CS306 Experiment Report

Anh Vu L Nguyen

April 26, 2016

I. Introduction

Throughout the semester, we have explored different ways to solve Maximum Knapsack, an NP-Hard problem, in reasonable time. This experiments aim to display the differences quantitatively in terms of algorithms' result as well as running time. Additionally, Maximum Knapsack is versatile in its use to solve other problems, as it can be used to solve the decision vertion of the problem. The decision version of this problem is NP-Complete and therefore can be used to solve other NP-Complete problems, given the correct reductions. Therefore, this paper also aim to show the differences in time needed to solve a Maximum Knapsack instance that is reduced from 3SAT.

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, which are discussed below, using 100 large random instances.**

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

3. **Assessment on running time involving 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 for solving Maximum Knapsack instances. Below are the abbreviations:

- 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-approximation from the textbook.
- FPTAS – This refers to the FPTAS based on scaling with the optimal dynamic programming algorithm from DP2.

II. Experiment Setup

The algorithms used in this experiment 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 three reductions for decision problem (3SAT, 1in3SAT, and SubsetSum).

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

Maxmim Knapsack Experiments

For this experiment, I generated 100 random instances of Maximum Knapsacks. There was a need to arbitrarily curb the maximum values for number of items, as well as each item's value and cost in order to make sure that each instances runs in reasonable amount of time (which I arbitrarily established to be below 1 minutes for each algorithm). Each knapsack problem is constrained to the following attributes:

- Each instance contains 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 these constraints, I limited the range of Maximum Knapsack problem presented so that we have a sample with specified attributes. This helped us to avoid 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 instance was solved using all four algorithms for solving Maximum Knapsack. The results (maximum value returned) as well as the running time in miliseconds (ms) from each algorithm was logged in 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 generated 200 random instances of 3SAT. Each instance was then reduced to 1in3SAT, which was then subsequently reduced into Subset Sum and Knapsack respectively. The running time needed to solve the Knapsack instances and returns an answer for the decision version of the problem is recorded (the time for the reduction is not assessed as this would take same amount of time for all algorithms). All timings were still recorded in ms in order to maintain consistency.

Since the program is written in Java, where numbers are constrained to 64-bits, there is a limit to how many clauses and variables I could 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 solving Maximum Knasack, or we ran out of number (and loop back to negative number, which breaks the algorithms). More prominently, Java could not parse the array of digit into an int or double. Therefore, there was 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 found that the limit for 3SAT clause was 1 clause. However, I managed to come up with a tweak for our reduction that allow me to expand 3SAT clause to have 2 clauses.s

The tweak that I came up with is aimed to reduce the size of number that are passed in for Subset Sum instances as items' cost or values, and budget. In the original reduction, we use the number of literal (n) for v and v' in the digit that matches the literal number (e.g. literal #4 will have 1 in the 4th digit of the variable v and v' generated from this literal). The reason for this is to make sure that we do not use the same item more than once, which would mean that we chose true and false for the same literal. However, since we are adding v and v' into our list respectively always, we can use index in the arraylist to check for potentially taking two items that we cannot take. While using Knapsack algoithms to solve for this problem, if we take v, we would go through the list and make sure that we do not include v'. The opposite case would not happen since we would always take v' after taking v. This way, we would effectively only have to deal with the number of clauses as values for v and v'. However, this only helps us to get to 2 clauses in the orignal 3SAT instances. However, it's acknowledged that this tweak increase the actual running time (and space) for

the algorithms, especially for Greedy. Since Greedy sorts the values in the Knapsack based on the value over cost, this inevitably mixes our index up. To counter this, I use a HashMap to save the original index for each item. Every time we add an item, we check to see whether that item has index of an odd number, if it is, we exclude item at index+1 from the original set by adding that item into a HashSet. Before we do any evaluation with any item, we check to see whether this item is in the HashSet. If it is, meaning that it is supposed to be excluded, we would not evaluate that product and move on to the next item.

At the same time, in order to compensate for the small number of clauses in 3SAT, I generated 200 random instances instead of 100 like the Knapsack experiments. The larger sample size hopefully can give us a better understanding into the running time for this experiment.

