Team Members :

Student ID	Student Name in Arabic
20220480	مصطفي طه محمد
20220491	معاذ أحمد خليفه
20220458	مروان اسامة حسنين محمد
20220478	مصطفي درويش مصطفي
20220485	مصطفي محمد مصطفي عبدالعظيم
20220479	مصطفي سامح سمير

Project Github repo : https://github.com/bettercallmarwan/AI-Project

Documentation for Knapsack Problem Solver Using Genetic Algorithm

Introduction and Overview

Project Idea and Overview

This project implements a solution for the Knapsack Problem using Genetic Algorithms (GA). The Knapsack Problem is a classic optimization problem where the goal is to select a subset of items to maximize the total value, while keeping the total weight within a specified limit. This solution uses Genetic Algorithms, a form of evolutionary computation, to find the optimal or near-optimal solutions for both 0/1 Knapsack and Unbounded Knapsack variations. The application has a graphical user interface (GUI) built using Python's Tkinter library, allowing users to input parameters and visualize the results.

Applications and Similar Solutions

This project is similar to various optimization solvers found in industries such as logistics, financial portfolio management, and resource allocation. Many algorithms, such as Dynamic Programming, Greedy, and Genetic Algorithms, are used in these fields to solve variations of the Knapsack Problem. This solution specifically focuses on Genetic Algorithms as a heuristic method for solving complex optimization problems.

Literature Review

1. **Goldberg, D. E. (1989).** *Genetic Algorithms in Search, Optimization, and Machine Learning*. Addison-Wesley.
   This book discusses the basics of Genetic Algorithms and how they can be applied to optimization problems like the Knapsack Problem.
2. **Mitchell, M. (1996).** *An Introduction to Genetic Algorithms*. MIT Press.
   An introductory text on Genetic Algorithms, providing a foundational understanding of how to implement evolutionary search techniques.
3. **Holland, J. H. (1975).** *Adaptation in Natural and Artificial Systems*. University of Michigan Press.
   The foundational text on Genetic Algorithms, detailing their biological inspiration and initial applications in machine learning.
4. **Bäck, T., Fogel, D. B., & Michalewicz, Z. (1997).** *Handbook of Evolutionary Computation*. Oxford University Press.
   This resource provides detailed explanations of various evolutionary algorithms, including Genetic Algorithms.
5. **Whitley, D. (1994).** *A Genetic Algorithm Tutorial*. Statistics and Computing.
   A tutorial on Genetic Algorithms, offering insights into their use in optimization and their mathematical underpinnings.

Proposed Solution & Dataset

Main Functionalities

The Knapsack Problem Solver provides the following functionalities:

- **Input Parameters:** Users can input the number of items, maximum weight, and problem type (0/1 Knapsack or Unbounded Knapsack).
- **Item Generation:** Randomly generated items with weights and values, which the user can modify.
- **Genetic Algorithm Solution:** The solver uses a Genetic Algorithm to find the best subset of items that maximize the total value without exceeding the maximum weight.
- **Visualization:** The best fitness value over multiple generations is plotted to show the algorithm's convergence over time.

Applied Algorithms

Genetic Algorithm

The Genetic Algorithm used in this solution follows these steps:

1. **Initialization:** A population of individuals (potential solutions) is randomly generated. Each individual represents a set of items in the knapsack.
2. **Fitness Calculation:** The fitness of each individual is determined by the total value of selected items, with the constraint that the total weight does not exceed the maximum weight.
3. **Selection:** Parents are selected using tournament selection, where a random sample of individuals competes, and the best individuals are chosen.

4. **Crossover:** Two parents are combined to create two children by exchanging genetic material (chromosomes) at a random crossover point.
5. **Mutation:** A mutation operator is applied with a small probability to randomly alter the chromosomes and introduce variability into the population.
6. **Elitism:** The best individuals from the current generation are preserved to ensure that the solution quality does not deteriorate.
7. **Iteration:** This process is repeated for a specified number of generations to evolve the population towards better solutions.

Problem Variants

1. **0/1 Knapsack:** Each item can either be taken or not taken (binary selection).
2. **Unbounded Knapsack:** Items can be taken multiple times, but the total weight must still be within the specified limit.

Dataset

The dataset used in this project is synthetic, generated on the fly by creating random weights and values for each item in the knapsack. Users can modify the number of items, maximum weight, and values/weights of each item through the GUI.

Applied Algorithms: Genetic Algorithm in Detail

Genetic Algorithm Workflow

1. **Chromosome Representation:**

- **0/1 Knapsack:** A chromosome is represented by a list of 0s and 1s, where each bit indicates whether an item is selected (1) or not selected (0).
- **Unbounded Knapsack:** A chromosome is represented by a list of integers, where each entry indicates the number of times an item is selected.

