UMT

Advanced Algorithms and Programming

Quantifying greedy strategies' efficiencies by statistical validation procedure.

Worked by: Eljon Zagradi **Professor: Rene Natowicz**

I conducted testing with 2000 instances for each algorithm in my program. This large sample size helped me analyze the performance differences between the greedy and dynamic programming approaches comprehensively.

For each of the problems I have provided "randomArray" function which generates a random array of integers 1D or 2D depending on the Problem. Also I provided the "getR" which calculates the relative distace of greedy and dynamic programming in (%). If needed also the display functions are provided.

For each Problem is provided a program that tests and compares the performance of the dynamic programming and greedy algorithms. The program conducts 2000 iterations, generating random instances of the problem and measuring the execution times for both algorithms. It calculates the maximum -> (MVB, MSM, TB) or minimum -> (MCP) value obtained by each algorithm and collects the results. Finally, the results are visualized in a histogram, providing insights into the performance differences between the two approaches. The code demonstrates a systematic approach to evaluating and comparing algorithmic solutions for all the problems.

The printouts and explanation for each problem are saved at respective files:

Maxiimum Value Bag -> MVB.txt

Maximum Sum of Marks -> MSM.txt

Minimum Cost Path -> MCP.txt

Two Bags - > TB.txt

The full code for the Project is provided in this email but also can be found here:
https://github.com/eljonzagradi/UMT_Eljon_Zagradi

1. Maximum Value Bag Algorithm

```
static int[] randomArray(int n, int max) {
    int[] A = new int[n];
    Random r = new Random();
    for (int i = 0; i < n; i++) {
        A[i] = 1 + r.nextInt(max);
    }
    return A;
}
```

```
static double getR(double valDP, double valGA) {
    DecimalFormat df = new DecimalFormat("#.##");
    return valDP == 0 ? 0 : (Double.parseDouble(df.format((valDP - valGA) / valDP)) * 100);
}
```

```
    ArrayList<Double> results = new ArrayList<>();
    DecimalFormat df = new DecimalFormat("#.##");
    Random ran = new Random();
    final int SIZE = 50, MAX_ITERATIONS = 2000;
    int iteration = 0;

    while (iteration < MAX_ITERATIONS) {

        int[] V = randomArray(SIZE, 50);
        int[] S = randomArray(SIZE, 31);
        int C = 1 + ran.nextInt(40);
        System.out.println("Iteration: " + (iteration + 1));
        System.out.println("V = " + Arrays.toString(V));
        System.out.println("S = " + Arrays.toString(S));
        System.out.println("C = " + C);

        long timeDP = 0, timeGA = 0;
        long startDP = System.nanoTime();
        int valDP = MVB.dynamic_programming_computeM(V, S, C);
        long endDP = System.nanoTime();

        long startGA = System.nanoTime();
        int valGA = MVB.greedy_algorithm_computeM(V, S, C);
        long endGA = System.nanoTime();

        timeDP = endDP - startDP;
        timeGA = endGA - startGA;
        double result = Double.parseDouble(df.format(getR(valDP, valGA)));

        System.out.println("Max Value DP: " + valDP + " | Time: " + timeDP + " ns");
        System.out.println("Max Value GA: " + valGA + " | Time: " + timeGA + " ns");
        System.out.println("R : " + result + "%");
        results.add(result);

        iteration++;
        System.out.println();
    }
    statistics.StatisticalReport.createHistogram(results);
```

```java
public static int dynamic_programming_computeM(int[] V, int[] S, int C) {
    int n = V.length;
    int[][] M = new int[n + 1][C + 1];

    // base cases
    for (int c = 0; c < C + 1; c++)
        M[0][c] = 0;
    // general cases
    for (int k = 1; k < n + 1; k++)
        for (int c = 0; c < C + 1; c++)
            if (c - S[k - 1] < 0)
                M[k][c] = M[k - 1][c];
            else
                M[k][c] = max(M[k - 1][c], V[k - 1] + M[k - 1][c - S[k - 1]]);

    return M[V.length][C];
}
```

```java
static class Item {
    private int value;
    private int size;

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

    public int getValue() {
        return value;
    }

    public int getSize() {
        return size;
    }
}
```

```java
public static int greedy_algorithm_computeM(int[] V, int[] S, int C) {
    List<Item> items = new ArrayList<>();
    int n = V.length;

    // Create a list of items
    for (int i = 0; i < n; i++) {
        items.add(new Item(V[i], S[i]));
    }

    // Sort items based on value ratio in descending order
    Collections.sort(items, Comparator.comparingInt(Item::getValue).reversed());

    int totalValue = 0;
    int currentCapacity = C;

    // Greedy selection of items
    for (Item item : items) {
        if (item.size <= currentCapacity) {
            totalValue += item.value;
            currentCapacity -= item.size;
        }
    }

    return totalValue;
}
```

