Rose Reiner
CS306
5/1/20

Final Project Report

This report will cover several algorithms, Maximum 0-1 Knapsack algorithms and SAT algorithms, and it will analyze the results found after each were run on the same large, random instances. Not only will this report cover the quality of the solutions, but also their runtimes and any observations made along the way. Towards the end of this report, I will go over some of the errors throughout the experiment and instead analyze what I have observed while running smaller instances on my semi-errored algorithms.

The Maximum 0-1 Knapsack algorithms consisted of the two dynamic programming algorithms, the one from CS305 O(nB) and O(n ^2*v(a_max)) Mincost version, the FPTAS algorithm, and the Greedy 2-approximation. To see the quality of solutions that each of these algorithms returned, I generated 100 random items that each had a value and a cost with a range from 0 to 1000. For my implementation purposes, two arrays were generated; one array that held 100 values of values and the other array held 100 cost values. I set the budget to be randomly generated 100 times as well and ran the algorithms exactly 100 times and compared the results. I'll state now that when 100 instances were passed to the Greedy algorithm, it did not work well and I got an index out of bounds exception. I'll return to the Greedy algorithm later.

The results for the rest of the Maximum 0-1 Knapsack algorithms came out as expected. Looking first at the quality of solutions, both the CS305 algorithm and the Mincost algorithm had an average of returning an optimal value of 1,183. This was not surprising because they are both solving the knapsack problem and will return the optimal solution. On the other hand,

FPTAS is an approximation scheme algorithm and it is supposed to return a solution that is at least $(1-\varepsilon)OPT$. I chose $\varepsilon=0.1$. FPTAS returned an average value of 1,193. This approximation scheme algorithm did better than the optimal, which is what it is supposed to do: return a solution that is at least $(1-\varepsilon)OPT$. The median optimal value was 1,097 for the CS305 and Mincost algorithms, and 1,107 for the FPTAS algorithm. All algorithms had a maximum value of 3826 and a minimum value of 0.

As for the runtime solutions, the results were also how I expected. Please note, I ran it with *System.nanoTime()* because *System.currentTimeMillis()* didn't give enough precision for how fast the CS305 algorithm was running. Then I divided that number by 1,000 to get a microsecond(μs). The average runtime for the CS305 algorithm was 16,322 microseconds, the average runtime for MinCost was 36,500 microseconds, and 37,738 microseconds for FPTAS. The median values were 15,849μs for the CS305 algorithm, 35,572μs for the MinCost algorithm, and 35,933μs for the FPTAS algorithm. For the 100 runs, the highest runtime for the CS305 algorithm was 29,904μs, while Mincost's highest runtime was 48,723μs, and FPTAS was 54,241μs.The runtimes for these algorithms makes sense because there were less base cases to handle for the CS305 algorithm than for Mincost, which would also affect FPTAS to be slower. In addition, FPTAS has to do a few more calculations in the beginning. Thus, it's runtime to Mincost is a little worse, but almost exactly the same. Small calculations wouldn't cause the runtime to change dramatically, but I think mine changed more because I implemented it with for loops iterating over an array in order to scale every value and make the calculations.

There were a few observations I noticed while running the 0-1 Maximum Knapsack algorithms. At first, I ran the "one-hundred rounds" experiment a few times and noticed that the

average values returned for the dynamic programming knapsack algorithms were always exactly matched, except for some rounds they came out slightly different. However, I could not have observed this without individually going through each of the rounds conclusions. Instead, the average number for them was a bit different sometimes. On average, they had about a 2% difference between each other with the MinCost algorithm always being slightly better. I thought this was interesting because they both solve the knapsack problem, and to what I've thought, they are both supposed to get the optimal values. Secondly, I noticed that a higher budget resulted in a higher optimal value. I believe this is the case because there are more options of which items to take or leave when there is a bigger budget.

