiang University

Date: June 27, 2020

Abstract

This is my final report for *Design and Analysis of Algorithms* course. I love this course very much and also summarize some interesting knowledge. Because of my competition background, I choose the Knapsack Problem as the topic of my final report. In this report, I go deep into some classical Knapsack Problems like Unbounded Knapsack Problem, Bounded Knapsack Problem, Subset-sum Problem and Unbounded Subset-sum Problem with K items. I only focus on those beautiful or fast exact algorithms.

Keywords: Knapsack Problem, Exacting Algorithm, Unbounded, Bounded, Subset-sum

1 Introduction

From Wikipedia, the **Knapsack Problem** is a problem in combinatorial optimization: Given a set of items, each with a weight and a value, determine the number of each item to include in a collection so that the total weight is less than or equal to a given limit and the total value is as large as possible.

The knapsack problem has been studied for more than a century, and the most famous solution is a pseudo-polynomial time algorithm using **dynamic programming**.

However, the decision problem form of the knapsack problem (*Can a value of at least V be achieved without exceeding the weight W?*) is **NP-complete**, thus there is no known algorithm both correct and fast (polynomial-time) in all cases. It's not strange that many researches nowadays are dedicated to the approximate solutions of knapsack problem, which are usually tedious and boring.

Assume that **the knapsack size is also a polynomial**, things will become more interesting[1]. Denote the number of items as N and the total size of knapsack as M, we are supposed to design "polynomial time" algorithms for both N and M. In this report, I will make a simple review of classical exact algorithms for the Knapsack Problem, and focus on some novel and interesting exact algorithms which usually work well under some special conditions.

[1]This topic is also popular in the programming contests

2 Unbounded Knapsack Problem

Unbounded Knapsack Problem is a variation from 0-1 Knapsack Problem.

Problem: We have a knapsack with total size M. N items are given, each with weight $w_i (1 \leq w_i \leq u)$ and value v_i. We are supposed to maximize the total value without exceeding the backpack's size. Different from 0-1 Knapsack Problem, each item is unbounded so repetition is allowed.

Parameters: N, u, M.

2.1 Modification from 0-1 Knapsack Problem

0-1 Knapsack Problem has a classical pseudo-polynomial time algorithm using dynamic programming. After some minor modifications, it can be applied to the infinite knapsack problem.

The time complexity is $O(NM)$.

Algorithm 1 Modification from the 0-1 Knapsack Problem

```
f_0 ← 0
for i = 1 → n do
    for j = w_i → M do
        f_j ← max(f_j, f_{j−w_i} + v_i)
    end for
end for
```

2.2 An Important Theorem

Can we separate N and M from the time complexity? In many situations, M may be very huge but N and u can be relatively small. So we focus on the time complexity only related to N and u.

In 2018, Bateni et al[2] introduce an important theorem[3] and propose a $O(Nu + u^2 \min(N, u))$ algorithm (In fact, we can always assume that $n \leq u$, so this complexity can be simplified to $O(Nu^2)$).

Theorem 1: Let s be the item with the maximum value-size ratio (i.e. $\max_s \frac{v_s}{w_s}$). There must exist a optimal solution so that the sum of weights we select except s doesn't exceed u^2.

Basic Observation: Assume the items we select except s-th form a set S. If there is a subset $S'(S' \in S)$ satisfied that $w_s | (\sum_{i \in S'} w_i)$, we can replace S' with several s-th items and the sum of value won't decrease. So

[2]Bateni M H, Hajiaghayi M T, Seddighin S, et al. Fast algorithms for knapsack via convolution and prediction[C] Symposium on the Theory of Computing (STOC). 2018.

[3]It seems that this theorem should have been proposed a long before, but I can't find an earlier source.

there always exists an optimal solution S with no subset S' satisfied $w_s|(\sum_{i\in S'} w_i)$.

