UNIVERSITETI® METROPOLITAN TIRANA

REPUBLIC OF ALBANIA

FACULTY OF COMPUTER SCIENCES AND IT
Software Engineering Program (Spring II)

Algorithms and Advanced Programming

Project submission

Name Surname: Alket Sula

TIRANA 2023

Algorithms and Advanced Programming – UMT

April 2023

Quantifying greedy strategies' efficiencies by statistical validation procedure.

Ladies and gentlemen, here is the project.
During "in person" teachings we addressed optimisation problems, among which:
MVB: maximum value bag
MSM: maximum sum of marks
MCP: minimum cost path on a grid.
We solved them by Dynamic Programming (DP).

DP guarantees that the values it computes are optimal i.e. maximum (MVB, MSM) or minimum (MCP).
MVB: DP returns the maximum value $m(n,C)$ where n is the number of available items and C is the bag's capacity
MSM: it returns the maximum sum $m(n,H)$ where n is the number of topics and H is the number of working hours
MCP: it returns the minimum cost $m(L-1,C-1)$ from grid cell at coordinates $(0,0)$ to that at coordinates $(L-1,C-1)$.
DP achieves these results relying on Richard Bellman's principle: "an optimal solution is made of optimal sub-solutions to the problem."

Unlike DP, Greedy algorithms (GAs) perform local optimisations.
MVB: sort the items by decreasing values and fill the bag in this ranking; or the same by decreasing "value densities"
MSM: add one hour of work to one of the topic whose increase is the highest
MCP: step to a neighbour cell whose cost is minimum.

GAs do not always return optimal values. One can show it by designing an instance of the problem at which the GA value is not optimal: smaller than that of DP for MVB and MSM, higher for MCP. Nevertheless, they may be of interest because GAs' complexities are lesser that DP's ones.
MVB: $\Theta(n \times \log n)$ versus $\Theta(n \times C)$
MSM: $\Theta(n \times H)$ versus $\Theta(n \times H^2)$
MCP: $\Theta(L + C)$ versus $\Theta(L \times C)$.
Interested in using a GA rather than DP? Well... Better to have and idea of the the "risk".
The question to ask is "How far from the optimal values are the greedy ones?"
Statistical validation answers this question.

Let us suppose that one adresses a maximisation problem (MVB or MSM) and let p be an instance of the problem. Let us consider an MVB problem. Designing an instance p of MVB is fixing the number n of items, the capacity C of the bag and, for each item i, its value v_i and size s_i. Then, given this instance p of MVB, DP computes the optimal value $m(p)$ and GA computes a sub-optimal value $g(p)$. One has $0 \leq g(p) \leq m(p)$. The distance $d(p)$ between optimal and greedy values is $d(p) = m(p) - g(p)$. The relative distance is $r(p) = \frac{d(p)}{m(p)}$. Because this value is not defined when $m(p) = 0$, one states $r(p) = 0$ when $m(p) = 0$.

The relative distance $r(p)$ quantifies at problem instance p how far from the optimum the greedy value is.
For instance: $r(p) = 0.3$ "tells" that the greedy bag's value is 30% that of maximum value bag.

The statistical validation procedure consists in generatating at random a high number of problem instances. It computes and memorises all the relative distances. Then, given these relative distances one can draw the relative distances' histogram and one can compute meaningful statistical values: mean, standard deviation, median, maximum relative distance, and mean value of the "outliers" (mean of 5% highest relative distances.)

How to "generate at random a problem instance"?
Choose all the values at random. In the case of MVB: choose at random values n and C, and for each item i choose at random its value v_i and size s_i.

Do it for MVB, MSM and MCP problems.
Do it also for the maximum value pair of bags problem that we solved by dynamic programming in the hands on session.
1) Greedy filling of the two bags: any item enters the first bag if it can, otherwise it enters the second bag if it can.
Do it with items ranked by decreasing values then with items ranked by decreasing value densities $\frac{v_i}{s_i}$.
2) Serial optimisation: first bag of maximum value computed by dynamic programming over the whole set of items, second bag of maximum value computed by dynamic programming over the set of remaining items.

