Computability, Complexity, and Heuristics Final Report

Christian Eid

May 2022

1. Introduction

TSP, SAT, and 0-1 Knapsack are three NP-Complete problems with NP-Hard versions Max 0-1 Knapsack, Max SAT, and Min TSP. This report will compare approximation algorithms to solve these problems with the optimal solutions, and will compare the running time of the algorithms.

Max 0-1 Knapsack

For 0-1 Knapsack, there are four algorithms that are compared. Two of them are dynamic programming algorithms that will solve the Knapsack instance and return an optimal solution. The Greedy 2-Approximation gives an approximation within a factor of 2 of the optimal, and provides a solution in polynomial time. Lastly, the FPTAS is a fully time polynomial approximation scheme, which is polynomial in the instance size and a factor $1/\varepsilon$, which is provided to the algorithm along with the instance.

For 0-1 Knapsack, 100 instances of problems are randomly generated and written to a file, which will be read and passed through the algorithms. Important statistics including running time and solution are measured.

SAT

For SAT, the DPLL algorithm will decide the problem and determine whether there exists a solution that solves the SAT instance. A 7-8 Randomized Approximation algorithm is implemented for 3-SAT, as well as GSAT, a local search algorithm.

As with 0-1 Knapsack, 100 instances of 3-SAT are randomly generated and written to a file, which will be read and passed through the algorithms. Important statistics including running time and solution are measured.

TSP

For TSP, only one algorithm is implemented to solve it, and it is the Rosenkrantz, Lewis, and Stearns 2-approximation that using Kruskal's to walk the Minimum Spanning Tree of the STP instance.

The TSP algorithm is tested on TSP instances from TSPLIB, which provides the optimal tour to compare to the approximation.

2. Setup

In this section, the setup for the tests is discussed, including details of the programming implementation, and details of instance generating.

Knapsack Setup

A knapsack class is created in python which contains the knapsack instance as lists of item names, values, and costs, as well as a budget. The class has a few helper functions for the algorithms, and has a function to write the knapsack instance to a file. Each algorithm is implemented to use a knapsack object as input, and return the max value the algorithm finds.

Random Instance Generation

A knapsack generator function will create a random knapsack instance and use the class method to write it to a file. The random knapsack instance is created as follows: the number of items in an instance *itemCount* is randomly selected between 10 and 100, meaning all instances have between 10 and 100 items. A random budget is selected between 1 and 5000. Then, the value and cost list is populated with *itemCount* number of values and costs that range between 1 and 1000.

A new knapsack object is created based off of the randomly generated values, costs, and budget, and the knapsack instance is written to a csv file as a string. Each row of the file contains a knapsack instance.

Testing

A knapsack reader program will read the csv file full of knapsack instances represented as strings. For each row, it will parse the string into its components, extract the information, and create a new Knapsack object and adds it to a list of instances.

The main testing program will use this list of instances to test the algorithms. For each instance, each algorithm is called on a copy of the instance. The max val solution the algorithm returns and the time the algorithm takes are appended to lists for that algorithm, and the average, median, minimum, and maximum quality solutions as well as the average, median, minimum, and maximum running time is extracted from the lists. The average is simple to compute, and is just the sum of all solutions/times over the number of instances. For the quality of the solutions, the quality is measured as a total and percent of optimal. The dynamic programming algorithms both return the optimal solution, which allows us to measure the quality of the greedy and quality of the FTPAS. In comparing the four algorithms, the FTPAS will be passed a factor of 1.1.

SAT Setup

A SAT class is created in python which contains a SAT instance as lists of lists representing the SAT instance. Since the tests are to be done on 3-SAT, all clauses have 3 literals. A clause is represented as a list of numbers, where the number represents the literal, and the number negation represents whether the literal is negated or not. So, 3-SAT instance $(x_1 \vee x_2 \vee x_3) \wedge (\neg x_1 \vee x_3 \vee x_4)$ is represented in the program as $[[1, 2, 3], [-1, 3, 4]]$. The class also stores the truth assignment as a hash map where a number is mapped to a boolean variable True or False. The class also allows you to check the success of the truth assignment, and returns the number of satisfied clauses given an assignment. There also includes some helper functions for the algorithms.