Lemma: Assume set S contains several positive integers (repition allowed). We can always find a subset S' from S such that $L|\sum_{x\in S'} x$ where $L = |S|$. We can easily prove it by **Pigeonhole Principle**.

Proof: From Basic Observation, there exists an optimal solution such that no subset S' satisfied that $w_s|(\sum_{i\in S'} w_i)$. Combined with Lemma we know $|S| \le w_s$, and finally $\sum_{i\in S} w_i \le w_s \cdot u \le u^2$.

2.3 An Algorithm Applying Theorem 1

From the Theorem we know the sum of weights except s-th item won't exceed u^2 ($u \cdot w_s$). So the upper bound for dynamic programming can be limited by $u \cdot w_s$. After working out f_i, we just enumerate the sum of rest weights from 0 to $w_s \cdot u$ and update the answer.

The time complexity is $O(Nu^2)$.

Algorithm 2 An algorithm Applying Theorem 1

```
s ← max_s v_s/w_s
f_0 ← 0
for i = 1 → n do
    for j = w_i → w_s · u do
        f_j ← max(f_j, f_{j−w_i} + v_i)
    end for
end for
for i = 0 → w_s · u do
    ans ← max(ans, f_i + ⌊(M−i)/w_s⌋ × v_s)
end for
```

2.4 A Better algorithm

This algorithm appears in a contest in 2016 [4].

Consider how we can calculate f_M instead of all elements from f_1 to f_M: Enumerate a part X and $f_M = \max_X(f_X + f_{M-X})$. Because $w_i \le u$, there always exists an optimal partition so that $|X - (M - X)| \le u$. From $|X - (M - X)| \le u$ we know $X \in [\frac{M-u}{2}, \frac{M+u}{2}]$.

Now we continuously consider how to work out the interval $[\frac{M-u}{2}, \frac{M+u}{2}]$. Use the similar inequality, we should first calculate the interval $[\frac{M-6u}{4}, \frac{M+6u}{4}]$. And in k-th turn the interval will not exceed:

$$[\frac{M}{2^k} - 2u, \frac{M}{2^k} + 2u]$$

[4] https://codeforces.com/gym/101064/problem/L

There are $O(\log M)$ turns, so $O(u \log M)$ dp states need to be calculated.

The time complexity is $O(u^2 \log M)$.

Algorithm 3 A Better algorithm Applying Theorem 1

```
procedure SOLVER_INTERVALS(l, r)
    if l > r then
        return
    end if
    SOLVER_INTERVALS(max(1, (l−u)/2), min((r+u)/2, M))
    for i = l → r do
        if There is a item with size i then
            f_i ← v
        end if
        for x in possible Partitions do
            f_i ← max(f_i, f_x + f_{i−x})
        end for
    end for
end procedure
SOLVER_INTERVALS(M, M)
```

2.5 Frontier Research

Use the inference from Timothy and Qizheng[5] in SOSA 2020, there exists an algorithm to solve out all f_i under u^2 with the time complexity $O(u^2 \log u)$.

Axiotis [6] gives a $O(u^2)$ solution in ICALP 2019,

Combined with the condition lower bound **No algorithm exists with $O(N+u)^{2-\epsilon}$ time complexity** provided by Cygan[7] and Künnemann[8], Unbounded Knapsack Problem has been studied completely.

[5]Timothy M. Chan, Qizheng He. On the Change-Making Problem[C] 3rd Symposium on Simplicity in Algorithms (SOSA), 2020

[6]Axiotis K, Tzamos C. Capacitated Dynamic Programming: Faster Knapsack and Graph Algorithms[C] 46th International Colloquium on Automata, Languages, and Programming (ICALP), 2019.

[7]Cygan M, Mucha M, Wegrzycki K, et al. On Problems Equivalent to (min,+)-Convolution[C]//44th International Colloquium on Automata, Languages, and Programming (ICALP), 2017, 80: 22

[8]M, Paturi R, Schneider S. On the Fine-Grained Complexity of One-Dimensional Dynamic Programming[C]//44th International Colloquium on Automata, Languages, and Programming (ICALP), 2017.

