# Laborator 1 Neagu Mihnea

## Knapsack Problem

Problem Explanation: The Knapsack Problem is a combinatorial optimization problem: Given a set of items, each with a weight and a value, determine the number of each item to include in a collection so that the total weight is less than or equal to a certain limit and the total value is maximized. It gets its name from the problem faced by someone who is constrained by a fixed-size knapsack and must fill it with the most valuable items.

## Algorithm 1: Random Generation

**Step 1**: We generate an array of 0s and 1s randomly, with the size equal to the number of objects in the list of objects (we use the numpy library in Python for its random generation function).

```
def generate_random_solution(num_objects):
    return np.random.randint(2, size=num_objects)
```

**Step 2**: Here, we will use numpy again (this time the sum function) to check if the sum of the weights of the objects in the generated solution is less than or equal to the weight limit specified by the problem (in other words, max_weight equals W from the problem).

```
def is_valid(solution, weights, max_weight):
    total_weight = np.sum(solution * weights)
    return total_weight <= max_weight
```

**Step 3**: In this step, we will return the actual value of the solution, which is the sum of the scores of the objects in the solution.

```
def evaluate_solution(solution, values):
    return np.sum(solution * values)
```

**Step 4**: Here we have the random_search function, with parameters: the number of objects, values (scores) (v), weights (w), maximum weight (W), and the number of iterations. We start by setting the best solution to none and the best score to undefined. We will also use the time library to analyze the runtime of the function. We iterate through all the iterations using a for

loop and generate a solution. Then, we check its validity using the function from step 2, and if it passes the test, we calculate its score using the function from step 3. After that, we compare the score obtained in the current iteration with scores from the past and store the best score in best_score. We also calculate the average of all solutions that pass the validity test. Finally, we calculate the runtime of the search (subtracting the start time from the end time). In the end, we return the best solution, the best score, the average score, and the runtime.

```
def random_search(num_objects, values, weights, max_weight, iterations):
    best_solution = None
    best_score = float('-inf')
    valid_scores = []

    start_time = time.time()
    for _ in range(iterations):
        solution = generate_random_solution(num_objects)
        if is_valid(solution, weights, max_weight):
            score = evaluate_solution(solution, values)
            valid_scores.append(score)  # Collect scores of valid solutions
            if score > best_score:
                best_solution = solution
                best_score = score

    end_time = time.time()
    runtime = end_time - start_time

    # Calculate average score of valid solutions
    average_valid_score = sum(valid_scores) / len(valid_scores) if valid_scores else 0

    return best_solution, best_score, runtime, average_valid_score
```

**Step 5**: Parsing function for a .txt file to extract and use data from it.

```python
def parse_knapsack_data(file_path):
    values = []
    weights = []
    max_weight = None
    num_objects = None

    with open(file_path, 'r') as file:
        lines = file.readlines()

        # Extracting num_objects from the first line
        num_objects = int(lines[0])

        # Iterating through the rest of the lines
        for line in lines[1:]:
            parts = line.split()
            if len(parts) == 3:
                values.append(int(parts[1]))
                weights.append(int(parts[2]))
            elif len(parts) == 1:
                max_weight = int(parts[0])

    return num_objects, values, weights, max_weight
```

**Step 6**: Calling the parsing function to obtain the data, and then calling the function from step 4 to obtain and print the best solution, value, and runtime.

```python
'''values = np.array([10, 5, 15, 7, 6])
weights = np.array([2, 3, 5, 7, 1])
max_weight = 10
'''


iterations = 50
num_objects, values, weights, max_weight = parse_rucksack_data("data.txt")
best_solution, best_score, runtime, average_valid_score = random_search(num_objects, values, weights, max_weight, iterations)


print("Best Solution:", best_solution)
print("Best Score:", best_score)
print("Average score:", average_valid_score)
print("Runtime:", runtime, "seconds")
```

