HIGH SCHOOL EDITION
Journal of Student Research

Volume 12 Issue 4 (2023)

Using Heuristic Algorithms to Solve the 0-1 Knapsack Problem

Sanatan Mishra[1] and David Perkins[2#]

[1]Homestead High School
[2]Hamilton College
[#]Advisor

ABSTRACT

The 0-1 Knapsack problem is an NP-complete optimization problem with applications in many places where the most valuable items must be selected for a finite pool. Heuristic algorithms are used to solve that problem, as the exact solutions take far too much time to be useful in such applications. This paper introduces ten novel heuristic algorithms. After that, the paper analyzes their performance on 29 publicly available datasets against one existing heuristic algorithm named "Standard Greedy." The results show that one of the novel heuristics, "Defensive Greedy," is the best in balancing speed and accuracy with a time complexity of $O(n \log n)$. The advantages of "Defensive Greedy" include that it is around 44.8% faster than "Standard Greedy" and much faster than any exact solution. However, its primary disadvantage is that it lags behind "Standard Greedy" in accuracy by approximately 28%. The nine other novel heuristics were not as fast as "Defensive Greedy," but two novel heuristics, "Max of All" and "Max of Two," had a higher accuracy of 2.9% and 2.6%, respectively, than that of "Standard Greedy."

Introduction

A person is moving out of their home and must take as many possessions as possible. However, the combined weight of possessions they can take with them is constrained by the weight-carrying capacity of their vehicle. So, they need to optimize by taking the most valuable combination of items with them while keeping the combined weight below the upper limit (weight-carrying capacity of their vehicle). This type of problem is called the Knapsack problem. Now, if the person can only take whole items from a finite number of total items, it would be called the 0-1 Knapsack problem, a special case of the Knapsack problem that is the focus of this paper.

The Knapsack Problem and the 0-1 Variant

The Knapsack problem is a combinatorial optimization problem where one is trying to maximize the value of the items they are packing in their container.[1] However, the primary constraint here is that they can only pack a certain total weight of items. So, any solution to this problem requires trying two things:

$$\text{Maximize} \sum_{i=1}^{n} p_i \, x_i \tag{1}$$

and

$$\sum_{i=1}^{n} w_i \, x_i < W \tag{2}$$

These equations assume that i is the specific item in question, W is the capacity of the container, w_i is the item's weight, p_i is the item's profit or item's value, and x_i is how much of the item was taken.

The 0-1 Knapsack problem is a special case where items cannot be divided into smaller parts. This means that items can either be taken entirely or not taken. So, x_i, the number of items taken, can either be 1, which means that the item was taken, or 0, which means the item was not taken. This variant of the Knapsack problem, the 0-1 Knapsack problem, is NP-complete, so it does not have an exact solution in polynomial time.[2]

The 0-1 Knapsack problem can be applied in many cases, including finding the best way to cut raw materials, selecting investments, generating keys for knapsack-based cryptosystems, inventory management systems, yield management, college admissions, and many others.[3,4] As some of these scenarios have heavy time constraints, our greedy heuristics for the 0-1 Knapsack problem can be used to help improve the speed of decision-making in these cases.

NP-Complete Problems

An NP-complete problem is, by definition, a problem that is both NP and NP-hard.

NP problems are those with nondeterministic solutions that run in polynomial time. A nondeterministic solution verifies whether a solution is correct or not. This means the solution to an NP problem can be verified in polynomial time.[5]

If a problem is NP-hard, it is at least as intractable as the least tractable problem in NP,[6] and every single problem in NP can be Cook-reduced to it in polynomial time.[5] So, every single NP problem can be reduced to every NP-hard problem.

As NP-complete problems are NP, each NP-complete problem can generalize to each NP-hard problem. As NP-complete problems are NP-hard by definition, each NP-complete problem can generalize to all other NP-complete problems.[5]
To prove that a problem is NP-complete, it must be proven that it is both NP *and* NP-hard.[5]

The NP-Completeness of the 0-1 Knapsack Problem

The 0-1 Knapsack problem is an NP problem as its verification algorithm runs in $O(n)$ time. Verifying whether a given solution to the 0-1 Knapsack problem is the optimal solution requires verifying whether it comes within the problem's weight limits, which runs in $O(n)$ time to add up all the items' weights. This is followed by verifying whether the solution has the optimal value, which runs in $O(n)$ time to add up all the items' values. This can prove that the 0-1 Knapsack problem is an NP problem as its verification algorithm runs in $O(n)$ time, which is polynomial.[7]

To verify whether an NP problem like the 0-1 Knapsack problem is NP-complete, it is necessary to ensure that it is NP-hard. This can be done by reducing another NP-complete problem, which must be NP-hard by definition, to the 0-1 Knapsack problem. If all NP problems reduce to the NP-hard problem that reduces to the 0-1 Knapsack problem, all NP problems must reduce to the 0-1 Knapsack problem, making it NP-hard. In this case, the Exact Cover problem reduces to the Knapsack problem, which reduces to the 0-1 Knapsack problem.[7] So, we know that the 0-1 Knapsack problem is NP-hard as another NP-hard problem, the Exact Cover problem, reduces to it. As we know that the 0-1 Knapsack problem is NP and NP-hard, we know it is NP-complete.