3 Bounded Knapsack Problem

Bounded Knapsack Problem is similar with Unbounded Knapsack Problem by an extra condition.

Problem: We have a knapsack with total size M. N items are given, each with weight $w_i (1 \le w_i \le u)$ and has c_i copies. We are supposed to select a subset to maximize the total weight without exceeding the backpack's size and the limitation of each item.

Parameters: N, u, M, L, where $L = \sum_{i=1}^{N} c_i$.

3.1 Reduce to 0-1 Knapsack Problem

We can reduce Subset-sum Problem to 0-1 Knapsack Problem. For each item i, we add c_i copies to 0-1 Knapsack Problem with equal w_i and equal v_i. Then we calculate this 0-1 Knapsack Problem with $N' = L$.

The time complexity is $O(M \sum c_i)$ or $O(ML)$.

Algorithm 4 Reduce to 0-1 Knapsack Problem

```
f_{0,0} ← 0
for i = 1 → n do
    for j = 0 → c_i do
        for k = u_i · j → M do
            f_{i,k} ← max(f_{i,k}, f_{i−1,k−u_i·j} + v_i · j)
        end for
    end for
end for
```

3.2 Binary Grouping

From binary system, we know each number K can be expressed as the sum of the powers of 2. So we don't need to enumerate from 0 to c_i, only need to enumerate the powers of 2.

The time complexity is $O(M \sum \log c_i)$.

3.3 Monotone Priority Queue

For each item i, we divide $0 \sim M$ into v_i parts. For example, the first group is $\{0, v_i, 2v_i, \dots\}$ and the second is $\{1, v_i + 1, 2v_i + 1, \dots\}$. State transitions only occur in the same group.

Then we can apply **monotone priority queue**. The time complexity is $O(MN)$.

Algorithm 5 Binary Grouping

$f_{0,0} \leftarrow 0$
for $i = 1 \rightarrow n$ **do**
    for j **in** $[0, 1, 2, 4, 8, \dots]$ **do**
        for $k = u_i \cdot j \rightarrow M$ **do**
            $f_{i,k} \leftarrow \max(f_{i,k}, f_{i-1,k-u_i \cdot j} + v_i \cdot j)$
        end for
    end for
end for

4 Subset-sum Problem

Subset-sum Problem is a special case of of Knapsack Problem where each kind of item, the weight equals the value.

Problem: We have a knapsack with total size M. N items are given, each with weight $w_i (1 \leq w_i \leq u)$. We are supposed to select a subset to maximize the total weight without exceeding the backpack's size.

Parameters: N, u, M.

4.1 0-1 Knapsack Problem and Bitset Tricks

We can reduce Subset-sum Problem to 0-1 Knapsack Problem with $v_i = w_i$. The time complexity is $O(NM)$, and also $O(N^2 u)$ if M is implicit.

Besides, it's very useful for Subset-sum Problem to use bitset tricks.

In the calculation of Subset-sum Problem, the type of array can be transformed from *integer* to *boolean*, since we do not care the value. And bitset is a famous class that emulates an array of bool elements and optimized for space allocation: generally, each element occupies only one bit. Due to the calculation method of computers, doing bit operations between bitset classes can be very fast.

More specifically, we contruct N bitsets $f_1 \sim f_N$, each with length M. Then we modify the general 0-1 knapsack algorithm with bit operations.

The time complexity can be regard as is $O(NM/w)$ or $O(Nu^2/w)$ when N, u is small, where w means the word length of computer (typically 32 or 64).

Algorithm 6 Add Bitset Tricks to General 0-1 Knapsack Algorithm

$f_{0,} \leftarrow \{True, False, False, \dots, False\}$
for $i = 1 \rightarrow n$ **do**
    $f_i \leftarrow f_{i-1} \vee (f_{i-1} << w_i)$
end for

4.2 Reduce to Balanced Subset-sum Problem

Pisinger[9] proposes a beautiful algorithm with time complexity $O(Nu)$.

