Muhammad Ansari

November 9th, 2017

COMP 482 Algorithm Design

Project #2

In this project we utilize three different algorithms for solving 0-1 Knapsack Problem. Among the three algorithms, we design a solution to the problem using a greedy method. In this method, we calculate each knapsack item’s *value* which is given by the knapsack’s benefit divided by its weight. Upon calculating the values, we then sort the knapsacks in decreasing-value order. Up until this point the greedy algorithm is essentially the same as the solution to the fractional knapsack problem. The inherent difference in the two algorithms lying in how the last item that can be added under the weight limit is treated. While in the fractional knapsack problem we add in a *fraction* of the final knapsack to allow us to reach the weight capacity provided in the problem, in the greedy method for the 0-1 Knapsack problem we cannot do this. Instead the final item is simply not added, which may or may not leave the weight capacity completely fulfilled. This algorithm, much like the fractional knapsack problem, finds most of its run time to come from the sorting method that is utilized to sort the array containing the values in descending order. Usual sorting methods include merge sort, quick sort, etc. I utilized merge sort in my algorithm which has a time complexity of O(n log(n)). Since the other steps of the algorithm are completed under n-time, the entire algorithm’s time complexity totals up to O(n log(n)).

Knapsack.java:

**public** **class** Knapsack {

**private** **int** n;

**private** **int** W;

**private** **int**[] weights;

**private** **int**[] benefits;

**public** **void** BruteForceSolution() {

**if** (weights[n-1] > W)

**new** Knapsack(W, weights, benefits);

}

**public** **void** DynamicProgrammingSolution(**boolean** printBmatrix) {

**int** table[][] = **new** **int**[n+1][W+1];

**boolean** optimalSet[][] = **new** **boolean**[n+1][W+1];

//Build table[][]

**for** (**int** i = 0; i < n; i++)

{

**for** (**int** j = 0; j < W; j++)

{

**if** (i==0 || j==0)

table[i][j] = 0;

**else** **if** (weights[i-1] <= j) {

table[i][j] = Math.*max*(benefits[i-1] + table[i-1][j-weights[i-1]], table[i-1][j]);

optimalSet[i][j] = ((benefits[i-1] + table[i-1][j-weights[i-1]]) > (table[i-1][j]));

}

**else**

table[i][j] = table[i-1][j];

}

}

**int** w = W;

**boolean**[] take = **new** **boolean**[n+1];

**for** (**int** i = n; i > 0; i--) {

**if** (optimalSet[i][w]) {

take[i] = **true**;

w = w - weights[i];

}

**else** {

take[i] = **false**;

}

}

System.***out***.println("Optimal Set = {" + take + "} weights sum = " + w + " benefit sum = " + table[n-1][W-1]);

}

**public** **void** GreedyApproximateSolution() {

**int** values[] = **new** **int**[n+1];

**for** (**int** i = 0; i < n; i++)

values[i] = benefits[i]/weights[i];

*mergeSort*(values);

System.***out***.println("Optimal Set = {");

**int** i = 0, weightSum = 0;

**while**(weightSum < W) {

weightSum += benefits[i]/values[i];

}

//+ take + "} weights sum = " + w + " benefit sum = " + table[n][W]);

}

//merges sorted slices a[i.. j] and a[j +1 ...k] for 0 <= i <= j < k < a.length

**private** **static** **void** merge(**int**[] a, **int** i, **int** j , **int** k) {

// Determine size of the two sub-arrays to be merged

**int** size1 = j - i + 1;

**int** size2 = k - j;

// Initialize temporary arrays

**int** left[] = **new** **int** [size1];

**int** right[] = **new** **int** [size2];

//Copy data into temporary arrays

**for** (**int** x = 0; x < size1; ++x)

left[x] = a[i + x];

**for** (**int** y = 0; y < size2; ++y)

right[y] = a[j + 1 + y];

// Merge the temp arrays

**int** x = 0, y = 0; // Initial indices of first and second sub-arrays

**int** z = i; // Initial index of merged sub-array

**while** (x < size1 && y < size2) {

**if** (left[x] <= right[y]) {

a[z] = left[x];

x++;

}

**else** {

a[z] = right[y];

y++;

}

z++;

}

// Copy remaining elements of left array

**while** (x < size1) {

a[z] = left[x];

x++;

z++;

}

// Copy remaining elements of right array

**while** (y < size2) {

a[z] = right[y];

y++;

z++;

}

}

//sorts a[i .. k] for 0 <= i <= k < a.length

**private** **static** **void** mergeSort(**int**[] a, **int** i , **int** k) {

**if** (i < k) {

**int** middle = (i+k-1)/2;

*mergeSort*(a, i, middle);

*mergeSort*(a, middle+1, k);

*merge*(a, i, middle, k);

}

}

//Sorts the array a using mergesort

**private** **static** **void** mergeSort(**int**[] a){

**int** k = a.length;

*mergeSort*(a, 0, k-1);

}

**public** Knapsack(**int** weightCapacity, **int**[] weightsArray, **int**[] benefitsArray) {

n = weightsArray.length;

W = weightCapacity;

weights = weightsArray;

benefits = benefitsArray;

}

}

KnapsackDriver.java:

//Muhammad Ansari

//COMP 482 Project 2

**import** java.util.\*;

**public** **class** KnapsackDriver {

**public** **static** **void** main( String[] args)

{

**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};

System.***out***.println("Test Case #1");

System.***out***.println("Knapsack Problem Instance");

System.***out***.println("Number of Items = " + n + " Knapsack Capacity = " + W);

System.***out***.println("Input weights: [");

**for**(**int** i = 0; i < weights.length; i++)

System.***out***.println(weights[i] + ", ");

System.***out***.println("\nInput benefits: [");

**for**(**int** i = 0; i < benefits.length; i++)

System.***out***.println(benefits[i] + ", ");

System.***out***.println("\nBrute Force Solution");

Knapsack kp1 = **new** Knapsack(W, weights, benefits);

kp1.BruteForceSolution();

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();

}

}