AI PROJECT REPORT

Knapsack Problem | Using Genetic Algorithm

What is the Knapsack Problem?

The knapsack problem or rucksack problem is a problem in combinatorial optimization: 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 given limit and the total value is as large as possible.

Types of the Knapsack Problem

There are many types of this problem but our focus is going to be on two of them:

- 0-1 Knapsack Problem
- Unbounded Knapsack Problem

The 0-1 Knapsack Problem

This version of the problem dictates that the items are indivisible, each item from the set of items can only be used once, meaning you can either include an entire item in the knapsack or exclude it entirely.

Unbounded Knapsack Problem

In this version the items are indivisible, meaning that like the 0-1 version, they can't be divided but unlike the 0-1 version any given item can be picked an unlimited number of times.

What is a Genetic Algorithm?

A genetic algorithm is an optimization algorithm inspired by the process of natural selection and genetics. It is commonly used in computer science and engineering to find approximate solutions to optimization and search problems.

The main steps of a Genetic Algorithm:

1. Initialization: Generation of a population of individuals. Each individual has a potential solution to the problem.

2. The fitness of each individual in the population is assessed based on how well it solves the given problem.

3. Selection: Individuals are selected from the current population as parents for the next generation. This selection process is based on the fitness of the individuals - where the fitter individuals are more likely to get selected.

4. Crossover: Pairs of selected individuals exchange information to create new offspring.

5. Mutation: Random changes are introduced in the offspring's genetic information. This helps maintain genetic diversity.

6. Replacement: The new offspring replace some individuals in the current population, and the process repeats for a set number of generations or until a termination criterion is met.

Implementation

The `GeneticKnapsackSolver` class provides an implementation of a genetic algorithm to solve the 0-1 Knapsack Problem and the Unbounded Knapsack Problem.

Attributes

- `items`: A list of tuples representing items, where each tuple contains weight and value.

- `max_weight`: Maximum weight the knapsack can hold.

- `population_size`: Size of the population in each generation.

- `mutation_probability`: Probability of mutation for each gene in the population.

- `generations`: Number of generations for the genetic algorithm.

- `unbounded`: A boolean indicating whether the problem is unbounded or 0-1 knapsack.

Methods

1. `__init__(self, weights, values, max_weight, population_size, mutation_probability, generations, unbounded=False)`: Initializes the GeneticKnapsackSolver object with the provided parameters.

2. `generate_population(self)`: Generates an initial population of chromosomes based on the knapsack type (0-1 or unbounded).

3. `calculate_fitness(self, chromosome)`: Calculates the fitness of a given chromosome based on its total weight and value.

4. `select_chromosomes(self)`: Selects parents from the current population based on their fitness values.

5. `crossover(self, parents)`: Performs crossover between pairs of parents to create offspring.

6. `mutate(self, chromosome)`: Applies mutation to a given chromosome with a specified mutation probability when the chosen type in unbounded.

