The Knapsack Problem: Optimizing Larceny

Alex Jackson[1]

[1]School of Information, Technology, and Computing, Abilene Christian University

Abstract

The Knapsack Problem is a classic problem in Computer Science, and involves maximizing the value of a set of stolen goods, given a total set of goods to steal and a total weight constraint. In this paper, I will detail some of the methods of calculating optimal and sub-optimal solutions to this problem, and discuss the effectiveness of each. I will discuss some of the constraints for approaching this problem, as well as techniques for optimizing searches through the state space.

Introduction

The Knapsack Problem is foundational in its importance to the study of algorithms, computational complexity, and artificial intelligence, and also has major relevance outside of pedagogy, in fields such as logistics, shipping, aerospace engineering, and of course, thieving. Part of the reason that this problem is so useful in the instruction of computer science concepts is the applicability of a number of different approaches for finding solutions. At different scales of knapsack sets, exhaustive searches, greedy algorithms, and others are all feasible methods for searching the state space. For small scale knapsack sets, exhaustive searches can be executed quickly and yield provably optimal results. For larger scale sets, greedy algorithms run quickly, and algorithms based on hill-climbing or simulated annealing techniques yield good results at a much lower resource cost.

Methodologies

Methods for Finding Solutions

Six methods of searching the knapsack state space were used:

- Greedy Algorithms
  - Value Motivated
  - Weight Motivated
  - Value to Weight Ratio Motivated
- Hill-Climbing Algorithm
- Exhaustive
- Exhaustive with Pruning

These algorithms were all written in Rust[1] to maximize performance and minimize time spent fixing bugs.

Implementation

Utilities I wrote several functions and a script to make my research easier. Firstly, for ergonomic purposes, I created a Knapsack struct and an Item struct. This allowed me to create some quality of life functions to, for example, get the total value and/or weight of the knapsack. Secondly, I used a library named Clap[2] to parse command line arguments, allowing me to pass multiple tests at once, while specifying the methods to use and a time cap, resulting in the ability to benchmark all the methods with one command. Finally, I wrote a small script in python to automate the creation of knapsack files, to enable easier testing.

Greedy Algorithms Since the greedy algorithms merely fit whatever possible into the knapsack based on some motivator/discriminant, it is beneficial to sort the list of items to eliminate the need to traverse the list multiple times. With the method of sorting being the only difference between the three greedy approaches, I decided to write one greedy algorithm, and sort the list according to the desired discriminant to achieve the different methods. The single function merely iterates through the list of items and steals as many as possible until the weight limit restricts further theft.

Exhaustive To avoid writing the exhaustive search using recursion, and any of the stack overflows that may result, I wrote this method to do an iterative search. Since every knapsack configuration can be expressed as a binary string, with each item being represented by a bit (1 = stolen, 0 = not), I decided to have a loop that iterated through all integers from 0 to the total number of states (2^n), and treated each integer as a knapsack configuration. The algorithm would iterate through all the bits set to one, sum up the weights and values, and store that binary string if it was the best configuration seen so far.

Exhaustive with Pruning This method was more suited to being written in a recursive fashion, as a depth-first recursive search would result in the ability to break from the recursion upon finding a configuration that was over-weight, resulting in that branch of the state space being "pruned." This worked by pass-

ing the current optimal solution, the configuration to test, and the index of the bit just toggled to the recursive call. Each iteration of the function would toggle the bit to the right of the last, thereby avoiding loops. If a branch was pruned, then "earlier" bits, or bits to the left, could be changed and tested.

Hill Climbing This method combines aspects of the fast/sub-optimal greedy algorithms, and the slow/optimal exhaustive searches, in that it searches through the state space, but not exhaustively. This algorithm starts with a random configuration, and moves through the state space to find an optimal solution. If the configuration is under-weight, it flips zeroes until it's as close to the weight as possible, either equal, or over. If the configuration is over-weight, it flips ones until it is as close to the weight as possible. If the weight is exactly at the limit, it swaps ones until a local maximum is found.

Results

As expected, the exhaustive methods almost always outperformed the others for smaller knapsacks. However, for the larger knapsacks, the exhaustive methods were sometimes outperformed by even the greedy algorithms, as their searches through the state space were uninformed, and if the optimal configuration was at the end of the search, when the exhaustive algorithms were capped to 20 minutes, they tended to return a poor configuration. The greedy algorithms held up surprisingly well, out-performing the exhaustive searches frequently (admittedly capped at 20 minutes), showing that a heuristic need not be complex or provably optimal to find good solutions in constant time (at least for this problem). One point to note when comparing the exhaustive and exhaustive with pruning methods, is that the exhaustive with pruning method has somewhat of an erratic (depending on the input) search path. If there are heavier items towards the beginning of the knapsack, in terms of the binary string, it will recurse deeply into the search tree, into the high binary numbers, and work it's way back up the tree and down the numbers, so the two exhaustive methods aren't guaranteed to have similar performance.