III. The Four Algorithms for Maximum Knapsack

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

1. DP1 is expected to run slowly compared to the other algorithms.
2. Greedy is expected to run fastest compared to all other algorithms.
3. Since FPTAS is implemented based on DP2, the expectation is that the running times of the two are very similar.
4. As FPTAS scales the values of item, the expectation is that it would run faster than DP2 due to smaller size of table built along the way.

Assessment of Maximum Knapsack Algorithms

Maximum Knapsack Returned Values

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

Values Returned from DP1

```
##    Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
##       0    3967   11160   17700   26070   76930
```

Values Returned from DP2

```
##    Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
##       0    3967   11120   17610   25660   76470
```

Values Returned from Greedy

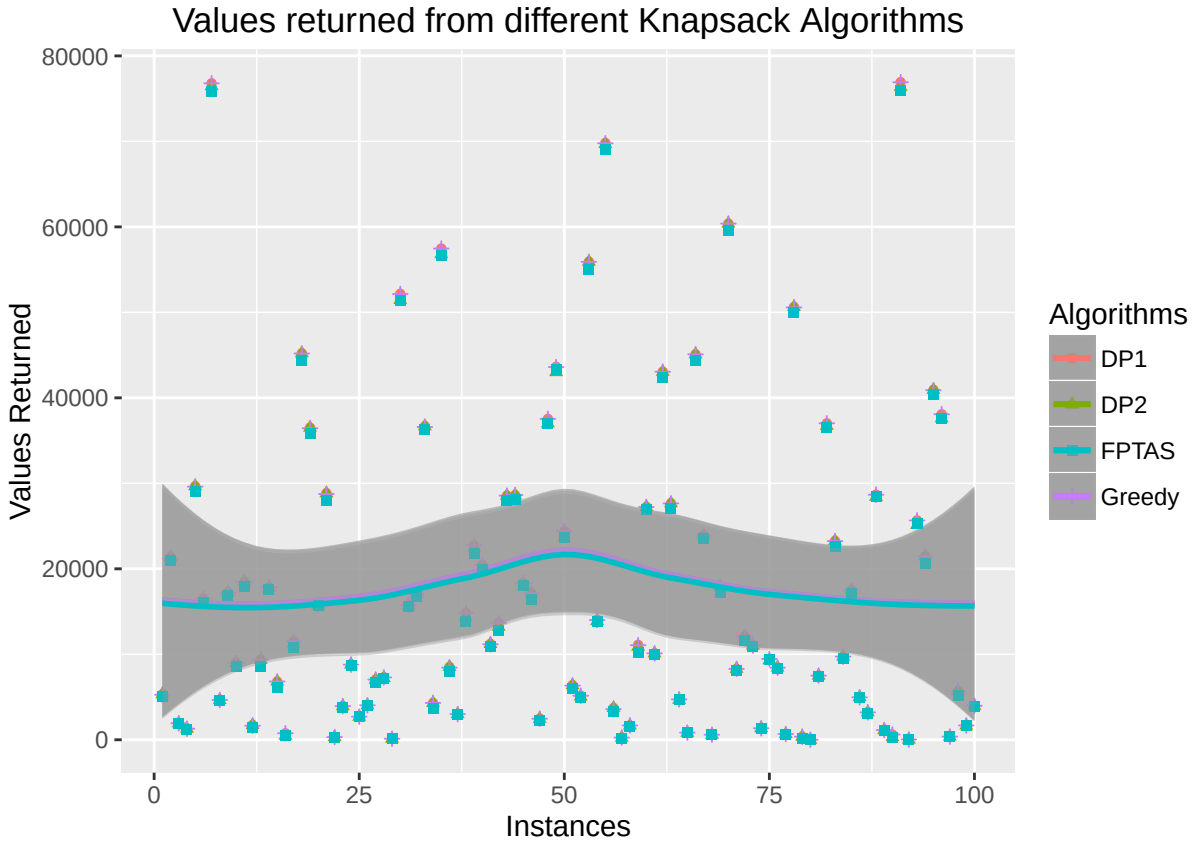
```
##    Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
##       0    3967   11120   17680   26050   76920
```

Values Returned from FPTAS

```
##    Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
##       0    3712   10860   17320   25700   75950
```

From the statistical summary, we see that DP1 has the highest mean while FPTAS has the lowest mean. However, the differences are small with the scale of the number that we see.

