PROJECT #2

Comp 482

Hrag Ayvazian

11/10/17

Before we can talk about performance of what the Greedy Approximation is or how it works we first need to understand what the Greedy Approximation method is. The Greedy Approximation method is a Greedy algorithm in which it is designed paradigm involves repeatedly making choices that optimize some objective function. An example of this in a knapsack problem is when we are given a set of n items, in which each value has its own weight and benefit. We are interested in choosing the set of items that will maximize our total benefit while not going over the max weight capacity of the knapsack. The greedy method is used to optimization problems that search through a given set of configurations to find one that maximizes or minimizes an objective function.

In project two I implemented the Greedy Approximation by first creating a double variable to temporally hold all the conversions. Then second, I went through the array and dividing the weights and benefits to find the values for all the numbers in the array and stored it in a new double array. I used double because it will get us a higher point of accuracy for future calculations rather than using integers and missing out on the decimals in which can play a big role. Then after everything is stored in the new double array we go through the array and find the largest value by running it through a while loop until the array reaches the length. After that we go through that array again to find the weight total and benefit total for all the values that in the most optimal set.

```
while (count != values.length) {
   if ((TotalWeight + w[index]) <= W) {
      TotalBenefit += b[index];
      TotalWeight += w[index];
      positions[num] = index;
      num ++;
   }
}
```

For example, looking at a portion of my code above this code is working by going checking to make sure count is going to be less than the length of value. After all that we check to make sure that the total weight and weight at the certain index is less than the total max weight. After it goes through that if statement and it's true it accepted and adds the total benefit and total weight and keeps it sorted. Also, I stored the position of the index that the optimal was located at so I can refer back to it later. Furthermore, I save all the points of where it was the optimal set so I can refer back to it later so it's easier for me to access when I have to print it. After all this I printed out an array of all the values that are the optimal set and the weight total and benefit total.

Test #1

Brute Force Solution
Optimal Set = { 4, 7 } weight sum = 95 benefit sum = 200
Optimal Set = { 2, 7 } weight sum = 95 benefit sum = 200

Dynamic Programming Solution
Optimal Set = { 2, 7 } weight sum = 95 benefit sum = 200

Greedy Approximate Solution
Optimal Set = { 1 } weight sum = 60 benefit sum = 180

Test #2

Brute Force Solution
Optimal Set = { 1, 2, 6, 9, 10, 11, 13, 18, 15, 18 } weight sum = 39 benefit sum = 115
Optimal Set = { 1, 2, 3, 6, 9, 10, 11, 12, 15, 18 } weight sum = 39 benefit sum = 115

Dynamic Programming Solution
Optimal Set = { 1, 2, 3, 6, 9, 10, 11, 18 } weight sum = 39 benefit sum = 114

Greedy Approximate Solution
Optimal Set = { 10, 11, 9, 1, 6, 3, 2, 18 } weight sum = 39 benefit sum = 114

Test #3

Brute Force Solution
Optimal Set = { 7, 9, 10, 16, 18, 19, 20 } weight sum = 29 benefit sum = 151

Dynamic Programming Solution
Optimal Set = { 7, 9, 10, 16, 18 } weight sum = 29 benefit sum = 151

Greedy Approximate Solution
Optimal Set = { 9, 16, 10, 18, 20, 14, 11 } weight sum = 28 benefit sum = 144

