Survey of Approximation Algorithms on Common Problems

John Dunagan, Durran Kelly, Zachary Mekus, Kun Zhang

1. Introduction

For our project, we look at some of the problems we have learned in class and implement approximation algorithms to solve them. We review the Set Cover/Maximum k Coverage, Bin Packing, and the Knapsack problems for which we implement multiple algorithms to get the exact or approximate solution. We then compared the algorithms to each other by looking at their times and objective values across varying parameters. The goal was to gain experience in implementing solutions to these problems, and insight to which solutions are the most feasible in practice.

2. Set Cover and Maximum *k* Coverage

For the Set Cover and Maximum *k* Coverage problems, we implemented a brute force algorithm, Greedy Cover, a LP Deterministic rounding algorithm, and a LP Random rounding algorithm. Our brute force and greedy algorithm were implemented in Java, whereas our linear program implementations were created in MatLab using linprog. We also wrote two input generation algorithms, one in Java for the brute force and greedy algorithms, and another in MatLab for the LP algorithms. We gathered data by running each algorithm with the same parameters and input whenever possible.

2.1 Input Generation

In order to test our implementations, we generated problem inputs in the following way. Given an input n of the size of the universe, we created an array of integers equal to that size, and set that as the universe U set. We then created sets by randomly selecting a set size between 1 and n, and uniformly assign values from the universe to a set until it was full. We did this for the total number of sets m defined by input to create our S set of sets. For the Java implementation, we ensured that a feasible solution always existed by continually creating more sets until the union between all the randomly generated sets covered the universe. It is however worth mentioning that, due to the fact each set was generated in a uniform way, having a good proportion of sets to the universe size generally guaranteed a feasible solution.

2.2 Set Cover

2.2.1 Algorithms

Brute Force

We decided to implement a brute force algorithm to solve the Set Cover problem to compare its optimum solutions to the approximation algorithms' solutions. As our brute force algorithm is O(2^n), we were only able to test with a small range of input. The algorithm is defined as such:

```
BruteForce(U, S):
Set size of the solution set equal to infinity
repeat for all set combinations
    Combine set combination
    Check if combination union covers the universe
    Set the combination as the solution set if it has less sets
    than the previously found solution set
```

Compared to the Greedy Cover algorithm described in the next section, implementation of the brute force algorithm was more difficult. Using the fact that the integers from 0 to 2^n - 1 will give every combination of subsets of *n* sets by selecting the sets that correspond to the 1 bit, we were able to compute the combinations linearly by essentially counting up from 0 to 2^n - 1. Throughout our experimentation, we compared our approximation algorithm solutions with the brute force solutions where possible.

Greedy Cover

One way to approximate the Set Cover problem is with an algorithm known as Greedy Cover. Greedy Cover works in the following way:

```
GreedyCover(U, S):
repeat
    Pick the set that covers the maximum uncovered elements
    Mark elements in the chosen set as covered
until done
```

The algorithm is very intuitive, simple to implement and admits a O(*logn*) approximation. From our experimentation however, we found that it produced an optimal solution each time we ran it. We were able to confirm this by comparing it to the brute force solution (whenever it was able to complete). We ran brute force and Greedy Cover with 4 different scenarios of universe *U* and set *S* sizes. We ran 10 trials of each to get an average run time. Our brute force algorithm could only compute the first two scenarios.

|U|	|S|	Brute Force Solution	Greedy Cover Solution
100	10	2	2
200	20	2	2
300	30	NA	2
10000	500	NA	2

We were able to observe that Greedy Cover ran optimally for the first two scenarios. We can safely assume that it was able to find the optimal solution in the last two scenarios as well as it would have found the sole set that covered all elements in the universe if it existed, simply by definition of the algorithm. Needless to say, the run time is not comparable as Greedy Cover as expected, is much faster.

|U|	|S|	Brute Force (seconds)	Greedy Cover
100	10	0.0193	0.0002
200	20	36.174	0.0005
300	30	NA	0.0009
10000	500	NA	0.2109

