Vanilla Knapsack

and Its Various Flavors

An Phi

Skidmore College

Introduction

In this project, we explore several different algorithms (flavors) to solve Maximum 0-1 Knapsack, a classic NP-Hard problem, *in a reasonable amount of time*. In our experiments, we want to show the difference between running time and the optimal value obtained using these various algorithms.

We also want to take a side-track and demonstrate how Maximum 0-1 Knapsack can be used to solve 3SAT, the iconic NP-Complete problem, via the polynomial time reduction from 3SAT to Decision 0-1 Knapsack (O'Connell, 2013).

We are going to explore 4 different flavors of Knapsack solver, including:

- Maximum Value Dynamic Programming
- Minimum Cost Dynamic Programming
- Greedy 2-Approximation
- Fully Polynomial-time Approximation Scheme

1. Methodology

As noted, there are 2 main parts to this project. First, we will explore the differences between 4 different algorithms to solve Maximum 0-1 Knapsack. For the sake of brevity, we will use abbreviations to shorten the names of these algorithms, whose details will be discussed more thoroughly in section 2.

- MaxValDP – This refers to the $\Theta(nB)$ standard *vanilla* dynamic programming algorithm for the problem.
- MinCostDP – This refers to the $\Theta(n^2 v_{max})$ dynamic programming algorithm based on the MinCost version of the problem.
- Greedy – This refers to the greedy 2-approximation approach.
- FPTAS – This refers to the fully polynomial-time approximation scheme based on scaling with the optimal dynamic programming algorithm from MinCostDP.

As mentioned, we will attempt to explore how these different algorithms perform when used to solve the native Maximum 0-1 Knapsack problem, and 3SAT via the reduction from 3SAT to Decision 0-1 Knapsack. As such, to accommodate these objectives, we set up 2 workflows as presented in the subsections below.

1.1 Maximum 0-1 Knapsack

For this experiment, we generate 100 random instances of Maximum 0-1 Knapsack problem. We specify several constrains to the problem, such as:

- Each problem instance contains N item(s).
- Each item's value is an integer not exceeding 1000.
- Each item's cost is an integer not exceeding 1000.

Since we are also interested in exploring the running time of each algorithm, we decide that we will vary N within a range of values. By experiment, we found out that 700 items is probably the upper bound for the number of items we can have given the constraints for each item's maximum cost and value, otherwise our machine will run out of memory. We try for relatively small N starting from 10 and increment by 10 per iteration, i.e. $N \in \{10, 20 \ldots 690, 700\}$. As

for the maximum value of each item's cost and value, because these just indicate the upper bounds for the cost and value of each item, there is little value in varying them.

For each problem instance, we solve them using the 4 mentioned algorithms and record the optimal values obtained as well as the running times.

1.2 3SAT

For this experiment, we plan to generate 100 random instances of 3SAT problem. For each, we reduce from 3SAT to 1in3SAT, to SubsetSum, and finally to Decision 0-1 Knapsack (O'Connell, 2013). For the reduction from SubsetSum to Decision 0-1 Knapsack, we obtain an usual Knapsack problem where the budget and the target value are identical and where each item's cost and value are identical. We thus, can use MaxValDP and MinCostDP to find the optimal value. This values is then compared with the target value to decide if the original 3SAT problem instance is satisfiable. The reason why we cannot use Greedy nor FPTAS is that we need the optimal value (with no error) to compare with the target.

The procedure seems clear and indeed, we were able to produce the corresponding Decision 0-1 Knapsack for a given instance of 3SAT problem, but we face an interesting problem, which halts our further progress in this experiment. We will discuss this in section 3, *Reduction, Beauty and Peril*.

2. Algorithms

In this section, we will discuss each algorithms used in details; for a more thorough explanation and elegant proof for these algorithms, please refer to *What Is a Computer and What Can It Do?* by Thomas O'Connell.

First, the standard MaxValDP algorithm (the vanilla flavor that we are all too familiar to hate) is really the most standard way we know to solve Maximum 0-1 Knapsack problem. It attempts to construct in bottom-up manner a table of dimension $n \times B$, where each cell $Cell[i][j]$ gives the maximum value that can be achieved using item 1 to i with budget j. After the computation process, the bottom right-most cell of the table gives us the optimal value. The running time for this algorithm is $\Theta(nB)$.