Each algorithm is passed a SAT object, and returns the SAT object containing the truth assignment. DPLL returns False when it determines the instance to be unsatisfiable, and returns the assignment when it is satisfiable.

Random Instance Generation

In running tests, certain values would produce 3-SAT instances that were always satisfiable, or never satisfiable, so fine tuning was done to generate a mix of instances that were satisfiable and unsatisfiable. The number of clauses $NumClauses$ in an instance was a random number ranging from 300 to 400. The number of literals $NumLiteral$ that were available to create clauses ranged from 100 to 150.

For each clause, a random literal was picked from the range 0 to $NumLiteral$, and added to the clause. The generator checked to ensure that the random number chosen was not already in the clause. Then, the clause is appended to the instance. If the same exact clause is already found in the instance, then the randomly created clause is discarded, and a new random clause in generated. This ensures that in each instance, no two clauses are the same, and that each clauses contains three distinct literals. A clause was added until the number of clauses reached $NumClauses$. A new sat object was created given the randomly generated list, and the sat object was written to a file as as the list. This was repeated 100 times, which resulted in 100 randomly generated 3-SAT instances.

Testing

A SAT reader program reads the csv file and extracts the data from each row, using it to create a new SAT object. Each SAT object is appended to a list.

The SAT testing program will use this list of instances to test the algorithms. For each instance, each algorithm is called on a copy of the instance. The success of the returned assignment and the time the algorithm takes is recorded as a list.

TSP

TSP instances are given as a TSPLIB file, and each comes with an opt tour. All instances are from the Symmetric TSP section. A graph class is created, which is given a TSPLIB tsp instance file and turns it into a graph using python's networkx library. The graph object contains the list of vertices, and a list of edges that connects each vertices, which has a hash map attached which contains information about the edge. The hash map gives the edge weight.

A disjoint set forest class is also created, used by Kruskals to find the MST. The disjoint set forest is based off the pseudocode in the CLRS textbook, and implements union by rank with path compression.

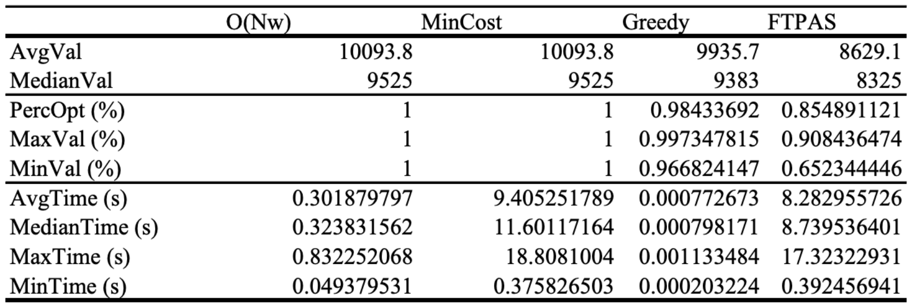
Unlike the two other problems, there is only one TSP algorithm, and it is tested against the value of the optimal tour given by the TSPLIP opt tour solution. Kruskal's algorithm is implemented using the disjoint set. The 2-approximation minTSP algorithm calls Kruskal's and walks the minimum spanning tree.

The files that I tested against are the pr76, ch150, bayg29, att48, where the number indicates the number of vertices in the instance.

3. Results

Knapsack Results

The results from the experiments are shown below, in Figure 1.



	O(Nw)	MinCost	Greedy	FTPAS
AvgVal	10093.8	10093.8	9935.7	8629.1
MedianVal	9525	9525	9383	8325
PercOpt (%)	1	1	0.98433692	0.854891121
MaxVal (%)	1	1	0.997347815	0.908436474
MinVal (%)	1	1	0.966824147	0.652344446
AvgTime (s)	0.301879797	9.405251789	0.000772673	8.282955726
MedianTime (s)	0.323831562	11.60117164	0.000798171	8.739536401
MaxTime (s)	0.832252068	18.8081004	0.001133484	17.32322931
MinTime (s)	0.049379531	0.375826503	0.000203224	0.392456941