LP Rounding

Another way to find an approximate solution to the Set Cover problem is to formulate the problem as an LP and round it. The following is the LP formulation of set cover that was used[2]:

$$\begin{array}{lll} \text{minimize} & \sum_{i=1}^{n} x_i & \\ \text{subject to} & & \\ & \sum_{i:v \in S_i} x_i \geq 1 & \forall v \in U \\ & x_i \leq 1 & \forall i \in \{1, \ldots, n\} \\ & x_i \geq 0 & \forall i \in \{1, \ldots, n\} \end{array}$$

The LP solution is a lower bound on the optimal solution since it can "cheat" by choosing to partially cover a set. To get a feasible integer solution to Set Cover, we use two rounding methods:

1. *Deterministic Rounding* - This algorithm chooses the value f as the maximum number of sets in which any element appears. Then, it chooses all sets where the $x_i \geq 1/f$. This algorithm is guaranteed to be feasible [3].

2. *Random Rounding* - This algorithm chooses sets by interpreting the LP solution as the probability that the integer solution contains that set. The expected value of this solution is the same as that of the LP solution, but it is often not feasible. To solve this, we implemented the algorithm so that it keeps randomly choosing sets according to the LP solution until it gets a feasible solution [4].

We implemented the LP solution and rounding, and then for each of 5 different number of elements and set combinations, we ran 10 trials and found the average solution value for the LP solution and the two rounding solutions.

|U|	|S|	LP solution	Deterministic Rounding Solution	Random Rounding Solution
100	20	1.6928	5.1	2.6
200	40	1.3591	7.8	2.2
500	100	1.1969	13.8	2.3
1000	200	1.1395	20.3	2.3
2000	400	1.0979	32.1	2.3

Our results show that the random rounding solution performs significantly better than the deterministic rounding solution. We also tested the average time that each portion of the solutions took.

|U|	|S|	LP Time (seconds)	Deterministic Rounding Time (seconds)	Random Rounding Time (seconds)
100	20	0.0596	0.0003	0.0005
200	40	0.1821	0.0001	0.0002
500	100	1.6573	0.0002	0.0005
1000	200	10.8157	0.001	0.0022
2000	400	38.9938	0.0026	0.0046

The vast majority of the time taken to find the solutions was taken up by solving the LP. So we conclude that it does not matter in terms of time which rounding algorithm is chosen. All that matters is how good of a solution the rounding gives. Even though random rounding might have to repeat its process multiple times before finding a feasible solution, it still finishes very quickly and is a practical solution to this problem. The deterministic rounding algorithm is impractical because it gives such bad results. The fact that it is guaranteed to be feasible is not enough of a practical benefit.

2.2.2 Greedy Cover vs. LP Random Rounding

We ran 5 scenarios with 10 trials each to produce the following data:

|U|	|S|	Greedy Cover (seconds)	LP Time (seconds)	Random Rounding Time (seconds)
100	20	0.0004	0.0596	0.0005
200	40	0.0006	0.1821	0.0002
500	100	0.0018	1.6573	0.0005
1000	200	0.0053	10.8157	0.0022
2000	400	0.0254	38.9938	0.0046

Although both algorithms produced optimal solutions, it is clear to see that Greedy Cover is significantly faster than our LP Random Rounding algorithm. From our experimentation, we can conclude in practice that the Greedy Cover algorithm works well.

2.3 Maximum *k* Coverage

2.3.1 Algorithms

Brute Force & Greedy Cover

Our brute force and Greedy Cover algorithms were implemented in the same exact way as their Set Cover counterparts with an extra parameter to stop at *k* for Greedy Cover, and to only evaluate set combinations that had *k* total sets for the brute force algorithm.

Similarly to our Set Cover setup, we created 4 scenarios and ran our brute force and Greedy Cover algorithms 10 times each to produce the following data:

|U|	|S|	k	Brute Force, Covered Elements	Greedy Cover, Covered Elements
100	10	3	100	100
200	20	3	200	200
300	30	3	NA	300
10000	500	3	NA	10000

|U|	|S|	k	Brute Force (seconds)	Greedy Cover (seconds)
100	10	3	0.0151	0.0001
200	20	3	13.695	0.0003
300	30	3	NA	0.0002
10000	1000	3	NA	0.3297

The Greedy Cover algorithm produced an optimum solution each time as it was able to cover all elements using up to *k* sets.

