The Knapsack Problem

To take, or not to take

Joshua Pondrom

October 26, 2017

1 Motivation and Background

The knapsack problem is a simple optimization problem. While the idea behind the problem is simple, the solutions are not. The knapsack problem is a NP-Complete problem, which generally means the solution comes at a high cost of computing. However, we can find a approximate solution for cheaper using heuristics. This is good if we have less computing power, and do not care about being accurate, as long as it is close to the optimal.

This problem has its roots in multiple fields, not only computer science. Some of these are combinatorics, complexity theory, cryptography, and applied mathematics. Not only does it deal with multiple branches of study, it also has many different applications. In general, the knapsack problem applies to any one-dimensional constraint optimization problem. Some of these are pattern layout, networking, scheduling, selecting investments, and more. It is also good to note that some of these problems will be fine with the faster, heuristical, not-accurate algorithm, while some of the problems must have the exact, more-expensive, algorithm.

2 Procedures

In this section, there will be pseudocode for both the greedy algorithm, and the dynamic programming algorithm. Here is the dynamic programming method:

```
Precondition:   W is space left, w is a list of weights, v is a list of weights
                n is the value of the bag currently.
                w, and v should be non-empty, and have the same cardinality
                W should be a non-negative value
procedure Knapsack(Int:W, List:w, List:v, Int:n)
        #This is a dynamic programming solution so we store already
        # computed values in a list
        List:V
        #Initialize array to 0 where weight is 0
        for(w = 0 to W)
                V[0][w] = 0
        for(i = 1 to n)
                Assert w[0..i] = Knapsack(W, w[0..i], v, n)
                for(j = 0 to W)
                        Assert w[0..1] = Knapsack(W, w[0..i-1], v, n)
                        if(w[i] <= j)
                                V[i][j] = max(V[i-1,j],
                                              v[i] + V[i-1][j-w[i]])
                        else
                                V[i][j] = V[i-1, j]
        return V[n,W]
```

For the greedy solution the algorithm can either sort by the weight, taking the smallest weights first, or sorting by value, and taking the greatest value first. The greedy solution is as follows:

```
procedure Knapsack(Int:W, List:w, List:v, Int:n)
        #Sort will sort by weight ascending and done by a heapsort
        sort(w,v)
        Assert w is sorted
        while(W <= w[0], and w and n have elements remaining)
                W = W - w.pop
                n = n + v.pop
                Assert W is non-negative
                Assert w and v have 0 or more elements
        return W
```

3 Implementation and Testing

These can both be implemented in any language, however, I think python would be a good fit. Python allows for focus on the algorithms without having to deal with programming 'on the metal'. As an additional bonus, I have already implemented heap sort in python for the previous assignment.

Invariants can be implemented by using the build in assert feature of python. It can check correctness of the current state of the programming every time it runs through the major loops.

Data can be fed to the problem through command line interfaces, or be generated by the program itself by using a random command or generating data in a predicatable way. We can time the program using the Unix tool 'time' to see how long the program takes to process different sets of data. If needed, we can also plot the data using python to show what happens with the data sets as the algorithm runs.

To see the testing look at the annotated data results and code on the following pages.