1. **Datatable Random Search**

| Problem Instance | k | Average value(score) | Best Value | Nr of executions | Average runtime |
|---|---|---|---|---|---|
| Rucsac-20.txt | 50 | 413.76 | 536 | 10 | 0.00099 |
| | 100 | 395.07 | 521 | | 0.00118 |
| | 500 | 401.16 | 617 | | 0.00599 |
| | 10000 | 413.54 | 674 | | 0.11244 |
| Rucsac-200.txt | 50 | 121162.9090 | 130874 | | 0.00145 |
| | 100 | 123366.6938 | 130706 | | 0.00199 |
| | 500 | 123393.9722 | 132090 | | 0.00894 |
| | 10000 | 122938.6474 | 132993 | | 0.19202 |
| `num_objects = 5`<br>`values = np.array([10, 5, 15, 7, 6])`<br>`weights = np.array([2, 3, 5, 7, 1])`<br>`max_weight = 10` | 50 | 16.3125 | 31 | | 0.0 |
| | 100 | 14.2253 | 30 | | 0.00100 |
| | 500 | 16.3312 | 31 | | 0.00450 |
| | 10000 | 16.3233 | 31 | | 0.09537 |

**Data Interpretation:** When working with larger datasets (such as "Rucsac-20.txt" and "Rucsac-200.txt"), we have observed that generally, the more executions of random search we perform (i.e., the more attempts we make), the higher the chance of finding a better solution (i.e., the best value).

However, it is important to realize that as the number of attempts increases, the time required to find these better solutions also increases. Therefore, there is a trade-off between the quality of the solution and the time required to find it.

For smaller scenarios (e.g., "num_objects = 5"), we observed that random search did not lead to significant improvements in solution quality as the number of executions increased. This may be because the problem is small enough to be efficiently solved with a small number of attempts.

## 2. Algorithm SAHC(Steepest Ascent Hill-Climbing)

**Step 1**: Reuse the functions from random search.

```python
import numpy as np
import time

def generate_random_solution(num_objects):
    return np.random.randint(2, size=num_objects)


def is_valid(solution, weights, max_weight):
    total_weight = np.sum(solution * weights)
    return total_weight <= max_weight


def evaluate_solution(solution, values):
    return np.sum(solution * values)
```

**Step 2**: Create a function that generates the neighborhood of a solution by performing swaps among the bits of this solution.

```python
def get_neighborhood(current_solution, p):
    neighborhood = []
    num_objects = len(current_solution)

    for i in range(num_objects):
        neighbor = np.copy(current_solution)
        neighbor[i] = 1 - neighbor[i]  #bit swap
        neighborhood.append(neighbor)

    return neighborhood
```

**Step 3**: Create the function for SAHC (Simulated Annealing Hill Climbing), selecting a solution, determining all neighbor solutions using the neighborhood function, checking how suitable the solution is compared to its neighbors, if we find a better solution, we continue with it, becoming the new current solution (c), otherwise, we save the current solution (c) and move on to a new randomly chosen solution. When the maximum number of evaluations is reached, we choose the best solution among those saved.

```python
def sahc(num_objects, values, weights, max_weight, iterations):
    best_score = float('-inf')
    valid_scores = []
    valid_weights = []
    start_time = time.time()

    for k in range(iterations):
        current_solution = generate_random_solution(num_objects)
        current_score = evaluate_solution(current_solution, values)

        for _ in range(iterations):
            neighborhood = get_neighborhood(current_solution)
            found_better = False
            for neighbor in neighborhood:
                if is_valid(neighbor, weights, max_weight):
                    neighbor_score = evaluate_solution(neighbor, values)
                    if neighbor_score > current_score:
                        current_solution = neighbor
                        current_score = neighbor_score
                        found_better = True
                        break  # Move to the next iteration if a better neighbor is found
            if not found_better:
                break  # Break out if no better neighbor is found

        # Update best solution and score if necessary
        if current_score > best_score:
            best_solution = current_solution
            best_score = current_score
```

```python
        # Collect scores and weights of valid solutions
        if is_valid(current_solution, weights, max_weight):
            valid_scores.append(current_score)
            valid_weights.append(np.sum(current_solution * weights))
    end_time = time.time()
    runtime = end_time - start_time
    # Calculate average score and weight of valid solutions
    average_score = sum(valid_scores) / len(valid_scores) if valid_scores else 0
    average_weight = sum(valid_weights) / len(valid_weights) if valid_weights else 0
    return iterations, average_score, average_weight, best_score, runtime
```