Test #4

Brute Force Solution

```
Optimal Set = { 4, 5, 6 } weight sum = 10 benefit sum = 40
Optimal Set = { 3, 4, 5 } weight sum = 10 benefit sum = 40

Dynamic Programming Solution
0 0 0 0 0 0 0 0 0 0 0
0 0 5 5 5 5 5 5 5 5 5
0 0 5 5 5 10 10 15 15 15 15
0 0 5 5 5 10 10 15 15 15 20
0 0 20 20 25 25 25 30 30 35 35
0 0 20 20 25 25 25 35 35 40 40
0 0 20 20 25 25 25 35 35 40 40
0 0 20 20 25 25 25 35 35 40 40
Optimal Set = { 1, 4, 5 }  weight sum = 9  benefit sum = 40

Greedy Approximate Solution
Optimal Set = { 4, 5, 1 } weight sum = 9 benefit sum = 40
```

```
/*
* Hrag Ayvazian
* Project 2  Comp 482
* 11/9/17
* */

import java.util.*;

public class Knapsack {
    //set private fields here
    private int W;
    private int[] w;
    private int[] b;

    public Knapsack(int W, int[] w, int[] b) {
        //constructor
        this.W = W;
        this.w = w;
        this.b = b;
    }

    public static int[] generateSubset(int k, int n) {
        //  0 <= k <= 2n - 1
        //  Generates the kth subset of { 0,1,...., n-1 }
        //  in the form of the binary representation of k
        int[] sub = new int[n];
        for (int i = n - 1; i >= 0; i--) {
            sub[i] = k % 2;
            k = k / 2;
        }
        return sub;
    }

    public void BruteForceSolution() {
        //Prints all optimal solutions to the 0-1 knapsack problem
        //using brute force algorithm described in Project 2
        //Print solution in format specified in Project 2
        int binLimit = (int) Math.pow(2, w.length);
        int[] binLimitArray = new int[binLimit];
        int weightTotal = 0;
        int benefitTotal = 0;
        int maxBenefit = 0;
        int maxWeight = 0;
        int[] pointer = new int[binLimit];
        int counter = 0;
        for (int a = 0; a < binLimit; a++) {
            binLimitArray[a] = a;
        }
        for (int i = 0; i < binLimit; i++) {
            weightTotal = 0;
            benefitTotal = 0;
            int[] subArray = generateSubset(binLimitArray[i], w.length);
            for (int s = 0; s < subArray.length; s++) {
                if (subArray[s] == 1) {
                    weightTotal += w[s];
                    benefitTotal += b[s];
                }
            }
            if (weightTotal >= maxWeight && weightTotal <= W) {
                if (benefitTotal >= maxBenefit) {
                    maxBenefit = benefitTotal;
                    if (weightTotal == maxWeight && maxWeight != 0) {
                        counter++;
                    }
                    pointer[counter] = i;
                }
                maxWeight = weightTotal;
            }
        }
        int[] arr = new int[w.length];
        for (int i = 0; i < pointer.length; i++) {
            if (pointer[i] != 0) {
                counter = 0;
                weightTotal = 0;
                benefitTotal = 0;
                int[] subArray = generateSubset(pointer[i], w.length);
                for (int s = 0; s < subArray.length; s++) {
                    if (subArray[s] == 1) {
                        weightTotal += w[s];
                        benefitTotal += b[s];
                        arr[counter] = s;
                        counter++;
                    }
                }
                if (weightTotal <= W && benefitTotal >= maxBenefit && weightTotal != 
0) {
                    System.out.print("Optimal Set = { ");
                    for (int p = 0; p < arr.length; p++) {
                        if (arr[p] != 0) {
                            System.out.print(arr[p]);
                            if (arr[p + 1] != 0) {
                                System.out.print(", ");
                            }
                        }
                    }
                    System.out.print(" } ");
                    System.out.println("weight sum = " + weightTotal + " " + "benefit 
sum = " + benefitTotal);
                }
            }
        }
    }

    public void DynamicProgrammingSolution(boolean printBmatrix) {
        //Prints one optimal solutions to the 0-1 knapsack problem
        // using dynamic programming algorithm described in Project 2
        // Print solution in format specified in Project 2
        // If printmatrix is true, print the OPT matrix.
        int WeightCol = W + 1;
        int BenefitRow = b.length;
        int[][] B = new int[BenefitRow][WeightCol];
        for (int k = 1; k < BenefitRow; k++) {
            for (int i = 1; i < WeightCol; i++) {
                if (i < w[k]) {
                    B[k][i] = B[k - 1][i];
                } else {
                    B[k][i] = Math.max(B[k - 1][i], B[k - 1][i - w[k]] + b[k]);
                }

            }
        }
        if (printBmatrix) {
            for (int p = 0; p < BenefitRow; p++) {
                for (int r = 0; r < WeightCol; r++) {
                    System.out.print(B[p][r] + " ");
                }
                System.out.println();
            }
        }
        int TrackRow = BenefitRow - 1;
        int TrackColumn = WeightCol - 1;
        int optimalbenefit = B[TrackRow][TrackColumn];
        int weightTrack = 0;
        int benefitTrack = 0;
        int count = 0;
        int[] backtrack = new int[BenefitRow];
        while (optimalbenefit != 0) {
            if (optimalbenefit == B[TrackRow - 1][TrackColumn]) {
                TrackRow--;
            } else {
                backtrack[count] = TrackRow;
                count++;
                weightTrack = weightTrack + w[TrackRow];
                benefitTrack = benefitTrack + b[TrackRow];
                TrackColumn = TrackColumn - w[TrackRow];