Existing Exact Solutions

The first exact solution to this problem is brute force, where every possible combination of items is generated and the most valuable combination of items whose combined weight is within the capacity is the one that is chosen as the final answer.[8]

Another solution to this problem is to use a Dynamic Programming approach,[8] where we make a table as described next. The table has i columns and j rows, where i is the number of items and j is the capacity of the knapsack,

previously defined as W. Let items 1 through i have cumulative weight w_i and cumulative price p_i, and let $Table[i, j]$ be the optimal solution for items 1 through i if the knapsack has capacity j. The value of $Table[i, j]$ can be split into two cases, one of which is where the knapsack it represents includes item i and the other is where that knapsack does not include item i. If $j < w_i$, or if the combined weight of all items chosen up to i is greater than the capacity of the knapsack, then $Table[i, j] \leftarrow Table[i-1, j]$, meaning that the optimal price for all items before item i is the optimal price for all items up to and including i.[8] Otherwise, $Table[i, j] \leftarrow maximum(Table[i-1, j], p_i + Table[i-1, j-p_i])$, meaning that either the knapsack with item i or without item i will be used, depending on which one has a higher value.[8] Also, $Table[0, j] \leftarrow 0$ and $Table[i, 0] \leftarrow 0$, as the value has to be 0.[8]

The third and last well-known exact solution to this problem is a partition-based approach.[9] In it, the set of all the items is partitioned into two sets A and B containing approximately half of all the items, and for each subset C of set A, the subset D of set B with the highest value such that the combined weight of sets C and D is less than W. Then, the greatest combined value out of all the valid combinations of C and D is chosen as the correct solution with the most optimal value.

Brute force has a time complexity of $O(n!)$, dynamic programming has a time complexity of $O(2^n)$,[8] and the partition-based approach has a time complexity of $O(2^n)$.[9]

Existing Heuristic Solutions

While the exponential time complexity of dynamic programming and the partition-based approach is stellar compared to the factorial time complexity of brute force, it is still extremely slow. So, to speed things up, a polynomial time solution is necessary. However, a polynomial time exact solution does not exist. Therefore, a polynomial time heuristic must be used. One such polynomial-time heuristic is a Fully Polynomial-Time Approximation Scheme.[10]

Another one, which shall be used in this paper, is the greedy approximation scheme. A greedy algorithm does what is best for itself in the short term.[11]

In the rest of this paper, we will introduce one standard heuristic algorithm for the 0-1 Knapsack problem and ten novel ones developed by us. Their performance (run time) and accuracy will be analyzed by running them on 29 standard datasets. First, the “Methods” section introduces the algorithms, analysis methodology and the exact metrics used in the analysis. Then, the “Results and Discussion” section introduces the expected results for the algorithms on time and value and analyzes the algorithms on the metrics introduced in the “Methods” section. Finally, the “Conclusion” section concludes the analysis.

Methods

Algorithms Used

This paper analyzes novel heuristic algorithms for the 0-1 Knapsack problem. They fall into four categories: greedy, stingy, sliding threshold, and composite.

Greedy Algorithms

The first greedy algorithm used is effectively the “Standard Greedy” algorithm generally used for the fractional knapsack problem made by Dantzig.[12] It sorts items by the ratio of their weight to their value and takes the first items that fit. A novel variation on this would be a “Scored Greedy” algorithm, which uses a score that is not the ratio of an item’s value to its weight, but instead p_i^a / w_i^b, where a and b have been determined experimentally to be 3 and 1.5, respectively.

They were determined by randomly generating small-scale data and using different values of a and b until we figured out values where changing them in either direction would make the algorithm perform worse.

