The Knapsack Problem:
An Implementation in the Furman Dining Hall

Andrew Emerson
Department of Computer Science, Furman University, Greenville, SC

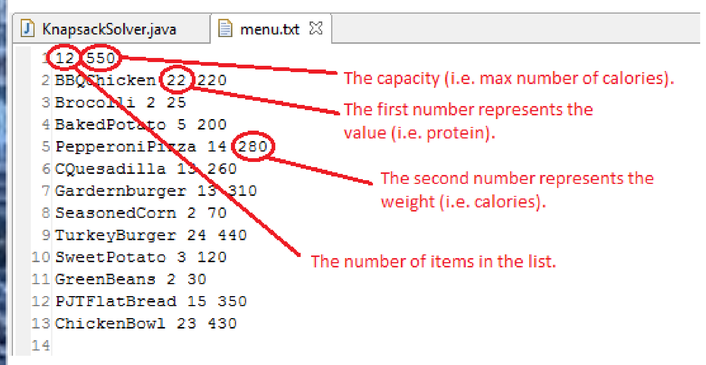
Introduction

Oftentimes, people choose a meal based on a particular goal. For example, a person may want to maximize the amount of protein they eat in a meal while not surpassing a set number of calories consumed. This study investigates this type problem by formulating the problem as a 0-1 Knapsack Problem, which can be solved using a strategy called dynamic programming.

What is Dynamic Programming?

Solving the 0-1 Knapsack Problem with Dynamic Programming can be thought of separating the overall problem into multiple sub-problems. Once the sub-problems have been solved, the solutions can be compared to see which one is optimal. In terms of the optimal meal problem, this would consist of comparing a small number of foods at a time.

Problem Parameters



The Furman University Dining Hall uploads its daily menu on a website, where it lists nutritional values for each item. After extracting the appropriate data to a text file, dynamic programming may be used to optimize the items. In the above text file, the list of food items also lists the protein and caloric content (for one serving). In order to comply with a 0-1 formulation of the Knapsack Problem, the optimized solution will determine whether or not to take a single serving of a given food item.

Code/Algorithm

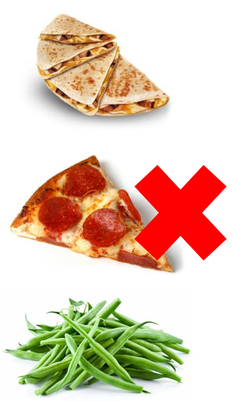
KnapsackSolver.java | menu.txt

```
1  import java.io.*;
2
3  /**
4   * @author Andrew Emerson - Furman University
5   * Seminar in Computer Science Spring 2016
6   * Description: Knapsack problem implementation in the Furman Dining Hall
7   *  All data comes from the DH Menu;
8   *  Takes in N food items with their appropriate nutritional value & weight;
9   *  User sets desired nutritional value W (weight);
10  *  Optimizes what items the user should take.
11  */
12 public class KnapsackSolver {
13     static int N = 0;          //Number of items
14     static int W = 0;          //Max weight desired (i.e. calories)
15     static String[] names;     //Food items
16     static int[] values;       //Respective nutritional values (i.e. protein content)
17     static int[] weights;      //Respective nutritional weights (i.e. calories, fats...
18
19     public static void main(String[] args) {
20         String fileName = "menu.txt";
21
22         //Reads text file to store input
23         try {
24             FileReader fileReader = new FileReader(fileName);
25             BufferedReader bufferedReader = new BufferedReader(fileReader);
26             String currLine = bufferedReader.readLine();
27             String[] firstLineResults = currLine.split(" ");
28             N = Integer.parseInt(firstLineResults[0]);
29             W = Integer.parseInt(firstLineResults[1]);
30             names = new String[N+1];
31             values = new int[N+1];
32             weights = new int[N+1];
33             currLine = bufferedReader.readLine();
34             int counter = 1;
35             while (currLine != null) {
36                 String[] lineResults = currLine.split(" ");
37                 names[counter] = lineResults[0];
38                 values[counter] = Integer.parseInt(lineResults[1]);
39                 weights[counter] = Integer.parseInt(lineResults[2]);
40                 counter++;
41                 currLine = bufferedReader.readLine();
42             }
43             bufferedReader.close();
44         }
45         catch (IOException e) {
46             System.out.println("File not found.");
47             e.printStackTrace();
48         }
49
50         //Algorithm to solve
51         // opt[i][j] = max profit of packing items 0..i-1 with weight limit j
52         // sol[i][j] = does opt solution to pack items 0..i-1 with weight limit j include item i?
53         int[][] option = new int[N+1][W+1];
54         boolean[][] solution = new boolean[N+1][W+1];
55
56         for (int i = 1; i <= N; i++) {
57             for (int j = 1; j <= W; j++) {
58
59                 // don't take item i
60                 int option1 = option[i-1][j];
61
62                 // take item i
63                 int option2 = Integer.MIN_VALUE;
64                 if (weights[i] <= j) option2 = values[i] + option[i-1][j-weights[i]];
65
66                 // select better of two options
67                 option[i][j] = Math.max(option1, option2);
68                 solution[i][j] = (option2 > option1);
69             }
70         }
71
72         // determine which items to take
73         boolean[] take = new boolean[N+1];
74         for (int i = N, j = W; i > 0; i--) {
75             if (solution[i][j]) { take[i] = true;  j = j - weights[i]; }
76             else { take[i] = false; }
77         }
78
79         // print results
80         System.out.println("Item" +"\t"+"\t" + "Value" + "\t" + "Weight" + "\t" + "Take");
81         for (int i = 1; i <= N; i++) {
82             System.out.println(names[i] +"\t" + values[i] + "\t" + weights[i] + "\t" + take[i]);
83         }
84     }
85 }
```