Figure 1. Knapsack Data

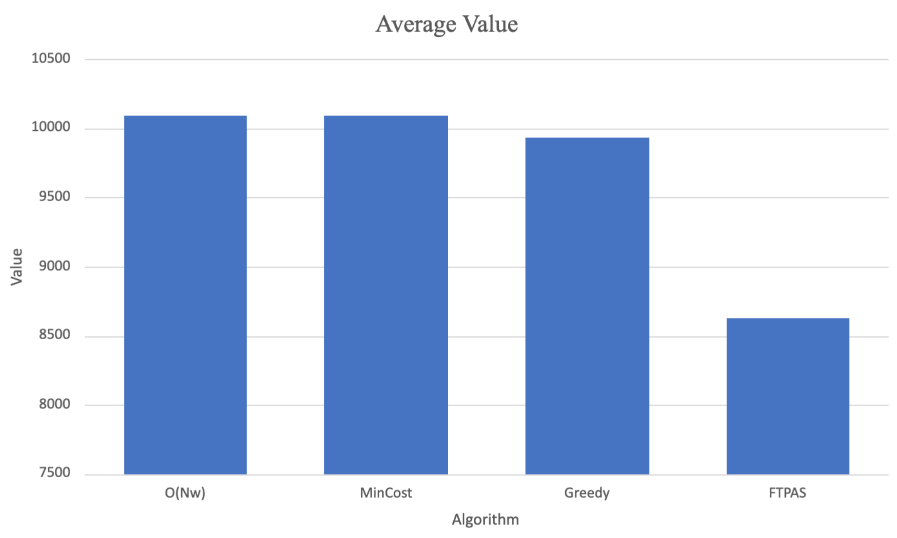


Figure 2. Average Values between the four Knapsack Algorithms

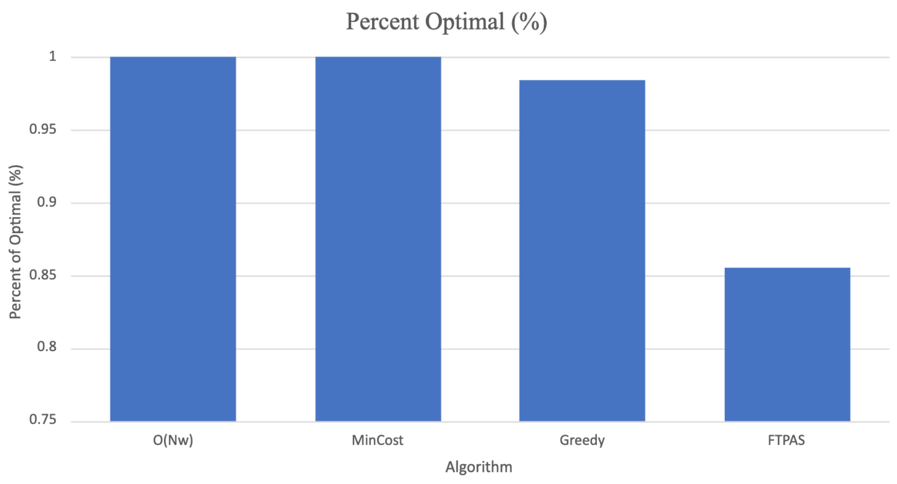


Figure 3. Average Value as a percent of optimal solution.

