Project Documentation

Name: Darling Selita.

Subject: Algorithms.

Date of submission: 25/05/2023.

Firstly, I want to talk about the 2 main topics of this project:

- Dynamic Programming.
- Greedy Algorithms.

Dynamic programming is a problem-solving technique used in computer science and mathematics to solve optimization problems by breaking them down into smaller overlapping subproblems. It is based on the principle of "optimal substructure," which states that an optimal solution to a larger problem can be constructed from optimal solutions to its smaller subproblems. Dynamic programming is particularly useful when the problem has overlapping subproblems, meaning that the same subproblems are solved multiple times. By avoiding redundant computations, dynamic programming can significantly reduce the time complexity of the algorithm and provide efficient solutions to complex optimization problems.

Greedy algorithms are a class of algorithms that make locally optimal choices at each step with the aim of finding a global optimum. In other words, they make the best possible choice in the current moment without considering the potential consequences of that choice on future steps. They are generally efficient and easy to implement compared to other algorithmic approaches, such as dynamic programming. However, it's important to note that greedy algorithms do not always guarantee finding the globally optimal solution for all types of problems.

Disclaimer: These problems have been solved using python because of its histogram library called matplotlib which gives me the option to make a histogram of specific data using the plot() command.

1-st Problem: Maximum Value Bag (MVB).

Firstly I took all the code you gave us on the Drive and redid it in Python. This is the DP runaround of the problem.

```python

# Dynamic programming approach
1 usage
def dynamic_programming(values, size, capacity):
    n = len(values)
    dp = [[0] * (capacity + 1) for i in range(n + 1)]
    for i in range(1, n + 1):
        for w in range(1, capacity + 1):
            if size[i - 1] <= w:
                dp[i][w] = max(dp[i - 1][w], dp[i - 1][w - size[i - 1]] + values[i - 1])
            else:
                dp[i][w] = dp[i - 1][w]
    return dp[n][capacity]
```

What I'm trying to achieve here is to use two loops to iterate over all the items' sizes and compare them to the max quantity of the bag. If the size of the current item is less than or equal to the current capacity, the code compares two values:

-dp[i - 1][w], which represents the maximum value achieved without including the current item.

-dp[i - 1][w - size[i - 1]] + values[i - 1], which represents the maximum value achieved by including the current item.

Here, values[i - 1] refers to the value of the current item. If the size of the current item is greater than the current capacity, it means the current item cannot be included in the bag. In this case, the maximum value achieved without including the current item is copied to dp[i][w].

I also solved the problem using GA.

```python
def greedy_knapsack_value(values, weights, capacity):
    n = len(values)
    items = list(zip(values, weights))
    items.sort(key=lambda x: x[0], reverse=True)
    max_value = 0
    for v, w in items:
        if w <= capacity:
            max_value += v
            capacity -= w
        if capacity == 0:
            break
    return max_value
```

The items list is sorted in descending order based on the item values using the sort method and a lambda function as the sorting key. The function iterates over each item in the sorted items list. For each item, it checks if the size of the item (w) is less than or equal to the remaining capacity (capacity). If the item size is less than or equal to the remaining capacity, it means the item can be fully included in the bag. In this case, the item's value (v) is added to the max_value, and the remaining capacity is reduced by the item's size. If the item size is greater than the remaining capacity, it means the item cannot be fully included in the bag. Instead, a fraction of the item is included, proportional to the remaining capacity and the item's size-to-value ratio. The function calculates this fraction as (capacity / w) * v. The fractional value is added to the max_value, and the loop is terminated using break.

After doing both approaches, I generated 1000 random cases with values close to each-other and also found the statistic values required. After that I plotted the histogram using the values given to display us the results.

```python
# Generate 1000 random cases
random.seed(42)
cases = []
for i in range(1000):
    n = random.randint(10, 50)
    values = [random.randint(1, 40) for j in range(n)]
    size = [random.randint(1, 40) for j in range(n)]
    capacity = random.randint(sum(size) // 2, sum(size))
    cases.append((values, size, capacity))
```

Random Instance Generator Function

```python
# Compute maximum value using dynamic programming and greedy algorithm for each case
dp_results = []
greedy_results = []
for i in range(1000):
    values, size, capacity = cases[i]
    dp_result = dynamic_programming(values, size, capacity)
    dp_results.append(dp_result)

    greedy_result = greedy_algorithm_value(values, size, capacity)
    greedy_results.append(greedy_result)


# Compute relative distances between DP and greedy-by-value solutions
rel_distances = []
for i in range(1000):
    dp_result = dp_results[i]
    greedy_result = greedy_results[i]
    rel_distance = (dp_result - greedy_result) / dp_result if dp_result > 0 else 0
    rel_distances.append(rel_distance)
```

Calculation of maximum relative distance between DP and GA.