LP Rounding

We used the following LP formulation of Maximum k Coverage [5]:

- *variables:* $x_1, \dots, x_m$ and $y_1, \dots, y_n$
- *constraints:*

$$\begin{aligned} &\sum_{i=1}^{m} x_i = k\,, && \text{(cardinality constraint)} \\ &\sum_{i:\, u_j \in S_i} x_i \geq y_j \quad \text{for } j \in \{1, \dots, n\}\,, && \text{(coverage constraints)} \\ &0 \leq x_i \leq 1 \quad \text{for } i \in \{1, \dots, m\}\,, \\ &0 \leq y_j \leq 1 \quad \text{for } j \in \{1, \dots, n\}\,. \end{aligned}$$

- *objective:* maximize $\sum_{j=1}^{n} y_j$

We rounded the solution using a similar random rounding scheme as in Set Cover. To calculate this rounding, we used the LP solution as weights and randomly draw an item without replacement from the population based on those weights. As before, we tested this rounding method 10 times for each of 5 different problem sizes.

|U|	|S|	number of sets to choose	LP Solution	Random Rounding Solution
100	20	3	100	98.8
200	40	3	200	198.9
500	100	3	500	498.2
1000	200	3	1000	999
2000	400	3	2000	1993.7

In every case, the LP finds a perfect solution, although there might not be an integer solution that good. The random rounding solution, on average does an excellent job of getting results that are either optimal or almost optimal. This implementation was also timed.

number of elements	number of sets	number of sets to choose	LP Time (seconds)	Random Rounding Time (seconds)
100	20	3	0.0551	0.0026
200	40	3	0.1033	0.0004
500	100	3	0.3981	0.0005
1000	200	3	0.6028	0.0006
2000	400	3	1.4011	0.0006

As before, solving the LP takes nearly the entire time, while rounding is practically instant. It is interesting to note that the LP for the Maximum *k* Coverage problem took significantly less time to solve than the LP for Set Cover, and it seems to also grow at a slower rate.

2.3.2 Maximum *k* Coverage and LP Random Rounding Comparison

Just as we did in the Set Cover and LP Random Rounding comparison, we ran 5 scenarios with 10 trials each to produce the following data:

|U|	|S|	k	Greedy Cover, Covered Elements	LP Solution	Random Rounding Solution
100	20	3	100	100	98.8
200	40	3	200	200	198.9
500	100	3	500	500	498.2
1000	200	3	1000	1000	999
2000	400	3	2000	2000	1993.7

|U|	|S|	k	Greedy Cover (seconds)	LP Time (seconds)	Random Rounding Time (seconds)
100	20	3	0.0004	0.0551	0.0026
200	40	3	0.0007	0.1033	0.0004
500	100	3	0.0017	0.3981	0.0005
1000	200	3	0.0052	0.6028	0.0006
2000	400	3	0.0245	1.4011	0.0006

Greedy Cover once again was able to cover all elements with $k = 3$ due to the uniform random set generation. Interestingly however, the LP with Random rounding did not always find k sets to cover all items. In the end, Greedy Cover turned out to be the most performant algorithm for the Maximum k Coverage problem.

3. Knapsack Problem

3.1 Input Generation

We created problems using two types of problem generators:

3.1.1 Uniform

The uniform generator generates *n* items with uniformly distributed sizes and profits within a provided range for each. Additionally, the uniform generator functions in 1 of 2 modes: *independent* and *density*. In *independent mode*, the size and profit for each item are generated separately (i.e. an item with the smallest possible size may have the largest possible profit, and the largest item may have the smallest). In *density mode*, the size and density (profit per unit size) are generated independently, and the profit for the item is calculated as the product of size and density. We introduced density mode as a method of correlating size and profit, and reducing the probability of "shoo-ins", tiny items with large profits.