2. **Fitness Function:** The fitness of a solution is calculated by summing the values of selected items. If the total weight exceeds the maximum allowed weight, the fitness is set to zero (invalid solution).
3. **Selection (Tournament Selection):** Randomly select a subset of individuals, and the two best individuals are chosen as parents for the next generation.
4. **Crossover:** If a randomly generated number is less than 0.8 (crossover rate), a crossover occurs between the two parents at a randomly chosen point, exchanging portions of their chromosomes.
5. **Mutation:** A mutation occurs with a probability of 0.1, where each gene (item) has a chance to change (flip or modify its value). This helps introduce new genetic material and prevents premature convergence.
6. **Elitism:** The top individuals from the population are retained in the next generation, ensuring that the solution quality improves or remains the same.

Parameters

- **Population Size:** Set to 100 individuals.
- **Crossover Rate:** 0.8, meaning there is an 80% chance that two selected parents will undergo crossover.
- **Mutation Rate:** 0.1, meaning each gene has a 10% chance of mutation.
- **Elitism:** Top 2 individuals are preserved in each generation.
- **Generations:** The number of generations is user-defined (e.g., 100).

Experiments & Results

Experiment Setup

- **Inputs:** Number of items (5), maximum weight (100), number of generations (100), and randomly generated weights and values.
- **Algorithm:** Genetic Algorithm with elitism, tournament selection, crossover, and mutation.

Results

- **Output:** The algorithm's progress is shown at each generation, displaying the total value, total weight, and the items selected.
- **Best Fitness Plot:** A plot of the best fitness value over generations shows how the solution improves with each generation.

Sample Output:

mathematica

Copy code

Generation 1:

Total Value: 450

Total Weight: 80/100

Selected Items: Item 1(1 units), Item 3(1 units), Item 5(2 units)

Generation 2:

Total Value: 520

Total Weight: 85/100

Selected Items: Item 1(1 units), Item 2(2 units), Item 5(3 units)

Plot:

A plot showing the best fitness value over 100 generations is generated. It demonstrates the improvement in the quality of the solution over time.

Analysis, Discussion, and Future Work

Results Analysis

- **Convergence:** The algorithm converges to a high-quality solution over multiple generations, with the fitness improving significantly.
- **Efficiency:** The algorithm provides a good solution in a reasonable amount of time, though there are opportunities to further optimize the performance for larger problem sizes.

Advantages:

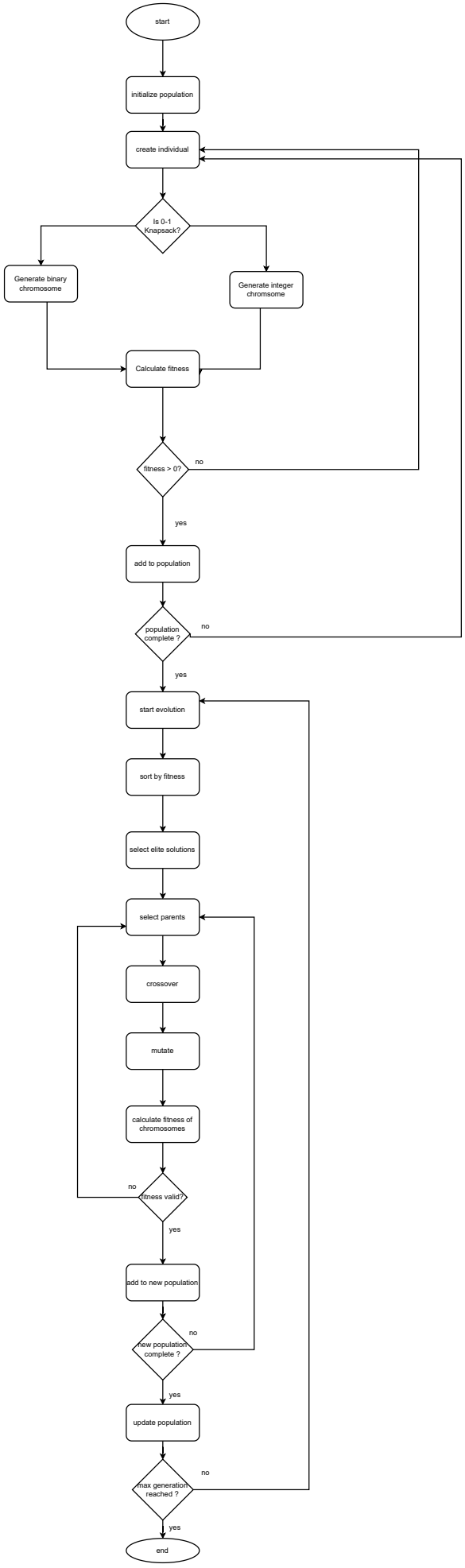
- **Heuristic Approach:** Genetic Algorithms can solve complex optimization problems where traditional methods (like Dynamic Programming) may be inefficient for larger inputs.
- **Flexibility:** The algorithm can handle both 0/1 Knapsack and Unbounded Knapsack problems.

