République Algérienne Démocratique et Populaire
Ministère de l'Enseignement Supérieur
et de la Recherche Scientifique
ECOLE SUPÉRIEURE EN INFORMATIQUE
8 Mai 1945 - Sidi-Bel-Abbès

الجمهورية الجزائرية الديمقراطية الشعبية
وزارة التعليم العالي والبحث العلمي
المدرسة العليا للإعلام الآلي
8 ماي 1945 - سيدي بلعباس

Advanced AI Report

Knapsack Optimization using Hybrid GA + Local Search

Team Members:

- Hafsa BENGHENIMA
- Lamia ABDELMALEK
- Amina GHANDOUZ
- Yamina YAHIAOUI
- Melissa BOUKRARA
- Firdaws BENAHMED

Supervised by:

- Dr. Samiha MEZRAR

Contents

1 Introduction

Optimization is a fundamental concept in computer science and operations research that involves finding the best possible solution among a set of feasible alternatives. It plays a crucial role in various real-world applications such as resource allocation, scheduling, route planning, and decision-making. Many of these problems are combinatorial in nature, meaning that the number of possible solutions grows exponentially with the size of the input, making exact methods computationally expensive or even impractical.

Among the well-known combinatorial optimization problems is the **Knapsack Problem**. It consists of selecting a subset of items, each with a given weight and value, to include in a knapsack such that the total value is maximized without exceeding the weight capacity. Despite its simple formulation, the Knapsack Problem is *NP-hard*, meaning that no polynomial-time algorithm is known to solve it optimally for large instances.

To address such complex optimization problems, researchers have developed heuristic and metaheuristic approaches that aim to provide near-optimal solutions in a reasonable amount of time. **Metaheuristic algorithms**, such as Genetic Algorithms (GA), Particle Swarm Optimization (PSO), and Local Search methods, are inspired by natural or physical processes and are particularly effective for large and complex search spaces.

In this project, we focus on the **Knapsack Optimization Problem** using a **Hybrid Genetic Algorithm combined with Local Search**. The hybrid approach leverages the global exploration capability of the Genetic Algorithm and the local refinement ability of Local Search. The goal is to improve solution quality and convergence speed by combining the strengths of both techniques. Specifically, the first phase involves applying a standard Genetic Algorithm to generate promising candidate solutions, while the second phase enhances these solutions using a Local Search procedure.

This report presents the methodology, implementation details, and experimental analysis of the proposed hybrid approach. Each step of the algorithm is described in detail, with particular emphasis on the design and functioning of the Genetic Algorithm component.

2 Problem Definition

The **Knapsack Problem** is a classic combinatorial optimization problem that serves as a benchmark for testing optimization algorithms. It can be described as follows: given a set of n items, each item i has an associated value v_i and weight w_i. The goal is to determine which items to include in a knapsack so that the total value is maximized without exceeding the knapsack's weight capacity W.

Formally, the **0/1 Knapsack Problem** can be expressed as:

- a value v_i
- a weight w_i
  and the knapsack has a maximum capacity W

We define a binary decision variable x_i for each item:

- $x_i = 0$ if the item is included in the knapsack,
- $x_i = 1$ otherwise.

The objective is to maximize the total value of the selected items:

$$maximize \sum_{i=1}^{n} v_i x_i$$

subject to the constraint:

$$\sum_{i=1}^{n} w_i x_i \leq W$$

and $x_i \in \{0, 1\}$ for all $i = 1, 2, ..., n$

In simple terms, the problem aims to find the best combination of items that gives the highest total value without exceeding the weight limit. Because the number of possible combinations grows exponentially with the number of items, solving this problem exactly becomes very difficult for large datasets. For this reason, heuristic and metaheuristic algorithms such as Genetic Algorithms are often used to find good approximate solutions efficiently.

3 Our Approach

3.1 Dataset Description

For this project, we used the 0/1 Knapsack Problem dataset available from the official FSU repository at FSU Knapsack Dataset. This dataset collection is widely used in research for benchmarking optimization algorithms.

The dataset contains eight problem instances labeled from p01 to p08. Each instance represents a different version of the knapsack problem, varying in:

- the number of items
- the values and weights assigned to each item
- the knapsack capacity.

For example, p01 corresponds to a small instance with a limited number of items, while larger instances such as p07 or p08 contain more complex configurations that are more challenging for optimization algorithms.

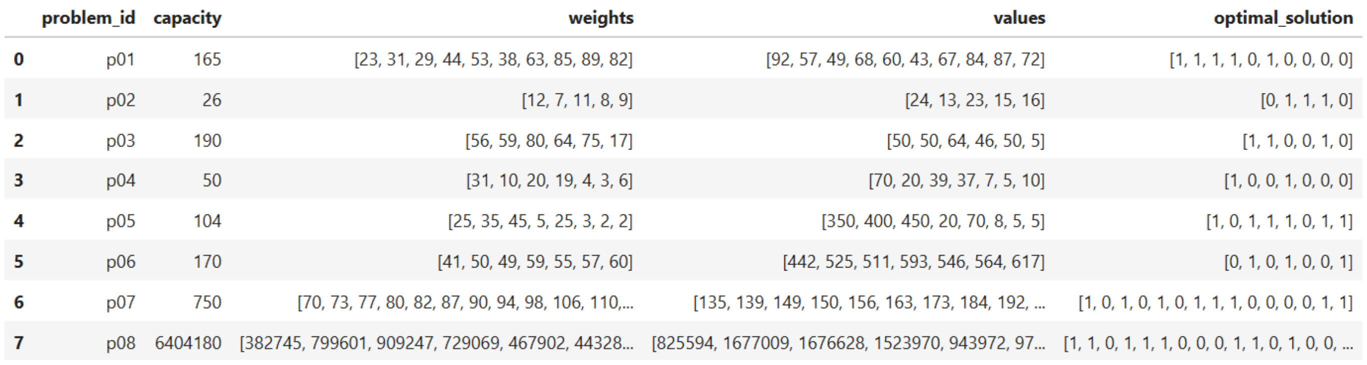
The data for each instance is provided in text files (.txt), each listing:

- a header specifying the number of items and the knapsack capacity, and
- a list of item descriptions (value and weight pairs).

In addition to these input files, the dataset also provides reference optimal solutions for each instance (files ending with _s.txt), which specify the known best possible solution and its corresponding total value. These optimal solutions serve as a benchmark to evaluate the accuracy and effectiveness of our algorithmic results.

To make the data easier to process, we used web scraping to automatically extract all the instances from the repository and compile them into a single structured DataFrame using Python. Each row in the DataFrame corresponds to one item, containing its:

- problem ID (e.g., p01, p02, ...),
- item index,
- value,
- weight,
- the capacity of the corresponding problem instance.
- the optimal solution



	problem_id	capacity	weights	values	optimal_solution
0	p01	165	[23, 31, 29, 44, 53, 38, 63, 85, 89, 82]	[92, 57, 49, 68, 60, 43, 67, 84, 87, 72]	[1, 1, 1, 1, 0, 1, 0, 0, 0, 0]
1	p02	26	[12, 7, 11, 8, 9]	[24, 13, 23, 15, 16]	[0, 1, 1, 1, 0]
2	p03	190	[56, 59, 80, 64, 75, 17]	[50, 50, 64, 46, 50, 5]	[1, 1, 0, 0, 1, 0]
3	p04	50	[31, 10, 20, 19, 4, 3, 6]	[70, 20, 39, 37, 7, 5, 10]	[1, 0, 0, 1, 0, 0, 0]
4	p05	104	[25, 35, 45, 5, 25, 3, 2, 2]	[350, 400, 450, 20, 70, 8, 5, 5]	[1, 0, 1, 1, 1, 0, 1, 1]
5	p06	170	[41, 50, 49, 59, 55, 57, 60]	[442, 525, 511, 593, 546, 564, 617]	[0, 1, 0, 1, 0, 0, 1]
6	p07	750	[70, 73, 77, 80, 82, 87, 90, 94, 98, 106, 110,...	[135, 139, 149, 150, 156, 163, 173, 184, 192, ...	[1, 0, 1, 0, 1, 0, 1, 1, 1, 0, 0, 0, 0, 1, 1]
7	p08	6404180	[382745, 799601, 909247, 729069, 467902, 44328...	[825594, 1677009, 1676628, 1523970, 943972, 97...	[1, 1, 0, 1, 1, 1, 0, 0, 0, 1, 1, 0, 1, 0, 0, ...

Data frame structure after preprocessing

This preprocessing step facilitated efficient manipulation and iteration over the data when testing our Genetic Algorithm, Local Search, and Hybrid implementations.