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.
##   0.038   4.808  24.780  53.620  60.530 449.000
```

DP2 Running Time

```
##     Min.  1st Qu.   Median     Mean  3rd Qu.     Max.
##    0.073   49.170  170.500  433.300  596.200 3063.000
```

Greedy Running Time

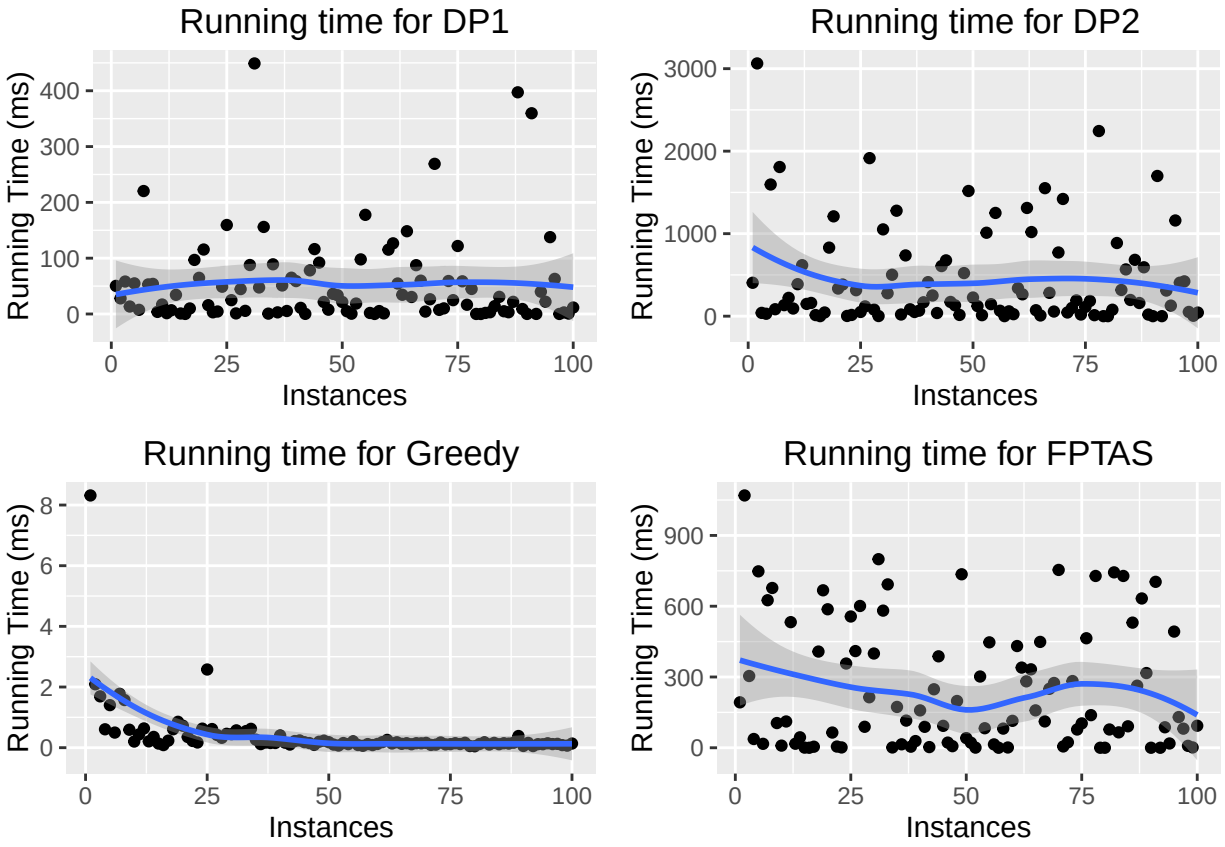
```
##    Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
##  0.0470  0.1178  0.1610  0.4094  0.3888  8.3160
```

FPTAS Running Time

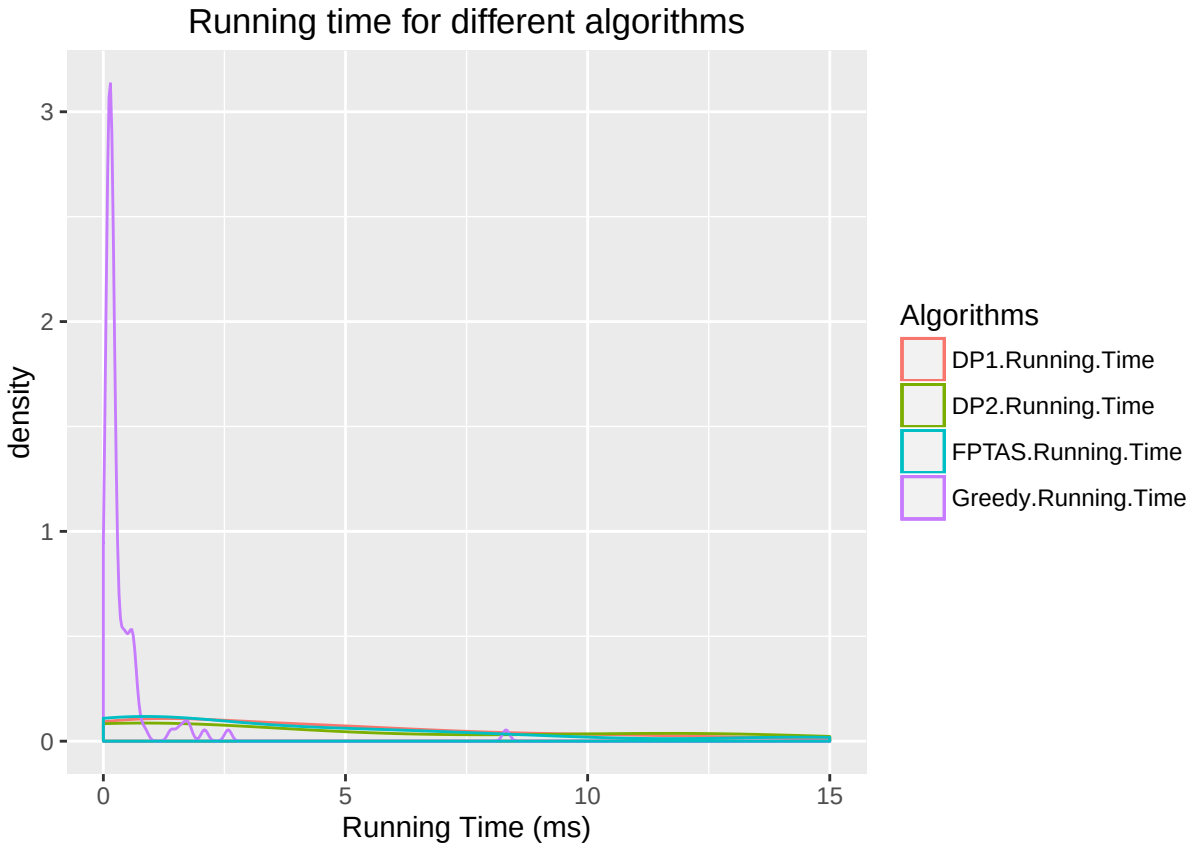
```
##     Min.  1st Qu.   Median     Mean  3rd Qu.     Max.
##    0.043   20.660  114.600  245.800  415.200 1069.000
```

As expected, Greedy has the lowest mean of running time, indicating that it runs the fatest out of all other algorithm. However, DP1 is not the slowest algorithm. Instead, the slowest was DP2. The mean of running time for FPTAS is roughly half of that of DP2.

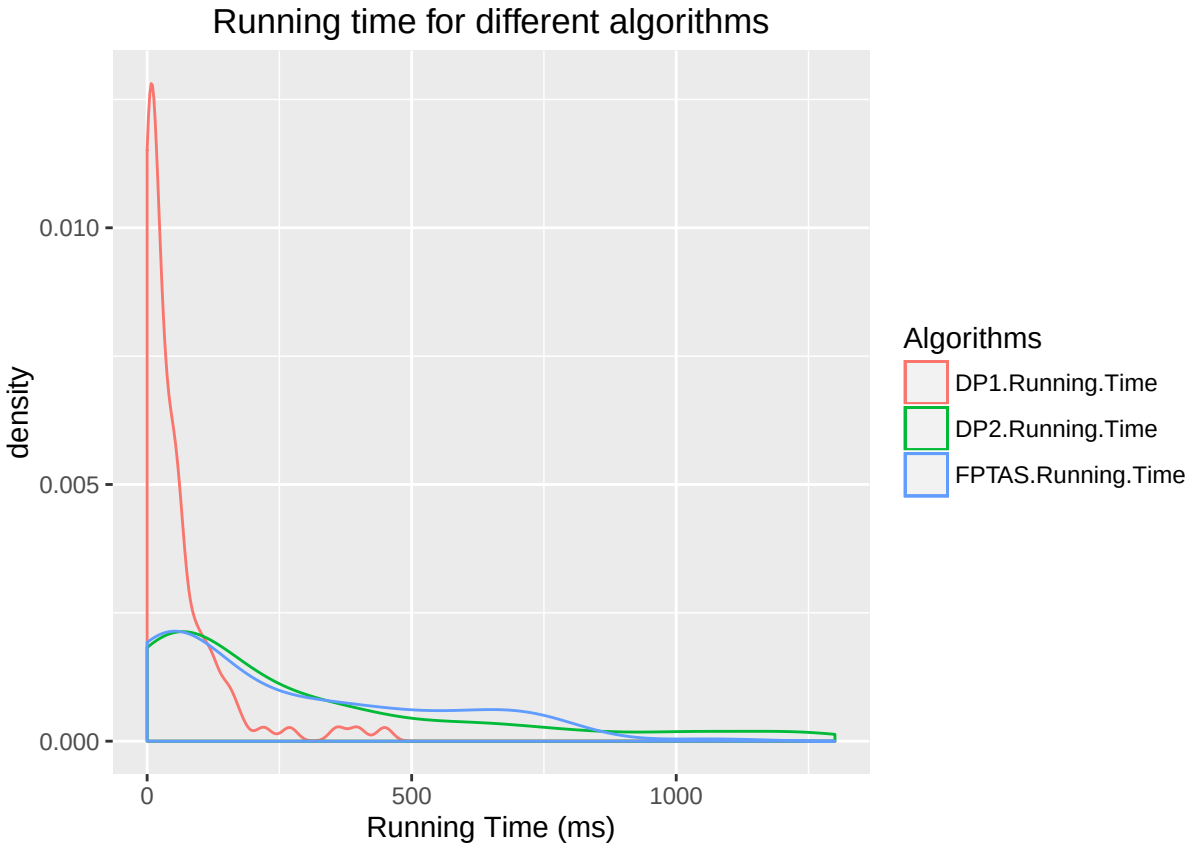
