David Kopp

11-27-15

Comp496 ALG

Project #3: Knapsack Problem

Test Case Outputs:

Test Case#1

Number of Items = 6

Weight Capacity of Knapsack: 10

Brute Force Optimal Solutions (List all solutions):

Subset of Items = {1,3,4,6}; Subset Weight = 10; Subset Value = 13;

Subset of Items = {1,3,5,6}; Subset Weight = 10; Subset Value = 13;

Runtime (ms) = 0.150517

Dynamic Programming Optimal Solution (List one Solution):

Subset of Items = {1,3,4,6}; Subset Weight = 10; Subset Value = 13;

Runtime (ms) = 0.201181

OPT Matrix:

0 0 0 0 0 0 0 0 0 0 0

0 0 1 1 1 1 1 1 1 1 1

0 0 1 1 2 2 3 3 3 3 3

0 0 1 3 3 4 4 5 5 6 6

0 0 1 3 3 4 4 6 6 7 7

0 0 1 3 3 4 4 6 6 7 7

0 6 6 7 9 9 10 10 12 12 13

Test Case#2

Number of Items = 10

Weight Capacity of Knapsack: 12

Brute Force Optimal Solutions (List all solutions):

Subset of Items = {1,2,6,7,10}; Subset Weight = 12; Subset Value = 33;

Runtime (ms) = 1.1419

Dynamic Programming Optimal Solution (List one Solution):

Subset of Items = {1,2,6,7,10}; Subset Weight = 12; Subset Value = 33;

Runtime (ms) = 0.071683

Test Case#3

Number of Items = 20

Weight Capacity of Knapsack: 18

Brute Force Optimal Solutions (List all solutions):

Subset of Items = {3,9,10,12,15,16,17,20}; Subset Weight = 18; Subset Value = 40;

Runtime (ms) = 250.256709

Dynamic Programming Optimal Solution (List one Solution):

Subset of Items = {3,9,10,12,15,16,17,20}; Subset Weight = 18; Subset Value = 40;

Runtime (ms) = 0.178393

Test Case#4

Number of Items = 25

Weight Capacity of Knapsack: 100

Brute Force Optimal Solutions (List all solutions):

Subset of Items = {2,4,5,7,8,9,11,12,14,19,25}; Subset Weight = 100; Subset Value = 50;

Subset of Items = {2,4,5,6,7,8,9,11,14,15,19,25}; Subset Weight = 100; Subset Value = 50;

Runtime (ms) = 9759.527436

Dynamic Programming Optimal Solution (List one Solution):

Subset of Items = {2,4,5,7,8,9,11,12,14,19,25}; Subset Weight = 100; Subset Value = 50;

Runtime (ms) = 0.275683

**Run Time Analysis of Brute Force verses Dynamic Programming for the Knapsack Problem:**

When comparing brute force and dynamic programming for the knapsack problem one might assume that dynamic programming is faster. Upon further investigation one would find that when the knapsack weight capacity is very large and the number of items are small the runtime could be just the opposite. Take into consideration the graph below:

In the first four of the above examples the runtime for dynamic programming was higher than that of brute force. The brute force runtime is greatly dependent on the number of items. As the number of items increases so does the number of subsets that brute force has to check, which increases its runtime. Dynamic programming’s runtime is greatly influenced on both the number of items and the capacity of the knapsack. The main reason why the runtime increases while using dynamic programming is the OPT matrix starts to get very large. Generating the OPT matrix is the largest time sink for dynamic programming.

In conclusion, brute force is not always the slowest and dynamic programming is not always the fastest methods of programmatic problem solving. If the numbers of subsets is low enough and the weight of the knapsack is high enough, then brute force will have a faster runtime than that of dynamic programming. Dynamic programming is much faster than brute force when the number of items is increased. Look at the above graph for the examples of when n = 20 and n = 22, and W = 10000000. One can see that as the number of items increase between the two examples dynamic programming’s runtime starts to increase at a lower rate than that of brute force, which is increasing exponentially. If we increased n by one to n = 23 then dynamic programming would have a faster runtime than that of brute force. (See the last example in the above graph)

Source Code:

/\*

\* Programmer: David Kopp

\* Project #: Project 3

\* File Name: TestKSP.java

\* Date: 11-24-15

\* Class: Comp496ALG

\* Description: This class is the main driver for the BFS and DPM solutions to the

\* knapsack problem.