3.2 Genetic Algorithm (GA)

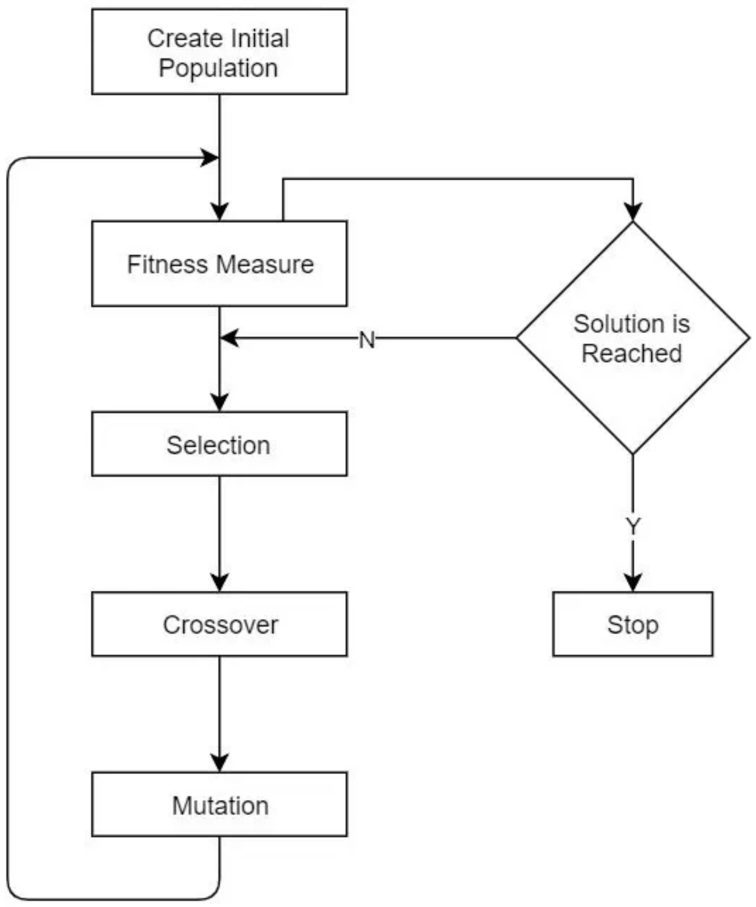
The Genetic Algorithm (GA) is a metaheuristic optimization technique inspired by the process of natural selection and evolution in biology. It is widely used to solve complex optimization problems, especially those where the search space is large and traditional exact methods are computationally expensive.

A metaheuristic is a high-level strategy designed to guide lower-level heuristics to explore and exploit the search space efficiently. Metaheuristics do not guarantee an optimal solution, but they are effective at finding good approximate solutions in reasonable time for NP-hard problems, like the 0/1 Knapsack Problem.

The GA mimics the evolutionary process by maintaining a population of candidate solutions (called chromosomes) and iteratively applying selection, crossover, and mutation operators to evolve the population toward better solutions. Over successive generations, chromosomes with higher ”fitness” (better solutions) are more likely to survive and contribute to the next generation.

The general steps of a Genetic Algorithm are as follows:

1. **Initialization** : Generate an initial population of chromosomes randomly, ensuring diversity and feasibility.
2. **Fitness Evaluation** : Define a fitness function to evaluate how ”good” each chromosome is. In the knapsack problem, this usually involves computing the total value while penalizing overweight solutions.
3. **Selection** : Choose parent chromosomes from the current population based on their fitness, favoring higher-quality solutions to produce offspring.
4. **Crossover** : Combine two parent chromosomes to produce one or more offspring, allowing the algorithm to explore new regions of the search space.
5. **Mutation** : Introduce small random changes to offspring chromosomes to maintain genetic diversity and prevent premature convergence.
6. **Replacement / Next Generation** : Form the new population by selecting the best individuals among parents and offspring, often using elitism to preserve the best solutions.
7. **Termination** – Repeat the process until a stopping criterion is met, such as a maximum number of generations or convergence of the fitness value.



3.2.1 Initialization

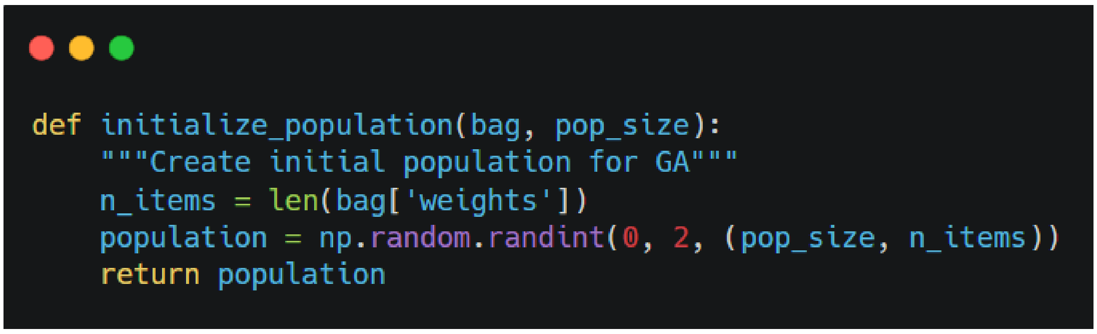
In our implementation, each potential solution to the Knapsack Problem is represented as a chromosome, following the standard binary encoding scheme used in Genetic Algorithms.

Each chromosome is a binary vector whose length corresponds to the total number of items in the dataset instance.

- A gene value of 1 indicates that the corresponding item is included in the knapsack.
- A gene value of 0 means that the item is excluded.

For example, if a problem instance contains 10 items, a chromosome such as $[1, 0, 1, 1, 0, 0, 1, 0, 0, 1]$ represents a possible selection of items where the 1st, 3rd, 4th, 7th, and 10th items are included in the knapsack. The initial population is generated randomly, with each chromosome constructed by randomly assigning 0s and 1s while maintaining diversity across individuals. However, to ensure that the total weight of each generated solution does not exceed the knapsack's capacity, items are selected based on their weight sizes—this helps keep the generated solutions feasible.

The population size was set to 100, meaning that at the start of the algorithm, 100 distinct chromosomes are generated, representing 100 different possible solutions to the problem. This provides sufficient diversity for the evolutionary process and ensures that the search space is well explored in the early generations.



```python
def initialize_population(bag, pop_size):
    """Create initial population for GA"""
    n_items = len(bag['weights'])
    population = np.random.randint(0, 2, (pop_size, n_items))
    return population
```

3.2.2 Fitness Function

The fitness function is a key component of the Genetic Algorithm, as it determines how well each chromosome (candidate solution) satisfies the objective of the knapsack problem. In the 0/1 Knapsack Problem, the objective is to maximize the total value of selected items while ensuring that the total weight does not exceed the knapsack's capacity.

Each chromosome is represented as a binary vector of length n, where n is the number of available items:

- $x_i = 1$ if the i-th item is included in the knapsack.
- $x_i = 0$ otherwise

Let:

- v_i be the value of item i
- w_i be the weight of item i
- C be the maximum capacity of the knapsack.

The total value and weight of a chromosome are calculated as follows:

$$V = \sum_{i=1}^{n} v_i x_i$$

$$W = \sum_{i=1}^{n} w_i x_i$$

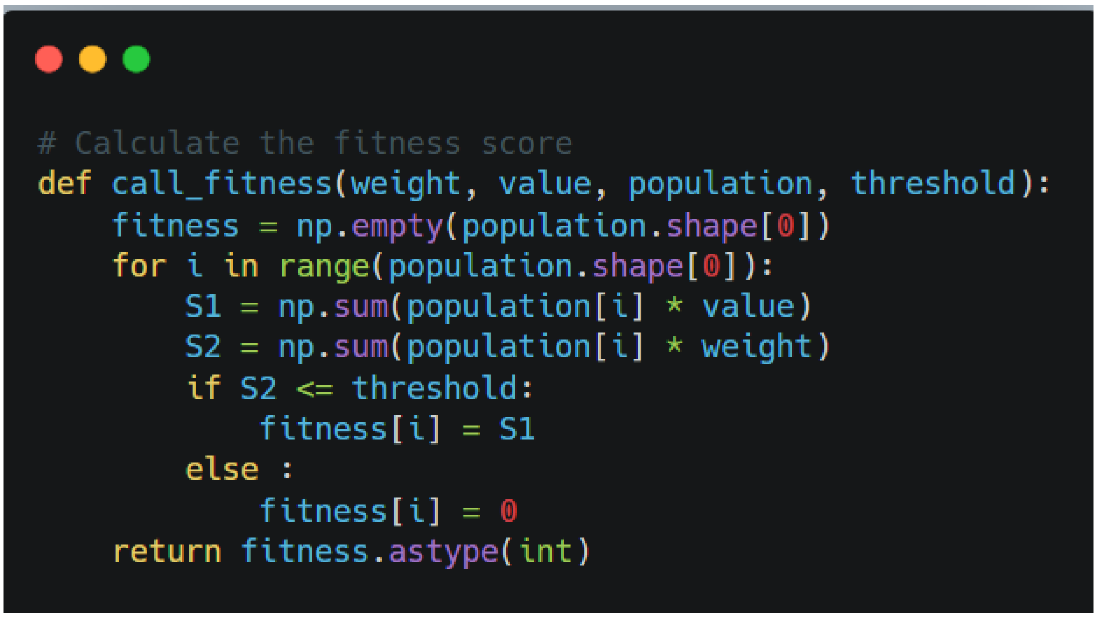
The fitness function $f(x)$ is then defined as :

$$f(x) = \begin{cases} V = \sum_{i=1}^{n} v_i x_i & \text{if } W \leq C \\ 0 & \text{if } W > C \end{cases}$$

In other words, if the total weight W of the selected items exceeds the knapsack's capacity C, the fitness is set to zero. This penalization ensures that infeasible solutions are eliminated from the search space.

By evaluating all chromosomes in the population using this fitness function, we obtain a vector of fitness scores, which will later be used to select the best-performing individuals for the next generation.

