Final Report

Anh Vu Nguyen

April 26, 2016

Introduction

Throughout the semester, we have gone through different reductions and different way to solve Knapsack problem in reasonable time. This experiments aim to display the differences quantitatively in terms of algorithms' result as well as running time.

There are three parts to this experiment:

1. A general assessment of the value returned from the four algorithms implemented to solve Maximum Knapsack problem using 100 large random instances.

2. Assessment on running time using 100 large random instances of Maximum Knapsack.

3. Assessment on running time involving 100 large random instances of 3SAT, which are reduced into Subset Sum, and solved using alogirthms for Maximum Knapsack problem.

Notation

This report will use abbreviations to make it easier to know what algorithm that is being discussed:

- DP1 – This refers to the $O(nW)$ dynamic programming algorithm that we discussed in CS305
- DP2 – This refers to the $O(n^2 \cdot v(a_{max})$ dynamic programming algorithm from the textbook based on the MinCost version of the problem
- Greedy – This refers to the greedy 2-approximatino from the textbook
- FPTAS – This refers to the FPTAS based on scaling with the optimal dynamic programming algorithm from DP2.

Experiment Setup

The algorithms to generate running time are taken from pseudocode provided in CS305 as well as from "What is a Computer and What Can It Do?" (O'Connell). In total, I implemented four algorithms for solving Maximum Knapsack problem and also two reductions for decision problem (3SAT, 1in3SAT, and SubsetSum).

Since the reports focus on two different experiments, I set up two different workflows to accomodate this.

Maxmim Knapsack experiment

For this experiment, I set up 100 random instances of Maximum Knapsacks. There was a need to arbitrarily curb the maximum values for number of item, as well as each item's value and cost in order to make sure that each instances runs in reasonable amount of time (below 5 seconds for each algorithm). Each knapsack problem is constrained to the following attribute:

- At most 200 items

- Each item's value cannot exceeds 1000 and is an integer
- Each item's cost cannot exceeds 1000 and is an integer

Using this constraints, I also limit the range of Maximum Knapsack problem presented so that we have a sample with specified attributes. This limits the problem we would face when we have to group instances with 50 items and 5000 items should we not enforce the constraint. The calculation of density of instances then would not be as reflective of the actual algorithms due to skewness introduced by instances with a very large number of items.

Each instances solved using the four algorithms for solving Maximum Knapsack. The results (maximum value returned) as well as running time from each algorithm is logged into a text file (can be found in log folder of this project). I also wrote a simple Python script (can be found in scripts folder of this project) to parse this log file into a csv file for statistical analysis with R. This report as well as all the graphs are generated using R.

Reductions For 3SAT

For this experiment, I set up 100 instances of 3SAT. Each instance is then reduced to 1in3SAT, which is then subsequently reduced into Subset Sum and Knapsack respectively.

Since the program is written in Java, where number is constrained to 64-bits, there is a limit to how many clauses and variables I can have in 3SAT instances. This is due to the way we performed our reduction. When reduced from 3SAT to 1in3SAT, we know that the number of clauses will triple, and for each clause will introduce four new literals as well. When we reduce 1in3SAT, the total number of literals and clauses makes up the target for Subset Sum. Therefore, I have instances where we either exceed Java Heap Space while doing the reduction, or we ran out of number (and loop back to negative number, which breaks the algorithm). Therefore, there's first a need to perform a check on the maximum number of clauses 3SAT instances can have. Since the assignment of literals in clauses is randomized in my algorithms, I do not have control on this for now. After testing with several instances, I decide to limit the number of clauses to 6 clauses. This is also due to the way that I implement the reduction, which originally use integer, using double. I standardized the process so that we would always have double. An example is as follow:

If we have the item for subset originally as {100100, 100000, 010010, 010001} and a target of 111111, this would introduce .0 when dealing with double. To avoid this, I could scale the instances into the following {100.100, 100.000, 010.010, 010.001} and a target of 111.111. The results would be the same as before, but we deal with much smaller numbers.

The Four Algorithms for Maximum Knapsack

The expectation for the four algorithsm being implemented to solve Maximum Knapsack problem are summarized as follow:

1.
2.
3.
4.