                TrackRow--;
            }
            optimalbenefit = B[TrackRow][TrackColumn];
        }
        System.out.print("Optimal Set = { ");
        for (int i = backtrack.length - 1; i >= 0; i--) {
            if (backtrack[i] != 0) {
                System.out.print(backtrack[i]);
                if (i - 1 >= 0) {
                    System.out.print(", ");
                }
            }
        }
        System.out.println(" }  weight sum = " + weightTrack + "  benefit sum = " +
benefitTrack);
    }

    public void GreedyApproximateSolution() {
        //Prints one approximate solution to the 0-1 knapsack problem using a
variation of
        // used to solve the Fractional Knapsack Problem.
        double[] values = new double[w.length];
        double divv;
        for (int i = 1; i < values.length; i++) {
            divv = (double) b[i] / (double) w[i];
            values[i] = divv;
        }
        int TotalWeight = 0;
        int index = 1;
        int TotalBenefit = 0;
        int count = 1;
        int[] positions = new int[w.length];
        int num = 0;

        double max = values[0];
        int m = 0;
        while (m < values.length) {
            if (max < values[m]) {
                max = values[m];
                index = m;
            }
            m++;
        }
        values[index] = 0;
        while (count != values.length) {
            if ((TotalWeight + w[index]) <= W) {
                TotalBenefit += b[index];
                TotalWeight += w[index];
                positions[num] = index;
                num ++;
            }
            max = values[0];
            m = 0;
            index = 1;
            while (m < values.length) {
                if (max < values[m]) {
                    max = values[m];
                    index = m;
                }
                m++;
            }
            values[index] = 0;
            count++;
        }
        System.out.print("Optimal Set = { ");
        for (int p = 0; p < positions.length; p++) {
            if (positions[p] != 0) {
                System.out.print(positions[p]);
                if (positions[p + 1] != 0) {
                    System.out.print(", ");
                }
            }
        }
        System.out.print(" } ");
        System.out.println("weight sum = " + TotalWeight + " benefit sum = " +
TotalBenefit);
    }
}
```

```
/*
* Hrag Ayvazian
* Project 2  Comp 482
* 11/9/17
* */

import java.util.*;

public class KnapsackDriver
{

    public static void main( String[] args)
    {
        //Testcase #1
        System.out.println("Test #1");

        int n = 7;
        int[] weights = {-1, 60, 50, 60, 50, 70, 70, 45};
        int W = 100;
        int[] benefits = {-1, 180, 95, 40, 95, 40, 40, 105};

        // Print input values as required in Project 2


        System.out.println("\nBrute Force Solution");
        Knapsack kp1 = new Knapsack(W, weights, benefits);
        kp1.BruteForceSolution();

        int [] weightss = {-1, 2, 4, 3, 4, 4, 1};
        int Ws = 10;
        int [] valuess = {-1, 1, 2, 3, 3, 3, 6};
        System.out.println("\nDynamic Programming Solution");
        Knapsack kp3 = new Knapsack(W,weights, benefits);
        kp3.DynamicProgrammingSolution(false);


        System.out.println("\nGreedy Approximate Solution");
        Knapsack kp4 = new Knapsack(W, weights, benefits);
        kp4.GreedyApproximateSolution();

        //Testcase #2
        System.out.println("\nTest #2");

        int n2 = 18;
        int[] weights2 = {-1,25,4,2,5,6, 2,7,8,2,1, 1,3,5,8,9,  6,3,2};
        int W2 = 39;
        int[] benefits2 = {-1,75,7,4,3,2,  6,8,7,9,6,  5,4,8,10,8,  1,2,2};


        System.out.println("\nBrute Force Solution");
        kp1 = new Knapsack(W2, weights2, benefits2);
        kp1.BruteForceSolution();


        System.out.println("\nDynamic Programming Solution");
        kp3 = new Knapsack(W2, weights2, benefits2);
        kp3.DynamicProgrammingSolution(false);
        System.out.println("\nGreedy Approximate Solution");
        kp4 = new Knapsack(W2, weights2, benefits2);
        kp4.GreedyApproximateSolution();

        //Testcase #3
        System.out.println("\nTest #3");

        int n3 = 20;
        int[] weights3 = {-1, 10,14,35,12,16, 20,13,7,2,4,
                3,10,5,6,17, 7,9,3,4,3};
        int W3 = 29;
        int[] benefits3 = {-1, 2,13,41,1,12, 5,31,2,41,16,
                2,12,1,13,4, 51,6,12,1,9};

        System.out.println("\nBrute Force Solution");
        kp1 = new Knapsack(W3, weights3, benefits3);
        kp1.BruteForceSolution();

        System.out.println("\nDynamic Programming Solution");
        kp3 = new Knapsack(W3, weights3, benefits3);
        kp3.DynamicProgrammingSolution(false);

        System.out.println("\nGreedy Approximate Solution");
        kp4 = new Knapsack(W3, weights3, benefits3);
        kp4.GreedyApproximateSolution();

        //Testcase #4
        System.out.println("\nTest #4");

        int n4 = 7;
        int[] weights4 = {-1, 2,5,3,2,5,3,7 };
        int W4 = 10;
        int[] benefits4 = {-1, 5,10,5,20,15,5,10};

        System.out.println("\nBrute Force Solution");
        kp1 = new Knapsack(W4, weights4, benefits4);
        kp1.BruteForceSolution();

        System.out.println("\nDynamic Programming Solution");
        kp3 = new Knapsack(W4, weights4, benefits4);
        kp3.DynamicProgrammingSolution(true);

        System.out.println("\nGreedy Approximate Solution");
        kp4 = new Knapsack(W4, weights4, benefits4);
        kp4.GreedyApproximateSolution();
    }
}
```