2. Maximum Sum of Marks:

```
static int[][] randomArray(int n, int H) {
    int[][] A = new int[n][H + 1];
    Random r = new Random();
    for (int i = 0; i < n; i++) {
        for (int j = 0; j < H + 1; j++) {
            A[i][j] = r.nextInt(H + 1);
        }
    }
    return A;
}
```

```
static void displayE(int[][] E) {
    int L = E.length;
    for (int i = L - 1; i > -1; i--)
        System.out.printf("\ti:%d\t %s\n", i, Arrays.toString(E[i]));
}
```

```
ArrayList<Double> results = new ArrayList<>();
DecimalFormat df = new DecimalFormat("#.##");
final int n = 10, H = 20, MAX_ITERATIONS = 2000;
int iteration = 0;

while (iteration < MAX_ITERATIONS) {

    int[][] E = randomArray(n, H);
    System.out.println("Iteration: " + (iteration + 1));
    System.out.println("E = ");
    displayE(E);

    long timeDP = 0, timeGA = 0;
    long startDP = System.nanoTime();
    int valDP = MSM.dynamic_programming_computeMSM(E);
    long endDP = System.nanoTime();

    long startGA = System.nanoTime();
    int valGA = MSM.greedy_algorithm_computeMSM(E);
    long endGA = System.nanoTime();

    timeDP = endDP - startDP;
    timeGA = endGA - startGA;
    double result = Double.parseDouble(df.format(getR(valDP, valGA)));

    System.out.println("Max Sum DP: " + valDP + " | Time: " + timeDP + " ns");
    System.out.println("Max Sum GA: " + valGA + " | Time: " + timeGA + " ns");
    System.out.println("R : " + result + "%");
    results.add(result);

    iteration++;

    System.out.println();
}

statistics.StatisticalReport.createHistogram(results);
```

```java
static int[][][] computeMA(int[][] E) {
    int n = E.length, H = E[0].length - 1;
    int[][] M = new int[n + 1][H + 1], A = new int[n + 1][H + 1];

    for (int h = 0; h < H + 1; h++)
        M[0][h] = 0;

    for (int k = 1; k < n + 1; k++)
        for (int h = 0; h < H + 1; h++) {
            int mkh = Integer.MIN_VALUE, akh = -1;
            for (int hprime = 0; hprime < h + 1; hprime++) {
                int mkh_hprime = E[k - 1][hprime] + M[k - 1][h - hprime];
                if (mkh_hprime > mkh) {
                    mkh = mkh_hprime;
                    akh = hprime;
                }
            }
            M[k][h] = mkh;
            A[k][h] = akh;
        }
    return new int[][][] { M, A };
}

public static int dynamic_programming_computeMSM(int[][] E) {
    int n = E.length, H = E[0].length - 1;
    int[][][] MA = computeMA(E);
    int[][] M = MA[0];
    return M[n][H];
}
```

```java
public static int greedy_algorithm_computeMSM(int[][] matrix) {
    int rows = matrix.length;
    int cols = matrix[0].length - 1;
    int[][] M = new int[rows + 1][cols + 1];

    // Ground cases
    for (int col = 0; col <= cols; col++) {
        M[0][col] = 0;
    }

    // Recurrence cases
    for (int row = 1; row <= rows; row++) {
        for (int col = 0; col <= cols; col++) {
            int maxElement = -1;
            int maxColPrime = 0;
            for (int colPrime = 0; colPrime <= col; colPrime++) {
                if (matrix[row - 1][colPrime] > maxElement) {
                    maxElement = matrix[row - 1][colPrime];
                    maxColPrime = colPrime;
                }
            }
            M[row][col] = maxElement + M[row - 1][col - maxColPrime];
        }
    }

    int maxSum = Integer.MIN_VALUE;
    for (int col = 0; col <= cols; col++) {
        if (M[rows][col] > maxSum) {
            maxSum = M[rows][col];
        }
    }

    return maxSum;
}
```