How to “generate at random a problem instance”?

1- MVB(Maximum Value Bag):

```
import java.util.ArrayList;
import java.util.List;
import java.util.Random;
class Item {
   int value;
   int size;
   public Item(int value, int size) {
      this.value = value;
      this.size = size;
   }
}
public class RandomProblemGenerator {
   public static void main(String[] args) {
      Random random = new Random();
      int n = random.nextInt(10) + 1; // Randomly choose a number of items between 1 and 10
      int C = random.nextInt(20) + 1; // Randomly choose a capacity of the bag between 1 and 20
      List<Item> items = new ArrayList<>();
      // Generate random values for each item's value and size
      for (int i = 0; i < n; i++) {
         int value = random.nextInt(10) + 1; // Randomly choose a value between 1 and 10
         int size = random.nextInt(10) + 1; // Randomly choose a size between 1 and 10
         items.add(new Item(value, size));
      }
      // Print the generated problem instance
      System.out.println("Number of items: " + n); // Print the number of items generated
      System.out.println("Capacity of the bag: " + C); // Print the capacity of the bag generated
      System.out.println("Items:");
      // Print the details of each item
      for (int i = 0; i < n; i++) {
         Item item = items.get(i);
         System.out.println("Item " + (i + 1) + ": Value=" + item.value + ", Size=" + item.size);
                        }
            }
}
```

What are we taking into consideration :

- *Number of bags and capacity*
- *Random value and size for each item*

2- MSM(Maximum Sum of Marks)

```
import java.util.Random;

public class RandomProblemGenerator {

   public static void main(String[] args) {
      Random random = new Random();

      int n = random.nextInt(10) + 1; // Randomly choose the number of students or marks between 1 and 10

      int[] scores = new int[n]; // Array to store the scores

      // Generate random scores for each student or mark
      for (int i = 0; i < n; i++) {
         int score = random.nextInt(101); // Randomly choose a score between 0 and 100
         scores[i] = score;
      }

      // Print the generated problem instance
      System.out.println("Number of students or marks: " + n); // Print the number of students or marks generated
      System.out.println("Scores:");

      // Print the scores of each student or mark
      for (int i = 0; i < n; i++) {
         System.out.println("Student or Mark " + (i + 1) + ": Score=" + scores[i]);
      }
   }
}
```

What are we taking into consideration:

- *The number of students or marks*
- *Random scores for each student or mark*

3- MCP(Minimum Cost Path)

```
import java.util.Random;

public class RandomProblemGenerator {

   public static void main(String[] args) {
      Random random = new Random();

      int m = random.nextInt(10) + 1; // Randomly choose the number of rows between 1 and 10
      int n = random.nextInt(10) + 1; // Randomly choose the number of columns between 1 and 10

      int[][] grid = new int[m][n]; // 2D array to store the costs

      // Generate random costs for each cell
      for (int i = 0; i < m; i++) {
         for (int j = 0; j < n; j++) {
            int cost = random.nextInt(10) + 1; // Randomly choose a cost between 1 and 10
            grid[i][j] = cost;
         }
      }

      // Print the generated problem instance
      System.out.println("Size of grid: " + m + " x " + n); // Print the size of the grid generated
      System.out.println("Costs:");

      // Print the costs of each cell
      for (int i = 0; i < m; i++) {
         for (int j = 0; j < n; j++) {
            System.out.print(grid[i][j] + " ");
         }
         System.out.println();
      }
   }
}
```

What are we taking into consideration :

- *The size of the grid*
- *Random costs to each cell*

Using Greedy Algorithm to generate at random a problem instance.
Examples given below are regarding the three above exercises to generate random value, and now their implementation in exercises.