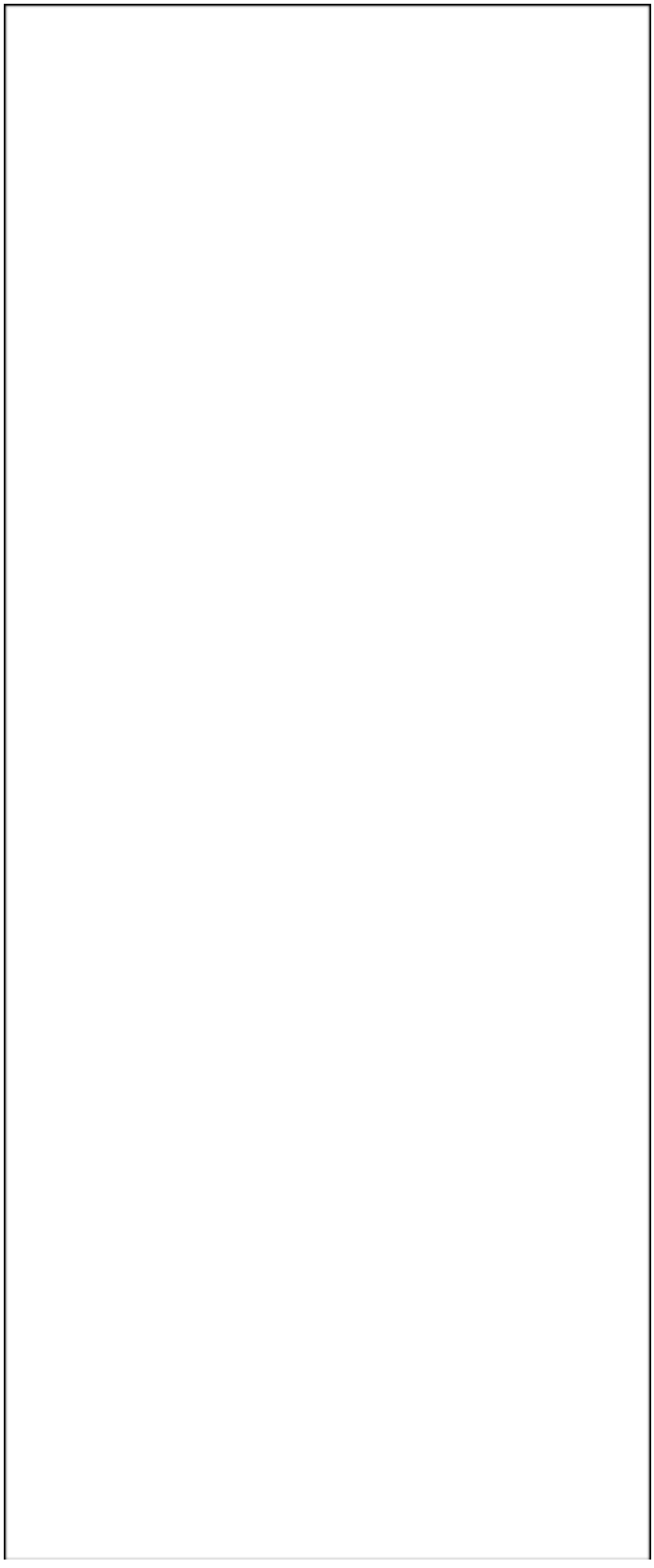
7. `evolve_population(self)`: Evolves the population through selection, crossover, and mutation for a specified number of generations.