3.1.2 Gaussian

The gaussian generator generates *n* items with normally distributed sizes and profits, given a provided mean and standard deviation for each. As the uniform generator, the gaussian generator has both *independent* and *density* modes. We introduced this generator to simulate a more "natural" distribution of items, where probabilities taper off at extreme values, and also to test the effects of mean item size on algorithm performance.

3.2 Algorithms

3.2.1 Exact Solver

An itemset-tracking implementation of the dynamic knapsack algorithm presented by William and Shmoys in *The Design of Approximation Algorithms* (Section 3.1, Algorithm 3.1), this algorithm maintains a list of dominant, feasible itemsets in the first *i* input items, iteratively increasing *i* from 0 to *n* and then returning the maximum-profit itemset. Because every itemset in the final list is feasible, the maximum-profit itemset must also be feasible. Furthermore, because every item in the final list dominates (has smaller size and greater profit than) every item not the final list, the maximum-profit itemset in the final list be also be the global maximum. This algorithm contains a fixed *n* iterations, but every iteration may as much as double the number of potential itemsets yielding exponential time.

The optimality of this algorithm was verified on a set of knapsack problems with known optimal solutions, taken from the Florida State University Department of Computing [6]. We used the optimal solutions from this algorithm to determine the approximation ratio for the greedy algorithm.

3.2.2 Modified Greedy Approximation

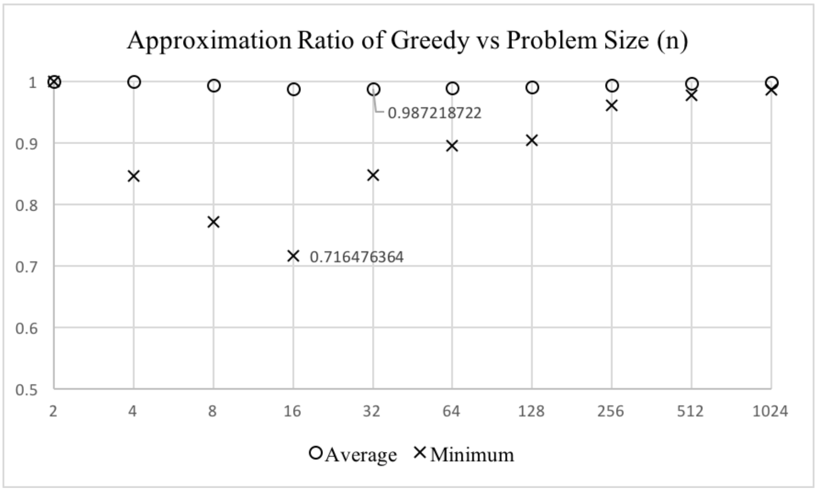
An implementation of the modified greedy algorithm presented in class, this algorithms takes the better solution from two constituent greedy algorithms. The first constituent algorithm sorts items by decreasing density and then greedily fills the knapsack with each successive item that fits. The second constituent algorithm functions similarly, but sorts by decreasing profit, regardless of size. This method minimizing the damage of potential input sets in which tiny-but-dense items leave insufficient room for larger, less-dense-yet-more-profitable items, displacing them from the solution set.

3.3 Results

3.3.1 Effects of Problem Size

We began by comparing the results of the greedy algorithm against the exact algorithm on inputs of identical distribution--a knapsack of capacity 1.0, sizes and densities of items both uniformly distributed on interval from 0 to 1-- but varying problem size (number of items) by powers of 2. For each problem size, we generated a batch of 1000 problems. Within each batch, every problem was fed to both the exact solver and the greedy approximation. We then divided the profit of each greedy solution by the profit of the corresponding exact solution to yield an approximation ratio. Finally, we calculated the average and lowest approximation ratio for each batch and plotted the results against the problem sizes.

