Different approaches to solving the 0-1 Knapsack problem

Grant Zvolský
zvolsgra@fit.cvut.cz

November 3, 2016

Abstract

This report summarizes the implementation of different approaches to solving the 0-1 Knapsack problem. A simple heuristic as well as a naive solution were implemented, tested and evaluated. The implementation language is Scala.

1 Problem specification

The Knapsack problem is an optimization problem. Let me quote Wikipedia for its definition.

> The knapsack problem or rucksack problem is a problem in combinatorial optimization: Given a set of items, each with a weight and a value, determine the number of each item to include in a collection so that the total weight is less than or equal to a given limit and the total value is as large as possible.[1]

In this paper we look into a variation of the problem where each item can be included at most once. This variation is known by the name 0-1 knapsack problem. In other words:

Let $M, n \in \mathbb{N}$ and $\forall i \in \{1, ..., n\} : v_i, m_i \in \mathbb{N}$.

Find $\{x_1, ..., x_n\} \in \{0, 1\}$ such that

$$\sum_{i=1}^{n} x_i w_i \leq M \quad \text{and} \quad \sum_{i=1}^{n} x_i v_i = MAX$$

2 Breakdown of different approaches

Solutions to the knapsack problem can be divided into these that guarantee finding the optimal solution, and these that try to approximate it. I developed three solutions while working on this project. The first is an approximative solution which uses a simple heuristic. The other two methods find the optimal solution by searching the whole state space.

3 Solutions

3.1 Value/Weight Ratio Heuristic

The following algorithm has the asymptotic computational complexity of the sorting algorithm it uses, which is $\mathcal{O}(n \log n)$ in our case.

Step 1: Sort the array of items by their value/weight ratio.
Step 2: Try adding items in this order. If an item is too large to fit, skip it.

That's all. The heuristic is very straightforward and yields surprisingly accurate results, see figure 3.

3.2 Naive Iteration

The knapsack can be seen as a vector of booleans, where each boolean signifies the presence of absence of a particular item. In a computer, an integer is also, technically, a vector of booleans. Therefore each knapsack configuration can be seen as a numeric value, and iterating from 0 to 2^n is analogous to iterating over all configurations. This naive solution seems quite elegant, but there's a catch. Iterating over individual bits of each configuration in order to calculate its weight and value has an asymptitic complexity of O(n). The overall asymptotic complexity of this approach thus $\mathcal{O}(n \cdot 2^n)$. Can we do better?

3.3 Naive Recursion

Another way to look at all the possible item combinations is a binary tree.

1. The root represents an empty knapsack.
2. Right children at level i+1 include item i.
3. Left children at level i+1 do not include item i.

Traversing such tree makes it easy to keep track of the weight and value of given node. No significant computations are required, which makes the worst case complexity the same as the number of nodes, $\mathcal{O}(2^n)$.

4 Implementation

I chose Scala as the implementation language for its simplicity. See scaladoc comments in the code for implementation details.

The project is structured into classes. Different algorithms are in separate files, e.g. NaiveIteration.scala. The following code listing is my most effective naive solution, from the file NaiveRecursionSansConfigVars.scala. It traverses the state space tree described in 3.3.

```
val items: Array[(Int, Int)] = ???
val capacity: Int = ???
var bestW, bestV = 0

def go(w: Int, v: Int, idx: Int): Unit = {
  if (w > capacity) return
  if (v > bestV) { bestV = v; bestW = bestW }
  if (idx == items.length) return
  go(w, v, idx + 1)
  go(w + items(idx)._1, v + items(idx)._2, idx + 1)
}
```

5 Results

The first graph demonstrates the performance of the Naive Recursion Sans Config Vars – a variation of the recursive solution which doesn't keep track of exactly which items are inside the knapsack. It shows the relationship between computation time and n, where n is the total number of items. As expected, it matches the 2^n reference curve.

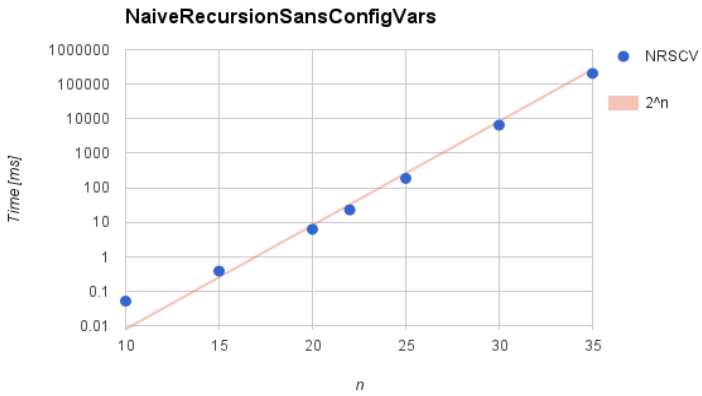

Figure 1: Naive Recursion

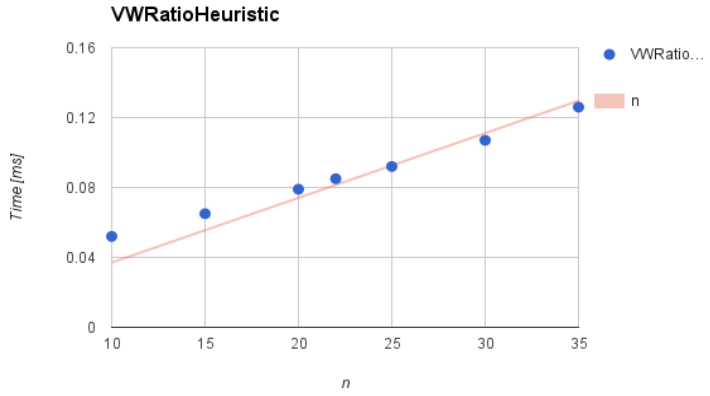

Figure 2: Value/Weight Ratio Heuristic

The following table shows how the solutions of the value/weight ratio heuristic compare to optimal solutions. The following formula was used to calculate the relative error of a signle solution:

$$\varepsilon = \frac{V(OPT) - V(APX)}{V(OPT)}$$

Where $V(OPT)$ is the value of the optimal solution and $V(APX)$ is the value of the optimal solution.

n	4	10	15	20	22	25	30	35	40
avgE	1.94%	1.43%	.49%	.54%	.69%	.72%	.72%	.50%	.53%

Figure 3: Average error of the V/W ratio heuristic.

n	4	10	15	20	22	25	30	35	40
maxE	24.7%	11.5%	8.54%	8.43%	7.23%	3.68%	5.51%	4.60%	2.34%

Figure 4: Maximal error of the V/W ratio heuristic.

As we can see, the heuristic was fairly successful at finding near-optimal solutions and its precision increases with the number of items.

6 Conclusion

The Naive Iteration algorithm turned out to perform much worse than recursion. On the other hand, the recursive algorithm and the heuristic performed to our expectations in regard to computational complexity.

The value/weight ratio heuristic yielded surprisingly good results, within 2% of the optimal solutions. Other methods, such as simulated evolution, may give even better approximations with little increase in time complexity.

References

[1] Knapsack problem. In: *Wikipedia* [online]. [vid. 12.10.2016]. Available at: `https://en.wikipedia.org/wiki/Knapsack_problem`