Computability and Complexity

Final Project Report

Sten Leinasaar

Date: 6th of May, 2022

Introduction

I generated 100 different instance of size 500 for each knapsack problem. However due to the unfortunate error I overwrote my graphs. I was too lazy to run instance of 500 for 100 times again, so I did it for instance size of 300. Therefore, the knapsack graphs are for the instances of size 300, whilst the numeric data is for instance size 500. It offers a unique view when looking at data as I can see the runtime on the graph and I have the runtime for the bigger instance size. I ran the test methods for each instance in the same file that contained all the algorithms (*runKnapSackTest.py*).

For SAT algorithms I generated instances with 700 clauses. Each instance will have 16 literals. Looking back that seems pointless because there are only certain number of different clauses, I can create with 16 literals. Therefore, to satisfy all 700 with 16 literals is a very unlikely chance. However, it offers a unique view on what is the maximum number of clauses out of 100 test instances that I could satisfy. I ran the test methods for each instance in the same file that contained all the algorithms (runSATTests.py).

The goal of this final project was to compare the results, running times, and overall complexity of four different knapsack algorithms and 3 different SAT algorithms. There was also Travelling Salesman Algorithms to be tested, but I decided not to do them due to the big amount of work I have to finish for other courses. I apologize. I did however begin to write out the algorithms. I included them with my zip file.

KNAPSACK

Setup and organization.

Knapsack problems were created in *GenerateProblems.py* file. The code for it can be seen below. The size of each problem is 501. I used that size for the first test and that is where the data is from. However, I accidentally overwrote my graphs by running a test for fun with size 20. Therefore, to get the graphs, I ran the tests with knapsack problems of size 300. **Important to note** that numeric table data is from the testing with knapsack problem of size 500, but graph is for knapsack problem of size 300.

```
def generateKnapSack():

    instance = []
    for x in range(501):
    #Items value will be between 1 and 100
        value = random.randint(1, 100)
        #Item's cost will be between 1 and 75.
        cost = random.randint(1, 75)
        item = (value, cost)
        instance.append(item)
    printKnapsackToFIle(instance)

    return instance
```

This method generates mere tuples of value and cost. The question arises how I am storing and accessing it when running the test. I wrote a separate class called KnapSackItem.py. That allowed me to store an item that has a value, cost, and ratio associated with it, which made my testing little bit easier and more comfortable. Note that my first step in this project was to figure out and design how I am storing my knapsack items as this determine my approach with algorithms.

```
def generateKnapSack():
    forTesting = []
    knapSack = []
    for x in range(101):
        #print("test generation:", x)
        instance = GenerateProblems.generateKnapSack()
        #print("Instance generated is: " , instance)

    #value is the 0 index, cost is 1
        #print("For testing variable before I start appending.", forTesting)
        for tuple in instance:
            # print("Value in tuple", tuple[0], tuple[1])
            item = KnapSackItem.KnapSackItem(tuple[0], tuple[1])
            knapSack.append(item)
            # print("Item appended is with value of:" , item.getValue())
            # print("item is with the cost of: ", item.getCost())
        #print("For testing before appending knapsacl", forTesting)
        forTesting.append(knapSack)
        knapSack = []
    #print(forTesting)
```

In the test class after receiving the instances with tuples inside. I go through it tuple by tuple and I generate items which I store in a list called knapsack. That list is then stored in forTesting list, which is a list of lists. This was all of the instances in one big list.

Here is a class overview of my knapsack item.

```
class KnapSackItem:

    def __init__(self, value, cost):
        self.value = value
        self.cost = cost
        self.ratio = round(value/cost,3)

    def getValue(self):
        return self.value

    def changeValue(self, newValue):
        self.value = newValue

    def getCost(self):
        return self.cost

    def getRatio(self):
        return self.ratio
    def setValue(self, value):
        self.ratio = round(value/self.cost, 3)
```

The constructor takes in the value and the cost. Ratio is immediately calculated and rounded to 3 decimal points.

There are get methods for ratio, value, and cost. For value I have accidentally left 2 methods to set it, but that is ok. Little refactoring doesn't hurt anybody.

Algorithms and their analysis

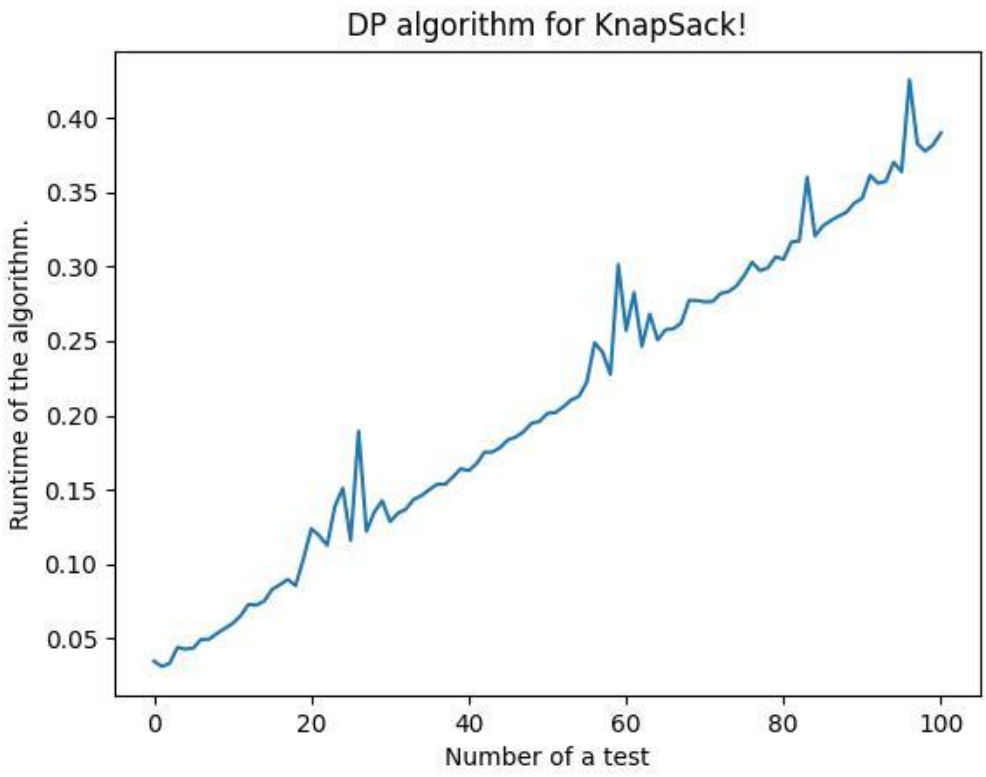
After running all the tests, I generated a report for each test (See AppendixA for raw data). I compiled from that report a general table that shows minimum, maximum, median, and average runtime for each algorithm. Remember, this is for 100 tests, each being of size 500 items. As we can note, the maximum knapsack algorithm takes the longest. On average, one instance takes 29 seconds, which for 100 instances is almost 3000 seconds. That is 50 minutes. Similar runtime can be seen for FPTAS that uses maximum knapsack algorithm after scaling the values of the items. The runtime decreases a little, but not significantly. Modified greedy algorithm that uses ratio as an approximation, has a very low running time, but gets, on average, close to 30% from the optimal solution. Rest of the algorithms get close to 100% optimal but have incredibly large runtime.