Algorithm 1 Standard Greedy Algorithm

given items i with price p_i and weight w_i, capacity W
for all i in items **do**
$ratio_i \leftarrow w_i/p_i$
end for
sort items by ratio in ascending order total-
weight ←0
totalvalue ←0
for all i in items **do**
if w_i+ totalweight < W **then add** item **i** to
knapsack totalweight ←totalweight +w_i
totalvalue ←totalvalue +w_i
end if
end for
return knapsack, totalvalue

Algorithm 2 Scored Greedy Algorithm (Variation on "Standard Greedy" Algorithm)

given items i with price p_i and weight w_i, capacity W
for all i in items **do**
$score_i \leftarrow p^3/w^{1.5}\ _i$
end for
sort items by score in descending order total-
weight ←0
totalvalue ←0
for all i in items **do**
if w_i+ totalweight < W **then add** item **i** to
knapsack totalweight ←totalweight +w_i
totalvalue ←totalvalue +w_i
end if
end for
return knapsack, totalvalue

Another novel greedy algorithm used is one where items are sorted by value and new items are taken as long as they fit, defined as the "Heavy Greedy" approach. Another novel variation on this would be to use limits that only allow a specific weight of items to enter to maximize the number of items taken in, called "Limited Greedy". These limits are needed to ensure that no items with too much weight for their high value are put in the knapsack, ensuring that only good deals (items with a high ratio of value to weight) are kept.

Algorithm 3 Heavy Greedy Algorithm

given items i with price p_i and weight w_i, capacity W
sort items by price in descending order total-
weight ←0
totalvalue ←0
for all i in items **do**
if w_i+ totalweight < W **then add** item **i** to
knapsack totalweight ←totalweight +w_i
totalvalue ←totalvalue +w_i
end if
end for
return knapsack, totalvalue

Algorithm 4 Limited Greedy Algorithm (Variation on "Heavy Greedy" Algorithm With Limits)

given items i with price p_i and weight w_i, capacity W **sort** items
by price in descending order remainingcapacity ←W
totalvalue ←0
for all i in items **do**
if remainingcapacity > 0.4 × W AND w_i ≤ 0.8 × remainingcapacity **then add** item **i** to
knapsack
remainingcapacity ←remainingcapacity − w_i
totalvalue ←totalvalue +w_i
else if remainingcapacity ≤ 0.4 × W AND w_i ≤ remainingcapacity **then add** item **i** to
knapsack
remainingcapacity ←remainingcapacity − w_i
totalvalue ←totalvalue +w_i
end if
end for
return knapsack, totalvalue

Similarly, there is a novel greedy algorithm called "Transitioning Greedy" that first sorts by value and gets the most valuable items until the knapsack is x% full (experimentally determined to be 40% after fitting constant to randomly-generated small-scale data), after which the remaining items are sorted by the score ${p_i}^a$ / ${w_i}^b$ used in "Scored Greedy" (where a is 3 and b is 1.5), and then the first items that fit are taken. Essentially, "Transitioning Greedy" transitions from "Heavy Greedy" at the start to "Scored Greedy" once the knapsack is x% full.

Algorithm 5 Transitioning Greedy Algorithm

given items i with price p_i and weight w_i, capacity W **sort** items
by weight in descending order remainingcapacity $\leftarrow W$
totalvalue $\leftarrow 0$
for all i in items **do**
if remainingcapacity $> 0.6 \times W$ AND $w_i \leq 0.4 \times$ remainingcapacity **then add** item **i** to
knapsack
remainingcapacity $\leftarrow$remainingcapacity $- w_i$
totalvalue $\leftarrow$totalvalue $+w_i$
remove item **i** from list of items
end if
end for
for all i in items **do**
$score_i \leftarrow p^3/w^{1.5}{}_i$
end for
sort items by score in descending order
for all i in items **do**
if w_i+ totalweight $< W$ **then add** item **i** to
knapsack totalweight $\leftarrow$totalweight $+w_i$
totalvalue $\leftarrow$totalvalue $+w_i$
end if
end for
return knapsack, totalvalue

Lastly, there is a novel "Defensive Greedy" algorithm, which packs the lightest items to get as many items as possible.

Algorithm 6 Defensive Greedy Algorithm

given items i with price p_i and weight w_i, capacity W
sort items by weight in ascending order total-
weight $\leftarrow 0$
totalvalue $\leftarrow 0$
for all i in items **do**
if w_i+ totalweight $< W$ **then add** item **i** to
knapsack totalweight $\leftarrow$totalweight $+w_i$
totalvalue $\leftarrow$totalvalue $+w_i$
end if
end for
return knapsack, totalvalue

Stingy Algorithms

A stingy algorithm is one where the algorithm starts with everything and has to remove items to get the solution under the constraints. Stingy algorithms are novel for the 0-1 Knapsack problem and have never been used before for this problem. The first stingy algorithm used is "Deal Stingy," which sorts items by the score $p_i^3/w_i^{1.5}$ used in scored greedy and removes the "worst deals" or the items that have the lowest score.