3. Minimum Cost Path

```
static int[][] randomArray(int size, int max) {
    int[][] A = new int[size][size];
    Random r = new Random();
    for (int i = 0; i < size; i++) {
        for (int j = 0; j < size; j++) {
            A[i][j] = r.nextInt(max);
        }
    }
    return A;
}
```

```
static void displayG(int[][] G) {
    int L = G.length;
    for (int l = L - 1; l > -1; l--)
        System.out.println(Arrays.toString(G[l]));
}
```

```
static double getR(double valDP, double valGA) {
    DecimalFormat df = new DecimalFormat("#.##");
    return valGA == 0 ? 0 : (Double.parseDouble(df.format((valGA - valDP) / valGA)) * 100);
}
```

```
ArrayList<Double> results = new ArrayList<>();
DecimalFormat df = new DecimalFormat("#.##");
final int SIZE = 10, MAX_ITERATIONS = 2000;
int iteration = 0;

while (iteration < MAX_ITERATIONS) {

    int[][] G = randomArray(SIZE, 31);
    System.out.println("Iteration: " + (iteration + 1));
    System.out.println("G = ");
    displayG(G);

    long timeDP = 0, timeGA = 0;
    long startDP = System.nanoTime();
    int valDP = MCP.dynamic_programming_computeMCP(G);
    long endDP = System.nanoTime();

    long startGA = System.nanoTime();
    int valGA = MCP.greedy_algorithm_computeMCP(G);
    long endGA = System.nanoTime();

    timeDP = endDP - startDP;
    timeGA = endGA - startGA;
    double result = Double.parseDouble(df.format(getR(valDP, valGA)));

    System.out.println("Min Cost DP: " + valDP + " | Time: " + timeDP + " ns");
    System.out.println("Min Cost GA: " + valGA + " | Time: " + timeGA + " ns");
    System.out.println("R : " + result + "%");
    results.add(result);

    iteration++;
    System.out.println();
}

statistics.StatisticalReport.createHistogram(results);
```

```java
public static int dynamic_programming_computeMCP(int[][] G) {
    int[][][] MA = computeMA(G);
    int[][] M = MA[0];
    int L = M.length, C = M[0].length;
    return M[L - 1][C - 1];
}

static int[][][] computeMA(int[][] G) {
    int L = G.length, C = G[0].length;
    int[][] M = new int[L][C], A = new int[L][C];
    M[0][0] = G[0][0];
    for (int i = 0; i < L; i++)
        for (int j = 0; j < C; j++)
            if (!(i == 0 && j == 0)) {
                int m_e = minCost(M, i, j - 1, L, C), m_ne = minCost(M, i - 1, j - 1, L, C),
                        m_n = minCost(M, i - 1, j, L, C);
                if (m_e == min(m_e, m_ne, m_n))
                    A[i][j] = 0;
                else if (m_ne == min(m_ne, m_n))
                    A[i][j] = 1;
                else
                    A[i][j] = 2;
                M[i][j] = min(m_e, m_ne, m_n) + G[i][j];
            }

    return new int[][][] { M, A };
}
```

```java
  public static int greedy_algorithm_computeMCP(int[][] G) {
      int totalCost = G[0][0];
      int currentRow = 0;
      int currentCol = 0;

      while (currentRow != G.length - 1 || currentCol != G[0].length - 1) {
          int costEast = Integer.MAX_VALUE;
          int costNortheast = Integer.MAX_VALUE;
          int costNorth = Integer.MAX_VALUE;

          if (currentCol + 1 < G[0].length) {
              costEast = G[currentRow][currentCol + 1];
          }
          if (currentRow + 1 < G.length && currentCol + 1 < G[0].length) {
              costNortheast = G[currentRow + 1][currentCol + 1];
          }
          if (currentRow + 1 < G.length) {
              costNorth = G[currentRow + 1][currentCol];
          }

          if (costEast <= costNortheast && costEast <= costNorth) {
              currentCol += 1; // Move east
          } else if (costNortheast <= costEast && costNortheast <= costNorth) {
              currentRow += 1; // Move northeast
              currentCol += 1;
          } else {
              currentRow += 1; // Move north
          }

          totalCost += G[currentRow][currentCol];
      }

      return totalCost;
  }
```