Figure 1. Descriptive data for all Knapsack algorithms

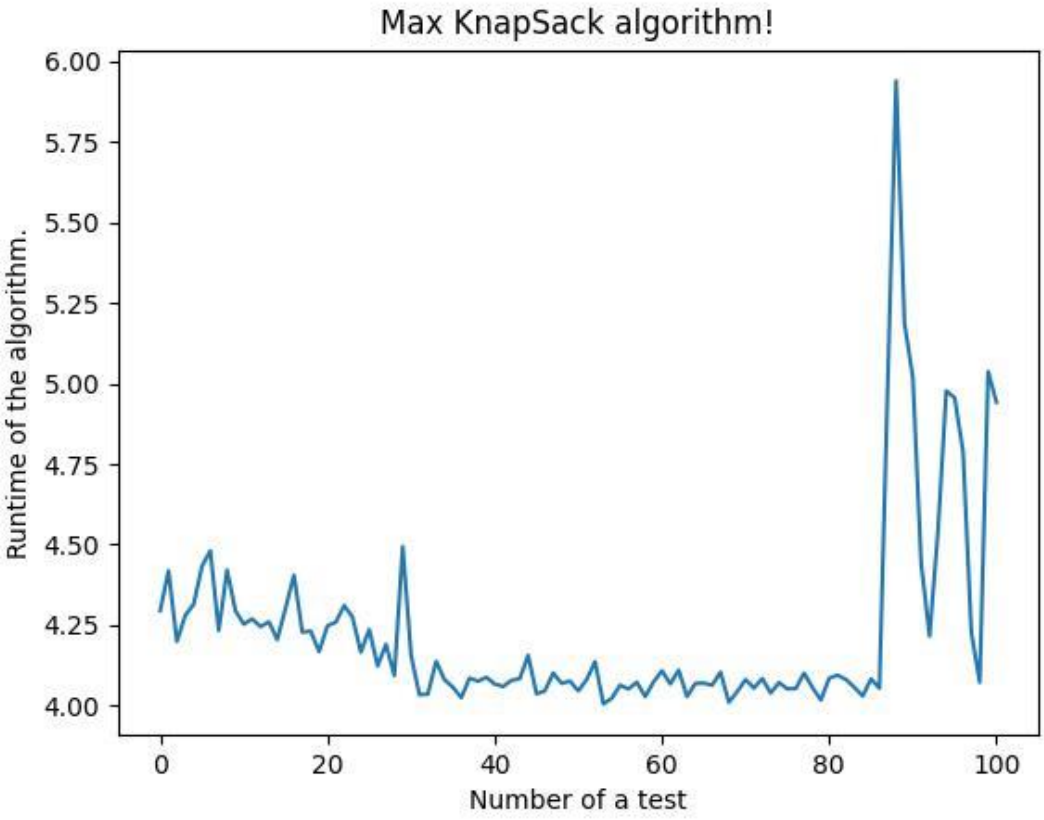
	Dynamic Programming	Maximum Knapsack	Modified Greedy	FPTAS
Maximum Runtime (Seconds)	1.5	33.345	0.046	33.009
Minimum Runtime (Seconds)	0.068	26.139	0.012	25.592
Median runtime (Seconds)	0.572	28.808	0.017	27.946
Average runtime (Seconds)	0.617	28.989	0.018	27.955
From optimal solution	100.00%	99.80%	26.54%	99.80%

I generated a graph with matplotlib for each algorithm. Let' s look at those as well. Each graph has an x axis to be a number of tests conducted and y axis to be the runtime of each