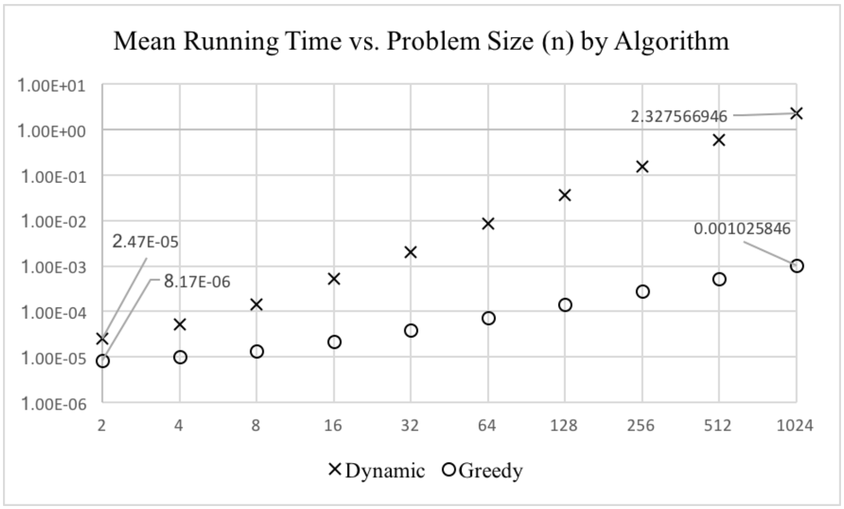
Shown below are the results, with circles representing the batch-wide average, and exes representing the batch-wide minima. Note that the x-axis scale (problem size) is logarithmic, and the y-scale (approximation ratio) is only displayed over the interval 0.5 to 1.0, the hard mathematical limits for the greedy approximation.



At a problem size of 2, it is impossible for the greedy algorithm to get anything but the optimal solution, so all ratios for that batch are 1. For larger problems, the average approximation ratio remains excellent, reaching a minimum of 0.987 at n=32. Interestingly, the approximation minima fall considerably between 8 and 32, but afterward ascend back towards the average case as n increases from there forward.

Clearly, over large random sets, the greedy algorithm has a statistical performance far greater than the adversarial case. As n grows, we observe that bad cases become increasingly unlikely, an additional measure of statistical robustness that makes the greedy algorithm more attractive in practice than the theoretical bound might suggest.

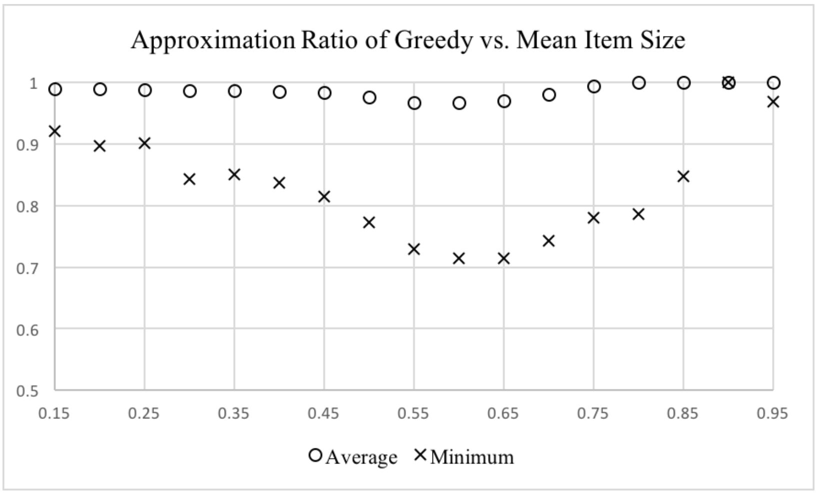
Furthermore, to illustrate the efficiency of the greedy algorithm over the dynamic algorithm, we measured the mean run time of each algorithm on problems from each problem batch. Note that this graph is drawn log-log to better articulate the curvature of each algorithm.



Interestingly, the exact algorithm appears approximately linear in log-log scale which corresponds to a polynomial in linear scale. This suggests that, within range 2 to 1024, the average case for the exact algorithm is approximately polynomial.

3.3.2 Effects of Mean Item Size

We conducted these tests by fixing the parameters of the gaussian generator in density mode and gradually sweeping the mean item size from 0.15 to 0.95. We fixed size and profit deviation at 0.15, and profit mean centered at 0.5. As the mean approaches 0, one can expect an increasing number of items to be generated with sizes 0 or less. We rounded these sizes to a very, very small positive value to avoid division errors. Observe that, given the density mode method of generation, these items cannot have profit greater than their size, and so have little effect on the problem, rather than being "shoo-ins" and inflating the approximations ratio by adding profit to both the approximation and optimal.