**Step 4**: Reuse the parse_rucksack_data function, as the rucksack20 and rucksack200 files remain in the same format, and call it in the main_SAHC function to obtain the desired results.

```python
def main_sahc():
    file_path = "data.txt"
    num_objects, values, weights, max_weight = parse_rucksack_data(file_path)
    iterations = 1000

    k, avg_score, avg_weight, best_score, runtime = sahc(num_objects, np.array(values), np.array(weights), max_weight, iterations)

    print("SAHC Algorithm Results:")
    print("k (iterations):", k)
    print("Average Score:", avg_score)
    print("Average Weight:", avg_weight)
    print("Best Score:", best_score)
    print("Runtime:", runtime, "seconds")

if __name__ == "__main__":
    main_sahc()
```

## 3. Datatable SAHC

| Problem instance | K | Average value(score) | Average weight | Best value | Nr of executions | Average runtime |
|---|---|---|---|---|---|---|
| `num_objects = 5`<br>`values = np.array([10, 5, 15, 7, 6])`<br>`weights = np.array([2, 3, 5, 7, 1])`<br>`max_weight = 10` | 50 | 25.3548 | 8.8709 | 37 | 10 | 0.0032 |
| | 100 | 24.3035 | 9.1964 | 43 | | 0.0051 |
| | 200 | 25.992 | 9.248 | 43 | | 0.0133 |
| | 1000 | 25.8061 | 9.0048 | 43 | | 0.0688 |
| | 10000 | 25.7399 | 8.9741 | 43 | | 0.0673 |
| Rucsac-20.txt | 50 | 533.4333 | 517.8 | 856 | | 0.0100 |
| | 100 | 532.44 | 516.34 | 790 | | 0.0243 |
| | 200 | 542.9897 | 516.5612 | 831 | | 0.0371 |
| | 1000 | 545.7178 | 516.7136 | 890 | | 0.1783 |
| | 10000 | 546.7736 | 516.1049 | 919 | | 0.1869 |
| Rucsac-200.txt | 50 | 132129.7307 | 112602.8076 | 164864 | | 0.1161 |
| | 100 | 132019.1860 | 112600.5813 | 155174 | | 0.1933 |
| | 200 | 132029.0196 | 112600.5882 | 162005 | | 0.4146 |
| | 1000 | 132027.9824 | 112602.1619 | 165009 | | 2.0220 |
| | 10000 | 131955.7888 | 112605.7888 | 164301 | | 1.9003 |

**Data Interpretation**: For the instance with "num_objects = 5", we observe that, generally, as the value of K (the number of iterations of the SAHC algorithm) increases, the average value and the average weight of the solutions increase. This could indicate a trend that the algorithm explores more of the solution space. However, the best value found does not seem to vary significantly with the increase in K, and the average execution time remains relatively constant.

For the instances "Knapsack-20.txt" and "Knapsack-200.txt", we observe a similar pattern. As K increases, the average value and average weight of the solutions increase, but the best value found and the average execution time may remain relatively stable or fluctuate slightly.

This suggests that the SAHC algorithm may have a significant impact on the quality of the average solution and the average weight but does not guarantee improvement in the best value found during the algorithm's runtime. Additionally, the execution time may be influenced by the number of iterations, increasing with it.