BACKTRACKING ALGORITHM FOR THE 0-1 KNAPSACK PROBLEM

Student’s Name

Institutional Affiliation

Course Name

Professor’s Name

Date

BACKTRACKING ALGORITHM FOR THE 0-1 KNAPSACK PROBLEM

The knapsack problem is whereby the problem exists in a combinatory manner. A 0-1 Knapsack problem can be solved using several approaches. The most used approach to solve a knapsack problem is by considering using optimization methods. The optimization method is whereby the item profits are expanded in the knapsack without exceeding its capability as recommended by (Feng, 2017). Whenever a set of items have been stated and each of the items has got a weight and a value. The first activity to consider doing while given this is determining the number each item holds. This helps to ensure that the total weight realized gives a total value that is less than or equal to a specific limit.

Different algorithms can be used to solve a knapsack problem. The commonly used algorithms include greedy algorithm, brute force algorithm and dynamic programming. Greedy algorithm is whereby programming is implemented considering the optimization problems. However, dynamic programming is used where there are overlapping problems which are recurrence in nature. Dynamic programming records the solutions in a table and the problem is divided into sub problems and solved only once. There is always a straightforward approach method of solving a knapsack problem. This approach is referred to as a brute force algorithm, whereby the problem is forthwith based on its concepts involved and problem’s statement.

In conclusion, the optimal method of solving a knapsack problem is by using dynamic programming. This is whereby the knapsack problems existing in small sets are solved and then expanded into the bigger sets. To solve this, mostly the weight and value of the item exists in the form of an array. The array consists of rows and columns that are used to build an item. For instance, if we consider taking the 0th column. Means that the knapsack has got 0 capacity and therefore can hold nothing.

Code

package knapsack;

/\*A recursive implementation of 0-1 Knapsack problem \*/

import java.util.\*;//importing java libraries

class Knapsack {

// A utility function max that returns a maximum of two integers

static int max(int a, int b)

{

return (a > b) ? a : b;

}

// Returns the maximum value that can be put in a knapsack of capacity W

static int knapSack(

int W, int wt[],

int val[], int n)

{

// Base Case

if (n == 0 || W == 0)

return 0;

// If weight of the nth item is

// more than Knapsack capacity W,

// then this item cannot be included

// in the optimal solution

if (wt[n - 1] > W)

return knapSack(W, wt, val, n - 1);

// Return the maximum of two cases:

// (1) nth item included

// (2) not included

else

return max(

val[n - 1] + knapSack(W - wt[n - 1],

wt, val, n - 1),

knapSack(W, wt, val, n - 1));

}

// Driver program to test

// above function

public static void main(String args[])

{

Scanner s = new Scanner(System.in);

int val[] = new int[] { 6, 90, 78 };

int wt[] = new int[] { 11, 50, 38 };

int n = val.length;

int W;

System.out.println("Enter an integer value of the Knapsack Capacity W: ");

//this try catch block prohibits a user from entering a

//different value other than an integer for the Knapsack capacity.

try{

W= s.nextInt(); //The variable W is the Knapsack capacity.

System.out.println(knapSack(W, wt, val, n));

}

catch(InputMismatchException ex){

System.out.println("Please enter only an integer for value W!");

}

}

}

# References

Feng, Y. W. ( (2017)). *Solving 0–1 knapsack problem by a novel binary monarch butterfly optimization.* Neural computing and applications.