Below is the implementation of the fitness function:



```python
# Calculate the fitness score
def call_fitness(weight, value, population, threshold):
    fitness = np.empty(population.shape[0])
    for i in range(population.shape[0]):
        S1 = np.sum(population[i] * value)
        S2 = np.sum(population[i] * weight)
        if S2 <= threshold:
            fitness[i] = S1
        else :
            fitness[i] = 0
    return fitness.astype(int)
```

After executing the fitness function, we obtain a vector that contains the fitness value of each chromosome in the population.
Each element of this vector represents the total value achieved by a specific chromosome, after checking that the total weight does not exceed the knapsack's capacity. Thus, higher values correspond to better (more optimal) solutions.
An example of the resulting fitness vector is illustrated below:

```
[187 141   0 192 221 168 236 174 241 241 260 269 125 236 220 309 241 241
 241 241 184 141 241 184 266 266 241 184 258 241 217 217 192 241 141 260
 260 198 135   0 203 252   0 198 266 213 260 198 241  92 198 141 125 220
 309 172 172 184 184 192 213 217 217 217 241 241 241 258 263 266]
```

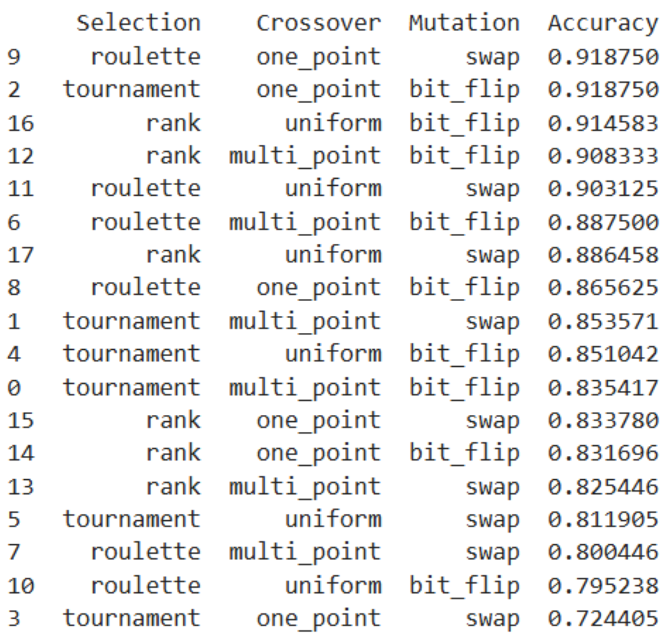
3.2.3 Selection

The selection step determines which chromosomes from the current population will be chosen as parents to produce offspring for the next generation.
The main goal of this process is to give better-performing individuals (those with higher fitness values) a higher probability of being selected, while still maintaining genetic diversity to avoid premature convergence.
There are several common selection methods in Genetic Algorithms, each with its advantages and limitations.

Selection Method	**Description**	**Advantages**	**Disadvantages**
Roulette Wheel Selection	Each individual is assigned a probability proportional to its fitness. Selection is performed randomly according to these probabilities.	Simple to implement; ensures fitter individuals have higher chances.	Can lead to loss of diversity if a few individuals dominate.
Rank Selection	Individuals are ranked according to fitness, and selection probabilities are assigned based on ranks rather than raw fitness values.	Prevents by highly fit individuals; maintains more stability.	Slower convergence; may ignore small fitness differences.
Tournament Selection	A small random group (size k) of individuals is selected, and the one with the highest fitness in the group is chosen as a parent.	Maintains diversity; flexible control via tournament size k; efficient and easy to implement.	Performance depends on tournament size; small k may slow convergence.

Comparison of Selection Methods in Genetic Algorithms

In this project, we experimented with different combinations of selection, crossover, and mutation methods to determine the most effective configuration. We compared all possible combinations and analyzed their performance based on the final fitness and convergence speed.

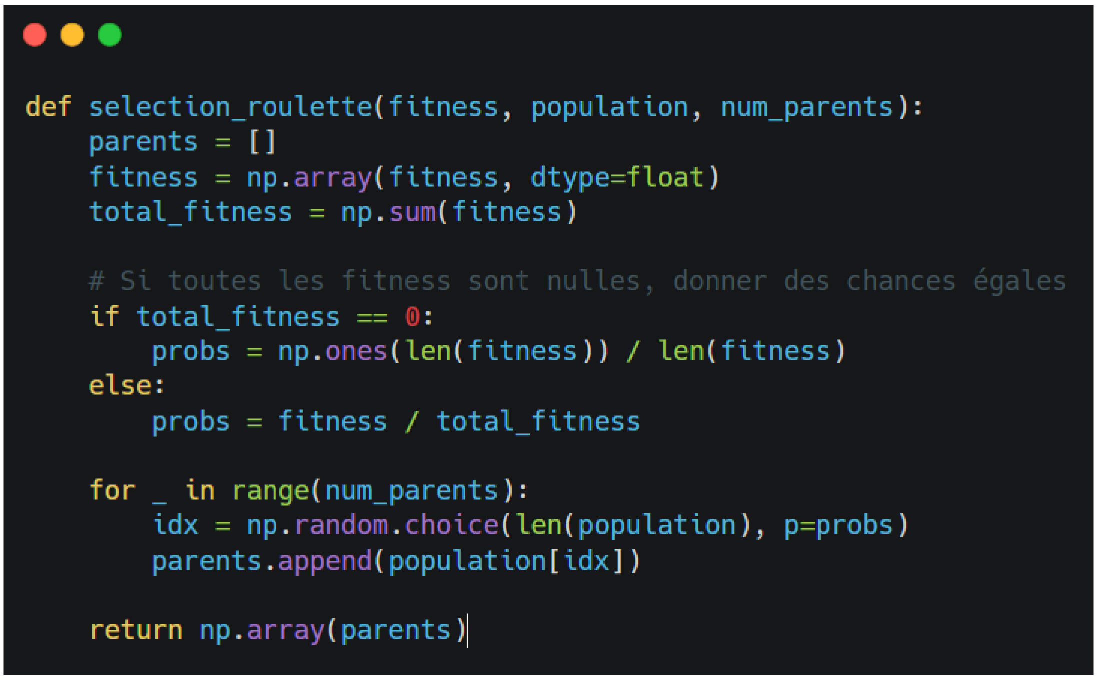


```
     Selection   Crossover  Mutation  Accuracy
9     roulette   one_point      swap  0.918750
2   tournament   one_point  bit_flip  0.918750
16        rank     uniform  bit_flip  0.914583
12        rank multi_point  bit_flip  0.908333
11    roulette     uniform      swap  0.903125
6     roulette multi_point  bit_flip  0.887500
17        rank     uniform      swap  0.886458
8     roulette   one_point  bit_flip  0.865625
1   tournament multi_point      swap  0.853571
4   tournament     uniform  bit_flip  0.851042
0   tournament multi_point  bit_flip  0.835417
15        rank   one_point      swap  0.833780
14        rank   one_point  bit_flip  0.831696
13        rank multi_point      swap  0.825446
5   tournament     uniform      swap  0.811905
7     roulette multi_point      swap  0.800446
10    roulette     uniform  bit_flip  0.795238
3   tournament   one_point      swap  0.724405
```

After evaluation, we found that the combination of **Roulette Wheel Selection**, **One-Point Crossover**, and **Swap Mutation** yielded the best results in terms of solution quality and stability. Therefore, we adopted this configuration for our final implementation.

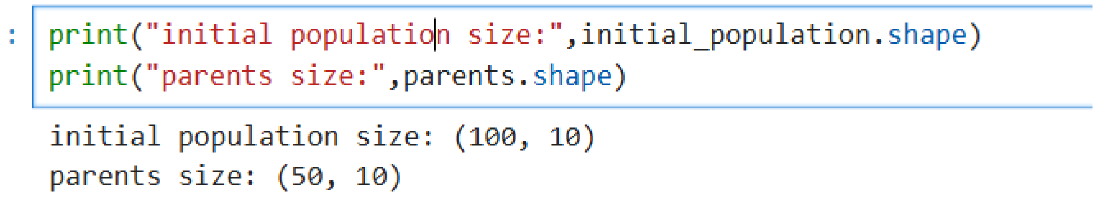
For the selection step, we used the **Roulette Wheel Selection** method, which ensures a probabilistic yet fitness-proportionate chance of selection, maintaining diversity while favoring individuals with higher fitness values.