4. Two Bags

```
static int[] randomArray(int n, int max) {
    int[] A = new int[n];
    Random r = new Random();
    for (int i = 0; i < n; i++) {
        A[i] = 1 + r.nextInt(max);
    }
    return A;
}
```

```
static double getR(double valDP, double valGA) {
    DecimalFormat df = new DecimalFormat("#.##");
    return valDP == 0 ? 0 : (Double.parseDouble(df.format((valDP - valGA) / valDP)) * 100);
}
```

```
ArrayList<Double> results_v = new ArrayList<>();
ArrayList<Double> results_d = new ArrayList<>();
DecimalFormat df = new DecimalFormat("#.##");
final int n = 50, maxSize = 27, maxValue = 34, C0 = 24, C1 = 31, MAX_ITERATIONS = 1000;
int iteration = 0;

while (iteration < MAX_ITERATIONS) {

    int[] V = randomArray(n, maxValue);
    int[] S = randomArray(n, maxSize);

    System.out.println("Iteration: " + (iteration + 1));
    System.out.println("V = " + Arrays.toString(V));
    System.out.println("S = " + Arrays.toString(S));
    System.out.println("C0 = " + C0);
    System.out.println("C1 = " + C1);

    long timeDP = 0, timeGA_v = 0, timeGA_d = 0;
    long startDP = System.nanoTime();
    int valDP = TB.dynamic_programming_computeMV(V, S, C0, C1);
    long endDP = System.nanoTime();

    // Sort items based on value ratio in descending order
    long startGA_v = System.nanoTime();
    int valGA_v = TB.greedy_algorithm_sorted_by_value_computeMV(V, S, C0, C1);
    long endGA_v = System.nanoTime();

    // Sort items based on density ratio in descending order
    long startGA_d = System.nanoTime();
    int valGA_d = TB.greedy_algorithm_sorted_by_density_computeMV(V, S, C0, C1);
    long endGA_d = System.nanoTime();

    timeDP = endDP - startDP;
    timeGA_v = endGA_v - startGA_v;
    timeGA_d = endGA_d - startGA_d;
    double result_v = Double.parseDouble(df.format(getR(valDP, valGA_v)));
    double result_d = Double.parseDouble(df.format(getR(valDP, valGA_d)));

    System.out.println("Max value of the pair of bags DP: " + valDP + " | Time: " + timeDP + " ns");
    System.out.println("Max value of the pair of bags GA_v: " + valGA_v + " | Time: " + timeGA_v + " ns");
    System.out.println("Max value of the pair of bags GA_d: " + valGA_d + " | Time: " + timeGA_d + " ns");
    System.out.println("R_v : " + result_v + "%");
    System.out.println("R_d : " + result_d + "%");
    results_v.add(result_v);
    results_d.add(result_d);

    iteration++;
    System.out.println();
}
System.out.println("Greedy Algorithm sorted by values: ");
statistics.StatisticalReport.createHistogram(results_v);
System.out.println("Greedy Algorithm sorted by densities: ");
statistics.StatisticalReport.createHistogram(results_d);
```

```java
static int dynamic_programming_computeMV(int[] V, int[] S, int C0, int C1) {
    int n = V.length;
    int[][][] M = new int[n + 1][C0 + 1][C1 + 1];
    for (int k = 1; k < n + 1; k++)
        for (int c0 = 0; c0 < C0 + 1; c0++)
            for (int c1 = 0; c1 < C1 + 1; c1++) {
                int m0 = -1, m1 = -1;
                if (S[k - 1] <= c0)
                    m0 = V[k - 1] + M[k - 1][c0 - S[k - 1]][c1];
                if (S[k - 1] <= c1)
                    m1 = V[k - 1] + M[k - 1][c0][c1 - S[k - 1]];
                M[k][c0][c1] = max(M[k - 1][c0][c1], max(m0, m1));
            }

    return M[n][C0][C1];
}

static int max(int x, int y) {
    if (x >= y)
        return x;
    return y;
}
```

```java
static int greedy_algorithm_sorted_by_value_computeMV(int[] V, int[] S, int C0, int C1) {
    List<Item> items = new ArrayList<>();
    for (int i = 0; i < V.length; i++) {
        items.add(new Item(V[i], S[i]));
    }

    // Sort items by value in descending order
    Collections.sort(items, Comparator.comparingInt(Item::getValue).reversed());

    return computeMV(items, C0, C1);
}

static int greedy_algorithm_sorted_by_density_computeMV(int[] V, int[] S, int C0, int C1) {
    List<Item> items = new ArrayList<>();
    for (int i = 0; i < V.length; i++) {
        items.add(new Item(V[i], S[i]));
    }

    // Sort items by density (value/size) in descending order
    Collections.sort(items, Comparator.comparingDouble(Item::getDensity).reversed());

    return computeMV(items, C0, C1);
}
```

```java
static int computeMV(List<Item> all_items, int C0, int C1) {
    List<Item> remaining_items = new ArrayList<>();

    int totalValue = 0;
    int currentCapacity = C0;

    // Greedy selection of items for C0
    for (Item item : all_items) {
        if (item.size <= currentCapacity) {
            totalValue += item.value;
            currentCapacity -= item.size;
        } else {
            remaining_items.add(item);
        }
    }

    currentCapacity = C1; // Reset currentCapacity for C1

    // Greedy selection of remaining items for C1
    for (Item item : remaining_items) {
        if (item.size <= currentCapacity) {
            totalValue += item.value;
            currentCapacity -= item.size;
        }
    }

    return totalValue;
}
```

```java
static class Item {
    private int value;
    private int size;

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

    public int getValue() {
        return value;
    }

    public int getSize() {
        return size;
    }

    public double getDensity() {
        return (double) value / size;
    }
}
```