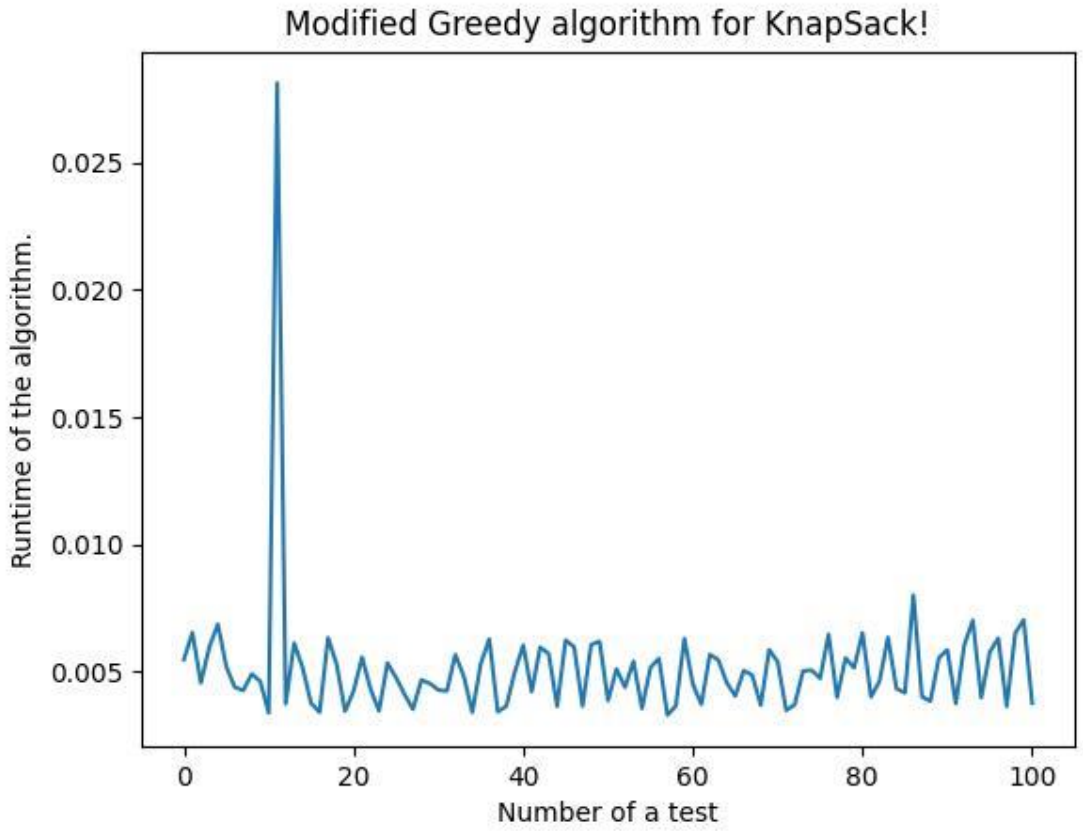
test. Note that because of my stupid mistake, I don't have graphs done for size 500 test instance, they are instead of size 300.



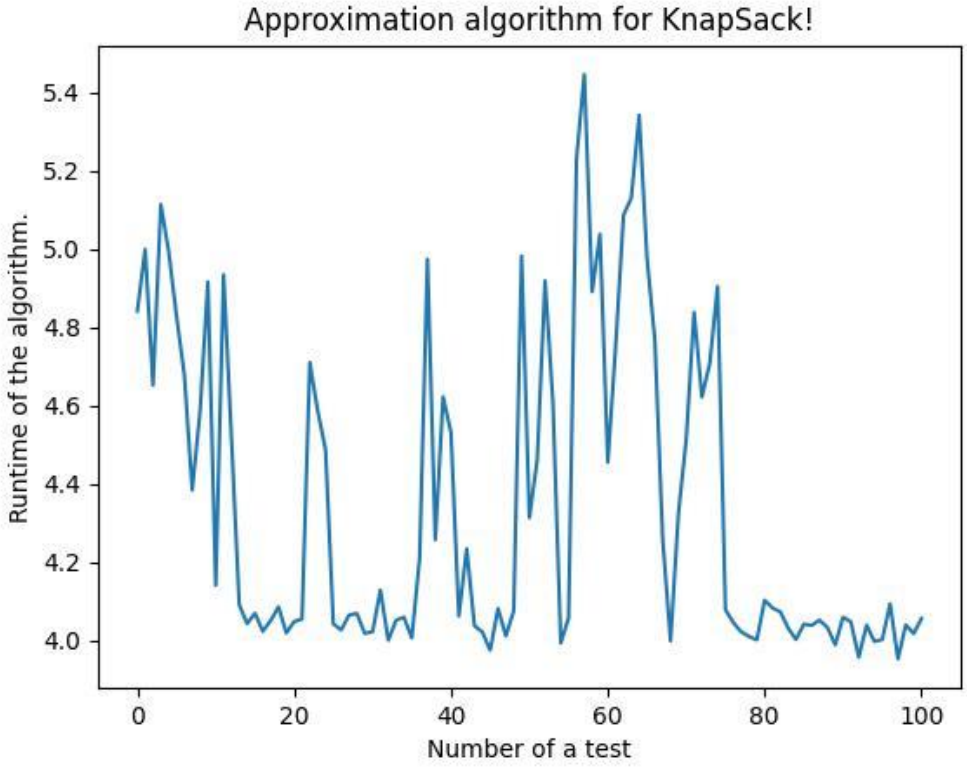
The dynamic programming algorithm shows the runtime graph to be rather consistent. Yes, there are some instances it runs little higher, but the bigger the size of the instance, the more linear the model is. For 500 instances—before I accidentally overwrote it—the graph was much more linear as can be observed here. With each test, the weight being passed with the algorithm was increased. The change can be noted from the raw data.