Below is the corresponding implementation of the best selection method:



```python
def selection_roulette(fitness, population, num_parents):
    parents = []
    fitness = np.array(fitness, dtype=float)
    total_fitness = np.sum(fitness)

    # Si toutes les fitness sont nulles, donner des chances égales
    if total_fitness == 0:
        probs = np.ones(len(fitness)) / len(fitness)
    else:
        probs = fitness / total_fitness

    for _ in range(num_parents):
        idx = np.random.choice(len(population), p=probs)
        parents.append(population[idx])

    return np.array(parents)
```

After executing this function, we obtain a new subset of the population composed of the best individuals from each tournament.
For example , below is population and parents selected size for 10 items:

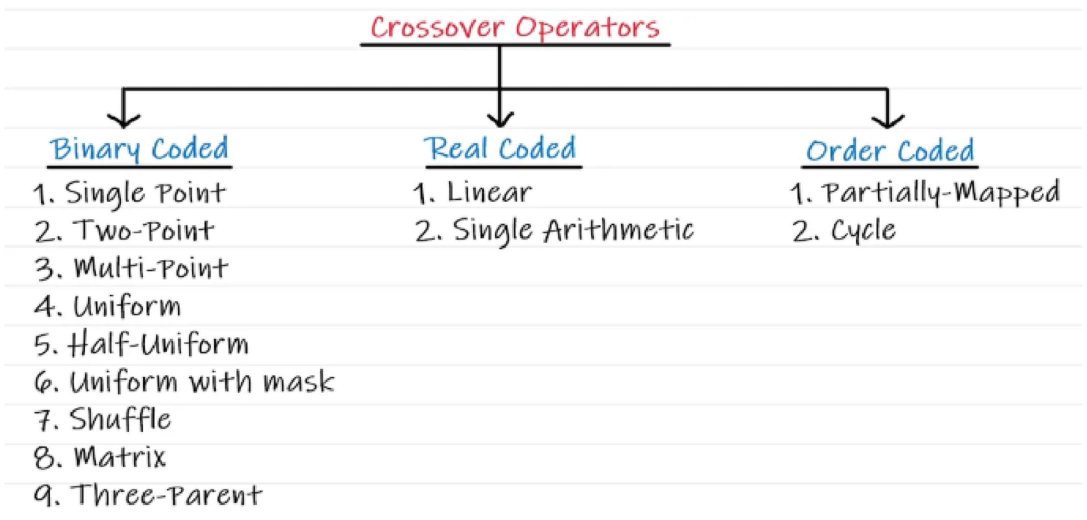


```
: print("initial population size:",initial_population.shape)
  print("parents size:",parents.shape)

  initial population size: (100, 10)
  parents size: (50, 10)
```

3.2.4 Crossover

Also called *recombination*, it is similar to reproduction where two *parents* merge to create *children*. The parents are two fit solutions and the children are new solutions. It is hoped that the best traits of each parent will be passed on to the children. The goal of crossover is to create solutions that are more fit than the originals.

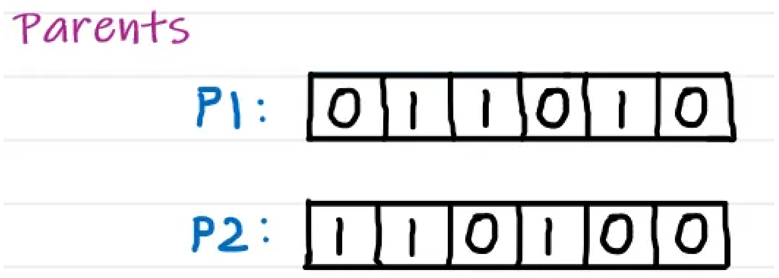
Crossover operators are generally classified into three main categories: *Binary*, *Real* and *Order Coded*.



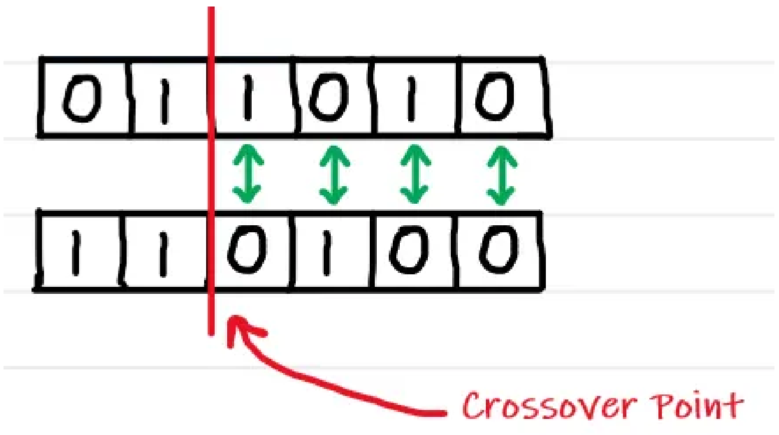
In the context of the Knapsack Problem, where solutions are encoded as binary strings, the binary crossover category is applied. This category includes several variants, typically around nine but in our case, we focus on the most commonly used ones:

3.2.4.1 Single Point

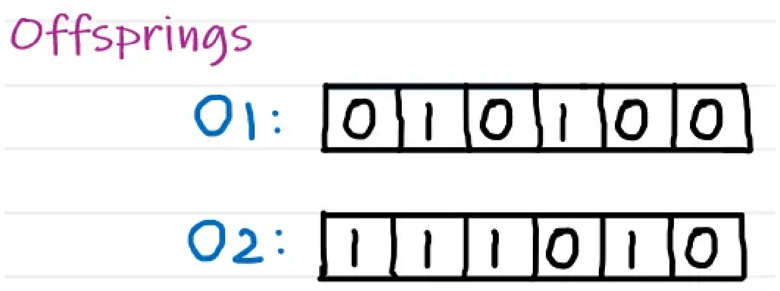
1. We select two parents for mating.



2. Select a crossover point at random and swap the bits at the right site.

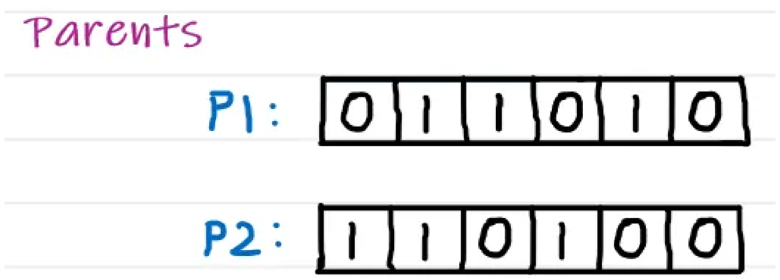


3. Now, the new offsprings generated look as follows

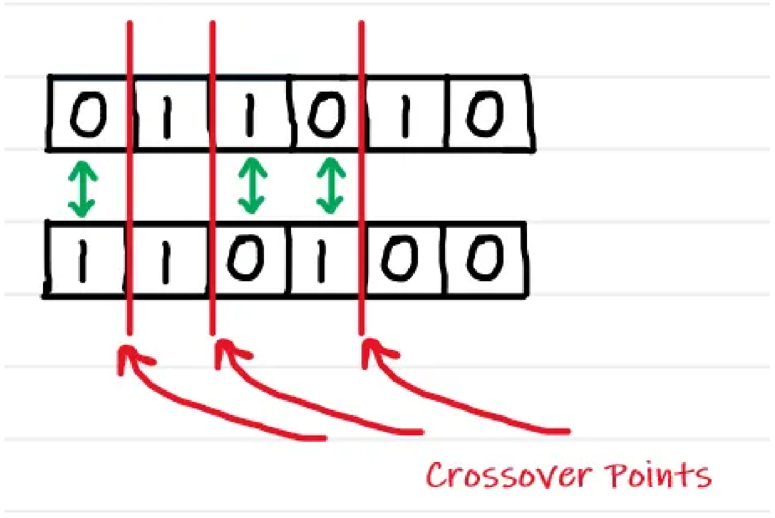


3.2.4.2 Multi Point

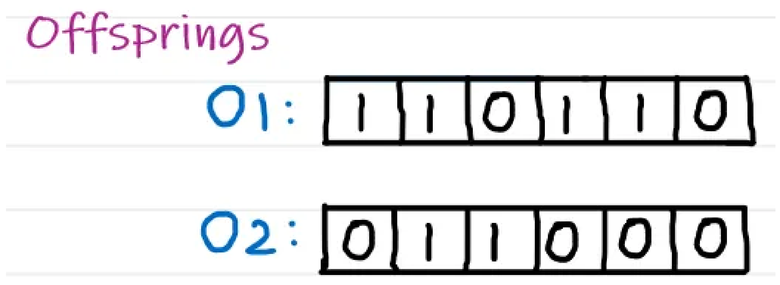
1. We select two parents for mating.