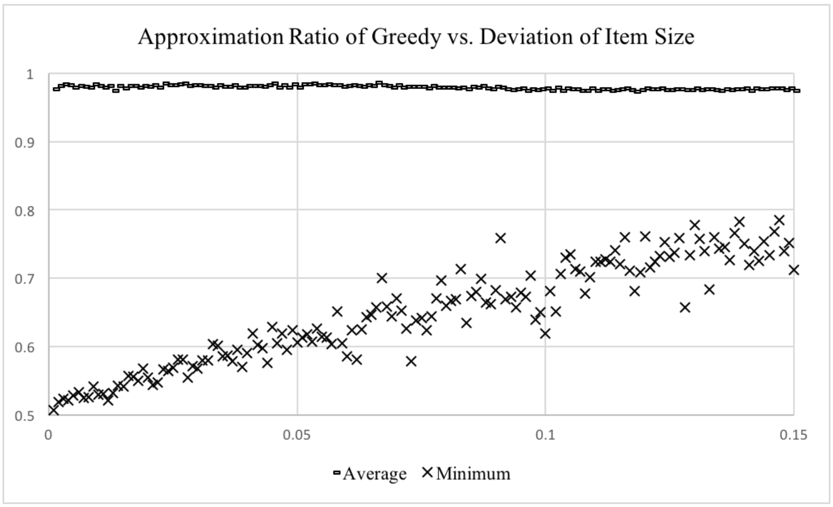
Interestingly, the solution quality reaches a clear minimum between 0.55 and 0.65, and not centered on 0.5 as one might expect given the adversarial case presented in proof.

3.3.3 Effects of Item Size Deviation

The last test we ran were the effects of the spread of item sizes. For this test, we used the gaussian generator to generate normally distributed problem sets, each with a knapsack capacity of 1, mean item size of 0.5, and normal density from 0.0 to 1.0, centered on 0.5.

This test was motivated by the insight that the worst-case problem for the ½-approx bound is generated using objects of similar size, and close to ½. seem to occur in situations

We ran 150 batches of 1000 problems, each with n=32 items, and with item standard deviation varying from 0.01 to 0.15 (approximately normal over 0 to 1).



Again, while item deviation had influence effect on the probability of getting a “tough” problem, it had no appreciable effect on average case performance, further demonstrating the statistical robustness of the greedy algorithm on a large numbers of problems.

4. Bin Packing

4.1 Overview

We set up the problem as a discrete bin packing problem. Each bin has size 100. Each item has size with half chosen from the uniform distribution between 1-100 and half from the uniform distribution between 10-100. We found that with using just the uniform distribution between 1-100 that too many small items made it hard to differentiate between the performance of algorithms, so we made items of size less than 10 more unlikely. We implemented five algorithms in Java. Each algorithm has a version where the items are not taken in the order they're given, and a "sorted" version where the items are sorted in descending order before packing.

We wanted to test the performance of each algorithm by measuring how many bins each uses to pack the same items compared to the other algorithms. To test this, we used 5 different problem sizes with the number of items being 1250, 2500, 5000, 10000, 20000. For each problem size, we do the following procedure 10 times: generate items, run each algorithm on them, while recording the number of bins used to pack them and the time taken for each, sort the items, and run each algorithm again and record the number of bins used and the time taken. We then average the performance and times for each of the ten algorithms (including sorted algorithms) for each of the 5 problem sizes, and analyze the data.

4.2 Algorithms

Next fit - Put each item into the same bin as the previous one. If it doesn't fit, open a new bin and put it there.

First fit - Put each item into the first (earliest opened) bin into which it fits. If it fits in no bins, open a new one and put it there.

Worst fit - Put each item into the open bin with the most available space. If it fits in no bins, open a new one and put it there. Break ties by choosing the earlier bin.

Best Fit - Put each item into the open bin with the most least space where this bin still fits. If it fits in no bins, open a new one and put it there. Break ties by choosing the earlier bin.