Greedy refers to the greedy 2-approximation algorithm that tackles the Maximum Fractional Knapsack version. As such, firstly, we sort the item in ascending order based on value/cost ratio. We continuously add items to the knapsack until it is full. There might be a case where we can actually just take the item with maximum value and use it instead. Here, we need to make an assumption that the budget is higher than the maximum cost of any item–*we have enforced this condition while generating the knapsack problem instance so we can safely make this assumption.* This algorithm's running time depends on the running time of the sorting step and thus, it is $\Theta(n \log n)$.

MinCostDP deals with the MinCost version of the Maximum 0-1 Knapsack problem. Instead of looking for the set of item whose total costs does not reach the budget and whose value is the maximum, this version looks for set of item whose total values does not fall below the target value and whose budget is the minimum. As such, it employs the same strategy as MaxValDP. First, it constructs in bottom-up manner a table of dimension $n \times nv_{max}$; hence the columns of this table lists out all possible values that different subsets of item can take. Each cell of the table, $Cell[i][j]$ gives the minimum budget required using item 1 to i to achieve target value j. After the computation process, we scan through the bottom row of the table and find the cell with maximum column index whose value equals to the budget. The running time for this algorithm is $\Theta(n^2 v_{max})$. As we can see, by changing the original problem slightly, we now put the burden of the running on the maximum value rather than the budget, like in MaxValDP.

FPTAS attempts to reduce the running time of MinCostDP at the cost of accuracy. It scales the items' values down by a factor of $\frac{\epsilon \times v_{max}}{n}$, where ϵ stands for the percentage of error allowed, and performs truncation to obtain a new integer value for each item. It then runs MinCostDP and the obtained result is then re-scaled. As such, the running time for this algorithm is $\Theta(n^3 \frac{1}{\epsilon})$. For this particular assignment, we want to set ϵ to 0.5, i.e. this should make FPTAS comparable to Greedy (2-approximation), thus we can compare the performance of these 2 algorithms.

Here, we should note that since other than Greedy, other algorithms construct large table to perform dynamic programming. As such, this might set a limit on the size of input we use. As mentioned before in section 4, this puts an upper bound on the number of items used to 700. And yet, we will face this plaguing issue once again while performing the second experiment with 3SAT reduction, which will be discussed in the next section.

3. Reduction, Beauty and Peril

Initially, we tried to generate 100 large instances of 3SAT and try to reduce those to Decision 0-1 Knapsack and use either MaxValDP or MinCostDP to solve them. Nevertheless, we face problems with memory. This ceases any further attempts we made. We found 2 optimization that potentially allows us to have instances of 3SAT with just 1 clause working, but we deem this as minimal and not actually put in effort to execute that. We will give our excuse for this *sloth* later in this section.

There is definitely something of beauty about the reduction from 3SAT to Decision 0-1 Knapsack. We will not go through the full proof but just key points in the reductions; (again) for the elegant, complete explanation, we recommend reading *What Is A Computer and What Can It Do?*.

In the reduction from 3SAT to 1in3SAT, a clause like $(x_1 \vee x_2 \vee x_3)$ in the 3SAT problem instance will correspond to 3 clauses in 1in3SAT problem instance, i.e.

$$(\bar{x}_1 \vee a \vee b) \wedge (b \vee x_2 \vee c) \wedge (c \vee d \vee \bar{x}_3)$$

As we can see, we must triple the number of clauses and add 4 new variable. The reduction from 1in3SAT to SubsetSum, we create a pair of number v_i and v_i' for each variable x_i–this pair of numbers are made up of 1's and 0's to represent truth assignment of the corresponding variable as well as its impact to each clause–in the 1in3SAT instance; the target sum is of the length of each number in the set but made up of all 1's. The number of digits used for each new number equals to the sum of the number of clauses in the number of variables in the 1in3SAT problem instance. To reduce from SubsetSum to Decision 0-1 Knapsack, for each number in the set, we create an item with cost and value equal to that number. Then both the target and the budget are set to the target sum. As such, when using Maximum 0-1 Knapsack solvers, we are forced to choose the optimal value that equals (no more, no less) than the budget; and because we require this exactness, we must use dynamic programming algorithms to find the optimal value instead of using any approximation-based algorithms.