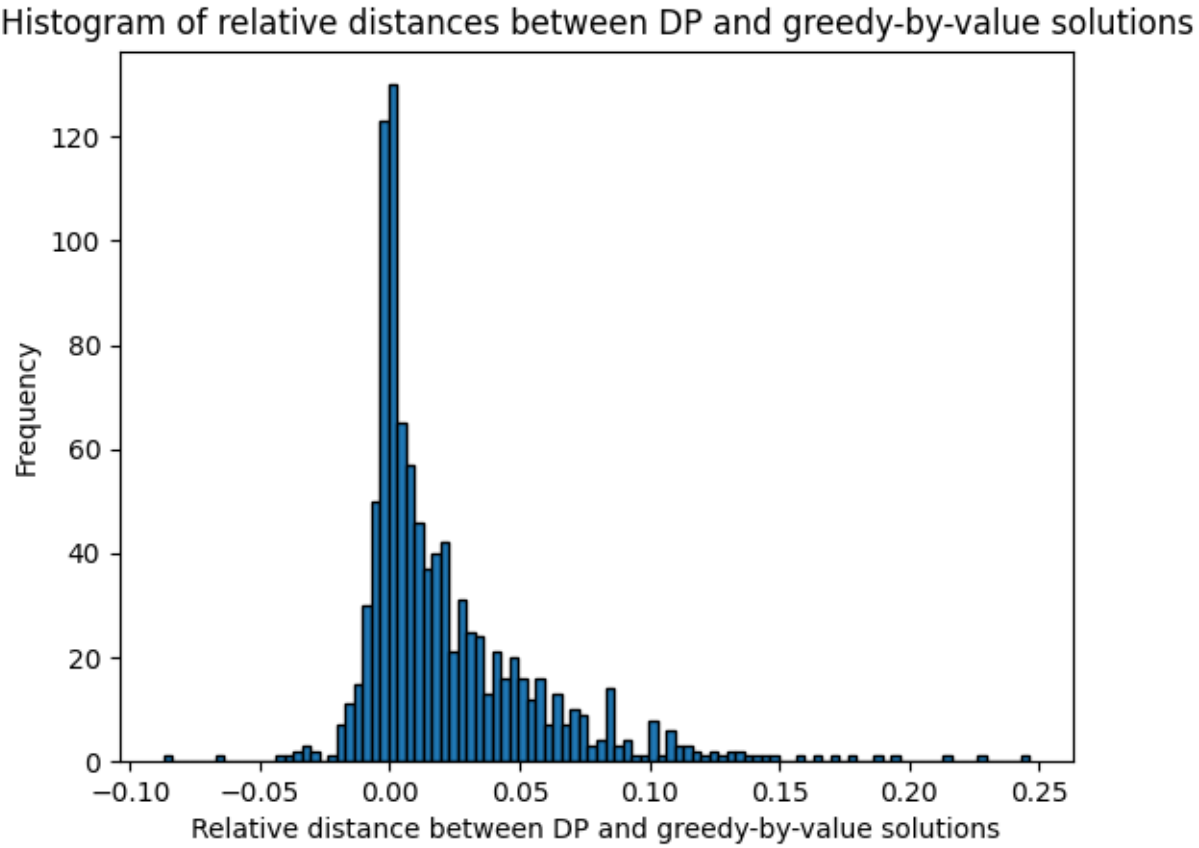
```python
# Calculate mean, standard deviation, median, and maximum relative distance for greedy-by-value
mean_value = np.mean(rel_distances)
std_dev_value = np.std(rel_distances)
median_value = np.median(rel_distances)
max_distance_value = np.max(rel_distances)

print(f"Greedy by value:")
print(f"Mean: {mean_value:.2f}")
print(f"Standard Deviation: {std_dev_value:.2f}")
print(f"Median: {median_value:.2f}")
print(f"Maximum Relative Distance: {max_distance_value:.2f}")

# Calculate mean of outliers (5% highest relative distances)
outliers_value = sorted(rel_distances)[-int(len(rel_distances) * 0.05):]
mean_outliers_value = np.mean(outliers_value)

print(f"Mean of Outliers (Greedy by value): {mean_outliers_value:.2f}")
```

Calculation of mean, median standard deviation and max relative distance.

```python
# Plot histogram of relative distances
plt.hist(rel_distances, bins=100, edgecolor='black')
plt.xlabel('Relative distance between DP and greedy-by-value solutions')
plt.ylabel('Frequency')
plt.title('Histogram of relative distances between DP and greedy-by-value solutions')
plt.show()

print("Max relative distance:", max(rel_distances))
```

Histogram function.

And, here are the results->




```
Max relative distance: 0.2464768478573483
Mean: 0.02
Standard Deviation: 0.04
Median: 0.01
Maximum Relative Distance: 0.25
Mean of Outliers (Greedy by value): 0.13
```

2-nd Problem: Maximum Sum of Marks (MSM).