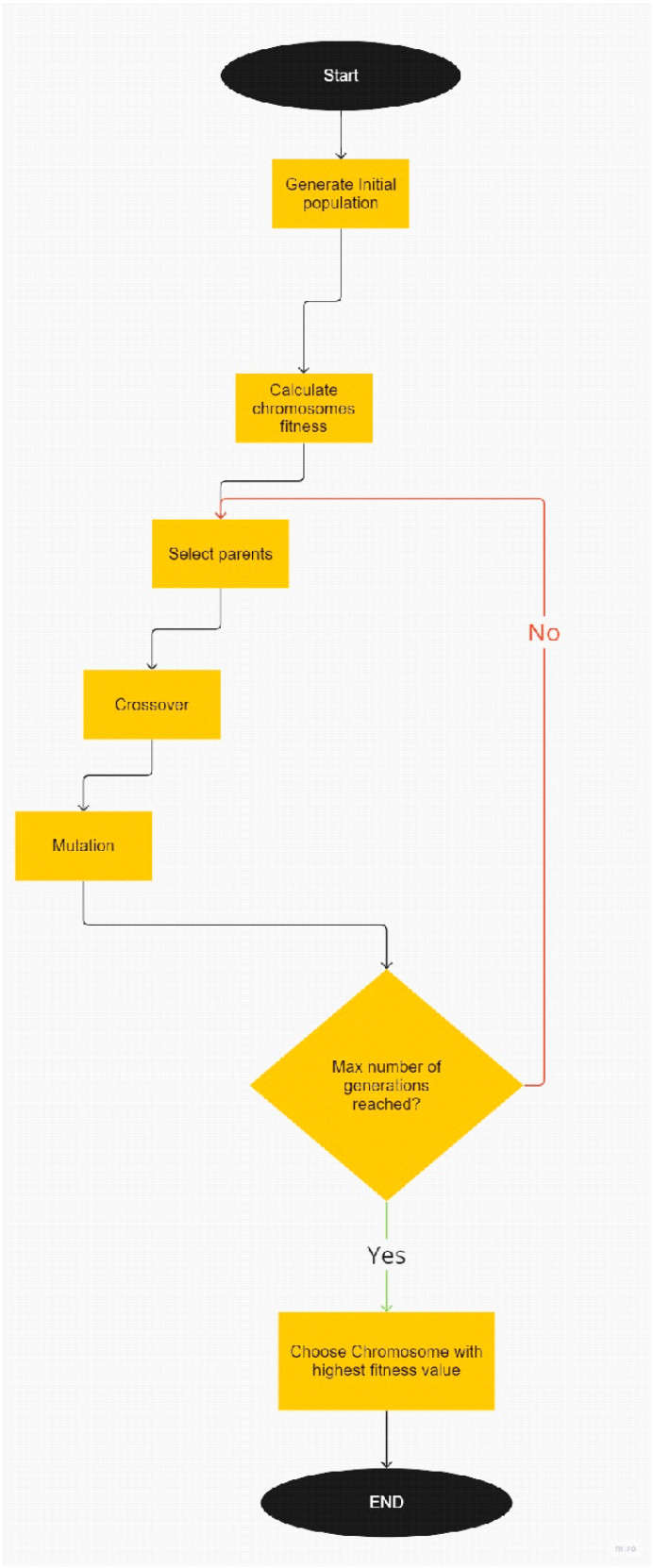
8. `get_best_solution(self)`: Returns the best solution (chromosome) and its corresponding weight and value from the final population.

9. `correct_weight(self, chromosome)`: Corrects the weight of a chromosome if it exceeds the maximum weight.

