Walker Mitchell
Kyle Short
CS361
6/12/2019

CS361 Final Project: 0-1 Knapsack

Part 1: Choosing the Problem

This task was to select an NP-Complete problem that has an associated approximation algorithm. We were intrigued by the 0-1 knapsack problem when it was presented during lecture, so we chose to focus on it for our project.

Part 2: Researching the Problem

This task was to thoroughly research the problem using at least three resources, and then answer these questions:

- **What is your NP-complete problem? Give specific examples.**
  The 0-1 knapsack problem. From Wikipedia: “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. It derives its name from the problem faced by someone who is constrained by a fixed-size knapsack and must fill it with the most valuable items.”
  There are multiple real-world examples of the knapsack problem, including “finding the least wasteful way to cut raw materials [and] selection of investments and portfolios.”
- **How do we know your problem is NP-complete? Did someone prove it?**
  There are multiple sources available providing a proof of the 0-1 knapsack problem’s NP-completeness, including the listed source from Cornell. However, we were unable to find a source detailing who first proved that it is NP-complete.
- **Why is your problem important? How does it work, and who invented it?**
  From the Tripod source: “The knapsack problem arises whenever there is resource allocation with cost constraints. Given a fixed budget, how do you select what things to buy? Everything has a cost and value, so we seek the most value for a given cost.”
  According to Wikipedia, “the knapsack problem has been studied for more than a century, with early works dating as far back as 1897. The name ‘knapsack problem’

dates back to the early works of mathematician Tobias Dantzig and refers to the commonplace problem of packing the most valuable or useful items without overloading the luggage."

- **Why is the approximation algorithm good enough? Do we know how far off it can be from the optimal solution?**

  The approximation algorithm calculates all possible combinations of the items, immediately discarding all combinations with a weight larger than the knapsack's capacity. It then selects and returns the combination with the greatest value. It is good enough because it runs in pseudopolynomial time since "the dynamic programming solution gives a running time dependent on a value, i.e. O(nW), where W is a value representing the max capacity." (StackExchange)

  According to Wikipedia, while the 'decision' form of the knapsack problem ("Can a value of at least V be achieved without exceeding the weight W?") is NP-complete, "the optimization problem is NP-hard, [meaning] its resolution is at least as difficult as the decision problem." Further, "there is no known polynomial algorithm which can tell, given a solution, whether it is optimal (which would mean that there is no solution with a larger V, thus solving the NP-complete decision problem)."

- **What is the runtime complexity of your approximation algorithm? How do you know?**

  The time complexity of the approximation algorithm we used is generally represented as O(nW) where n is the number of items and W is the capacity of the knapsack. However, the knapsack problem is NP-complete because, according to Quora, "using W in big O notation should in itself raise red flags because W is just a value of one of the variables in input and has nothing to do (directly) with size of input. Thus, the correct way to say complexity of knapsack would be O(n*2^m) where m is the number of bits in W." Therefore, in the worst case our approximation algorithm would actually run in exponential time.

Part 3: Coding the Approximation Algorithm, Testing, Analyzing.

```
/*
Kyle Short
Walker Mitchell
CS 361 Final Project
Below is our approximation algorithm for the 0-1
knapsack problem based off of
https://www.geeksforgeeks.org/0-1-knapsack-problem-dp-10/
*/
#include<stdio.h>
#include <iostream>

using namespace std;

// A function needed to find the max value between two integers
int max(int a, int b) { return (a > b)? a : b; }
```

```
/* A function that returns the maximum value that can be put in the knapsack
with a capacity of W. val[] is an array of values, wt[] is an array of the
weights of items.
*/
int knapSack(int W, int wt[], int val[], int n)
{
   int i, w;
   int K[n+1][W+1]; // table K is used for storing results to reference

   // Build table K[][] in bottom up manner
   for (i = 0; i <= n; i++)
   {
       for (w = 0; w <= W; w++)
       {
           if (i==0 || w==0)
               K[i][w] = 0;
           else if (wt[i-1] <= w)
                 K[i][w] = max(val[i-1] + K[i-1][w-wt[i-1]],  K[i-1][w]);
           else
                 K[i][w] = K[i-1][w];
       }
   }
```

```
int main()
{
    int val[] = {30, 50, 60};        //array of values
    int wt[] = {5, 10, 15};            // array of weights
    int  W = 25;                       // Total capacity of knapsack
    int n = sizeof(val)/sizeof(val[0]); //
    int maxValue = knapSack(W, wt, val, n);
    cout << "Maximum Value for knapsack of weight " << W << " is: " << maxValue;
    return 0;
}
```

Input 1:

```
int val[] = {120, 200, 240};
int wt[] = {20, 40, 60};
int  W = 100;
```

Output 1:

```
C:\Users\kyleshort\Desktop>Final
Maximum Value for knapsack of weight 100 is: 440
```

Input 2:

```
int val[] = {30, 50, 60};
int wt[] = {5, 10, 15};
int  W = 25;
```

Output 2:

```
C:\Users\kyleshort\Desktop>Final
Maximum Value for knapsack of weight 25 is: 110
```

Input 3:

```
int val[] = {700, 200, 1200, 100, 50};
int wt[] = {500, 100, 1500, 100, 650};
int  W = 2000;                          //
```

Output 3:

```
C:\Users\kyleshort\Desktop>Final
Maximum Value for knapsack of weight 2000 is: 1900
```

Our code was based heavily off of the sample code in the GeeksforGeeks source, and so we can reasonably assume that it has similar runtime complexity, namely O(nW). This code was derived directly from the well-known memoized DP approximation algorithm.

References

https://www.geeksforgeeks.org/0-1-knapsack-problem-dp-10/

https://en.wikipedia.org/wiki/Knapsack_problem

http://www.swatijain.tripod.com/knapsack2.htm

https://www.quora.com/Why-is-the-Knapsack-problem-NP-complete-even-when-it-has-complexity-O-nW

https://cs.stackexchange.com/questions/909/knapsack-problem-np-complete-despite-dynamic-programming-solution

https://people.orie.cornell.edu/dpw/orie6300/Lectures/lec25.pdf