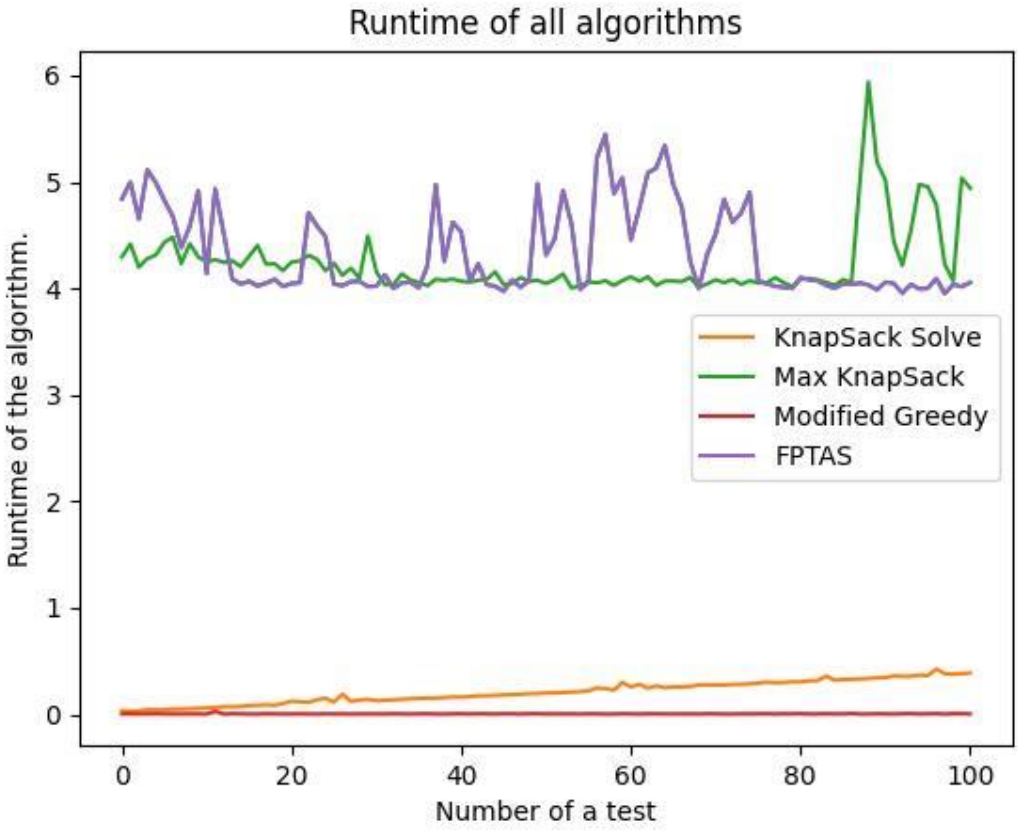
The MaxKnapSack algorithm's runtime surprised me. There was a sudden uptick in runtime after 85th test. It is important to note that for each test, the weight/budget/target being passed increased. Therefore, this graph points us towards the critical line where target value is too high, so runtime is increasing considerably. It is also fascinating to consider how with size 500 for each instance, the average runtime was proximately 29 seconds. The increase compared to average runtime of around 5 seconds is insane.



The runtime graph for modified greedy algorithm shows a lot of consistency in terms of the runtime. There is a sudden uptick with the runtime in the beginning of testing, which was surprising to me. I considered the runtime to be almost the same each time or with very little change, but that considerable momentary rise in runtime was surprising. I cannot think of any reason that would happen for a greedy algorithm that follows the same procedure.



Fully polynomial approximation algorithm that can be seen here, is seemingly all around the place with the runtimes. However, we can note that after 80th test with the larger target, the runtime begins to standardize. I considered running a test with incredibly big target and size 1000 items for each test. I predict the runtime of the algorithm would flatten out because it is fully polynomial time approximation.



Here we can see all the runtimes for each algorithm on the same graph. Notice how modified greedy is almost completely flat. Look at the max knapsack and FPTAS. The change between two around 80th test is contrasting. The scaling in FPTAS is crucial to decrease runtime when numbers get large, however maxKnapSack is better for smaller numbers as it has more consistent runtime before. Dynamic programming approach is following a expected trajectory.

SAT algorithms

Setup

SAT problems were created in *GenerateProblems.py* file. The code for it can be seen below. The size of each problem is 701. I am enforcing that no clause can have repeated literals. That is, each literal must appear in that clause only once. This was done as follows:

```
def generateTestSAT():
    instance = []
    for x in range(701):
        literal = random.randint(0,15)
        if(random.randint(0,11) <= 5):
            #Positive
            literal = literal
        else:
            literal = -literal

        literalSecond = random.randint(0,15)
        while True:
            if literalSecond == abs(literal):
                literalSecond = random.randint(0,15)
            else:
                break

        if(random.randint(0,11) <= 5):
            #Positive
            literalSecond = literalSecond
        else:
            literalSecond = -literalSecond
```

The first literal is simply created and randomly assigned to true or false. The second literal is checked to not be equal to the first literal. Note that randInt returns only positive numbers. That made the checking easier. I could have perhaps randomly generate the numbers between -15 and 15. The code illustrates how the checking was done so no two literals are the same in one clause. The SAT formula was stored in the list of lists, where each list inside the list refers to a clause. Number refers to the index of that literal such as 1 referring to x1, and 2 referring to x2. And -1 referring to -x1 and -2 referring to -x2. Finally, the SAT formula looked something like this:

[[1,2,3], [-1, -4,6], [2,4, -5]]

For the sake of time, I won't write out all 700 clauses. I'm sure you get the point. The tests were called from the file runSATTests.py in a same manner as knapsack tests were called. Reports were generated by the class ReportGenerator.py. All the code for the report generator can be seen under UTILS folder. I am using matplotlib to generate the graphs.