\*/

**package** project3;

**public** **class** **TestKSP** {

// Test Case #1 Data

**final** **static** **int** ***N1*** = 6;

**final** **static** **int** ***W1*** = 10;

**final** **static** **int**[] ***WEIGHTS1*** = {2,4,3,4,4,1};

**final** **static** **int**[] ***VALUES1*** = {1,2,3,3,3,6};

// Test Case #2 Data

**final** **static** **int** ***N2*** = 10;

**final** **static** **int** ***W2*** = 12;

**final** **static** **int**[] ***WEIGHTS2*** = {3,4,2,5,6,1,2,7,8,2};

**final** **static** **int**[] ***VALUES2*** = {6,7,4,3,2,6,8,7,9,6};

// Test Case #3 Data

**final** **static** **int** ***N3*** = 20;

**final** **static** **int** ***W3*** = 18;

**final** **static** **int**[] ***WEIGHTS3*** = {2,3,4,2,6,5,3,7,2,4,3,1,5,6,2,1,1,3,4,3};

**final** **static** **int**[] ***VALUES3*** = {2,3,4,1,2,5,3,2,4,6,2,2,1,3,4,5,6,2,1,9};

// Test Case #4 Data

**final** **static** **int** ***N4*** = 25;

**final** **static** **int** ***W4*** = 100;

**final** **static** **int**[] ***WEIGHTS4*** = {9,16,12,8,7,14,7,8,9,14,15,18,20,2,4,5,10,11,3,17,15,18,15,9,7};

**final** **static** **int**[] ***VALUES4*** = {1,7,3,4,5,5,3,4,6,2,6,6,4,2,1,1,2,2,4,5,4,3,2,1,3};

***@SuppressWarnings***("unused")

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

// Problem from lecture

**final** **int**[] **v** = {2,4,5,3};

**final** **int**[] **w** = {3,2,6,4};

**BFSKSP** **test1** = **new** BFSKSP(v, w, 7, 4, "User Test1");

**DPMKSP** **test2** = **new** DPMKSP(v, w, 7, 4);

// Custom problem to show when dynamic programming takes long than brute force (W is very large, n is very small)

**final** **int**[] **v1** = {2,3,3,8,7,12,10,8,3,5};

**final** **int**[] **w1** = {2,3,6,9,11,12,20,7,1,30};

**BFSKSP** **faster** = **new** BFSKSP(v1, w1, 1000000, 10, "User Test2");

**DPMKSP** **slower** = **new** DPMKSP(v1, w1, 1000000, 10);

**final** **int**[] **v2** = {2,3,3,8,7,12,10,8,3,5,11,12,9,11,20,11,15,17,2,3};

**final** **int**[] **w2** = {2,3,6,9,11,12,20,7,1,30,20,40,14,19,22,12,18,2,3,30};

**BFSKSP** **faster2** = **new** BFSKSP(v2, w2, 10000000, 20, "User Test3");

**DPMKSP** **slower2** = **new** DPMKSP(v2, w2, 10000000, 20);

**final** **int**[] **v3** = {2,3,3,8,7,12,10,8,3,5,11,12,9,11,20,11,15,17,2,3,4,6};

**final** **int**[] **w3** = {2,3,6,9,11,12,20,7,1,30,20,40,14,19,22,12,18,2,3,30,20,43};

**BFSKSP** **faster3** = **new** BFSKSP(v3, w3, 10000000, 22, "User Test4");

**DPMKSP** **slower3** = **new** DPMKSP(v3, w3, 10000000, 22);

// Instructor Test Cases

**BFSKSP** **BFTestCase1** = **new** BFSKSP(***VALUES1***, ***WEIGHTS1***, ***W1***, ***N1***, "Test Case#1");

**DPMKSP** **DPTestCase1** = **new** DPMKSP(***VALUES1***, ***WEIGHTS1***, ***W1***, ***N1***);

**BFSKSP** **BFTestCase2** = **new** BFSKSP(***VALUES2***, ***WEIGHTS2***, ***W2***, ***N2***, "Test Case#2");

**DPMKSP** **DPTestCase2** = **new** DPMKSP(***VALUES2***, ***WEIGHTS2***, ***W2***, ***N2***);

**BFSKSP** **BFTestCase3** = **new** BFSKSP(***VALUES3***, ***WEIGHTS3***, ***W3***, ***N3***, "Test Case#3");

**DPMKSP** **DPTestCase3** = **new** DPMKSP(***VALUES3***, ***WEIGHTS3***, ***W3***, ***N3***);

**BFSKSP** **BFTestCase4** = **new** BFSKSP(***VALUES4***, ***WEIGHTS4***, ***W4***, ***N4***, "Test Case#4");

**DPMKSP** **DPTestCase4** = **new** DPMKSP(***VALUES4***, ***WEIGHTS4***, ***W4***, ***N4***);

}

}

/\*

\* Programmer: David Kopp

\* Project #: Project 3

\* File Name: BFSKSP.java

\* Date: 11-24-15

\* Class: Comp496ALG

\* Description: This class is the BFS for solving the knapsack problem. Subset permutations was found

\* at http://stackoverflow.com/questions/22280078

\* /how-to-write-iterative-algorithm-for-generate-all-subsets-of-a-set

\* (Modified by David Kopp)