I did the same thing from the first problem (Took the code you gave us in the drive and rewriting it in Python). The DP method of solving is shown below:

```
def dp_allocation(H, n, e):
    m = [[0 for _ in range(H+1)] for _ in range(n+1)]

    for k in range(1, n+1):
        for h in range(H+1):
            for h_prime in range(h+1):
                m[k][h] = max(m[k][h], e[k-1][h_prime] + m[k-1][h-h_prime])

    hours = [0] * n
    k = n
    h = H
    while k > 0 and h > 0:
        for h_prime in range(h+1):
            if m[k][h] == e[k-1][h_prime] + m[k-1][h-h_prime]:
                hours[k-1] += h_prime
                h -= h_prime
                k -= 1
                break

    return hours
```

The variable m is initialized as a 2D list of size (n+1) x (H+1) filled with zeros. This matrix will store the maximum efficiency values for each task and remaining hours. The function iterates over each task k from 1 to n and each remaining hours h from 0 to H. For each task and remaining hours, it further iterates over h prime from 0 to h to consider all possible allocations of hours for the current task. It updates m[k][h] with the maximum efficiency value between the current efficiency e[k-1][h prime] plus the maximum efficiency for the remaining hours m[k-1][h-h prime]. After completing the nested loops, the m matrix will store the maximum efficiency values for each task and remaining hours. The function initializes a list hours of size n to store the allocated hours for each task. Starting

from the last task k = n and total remaining hours h = H, it iteratively determines the hours allocated for each task by backtracking through the m matrix. It searches for the h prime value that corresponds to the maximum efficiency value m[k][h] by comparing e[k-1][h prime] + m[k-1][h-h prime] for each h prime from 0 to h. Once the h prime value is found, it updates the allocated hours for task k-1 by adding h prime to hours[k-1]. It subtracts the allocated hours h prime from the total remaining hours h, decrements k by 1, and continues the process until all tasks are allocated or there are no more remaining hours. Finally, the function returns the hours list, which represents the allocated hours for each task in order to achieve maximum efficiency while respecting the resource constraints.

After that, I took the Greedy Algorithm approach and the code looks like this:

```
27  def allocate_hours(H, n, e):
28      hours = [H // n] * n
29      for _ in range(H % n):
30          best_topic = -1
31          best_increase = 0
32          for i in range(n):
33              if hours[i] < H // n + 1:
34                  increase = (e[i][hours[i] + 1] - e[i][hours[i]]) / (hours[i] + 1 + 1e-9)
35                  if increase > best_increase:
36                      best_increase = increase
37                      best_topic = i
38          hours[best_topic] += 1
39      return hours
```

The function initializes the hours list with an equal allocation of H // n hours to each task. It calculates the remainder hours H % n that are left after the equal allocation. It enters a loop that iterates H % n times to distribute the remaining hours among the tasks. Within each iteration, the function searches for the task (best topic) that has the potential to increase its efficiency the most by allocating an additional hour. It initializes best topic as -1 and best increase as 0 to keep track of the task with the highest efficiency increase. The function iterates over each task index i from 0 to n-1. For each task, it checks if the allocated hours (hours[i]) are less than the equal allocation (H // n + 1). If the condition is satisfied, it calculates the increase in efficiency per hour as (e[i][hours[i] + 1] - e[i][hours[i]]) / (hours[i] + 1 + 1e-9). The 1e-9 is added to avoid division by zero. If the calculated

increase is higher than the current best increase, it updates best increase and best topic with the new values. After iterating through all the tasks, the function identifies the task (best topic) that has the highest potential efficiency increase. It increments the allocated hours for the best topic task by 1. This process is repeated until all the remaining hours are distributed among the tasks.

After doing this, I basically repeated everything from the first problem, calculated relative distance, found all the statistics required and plotted the histogram.

```
42  def generate_instances(num_instances, n, H, mark_range, hour_range):
43      instances = []
44      for _ in range(num_instances):
45          e = [[random.randint(*mark_range) for _ in range(H+1)] for _ in range(n)]
46          instances.append((n, H, e))
47      return instances
48
```

Generating 1000 random instances.

```
50  def compute_relative_distances(instances):
51      dp_distances = []
52      greedy_distances = []
53      for instance in instances:
54          n, H, e = instance
55          dp_hours = dp_allocation(H, n, e)
56          greedy_hours = allocate_hours(H, n, e)
57
58          dp_dist = [e[i][dp_hours[i]] for i in range(n)]
59          greedy_dist = [e[i][greedy_hours[i]] for i in range(n)]
60
61          dp_distances.extend(dp_dist)
62          greedy_distances.extend(greedy_dist)
63
64      relative_distances = [(dp_distances[i] - greedy_distances[i]) / dp_distances[i] for i in range(len(dp_distances))]
65      return relative_distances
66
```

Computing the relative distance.

```python
def calculate_statistics(relative_distances):
    mean = np.mean(relative_distances)
    std_dev = np.std(relative_distances)
    median = np.median(relative_distances)
    max_dist = np.max(relative_distances)

    num_outliers = int(0.05 * len(relative_distances))
    outliers = np.partition(relative_distances, -num_outliers)[-num_outliers:]
    outliers_mean = np.mean(outliers)

    return mean, std_dev, median, max_dist, outliers_mean


1 usage
def sort_by_value(instance):
    n, H, e = instance
    value_densities = [e[i][H] / (H+1) for i in range(n)]
    sorted_indices = np.argsort(value_densities)[::-1]
    sorted_instance = (n, H, [[e[i][j] for j in range(H+1)] for i in sorted_indices])
    return sorted_instance
```

Calculating Statistics.