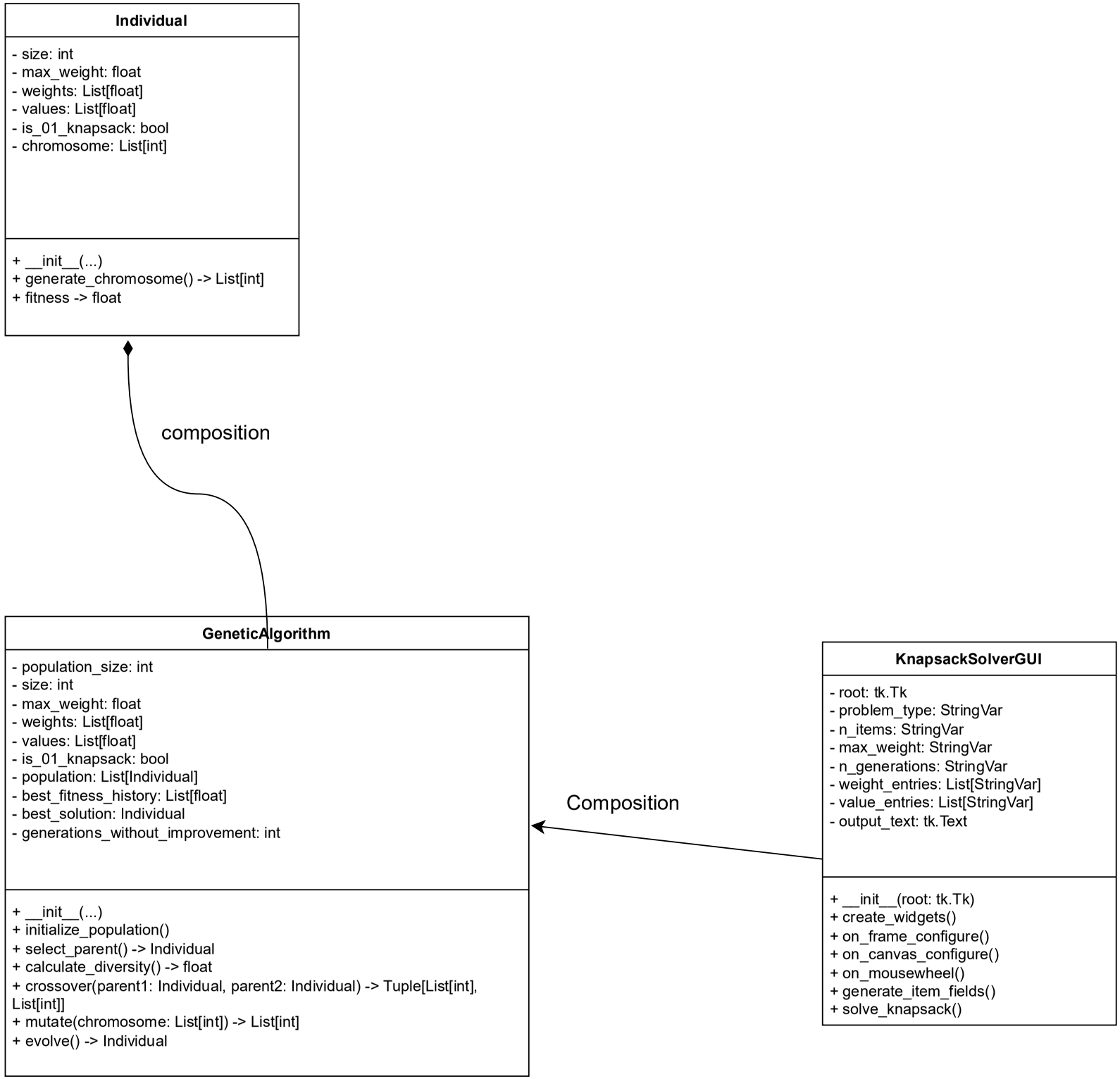
Disadvantages:

