Knapsack Problem Solver

Authors: Innocent Chikwanda, Princess Asiru-Balogun, Arnold Aryeequaye

Greedy KnapSack

This Java program implements the greedy algorithm to solve the knapsack problem. The knapsack problem is a classic optimization problem where given a set of items, each with a weight and a value, determine the items to include in a knapsack of fixed capacity to maximize the total value while not exceeding the capacity.

Implementer

This code was implemented by Innocent Farai Chikwanda.

Usage

To use this program, follow these steps:

1. Compile the Java file: `javac GreedyKnapSack.java`
2. Run the compiled class file: `java GreedyKnapSack`

Explanation

The `GreedyKnapSack` class contains a method `choose` that implements the greedy algorithm for solving the knapsack problem. The algorithm repeatedly chooses the best local optimum at each step until the knapsack capacity is reached.

Algorithm Steps:

1. Calculate the value-to-weight ratio for each item.
2. Sort the items based on the value-to-weight ratio in non-increasing order.
3. Iterate through the sorted items and add the items to the knapsack until the capacity is reached.
4. Return the chosen items and the maximum value.

Example

The program provides two examples of solving the knapsack problem with different parameters. After running the program, it prints the chosen items and the maximum value obtained for each example.

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

        String[] header = { "element", "v", "w", "v/w" };
        // Example 1
        // {{element, value, weight, v/w}}
        double[][] Options1 = { { 1, 10, 2, 0 },
                { 2, 20, 5, 0 },
                { 3, 30, 10, 0 },
                { 4, 40, 15, 0 } };

        double MaxWeight1 = 25;

        GreedyKnapSack greedyKnapSack = new GreedyKnapSack(Options1,
MaxWeight1);
        ArrayList<Double> knapSack1 = greedyKnapSack.choose();

        System.out.println("\nFirst example");
        System.out.println();

        System.out.println(Arrays.toString(header));
        for (double[] opt : Options1) {
            System.out.println(Arrays.toString(opt));
        }

        System.out.println("\nThis is the choice of elements in the first
example");
        int a = 0;
        for (double choice : knapSack1) {
            if (++a < knapSack1.size())
                System.out.println("Element " + (int) choice + " was
chosen ");
        }
```

```
        System.out.println("Maximum value: " +
knapSack1.get(knapSack1.size() - 1));

        // Example 2
        // {{element, value, weight, v/w}}
        double[][] Options2 = { { 1, 5, 2, 0 },
                { 2, 21, 3, 0 },
                { 3, 10, 10, 0 },
                { 4, 40, 8, 0 } };

        double MaxWeight2 = 15;

        GreedyKnapSack greedyKnapSack2 = new GreedyKnapSack(Options2,
MaxWeight2);
        ArrayList<Double> knapSack2 = greedyKnapSack2.choose();
```

Output

```
First example

[element, v, w, v/w]
[4.0, 40.0, 15.0, 2.6666666666666665]
[3.0, 30.0, 10.0, 3.0]
[2.0, 20.0, 5.0, 4.0]
[1.0, 10.0, 2.0, 5.0]

This is the choice of elements in the first example
Element 1 was chosen
Element 2 was chosen
Element 3 was chosen
Maximum value: 60.0
```

Note

This program provides a basic implementation of the greedy algorithm for the knapsack problem. It may not always produce the optimal solution, especially for cases where the greedy choice does not lead to the global optimum. For more complex scenarios, alternative algorithms or optimizations may be required.

Dynamic Programming KnapSack

This Java program implements a tabulation (bottom-up) algorithm to solve the 0-1 knapsack problem. The 0-1 knapsack problem is a classic optimization problem in computer science, where given a set of items, each with a weight and a value, determine the items to include in a knapsack of fixed capacity to maximize the total value while not exceeding the capacity.

Implementer

This code was implemented by Princess Asiru-Balogun

Usage

To use this program, follow these steps:

1. Compile the Java file: `javac DPKnapSack.java`
2. Run the compiled class file: `java DPKnapSack`

Explanation

The `DPKnapSack` class contains a method `choice` that takes the knapsack capacity, an array of item values, and an array of item weights as input, and returns the maximum value that can be obtained. The algorithm uses dynamic programming to fill a 2D table with the maximum values for different combinations of items and knapsack capacities.

Algorithm Steps:

1. Initialize a 2D table `table` of size `(n + 1) x (knapsack_Capacity + 1)`, where `n` is the number of items.
2. Iterate through each item and each possible knapsack capacity, filling in the table with the maximum values.
3. For each item, calculate the maximum value of including the item or excluding the item in the knapsack.
4. The maximum value will be in the cell corresponding to the last item and the knapsack's full capacity.

Example

The program provides an example usage by solving two instances of the knapsack problem with different parameters. After running the program, it prints the maximum value obtained for each instance.

```
int knapsack_Capacity = 25; // Weight capacity of knapsack
        int[] item_values = { 40, 30, 20, 10 }; // Values of each item
        int[] item_weights = { 15, 10, 5, 2 }; // Weights of each item

        DPKnapSack knapsack = new DPKnapSack();
        int max_value = knapsack.choice(knapsack_Capacity, item_values,
item_weights);
        System.out.println("Maximum value is " + max_value);
```

Output

```
Maximum value is 70
```

Note

This program provides a basic implementation of the 0-1 knapsack problem using dynamic programming. It assumes non-negative integer weights and values for simplicity. For more complex scenarios, additional optimizations or algorithms may be required.