Algorithm 7 Deal Stingy Algorithm

given items i with price p_i and weight w_i, capacity W
totalweight ←0
totalvalue ←0
for all i in items **do**
$score_i \leftarrow p_i^3 / w_i^{1.5}$
totalweight ←totalweight +w_i to-
talvalue ←totalvalue +w_i **add** item **i** to
knapsack
end for
sort items by score in ascending order
for all i in items **do**
if w_i+ totalweight > W **then remove** item **i**
from knapsack totalweight ←totalweight
− w_i
end if
end for
return knapsack, totalvalue

The other stingy algorithm used is "Weight Stingy," which sorts items by their weight, assumes all items to be in the knapsack, and removes the lightest items until the weight is lower than the limit.

Algorithm 8 Weight Stingy Algorithm

given items i with price p_i and weight w_i, capacity W
totalweight ←0
totalvalue ←0
for all i in items **do**
add item **i** to knapsack totalweight
←totalweight +w_i totalvalue ←to-
talvalue +w_i
end for
sort items by weight in ascending order
for all i in items **do**
if w_i+ totalweight > W **then remove** item **i**
from knapsack totalweight ←totalweight
−w_i
end if
end for
return knapsack, totalvalue

Sliding Threshold

A "Sliding Threshold" algorithm randomly selects an item to be the value threshold, and if there are not enough items at a certain point, the threshold slides down by a specific factor. The threshold needs to slide down to ensure that the knapsack can be filled as much as possible (in other words use up all the capacity of the knapsack). This is a novel approach to this problem.

Algorithm 9 Sliding Threshold Algorithm

given items i with price p_i and weight w_i, capacity W
totalweight ←0
totalvalue ←0
threshold ←p_i/w_i of random i
for all i in items **do**
if i > len(items) / 4 AND len(knapsack) ≤ len(items) / 4 **then**
threshold ←threshold × 0.8
else if i > len(items) / 2 AND len(knapsack) ≤ len(items) / 8 **then**
threshold ←threshold × 0.8
end if
if w_i+ totalweight > W AND threshold ≤ p_i/w_i **then remove** item
i from knapsack
totalweight ←totalweight −w_i
end if
if totalweight == W **then exit**
loop
end if
end for
return knapsack, totalvalue

Composite Algorithms

One composite algorithm for this problem takes the maximum of using all the other heuristics mentioned above to find the most optimal solution, called "Max of All." A slimmed-down version of this algorithm, called "Max of Two", uses only the "Standard Greedy" and "Heavy Greedy" approaches for the maximum. The "Standard Greedy" and "Heavy Greedy" algorithms were chosen because the times when they had the highest value out of all the heuristics were the most different.

Analysis Methodology

We ran each of the 11 algorithms (10 novel and one pre-existing) on 29 unique datasets, of which 21 were large-scale datasets from Unicauca and eight were small-scale datasets from Florida State University.[13, 14] We used that many diverse datasets to ensure that our results would be statistically significant, indicating a general trend and not just the result of chance.

We ran each algorithm ten times on each dataset and then took the mean value and mean time over the 10 trials to prevent abnormalities caused by the algorithms' randomness and the computer on which the algorithms were running.

As far as the timing was concerned, we used wall clock time on a 2020 13" MacBook Pro with 4 Thunderbolt 3 Ports, a 1 TB SSD, 16 GB 3733 MHz LPDDR4X, and an Intel Core i5-1038NG7 CPU with 4 cores and 8 threads running macOS version 12.4 "Monterey" and Python 3.10.

Our procedure can be improved by replacing wall clock time with system time and using a computer with fewer background processes running.

Metrics Used

We measured the number of datasets on which an algorithm got the highest value amongst all the algorithms presented here as its "win count."

We also measured the mean percentage error, E, from the optimal value of the knapsack, of the algorithm across all datasets, where n is the number of datasets, o_i is the optimal value of the knapsack on dataset i, and v_i is the value of the knapsack on dataset i given by algorithm a.

$$E_a = \frac{1}{n}\sum_{i=1}^{n}\frac{o_i - v_i}{o_i} \times 100 \qquad (3)$$

Afterwards, each algorithm's mean normalized time across all datasets (normalized to the fastest algorithm on each dataset), τ_a, was taken; to make comparisons fair, we normalize within results on the same dataset. In the following equation for τ_a, n represents the number of datasets, t_i represents the time taken by algorithm a on dataset i, and T_i represents the time taken by the algorithm that took the least time on dataset i.

$$\tau_a = \frac{1}{n}\sum_{i=1}^{n}\frac{t_i}{T_i} \qquad (4)$$

Finally, we took the algorithm's mean normalized value of items (normalized to the minimum value found by an algorithm on each dataset), β_a, where n represents the number of datasets, v_i represents the value of the knapsack given by algorithm a on dataset i, and V_i represents the value of the knapsack provided by the algorithm that had the least valuable knapsack on dataset i.

$$\beta_a = \frac{1}{n}\sum_{i=1}^{n}\frac{v_i}{V_i} \qquad (5)$$

This was done so that we could analyze each algorithm on two axes: the time taken and the maximum value of items. Afterwards, we analyzed the trade-offs between the two axes.