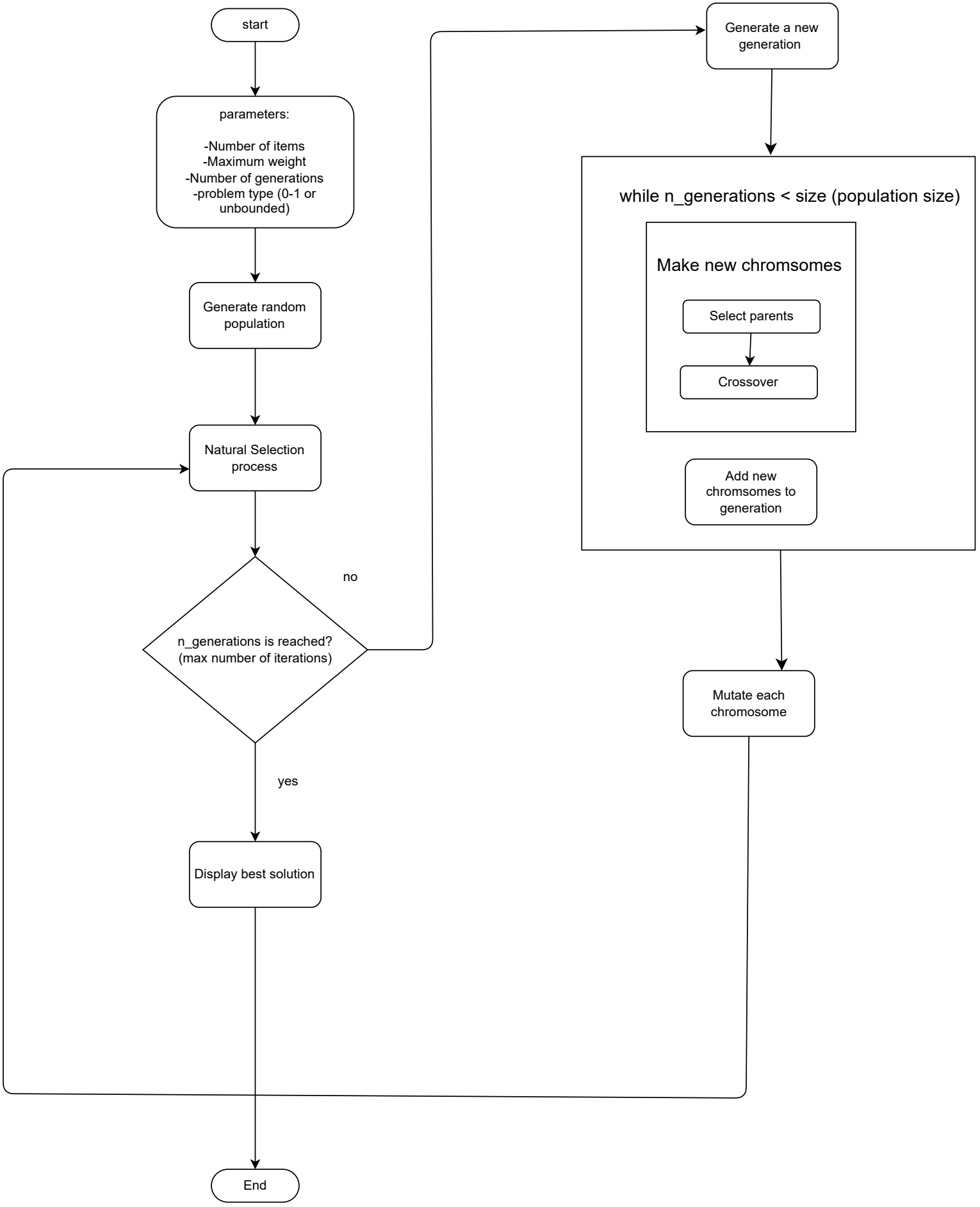
- **Randomness:** The GA relies on random processes, which may lead to suboptimal solutions in some cases.
- **Parameter Sensitivity:** The performance of the GA depends on the choice of parameters like population size, mutation rate, and crossover rate.