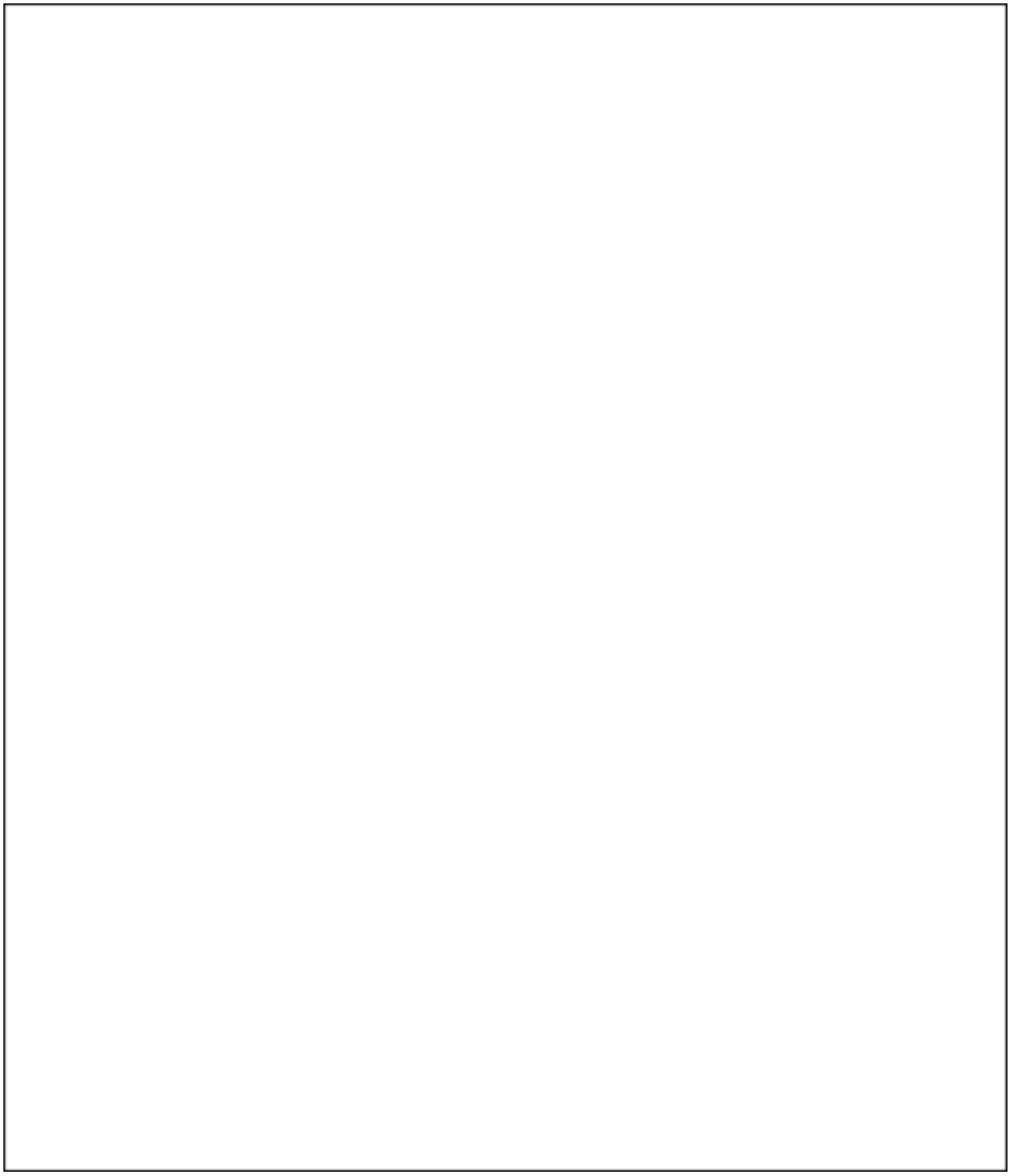
10. `mutation(self, chromo, p)`: Applies mutation to a given chromosome with a specified mutation probability when the chosen type in 0-1.