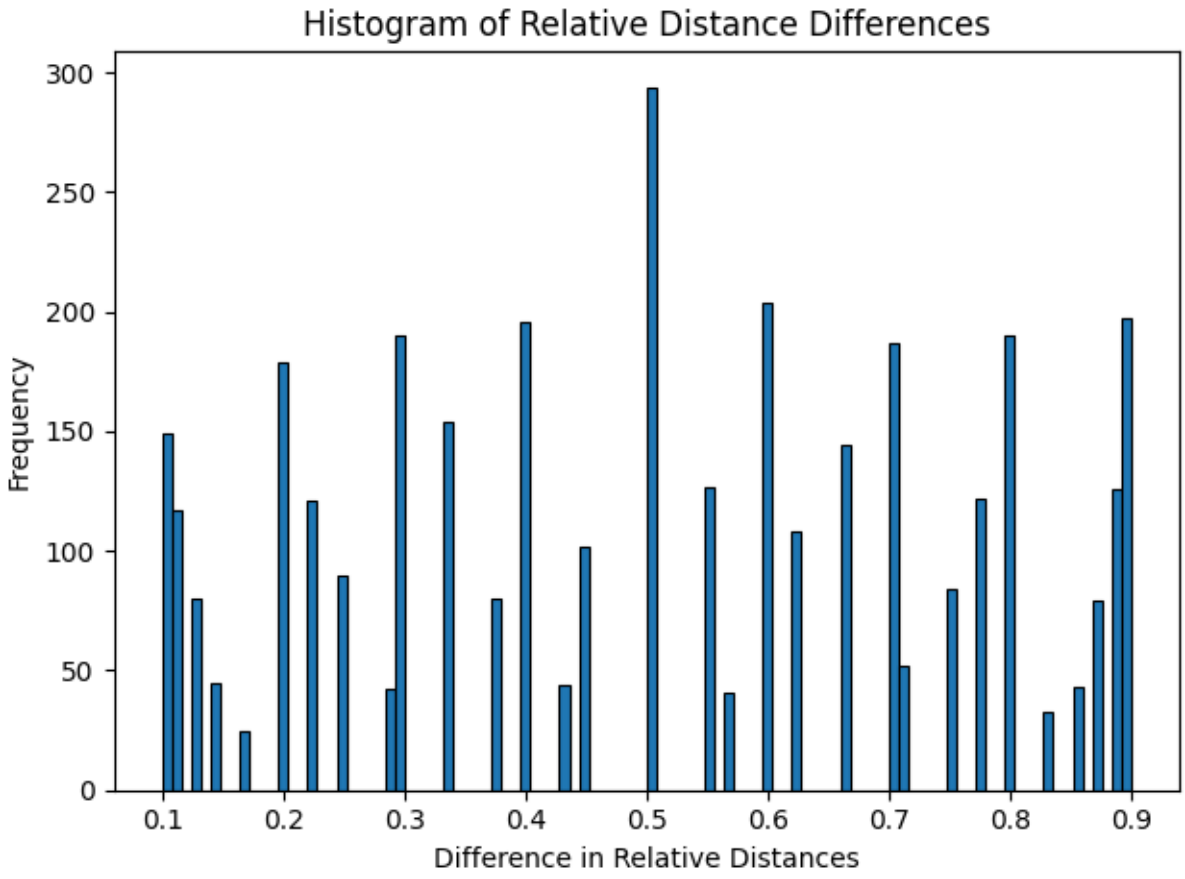
```python
# Greedy by Value Density
greedy_by_value_density_instances = sorted(instances, key=lambda x: sort_by_value(x))
greedy_by_value_density_relative_distances = compute_relative_distances(greedy_by_value_density_instances)
greedy_by_value_density_mean, greedy_by_value_density_std_dev, greedy_by_value_density_median, \
    greedy_by_value_density_max_dist, greedy_by_value_density_outliers_mean =\
    calculate_statistics(greedy_by_value_density_relative_distances)

print("Greedy by Value Density - Mean:", greedy_by_value_density_mean)
print("Greedy by Value Density - Standard Deviation:", greedy_by_value_density_std_dev)
print("Greedy by Value Density - Median:", greedy_by_value_density_median)
print("Greedy by Value Density - Maximum Relative Distance:", greedy_by_value_density_max_dist)
print("Greedy by Value Density - Mean of Outliers:", greedy_by_value_density_outliers_mean)

# Plotting Histogram
positive_relative_distances = [d for d in relative_distances if d > 0]
```