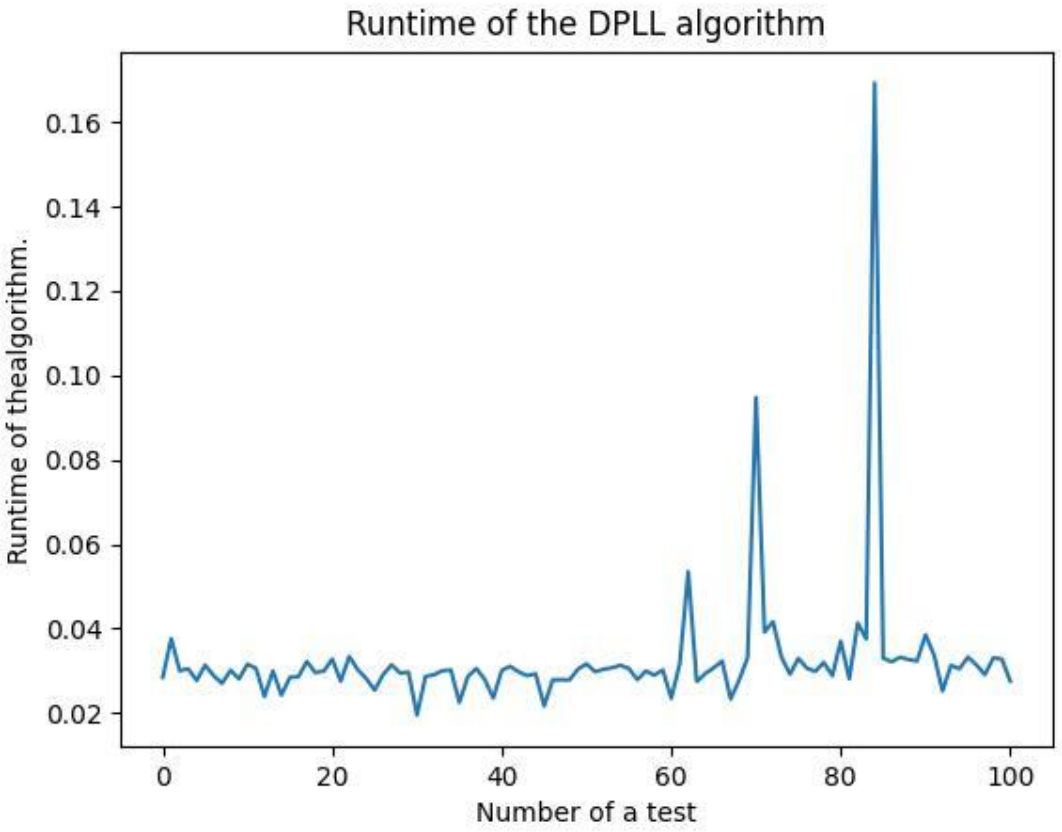
Analysis of the Algorithms

After running all the tests, I generated a report for each test (See AppendixB for raw data). I compiled from that report a general table that shows minimum, maximum, median, and average runtime for each algorithm. Remember, this is for 100 tests, each being of size 700 items. I found that my DPLL algorithm is wrong as it should produce the most optimal solution, but it seems like my GSAT and 7/8 approximation are producing way better solutions. The running time for DPLL however is very low, that prompted be to think that something is wrong. It shouldn't be that short time for that large instance. GSAT takes up to 51 seconds for the largest instance, whilst 7/8 approximation is quick. I know something is off with DPLL because 7/8 approximation takes longer.

Figure 2. Descriptive data for all SAT algorithms

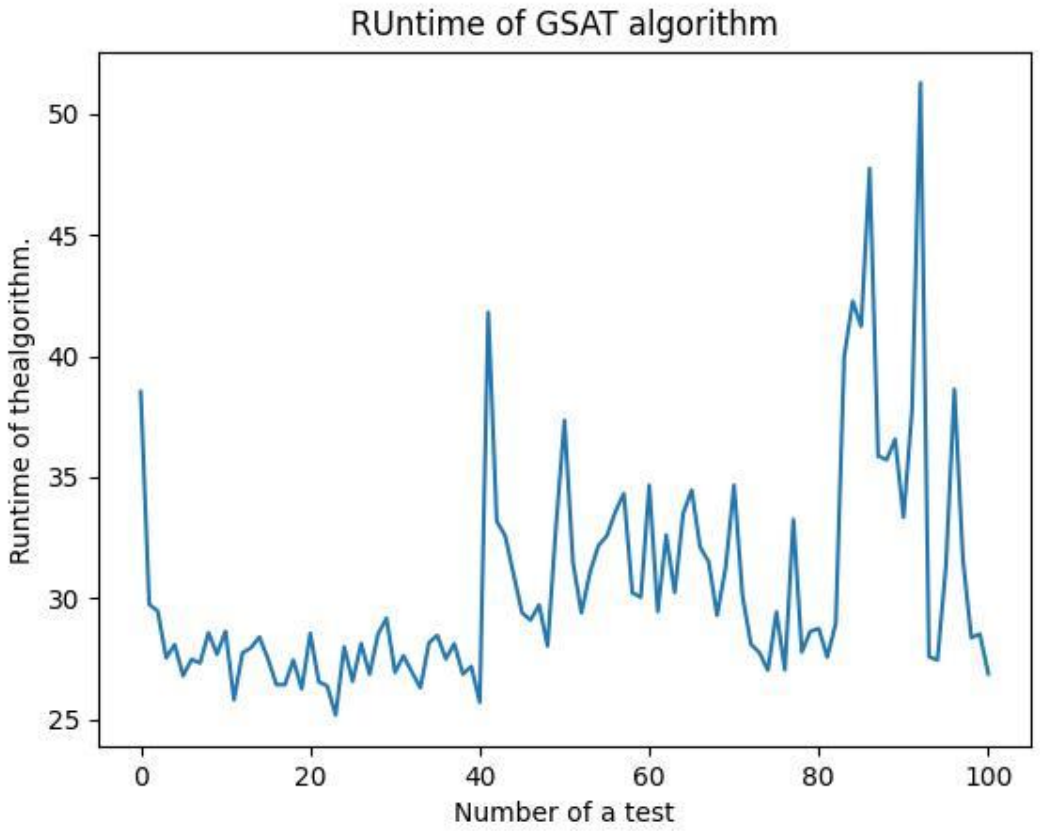
	DPLL	GSAT	7/8 Approximation
Maximum Runtime (Seconds)	0.169	51.288	0.696
Minimum Runtime (Seconds)	0.02	25.166	0.318
Median runtime (Seconds)	0.03	28.739	0.359
Average runtime (Seconds)	0.032	30.566	0.366
From optimal solution	100.00%	150.36%	152.05%