Assessment of Maximum Knapsack Algorithms

Maximum Knapsack Returned Values

Values Returned from DP1

```
##    Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
##      72    5066   10390   19350   31690   81360
```

Values Returned from DP2

```
##    Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
##       0    4983   10250   19290   31480   81340
```

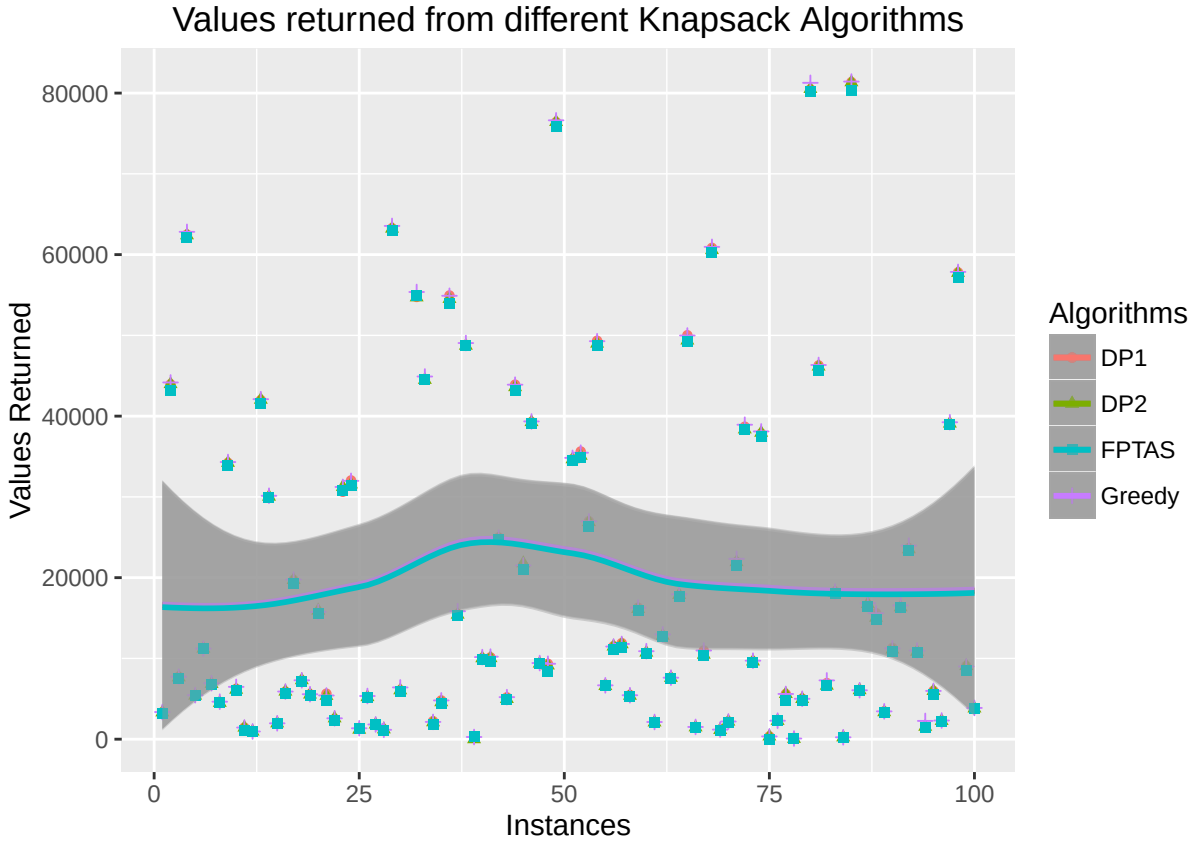
Values Returned from Greedy

```
##    Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
##      83    5055   10560   19450   31800   81430
```

Values Returned from FPTAS

```
##    Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
##       0    4758   10100   19070   31300   80330
```

Below is the graph plotting the returned values for each instances. The color indicates which algorithms is being used. At the same time, there's a trend line indicating the trend of values for each algorithm.



From the graph, we hardly see any differences in the values returned from the four different algorithms. The greedy algorithm seems to have the highest values returned, while DP2 returned the lowest. We cannot see the trend lines for DP1 as it is likely to be overlapped by another line.

Running Time

The following captures the general statistics of the running time of different algorithms (everything is in ms).