Below is the graphs of running time for different instances. There is again a smooth line to indicate the general trend of all the instances. Noticing the difference on the y-axis, which indicates the running time in ms, we can see that Greedy runs in the range below 10ms, while DP2 runs in the range 3000, or 3 seconds. FPTAS has a higher upper bound for running time compared to DP1. FPTAS and DP2 graphs are very similar in terms of trend line shape, 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 know that Greedy would have the best running time. It is, however, interesting to see that for the early instances, the running time of Greedy spike a lot compared to the rest of the instances while we hardly notice a difference in the graphs for DP1, DP2, and FPTAS. The lack of differences might be attributes to the scale on the y-axis, where a slight change in running time will stand out more in Greedy graphs compared to others.



Additionally, I also created a density graphs for the running time of the four algorithms with our 100 instances, which is shown below.



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. The rest of the algoirthms are hard to see so here are the densities for DP1, DP2 and FPTAS.



From this graph, we see that most of DP1 running time is in the lowest spectrum. The running time of DP2 and FPTAS is very similar to one another, though, DP2 has more instances running at more than 1000ms. From this, we can see that the running time is best with Greedy, followed by DP1, FPTAS and then DP2.

IV. Reduction For 3SAT

This analysis is focused on running time in the experiments involved 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.

Running Time

Below are the statiscal summary for the running time of each algorithms (everything is in ms).

DP1 Running Time