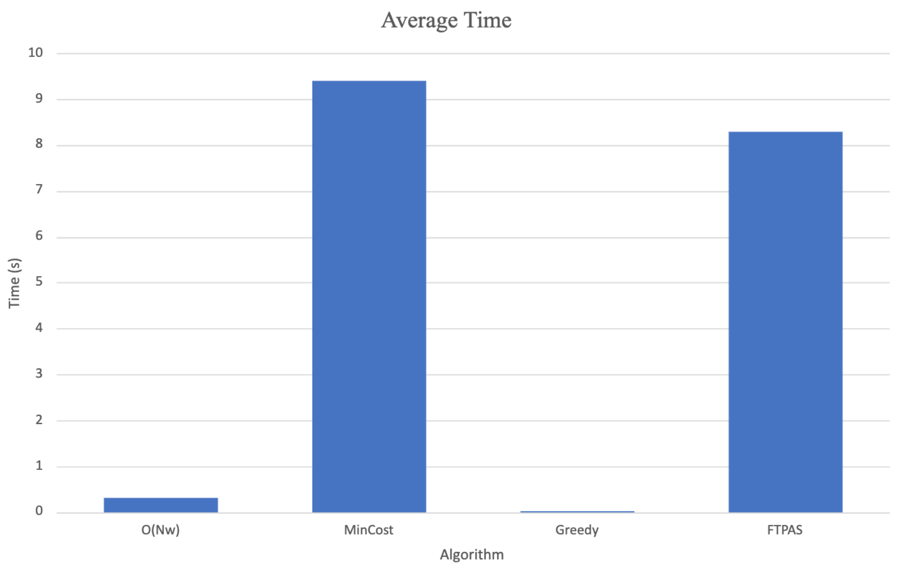


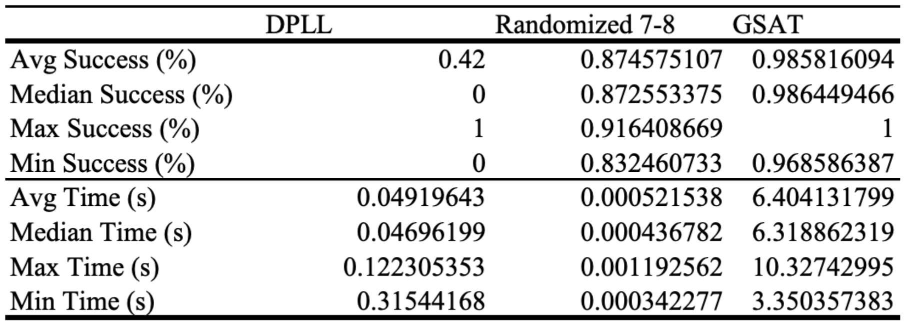
Figure 4. Time comparison between the Knapsack Algorithms

Discussion

If we compare the average value of each algorithm, Figure 2 shows that the O(nW) and MinCost algorithms have the same high value, while greedy is lower, and the FTPAS is even lower. As a fraction of the optimal solution, the two dynamic programming solve the problem, while greedy gets a high percent, and the FTPAS gets a lower solution, but still within 85% of optimal. The MinCost algorithm takes a very long time compared to the other algorithms, at an average of 9 seconds to solve each instance.

3-SAT Results

The results are first presented as a table with all results.



	DPLL	Randomized 7-8	GSAT
Avg Success (%)	0.42	0.874575107	0.985816094
Median Success (%)	0	0.872553375	0.986449466
Max Success (%)	1	0.916408669	1
Min Success (%)	0	0.832460733	0.968586387
Avg Time (s)	0.04919643	0.000521538	6.404131799
Median Time (s)	0.04696199	0.000436782	6.318862319
Max Time (s)	0.122305353	0.001192562	10.32742995
Min Time (s)	0.31544168	0.000342277	3.350357383