DP1 Running Time

```
##    Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
##   0.062   1.850   7.537  45.100  54.410 395.500
```

DP2 Running Time

```
##     Min.  1st Qu.   Median     Mean  3rd Qu.     Max.
##    0.054   33.920  114.000  360.800  615.700 1994.000
```

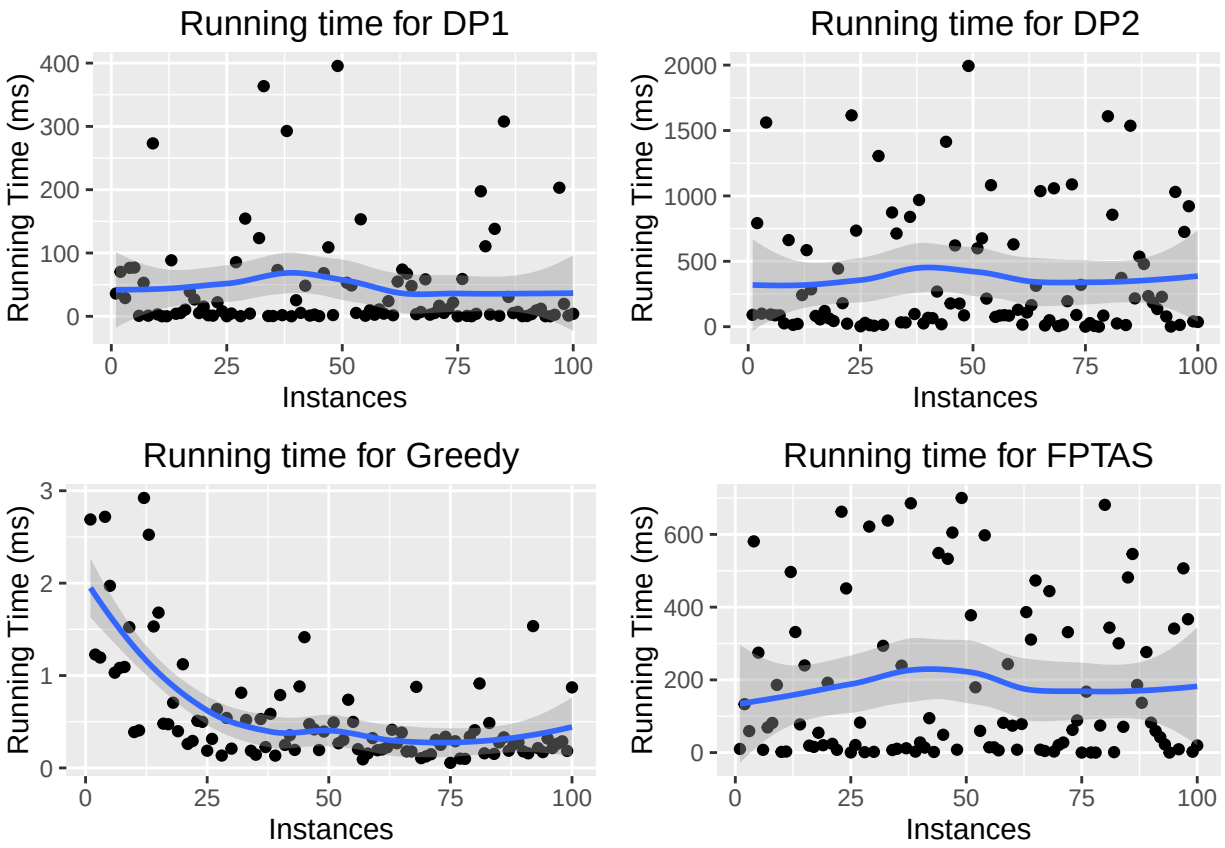
Greedy Running Time

```
##    Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
##  0.0550  0.1962  0.3270  0.5634  0.6240  2.9220
```