Results and Discussion

Expected Results

Here, we qualitatively discuss our results on two axes, time taken and maximum value, and then predict which algorithm would have the fewest tradeoffs.

Maximum Value of Items

Based off the algorithms, we expect that "Max of All," followed by "Max of Two," will give the maximum value of items because they take the maximum from many different algorithms.

After that, we expect "Transitioning Greedy" and "Limited Greedy" to have the maximum value of items because they change the method of operation as the knapsack fills up. We expect the stingy algorithms to perform similarly to "Transitioning Greedy" as they keep as many items as possible instead of working up from nothing.

Those should be followed by "Scored Greedy," "Heavy Greedy," and "Defensive Greedy." The one algorithm that will give the worst results is "Sliding Threshold," as it does not sort.

Time Taken for Execution

Most of the greedy algorithms contain two full passes through the entire data (one for calculating scores and adding items to the sorting queue, and one for adding items to the knapsack) and a sort, with a small exception for "Defensive Greedy," whose second pass is only partial as the sorting guarantees that when one item does not fit into the knapsack, no remaining items will, allowing for a quick exit in that one. "Transitioning Greedy" adds another sort on a part of the data and two partial passes through the data, for calculating the scores and adding more items to the knapsack, after the transition from "Heavy Greedy" to "Scored Greedy."

Most of the stingy algorithms, however, contain one full pass through the data to calculate scores and add items to the knapsack, one sort, and one partial pass through the data to remove items from the knapsack.

The "Sliding Threshold" requires only one pass through all the data and no sort, while the composite algorithms run many passes as they run multiple algorithms and find the maximum.

"Sliding Threshold" runs in $O(n)$ because it just passes through the data without sorting. In contrast, all the greedy and stingy algorithms run in $O(n \log n)$ because they contain passes through the data (all of which run in $O(n)$) and sorts of the data (all of which run in $O(n \log n)$ if a fast sorting algorithm like Merge Sort or Quicksort is used). As all the composite algorithms just run all of the other algorithms once, they run in $O(n \log n)$ as they have many $O(n \log n)$ or faster algorithms running together.

From this, the "Sliding Threshold" would take the least time, followed by the stingy algorithms and "Defensive Greedy." After the stingy algorithms come the non-transitioning greedy algorithms, "Transitioning Greedy," and finally, "Max of Two" and "Max of All." Table 1 summarizes the time complexity of all the heuristic algorithms discussed.

Table 1: Table that gives the time complexity of the three exact algorithms and each of the heuristic algorithms used here.

Algorithm	Time Complexity
Brute Force	$O(n!)$
Dynamic Programming Par-	$O(2^n)$
tition-Based	$O(2^n)$
Limited Greedy	$O(n \log n)$
Heavy Greedy De-	$O(n \log n)$
fensive Greedy Deal	$O(n \log n)$
Stingy	$O(n \log n)$
Weight Stingy	$O(n \log n)$
Sliding Threshold	$O(n)$
Scored Greedy	$O(n \log n)$
Transitioning Greedy	$O(n \log n)$
Standard Greedy	$O(n \log n)$
Max Of Two	$O(n \log n)$
Max Of All	$O(n \log n)$

Happy Medium (Balancing Time Taken with Value of Items)

We can see from our predictions that "Limited Greedy" would give a good enough value while having a decent time. In contrast, "Scored Greedy," "Heavy Greedy," and "Defensive Greedy" would have a slightly better time (as they have fewer comparisons) while also having worse results. All other algorithms would either take too long or would not give a good enough value.

Actual Results

All the metrics in this section are defined in the Methodology section. In this section, we analyze our results on the axes of time taken and maximum value, and then analyze each algorithm's tradeoffs.

Table 2: Performance table of all 11 heuristic algorithms measuring their accuracy and speed.

	Win Count	Percent Error from Optimal	Normalized Average Time	Normalized Average Value
Heavy Greedy	4	58.39	1.76	1.29
Limited Greedy	4	57.71	2.24	1.30
Sliding Threshold	2	55.22	1.83	1.69
Deal Stingy	1	54.41	4.71	2.40
Scored Greedy	4	46.85	3.47	3.13
Transitioning Greedy	0	46.34	5.49	3.19
Defensive Greedy	1	45.62	1.63	3.48
Weight Stingy	1	44.88	2.82	3.59
Standard Greedy	4	35.55	2.36	4.27
Max Of Two	7	33.86	3.92	4.38
Max Of All	29	33.71	25.86	4.39

Maximum Value of Items

If the number of times an algorithm achieved the maximum value relative to its peers is to be regarded as a measure of its quality, the composite algorithms are the best, as evidenced by "Max of All" having a "win count" (or number of times it got the maximum value) of 29 and "Max of Two" having a "win count" of 7, beating out the rest. By this logic, "Standard Greedy," "Limited Greedy," "Heavy Greedy," and "Scored Greedy" are the third best as each one of them has a "win count" of 4. After them, contrary to our expectations, comes "Sliding Threshold," with a "win count" of 2. In last place comes "Transitioning Greedy," with a "win count" of 0.