Figure 5. SAT Data

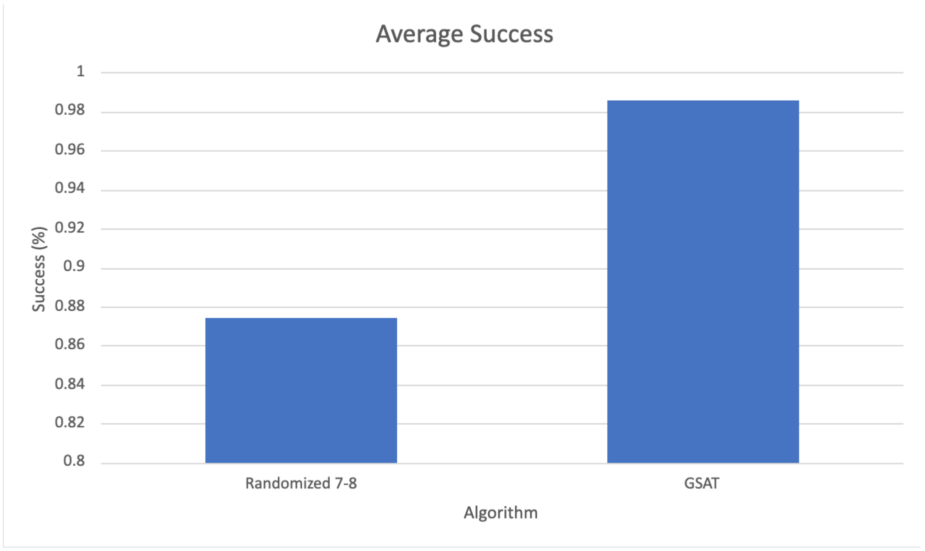


Figure 6. Average Success between the two approximation algorithms. Note that DPLL is not included because it either solves or doesn't solve. The data for DPLL included in the table shows how many SAT instances DPLL found to

be satisfiable, which was 42 out of 100.

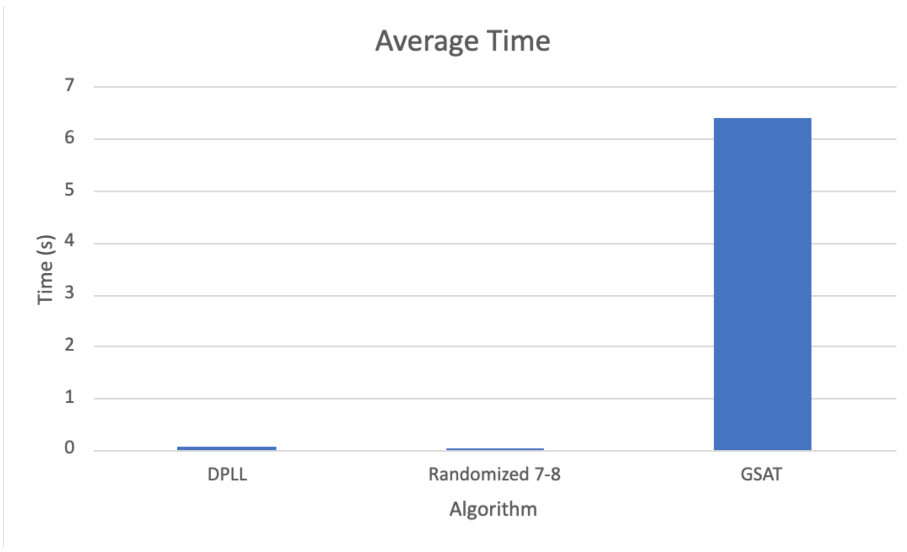
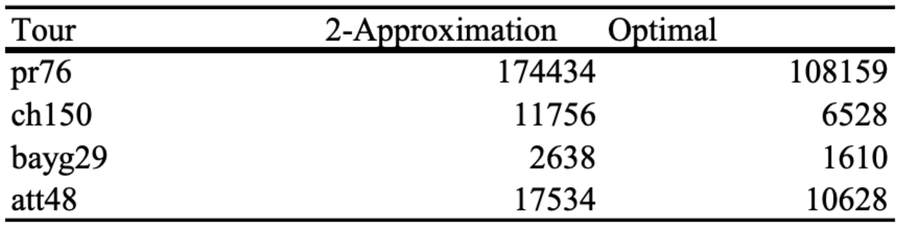