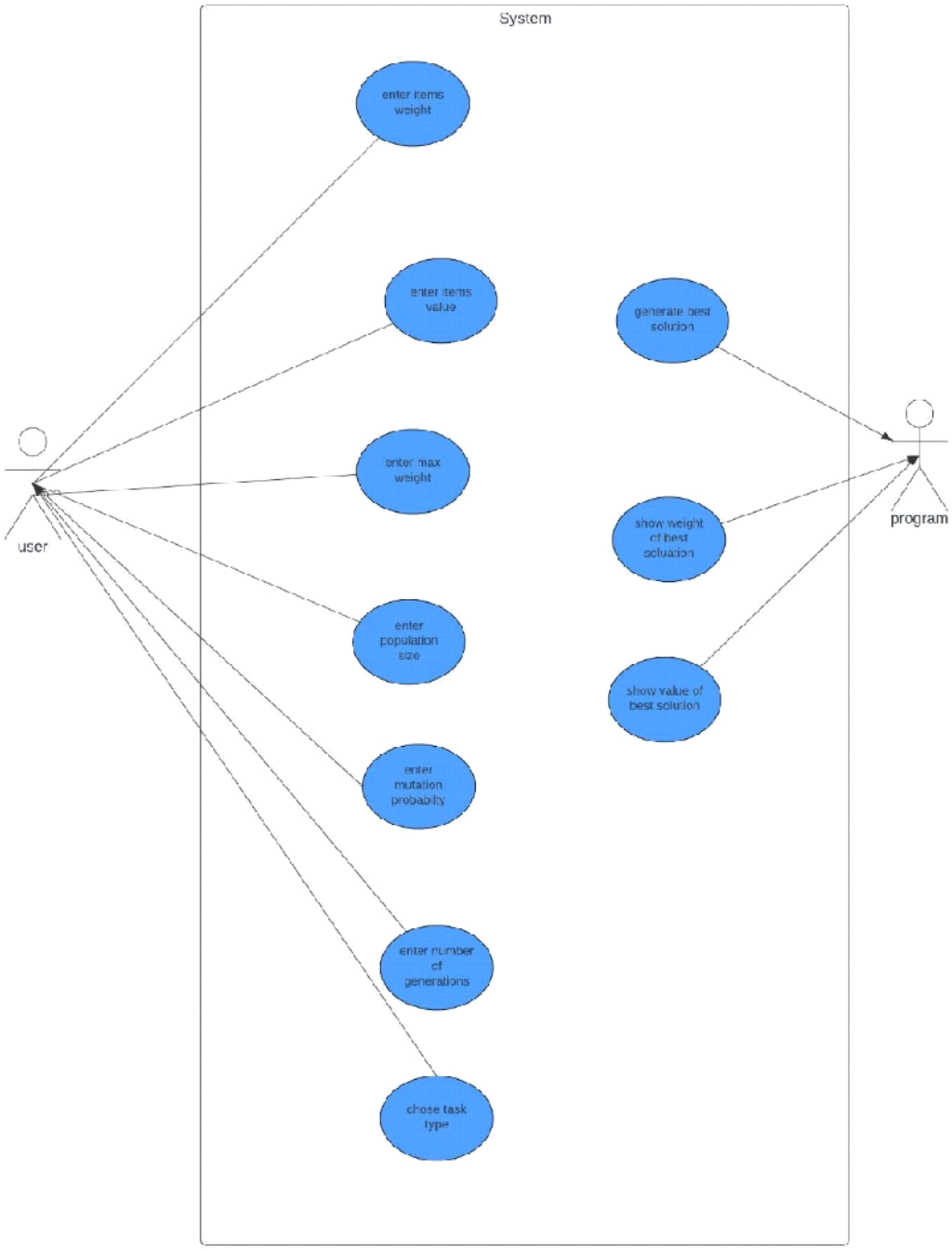
Flowchart




Start
Generate Initial population
Calculate chromosomes fitness
Select parents
Crossover
Mutation
Max number of generations reached?
No
Yes
Choose Chromosome with highest fitness value
END

Use case diagram





Testing

we will start by testing the main functions of the program

1.1(generate_population()) ->

starting with 0,1 it works with different values for populationSize

size =10

[[0, 1, 1, 0, 1],

[1, 1, 1, 0, 0],

[0, 0, 1, 1, 0],

[0, 0, 0, 1, 0],

[1, 0, 0, 0, 0],

[0, 1, 1, 0, 1],

[1, 0, 0, 1, 0],

[1, 0, 1, 1, 0]]

size =8

[0, 0, 1, 0, 1],

[0, 1, 0, 0, 1],

[0, 0, 0, 1, 0],

[1, 1, 0, 0, 1],

[0, 1, 0, 1, 0],

[0, 1, 0, 0, 0],

[0, 1, 0, 0, 0],

[1, 0, 0, 0, 1],

[0, 0, 0, 1, 0]]

size =5

[[1, 0, 0, 0, 0],

[0, 1, 1, 1, 0],

[1, 1, 0, 1, 0],

[1, 0, 0, 0, 1],

[1, 0, 0, 1, 1]]

It work the some in the case of unbounded knapsack

size =10

[[1, 0, 1, 0, 0],

[3, 2, 1, 0, 0],

[0, 4, 0, 0, 0],

[2, 1, 2, 0, 0],

[4, 0, 2, 0, 0],

[0, 0, 0, 1, 0],

[0, 0, 0, 0, 1],

[0, 2, 1, 0, 0],

[4, 3, 0, 0, 0],

[2, 2, 0, 0, 0]]

size =8

[[0, 0, 1, 0, 0],

[0, 1, 0, 2, 0],

[2, 1, 0, 1, 0],

[0, 0, 1, 0, 1],

[1, 1, 1, 1, 0],

[0, 1, 0, 0, 1],

[1, 1, 0, 1, 0],

[0, 1, 0, 1, 0]]

size =5

[[1, 0, 0, 0, 0],

[1, 1, 0, 0, 1],

[2, 0, 0, 1, 0],

[0, 0, 2, 0, 0],

[2, 2, 0, 0, 0]]