However, the "win count" might not be the best measure of algorithm quality. Taking the average percentage error from the optimal is a better measure of algorithm quality, where the best algorithm has the lowest average percentage error. Using this metric, "Max of All" and "Max of Two" come under 34% error of the optimal, while "Standard Greedy" comes under 36% error, making them the top 3 best algorithms. "Weight Stingy," "Defensive Greedy," "Scored Greedy," and "Transitioning Greedy" all come under 47% error, with "Heavy Greedy" having the largest error at over 58% error.

Time Taken for Execution

We used the mean normalized time to find the algorithms that took the least amount of time. This was lowest (below 2) for "Defensive Greedy," "Heavy Greedy," and "Sliding Threshold," while it was highest for "Deal Stingy," "Transitioning Greedy," and "Max of All". Surprisingly, "Defensive Greedy" and "Heavy Greedy" are better than "Sliding Threshold," even though they have a sort and running time $O(nlogn)$ as opposed to $O(n)$. Also, surprisingly, the stingy algorithms did not do better than "Heavy Greedy," even though they had fewer passes through the list. More investigation, as well as a better methodology and a better implementation of each algorithm, is needed.

Happy Medium (Balancing Time Taken with Value of Items)

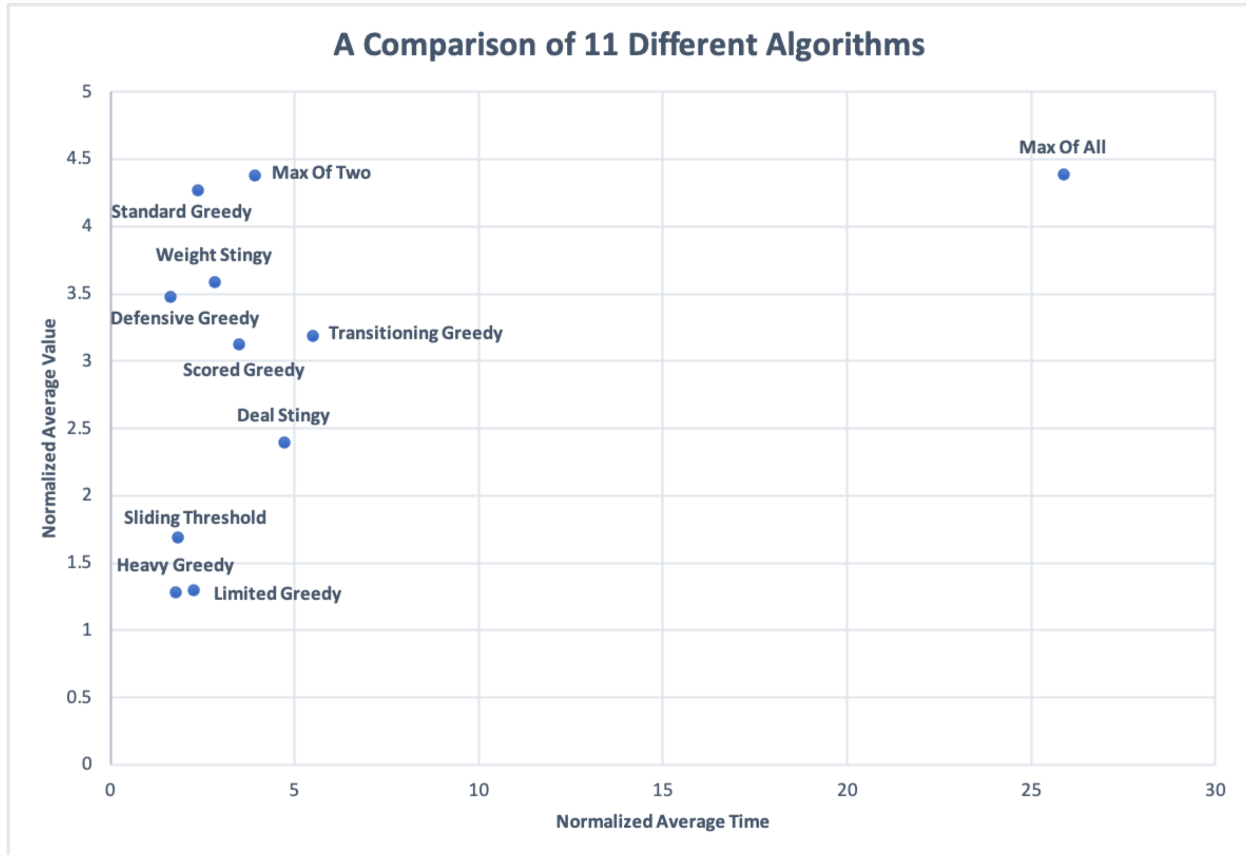

Figure 1: Scatterplot of all the 11 heuristic algorithms' normalized average values over their normalized average time