Consider the **Balanced Subset-sum Problem**: We are given a vector v of integers (does not have to be positive). Let $u = |v|_\infty$. We want to find a subset that sums to t. Particularly, $t \leq u$ instead of $t \leq u \cdot |v|$.

Theorem 2: Each Subset-sum Problem on N items can be reduced to a Balanced Subset-sum Problem within $O(N)$ time.

Proof: Greedily find a subset S from the Subset-sum Problem, such that adding other element will exceed M. Denote the sum of weights in S is M'(i.e. $M' = ||S'||_1$). Now we **negate** all the weights in S (Other weights remain same), and ask for Balanced Subset-subset Problem with target number $M - M'$.

Pisinger proposed a complex algorithm with $O(Nt)$ time to solve the Balanced Subset-subset Problem. So the original Subset-sum Problem has an algorithm with $O(Nu)$.

4.3 Frontier Research

In 2015, Koiliaris and Xu[10] found a deterministic $O(M\sqrt{N})$ algorithm for the Subset-sum Problem.

In 2017, Bringmann[11] found a randomized $O(M)$ time algorithm, which reaches the lower bound.

5 Unbounded Subset-sum Problem with K items

In this chapter, we add two more conditions to Subset-sum Problem: Unbounded and exactly K items.

Problem: We have a knapsack with total size M. N items are given, each with size $w_i (1 \leq w_i \leq u)$. We are supposed to find all possible S so that $S \leq M$ and there exists at least one solution to fill the knapsack with size S using exactly K objects. Each item is unbounded so repetition is allowed.

Parameters: N, K, u. (Note that M is implicit and $M \leq Ku$.)

5.1 Direct Implementation

We can directly apply general dynamic-programming algorithm from Unbounded Knapsack Problem. Use $f_{j,k}$ to describe whether size k can be made up if we have selected j objects from first i items.

The time complexity is $O(NK^2u)$.

[9]Pisinger D. Linear Time Algorithms for Knapsack Problems with Bounded Weights[J]Journal of Algorithms, 1999, 33(1):1-14.

[10]Koiliaris K, Xu C. A Faster Pseudopolynomial Time Algorithm for Subset Sum[J]ACM Transactions on Algorithms, 2015, 15(3).

[11]Bringmann K. A near-linear pseudopolynomial time algorithm for subset sum[C]Proceedings of the Twenty-Eighth Annual ACM-SIAM Symposium on Discrete Algorithms. Society for Industrial and Applied Mathematics, 2017: 1073-1084

Algorithm 7 Direct Application from the Unbounded Knapsack Problem

```
f_{0,0} ← True
for i = 1 → n do
    f_{0,0} ← True
    for j = 1 → K do
        for k = w_i → j · u do
            f_{j,k} ← f_{j,k} ∨ f_{j−1,k−w_i}
        end for
    end for
end for
```

5.2 Bitset Tricks Again

We apply the bitset again in the second dimension and adjust some operations.

The time complexity can be regard as is $O(NK^2u/w)$ when N, K, u is small, where w means the word length of computer (typically 32 or 64).

Algorithm 8 Biset Trick for Subset-sum Problem

```
f_0 ← {True, False, False, ..., False}
for i = 1 → n do
    for j = 1 → K do
        f_j ← f_j ∨ (f_{j−1} << w_i)
    end for
end for
```

5.3 Fantastic Transformation

The above approach seems "waste of space", because the type of array during dynamic programming is boolean. Now let me introduce a improved version [12].

Theorem 3: The original problem can be reduced to the following form: *Select **at most** K objects from N items to reach size S (and repetition is still allowed).*

Reduction: Consider the smallest item w_1 in the original problem. Construct a new problem with $N-1$ items $w_2 - w_1, w_3 - w_1, \ldots, w_N - w_1$, and K remains same. Whether size S can be formed in the original problem is equivalent to whether size $S + w_1 \cdot K$ can be formed in the new problem.