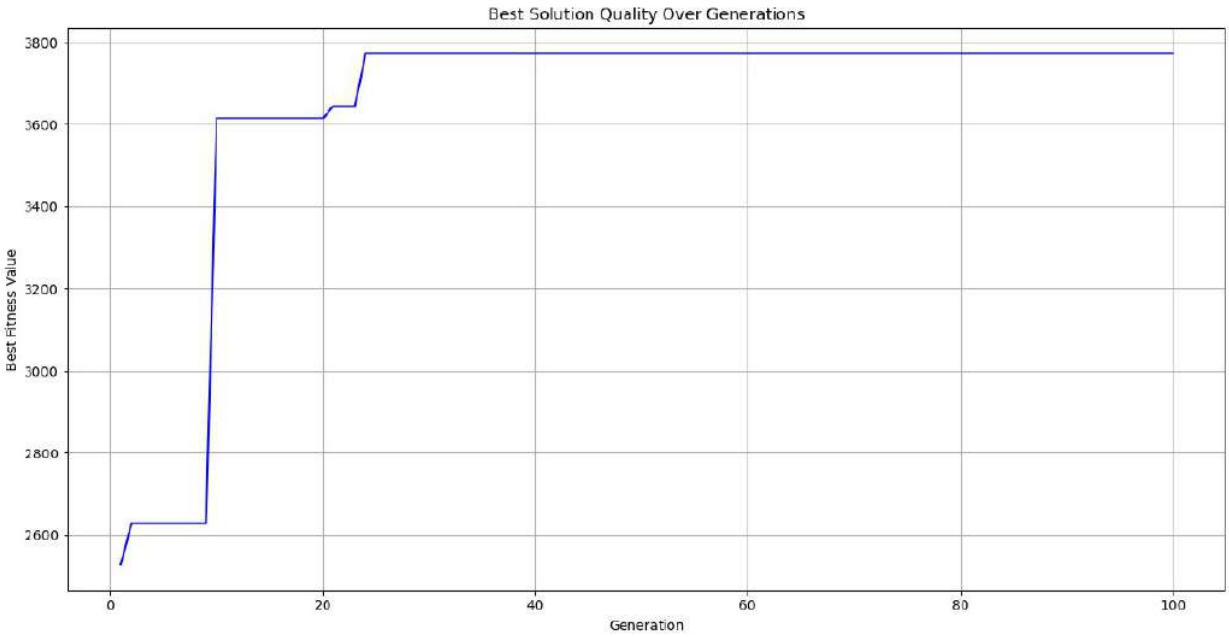
Future Modifications:

- **Parallelization:** To handle larger problem sizes, parallelizing the fitness evaluations or selection process could speed up the algorithm.
- **Hybridization:** Combining the Genetic Algorithm with other optimization techniques (e.g., local search) could improve solution quality.
- **Dynamic Parameters:** Adaptive algorithms that adjust parameters like mutation rate based on the progress of the algorithm could be explored.


start
initialize population
create individual
Is 0-1 Knapsack?
Generate binary chromosome
Generate integer chromsome
Calculate fitness
fitness > 0?
no
yes
add to population
population complete ?
no
yes
start evolution
sort by fitness
select elite solutions
select parents
crossover
mutate
calculate fitness of chromosomes
no
fitness valid?
yes
add to new population
new population complete ?
no
yes
update population
max generation reached ?
no
yes
end


Individual
- size: int
- max_weight: float
- weights: List[float]
- values: List[float]
- is_01_knapsack: bool
- chromosome: List[int]
+ __init__(...)
+ generate_chromosome() -> List[int]
+ fitness -> float
composition
GeneticAlgorithm
- population_size: int
- size: int
- max_weight: float
- weights: List[float]
- values: List[float]
- is_01_knapsack: bool
- population: List[Individual]
- best_fitness_history: List[float]
- best_solution: Individual
- generations_without_improvement: int
+ __init__(...)
+ initialize_population()
+ select_parent() -> Individual
+ calculate_diversity() -> float
+ crossover(parent1: Individual, parent2: Individual) -> Tuple[List[int], List[int]]
+ mutate(chromosome: List[int]) -> List[int]
+ evolve() -> Individual
Composition
KnapsackSolverGUI
- root: tk.Tk
- problem_type: StringVar
- n_items: StringVar
- max_weight: StringVar
- n_generations: StringVar
- weight_entries: List[StringVar]
- value_entries: List[StringVar]
- output_text: tk.Text
+ __init__(root: tk.Tk)
+ create_widgets()
+ on_frame_configure()
+ on_canvas_configure()
+ on_mousewheel()
+ generate_item_fields()
+ solve_knapsack()


start
parameters:
-Number of items
-Maximum weight
-Number of generations
-problem type (0-1 or unbounded)
Generate random population
Natural Selection process
n_generations is reached? (max number of iterations)
no
yes
Display best solution
End
Generate a new generation
while n_generations < size (population size)
Make new chromsomes
Select parents
Crossover
Add new chromsomes to generation
Mutate each chromosome


Best Solution Quality Over Generations
Best Fitness Value
Generation
3800
3600
3400
3200
3000
2800
2600
0
20
40
60
80
100