2. Select multiple crossover points at random and swap the bits at the alternate sites.

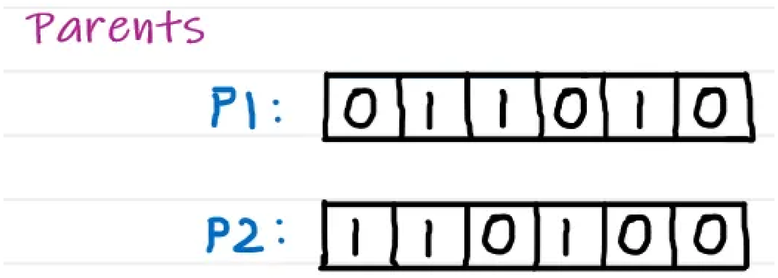


3. Now, the new offsprings generated look as follows

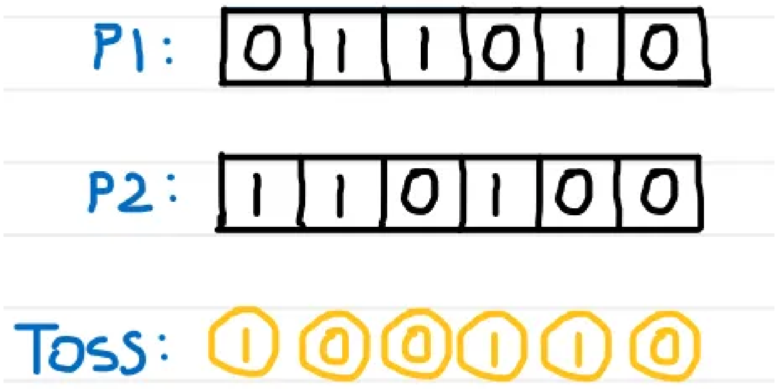


3.2.4.3 Uniform Crossover

1. We select two parents for mating.

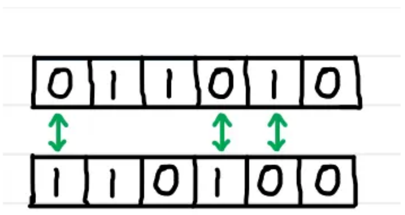


2. At each bit position of the parents, toss a coin (let H=1 and T=0).

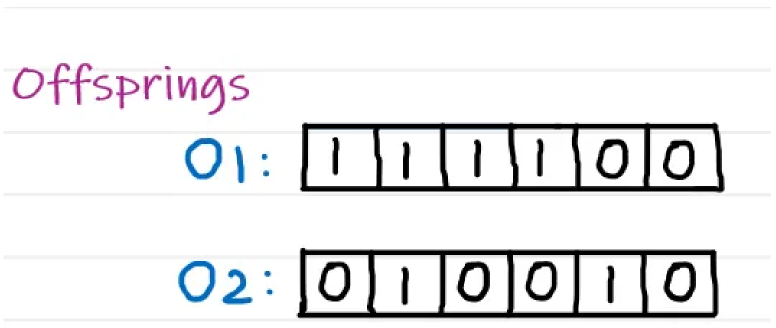


3. Following the algorithm mentioned below we'll generate both offsprings

```
if Toss=1,
 then swap the bits
if Toss=0,
 then don't swap
```



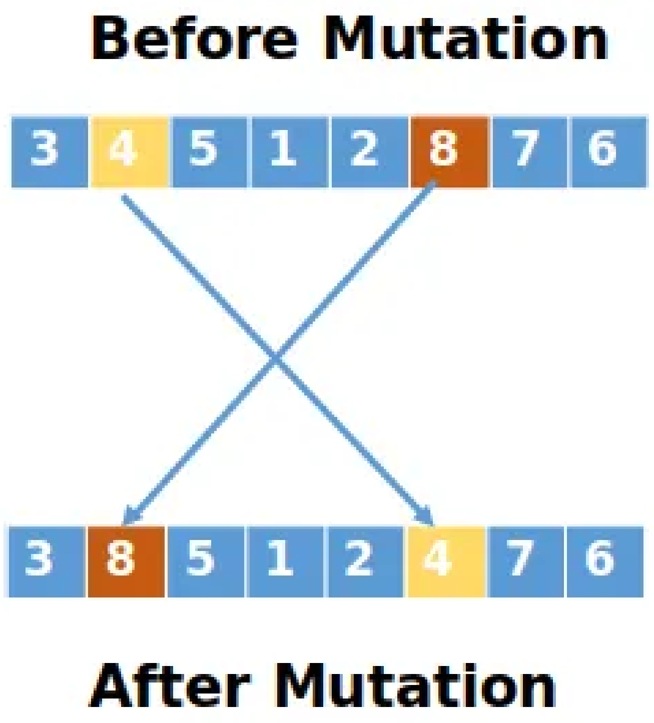
4. Now, the new offsprings generated look as follows



We chose single point crossover because it is simple, efficient and widely used for binary encoded problems like the Knapsack Problem. It allows the algorithm to combine good traits from two parents by exchanging contiguous segments of their chromosomes, promoting the inheritance of partial solutions. Compared to multi-point or uniform crossover, single-point crossover introduces enough variation while keeping offspring structurally similar to parents, which helps maintain solution stability and facilitates convergence toward high-quality solutions.

3.2.5 Mutation

It is a fundamental operation in genetic algorithms, which involves the random alteration of a small number of genes in a chromosome. This process occurs by exchanging genetic information within the chromosome. Adjusting and tuning the mutation operator is crucial for optimizing the performance of genetic algorithms and other evolutionary computation methods, here we see a chromosome example before and after a mutation



For the Knapsack Problem, two common mutation methods are applied:

1. *Bit-flip Mutation*: for binary encoded chromosomes, the selected gene or bit is flipped from 0 to 1 or 1 to 0.
2. *Swap Mutation*: where two genes in the chromosome are exchanged, which is particularly

useful when maintaining certain constraints or exploring permutations.

These operators complement crossover by exploring new solution spaces and preventing premature convergence of the algorithm.
We chose to work with swap mutation because, for the Knapsack Problem, it allows controlled exploration of the solution space while preserving feasibility. Unlike bit-flip mutation, which can arbitrarily turn an item "in" or "out" of the knapsack (potentially violating constraints or creating large disruptions), swap mutation exchanges the positions of two items, maintaining the overall structure of the solution. This helps the algorithm refine solutions without introducing excessive randomness, improving convergence toward high-quality solutions.

3.2.6 Replacement/elitism strategy

In our Genetic Algorithm, we employ an elitist replacement strategy to ensure that the best solutions are preserved across generations. Elitism works by retaining a fraction of the top-performing individuals from the current population and combining them with newly generated offspring. This prevents the loss of high-quality solutions due to stochastic effects of crossover or mutation.

In our implementation, the elitism rate determines the proportion of the population considered elite. These elite individuals are identified by sorting the population according to their fitness values and selecting the top performers. The new generation is then formed by combining the elites with the offspring and mutated offspring, followed by truncation to maintain a constant population size. This approach balances exploitation of the best solutions with exploration introduced by genetic operators, improving convergence without sacrificing diversity.

3.2.7 Stopping criterion

The stopping criterion defines when the Genetic Algorithm should terminate its iterative process of evolution.

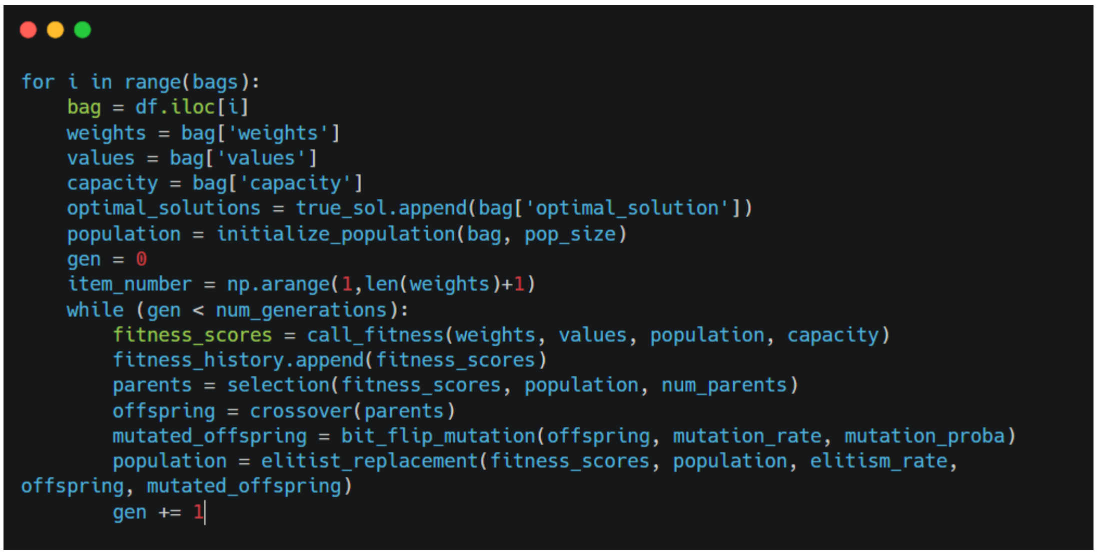
After generating the initial population, the algorithm repeatedly applies the main evolutionary operators (selection, crossover, and mutation) to produce new generations that progressively improve in fitness.
In our implementation, the stopping condition is based on a fixed number of generations. Specifically, we set:

$num\_generations = 50$ This means that the algorithm continues to evolve the population for 50 iterations, after which the process stops automatically.

At the end of these iterations, the best solution (chromosome) found so far is considered the final result.

This fixed-generation stopping rule helps ensure convergence toward a good solution while keeping the computational cost under control.