Greedy by Value Statistics

```
# Plotting Histogram
positive_relative_distances = [d for d in relative_distances if d > 0]

plt.hist(positive_relative_distances, bins=100, edgecolor='black', linewidth=0.8)
plt.xlabel('Difference in Relative Distances')
plt.ylabel('Frequency')
plt.title('Histogram of Relative Distance Differences')
plt.tight_layout()
plt.show()
```

Histogram Function

Here are the results:



```
Mean: 0.3648807142857143
Standard Deviation: 0.3220809026248998
Median: 0.3333333333333333
Maximum Relative Distance: 0.9
Mean of Outliers: 0.8976444444444446
Greedy by Value Density - Mean: 0.3648807142857143
Greedy by Value Density - Standard Deviation: 0.3220809026248998
Greedy by Value Density - Median: 0.3333333333333333
Greedy by Value Density - Maximum Relative Distance: 0.9
Greedy by Value Density - Mean of Outliers: 0.8976444444444446
```

3-rd Problem: Minimum Cost Path (MCP).

Firstly I took the Dynamic Programming approach.

```python
def dp_min_cost(L, C, grid):
    m = [[0] * C for _ in range(L)]
    m[0][0] = grid[0][0]

    for i in range(1, L):
        m[i][0] = m[i-1][0] + grid[i][0]

    for j in range(1, C):
        m[0][j] = m[0][j-1] + grid[0][j]

    for i in range(1, L):
        for j in range(1, C):
            m[i][j] = min(m[i][j-1], m[i-1][j-1], m[i-1][j]) + grid[i][j]

    return m[L-1][C-1]
```

The function initializes the matrix m of size L x C with zeros. This matrix will store the minimum cost values for reaching each cell. It sets the value of m[0][0] to the cost of the first cell in the grid. It iterates over the rows of the grid (from index 1 to L-1) and sets the minimum cost for each cell in the first column (m[i][0]) by adding the cost of the current cell to the minimum cost of the cell above it (m[i-1][0]). It iterates over the columns of the grid (from index 1 to C-1) and sets the minimum

cost for each cell in the first row (m[0][j]) by adding the cost of the current cell to the minimum cost of the cell to its left (m[0][j-1]). It iterates over the remaining cells in the grid (starting from the second row and second column) and calculates the minimum cost for each cell (m[i][j]) by taking the minimum value among the following three options:

-The cost of the cell to the left plus the cost of the current cell: m[i][j-1] + grid[i][j].

-The cost of the cell above and to the left plus the cost of the current cell: m[i-1][j-1] + grid[i][j].

-The cost of the cell above plus the cost of the current cell: m[i-1][j] + grid[i][j]. The minimum cost value is stored in m[i][j].

After doing that, I took the Greedy Algorithm approach.

```
22  def greedy_min_cost(L, C, grid):
23      path = []
24      path.append(grid[0][0])
25
26      i, j = 0, 0
27      while i != L-1 or j != C-1:
28          if i == L-1:
29              j += 1
30          elif j == C-1:
31              i += 1
32          else:
33              next_cell_costs = [grid[i][j+1], grid[i+1][j+1], grid[i+1][j]]
34              min_cost = min(next_cell_costs)
35              min_index = next_cell_costs.index(min_cost)
36
37              if min_index == 0:
38                  j += 1
39              elif min_index == 1:
40                  i += 1
41                  j += 1
42              else:
43                  i += 1
44
45          path.append(grid[i][j])
46
47      return sum(path)
```

The function initializes an empty list path to store the cost values of the path. It appends the cost of the starting cell (grid[0][0]) to the path list. It initializes i and j as 0, representing the current position in the

grid. It enters a loop that continues until the current position reaches the bottom-right cell (L-1, C-1). Within each iteration, it checks the current position and determines the next cell to move to based on the following conditions: If the current position is at the last row (i == L-1), it can only move horizontally to the right (j += 1). If the current position is at the last column (j == C-1), it can only move vertically downward (i += 1). If the current position is not at the last row or last column, it considers the costs of the three neighboring cells: the cell to the right (grid[i][j+1]), the cell diagonally below and to the right (grid[i+1][j+1]), and the cell below (grid[i+1][j]). It selects the minimum cost among the three options using the min function. It determines the index (min_index) of the minimum cost in the next_cell_costs list:

-If min_index is 0, it moves to the right (j += 1).

-If min_index is 1, it moves diagonally downward and to the right (i += 1 and j += 1).

-If min_index is 2, it moves downward (i += 1).

Within each iteration, it appends the cost of the current cell (grid[i][j]) to the path list.