[12] I got to know this approach when I was involved in programming contest, but I can't find its source.

Proof: Consider the components of S in the old problem: $S = c_1w_1 + c_2w_2 + \cdots + c_Nw_N$, where $w_i \geq 0$ denotes the number of item i we select. Since $w'_i = w_i - w_1$,wso $S - w_1 \cdot K = c_2(w_2 - w_1) + c_3(w_3 - w_1) + \cdots + c_N(w_N - w_1)$, which just corresponds to the new problem. Note that the condition "exact K items" changes to "at most K item" because of the arbitrariness of c_1.

In conclusion, we can reduce **exactly K items** to **at most K items**. This transformation allows us to design a new dynamic programming algorithm: Use f_k to describe the minimum items to compose size k from first i items.

The time complexity reduces to $O(NKu)$.

Algorithm 9 New DP Approach for the Reduced Problem

```
f ← {0, +∞, +∞, ..., +∞}
for i = 1 → n do
    for k = w_i → Ku do
        f_k ← min(f_k, f_{k−w_i} + 1)
    end for
end for
```

5.4 Generating function

From the perspective of **generating function**[13], We can use polynomials to express "choices". The exponent of polynomials represents the total size, and its coefficient represents the number of ways. The following formula indicates the number of ways to choose one object from N items.

$$P(x) = x^{w_1} + x^{w_2} + x^{w_3} + \cdots + x^{w_{N-1}} + x^{w_N}$$

And selecting K objects is equivalent to

$$S_K(x) = P^K(x)$$

The number of ways to compose size S is equal to the coefficient of x^S in $S_K(x)$. So all we need to do is to work out all the coefficients in polynomial $S_K(x)$.

Direct approach to calculate the polynomial $S_K(x)$ is really slow. We multiply $P(x)$ K times, each time with polynomials of order $N \cdot u$ and order u, so the time complexity is $O(K^2u^2)$. The complexity is not good enough, but it supports a different perspective and can be optimized further.

[13]Wikipedia: Generating function

5.5 Fast Fourier Transform

One of applications of Fast Fourier Transform [14] is accelerating the convolution. With the help of FFT, multiplication of two polynomials with order N can be accelerated to $O(N \log N)$ [15].

We can not use FFT tricks directly in this problem, because when we calculate $S_K(x)$ in order, two polynomials don't have the same order. That's to say, the time complexity will reach $O(K^2 u \log(Ku))$.

A optimization can be done using divider-and-conquer technique. We divide K polynomials into two parts, recursively find the corresponding polynomials, and multiply them finally. The time complexity will reduce to $O(Ku \log(Ku) \log K)$.

There is another method making use of the nature of FFT. The first step of FFT is called DFT, which will transform a polynomial to a different representation. We can directly apply K multiplications[16] in this representation and then execute iDFT returning to the normal representation. The total time complexity is $O(Ku \log(Ku))$.

5.6 Running Times Review

In order to feel the true efficiency of the above algorithms, I designed two sets of parameters and worked out a table containing their approximate executing times.

Table 1: Approximate Executing Times for Different Algorithms

Algorithm Description	Time Complexity	$N, u, K \sim 1000$	$u, N \sim 100, K \sim 10000$
Direct Implementation from USP	$O(NK^2u)$	10^{12}	10^{12}
Bitset Tricks	$O(NK^2u/w)$	2×10^{10}	2×10^{10}
Fantastic Transformation	$O(NKu)$	10^9	10^8
Direct Generating function	$O(K^2u^2)$	10^{12}	10^{12}
Divide-And-Conquer	$O(Ku \log(Ku) \log K)$	2×10^8	2.6×10^8
Multiplications after DFT	$O(Ku \log(Ku))$	2×10^7	2×10^7

From the above table, we will find that *Fantastic Transformation* is a greatest approach among dynamic progamming algorithms. And FFT-based algorithms usually have smaller time complexity.