Alternative criteria, such as a convergence threshold (when fitness improvement becomes negligible), could be used, but in our case, a fixed number of generations was sufficient to achieve stable results.



```
for i in range(bags):
    bag = df.iloc[i]
    weights = bag['weights']
    values = bag['values']
    capacity = bag['capacity']
    optimal_solutions = true_sol.append(bag['optimal_solution'])
    population = initialize_population(bag, pop_size)
    gen = 0
    item_number = np.arange(1,len(weights)+1)
    while (gen < num_generations):
        fitness_scores = call_fitness(weights, values, population, capacity)
        fitness_history.append(fitness_scores)
        parents = selection(fitness_scores, population, num_parents)
        offspring = crossover(parents)
        mutated_offspring = bit_flip_mutation(offspring, mutation_rate, mutation_proba)
        population = elitist_replacement(fitness_scores, population, elitism_rate,
offspring, mutated_offspring)
        gen += 1
```

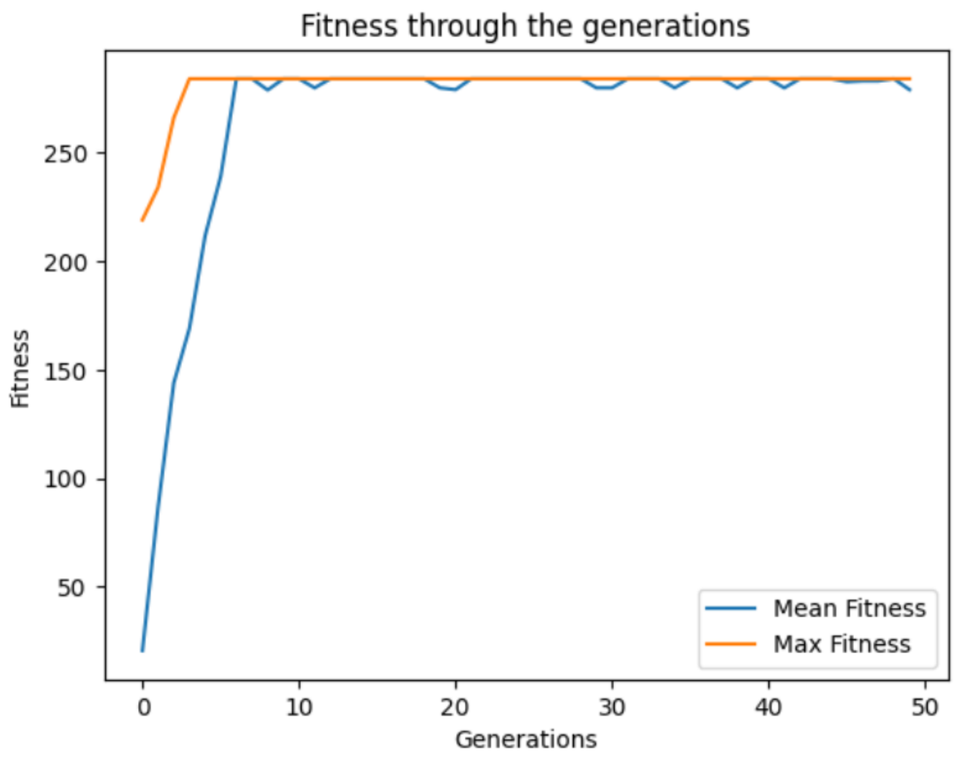
3.2.8 Results and Evaluation

After implementing all the steps of the Genetic Algorithm (GA), we applied it to the datasets provided in the knapsack_01 benchmark.

Each dataset (noted as P0, P1, ..., P7) corresponds to a different knapsack configuration with its own number of items, values, weights, and capacity.

1. **Example of Results for One Instance (P1)**
   Below is an example of the output obtained for the first dataset (P1):

```
P0 1
Fitness of the last generation:
[284 284 284 284 284 284 284 284 284 284 284 284 284 284 284 284 284 284
 284 284 284 284 284 284 284 284 284 284 284 284 284 284 284 284 284 284
 284 284 284 284 284 284 284 284 284 284 284 284 284 284   0 284 284 284
 284 284 284 284 284 284 284 284 284 284 284 284 284 284]

The optimized predicted_sol for the given inputs are:
[1 1 0 1 0 0 1 0 0 0]

Selected items that will maximize the knapsack without breaking it:
1

2

4

7
```



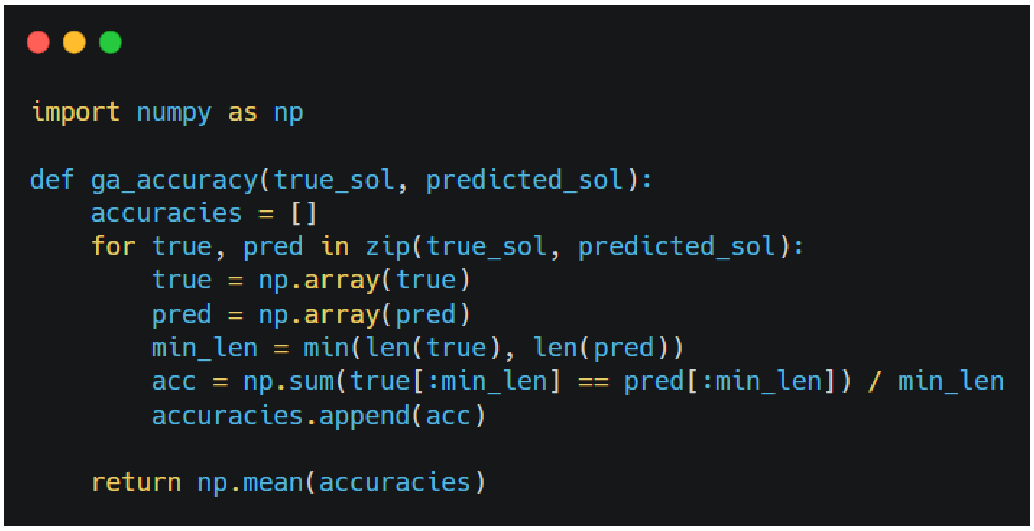
The following graph illustrates the evolution of fitness values across generations.
The blue curve represents the mean fitness of the population at each generation, while the orange curve represents the maximum fitness obtained in that generation.

As we can observe, the maximum fitness value increases rapidly in the early generations and then stabilizes after several iterations, indicating that the population has converged toward an optimal or near-optimal solution.

Meanwhile, the mean fitness curve also shows a gradual improvement, reflecting the overall enhancement of solution quality within the population as evolution progresses.

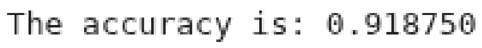
2. **Accuracy Evaluation:**
Each dataset also provides the optimal solution (known best configuration).
To evaluate the performance of our Genetic Algorithm, we compared the predicted best chromosome obtained by the GA to the optimal solution from the dataset.
The accuracy is calculated as the ratio of correctly selected items between the predicted and the optimal solution:

$$\text{Accuracy} = \frac{\text{Number of correctly selected items}}{\text{Total number of items in the optimal solution}}$$



```python
import numpy as np

def ga_accuracy(true_sol, predicted_sol):
    accuracies = []
    for true, pred in zip(true_sol, predicted_sol):
        true = np.array(true)
        pred = np.array(pred)
        min_len = min(len(true), len(pred))
        acc = np.sum(true[:min_len] == pred[:min_len]) / min_len
        accuracies.append(acc)

    return np.mean(accuracies)
```

However, due to the stochastic nature of the Genetic Algorithm — particularly the random initialization, crossover, and mutation — the obtained best solution may slightly differ between runs.
This randomness explains why the accuracy can vary:
sometimes the algorithm reaches the exact optimal solution, while in other cases, it finds a near-optimal (approximate) one with very close fitness values.



```
The accuracy is: 0.918750
```

3. **Discussion**
   Overall, the GA demonstrated a strong capacity to find high-quality solutions across all instances of the knapsack dataset.
   Even when not reaching the exact optimal solution, it consistently produced results with close-to-optimal total values, showing good convergence behavior over generations and maintaining a healthy level of population diversity.

3.3 Local Search

3.3.1 Definition

Local search is an *iterative optimization technique* that starts from an initial solution and explores its neighborhood to find improved solutions. Unlike population-based algorithms (like GA), LS focuses on intensifying search around a candidate solution to quickly converge to high-quality solutions.

3.3.2 How does Local Search work

In *local search*, the algorithm *randomly* selects an initial state. It then examines all neighboring states and moves to the neighbor with the highest fitness. This is considered a *local approach* because it only evaluates the immediate neighbors, not the entire search space. The process is repeated iteratively, moving from one state to its best neighbor.

A state where no neighbor improves the fitness is called a *locally optimal point*, which is a common challenge in local search because the algorithm may become trapped without reaching the global optimum. To address this, there are two main strategies:

- **Restart:** The algorithm restarts from a different state, or from the same state with slightly modified criteria. This can lead to exploring different neighbors and potentially reaching a different locally optimal point. However, it may still repeatedly converge to local optima.
- **Iterative Improvement:** In iterative local search, the algorithm modifies the final state to explore new solutions. This approach is effective for problems with a well-defined configuration space, where each state represents a distinct solution. When a *locally optimal point* is reached, small changes are made to the configuration to move to a new state, and the process is repeated. Termination can be handled in two ways:
  - **Best solution found:** Stop when no better solution exists, approximating the global optimum. In many cases, the search space is too large to guarantee the true global optimum.
  - **Time limit:** Stop after a predefined number of iterations or time. The best solution found at termination is treated as the approximate optimal solution.

3.3.3 Representation and Fitness Evaluation

Each solution (or individual) in the Knapsack Problem is represented as a *binary string*, where 1 indicates that the corresponding item is included in the knapsack, and 0 indicates it is excluded. This representation allows straightforward manipulation by genetic operators (crossover and mutation) as well as by local search.

The *fitness function* evaluates the quality of each solution by computing the total value of the selected items, while ensuring that the total weight does not exceed the knapsack capacity. Solutions that violate the weight constraint are considered infeasible and are assigned a fitness of 0. This ensures that only valid solutions contribute to the search and guides the algorithm toward feasible, high-value solutions.