DPLL



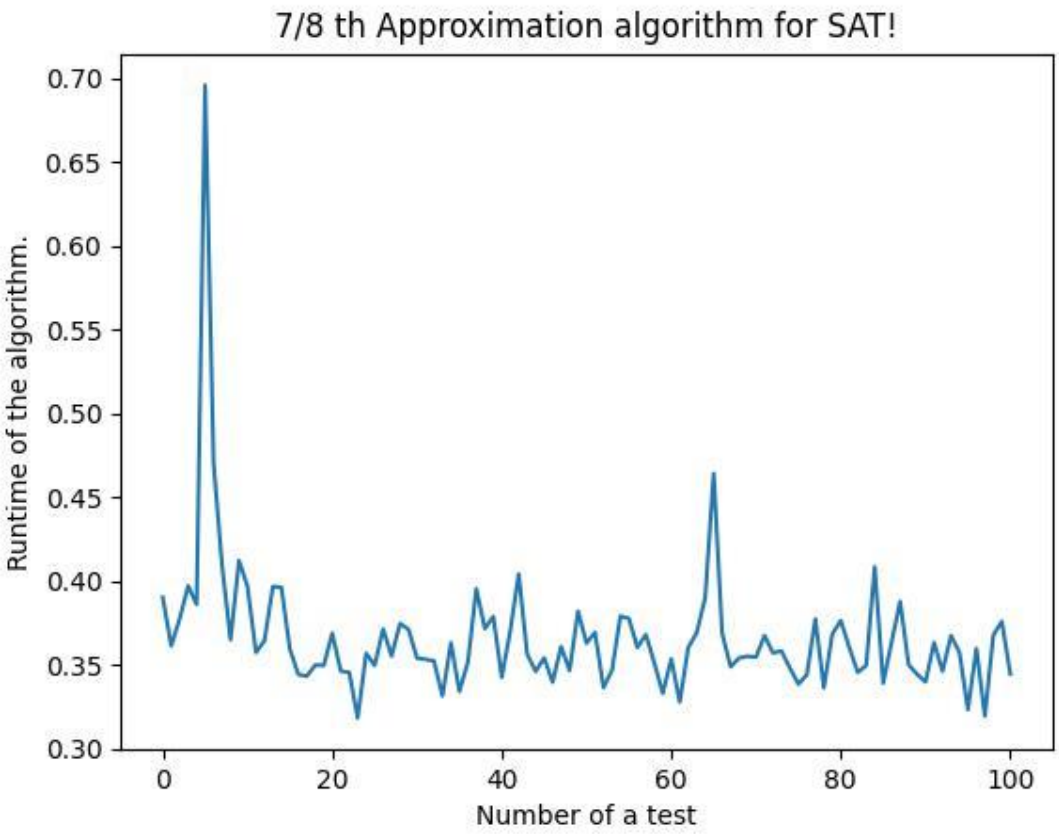
The runtime for DPLL is quite consistent, but we must consider that there is something wrong with it. I am not handling the recursion correctly, I assume. At one point in my life I shall sit down and figure it out, but that will not be this semester. Or if you have some feedback, please let me know. I thought I had it, but I didn't.

GSAT

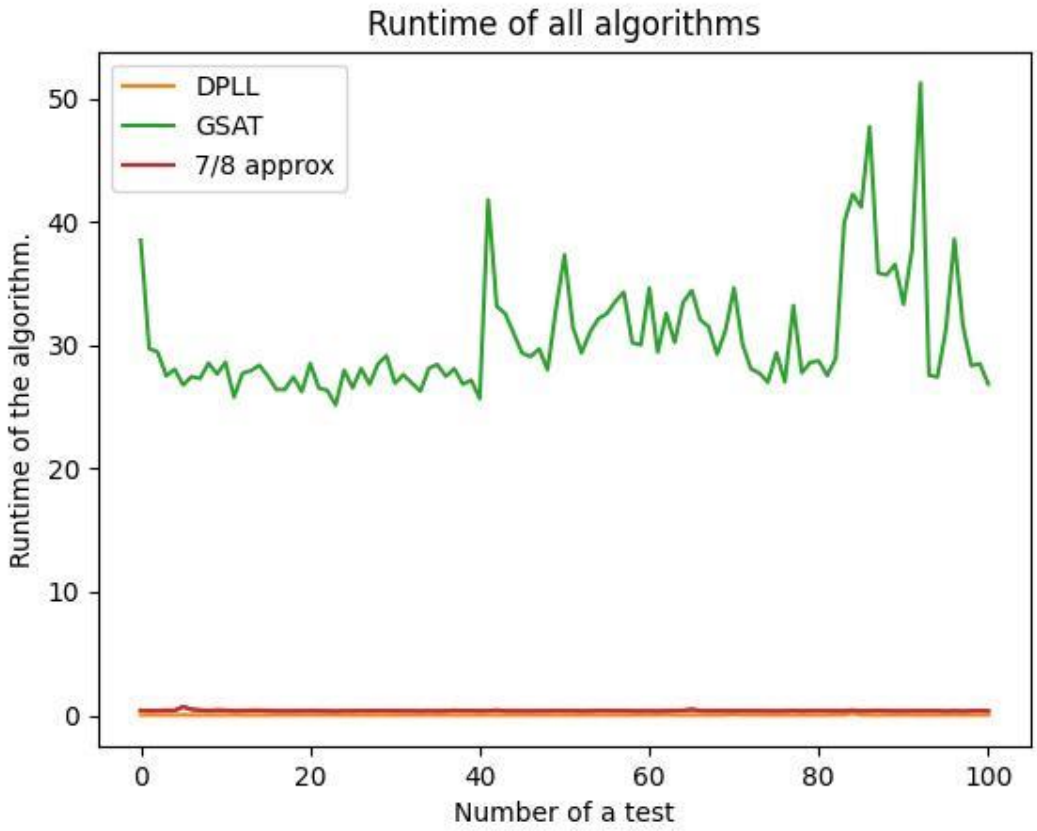


We can see that the runtime of GSAT stays pretty much the same. That is because I didn't really change the values for maxflip and maxtries. I kept maxtries at 6 and maxflips at 10. Each instance was the same size. I assume the runtime difference comes from how long it takes to check if formula satisfies because during that time the clauses are being removed. Therefore, if more clauses are removed in first two iterations, then further iterations will take less time and true or false is returned quicker. However, if more iterations have to go through, then it takes longer.