I was disappointed when the Greedy algorithm couldn't handle a large amount of values to test in comparison to the other 0-1 Maximum Knapsack algorithms. Instead, I decided to run a smaller test case against the CS305 algorithm and compare its results. I was able to get it to run 10 rounds successfully without it crashing. As seen previously, the CS305 algorithm got the optimal solution and the lowest runtime out of the other two algorithms. When comparing the results of Greedy to the CS305 algorithm, I noticed they seemed very off with each other for multiple rounds. On average, they got around the same results. Greedy got an average value of 1,034, while the CS305 algorithm got an average of 1,073. However, when looking at each individual round, sometimes the Greedy algorithm's numbers were way off when the budget got large. It was noticeable that the Greedy algorithm did well when the budget was low. For instance, a budget of 12 got greedy to return a value of 992, while the CS305 algorithm returned 0, and it's runtime was significantly higher than Greedy's for getting a value of nothing. Whereas, when the budget was higher at 82, Greedy returned a value of 1,055, while the CS305

algorithm returned a higher optimal value of 1,606; the runtimes were still around the same, but this time the Greedy algorithm did worse. Unfortunately, I couldn't run the Greedy algorithm among the others in a large experiment, but I would assume that the same observations for all of them would still hold, and I believe Greedy would ultimately do worse especially than FPTAS if the budget were high.

Running the same experiments on the Sat algorithms also gave similar results as I was expecting. While doing this experiment, my DPLL algorithm also had an index error so I will not be including it in the comparison with the other Sat algorithms. I generated 50 random clauses, with 3 literals in each clause, and five different literals all together. Similarly to the Maximum 0-1 Knapsack algorithms, I ran these two algorithms 100 times and generated 100 new random clauses for every run I did. Based off of the ⅞ -approximation algorithm, the $E[y] = 43.75$ clauses satisfied for 50 clauses. I noticed that GSAT returned on average 42 clauses satisfied, while MAX3SAT returned an average of 40 clauses satisfied. The highest number of clauses satisfied for the 100 runs were 47 clauses from GSAT and 46 from MAX3SAT. The lowest number of clauses satisfied during the 100 runs were 39 clauses satisfied by GSAT and only 31 satisfied from MAX3SAT. The average runtimes, this time in nanoseconds, were 1441ns for GSAT and 511ns for MAX3SAT. GSAT also had a median time of 466,752ns, it's maximum time was 2,008,402ns, and it's minimum time was 139,102ns. MAX3SAT had a median time of 186,348ns, it's highest runtime was 3,795,900ns, and it's lowest runtime was 50,798ns. Based on these results, GSAT would get closer to the expected clauses and would get higher than that of MAX3SAT. On average, it's runtime was longer. This makes sense because GSAT does multiple checks on different truth assignments to configure which satisfies the most clauses.

Some problems I had during the experiments was my GSAT and DPLL not working correctly once given a larger assignment. However, they did work on my smaller test cases. I think the trouble with DPLL is I implemented the formula as an ArrayList of ArrayLists of Literals, so when an ArrayList is removed, or known as a clause, from a higher position in the outer ArrayList, the index of the for loop will be at the size of the ArrayList because a clause was previously removed and that case for the index was not handled. Another issue I had when running these experiments was making the epsilon for FPTAS too big. I first set it to $\varepsilon = 0.5$, but the numbers seemed too off from the other Maximum 0-1 Knapsack algorithms. I changed it to be lower at $\varepsilon = 0.1$ error tolerance. When the change was made, the FPTAS numbers increased way higher. They looked more similar to the results of the other algorithms, and even doing better than them.

Overall, this experiment was really interesting. I felt that I understood the algorithms better than I did before. Running large experiments on them and comparing the values was really fun because I got to see how they actually worked in a larger scenario as well as observing which ones gave the optimal value, and which had the best and worst running times on average. If I had more time, I wish I could have fixed my index of bounds errors for GSAT and DPLL so I could see them run on larger instances in comparison to the other algorithms. Ultimately, this was a very rewarding project and it was exciting to look at the results of each algorithm and see if they behaved how they were expected to.