3.3.4 Neighborhood Definition

In *Local Search*, the neighborhood of a solution defines the set of candidate solutions that can be reached from the current solution through a small modification. For the *Knapsack Problem*, each solution is represented as a binary string, so a neighbor is generated by flipping a single bit changing a 0 to 1 or a 1 to 0.

This simple modification produces a feasible variation of the current solution, enabling the algorithm to explore the search space incrementally. By evaluating all neighbors of a solution, the algorithm identifies improvements without making large disruptive changes that could violate problem constraints.

The neighborhood definition is crucial because it determines the scope of the search:

- A small, local neighborhood ensures fine-grained exploration and faster convergence.
- Larger or more complex neighborhoods (e.g., flipping multiple bits or swapping items) increase diversity but may slow convergence.

In our implementation, the single bit flip neighborhood is sufficient to efficiently explore feasible solutions while preserving the simplicity of the search process.

3.3.5 Local Search Procedure

The *Local Search algorithm* is applied iteratively to refine candidate solutions for the *Knapsack Problem.* The procedure aims to improve an initial solution by exploring its local neighborhood and moving toward better solutions based on the fitness function. The main steps are as follows:

1. **Initialization:** The algorithm begins with a randomly generated or feasible binary solution representing item selection. Each bit corresponds to whether an item is *included 1* or *excluded 0* from the knapsack. In the implementation, the initial population is created using the function *initialize_population_ls()* which generates a set of random binary vectors.
2. **Generate Neighbors:** For each bit position in the current solution, the algorithm creates a neighbor by flipping that bit ($0 \rightarrow 1$ or $1 \rightarrow 0$). This is done within the $local_{s}earch()function, whereeachneighborrepresentsaslightlydifferentselectionofitems.$
3. **Evaluate Neighbors:** Each neighbor's fitness is evaluated using the *call_fitness()* function, which computes the total value of selected items while ensuring that the total weight does not exceed the knapsack's capacity. If a neighbor exceeds the capacity constraint, it is assigned a fitness of 0, marking it as infeasible.
4. **Move to the Best Neighbor:** After evaluating all neighbors, the algorithm selects the one with the highest fitness. If this neighbor's fitness is greater than the current solution's fitness, it becomes the new current solution. This process continues iteratively to improve solution quality.
5. **Termination:** The algorithm terminates when no neighbor provides an improvement means it reaches a locally optimal point or when the maximum number of iterations **max_iter_ls = 50** is reached. This prevents infinite loops and controls computational time.
6. **Track Progress:** Throughout the iterations, the algorithm records the fitness values of each improvement step. This history is used to plot the mean and maximum fitness evolution over time, allowing visualization of the algorithm's convergence behavior.

Algorithm 1 Local Search Algorithm for the Knapsack Problem

```
Input :
        weights[ ] // item weights
        values[ ] // item values
        capacity // maximum allowed weight
        max_iter_ls // maximum iterations (50)
Output: best_solution, best_fitness
1  Procedure LocalSearch(weights, values, capacity, max_iter_ls):
2      Initialize current_solution ← random binary vector
       Evaluate current_fitness ← Fitness(current_solution)
3      for iter ← 1 to max_iter_ls do
4          improved ← False
5          foreach bit position j in current_solution do
6              neighbor ← copy(current_solution)
               neighbor[j] ← 1 − neighbor[j]  // flip bit j
7              if TotalWeight(neighbor) ≤ capacity then
8                  neighbor_fitness ← Fitness(neighbor)
9                  if neighbor_fitness > current_fitness then
10                     current_solution ← neighbor
                       current_fitness ← neighbor_fitness
                       improved ← True
11                 end
12             end
13         end
14         if improved = False then
15             break // no better neighbor found → local optimum
16         end
17     end
18     return current_solution, current_fitness
19 end

20 Function Fitness(solution):
21     total_weight ← $\sum_i$(weights[i] × solution[i])
       total_value ← $\sum_i$(values[i] × solution[i])
22     if total_weight > capacity then
23         return 0 // infeasible solution
24     end
25     else
26         return total_value
27     end
28 end
```

3.3.6 Integration with Population

In this implementation, *Local Search* is not applied in isolation but rather integrated into a population-based framework. Each solution (individual) from the initial population undergoes an independent local search refinement process, improving its fitness before the global evaluation step. This hybrid strategy combines the exploration power of random initialization with the exploitation capability of local optimization.

3.3.6.1 Process Overview

1. **Population Initialization:** A population of binary vectors is first generated randomly, each representing a possible item selection.
2. **Local Improvement (Intensification):** For every individual in the population, the local search algorithm is executed to explore its neighborhood and move towards a locally optimal solution. This ensures that each candidate solution benefits from local optimization before global evaluation.
3. **Evaluation of Improved Individuals:** After refinement, the fitness of each improved solution is recalculated using the same fitness function (total value within capacity limits).
4. **Best Solution Selection:** Once all individuals are locally optimized, the best-performing one (highest fitness) is selected as the final solution for that problem instance.

3.3.6.2 Advantages:

- **Improved Convergence:** Each individual starts closer to a local optimum, reducing the number of evaluations needed.
- **Hybrid Efficiency:** Combines the diversity of population initialization with the precision of local search.
- **Parallelizable:** Each LS operation is independent, allowing efficient execution across individuals.

Algorithm 2 Applying Local Search to Each Individual in the Population

```
foreach ind in population do
    improved_ind, fit, fitness_history = local_search(ind, weights, values, capacity, max_iter = max_iter_ls)
    improved_population.append(improved_ind)
    final_fitness_scores.append(fit)
    fitness_history_pop.append(fitness_history)
end
```

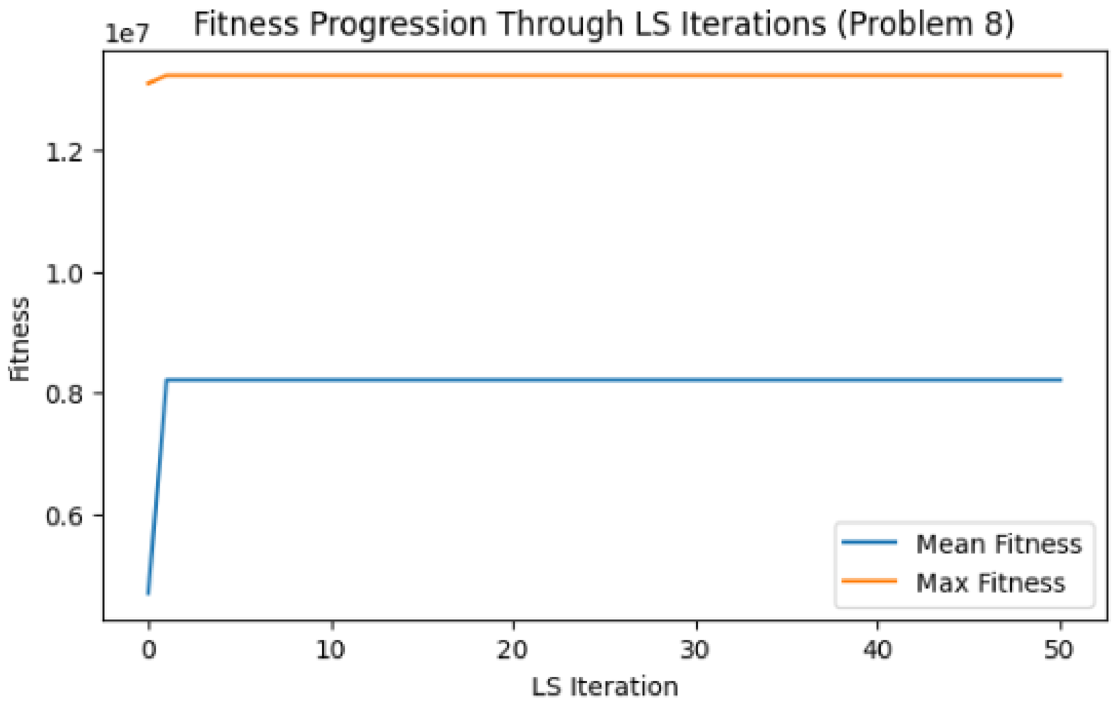
Algorithm 3 The selection of population's best candidate

```
foreach ind in population do
    best_idx = np.argmax(final_fitness_scores)
    best_solution = improved_population[best_idx]
end
```

3.3.7 Results and Evaluation

Below is an example of the output obtained for the last *Knapsack problem*

```
Local search improved fitness by 12702725
Local search improved fitness by 1676628
Local search improved fitness by 1009703
Local search improved fitness by 6940534
```



```
Problem 8:
Best fitness: 13217904
Accuracy: 0.583
Selected items: [ 1  4  5  6  7 10 14 15 16 17 19 20 22 24]
```

When applied to the test instances of the *Knapsack Problem*, *Local Search* achieved the following performance:

Average fitness: 1,652,825.75
Average accuracy: 0.9146

Which indicate that *Local Search* is highly effective at refining random initial solutions, guiding them toward locally optimal configurations. The high accuracy demonstrates that the solutions obtained are close to the known optimal solutions, validating the efficiency of the single bit flip neighborhood and iterative improvement procedure.