This chain of reductions is straight forward to code, but there is a problem. Let say we start with the minimal instance of 3SAT, with just 1 clause and 2-3 variables, then the budget and the maximum value of an item will be at of 9-10 digits (3 clauses and 6-7 variables in

the 1in3SAT problem instance). We tested the 3SAT instance $(x_0 \vee \bar{x}_0 \vee \bar{x}_1)$, which, after reduction, results in the following Knapsack problem instance:

```
Budget: 111111111
Target: 111111111
Max Value: 100000100
Item:
  1. Value: 100000010, Cost: 100000010
  2. Value: 100000100, Cost: 100000100
  3. Value: 10000000, Cost: 10000000
  4. Value: 10000001, Cost: 10000001
  5. Value: 1000100, Cost: 1000100
  6. Value: 1000000, Cost: 1000000
  7. Value: 100110, Cost: 100110
  8. Value: 100000, Cost: 100000
  9. Value: 10011, Cost: 10011
  10. Value: 10000, Cost: 10000
  11. Value: 1001, Cost: 1001
  12. Value: 1000, Cost: 1000
```

For this kind of situation, when we try to solve this Decision 0-1 Knapsack problem using either dynamic programming approach, Java will respond with the exception **java.lang.OutOfMemoryError: Java heap space.** The problem exacerbates when we use more than 2 clauses. For example, we tested out

$$(x_0 \vee \bar{x}_1 \vee \bar{x}_2) \wedge (x_0 \vee x_2 \vee x_3)$$

and go back the exception **java.lang.NumberFormatException: For input string: "1111111111111111"** as we try to construct the target sum for our SubsetSum problem. The reason is that we use Integer type for numbers in the SubsetSum problem instance, which only accommodates to the maximum value of $2^{32} \approx 4.3e9$.

As such, we thought of a few optimization in hope that it can improve the situation. For the number format problem, we can use type Long instead of Integer; even better, we can leave the number as String and use BigInteger class in Java to perform calculation with big numbers. The bigger problem, though, lies in the implementations of the algorithms used to solve Decision 0-1 Knapsack. Both dynamic programming approaches requires the construction of tables which base on either the budget (for MaxValDP) or the maximum value of an item (for MinCostDP). These numbers as shown above are too big and thus, such table drains memory really fast.

We observe that although these are really big table, they are really sparse. As such, we can try to use linked list to represent such sparse table. If we use a single linked-list, if we increases the number of clauses, time taken to iterates through all the nodes will be absurd.

If we use 2 linked-list, each of which represent header cells of the table' columns and rows, then we will end up with a linked-list of about a billion nodes or more, which also causes Java heap space to run out of memory.

Another observation we have is that since all numbers coming out of the SubsetSum problem instance consist of only 0 and 1, we can think of these number as binary and feed those number to Maximum 0-1 Knapsack solvers. Nevertheless, for this approach, we have tried to use 2 SAT clauses and yet again a table with 2^{20} column is still too big for Java to handle. We also think of an optimization where while reducing from 1in3SAT to SubsetSum, we can actually disregard the variable part in each number and only care about the clause part for each pair v_i and v'_i. The reason why we include the literal part is to make sure that we cannot choose both v_i and v'_i for the same value of i. As such, we can omit this part of the number and modify our algorithm slightly to do this check. One potential way is to keep track of which item is being taken, i.e. v_i is chosen, and thus skip v'_i while building further rows.

Even with this optimization, we only increase the number of clauses possible for the 3SAT problem instance to have to 2. And 2 clauses will correspond to a knapsack problem instance with 14 items. The result obtained from these considerably *small* Knapsack problem instance are not really reliable due to randomness in the problem generation phase. Also, it is expected that the running time of MinCostDP will be punished by the term n^2 whereas by the way we construct the numbers for the SubsetSum problem instance, the ratio $\frac{\text{budget}}{\text{maxValue}}$ will be approximately 1; and so we will find MinCostDP takes absurdly longer to run as compared to MaxValDP.

Through this section, we have presented the *peril* part of this experiment. We surely learned a lot out of it, i.e. that sometimes a theoretically brilliant approach might end up with no practically sound implementation. Perhaps, this could also due to our incompetence but for now, we decide to just move on with the other experiment, exploring the 4 flavors of Maximum 0-1 Knapsack solvers.

4. Result and Discussion

4.1 Accuracy

Below is the statistical summary for the values returned from the Knapsack experiments.

Name	Mean	Median	L.Quartile	U.Quartile	IQR	Min	Max
MaxValDP	131586	117691	55673	196774	141102	1736	361083
Greedy	131553	117650	55634	196754	141120	1706	361060
MinCostDP	131586	117691	55673	196774	141102	1736	361083
FPTAS	131437	117545	55555	196629	141074	1706	360857

It is totally expected that MaxValDP and MinCostDP have equal accuracy since both are supposed to produce the optimal value for each instance. To compare the accuracy of Greedy and FPTAS, we should look at figure 4.1 of the error produced by these 2 algorithms. Note that in this plot, we omit the outliers whose values range outside 1.5 times of the inner quartile range (IQR).