1- MVB(Maximum Value Bag):

//the exercise is solved based in the request in the main project page.

```
import java.util.ArrayList;
import java.util.Collections;
import java.util.Comparator;
import java.util.List;
import java.util.Random;

class Item {
   int value;
   int size;

   public Item(int value, int size) {
      this.value = value;
      this.size = size;
   }
}

public class GreedyFillingBags {

   public static void main(String[] args) {
      Random random = new Random();

      int n = random.nextInt(10) + 1; // Randomly choosing the number of items between 1 and 10
      int C1 = random.nextInt(20) + 1; // Randomly choosing  the capacity of the first bag between 1 and 20
      int C2 = random.nextInt(20) + 1; // Randomly choosing the capacity of the second bag between 1 and 20
//It could have been any number!!!!!!!

      List<Item> itemsByValue = new ArrayList<>();
      List<Item> itemsByDensity = new ArrayList<>();

      // Generating random values and sizes for each item
      for (int i = 0; i < n; i++) {
         int value = random.nextInt(10) + 1; // Randomly choose a value between 1 and 10
         int size = random.nextInt(10) + 1; // Randomly choose a size between 1 and 10

         itemsByValue.add(new Item(value, size));
         itemsByDensity.add(new Item(value, size));
      }

      // Sort items by decreasing values
      Collections.sort(itemsByValue, Comparator.comparingInt((Item item) -> item.value).reversed());

      // Sort items by decreasing value densities (value/size)
      Collections.sort(itemsByDensity, Comparator.comparingDouble((Item item) -> (double) item.value / item.size).reversed());

      // Greedy filling of bags based on decreasing values
      List<Item> firstBagByValue = new ArrayList<>();
      List<Item> secondBagByValue = new ArrayList<>();

      int remainingCapacity1 = C1;
      int remainingCapacity2 = C2;

      for (Item item : itemsByValue) {
```

```
        if (item.size <= remainingCapacity1) {
            firstBagByValue.add(item);
            remainingCapacity1 -= item.size;
        } else if (item.size <= remainingCapacity2) {
            secondBagByValue.add(item);
            remainingCapacity2 -= item.size;
        }
    }

    // Greedy filling of bags based on decreasing value densities
    List<Item> firstBagByDensity = new ArrayList<>();
    List<Item> secondBagByDensity = new ArrayList<>();

    int remainingCapacity3 = C1;
    int remainingCapacity4 = C2;

    for (Item item : itemsByDensity) {
        if (item.size <= remainingCapacity3) {
            firstBagByDensity.add(item);
            remainingCapacity3 -= item.size;
        } else if (item.size <= remainingCapacity4) {
            secondBagByDensity.add(item);
            remainingCapacity4 -= item.size;
        }
    }

    // Print the filled bags for both rankings
    System.out.println("Filled bags based on decreasing values:");
    System.out.println("First Bag:");
    for (Item item : firstBagByValue) {
        System.out.println("Value: " + item.value + ", Size: " + item.size);
    }

System.out.println("Second Bag:");
for (Item item : secondBagByValue) {
    System.out.println("Value: " + item.value + ", Size: " + item.size);
}

System.out.println("\nFilled bags based on decreasing value densities:");
System.out.println("First Bag:");
for (Item item : firstBagByDensity) {
    System.out.println("Value: " + item.value + ", Size: " + item.size);
}

System.out.println("Second Bag:");
for (Item item : secondBagByDensity) {
    System.out.println("Value: " + item.value + ", Size: " + item.size);
}
```

Explanation:

First of all, we randomly generate the number of items (n) and capacities of two bags (C1 and C2). Then, create two lists: itemsByValue and itemsByDensity, to store the items. Each item, is going to have its value and size, such that can be added to both lists.

ItemsByValue is used to sort them, as we were asked in descending order regarding the values. The same process is applied for density as well (ItemsByDensity).