After doing all this, I calculated all the required statistics as I did for the previous problems, found relative distance and created a histogram to display it.

```
def generate_instances(num_instances, L, C, cost_range):
    instances = []
    for _ in range(num_instances):
        grid = [[random.randint(*cost_range) for _ in range(C)] for _ in range(L)]
        instances.append((L, C, grid))
    return instances
```

Generating Instances function.

```python
def compute_relative_distances(instances):
    dp_distances = []
    greedy_distances = []
    for instance in instances:
        L, C, grid = instance
        dp_dist = dp_min_cost(L, C, grid)
        greedy_dist = greedy_min_cost(L, C, grid)

        dp_distances.append(dp_dist)
        greedy_distances.append(greedy_dist)

    relative_distances = [(dp_distances[i] - greedy_distances[i]) / dp_distances[i] for i in range(len(dp_distances))]
    return relative_distances
```

Computing relative distance

```python
def calculate_statistics(relative_distances):
    mean = np.mean(relative_distances)
    std = np.std(relative_distances)
    median = np.median(relative_distances)
    max_dist = max(relative_distances)
    outliers = np.percentile(relative_distances, 95)
    outliers_mean = np.mean([dist for dist in relative_distances if dist > outliers])

    return mean, std, median, max_dist, outliers_mean
```

Calculate the statistics.

```python
# Parameters
num_instances = 1000
L = 5  # Number of rows
C = 5  # Number of columns
cost_range = (1, 10)  # Range of costs for each cell

# Generate instances
instances = generate_instances(num_instances, L, C, cost_range)

# Compute relative distances
relative_distances = compute_relative_distances(instances)

# Calculate statistics
mean, std, median, max_dist, outliers_mean = calculate_statistics(relative_distances)
```

Generating 1000 different instances.

```
print("Mean:", mean)
print("Standard Deviation:", std)
print("Median:", median)
print("Maximum Relative Distance:", max_dist)
print("Mean of Outliers (5% highest):", outliers_mean)

# Greedy by Value
greedy_by_value_instances = sorted(instances, key=lambda x: np.sum(x[2]), reverse=True)
greedy_by_value_relative_distances = compute_relative_distances(greedy_by_value_instances)
greedy_by_value_mean, _, _, _, _ = calculate_statistics(greedy_by_value_relative_distances)

print("Greedy by Value - Mean:", greedy_by_value_mean)

# Greedy by Value Density
greedy_by_value_density_instances = sorted(instances, key=lambda x: np.sum(x[2]) / (x[0] * x[1]), reverse=True)
greedy_by_value_density_relative_distances = compute_relative_distances(greedy_by_value_density_instances)
greedy_by_value_density_mean, _, _, _, _ = calculate_statistics(greedy_by_value_density_relative_distances)

print("Greedy by Value Density - Mean:", greedy_by_value_density_mean)
```

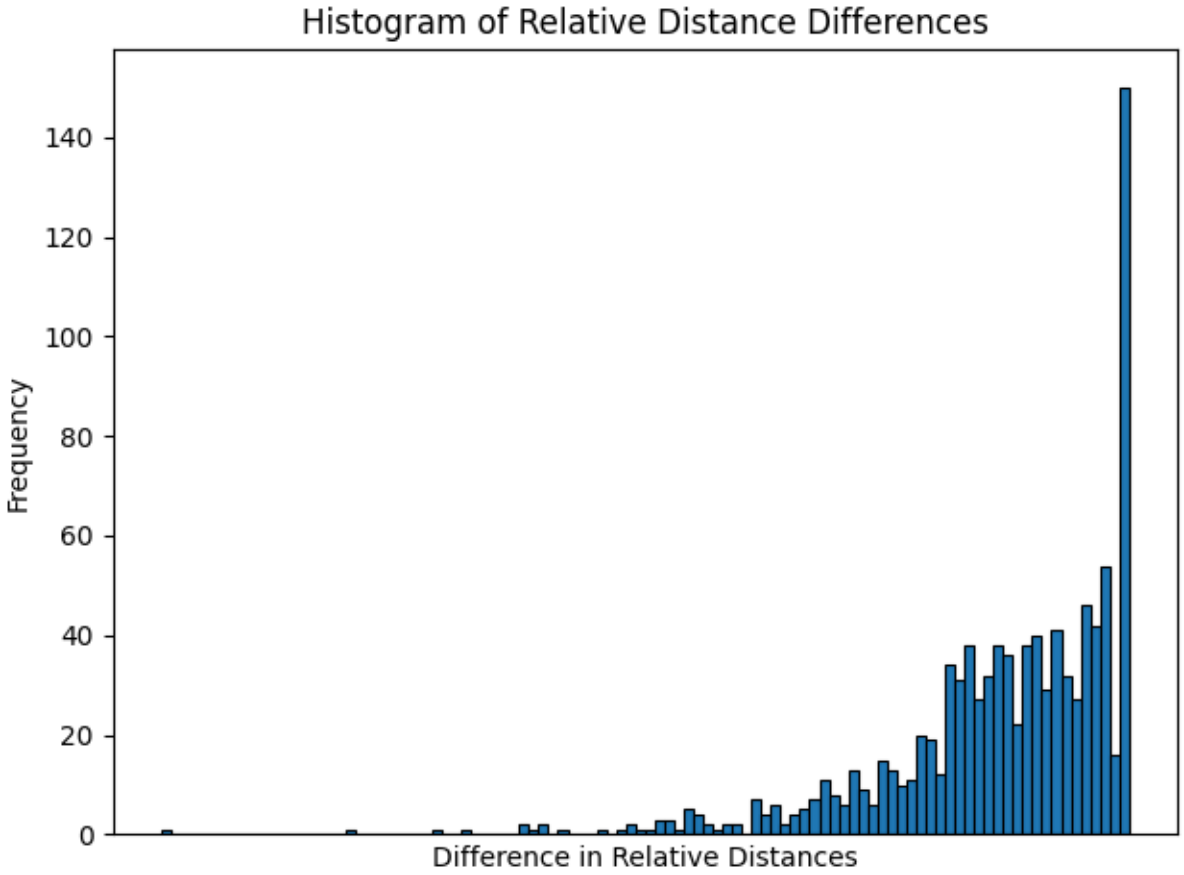
Greedy by value Statistics.

```
# Plot histogram
plt.hist(relative_distances, bins=100, edgecolor='black', linewidth=0.8)
plt.xlabel('Difference in Relative Distances')
plt.ylabel('Frequency')
plt.title('Histogram of Relative Distance Differences')
plt.xticks(np.arange(1, int(max_dist)+1, 1))
plt.tight_layout()
plt.show()
```