1.2 (select_chromosomes()) ->

-in the 0,1 it returns the best chromosome(parents)

[[1, 1, 1, 1, 0], [1, 1, 1, 1, 0]]

- so for the case of unbounded

[[2, 1, 2, 0, 0], [4, 2, 0, 0, 0]]

1.3 (evolve_population)

It's the main part of algorithm, both the crossover and mutation happen in its scope, it takes the population as it input evolve it and then returns at the end.

Example with generations = 3

```
start of cycle ------------------------

pop before : [[4, 0, 0, 0, 0], [2, 4, 0, 0, 0], [0, 0, 0, 1, 1], [3, 2, 1,
0, 0], [0, 0, 0, 2, 0], [1, 1, 1, 0, 0], [1, 0, 1, 0, 1], [1, 1, 2, 0, 0],
[2, 2, 1, 0, 0], [1, 0, 3, 0, 0]]

pop after : [[3, 2, 1, 0, 0], [2, 4, 0, 0, 0], [2, 2, 1, 0, 0], [1, 0, 3,
0, 0], [1, 1, 2, 0, 0], [1, 0, 1, 0, 1], [0, 0, 0, 1, 1], [0, 0, 0, 2, 0],
[3, 2, 1, 0, 0], [2, 4, 0, 0, 0]]

start of cycle ------------------------

pop before : [[3, 2, 1, 0, 0], [2, 4, 0, 0, 0], [2, 2, 1, 0, 0], [1, 0, 3,
0, 0], [1, 1, 2, 0, 0], [1, 0, 1, 0, 1], [0, 0, 0, 1, 1], [0, 0, 0, 2, 0],
[3, 2, 1, 0, 0], [2, 4, 0, 0, 0]]

pop after : [[3, 2, 1, 0, 0], [2, 4, 0, 0, 0], [2, 2, 1, 0, 0], [1, 0, 3,
0, 0], [1, 1, 2, 0, 0], [1, 0, 1, 0, 1], [0, 0, 0, 1, 1], [0, 0, 0, 2, 0],
[3, 2, 1, 0, 0], [2, 4, 0, 0, 0]]

start of cycle ------------------------

pop before : [[3, 2, 1, 0, 0], [2, 4, 0, 0, 0], [2, 2, 1, 0, 0], [1, 0, 3,
0, 0], [1, 1, 2, 0, 0], [1, 0, 1, 0, 1], [0, 0, 0, 1, 1], [0, 0, 0, 2, 0],
[3, 2, 1, 0, 0], [2, 4, 0, 0, 0]]

pop after : [[3, 2, 1, 0, 0], [2, 4, 0, 0, 0], [2, 2, 1, 0, 0], [1, 0, 3,
0, 0], [1, 1, 2, 0, 0], [1, 0, 1, 0, 1], [0, 0, 0, 1, 1], [0, 0, 0, 2, 0],
[0, 4, 0, 0, 0], [2, 2, 1, 0, 0]]
```

1.3.1 example of crossover

```
parents : [[4, 0, 1, 1, 0], [1, 1, 0, 2, 0]]

children :[[4, 0, 1, 0, 0], [1, 1, 0, 1, 0]]
```

1.3.2 example of mutation

```
[0, 2, 1, 0, 0]

[4, 2, 1, 0, 0]
```

2.testing edge cases

Small population=10 & limited generations =3 & low max_wieght =10 & unbounded

```
Items [(1,10),(2,15),(3,18),(4,20),(5,25)]

The output                    max_weight : 10

                      best_chromosome : [4, 2, 0, 0, 0]

                              total_value : 70

                              total weight : 8
```