The greedy filling of bags based on decreasing values:

Then we make sure to have a data structure to store the values of items and their densities. I have chosen lists, because the operations can be easier regarding the adding or removing the items. The first list contains the value of the item, and the second list contains the density of the item.

We iterate through the items in itemsByValue and place an item in the first bag if there is enough capacity. Otherwise, we place it in the second bag if possible.

We perform the greedy filling of bags based on decreasing value densities:

Here goes the same process regarding the data structure.

We iterate through the items in itemsByDensity and place an item in the first bag if there is enough capacity. Otherwise, we place it in the second bag if possible.

Finally, we print the items in each bag based on decreasing values and decreasing value densities.

Serial Optimization

- ***MVB (Maximum Value Bag)***

Optimizing the solution by using dynamic programming for computing the maximum value of the first bag and the remaining items for the second bag:

```
import java.util.ArrayList;
import java.util.Collections;
import java.util.Comparator;
import java.util.List;
import java.util.Random;

class Item {
   int value;
   int size;

   public Item(int value, int size) {
      this.value = value;
      this.size = size;
   }
}

public class MaximumValueBag {

   public static void main(String[] args) {
      Random random = new Random();
      int n = random.nextInt(10) + 1; // Number of items (adjust the range as needed)
      int C = random.nextInt(20) + 1; // Capacity of the bag (adjust the range as needed)

      List<Item> items = new ArrayList<>();
      for (int i = 0; i < n; i++) {
         int value = random.nextInt(10) + 1; // Value of the item (adjust the range as needed)
         int size = random.nextInt(10) + 1; // Size of the item (adjust the range as needed)
         items.add(new Item(value, size));
      }

      // Sort items by decreasing values
      Collections.sort(items, Comparator.comparingInt((Item item) -> item.value).reversed());

      List<Item> bag1 = new ArrayList<>();
      List<Item> bag2 = new ArrayList<>();

      int[][] maxValues = new int[n + 1][C + 1];

      // Dynamic programming for computing the maximum value of the first bag
      for (int i = 1; i <= n; i++) {
         for (int j = 1; j <= C; j++) {
            if (items.get(i - 1).size <= j) {
               maxValues [i][j] = Math.max(maxValues [i - 1][j], maxValues [i - 1][j - items.get(i - 1).size] + items.get(i - 1).value);
            } else {
               maxValues [i][j] = maxValues [i - 1][j];
            }
         }
      }

      int remainingCapacity = C;
      int remainingItems = n;

      // Computing the second bag using dynamic programming over the set of remaining items
```

```
    while (remainingItems > 0 && remainingCapacity > 0) {
      if (maxValues [remainingItems][remainingCapacity] != maxValues [remainingItems - 1][remainingCapacity]) {
        Item item = items.get(remainingItems - 1);
        bag2.add(item);
        remainingCapacity -= item.size;
      }
      remainingItems--;
    }

    // Print the items in each bag
    System.out.println("Items in Bag 1:");
    for (Item item : bag1) {
      System.out.println("Value: " + item.value + ", Size: " + item.size);
    }

    System.out.println("Items in Bag 2:");
    for (Item item : bag2) {
      System.out.println("Value: " + item.value + ", Size: " + item.size);
    }
  }
}
```

Explanation:

In the code above, we are using dynamic programming to calculate the maximum value that the first bag can have. We create a 2D array called maxValues of size (n + 1) x (C + 1), where each element maxValues[i][j] represents the maximum value that can be achieved using the first i items and a capacity of j.

After computing the maxValues array, we iterate through it to specify the items remaining for the second bag. We start with the remaining capacity and remaining items, and at each step, we check if the value at maxValues [remainingItems][remainingCapacity] is different from the value above it (maxValues [remainingItems-1][remainingCapacity]). If they are different, it means that the current item contributes to the maximum value, so we add it to the second bag and reduce the remaining capacity.