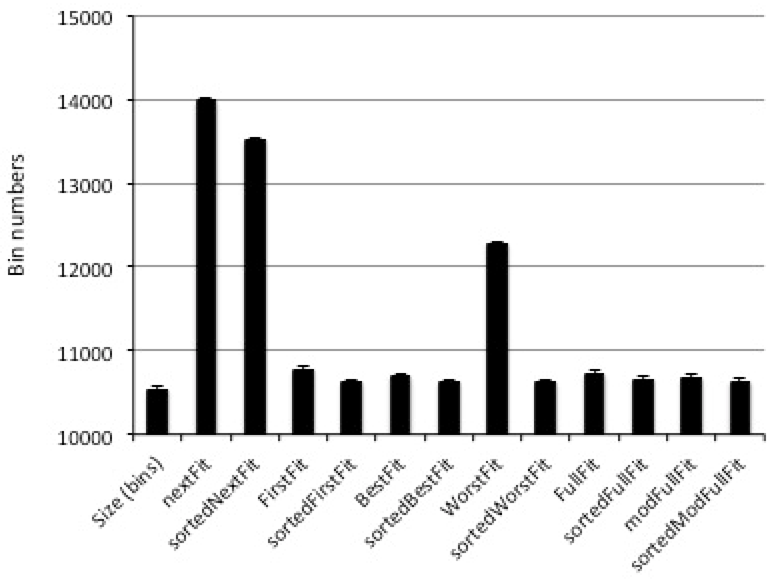
Full Fit - This is a custom algorithm. Each iteration, with dynamic programming, find a combination of items that fits in the bin perfectly. If no combination fits perfectly, then try to find a combination that fits into the bin size minus one. Continually do this until you fill the bin as well as you can. Then remove the items that were just used and move on to the next bin.

Modified Full Fit - This is a custom algorithm. Divide items into 2 groups (BIG and SMALL), with BIG items larger than 25, and SMALL items less than or equal to 25. Pack BIG items with Full Fit algorithm, then pack SMALL items with First Fit algorithm.

Each algorithm has an "offline" variation, where the algorithm is run on the items after they have been` sorted in decreasing order (from largest to smallest).

4.3 Results

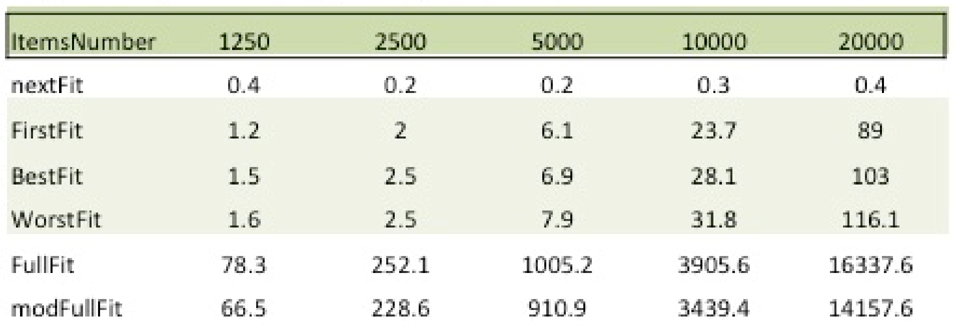
4.3.1 Solution Quality



Shown above are the number of bins used by each of the five algorithms with and without sorting for 20000 items. The results look practically the same for any large problem sizes, so performance is not dependent on problem size. Also shown on the left is the sum of the item sizes divided by the bin size, which must be at most OPT. The first observation is that most of these algorithms perform very well. They are very close to the sum of the bin sizes, so they must be very close to OPT. Every algorithm performs better using already sorted items. WorstFit particularly improves, going from performing badly to performing very well. NextFit is the only algorithm that performs badly even after being given sorted items. After sorting the items, BestFit and FirstFit actually pack items into exactly the same number of bins no matter what items are given. If sorting is possible, then BestFit performs best, but if it is not possible, then ModifiedFullFit performs best.