Histogram Function.

And here are the final results ->

```
Mean: -0.25028437513237856
Standard Deviation: 0.2353524702263245
Median: -0.19444444444444445
Maximum Relative Distance: 0.0
Mean of Outliers (5% highest): nan
Greedy by Value - Mean: -0.25028437513237856
Greedy by Value Density - Mean: -0.25028437513237856
```



4-th Problem: Maximum Value Bag 2 (MVB2).

I started with the Dynamic Programming approach.

```python

# Dynamic programming approach for the double knapsack problem
1 usage
def dynamic_programming_double_knapsack(values1, size1, values2, size2, capacity1, capacity2):
    n = len(values1)
    dp = [[0] * (capacity2 + 1) for _ in range(capacity1 + 1)]

    for i in range(1, n + 1):
        for w1 in range(capacity1, 0, -1):
            for w2 in range(capacity2, 0, -1):
                if size1[i - 1] <= w1 and size2[i - 1] <= w2:
                    dp[w1][w2] = max(dp[w1][w2],
                                     dp[w1 - size1[i - 1]][w2 - size2[i - 1]] + values1[i - 1] + values2[i - 1])

    return dp[capacity1][capacity2]
```

The variable n is assigned the length of values1, which is assumed to be the same as the length of values2 and size1 and size2. The variable dp is initialized as a 2D list of zeros with dimensions (capacity1 + 1) × (capacity2 + 1). This list will store the maximum combined values for different subproblems. The nested loops iterate over the items and the capacities in reverse order. Starting from the last item and the maximum capacities, the algorithm fills in the dp table. Inside the innermost loop, the algorithm checks if the current item can fit in both knapsacks (size1[i - 1] <= w1 and size2[i - 1] <= w2). If it can, the algorithm calculates the new combined value by adding the values of the current item to the values obtained from the remaining capacities in the two knapsacks. The maximum value is updated in the dp table at position (w1, w2) using the max function. After the loops finish, the maximum combined value is retrieved from dp[capacity1][capacity2] and returned as the result.

After that I took the GA approach.

```python
# Greedy algorithm approach for the double knapsack problem (by value)
1 usage
def greedy_algorithm_double_knapsack_value(values1, size1, values2, size2, capacity1, capacity2):
    n = len(values1)
    items = list(zip(values1, size1, values2, size2))
    items.sort(key=lambda x: x[0] + x[2], reverse=True)

    max_value = 0
    remaining_capacity1 = capacity1
    remaining_capacity2 = capacity2

    for v1, w1, v2, w2 in items:
        if w1 <= remaining_capacity1 and w2 <= remaining_capacity2:
            max_value += v1 + v2
            remaining_capacity1 -= w1
            remaining_capacity2 -= w2

    return max_value


```

The variable n is assigned the length of values1, which is assumed to be the same as the length of values2, size1, and size2. The items list is created by zipping together the values and sizes of items from both knapsacks. Each item is represented as a tuple (value1, size1, value2, size2). The items list is sorted in descending order based on the combined value of items (value1 + value2). This sorting is performed using the sort method of the items list and a lambda function as the key. Variables max value, remaining_capacity1, and remaining_capacity2 are initialized to 0 and the capacities of the knapsacks, respectively. The code iterates over the items in the sorted order. For each item, it checks if both its sizes (w1 and w2) are less than or equal to the remaining capacities of the knapsacks (remaining_capacity1 and remaining_capacity2). If the item can fit into both knapsacks, its value is added to max value, and the corresponding capacities are reduced by its sizes.