After reading the text file with the Dining Hall menu information, the above program implements a dynamic programming algorithm to find the optimal combination of foods. Essentially, the algorithm decides if it is a better option to take a given food item rather than leaving it. By determining the max profit of taking a particular item, the program determines if another item may produce a better value for its respective weight.

Sample Result

Item	Value	Weight	Take
BBQChicken	22	220	true
Broccoli	2	25	true
BakedPotato	5	200	false
PepperoniPizza	14	280	false
CQuesadilla	13	260	true
Gardernburger	13	310	false
SeasonedCorn	2	70	false
TurkeyBurger	24	440	false
SweetPotato	3	120	false
GreenBeans	2	30	true
PJTFlatBread	15	350	false
ChickenBowl	23	430	false



Shown above are the results of the sample text file discussed previously. On a day where the above items are served at the Dining Hall, a person wanting to maximize protein intake while not surpassing 550 calories would ideally like to eat one serving of the BBQ chicken, broccoli, cheese quesadilla, and green beans. With a much larger selection of food, the sample result might be different. Notice that the calories of the selected foods add up to 535, which has more room to increase before reaching the pre-determined maximum of 550.

Conclusion

Turkey Burger

Char-grilled turkey burger on a kaiser roll

500

Serving Size 1 each

Amount Per Serving	
Calories 440	
Calories From Fat 150	
Total Fat	17 g
Saturated Fat	4.5 g
Trans Fat	0 g
Cholestrol	90 mg
Sodium	760 mg
Total Carbohydrate	44 g
Dietary Fiber	0 g
Sugars	2 g
Protein	24 g

What is the point of applying dynamic programming to optimizing different aspects of a meal? The point is that we can apply dynamic programming to many different applications, food optimization being one of them. For instance, instead of looking at protein and calories, the algorithm could maximize fiber intake while not surpassing a certain carbohydrate intake. Essentially, this investigation shows that it is possible to formulate food optimization into a 0-1 Knapsack Problem.

Further Study

For further investigation, it would be possible to study the application of dynamic programming food optimization with more parameters. For instance, what if somebody wanted to maximize their protein and carbohydrate intake, but not surpass a fat intake? A future study could determine techniques to formulate and solve this problem in terms of the Knapsack Problem. Another interesting investigation would be studying different optimization techniques in relation to this problem, such as the branch and bound method.