Figure 7. Average

Discussion

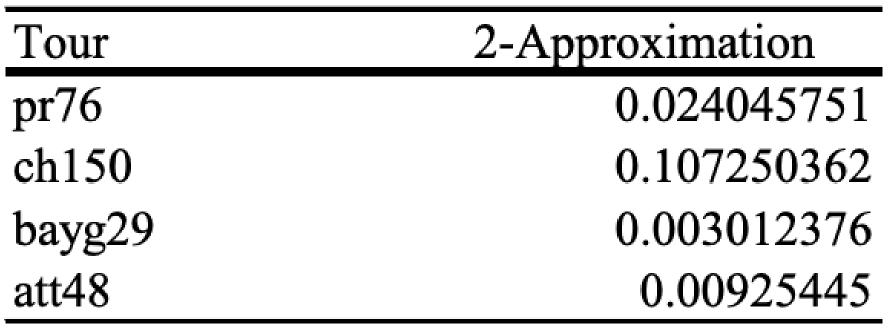
Interstingly, DPLL took little time to determine whether an instance was satisfiable or not compared to GSAT. This is likely due to the ratio of literals and clauses, as it was quickly able to go through and find empty clauses, or quickly able to determine the instance as satisfiable. Over the 100 instances, DPLL found 42 satisfiable. Of those 42, GSAT was able to find a satisfiable truth assignment for at least one, as it's max value was 1. The randomized 7-8 was able to satisfy between 87-91 percent of the clauses.

TSP Results



Tour	2-Approximation	Optimal
pr76	174434	108159
ch150	11756	6528
bayg29	2638	1610
att48	17534	10628

Figure 8. TSP Data



Tour	2-Approximation
pr76	0.024045751
ch150	0.107250362
bayg29	0.003012376
att48	0.00925445

Figure 8. TSP Time accross the four TSPLIB instances

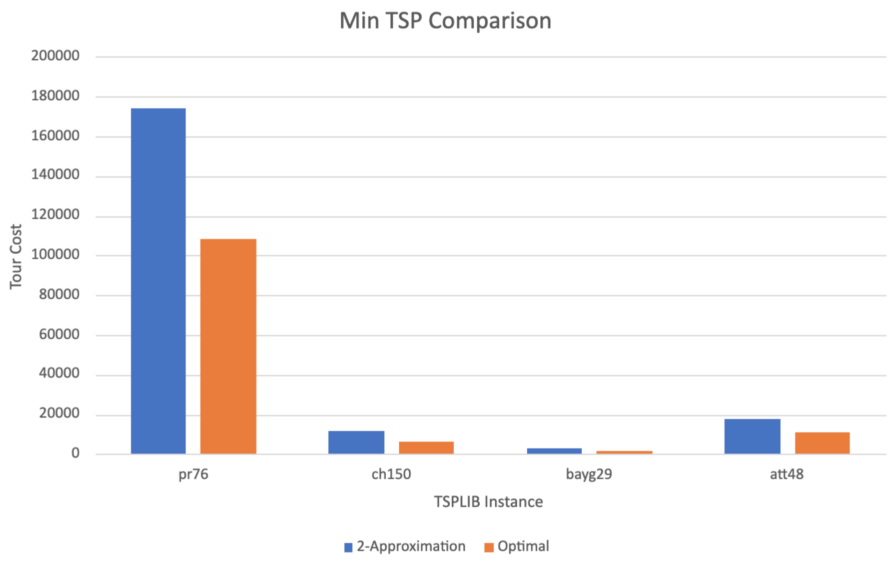


Figure 9. Comparison between the optimal tour and the approximation algorithm accross four TSPLIB instances.

Discussion

Figure 9 shows that the 2-approximation for MinTSP of Rosenkrantz, Lewis, and Stearns is always higher than the optimal tour, but not more than twice the optimal tour.

4. Conclusion

For the Knapsack tests, it seems that the greedy algorithm is quite close to the optimal, while running in less time. The MinCost and FTPAS both take a lot of time, and the factor inputting while have a large effect on the result and time the FTPAS algorithm takes. If I could explore something further, I would look at how changing the Factor affects the running time and the result, and make a plot of both these graphs. Another note is that the way the instances are randomly generated probably has a huge effect on the results. I tried to fine tune the way I randomly generated instances to make the running time feasible and the results interesting, i.e. not all the same. I am sure that with longer instances, different kinds of values in the instances, the data would be different. I am also interested in further experimenting with this. Seeing how the clause/literal ratio affects the SAT algorithms seems particularly interesting. Though these instances took a long time for the machine to run, I imagine that there are many uses for these algorithms with significantly larger instances, and it becomes clear that these problems are very hard to solve optimally. The approximations do a good job of finding a solution that gets us most of the way there, while allowing us to solve the problem in a reasonable time. One notable exception is that GSAT took more time to solve on average than DPLL.