Besides, I designed a set of data with $N, K, u \leq 1000$ and implemented these methods in C++ to check their actual running times. I tried all the methods except *Direct Implementation*, which runs too slow.

[14] Wikipedia: Fast Fourier Transform

[15] The standard time complexity for FFT is $O(N \log N \log \log N)$, but we usually omit the right part. Nowadays some researcher points out that the right part can be truly left.

[16] To avoid the accuracy error, we usually use Number Theoretic Transform instead of Fast Fourier Transform.

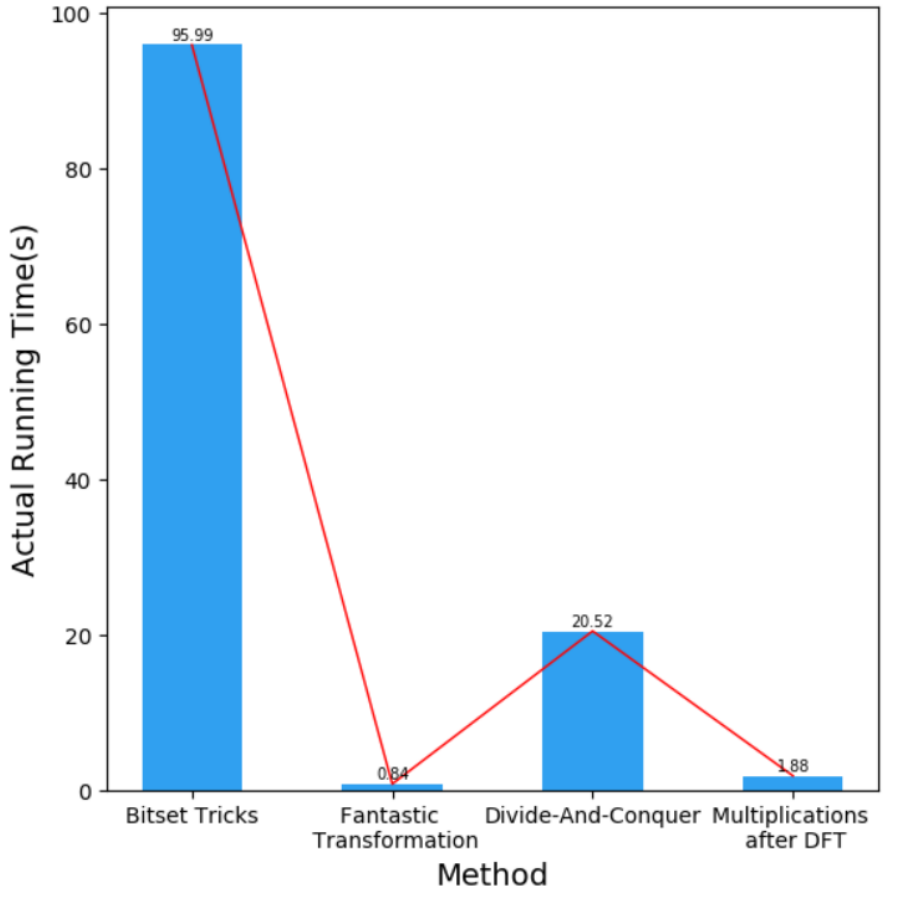

Figure 1: Actual Running Times for Different Algorithms when $N, u, k \sim 1000$ [17]

From the figure above, I found that the time complexity plays a decisive role in actual operation time. *Bitset Tricks* has the worst time complexity, so the executing time also takes much time.

However, due to the **huge constant factor** of FFT, I also found that *Fantastic Transformation* runs faster than *Multiplications after DFT* in the real case. Theoretically, the latter is 50 times faster than the former.

[17] **Compiling Command**: g++ -std=c++11 -O2