Figure 1 shows that the greatest advantage of "Max of All" is that it has the highest normalized average value. However, due to its disadvantage of having such a high normalized average time, 25.866, compared to the following most time-consuming algorithm, "Transitioning Greedy," which has a normalized average time of 5.492, it is not very practical.

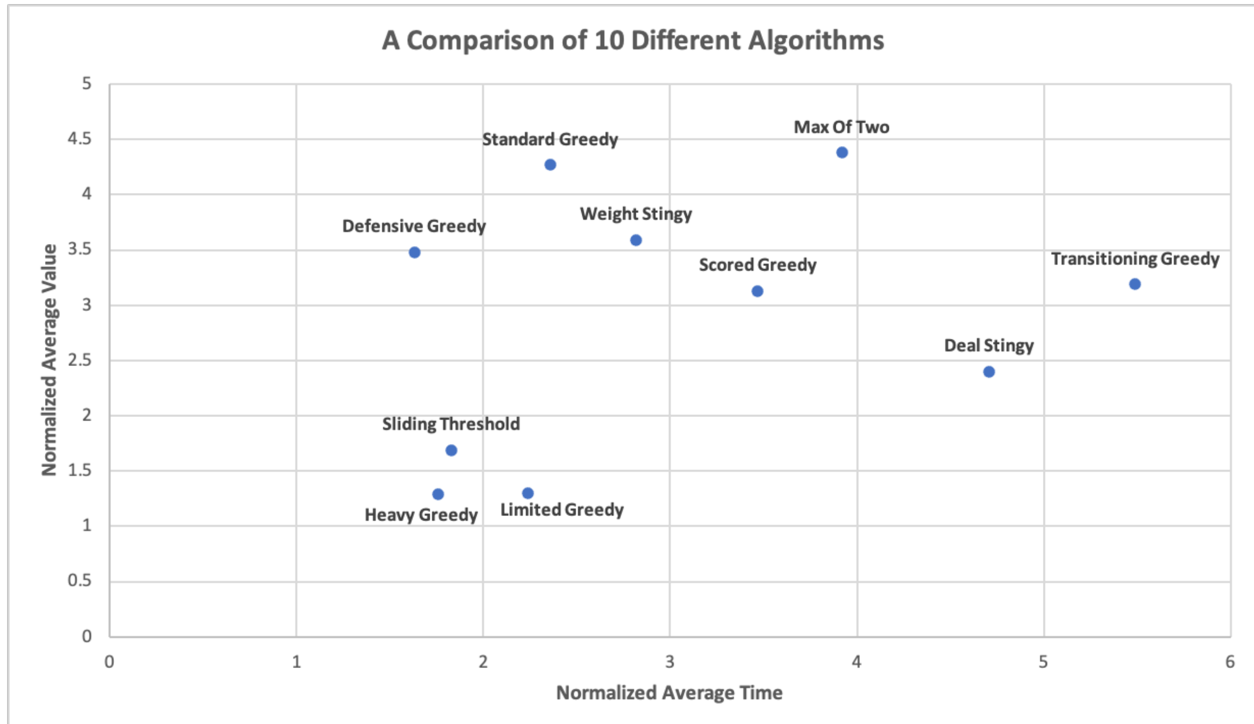

Figure 2: Scatterplot of 10 heuristic algorithms' normalized average value over their normalized average time, with outliers removed

If we remove "Max of All," the scatterplot looks much better and is much more useful. The main disadvantage of "Transitioning Greedy" is that it takes much time, with a normalized average time of over 5. However, its main advantage is that it performs somewhat well with a normalized average value of around 3.199. On the other hand, as "Transitioning Greedy" is outperformed for the normalized average value by "Max of Two" (4.387), "Weight Stingy" (3.597), "Standard Greedy" (4.271), and "Defensive Greedy" (3.482), all of which are faster than it, its advantage fades away. The same can be said about "Deal Stingy" (2.407) and "Scored Greedy" (3.133). On the contrary, the latter is faster than "Max of Two," having a normalized average of time of 3.470 while "Max of Two" has a normalized average time of 3.930, meaning it gains some advantage. Still, "Deal Stingy" does not even get to the same normalized average time (as its normalized average time is around 4.716), making its disadvantage worse.