4.3.2 Speed

Table: Running time of the algorithms (microsecond)

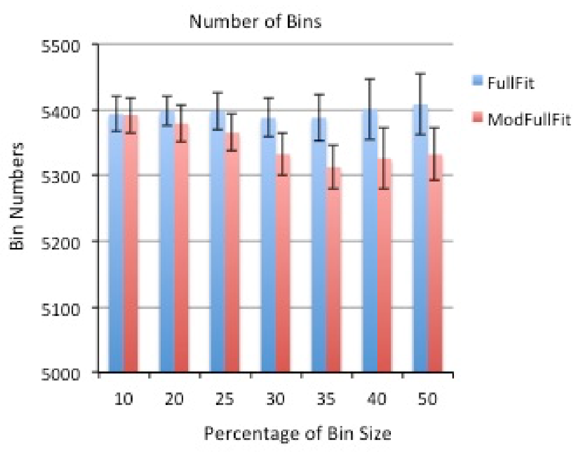


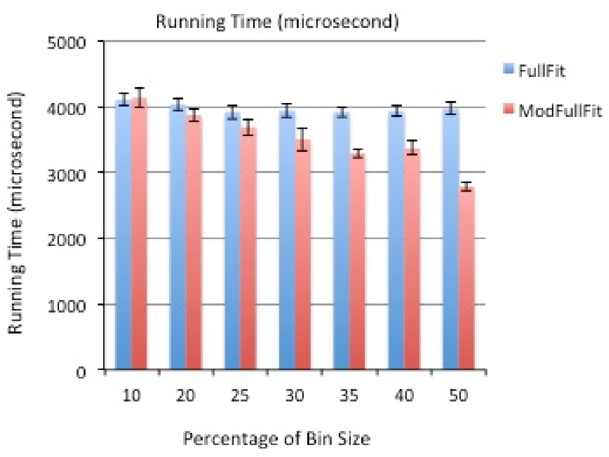
ItemsNumber	1250	2500	5000	10000	20000
nextFit	0.4	0.2	0.2	0.3	0.4
FirstFit	1.2	2	6.1	23.7	89
BestFit	1.5	2.5	6.9	28.1	103
WorstFit	1.6	2.5	7.9	31.8	116.1
FullFit	78.3	252.1	1005.2	3905.6	16337.6
modFullFit	66.5	228.6	910.9	3439.4	14157.6

Shown above are the speeds of each of the five algorithms in milliseconds on different problem sizes. The results look practically the same when comparing the times of these algorithms after sorting, so the sorted algorithm run times are omitted.

NextFit is the only linear time algorithm, and it is by far the fastest. FirstFit, BestFit, and WorstFit are all similar in their speed as they perform similair greedy operations. FirstFit is the fastest since it does not need to always look at every bin. FullFit and ModifiedFullFit are also similar to each other, being much slower than the previous options. They seem to be about 100 times slower than the greedy algorithms in this time frame, and also grow at a faster rate as the problem size grows.

4.3.3 Modified Full Fit





The Modified Full Fit algorithm improves on the full fit algorithm by holding the “small” items until the end so they are not used wasted trying to get a perfect fit. The above graphs show the differences in performance and speed between the two algorithms with constant problem size of 10000 items and a changing threshold for “big” and “small” items. A higher threshold means more time is spent on the first fit stage, while a smaller threshold means more items are being packed by full fit. Since first fit is faster than full fit, any threshold makes modified full fit faster than full fit, and an increasing threshold makes it increasingly faster. As the threshold increases, the number of bins that modified full fit packs decreases then increases. The optimal value that tested proved to be 35. The modified algorithm is an improvement of both performance and speed over the original full fit algorithm.

5. References

[1]:
https://research.engineering.wustl.edu/~bmoseley/CS581T-SP17/additionalmaterial/lecture_3.pdf

[2]: http://theory.stanford.edu/~trevisan/cs261/lecture08.pdf

[3]: Approximation Algorithms Section 1.3

[4]: Approximation Algorithms Section 1.7

[5]: http://www.cs.cornell.edu/courses/cs4820/2014sp/notes/maxcoverage.pdf

[6]: https://people.sc.fsu.edu/~jburkardt/datasets/knapsack_01/knapsack_01.html