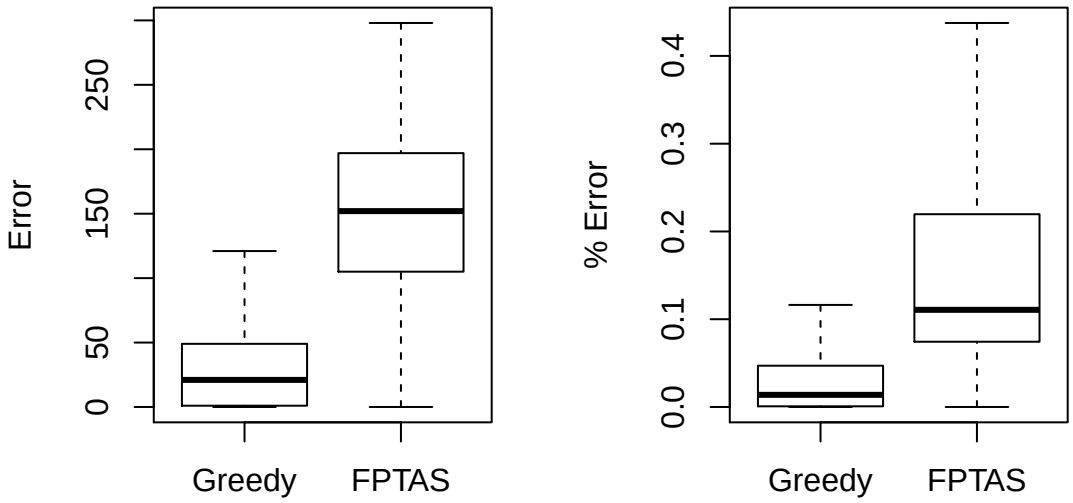

Figure 4.1: Error of FPTAS and Greedy algorithms

Clearly, FPTAS does a lot worse than GREEDY. This is expected to us because GREEDY is designed to obtain an optimal value in 50% range of the actual optimal value, while FPTAS is designed to work for arbitrary precision–in this case, we set ϵ to 0.5 which should also guarantee values in 50% range of the optimal value. To see the range of error that each algorithm produce, we can refer to the summary of percentage error below.

Name	Mean	Median	L.Quartile	U.Quartile	IQR	Min	Max
Greedy	0.096	0.014	0.001	0.047	0.046	0	16.935
FPTAS	0.26	0.111	0.074	0.22	0.145	0	7.02

From both the plot and the table, we can see that both algorithms actually does much better than expected, the percentage error falls way below 50%. This might have to do with the way we **randomly** generate the problem instances, since it is often the edge case that results in the 50% error. Also, we can see that the maximum percentage error produced by FPTAS is way less than that of GREEDY. This could be due to the mechanism of each algorithm. In section 2, we mentioned that GREEDY has a totally different approach to choose the items and has a final step where it might disregard previously obtained results and just choose the item with maximum value; whereas FPTAS essentially uses MINCOSTDP and its errors are due to the truncation while scaling up and down the values of items.

4.2 Running Time

Next, we present the summary for the running time (milliseconds) of the 4 different algorithms.

Name	Mean	Median	L.Quartile	U.Quartile	IQR	Min	Max
MaxValDP	151.532	68.02	14.452	224.857	210.406	0.038	956.868
Greedy	0.071	0.073	0.038	0.1	0.062	0.006	1.233
MinCostDP	419.992	282.004	54.472	725.475	671.003	0.179	1508.587
FPTAS	480.643	192.605	19.364	860.974	841.61	0.018	2242.264

These are just the summary of the result. Beside these, for each problem instance, we also collect the number of items N, the budget, and the maximum value, as these might help us understand the running time results better.

Name	Mean	Median	L.Quartile	U.Quartile	IQR	Min	Max
N	355	355	180	530	350	10	700
Budget	88645	66829	25605	134347	108742	877	341526

Name	Mean	Median	L.Quartile	U.Quartile	IQR	Min	Max
Max Value	994	998	995	1000	5	642	1000

Nonetheless, through the running time summary only, we can already see a general trend in running time. GREEDY is the clear winner in terms of speed. We have mentioned in section 2 that the running time of GREEDY is majorly contributed by the sorting step–the choosing step after that is rather trivial. As such, the running time depends on the input size, not the values of the items, not the budget.