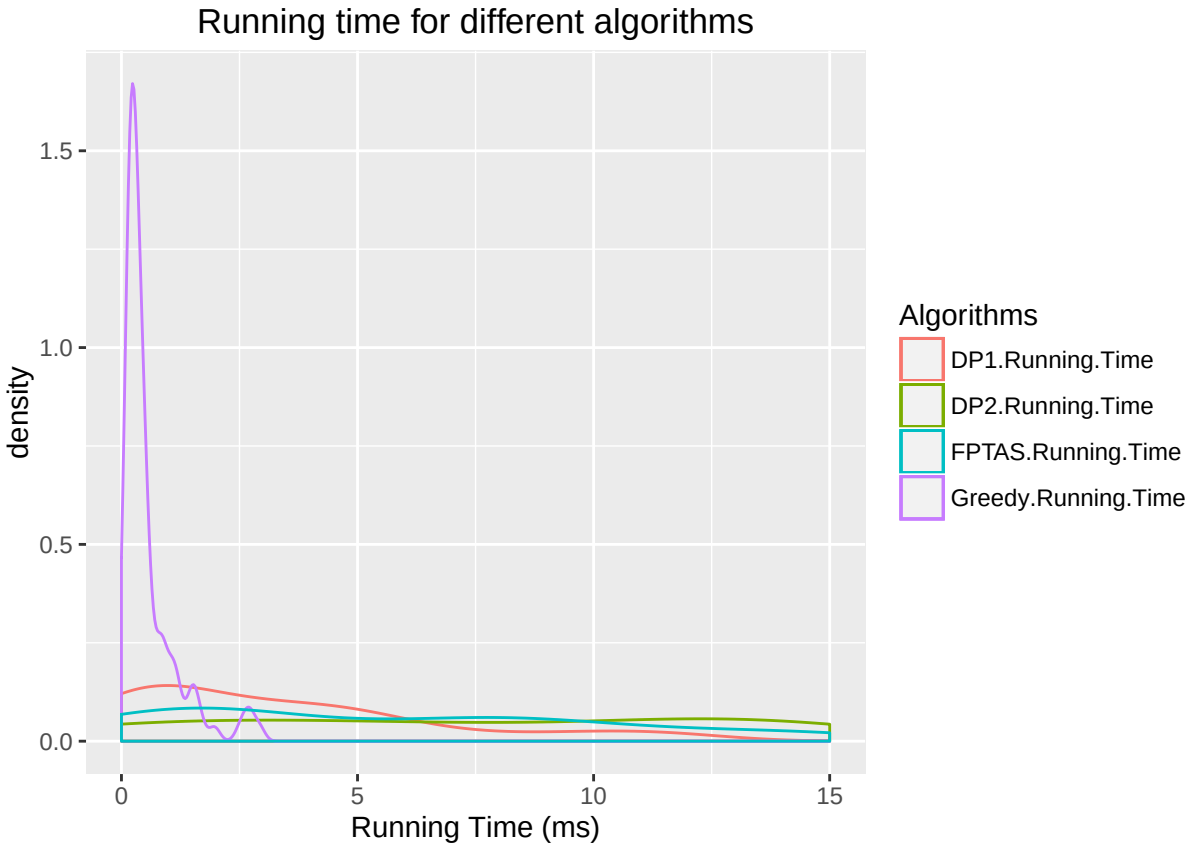
FPTAS Running Time

```
##    Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
##   0.046  10.830  74.260 181.800 326.200 700.300
```

Below is the graphs of running time for different instances. There is a smooth line to indicate the general trend of all the instances. Noticing the difference on the y-axis, which indicates the running time in millisecond (ms), we can see that Greedy runs in the range below 3ms while DP2 runs in the range 2000ms, or 2 seconds. FPTAS seems to have a higher upper bound for running time compared to DP1. FPTAS and DP2 graphs are very similar, despite the differences in scale on the y-axis. This is somewhat expected as FPTAS implements DP2 as part of its algorithm. From this graph alone, we would expect that Greedy would have the best running time.



Below is the density graphs for the running time of the four algorithms with our 100 instances. This graphs gives us information on how the running time is distributed among the testing instances.



From the graph, we can see clearly that Greedy stands out as having the most of it running time in the lower spectrum of the graphs. This aligns with our expectation that this would have the best running time. DP1 running time at the same time does not span as much as the DP2 and FPTAS.

Reduction For 3SAT

This analysis is focused on running time for the following sequence of actions:

1. Reduce 3SAT into 1in3SAT.
2. Reduce 1in3SAT into Subset Sum.
3. Reduce Subset Sum into 0-1Knapsack
4. Solve 0-1Knapsack using values returned from the four algorithms with Maximum Knapsack.

Improvements For Experiments

There are definitely rooms for improvements with this experiments that I could have done in order to yield better comparison.

Firstly, I could have increased the sample size from 100 to 1000, which should give us a better understanding regarding the values and running time. However, due to the circumstances of limited time frame to conduct this experiment, the limit of 100 instances in each experiment should be sufficient to give us a survey into our investigations.

Secondly, we could have implemented the same algorithms in different languages such as Python and C. By taking the average of the same algorithm in three different languages, it would help us get a running time

that is pseudo language-agnostic. The average for this number can be compared in the same manner in order to get the differences between different algorithms.

Conclusion