Seven Eight Apporximation



Seven eight approximation is very simplistic algorithm. And that can be seen with the consistent runtime across all the tests. There is a sudden uprise in the beginning, but I think it was due to the complexity of the formula. Because across all tests, that specific test seemed to cause the uprise. It is exciting to see the consistency with the runtime.



This graph shows the runtime for all the algorithms. Due to the GSAT the scale is off and it looks like DPLL and approximation are almost exactly the same runtime. GSAT seems to be very dependent on the complexity of the SAT formula and how many clauses could be removed for one literal.

Conclusion

What I learned from this project is not rushing. I ended up having a lot of small bugs in my code because I wasn't ensuring my understanding before writing code. Once I talked out the GSAT problem, I truly realized how the heuristic part works and what is essentially the point of the algorithm. It was fascinating to look at the runtimes of the algorithms and see first-hand how the runtime is affected by size of the problem and some cases, the complexity of the problem. Especially truth assignments and knapsack items with various values and costs. I can imagine how algorithms we implemented work and function. All in all, I enjoyed working on

them and running the tests. Given it took time and I decided to learn a completely new language for myself, it was still fun. And I now know python way better than before.

Once again, I apologize I won't be finishing testing for TSP, but I truly have way too much happening and I need to find the balance. So I decided to leave the code I did start to write for TSP in my zipped files, but not make an effort to finish testing and write analysis on it.

APPENDIX A

	Dynamic Programming	Maximum Knapsack		Modified Greedy		FPTAS	
Test 1	24217	24204	0.999	1844	0.076	24204	0.999
Test 2	25377	25369	1.000	2283	0.090	25369	1.000
Test 3	24334	24294	0.998	2141	0.088	24294	0.998
Test 4	26257	26220	0.999	2967	0.113	26220	0.999
Test 5	26464	26369	0.996	2941	0.111	26369	0.996
Test 6	24457	24393	0.997	2727	0.112	24393	0.997
Test 7	25199	25101	0.996	3382	0.134	25101	0.996
Test 8	24739	24656	0.997	2594	0.105	24656	0.997
Test 9	24431	24426	1.000	3524	0.144	24426	1.000
Test 10	23307	23208	0.996	3056	0.131	23208	0.996
Test 11	25523	25501	0.999	3766	0.148	25501	0.999
Test 12	23870	23807	0.997	3527	0.148	23807	0.997
Test 13	25086	25062	0.999	3456	0.138	25062	0.999
Test 14	24961	24932	0.999	3564	0.143	24932	0.999
Test 15	25823	25745	0.997	4404	0.171	25745	0.997
Test 16	25552	25523	0.999	4691	0.184	25523	0.999
Test 17	25928	25883	0.998	5592	0.216	25883	0.998
Test 18	25892	25829	0.998	4635	0.179	25829	0.998
Test 19	24633	24616	0.999	4837	0.196	24616	0.999
Test 20	25226	25188	0.998	4395	0.174	25188	0.998
Test 21	25134	25083	0.998	4858	0.193	25083	0.998
Test 22	25889	25796	0.996	5453	0.211	25796	0.996
Test 23	25970	25930	0.998	5180	0.199	25930	0.998
Test 24	26817	26803	0.999	4796	0.179	26803	0.999
Test 25	23837	23738	0.996	5306	0.223	23738	0.996
Test 26	24408	24406	1.000	5215	0.214	24406	1.000
Test 27	25289	25285	1.000	5348	0.211	25285	1.000

Test 28	26230	26130	0.996	5452	0.208	26130	0.996
Test 29	25744	25734	1.000	5095	0.198	25734	1.000
Test 30	25849	25790	0.998	5457	0.211	25790	0.998
Test 31	25164	25161	1.000	5997	0.238	25161	1.000
Test 32	25332	25295	0.999	5712	0.225	25295	0.999
Test 33	25747	25650	0.996	5133	0.199	25650	0.996
Test 34	25767	25736	0.999	5932	0.230	25736	0.999
Test 35	25754	25674	0.997	5706	0.222	25674	0.997
Test 36	25134	25109	0.999	5568	0.222	25109	0.999
Test 37	26700	26617	0.997	6414	0.240	26617	0.997
Test 38	24935	24857	0.997	6813	0.273	24857	0.997
Test 39	25431	25353	0.997	6007	0.236	25353	0.997
Test 40	25123	25077	0.998	6331	0.252	25077	0.998
Test 41	25358	25331	0.999	6732	0.265	25331	0.999
Test 42	24336	24277	0.998	5845	0.240	24277	0.998
Test 43	25835	25789	0.998	7026	0.272	25789	0.998
Test 44	26063	26013	0.998	6768	0.260	26013	0.998
Test 45	25609	25571	0.999	6170	0.241	25571	0.999
Test 46	25157	25152	1.000	6875	0.273	25152	1.000
Test 47	25275	25219	0.998	6706	0.265	25219	0.998
Test 48	26136	26106	0.999	6836	0.262	26106	0.999
Test 49	24994	24982	1.000	6628	0.265	24982	1.000
Test 50	24756	24685	0.997	6743	0.272	24685	0.997
Test 51	26075	26019	0.998	6857	0.263	26019	0.998
Test 52	25095	25091	1.000	7189	0.286	25091	1.000
Test 53	24880	24854	0.999	6825	0.274	24854	0.999
Test 54	25783	25711	0.997	6756	0.262	25711	0.997
Test 55	25158	25093	0.997	7452	0.296	25093	0.997
Test 56	25054	24968	0.997	7432	0.297	24968	0.997
Test 57	24371	24369	1.000	6942	0.285	24369	1.000