MAXVALDP has a significantly higher running time as compared to that of GREEDY. This is expected as we can see the in the summary of instance information, the budget is much larger than the number of instance.

Another observation is that FPTAS and MINCOSTDP have almost similar running time. Again this can be explained by the fact that FPTAS uses MINCOSTDP internally. Their running time, though, is almost 3 times as big as that of MAXVALDP. This totally depends on the choice of budget and the maximum value of item in the problem instance.

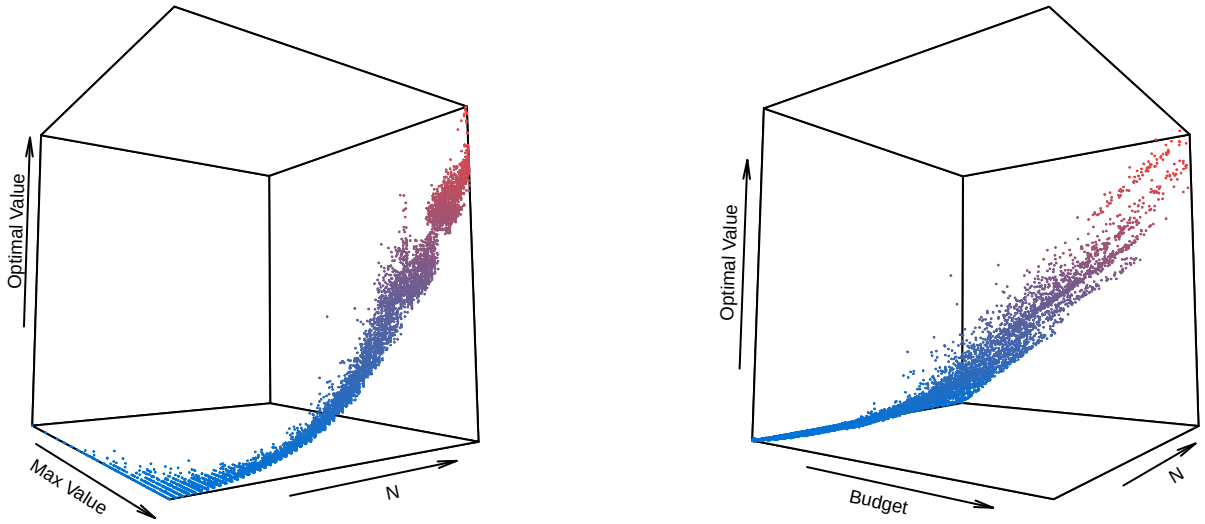

Figure 4.2: Running time of MinCostDP (left) and MaxValDP (right)

Since we conduct the experiment with changing value of N, we can then make 3D plots of running time against N and the budget for MAXVALDP and running time against N and the maximum value for MINCOSTDP. Refer to figure 4.2, for MINCOSTDP, based on the curvature of the plot surface, we can see that the relationship between N and the optimal value is clearly non-linear, in fact, we can even see that it is indeed quadratic as mentioned in section 2. Whereas, for MAXVALDP, we can see that the budget and N seem to be linearly related to the optimal value.

5. Conclusion

We can come up with the conclusion that MaxValDP, the vanilla flavor, shows why it is generally favored. It guarantees the optimal value (100% accuracy) and relatively acceptable running time. The versatile FPTAS actually does worse in both aspect, accuracy and running time; but to do justice to it, it shines for its versatility; as compared to other approximation method which targets a specific range of error, e.g. Greedy, it will not perform as good, otherwise, we will deem Greedy as useless and get rid of it. Greedy, though, runs amazingly fast and yields acceptable accuracy. The last flavor, MinCostDP is like chocolate. Hardly can anyone argues whether chocolate or vanilla is ultimately the better flavor. It is actually pretty hard to come up with an example to show that a type of pastries must come with chocolate instead of vanilla ... even brownies can come with vanilla. We guess the moral of the story is not that you can use either dynamic programming approach to solve any Maximum 0-1 Knapsack problem, but some go better with MaxValDP and others go better with MinCostDP, i.e. those problems with really high maximum value.

Bibliography

O'Connell, T. C. (2013). *What Is a Computer and What Can It Do?* College Publications, Milton Keynes, UK, 1st edition. ISBN 978-1-84890-098-1.