\*/

**package** project3;

**import** java.util.ArrayList;

**public** **class** **BFSKSP** {

// Data Structures

**private** **ArrayList**<**ArrayList**<Integer>> solutions;

**private** **int**[] values;

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

// Variables

**private** **int** knapSackWeightCapacity;

**private** **int** numberOfItems;

**private** **int** currentMaxValue;

**private** **String** testLabel;

**private** **double** runtime;

// Main constructor for the Brute Force Solution for the Knapsack Problem

**public** **BFSKSP**(**int**[] values, **int**[] weights, **int** knapSackWeightCapacity, **int** numberOfItems, **String** testLabel) {

runtime = 0;

**long** **startTime** = **System**.*nanoTime*();

solutions = **new** ArrayList<**ArrayList**<Integer>>();

**int**[] **arr** = **new** **int**[numberOfItems];

makeItemArray(arr);

**this**.values = values;

**this**.weights = weights;

**this**.knapSackWeightCapacity = knapSackWeightCapacity;

**this**.numberOfItems = numberOfItems;

**this**.testLabel = testLabel;

currentMaxValue = 0;

allSubsets(arr);

**long** **endTime** = **System**.*nanoTime*();

runtime = (endTime - startTime) / **Math**.*pow*(10, 6); // Converts from ns to ms

generateStatistics();

}

// Sets the Array to items 1.....k (There is no item 0)

**private** **void** **makeItemArray**(**int**[] arr) {

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

arr[i] = i + 1;

}

}

// Generates all the Subsets found at:

// http://stackoverflow.com/questions/22280078/

// how-to-write-iterative-algorithm-for-generate-all-subsets-of-a-set (Modified by David Kopp)

**private** **void** **allSubsets**(**int**[] arr) {

**byte**[] **counter** = **new** **byte**[arr.length];

**while** (**true**) {

**ArrayList**<Integer> **temp** = **new** ArrayList<Integer>();

// Print combination

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

**if** (counter[i] != 0) {

//System.out.print(arr[i] + " ");

temp.add(arr[i]);

}

}

//System.out.println();

**int** **totalValue** = valueCalculation(temp);

// If you can put the subset into the knapsack without violating the constraints.

**if** (validWeight(temp) && totalValue >= currentMaxValue) {

**if** (totalValue > currentMaxValue) {

currentMaxValue = totalValue; // Set new max value

solutions.clear(); // Remove all subsets with the old max value

solutions.add(temp); // Add new subset with the new max value

}

**else** {

solutions.add(temp); // Add subset to list with same max value

}

}

temp = **null**;

// Increment counter

**int** **i** = 0;

**while** (i < counter.length && counter[i] == 1)

counter[i++] = 0;

**if** (i == counter.length)

**break**;

counter[i] = 1;

}

}

// Returns true or false depending on if the subset can fit in the bag or not

**public** **boolean** **validWeight**(**ArrayList**<Integer> arrayList) {

**int** **totalWeight** = 0;

**for** (**int** **i** = 0; i < arrayList.size(); i++) {

totalWeight += weights[arrayList.get(i) - 1];

}

**return** totalWeight <= knapSackWeightCapacity;

}

// Returns the value of the passed subset

**public** **int** **valueCalculation**(**ArrayList**<Integer> arrayList) {

**int** **total** = 0;

**for** (**int** **i** = 0; i < arrayList.size(); i++) {

total += values[arrayList.get(i) - 1];

}

**return** total;

}

// Returns the weight of the passed subset

**public** **int** **weightCalculation**(**ArrayList**<Integer> arrayList) {

**int** **total** = 0;

**for** (**int** **i** = 0; i < arrayList.size(); i++) {

total += weights[arrayList.get(i) - 1];

}

**return** total;

}

// Prints the passed subset

**public** **String** **printSolution**(**ArrayList**<Integer> list) {

**StringBuilder** **sb** = **new** StringBuilder();

**for** (**int** **i** = 0; i < list.size(); i++) {

**if** (i < list.size() - 1) {

sb.append(list.get(i) + ",");

}

**else** {

sb.append(list.get(i));

}

}

**return** sb.toString();

}

// Prints all the statistics that is required for the project's Rubric

**public** **void** **generateStatistics**() {

**System**.***out***.println(testLabel);

**System**.***out***.println(" Number of Items = " + numberOfItems);

**System**.***out***.println(" Weight Capacity of Knapsack: " + knapSackWeightCapacity);

**System**.***out***.println("\n Brute Force Optimal Solutions (List all solutions):");

**for** (**int** **i** = 0; i < solutions.size(); i++) {

**System**.***out***.println("\tSubset of Items = {" + printSolution(solutions.get(i)) + "}; Subset Weight = "

+ weightCalculation(solutions.get(i)) + "; Subset Value = " + valueCalculation(solutions.get(i)) + ";");

}

**System**.***out***.println("\tRuntime (ms) = " + runtime);

**System**.***out***.println();

}

}

/\*

\* Programmer: David Kopp

\* Project #: Project 3

\* File Name: DPMKSP.java

\* Date: 11-24-15

\* Class: Comp496ALG

\* Description: This class is the dynamic programming method to solving the knapsack

\* problem.