Test 58	25674	25636	0.999	7797	0.304	25636	0.999
Test 59	26434	26335	0.996	7081	0.268	26335	0.996
Test 60	25812	25721	0.996	7490	0.290	25721	0.996
Test 61	25047	25013	0.999	7343	0.293	25013	0.999
Test 62	25852	25849	1.000	7983	0.309	25849	1.000
Test 63	25740	25702	0.999	8206	0.319	25702	0.999
Test 64	25452	25382	0.997	7965	0.313	25382	0.997
Test 65	25832	25787	0.998	8332	0.323	25787	0.998
Test 66	24267	24248	0.999	7869	0.324	24248	0.999
Test 67	25111	25019	0.996	8840	0.352	25019	0.996
Test 68	24539	24534	1.000	8480	0.346	24534	1.000
Test 69	24788	24709	0.997	7713	0.311	24709	0.997
Test 70	25816	25784	0.999	8652	0.335	25784	0.999
Test 71	25405	25333	0.997	8213	0.323	25333	0.997
Test 72	25350	25312	0.999	8065	0.318	25312	0.999
Test 73	24763	24758	1.000	7828	0.316	24758	1.000
Test 74	24017	23936	0.997	7647	0.318	23936	0.997
Test 75	24744	24692	0.998	8199	0.331	24692	0.998
Test 76	24469	24397	0.997	8415	0.344	24397	0.997
Test 77	26363	26288	0.997	9287	0.352	26288	0.997
Test 78	25714	25638	0.997	8114	0.316	25638	0.997
Test 79	25391	25331	0.998	9057	0.357	25331	0.998
Test 80	25423	25327	0.996	7779	0.306	25327	0.996
Test 81	25708	25657	0.998	8698	0.338	25657	0.998
Test 82	26319	26233	0.997	8539	0.324	26233	0.997
Test 83	26546	26531	0.999	9328	0.351	26531	0.999
Test 84	25805	25756	0.998	8925	0.346	25756	0.998
Test 85	25381	25313	0.997	8640	0.340	25313	0.997
Test 86	25729	25720	1.000	8873	0.345	25720	1.000
Test 87	25675	25669	1.000	9357	0.364	25669	1.000

Test 88	24387	24352	0.999	9232	0.379	24352	0.999
Test 89	24182	24093	0.996	8838	0.365	24093	0.996
Test 90	25862	25772	0.997	8764	0.339	25772	0.997
Test 91	24351	24311	0.998	9033	0.371	24311	0.998
Test 92	25321	25259	0.998	9658	0.381	25259	0.998
Test 93	25384	25292	0.996	9012	0.355	25292	0.996
Test 94	24497	24420	0.997	9186	0.375	24420	0.997
Test 95	24867	24775	0.996	9456	0.380	24775	0.996
Test 96	26998	26949	0.998	10978	0.407	26949	0.998
Test 97	24130	24102	0.999	8764	0.363	24102	0.999
Test 98	25769	25688	0.997	9876	0.383	25688	0.997
Test 99	25553	25513	0.998	9076	0.355	25513	0.998
Test 100	26094	25997	0.996	10259	0.393	25997	0.996
Test 101	25653	25582	0.997	10661	0.416	25582	0.997

Appendix B

DPLL	GSAT	7/8 Approx		
		Before	After	Best
415	624	631	625	631
476	642	625	640	640
487	630	631	639	639
551	634	618	613	618
406	634	640	620	640
431	626	635	626	635
459	647	635	649	649
328	628	632	633	633
512	619	643	627	643
461	623	635	623	635
574	608	631	644	644
594	644	638	629	638
284	623	643	619	643
522	629	635	623	635
281	618	633	632	633
509	612	630	625	630
457	632	643	629	643
603	622	627	637	637
476	637	615	644	644
495	640	619	633	633
576	626	624	614	624
512	642	641	633	641
569	633	629	633	633
595	635	636	630	636
484	631	631	623	631
381	614	635	628	635
596	635	624	631	631
616	630	644	628	644
552	639	642	629	642
439	640	638	631	638
195	627	628	632	632
483	615	618	622	622
584	629	631	641	641
485	638	629	624	629

546	637	631	620	631
291	632	627	634	634
514	626	621	627	627
567	633	631	626	631
410	639	630	629	630
252	647	640	628	640
343	638	621	631	631
469	617	629	636	636
528	623	631	630	631
521	629	614	636	636
423	620	638	630	638
292	636	619	626	626
531	620	637	632	637
327	639	633	631	633
445	634	659	625	659
491	617	628	614	628
326	633	629	628	629
545	630	612	629	629
593	619	603	621	621
603	637	630	624	630
414	632	630	631	631
546	623	631	643	643
515	636	650	615	650
507	639	636	617	636
442	635	634	638	638
605	636	630	639	639
344	643	636	633	636
414	639	631	622	631
367	632	632	623	632
407	613	625	629	629
417	635	625	631	631
550	619	620	623	623
551	628	635	626	635
294	643	627	626	627
451	629	628	613	628
343	621	632	627	632
375	618	637	630	637
231	636	626	629	629
426	619	631	622	631
478	622	637	618	637
487	619	634	642	642
542	635	620	631	631
473	639	637	627	637

495	639	639	626	639
572	637	625	640	640
412	638	631	624	631
412	653	639	626	639
443	634	629	629	629
597	642	634	628	634
469	630	636	637	637
523	636	631	615	631
541	637	623	634	634
593	645	620	636	636
507	625	633	628	633
483	635	636	624	636
608	628	624	638	638
626	626	622	632	632
493	634	621	618	621
244	634	614	639	639
528	631	605	636	636
427	612	633	625	633
560	637	631	637	637
470	640	617	626	626
472	638	640	641	641
445	640	639	630	639
609	628	639	625	639
425	636	644	628	644