Although "Max of Two" has one of the highest normalized average values (4.387) as an advantage, its main disadvantage is that it is 66.342% slower than "Standard Greedy," but the former only has a normalized average value 2.644% better than the latter. "Heavy Greedy," "Limited Greedy," and "Sliding Threshold" have a similarly low normalized average time, with "Heavy Greedy" having a normalized average time of 1.762, "Heavy Greedy" having a normalized average time of 2.241, and "Sliding Threshold" having a normalized average time of 1.836. This is an advantage for all three, but their Achilles heel is that they have a very low normalized average value (below 1.7 for each one), making them suitable only for saving time. From Table 2, we can find that "Defensive Greedy" comes under 46% error and is the fastest algorithm here, making it a good choice for instances where speed is a little more important. Also, "Standard Greedy" is the fourth-fastest algorithm here (and is only 44.819% slower than "Defensive Greedy") but comes at around 35% error, making it an extremely good choice for many other instances where accuracy is more important. Those two algorithms would represent the "happy medium" for most use cases as they are well-rounded based on these numbers suggesting above-average to superb performance on both axes.

Conclusion

Our main results are that "Defensive Greedy" is the best algorithm as far as time is concerned (and is 7.435% faster than the next-fastest algorithm) and that "Standard Greedy" is the third-best algorithm as far as giving the maximum value is concerned (with a normalized average value 2.910% lower than the algorithm with the highest such value, "Max of All"). Still, both are very well-rounded as they have above-average values and below-average times. To improve on this research, there are three main things to do: using more datasets, comparing the heuristics proposed here to the exact solutions, and comparing the heuristics proposed here to state-of-the-art heuristics like genetic algorithms.[8]

References

[1] Michail G. Lagoudakis. *The 0-1 knapsack problem – an introductory survey.* 1996. https://www.semanticscholar.org/paper/The-0-%E2%80%93-1-Knapsack-Problem-An-Introductory-Survey-Lagoudakis/6bd62e0ba7233c5086fe4b9061926d191894714b *(accessed 2022-07-13).*

[2] V. Reddy Dondeti and Bidhu B. Mohanty. *Impact of learning and fatigue factors on single machine scheduling with penalties for tardy jobs.* 1996. https://www.sciencedirect.com/science/article/abs/pii/S0377221797000702 *(accessed 2022-08-07).*

[3] Kayode Badiru. *Knapsack problems; methods, models and applications.* 2009. https://shareok.org/handle/11244/319274 *(accessed 2022-07-16).*

[4] Hans Kellerer, Ulrich Pferschy, and David Pisinger. *Knapsack Problems.* Springer, 2004. https://link.springer.com/book/10.1007/978-3-540-24777-7 *(accessed 2022-07-16).*

[5] MIT OpenCourseWare. *Lecture 17 complexity and np-completeness.* https://ocw.mit.edu/courses/6-046j-design-and-analysis-of-algorithms-spring-2012/b4562881f2af637e09e806450e9b62c8_MIT6_046JS12_lec17.pdf *(accessed 2022-07-14).*

[6] Paul Black. *Np-hard.* https://xlinux.nist.gov/dads/HTML/nphard.html *(accessed 2022-06-28).*

[7] Xinyang Wang, Yuncheng Jia, and Simin Li. *The complete proof of knapsack problem is np-completeness.* https://www.scribd.com/document/446317042/The-Complete-Proof-of-Knapsack-Problem-is-NP-Completeness *(accessed 2022-08-05)*

[8] Maya Hristakeva and Dipti Shrestha. *Different approaches to solve the 0/1 knapsack problem.* 2005. https://www.micsymposium.org/mics_2005/papers/paper102.pdf *(accessed 2022-07-13).*

[9] Ellis Horowitz and Sartaj Sahni. *Computing partitions with applications to the knapsack problem.* 1974. https://dl.acm.org/doi/10.1145/321812.321823 *(accessed 2022-08-03).*

[10] Vijay Vazirani. *Approximation Algorithms.* Springer, 2001. https://www.ics.uci.edu/~vazirani/book.pdf *(accessed 2022-07-12).*

[11] Paul E. Black. *greedy algorithm.* https://xlinux.nist.gov/dads//HTML/greedyalgo.html *(accessed 2022-07-04).*

[12] George B. Dantzig. *Discrete-variable extremum problems.* 1957. https://www.jstor.org/stable/167356 *(accessed 2022-07-27).*

[13] Johny Ortega. *Instances of 0/1 knapsack problem.* http://artemisa.unicauca.edu.co/~johnyortega/instances_01_KP/ *(accessed 2022-07-23).*

[14] John Burkardt. *Knapsack_ 01 data for the 01 knapsack problem.* https://people.sc.fsu.edu/~jburkardt/datasets/knapsack_01/knapsack_01.html *(accessed 2022-07-21).*