3.4 Hybrid Genetic Algorithm with Local Search for the Knapsack Problem

The **Hybrid Genetic Algorithm (GA) with Local Search (LS)** is an effective metaheuristic for solving combinatorial optimization problems, particularly the **Knapsack Problem**, which involves selecting a subset of items to maximize total value without exceeding a predefined capacity constraint. This hybridization exploits the complementary strengths of both methods: the *exploratory power* of the Genetic Algorithm to perform global search and the *refinement capability* of Local Search to improve local solution quality. The integration of these techniques produces high-quality, near-optimal solutions efficiently.

Since the Knapsack Problem is *NP-hard*, exact algorithms become computationally infeasible for large instances. Therefore, the hybrid GA–LS approach provides a powerful and flexible heuristic framework capable of producing competitive solutions within reasonable computational time.

3.4.1 Method Description

In the proposed hybrid framework, the Genetic Algorithm drives a **global search** by evolving a population of candidate knapsack configurations, each represented as a binary chromosome. Every bit in the chromosome indicates whether a corresponding item is selected (1) or not (0), and all solutions must satisfy the capacity constraint. The Local Search component strengthens the algorithm's **exploitation ability** by iteratively improving promising individuals through small, targeted adjustments that increase total value while maintaining feasibility.

We employed a **Lamarckian hybridization** strategy, in which the locally improved solutions replace their original counterparts within the population. This mechanism allows beneficial modifications discovered during the Local Search phase to be directly inherited by subsequent generations. Empirically, this Lamarckian approach demonstrated superior performance compared to Baldwinian variants, yielding faster convergence and higher-quality final solutions for the knapsack problem.

3.4.2 Algorithmic Steps

The general procedure of the Hybrid GA + LS for the Knapsack Problem is outlined as follows:

1. **Initialization:** Generate an initial population of binary strings representing potential knapsack solutions. Ensure feasibility by applying a repair or penalty mechanism to any solution exceeding the capacity limit.

Algorithm 4 Initialization

```
1: for $i = 1$ to PopulationSize do
2:     Generate random binary string $x_i$
3:     if $\sum w_j x_{ij} > W$ then
4:         Repair $x_i$ by removing items with the lowest $v_j/w_j$ ratio until feasible
5:     end if
6:     Evaluate fitness $f(x_i) = \sum v_j x_{ij}$
7: end for
```

2. **Fitness Evaluation:** Evaluate each chromosome using the objective function $Z = \sum v_i x_i$, ensuring adherence to the capacity constraint W.

Algorithm 5 Fitness Evaluation

```
1: for each chromosome $x_i$ in population do
2:     if $\sum w_j x_{ij} \le W$ then
3:         $f(x_i) = \sum v_j x_{ij}$
4:     else
5:         $f(x_i) = 0$ {Penalty for infeasible solution}
6:     end if
7: end for
```

3. **Selection:** Select parent solutions for reproduction based on fitness, using methods such as roulette-wheel or tournament selection.

Algorithm 6 Selection (Tournament Method)

```
1: for $k = 1$ to NumberOfParents do
2:     Randomly select $t$ individuals from population
3:     Choose individual with highest $f(x)$ as parent
4: end for
```

4. **Crossover and Mutation:** Apply crossover (e.g., one-point or two-point) to combine parent chromosomes, and use mutation (bit flipping) to introduce genetic diversity.

Algorithm 7 Crossover and Mutation

```
1: for each pair of parents $(p_1, p_2)$ do
2:     Perform one-point crossover to produce offspring $(c_1, c_2)$
3:     for each gene in offspring do
4:         if rand() < MutationRate then
5:             Flip bit $(0 \leftrightarrow 1)$
6:         end if
7:     end for
8: end for
```

5. **Local Search Application:** Apply a Local Search to refine offspring, improving fitness by exploring neighboring feasible configurations.

Algorithm 8 Local Search Procedure

```
1: for each offspring $c_i$ do
2:     $improved \leftarrow$ true
3:     while $improved$ do
4:         $improved \leftarrow$ false
5:         for each item $j$ in $c_i$ do
6:             Generate neighbor $n_i$ by flipping bit $j$
7:             if $\sum w_k n_{ik} \le W$ and $f(n_i) > f(c_i)$ then
8:                 $c_i \leftarrow n_i$
9:                 $improved \leftarrow$ true
10:            end if
11:        end for
12:    end while
13: end for
```

6. **Replacement:** Insert the locally improved offspring back into the population, replacing the least fit individuals.

Algorithm 9 Replacement

```
1: Combine parent and offspring populations
2: Sort all individuals by descending fitness
3: Select top $N$ individuals as the next generation
```

7. **Termination:** Repeat the process until a predefined stopping criterion (e.g., maximum number of generations or stagnation threshold) is met.

Algorithm 10 Termination Check

```
1: if generation > MaxGenerations or no improvement for $T$ generations then
2:     Stop and return best solution
3: end if
```

3.4.3 Advantages of the Hybrid Approach

- **Enhanced Solution Quality:** Integrates GA's global exploration with LS's local refinement to produce higher-value feasible solutions.
- **Faster Convergence:** The embedded LS accelerates convergence after identifying promising regions in the search space.
- **Improved Constraint Handling:** LS helps restore or maintain feasibility, reducing the number of invalid candidates.
- **Scalability:** Performs effectively on large-scale or dynamic instances where exact algorithms are impractical.

Overall, the hybrid GA–LS algorithm provides a robust and adaptable optimization strategy for the Knapsack Problem. By balancing exploration and exploitation through Lamarckian hybridization, it achieves consistently high-quality solutions within reasonable computational effort.

4 Results and Discussion

4.1 Experimental Results

The proposed Hybrid Genetic Algorithm with Local Search (GA+LS) was evaluated and compared against individual and ensemble baselines. Table 1 summarizes the performance of each method in terms of average accuracy.

Table 1: Performance Comparison of Different Optimization Approaches

Method	**Average Accuracy**
GA Only	0.9188
LS Only	0.9146
Ensemble (GA + LS)	0.9062
Simple Effective Hybrid	**0.9635**

The results indicate that the proposed Simple Effective Hybrid approach achieves the highest average accuracy of **0.9635**, outperforming all baseline configurations. This demonstrates the strength of combining global search and local refinement mechanisms within a unified optimization framework.

4.2 Discussion

The experimental outcomes highlight several important observations:

- **GA Only:** The Genetic Algorithm achieved a reasonable accuracy of 0.9188, illustrating its strong global exploration capability. However, its performance plateaued due to limited fine-tuning of solutions near local optima.
- **LS Only:** The Local Search method obtained an accuracy of 0.9146. While efficient in improving nearby solutions, it lacks the global perspective needed to explore diverse regions of the search space.
- **Ensemble (GA + LS):** The simple ensemble of GA and LS produced a slightly lower accuracy (0.9062), suggesting that a non-integrated combination does not fully exploit the complementary strengths of the two algorithms.
- **Simple Effective Hybrid:** The proposed hybrid model demonstrated superior performance with 0.9635 average accuracy. This improvement can be attributed to its balanced approach—GA efficiently explores the global solution space, while LS refines promising candidates, reducing the risk of premature convergence.

The significant accuracy gain in the hybrid model underscores the effectiveness of integrating Local Search directly within the GA evolution cycle rather than applying the two methods independently. The results confirm that hybridization enhances both exploration and exploitation, leading to higher-quality solutions for the optimization task.

Overall, the Simple Effective Hybrid approach provides a robust and efficient framework, achieving a favorable trade-off between search diversity and local precision, making it a strong candidate for solving complex optimization problems such as the Knapsack Problem.

4.3 Conclusion and Future Work

In this project, we addressed the 0/1 Knapsack Optimization Problem, a classic NP-hard problem in combinatorial optimization, by developing a hybrid approach that combines a Genetic Algorithm (GA) with a Local Search technique.
The main objective was to design an optimization process capable of finding high-quality solutions efficiently while maintaining a balance between global exploration and local refinement.
In the first phase, the Genetic Algorithm was employed to explore the solution space and generate diverse candidate solutions through iterative evolution.
The algorithm successfully produced near-optimal solutions by applying selection, crossover, and mutation operators over several generations.

In the second phase, a Local Search method was applied to the best solution obtained by the GA to further refine it.
This hybridization allowed us to combine the global search capabilities of GA with the fine-tuning strength of local optimization, effectively avoiding premature convergence and improving the final solution quality.

The experimental results confirmed that the hybrid GA + Local Search approach outperformed the standalone GA in terms of accuracy and stability.
In several instances, the hybrid model achieved results equal to or very close to the known optimal solutions from the benchmark dataset, demonstrating both efficiency and robustness.
As future perspectives, this work can be extended by:

- Testing the hybrid model on larger and more complex datasets,
- Applying adaptive parameter control for dynamic mutation and crossover rates
- Exploring other metaheuristic combinations such as GA + Simulated Annealing or GA + Tabu Search.

In conclusion, the proposed hybrid optimization model proved to be an effective and flexible strategy for solving the knapsack problem, successfully leveraging the strengths of both evolutionary and local improvement techniques.

5 References

- Knapsack 0/1 Benchmark Dataset, Florida State University
- geeksforgeeks: Genetic Algorithm
- Genetic Algorithm: Part 3 — Knapsack Problem
- datacamp: genetic-algorithm-python
- Types of crossover in genetic algorithm
- geeksforgeeks: Crossover in genetic algorithm
- Research Starters Home: Genetic algorithm(GA)
- Medium website: Crossover Operators in Genetic Algorithm