If we increase the max_weight :

```
                              max_weight : 20

                      best_chromosome : [3, 4, 3, 0, 0]

                              total_value : 144

                              total weight : 20
```

If we increase the number of generation to 10

```
                              max_weight : 20

                      best_chromosome : [2, 4, 3, 0, 0]

                              total_value : 134

                              total weight : 19
```

And then again to 100

```
                              max_weight : 20

                      best_chromosome : [7, 1, 2, 0, 1]

                              total_value : 146

                              total weight : 20
```

Finally we will try 0,1 with the same items and variables except weight at 12

```
                              max_weight : 12
```

```
best_chromosome : [1, 1, 0, 1, 1]
total_value : 70
total weight : 12
```

Papers about Solving the Knapsack Problem

1. "Solving the 0-1 Knapsack Problem with Genetic Algorithms"

Authors: Maya Hristakeva &Dipti Shrestha

http://www.sc.ehu.es/ccwbayes/docencia/kzmm/files/AG-knapsack.pdf

- This paper introduces a genetic algorithm approach for solving the 0/1 knapsack problem. It explores the use of genetic algorithms to efficiently find near-optimal solutions for combinatorial optimization problems.

2. " Comparative analysis of genetic crossover operators in knapsack problem"

Authors: David Oyewola & Gblahan Bolarin

https://www.researchgate.net/publication/310578197_Comparative_analysis_of_genetic_crossover_operators_in_knapsack_problem

- The authors investigate different genetic algorithm representations for solving knapsack problems. The study compares the performance of binary and integer representations in the context of genetic algorithms.

3. "A Hybrid Genetic Algorithm for the Multi-Objective Multiple-Choice Knapsack Problem"

Authors: Farhad Djannaty & Saber Doustdargholi

https://www.researchgate.net/publication/228939007_A_Hybrid_Genetic_Algorithm_for_the_Multidimensional_Knapsack_Problem

- This paper presents a hybrid genetic algorithm for solving the multi-objective multiple-choice knapsack problem. It combines genetic algorithms with other optimization techniques to address a more complex variation of the knapsack problem.

4. "# Solving the 0–1 Knapsack problem using Genetic Algorithm and Rough Set Theory"

_Authors: Tribikram Pradhan, Akash Israni & Manish Sharma

https://www.researchgate.net/publication/301409726_Solving_the_0-1_Knapsack_problem_using_Genetic_Algorithm_and_Rough_Set_Theory

- The paper describes a hybrid algorithm to solve the 0–1 Knapsack Problem using the Genetic Algorithm combined with Rough Set Theory. The genetic algorithm provides a way to solve the knapsack problem in linear time complexity. This paper delves into that approach and explains it in great detail.

5. "Solving the Unbounded Knapsack Problem: An Empirical Study on the Efficiency of Metaheuristic Algorithms"

_Authors: Henrique Becker & Luciana S. Buriol

https://www.researchgate.net/publication/331056089_An_empirical_analysis_of_exact_algorithms_for_the_unbounded_knapsack_problem

- The authors conduct an empirical study comparing the efficiency of various metaheuristic algorithms, including genetic algorithms, for solving the unbounded knapsack problem. The paper provides insights into the performance of different optimization techniques.

Development Platform.

During the making of this project we the following technologies:

- Primary Development Environment: VS code.
- Programming Language: Python.
- Libraries: tkinter & random.
- Version Control Systems: Git & Github.
- Testing Tools: Jupyter Notebook.
- Documentation Tools: Obsidian.
- Collaboration and Communication Tools: Discord.

Similar Applications in the market

- Python's built-in Knapsack library `pip install knapsack` can be used to install the library.
- Google's Knapsack Solver. `from ortools.algorithms.python import knapsack_solver` can be used to use the library.
- Dynamic Knapsack Solver by caydin5 on Github.