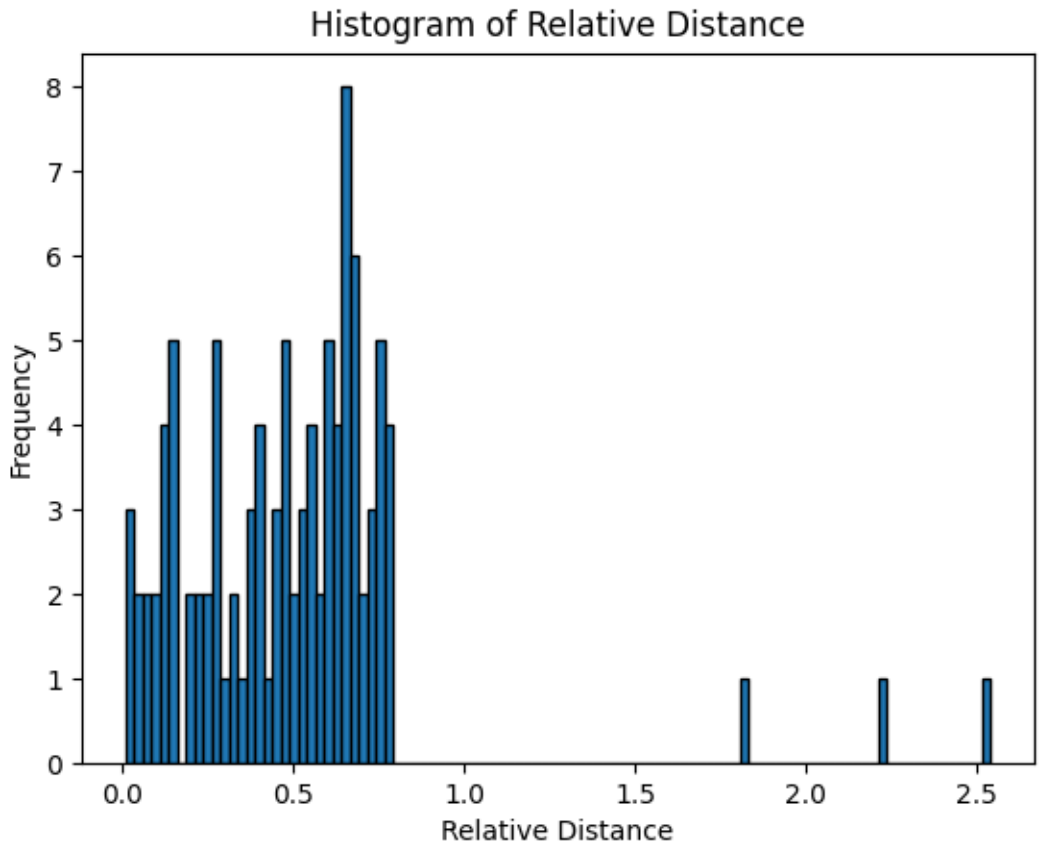
The only thing left to do now is to calculate the required statistical values and plot the histogram for the relative distance.

```
# Generate random instances for testing
np.random.seed(42)
num_instances = 100
greedy_distances = []

for _ in range(num_instances):
    capacity1 = np.random.randint(10, 100)
    capacity2 = np.random.randint(10, 100)
    weights = np.random.randint(1, 20, size=50)
    values = np.random.randint(1, 100, size=50)

    knapsack1_greedy, knapsack2_greedy, greedy_value = greedy_double_knapsack(capacity1, capacity2, weights, values)
    knapsack1_dp, knapsack2_dp, dp_value = dynamic_programming_double_knapsack(capacity1, capacity2, weights, values)

    relative_distance = abs(greedy_value - dp_value) / dp_value
    greedy_distances.append(relative_distance)
```

Random Instance Generating Function and Statistic Calculation.

```python
# Calculate statistics
mean = np.mean(greedy_distances)
std_dev = np.std(greedy_distances)
median = np.median(greedy_distances)
max_distance = np.max(greedy_distances)

print("Mean:", mean)
print("Standard Deviation:", std_dev)
print("Median:", median)
print("Maximum Relative Distance:", max_distance)
```

Finally, these are the results ->



```
Mean: 0.5044480327764955
Standard Deviation: 0.3784150039430218
Median: 0.5038422688422688
Maximum Relative Distance: 2.5397350993377485
```

Thank you for reading this, hope you don’t get bored 😊