```
##    Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
##   85.75  139.10  180.60  227.50  237.00 4228.00
```

DP2 Running Time

```
##    Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
##    4188    6394    8058    8564    9738   28570
```

Greedy Running Time

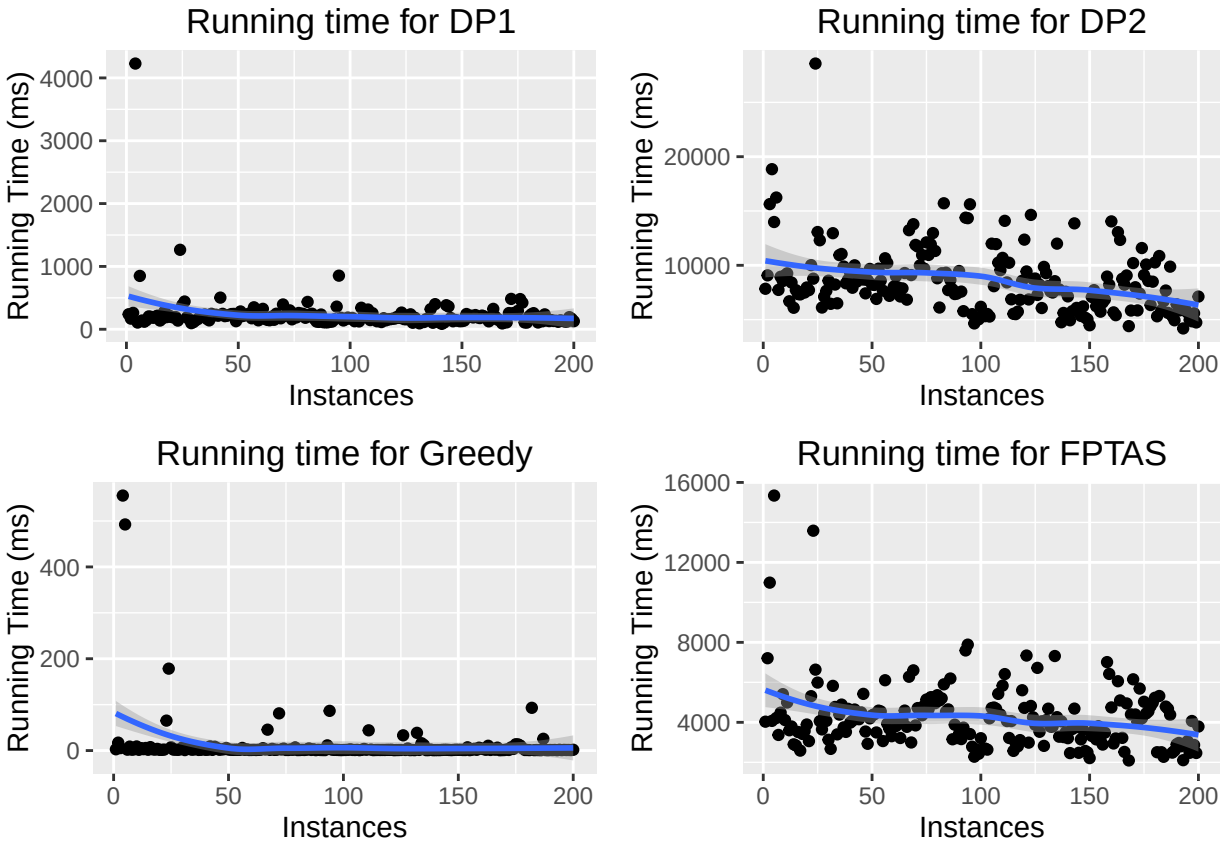
```
##    Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
##   0.809   1.274   1.699  11.330   3.731 555.400
```

FPTAS Running Time