\*/

**package** project3;

**import** java.util.ArrayList;

**import** java.util.Collections;

**import** java.util.HashSet;

**import** java.util.List;

**public** **class** **DPMKSP** {

// Data Structures

**private** List<Integer> solutions;

**private** **int**[][] OPT;

**private** **int**[] values;

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

// Variables

**private** **int** knapSackWeightCapacity;

**private** **int** numberOfItems;

**private** **double** runtime;

// Main constructor for the Dynamic Programming Solution for the Knapsack Problem

**public** **DPMKSP**(**int**[] values, **int**[] weights, **int** knapSackWeightCapacity, **int** numberOfItems) {

runtime = 0;

**long** **startTime** = **System**.*nanoTime*();

solutions = **new** ArrayList<Integer>();

OPT = **new** **int**[numberOfItems + 1][knapSackWeightCapacity + 1];

**this**.values = values;

**this**.weights = weights;

**this**.knapSackWeightCapacity = knapSackWeightCapacity;

**this**.numberOfItems = numberOfItems;

generateOPTArray();

solutions = findSolution(numberOfItems, knapSackWeightCapacity);

**long** **endTime** = **System**.*nanoTime*();

runtime = (endTime - startTime) / **Math**.*pow*(10, 6);

generateStatistics();

}

// Generates the OPT table

**private** **void** **generateOPTArray**() {

**for** (**int** **i** = 1; i <= numberOfItems; i++) {

**for** (**int** **j** = 1; j <= knapSackWeightCapacity; j++) {

**if** (weights[i - 1] > j) {

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

}

**else** {

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

}

}

}

//**for** (**int** **i** = 0; i <= numberOfItems; i++) { // Prints out the OPT Table for debugging

// **for** (**int** **k** = 0; k <= knapSackWeightCapacity; k++) {

// **System**.***out***.print(OPT[i][k] + " ");

// }

// **System**.***out***.println();

//}

}

// Recursive method to find the optimal set of items picked from the OPT table

**private** List<Integer> **findSolution**(**int** k, **int** w) {

List<Integer> **temp** = **new** ArrayList<Integer>();

**if** (k == 0 || w == 0) { // If the bag is full or there is no more items to put into the bag

**return** temp;

}

**else** **if** (weights[k - 1] > w) { // If you did not pick item

**return** findSolution(k - 1, w);

}

**else** **if** (OPT[k - 1][w] >= values[k - 1] + OPT[k - 1][w - weights[k - 1]]) { // If you did not pick item

**return** findSolution(k - 1, w);

}

**else** { // Put item into solution set of items

temp.add(k);

**return** union(temp, findSolution(k - 1, w - weights[k - 1]));

}

}

// Returns the sorted union of two lists

**public** List<Integer> **union**(List<Integer> list1, List<Integer> list2) {

**HashSet**<Integer> **set** = **new** HashSet<Integer>();

set.addAll(list1);

set.addAll(list2);

List<Integer> **unsorted** = **new** ArrayList<Integer>(set);

**Collections**.*sort*(unsorted);

**return** unsorted;

}

// Returns the value of the passed subset

**public** **int** **valueCalculation**(List<Integer> arrayList) {

**int** **total** = 0;

**for** (**int** **i** = 0; i < arrayList.size(); i++) {

total += values[arrayList.get(i) - 1];

}

**return** total;

}

// Returns the weight of the passed subset

**public** **int** **weightCalculation**(List<Integer> arrayList) {

**int** **total** = 0;

**for** (**int** **i** = 0; i < arrayList.size(); i++) {

total += weights[arrayList.get(i) - 1];

}

**return** total;

}

// Prints all the statistics that is required for the project's Rubric

**public** **void** **generateStatistics**() {

**System**.***out***.println(" Dynamic Programming Optimal Solution (List one Solution):");

**System**.***out***.print("\tSubset of Items = {");

**for** (**int** **i** = 0; i < solutions.size(); i++) {

**if** (i < solutions.size() - 1) {

**System**.***out***.print(solutions.get(i) + ",");

}

**else** {

**System**.***out***.print(solutions.get(i));

}

}

**System**.***out***.print("}; Subset Weight = " + weightCalculation(solutions) + "; Subset Value = "

+ valueCalculation(solutions) + ";\n");

**System**.***out***.println("\tRuntime (ms) = " + runtime);

**System**.***out***.println();

}

}