Overall, the Hill Climbing method seemed to perform the best on average, with the pruned exhaustive search and some of the greedy algorithms in a close second. If some optimizations were made to the exhaustive methods, they would likely be better performers, but this would only go so far, as no amount of optimizing could make a knapsack of 400 items quickly searchable. On my machine, with a knapsack of 30 items taking about 5 minutes to search

Table 1: *Sample of Exhaustive results (Capped at 20 Minutes)*

Total Items	Weight	Value	Items
3	6	8	2
15	48	67	3
20	50	260	10
25	50	127	9
30	50	265	11
40	50	312	12
50	50	324	6
75	50	333	11
100	200	255	16
200	200	343	21
400	200	289	18

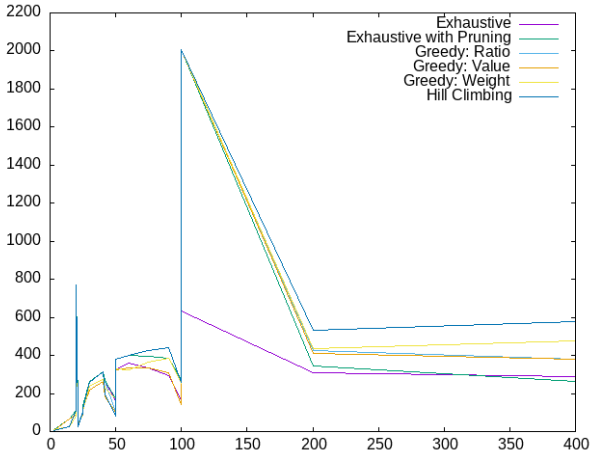

Figure 1: *Value of each solution per method*

Table 2: *Sample of Exhaustive with Pruning results (Capped at 20 Minutes)*

Total Items	Weight	Value	Items
3	6	8	2
15	48	67	3
20	50	113	8
25	50	136	9
30	50	265	11
40	50	312	12
50	50	382	14
75	50	395	14
100	250	1632	78
200	200	307	20
400	200	265	19

Table 3: *Sample of Greedy Ratio results*

Total Items	Weight	Value	Items
3	5	7	1
15	48	67	3
20	50	92	5
25	50	123	7
30	49	219	7
40	50	262	8
50	50	322	10
75	49	334	10
100	250	1672	73
200	197	427	22
400	197	380	19

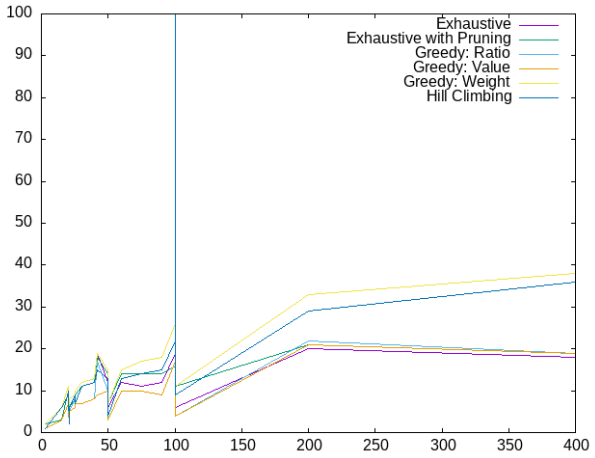

Figure 2: *Number of items in each solution per method*

Table 4: *Sample of Greedy Value results*

Total Items	Weight	Value	Items
3	5	7	1
15	48	67	3
20	50	92	5
25	50	92	6
30	49	219	7
40	50	262	8
50	50	322	10
75	49	334	10
100	250	1672	73
200	199	410	21
400	197	380	19

Table 5: *Sample of Greedy Weight results*

Total Items	Weight	Value	Items
3	6	8	2
15	39	25	6
20	50	250	11
25	49	114	10
30	49	240	12
40	48	275	13
50	50	329	15
75	50	363	17
100	249	1641	82
200	196	437	33
400	199	476	38

completely, it would take 2.28777 * 10^{106} years to completely search a state space of 2^{400}.

Discussion

Future Work

Many different methods for finding knapsack solutions remain. Simulated annealing searches would likely perform as well, if not better, than the hill climbing approach, and hybrids of the different methods would likely yield better and/or faster results as well. In a similar vein, if perfectly optimal solutions were desired, the exhaustive searches could be multi-threaded or even be written to run on highly parallel platforms.

References

[1] *Rust Programming Language*. URL: `https://www.rust-lang.org/`.

[2] *Clap: Command Line Argument Parser for Rust*. URL: `https://docs.rs/clap/4.1.6/clap/`.

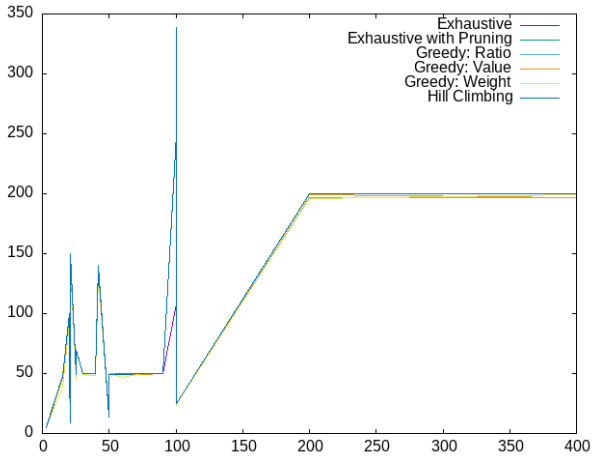

Figure 3: *Weight of each solution per method*

Table 6: *Sample of Hill Climbing results*

Total Items	Weight	Value	Items
3	5	7	1
15	47	23	6
20	50	113	8
25	50	123	7
30	50	258	11
40	50	312	12
50	49	382	13
75	50	426	14
100	249	1698	75
200	200	530	29
400	200	580	36