```
##    Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
##    2092    3188    4040    4228    4742   15340
```

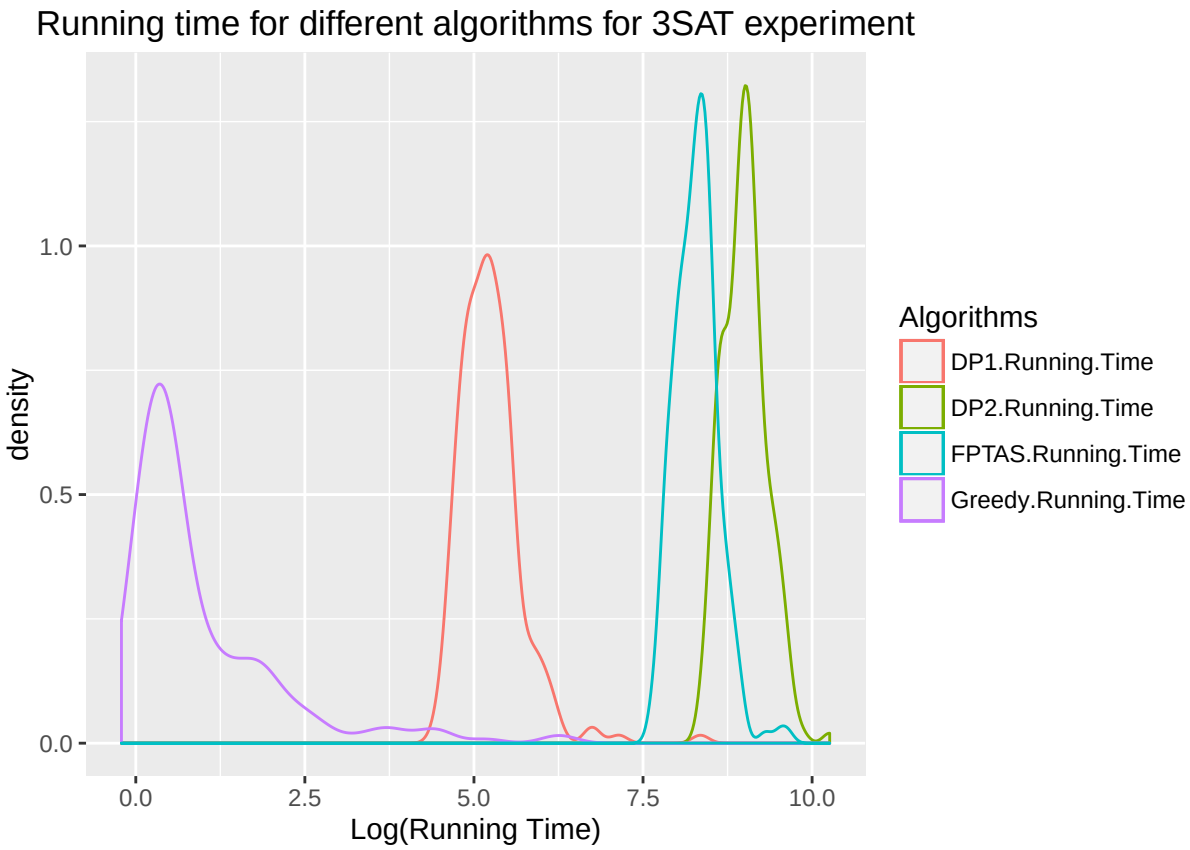
We see that Greed has the lowest mean of running time at 11.330 ms. DP2 and FPTAS still have the two highest running time as with the previous experiments.

Below are graphs of the running time of the instances with the trend lines in each graph.



Again, we see that the range of running time for Greedy is much slower than the rest of the algorithms with all instances running below 600ms. DP1 has the range of below 4500 ms.

Below is the density graphs for the running time of the four algorithms with our 200 instances. This graphs gives us information on how the running time is distributed among the testing instances. Since there is such a disparity between the distribution of running time between Greedy and DP2 and FPTAS, I decided to take the log scale of the running time of each instances.



Again, Greedy stays at the lowest left spectrum of the graph, followed by DP1, FPTAS and then DP2 respectively. This aligns with what we have discussed so far.

V. 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. Also, some languages might not restrict the number size the same way as Java. Apparently, implementing the reduction from 1in3SAT to Subset Sum would not face the same problem in Python due to the value of an integer is not restricted by the number of bits and can expand to the limit of the available memory (Rathi).

VI. Conclusion

The two experiments have shown us the potential differences in running time for different algorithms given the same problems. It is very interesting to see the numbers being quantified as shown in graphs and charts.

It was also very interesting to see theoretical problems translated to actual code and faces the limitations from the specific coding language. The fact that I have to think of way to go around the problem definitely taught me a lot.

VII. References

1. O'Connell, Thomas C. "What Is a Computer and What Can It Do?" Place of Publication London: College Publications, 2013. N. pag. Print.

2. Rathi, Abhay. "What Is the Maximum Possible Value of an Integer in Python ? - GeeksforGeeks." GeeksforGeeks. 2016. Web. 08 May 2016. http://www.geeksforgeeks.org/what-is-maximum-possible-value-of-an-integer-in-python/.